LAB # 2
ALU Designing in Verilog

Objective:

- To design an ALU module in Verilog

System Module (Hardware/Software)

- Visual Studio Code

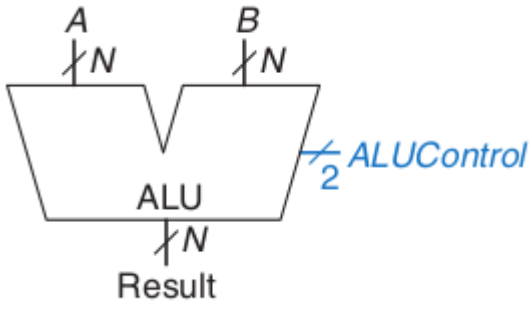
Theoretical Background:

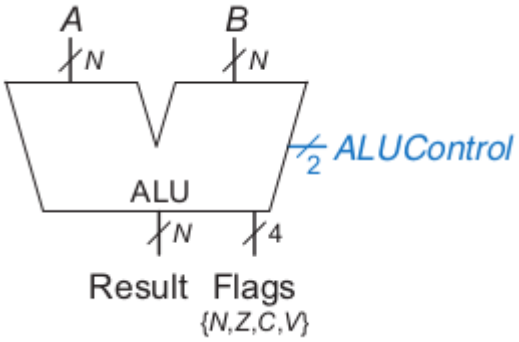
Introduction

In the first two labs, we designed simple circuits. In this lab, we tackle something more formidable, which is the heart of a computer – the arithmetic logic unit (ALU). We will implement an ALU. This ALU will be part of the Processor which we will build in later labs. This lab consists of two individual parts. In this first exercise we will write an HDL description of the ALU, and in the second part we will verify that it works correctly using a test bench.

The ALU

We will design an ALU that will be able to perform a subset of the ALU operations of a full Processor ALU.





In this exercise, we will develop an ALU that will take two 2-inputs, A and B, and is able to execute the following five instructions:

ALUControl	Instruction
000 (add)	lw, sw
001 (subtract)	beq
000 (add)	add
001 (subtract)	sub
101 (set less than)	slt
011 (or)	or
010 (and)	and

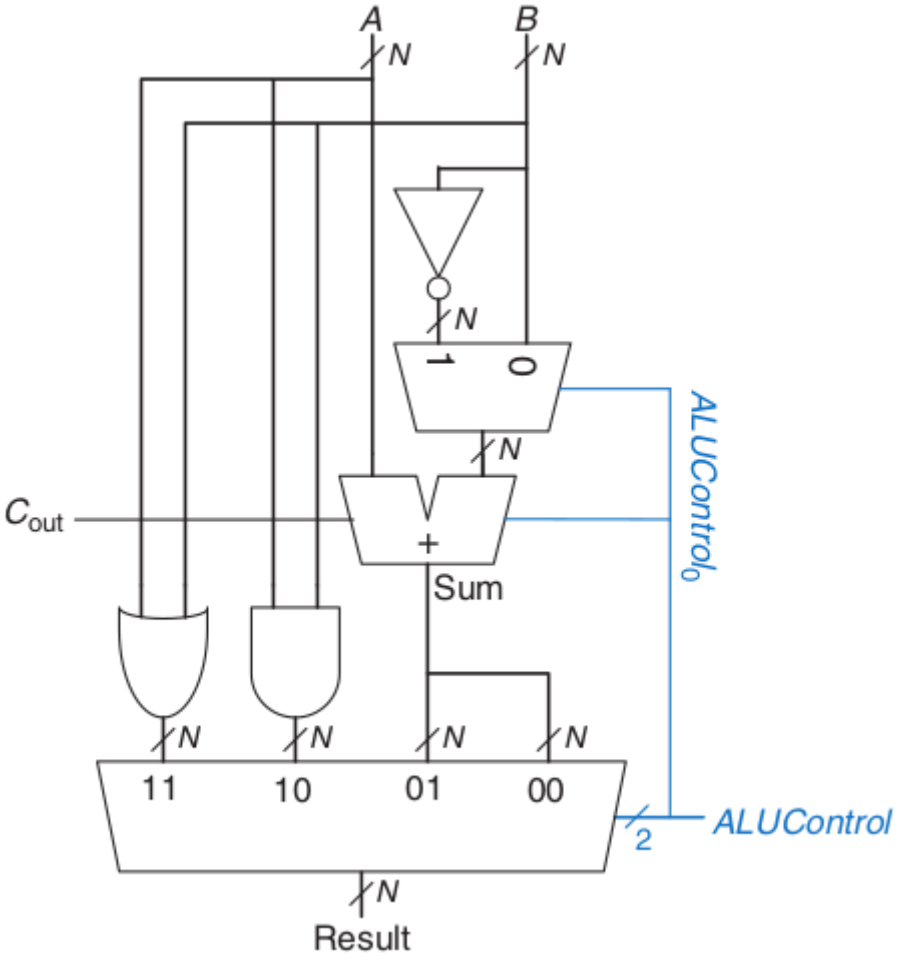
The ALU will generate a 32-bit output that we will call 'Result' and an additional 1-bit flag 'Zero' that will be set to 'logic-1' if all the bits of 'Result' are 0. The different operations will be selected by a 3-bit control signal called 'ALUControl' according to the following table.

For example, when the 'ALUControl' input is '011', the function Result = A or B should be calculated. It is easy to see that there are many values of 'ALUControl' for which no operation has been defined. It is not very important what the circuit does when 'ALUControl'

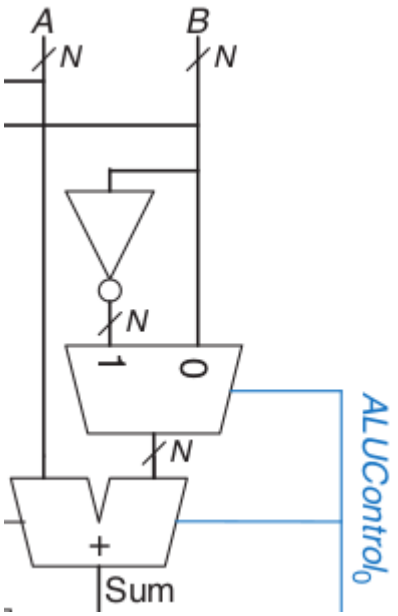
has these values, since the 'Result' will simply be ignored in these cases. You can use this to your advantage to simplify the circuit. Right now, the described operations may look random, but once we learn more about the Instruction set architecture, these choices will make more sense.

Block diagram

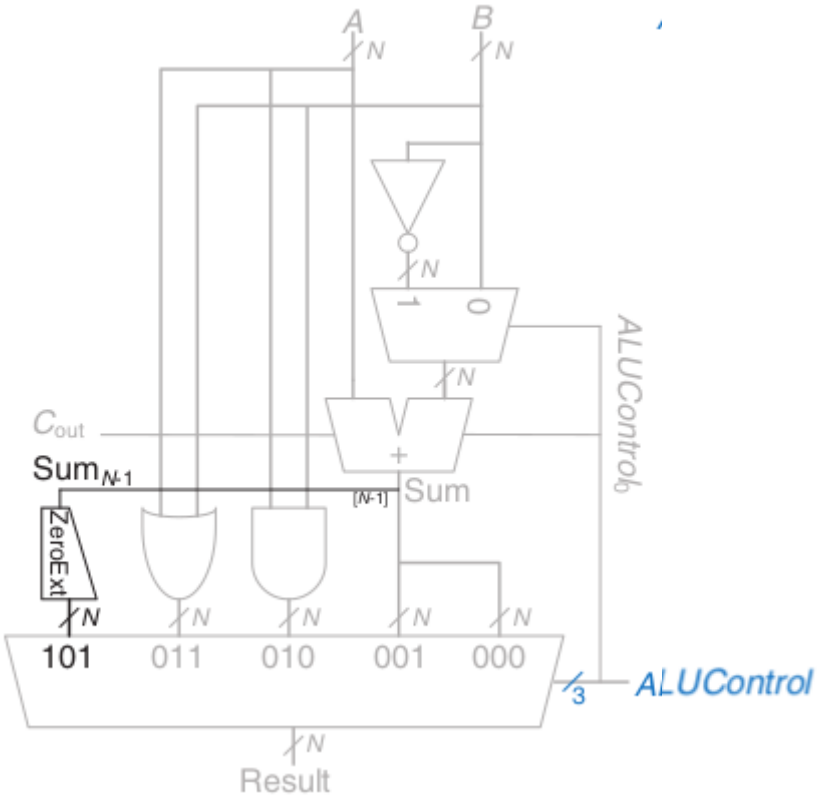
The first order of business should be drawing a block diagram.The following is one approach to analyze what is needed and come up with a block diagram. You are free to follow this example or come up with your own ideas. It is just important that you think about how the circuit should be implemented. Let us first examine the different commands. You should see that we have two types of instructions. The three instructions **add, sub,** and **slt** are arithmetic operations, whereas the two remaining **and, or** are logical operations. Therefore, we have two separate groups of operations. Now let us look at the figure above and determine for which values of ALUControl we perform an operation from which group. It should be pretty clear that when ALUControl[1] is logic-1 we select a logic operation and when ALUControl[1] is logic-0 we have an arithmetic operation. This means that the output of either group can be selected by a 4-input multiplexer that is controlled by ALUControl[1:0]. Figure below shows this distribution.



Now we can take a look at the two groups individually. In the first group we realize that we have an addition (add) or a subtraction (sub, slt). We again could observe that ALUControl[0] is logic-0 for additions and logic-1 for subtractions.. Figure below shows this arrangement.



There is one more thing left, depending on the ALUControl[2] we could select whether we take only the most significant bit (logic-1, slt instruction), or we take the output as it is.



Finally, we also have to add a small circuit that will generate the zero output when the result equals all zeroes. This method of breaking a larger block successively in smaller pieces is called Divide and Conquer and is one of the most important tricks that allow hardware engineers to design very complex circuits.

Note that in the above example, there are many values of ALUControl where the circuit would perform 'strange' operations (100 for example). This is not important because the circuit specification that was given to us said that these inputs were not relevant (this is probably because it can be guaranteed that these inputs will not appear during normal operation).

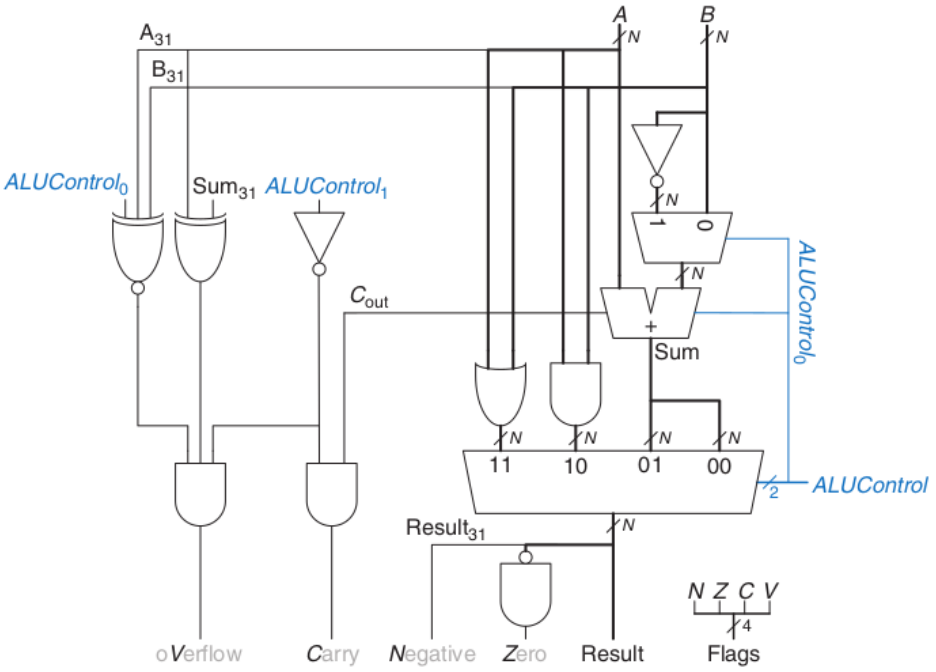
Procedure:

Open Visual Studio Code and perform the procedure mentioned in lab #00 in order to make a Verilog module and test bench.

Lab Tasks:

1. **Write a Verilog module for the above ALU unit. Make sure all input and output signals have a unique name. Make a test bench to test all the operations by taking inputs of your own choices. Attach the test bench and the output waveforms.**

2. **Write a Verilog module for the given ALU figure. Make a test bench to test all the operations by taking inputs of your own choices. Attach the test bench and the output waveforms.e**



Comparison	Signed	Unsigned
=	Z	Z
≠	$\overline{Z}$	$\overline{Z}$
<	N⊕V	$\overline{C}$
≤	Z + (N⊕V)	Z + $\overline{C}$
>	$\overline{Z}$ • ($\overline{N \oplus V}$)	$\overline{Z}$ • C
≥	($\overline{N \oplus V}$)	C

Conclusion:

What have you learnt from this lab?

Learning Outcomes:

Upon successful completion of the lab, students will be able to:

LO1: Design a ALU module in Verilog.