EE312 Lab #1 Report

20180610 정의준

1. Introduction

To make an ALU that is required for Lab 1, we have to design a Verilog module that reads the data from the wires, calculates the result of an operation, and saves it to the output register. The ALU should work for 16 kinds of operations, i.e., addition, subtraction, and so on. To implement such an ALU, we can use the '`case (operation)`' statement to execute the operation that we want to perform.

2. Design

Lab 1's module design is very simple; the module only consists of a single ALU module. For the input, the operands A, B and the operation OP is given. For the output, the operation result is saved to the register C and the overflow flag is saved to Cout.

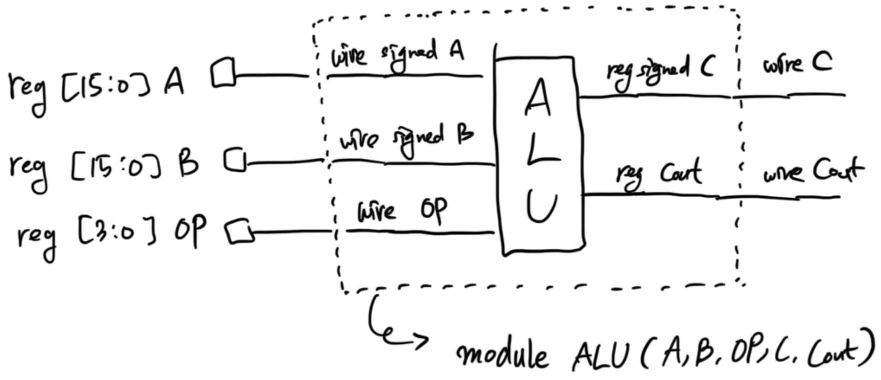


Figure 1 The overall scheme of the module

When we run the testbench code, ALU_TB module loads the ALU module and connects the input and output wires with the registers. Then, the ALU_TB module puts the values to the registers and obtains the operation results from the output register.

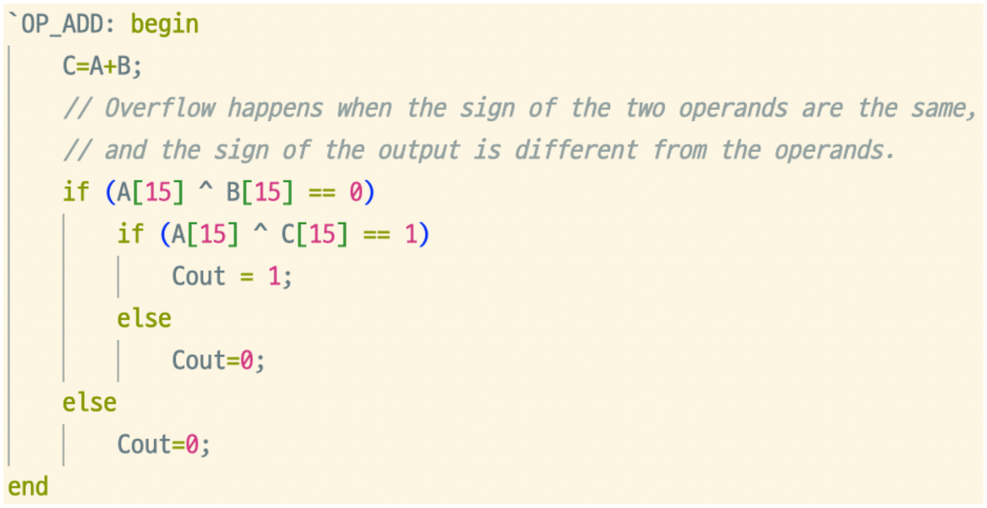
3. Implementation

First, we declare the wires A, B, OP. Since we are dealing with signed 16-bit integers, we use the ‘wire signed’ statement. This will help us to easily calculate arithmetic shift operations with a single line of code. Similarly, we declare the registers C and Cout, which will be connected to the wires of the testbench module.

With the statement `always @(*)` and `case (OP),` the module will run the operation depending on the value of OP, if the data from the input wires are changed. The rest of the code goes as enumerating the pseudo-code below for 16 times (for 16 distinct operations). The code below is an example for the operation No. 1.

```
Operation_1: begin
        C = A Operation_1 B;
        Cout = overflow? 1 : 0;
end
```

Implementing Cout for addition and subtraction was not obvious. By recalling some aspects of the 2’s complement bit addition and subtractions, I was able to implement the code calculating Cout with several if statements. The key of calculating Cout is to compare the first digit of the operands and the sum. The details are written in the comments of the code below.



```
`OP_ADD: begin
    C=A+B;
    // Overflow happens when the sign of the two operands are the same,
    // and the sign of the output is different from the operands.
    if (A[15] ^ B[15] == 0)
        if (A[15] ^ C[15] == 1)
            Cout = 1;
        else
            Cout=0;
    else
        Cout=0;
end
```

For the other operations, most of them were already natively implemented in the Verilog and I did not have hard time using them. Cout is fixed to 0, since there were no overflow cases.

4. Evaluation

By running and debugging the code, I was able to know that my code is written well, and it also functions as I have expected. By running the testbench code, my code has passed 50/50 tests of ALU_TB.v.

5. Discussion

Implementing the ALU itself was not that hard. One key idea that I didn't know was that we were able to manually set the register's configuration whether it is signed or unsigned. By setting this, I was able to implement the shift operations much easily.

I had a hard time setting up the Modelsim environment. The program on my Windows desktop did not work properly; the transcript output did not show the 'blue texts'. I checked that the Modelsim program displays the 'blue texts'.when I used the virtualbox program on my Windows desktop, but it was too slow. But, until I find out the solution to this problem, I think I will have to use Virtualbox to run the code before the submission.

When I started the assignment at the first time, I used my Macbook and the 'icarus verilog' program to write and debug the code. However, I heard that some complicated Verilog codes may have different outputs when I compile it with Modelsim and iVerilog. So, I decided to code almost everything on my mac and use the Windows desktop + Virtualbox to do a final check before the submission.

6. Conclusion

From Lab 1 assignment, I was able to understand how the module is implemented with the Verilog language, and how the wires and registers are connected. Also, some low-level properties of the Verilog language were quite different from my intuition that I've learned through high-level languages, such as C++ and python. Finally, I was able to use and learn the basic idea of module programming and the basic grammars of the Verilog language.