EC311 Introduction to Logic Design
Spring 2022
Lab 1: 4-bit Binary Adders-Subtractors

Goals

- Introduction to Verilog modular code design
- Introduction to structural and behavioral Verilog
- Designing simple combinational circuits

Overview

In this lab you will use both *structural* and *behavioral* Verilog to design a 4-bit binary adder-subtractor. Please refer to the textbook for more information. You could review Figure 4.13 and Problem 4.13, Problem 4.37 in your textbook, and read supplement textbook material in sections Mano 4.5-4-6.

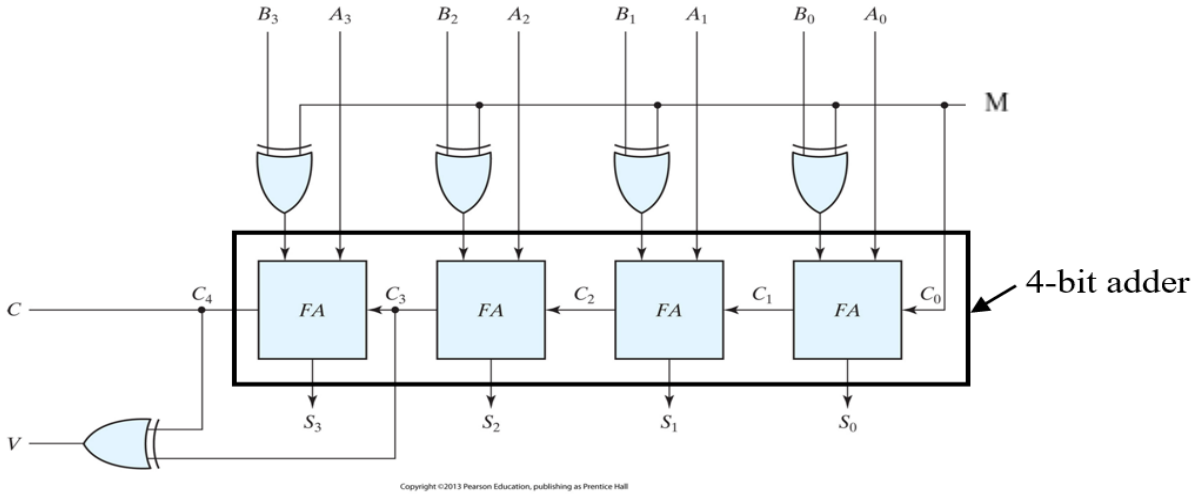
Tasks

1. Structural Verilog

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

IO specifications: You have 4-bit inputs `A` and `B` (signed radix), and 1-bit input `M` as control bit. You need to output 4-bit `S` as results, 1-bit `C` for carry (or borrow) out, 1-bit `V` for overflow. When `M` is 0, `S` is the result of `A` plus `B`; when `M` is 1, `S` is the result of `A` minus `B`. Overflow bit `V` indicates whether 4 bit signed radix is enough to represent the correct answer `S`. `V=1` means a 4-bit signed radix has not enough bits to represent `S` and more bits are needed. `V=0` means 4-bits is enough and the output is correct in 4-bit signed radix.

1. Design a 1-bit `half_adder` circuit in structural Verilog.
2. Design a 1-bit `full_adder` by instantiating the `half_adder`.
3. Make a 4-bit adder by instantiating the 1-bit `full_adder`.
4. Make a 4-bit adder-subtractor following the figure below (taken from page 142 of the Mano, Ciletti textbook). Note that here vector `B` will need to have a controlled complementor proceeding its input to the adder bits (XOR primitive).



Testing Tips:
In previous schematic labs, you have practiced serial assignment of test cases (or stimulus), such as below.

```
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 after the
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 equivalent code as the previous example 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 there are 9 bits of input in total, so the number of test cases will be $2^9 = 512$. It's unreasonable to manually assign all these stimuli one by one as above. You can follow the code example (a modification is required to include M) below for your test.

```
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 bit (1') binary number (b) '1' every 10 ns. When all bits become 1's, it will start over at all 0's again. This will always go on, unless you specify an ending time using `$finish`.

Another tip:
After you generate the waveforms, you can take advantage of the simulation tool to convert the multi-bit binary signals to signed decimal. This would make your verification easier.

Select signals → right click → Radix → Signed Decimal

2. Behavioral Verilog

This part is intended to show you a behavioral approach to the structural adder/subtractor you designed in Task 1. Similar to the previous approach, depending on `M`, take the 2's complement of `B` (behaviorally) and add it with `A` or just add `A+B` if subtraction is not dictated by `M`.

Note that IO specifications are different from Task 1. Instead of using `V`, another bit is added to the sum (`S`).

IO specifications: You have 4-bit inputs `A` and `B` (assume signed radix), and 1-bit input `M` as the control bit. You need to output a **5-bit** `S` as the result. When `M` is 0, `S` is the result of `A+B`; when `M` is 1, `S` is the result of `A-B`. A 5-bit output will always be enough to represent the sum or difference of two 4-bit signed numbers. In reality, the `carry_out_bit` in Task 1 is the 5th bit, and `V` decides whether the 4-bit answer is correct, which makes no difference here, because of moving up to 5-bits for the Sum (`S`).

We strongly suggest you use the conditional operator (`?:`), or if /else statements combined with the arithmetic method for summing two vectors, i.e. `S = A+B;`. In other words, use the `M` bit to steer yourself around. In the case of subtraction, take the 2's complement of `B`, and add it to `A`, for a solution that is parallel to the approach used in Task 1 of this lab.

Hints: If you have trouble to come up with a solution directly, do some addition examples on paper, and think about these questions:

1. Assume `A` and `B` are in 4-bit signed radix. Let the sum be a 5-bit signed radix. Go through some cases covered in Task 1 (structural).
2. Examine specifically the cases in Task 1 where you have a `carry_out` and an `overflow` bit that indicates there is an error in **four** bits.
3. Is the answer correct in **5-bit** signed radix? You can use these 5 cases for your own testing: `5+5, -6-(+1), 2+(-3), 0-2, -5+(-5).`

Then summarize your observations in general. Make sure you covered these cases:

3. `A` and `B` are both 4-bit and positive, what is the MSB of 5-bit output `S`?
4. `A` and `B` are both 4-bit and negative, what is the MSB of 5-bit output `S`?
5. `A` and `B` have different signs in 4-bits, (one positive and the other negative). Is the answer valid in 5-bits signed radix? What is the MSB of 5-bit output `S`?

3. FPGA

1. Use User Constraint File (UCF) to connect your inputs to switches on the board, and your outputs to LEDs on the board. You should use the 4 switches for each 4-bit input (A and B) and one switch for input M. For output (S), you should use the 5 rightmost LEDs. For both input and output, the rightmost switch or LED should show the least significant bit.
2. Synthesize your design and generate bitstream.
3. You can use the instructions in the Vivado tutorial for programming the FPGA.
4. Test your circuit to determine if it works correctly for different cases.

Deliverables

1. You need to show your design hierarchy (under design – implementation window), your Verilog code, and top module simulation of both parts to TA**.** Your design has to abide by the following requirements. Failure to follow the instructions will result in a grade reduction.

    a. For each testbench, identify one test case where **overflow** happens.

b. For each testbench, after creating the overflow test case in (a), create a complete test suite that **covers all the possible combinations** of input values to prove your logic is right.
c. Be hierarchical (consisting of other sub modules), use separate files for each module. (only for Task 1)
d. Write a testbench **for each module**. Demonstrate that both the half adder, the full adder work. Your 4-bit adder will fail if the submodules do not work properly. It is a good engineering practice to test submodules before going to the next level.
e. Upload your verilog codes to Blackboard in a compressed file. Note that you should upload only the verilog codes (*.v), not the whole project.

2. Demonstrate your adder/subtractor to TA on the FPGA board. Make sure that your design works correctly for all different scenarios.