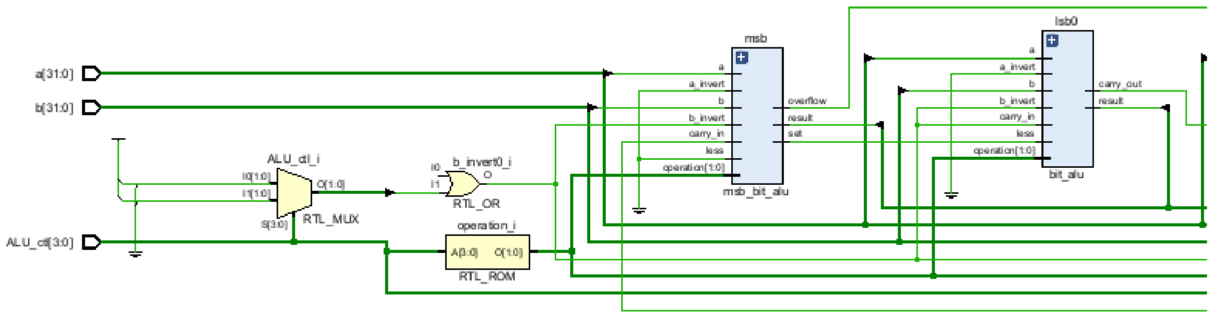
CO_Lab1_111550076

notion link:

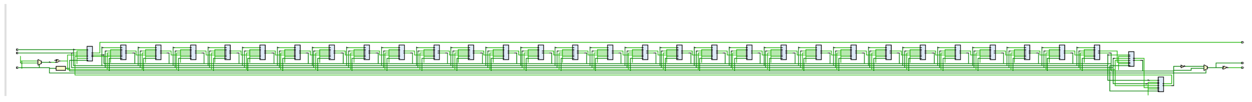
CO_Lab1_111550076

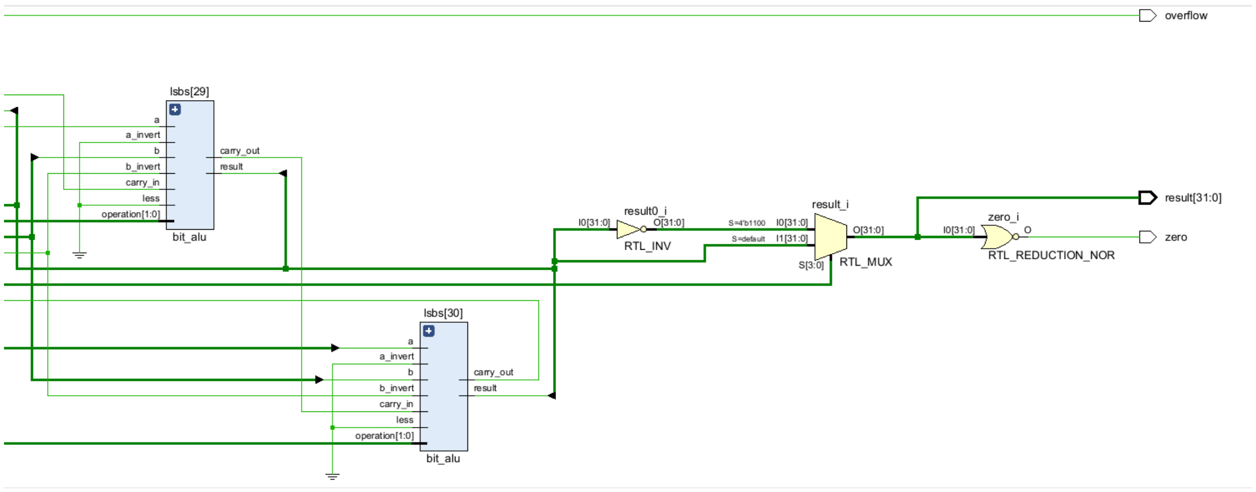
1. Architecture Diagrams



In the design, the MSB ALU should be positioned last, but Vivado displays it at the beginning.(possibly due to the 'set' signal being directed to the 'less' input of the first 1-bit ALU.)

I modified 'operation' to [1:0] so that each ALU receives the same operation, regardless of what the 'ALU_ctl' is.





Part of the schematic, the connections follow the method described in the textbook.

2. Answer the following Questions

1. How `overflow` is calculated?

```
//in msb_bit_alu.v
//overflow
    assign overflow = carry_in ^ carry_out;
    //carryin of msb != carryout msb
```

When there is an overflow, the highest bit of the calculation will be incorrect: what was originally positive becomes negative, and vice versa. Therefore, I check whether the carry-in and carry-out of the last MSB ALU for addition or subtraction are the same; if they are not, it indicates an overflow.

2. Explain why ALU control signal of SUB is `0110` and NOR is `1100` ?

In the case of Sub=0110, the initial "01" denotes a_invert and b_invert respectively. When the first digit is 0, it indicates that "a" remains unchanged, while the second digit being 1 means that "b" transforms into ~b. Subsequently, the carry_in is added. The latter "10" signifies the operation of addition, implying the execution of a+(-b).

As for Nor=1100, the preceding "11" similarly signifies a_invert and b_invert. "a" is transformed into ~a, and "b" into ~b. Due to the nature of the NOR operation, which involves performing an OR operation followed by a NOT operation or vice versa, the trailing "00" denotes the AND operation.

3. If you assign different signal to these operation, what problems you may encountered?

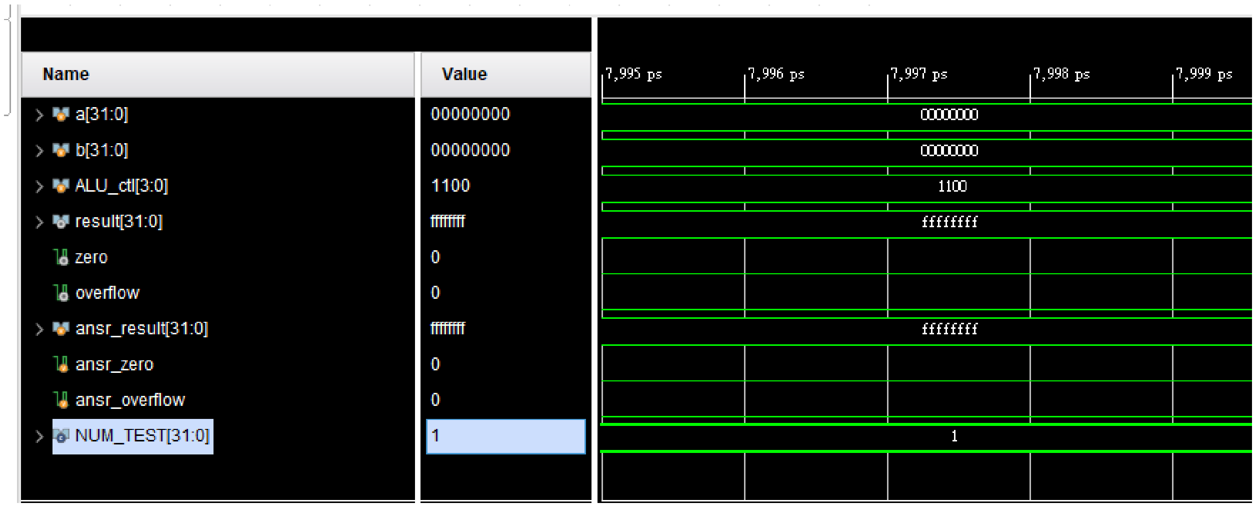
Assigning different signals to the operations of subtraction and NOR in an ALU would lead to confusion in interpretation, compatibility issues between processors, increased programming complexity, challenges in testing and debugging, and additional documentation and education requirements. Standardized signal assignments ensure consistency, compatibility, and ease of programming across different systems and architectures. Deviating from these standards can result in various operational and compatibility problems.

4. True or false: Because the register file is both read and written on the same clock cycle, any MIPS datapath using edge-triggered writes must have more than one copy of the register file. Explain your answer

`False`: In MIPS architecture, it is not necessary to have more than one copy of the register file despite it being read and written in the same clock cycle. This is because read operations generally occur at the beginning of the cycle, while write operations are executed at the end, typically on the edge of the clock signal. This timing separation allows for efficient simultaneous reading and writing within a single register file, eliminating the need for multiple copies.

3. Experiment Result(ALU Only)

1. Show the waveform screen shot of the testbench `tb_alu.0.txt` result



```
a[31:0]          00000000
b[31:0]          00000000
ALU_ctl[3:0]     1100
result[31:0]     ffffffff
zero             0
overflow         0
ansr_result[31:0] ffffffff
ansr_zero        0
ansr_overflow    0
NUM_TEST[31:0]   1
```

The last test case is '0x00000000 0x00000000 NOR 0xffffffff 0 0'; to verify the result, I changed the display radix to match the expected outcome.

2. What other cases you've tested? Why you choose them ?

```
0x7FFFFFFF 0x00000001 ADD 0x80000000 0 1  //overflow
0x80000000 0xFFFFFFFF ADD 0x7FFFFFFF 0 1  //overflow
0x80000000 0x00000001 SUB 0x7FFFFFFF 0 1 //overflow
0x00000003 0x00000002 SUB 0x00000001 0 0 //sub
0x00000001 0x00000002 SLT 0x00000001 0 0 //slt
0xffffffff 0x80000000 SLT          0 1 0 //zero
0x80000000 0x80000000 SLT          0 1 0 //same
```

I've incorporated several overflow scenarios along with straightforward subtraction and 'set less than' (SLT) cases into `tb_alu.0.txt` to validate the accuracy of my code, including equal case, zero case.

4. Problems Encountered & Solution

1. The process of full adder, set less than and overflow

The main focus is on 'msb_bit_alu.v', where the full adder and overflow handling are concepts from last year's digital circuit design course. Hence, I specifically revisited and reviewed the implementation details of these components. As for the 'slt' (set less than) operation, it is a new topic for me. I have been comparing the textbook and lecture slides to understand how to determine the final 'set' condition and how to pass it to the first bit alu.

2. The process of `SUB`

Initially, I was unsure how to implement the 'sub' function, but then I realized that it could be achieved by adjusting the 'b_inverter' and setting 'carryin[0]' to 1. This experience highlighted the importance of understanding the basics of digital circuit design.

3. How to handle the transfer of carry-in and carry-out signals

Initially, I was unsure how to implement the transfer of carry-out from one stage to the next stage's carry-in. After researching online, I addressed this issue by simple syntax of verilog.

```
assign carry_in[31:1] = carry_out[30:0];
```

4. How to configure the operation based on the ALU_ctl setting

Initially, I was uncertain about how to implement a system where a 4-bit ALU control (alu_ctl) dictates the 2-bit operation signals. Later, I simply assigned them as required. I also noticed that each bit in the ALU receives the same 2-bit operation signal, so I adjusted it to [2:0] and ensured they all received identical inputs.

```
 //operation
    reg [1:0] operation;  // flatten vector
    always @(*) begin
        case (ALU_ctl)
            4'b0000: operation = 2'b00;  // AND
            4'b0001: operation = 2'b01;  // OR
            4'b0010: operation = 2'b10;  // ADD
            4'b0110: operation = 2'b10;  // SUB
            4'b1100: operation = 2'b00;  // NOR
            4'b0111: operation = 2'b11;  // SLT
            default: operation = 2'b00;
        endcase
    end
```

5. Overflow and SLT problem

Because overflow is always 0 when operating slt, so I use simple logic to deal with it.

```
assign overflow = (operation == 2'b11) ? 0 : overflow_count;
assign overflow_count =  carry_in ^ carry_out;
assign set = sum ^ overflow_count;
```

6. The remaining bugs and errors

Currently, I'm less familiar with Verilog, encountering errors with almost every command (add, sub, slt, etc.). I address these issues by reviewing the Vivado error messages and making minor adjustments to the code accordingly.

5. Feedback

If there are more test cases, I can be more certain about the correctness of my code (though not absolutely necessary).