

Team Members:

Justin Calma

Fernando Duarte

Kaylynn Khem

Aaron Hung

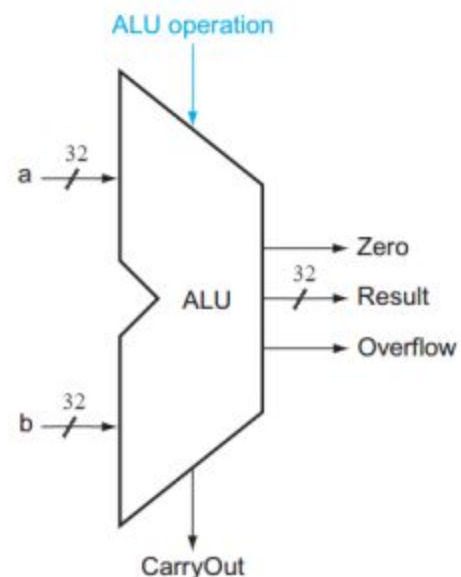
CECS 341 - 04

10/15/2020

### Lab 1: Arithmetic Logic Unit Lab Report

The arithmetic logic unit (ALU) is the brain of the computer, the device that performs the arithmetic operations like addition and subtraction or bitwise operations like AND and OR. It is a fundamental building block of many types of computing circuits such as the central processing unit, floating-point unit, and graphics processing units. The inputs to an ALU are data to be operated on and code indicating the operation to be performed. After the operation is performed results in an output. The ALU's has status inputs and outputs which convey information about a previous or current operation between the external status registers.

The ALU schematic that was given to use was to take a set of inputs, a and b, send them through a 4-bit control line and then compute it's output. In order to do this, we decided to have multiple case statements that will run based on the 4-bit control code that has been passed in. Some of the cases that our ALU will have are three logic gates, being AND, OR and NOR, and several operations to check the values of the inputs. These operations include, ADD, SUB, SLT and Equal Comparison Operation will contribute to verifying the output values of a and b. A few of the outputs include having a carry out flag, zero



flag and an overflow value. In order for us to determine the correct values of the 1-bit outputs, we would need to create conditional branches checking if the result produces a carry out, zero flag, or overflow value. We also need to take into consideration that some operations perform using unsigned or signed inputs.

The ALU design we made is one module that takes in two 32-bit registers, a and b, and a 4-bit ALU operation control line. The design we made also has 4 different types of outputs, a

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	Set less than
1100	NOR
1111	Equal comparison

32-bit register that will store the result of each operation, a 1-bit zero, overflow, and carry out flag. To make our ALU keep running until it has finished the testbench, we make an “always@(...)” loop that will keep looping as long as the testbench passes in inputs a, b, and the 4-bit

binary number determining which operation to perform. Within the “always@(...)” loop we made a case statement that will determine which operation to perform depending on the ALU\_Sel value that was passed in.

The first case checks if the ALU\_Sel value is for the AND operation. Within the first check we perform the AND operation with input a and b, and we store the 32-bit result into our ALU\_Result register. After performing the AND operation, we check if the 32-bit result is either all zeros or not, and we set the zero flag correspondingly. We do this check by having an if statement checking if the result is all zeros, if the condition is true then the zero flag is set to 1, else the zero flag is set to 0. After, carry out and overflow will be set to zero because they only need to be checked in other operations.

The second case checks if the ALU\_Sel value is for the OR operation. Similar to the AND case, we perform the OR operation on the two inputs and save it into the 32-result register.

After, we check if the zero flag is true or not. Then we set carry out and overflow to zero because they only need to be checked in other operations.

The third case checks if the ALU\_Sel value is for the ADD operation. Within the ADD operation case, we perform signed addition between the two inputs, and save the result into the ALU\_Result register. We then check if the ALU\_Result is 0 and set the zero flag to be 1 or 0.

We then have to determine if there was carry out or overflow. In order to do so, we perform unsigned addition and save it into a

temporary register. Checking for carry out, we have an if else if branch. If the MSB of both inputs are one, resulting in the addition operation performed on the last bit to cause a carry out. Else if, after performing the unsigned addition, we check if there was a carry out by checking if the 33rd bit in the temporary register is one. Else, carry

```
// ADD Operation
4'b0010: begin

    // Perform signed ADD and store the result into the Result Register
    Result = $signed(A_in) + $signed(B_in);

    // Check if Result is 0 and set Zero flag
    if (Result == 32'b0000_0000_0000_0000_0000_0000_0000_0000)
        ZeroResult = 1'b1;
    else
        ZeroResult = 1'b0;

    // Perform unsigned ADD to determine carry out
    temp = A_in + B_in;

    // Check if there is a Carry Out in the unsigned ADD operation
    if (A_in[31] == 1 & B_in[31] == 1)
        CarryOutResult = 1'b1;
    else if (temp[32] == 1)
        CarryOutResult = 1'b1;
    else
        CarryOutResult = 1'b0;

    // Check if there is a Overflow in the signed ADD operation
    if (A_in[31] == 0 & B_in[31] == 0 & Result[31] == 1)
        OverflowResult = 1'b1;
    else if (A_in[31] == 1 & B_in[31] == 1 & Result[31] == 0)
        OverflowResult = 1'b1;
    else
        OverflowResult = 1'b0;
```

out did not occur in the addition operation between the two inputs. We then have another if else if branch that will be checking for overflow after performing signed addition. The first if condition checks if both of the inputs are positive numbers and the result becomes a negative number, resulting in overflow. The else if branch checks if the inputs are both negative values and the result becomes a positive value, if this is true then overflow occurred. Else, overflow did not occur and the register is set to 0.

The fourth case checks if the ALU\_Sel value is for the SUB operation. This case is similar to the ADD operation case, however, there is no carry out so it will always be set to zero. Within the SUB operation case, we perform signed SUB with the two inputs and determine if the

zero flag should be set to 1 or 0. Using a similar if else if branch from the ADD case, we add another else if branch checking if overflow occurred when the MSB of the first input is 1, the MSB of the second input is 0, and if the MSB of the result is 0. If this is true then overflow occurred.










The fifth case checks if the ALU\_Sel value is for the SLT operation. To check if input a is less than input b, we compare both values as signed values. If  $\text{signed}(a)$  is less than  $\text{signed}(b)$ , then we set ALU\_Result to be 1, else we set ALU\_Result to be 0. Then we check if ALU\_Result is zero or 1, and set the zero flag. Finally, we set both carry out and overflow registers to have a 0 bit value, since they do not occur in SLT.

The sixth case checks if the ALU\_Sel value is for the NOR operation. We perform the NOR operation by doing the OR operation on inputs a and b, then we logically NOT the result. We then check if the zero flag has been raised or not. Finally, we set both carry out and overflow registers to have a 0 bit value, since they do not occur in the NOR operation.

The last case checks if the ALU\_Sel value is for the equal comparison operation. To do this we create an if condition checking if input a and input b are the same value. If true then ALU\_Result is set to be 1, else it is set to 0. We then check if the zero flag has been raised by using an if condition checking if ALU\_Result is 0. Finally, we set both carry out and overflow registers to have a 0-bit value, since they do not occur in the equal comparison operation.

Overall, our ALU design reuses the same checks for zero, carry out, and overflow outputs in each of the cases. The only difference between all of the cases in our design is that we perform the operation corresponding to the ALU\_Sel line.

The results that we receive from the waveform represent the output of our ALU design for each of the test cases in our testbench. Each wave represents the different

Name	Value
>  tb_alu_sel[3:0]	f
>  tb_din_a[31:0]	00000001
>  tb_din_b[31:0]	00000001
>  tb_alu_out[31:0]	00000001
 overflow	0
 carry_out	0
 zero	0
 point	9.0
>  grade[31:0]	0000003c

I/O registers and wires within our design and testbench. From top to bottom, the first register in our waveform is our 4 bit ALU select input, second is one of our 32 bit register input (a), third is our second 32 bit register input (b), and the last register in our waveform is the 32 bit output register. The next three wires represent our 1 bit outputs, one for overflow, carry out, and zero flag.

The waveform is separated into columns to indicate each function in use. In the first row we have 0, 1, 2, 6, 7, c, f. Each number represents the HEX value of the ALU control line being passed. The rows below the ALU selector are our inputs a and b with their respective functions. The fourth row being the output of the function being tested. In the ALU selector, 0 and 1 are the AND function and the OR function respectively.

In function 2 (addition), the first column of the two inputs result in a carry out. This carry out is confirmed by it being represented as a 1 in the wave form. In the second column of the ADD function, the addition of the two inputs results in an overflow; the overflow being represented as a 1 in this case.

In function 6 (subtraction), the first column has two inputs and their result. In the second column of function 6, the two inputs yield an output that also overflows.

Function 7 (set less than), has one column of inputs and the result being 1. As a result of the output being 1 this means the input a is confirmed to be less than input b.

Function c (NOR), we're comparing the binary digits to each other and perform the NOR function. When comparing the two inputs there does not exist an output that compares two individual bits that are 0, which would never yield 1. This is reflected in the waveform where the output is 0; and the zero flag is indicated as a 1.

Function f (equal comparison), compares two inputs a and b and compares them to see if they're equal. The inputs a and b are both 1 which are both equal. Therefore the result (ALU output) is 1.

359.719 ns

0.000 ns 50.000 ns 100.000 ns 150.000 ns 200.000 ns 250.000 ns 300.000 ns 350.000 ns

0	1	2		6		7	c	f
a5002d77	8086f09d	c182f088	4182f088	c182f088	b182f088	ffffff9	e491c062	00000001
f14c0a81	4e1b0072	d07915c2	507915c3	d07915c2	707915c3	00000006	5b7a7f9d	00000001
a1000801	ce9ff0ff	91fc064a	91fc064b	f109dac6	4109dac5	00000001	00000000	00000001



0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0
00000000	00000006	0000000c	00000012	00000018	0000001e	00000024	00000030	00000036	0000003c