Fall 2022 / EC311
Lab 1: Design 4-bit binary Adder-Subtractor and ALU
Due Date: Due by October 14, 2022

1. Goals

- Learn Verilog modular code design.
- Learn structural and behavioral Verilog.
- Design simple combinational circuits.

2. Overview

In this lab, you will use structural and behavioral Verilog to design a 4-bit binary adder-subtractor and a simple 4-bit Arithmetic-Logic Unit (ALU) that will be able to perform addition, multiplication, concatenation, and shift functions for both signed and unsigned numbers.

3. Part 1: 4-bit binary adder-subtractor (30 points)

Write the gate-level (structural) hierarchical description for a 4-bit binary adder-subtractor using Verilog.

3.1. I/O Specifications

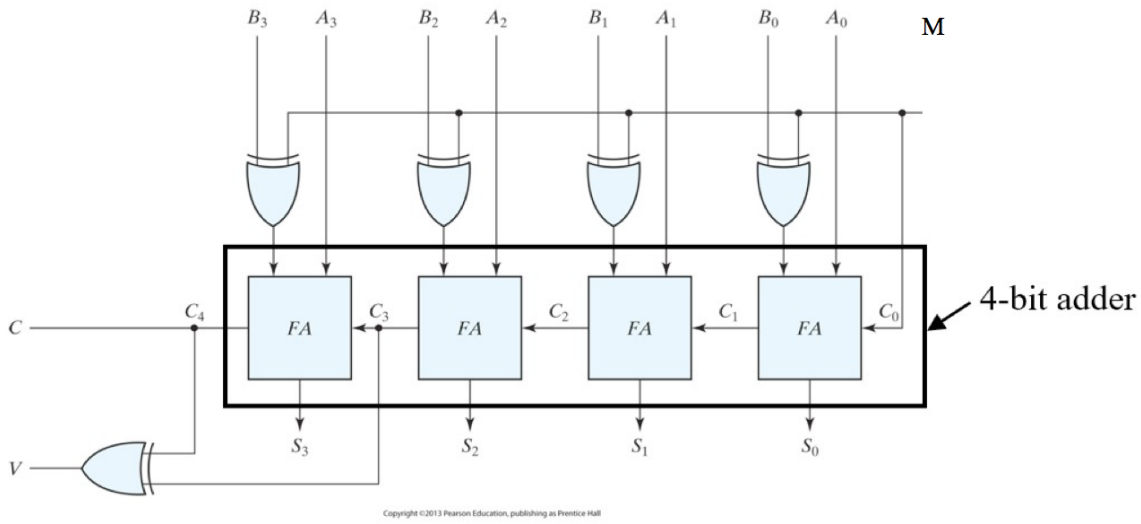
The adder-subtractor logic block must have two 4-bit inputs A ($A_3A_2A_1A_0$) and B ($B_3B_2B_1B_0$) and one 1-bit input M as control bit. The block must output a 4-bit result S ($S_3S_2S_1S_0$) representing the result of the selected operation. The circuit must also output a 1-bit carry signal C and a 1-bit signal V for overflow. When M is 0, S is result of A plus B, and when M is 1, S is the result of A minus B. If the numbers are unsigned then the C bit detects a carry after addition or borrow after subtraction. If numbers are signed then the V bit indicates an overflow. Overflow bit V indicates whether 4-bit signed radix is enough to represent the correct answer S. V=1 means 4-bit signed radix does not have enough bits to accurately represent S. V=0 means 4 bits are enough, and that S correctly represents output of the operation.

3.2. Methodology

Follow the steps listed below to design your 4-bit binary adder-subtractor.

1. Design a 1-bit half adder circuit using structural Verilog.
2. Design a 1-bit full adder by instantiating the half adder one or more times.
3. Design a 4-bit adder by instantiating the 1-bit full adder one or more times.
4. Design a 4-bit adder-subtractor using the figure below (taken from page 166 of the Mano, Ciletti textbook, 6th Ed.). Note: The XOR primitive in the figure acts as a controlled complementor for input B. The output of XOR is one of the input bits to the adder.

For each of the tasks listed above test your design with a test bench to ensure it is correct.



3.3. Testing Tips

To test your adder-subtractor design, you will need to write a testbench in Verilog. The testbench will contain the inputs that you want to feed to your design for testing. One way to assign test stimulus is using the following serial assignment statement for the test cases:

```
initial begin
        A = 0; B = 0;       // set AB = 00 at time 0
        #10 A = 0; B = 1;   // set AB = 01 at time 10
        #10 A = 1; B = 0;   // set AB = 10 at time 20
                            // (10 time units after previous time stamp)
        #10 A = 1; B = 1;   // set AB = 11 at time 30 (10 after 20)
end
```

There is another way to assign stimulus in parallel by using 'fork – join'. The code for previous stimulus example using fork – join is below. Here all of the #(delays) are referenced to time=0.

```
initial fork
        A = 0; B = 0;       // set AB = 00 at time 0
        #10 A = 0; B = 1;   // set AB = 01 at time 10
        #20 A = 1; B = 0;   // set AB = 10 at time 20
        #30 A = 1; B = 1;   // set AB = 11 at time 30
join
```

Given that in our adder-subtractor design there are 9 bits of input in total, the total number of test cases will be $2^9 = 512$. It is unreasonable to manually assign all these stimulus. You can use the code example shown below to formulate your test bench to test the 4-bit adder-subtractor design for all possible input combinations.

```
initial begin
        A = 0; B = 0; // assign initial values
end
always begin
        #10    {A,B} = {A,B} + 1'b1;
end
```

Here, {} means concatenation. You are adding 1 to {A,B} every 10 ns. When all bits become 1's, the value will overflow and start over at all 0's again. This will continue forever, unless you specify an ending time using $finish or $stop. The code below shows an example of using $stop. You will need to figure out how to use $stop for the test bench shown above.

```
initial begin
        A = 0; B = 0;
        #10 A = 0; B = 1;
        #10 A = 1; B = 0;
        #10 A = 1; B = 1;
        #10 $stop;
end
```

You can follow the directions in Lab A and the Vivado Tutorial to generate waveforms for observing the outputs of your simulations. To make verification easier, you can change the settings in the simulation tool to display the multi-bit binary signals as signed decimal numbers. To change the number system for a displayed signal:

1. Select the signal and right click on it.
2. Select Radix $\rightarrow$ Signed Decimal

4. Part 2: 4-bit Arithmetic-Logic Unit (ALU) (50 points)

Design a simple 4-bit ALU whose design is described below.

4.1. ALU I/O

The ALU must have two unsigned 4-bit inputs named 'A' and 'B' as well as a 2-bit operation code named 'S'. The ALU must output the result with an 8-bit output named 'Y'.

4.2. ALU Components

The ALU must have the following components[1]:

1. **Adder** - Design a 4-bit adder module with behavioral Verilog. You cannot reuse the adder from part 1. The 4-bit binary adder module must have two 4-bit inputs and one 8-bit output. You may borrow or reuse code from the part 1 test bench.
2. **Multiplier** - Design a 4-by-4 multiplication module, with two 4-bit inputs and one 8-bit output. This module should perform unsigned multiplication. You must use behavioral Verilog for designing this module.

[1]Each component should have a name that succinctly describes the functionality of the component, and not the lab or problem number. Test each component thoroughly using simulations to make be sure it works

3. **Concatenator** - Design a concatenator module, which concatenates two 4-bit inputs named 'A' and 'B' to generate one 8-bit output named 'AB'. Simulate the concatenation module to be sure it works.

4. **Shifter** - Design a Left Shift module with 4-bit inputs A and B. The module should left shift and extend the input A to 8-bits. The number of times left shift operation should be performed is specified by B bits. Note that if B is larger than 7, all of the bits in A will be shifted out, producing a result of 0x00.

5. **Multiplexer** - A multiplexer is a device that has N control inputs and 2^N data inputs. In Figure 1, the multiplexer is referred to as Mux, which has four 8-bit inputs (J, K, L, M), and one 2-bit control signal (S). Whenever S = 2'b00, the output Y = J; if S = 2'b01, then Y = K and so on.

   You will need to design such a multiplexer for the ALU. The easiest way to design a multiplexer in Verilog is to use a case statement. Do not forget that the case statement in Verilog has to be included in a procedural block (usually an always statement).

Combine all the modules you have created into a single top level module named 'alu'. Your top level module should instantiate five modules: Multiplier, Binary Adder, Concatenator, Left Shift, and Multiplexer. It should have two 4-bit inputs, A and B, a 2-bit control S, and an 8-bit output Y.
The 2-bit operation code control input is specified as follows:
00 - Concatenation
01 - Binary addition
10 - Left Shift
11 - Multiplication

5. Deliverables

You will need to get both part 1 and part 2 of your lab signed off by the instructor or the TA. You need to show your design hierarchy (under Project Manager → Sources → Design Sources), your Verilog code, and top module simulation of the half adder, full adder, 4-bit adder-subtractor and ALU to TA. **You don't have to implement your code on the FPGA board for this lab.**

Your design has to abide by the following requirements. Failure to follow the instructions will result in a grade reduction.

1. Use multi-bit vector/bus/array (just different ways of naming) instead of 1-bit signals for A, B, and S.

2. Follow the input/output specifications.

3. Be hierarchical (consisting of sub modules), and use separate file for each module.

4. Test each module separately and thoroughly. This means you need to cover all the possible combinations of input values to prove your logic is right.

5. Test submodules first. For example, demonstrate that the half adder and the full adder designs work as standalone units. Your 4-bit adder will fail if the submodules don't work. It is generally good engineering practice to test submodules before going to the next level. *You must turn in a test bench for each module.*

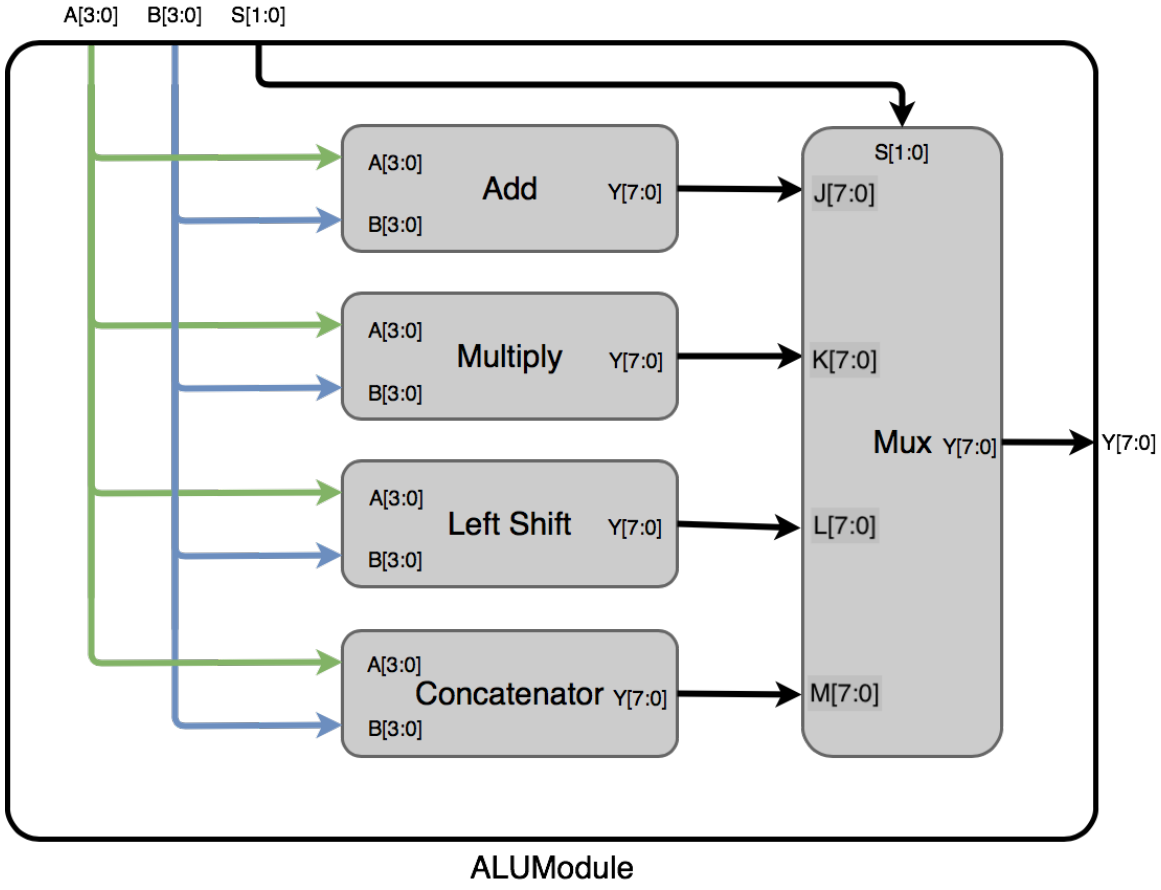


Figure 1: ALU block diagram.

For part 1 of the lab, the TA will test your code for 5 different test cases. Each test case is worth 6 points. For part 2 of the lab, the TA will test your code for 5 different test cases. Each test case is worth 10 points. To pass all the test cases in part 1 and part 2, make sure to think of all the corner cases while implementing your design.

You must turn in a lab report (in PDF form via BlackBoard) describing what you did in this lab, how you did it and what happened when you did it. The lab report should be broken up into the following sections: Objective, Methodology, Observation (can include screenshots timing diagrams, schematics, etc.) and Conclusion. Your report should contain enough detail such that any other engineer can replicate your experiments after reading your lab report. Please include all the required details in your lab report. Your lab report should not be more than 5 pages long. Use 1 inch margin from all four sides, and 12-point font. Your lab report is worth **20 points**.

You have to submit your Verilog code via GitHub classroom. Detailed instructions for submitting the Verilog code will be provided at a later date.