EEE304 – Digital Design with HDL (II)

Lecture 8

Dr. Ming Xu

Dept of Electrical & Electronic Engineering

XJTLU

In This Session

- Constructing an Arithmetic Logic Unit

ALU

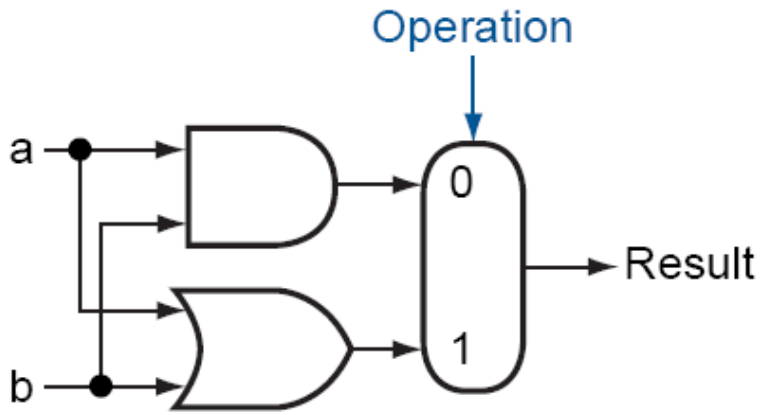
- We will construct an ALU that can perform:
  - Arithmetic operations, e.g. addition and subtraction
  - Logical operations, e.g. AND, OR and NOR
  - Set on less than
  - Shifting is not included, because it is normally done outside the ALU, by another circuit called a barrel shifter.

ALU

- The ALU will be constructed from the four building blocks:
  - AND gates, OR gates, inverters, and multiplexers.
- We need a 32-bit ALU, which can be built by connecting 32 1-bit ALUs.
- We will start from a 1-bit ALU.

AND and OR

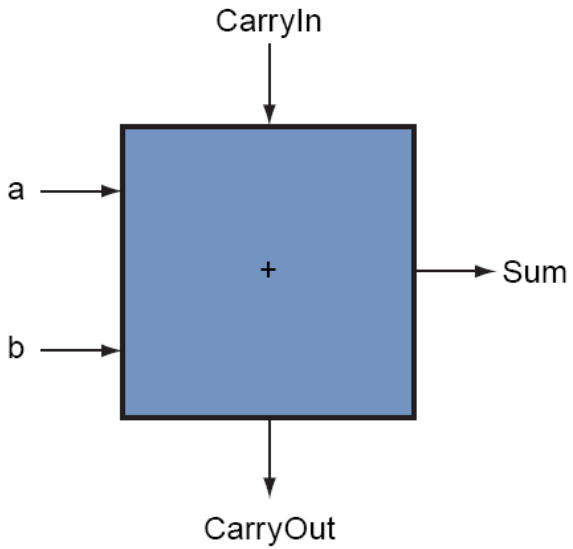
- The 1-bit logical unit for AND and OR



- When *Operation* is 0, a AND b is selected; otherwise a OR b is selected.

Addition

- The 1-bit adder

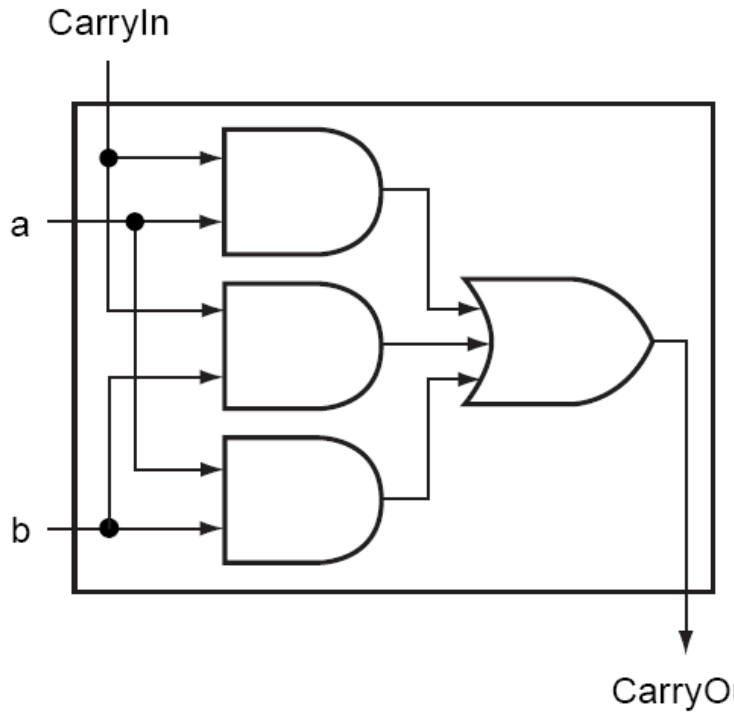


Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Addition

- The 1-bit adder

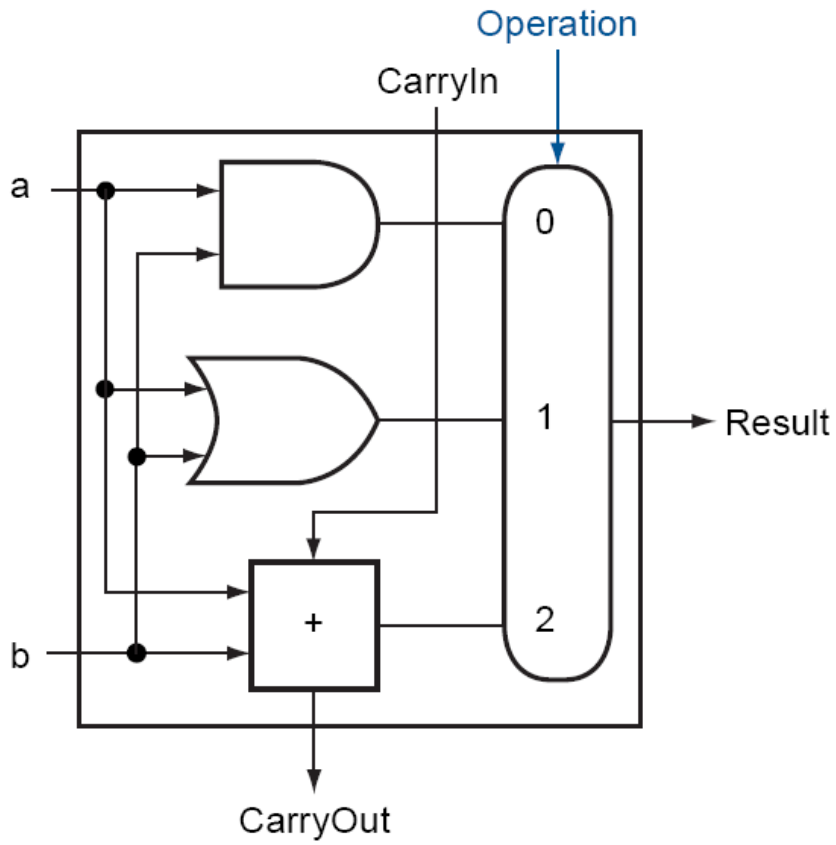
$$\begin{aligned}\text{CarryOut} &= (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn}) \\ &= (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)\end{aligned}$$



$$\begin{aligned}\text{Sum} &= a \oplus b \oplus \text{CarryIn} \\ &= (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) \\ &+ (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})\end{aligned}$$

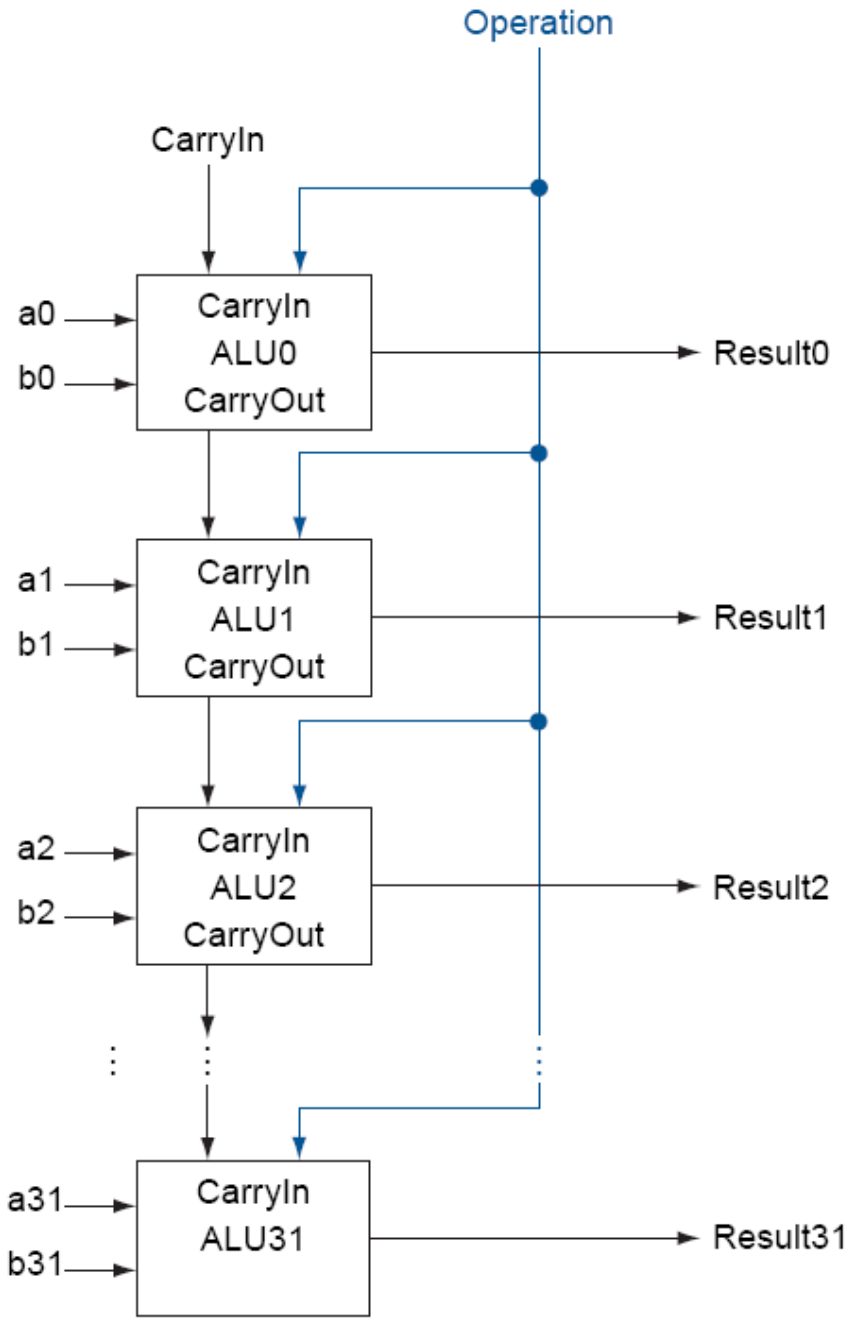
Addition

- The 1-bit ALU that performs AND, OR and addition



Addition

- A 32-bit ALU constructed from 32 1-bit ALUs.
- It is a *ripple carry* adder.

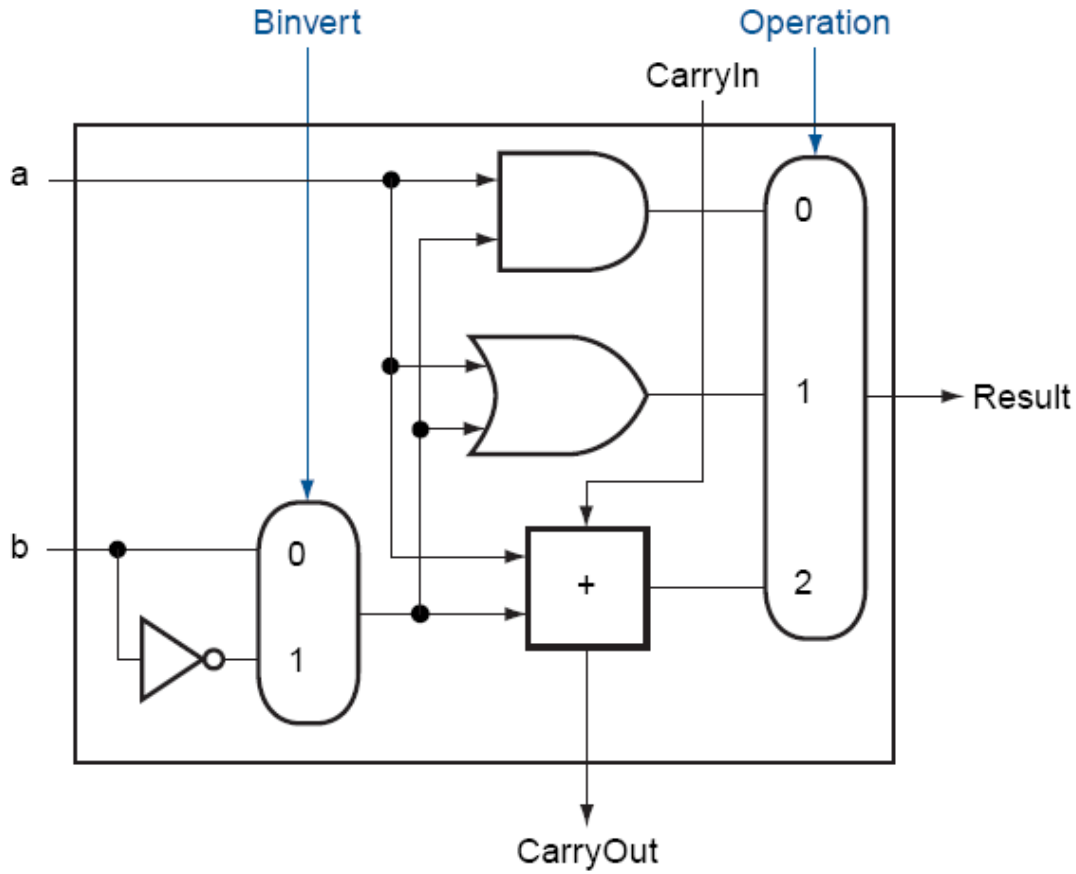


Subtraction

- Subtraction is the same as adding the negative version of a number.
- In 2’s complement system, negating a number is to invert each bit and then add 1.
- An additional control, *Binvert*, is used to decide whether to invert the second operand.

Subtraction

- The 1-bit ALU that performs AND, OR, addition of a and b, or addition of a and $\overline{b}$



Subtraction

- Suppose we connect 32 of these 1-bit ALUs, if we
  - set the CarryIn of the LSB to 1 instead of 0
  - select the inverted b by setting Binvert to 1
- Then it will implement the subtraction.

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

- The simplicity helps explain why 2's complement system has become universal.

NOR

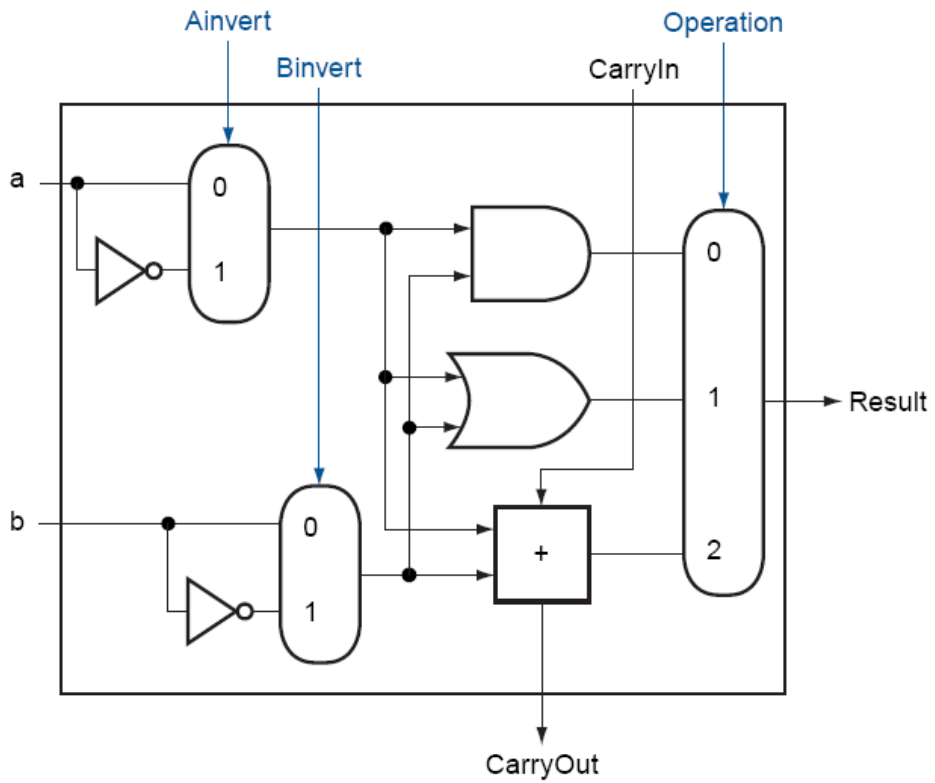
- We can implement NOR using the hardware already in the ALU.

$$\overline{(a + b)} = \bar{a} \cdot \bar{b}$$

- Since we have AND and NOT b, we only need to add NOT a to the ALU.
- By selecting $\bar{a}$ (Ainvert = 1) and $\bar{b}$ (Binvert = 1), we get a NOR b instead of a AND b.

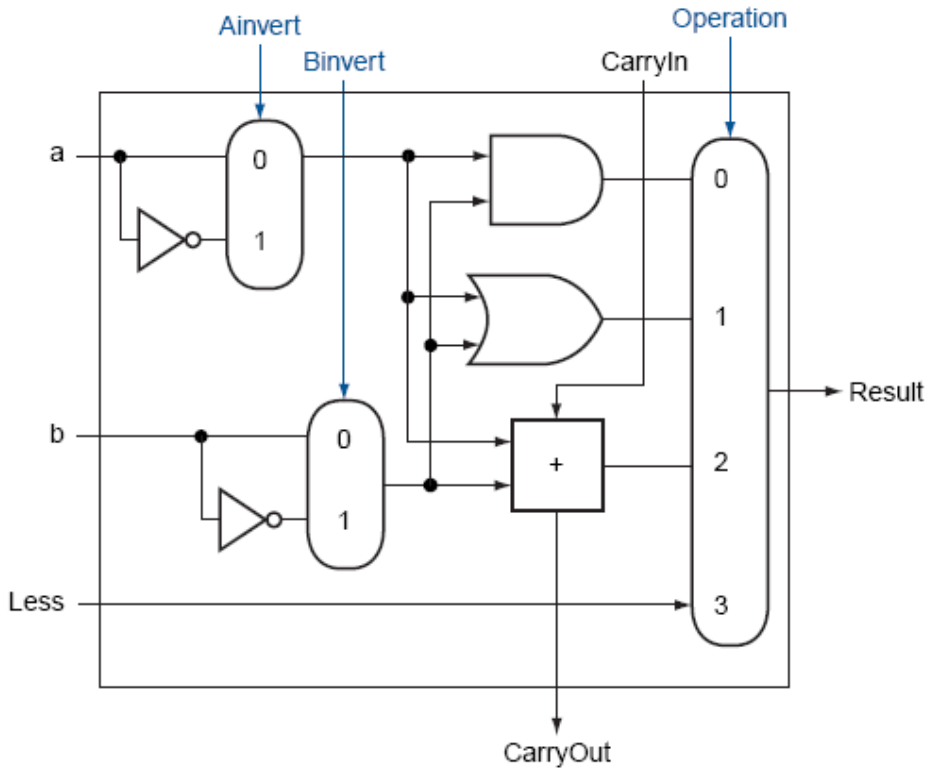
NOR

A 1-bit ALU that performs AND, OR, and addition on a and b or $\overline{a}$ and $\overline{b}$.



Set on less than

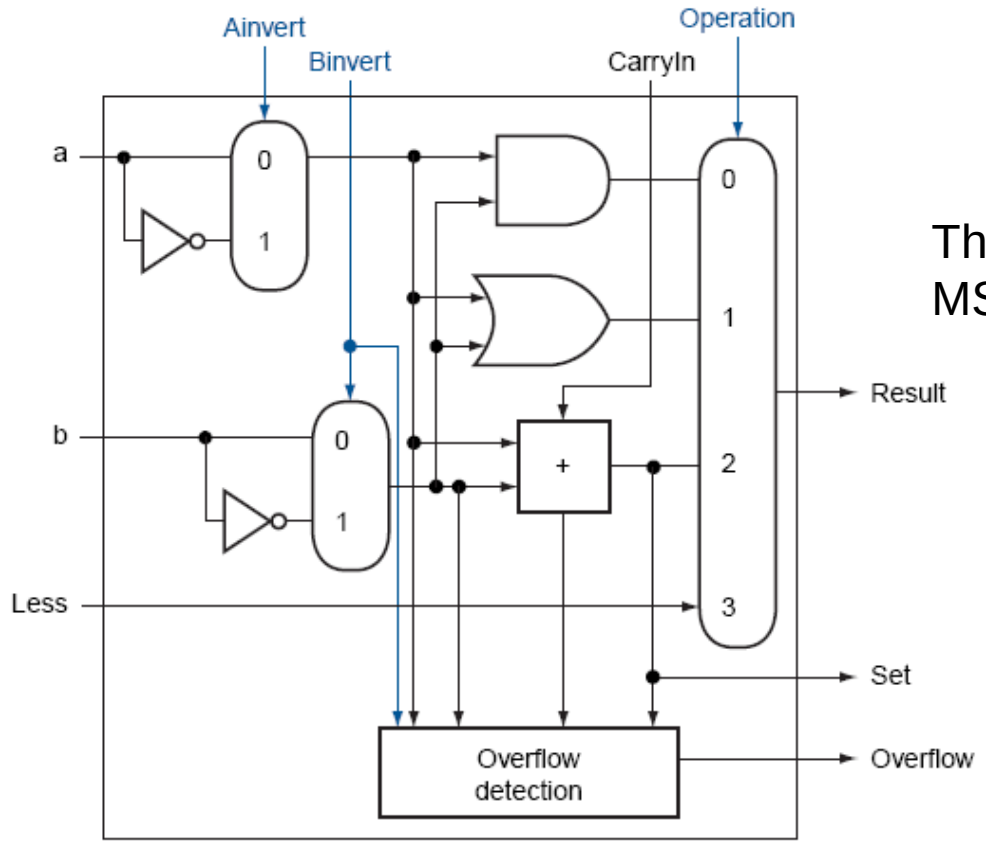
- It produces 1 if a < b, and 0 otherwise.
- We need a new input *Less*, which is always 0 for the upper 31 bits of the ALU.



The 1-bit ALU for lower 31 bits

Set on less than

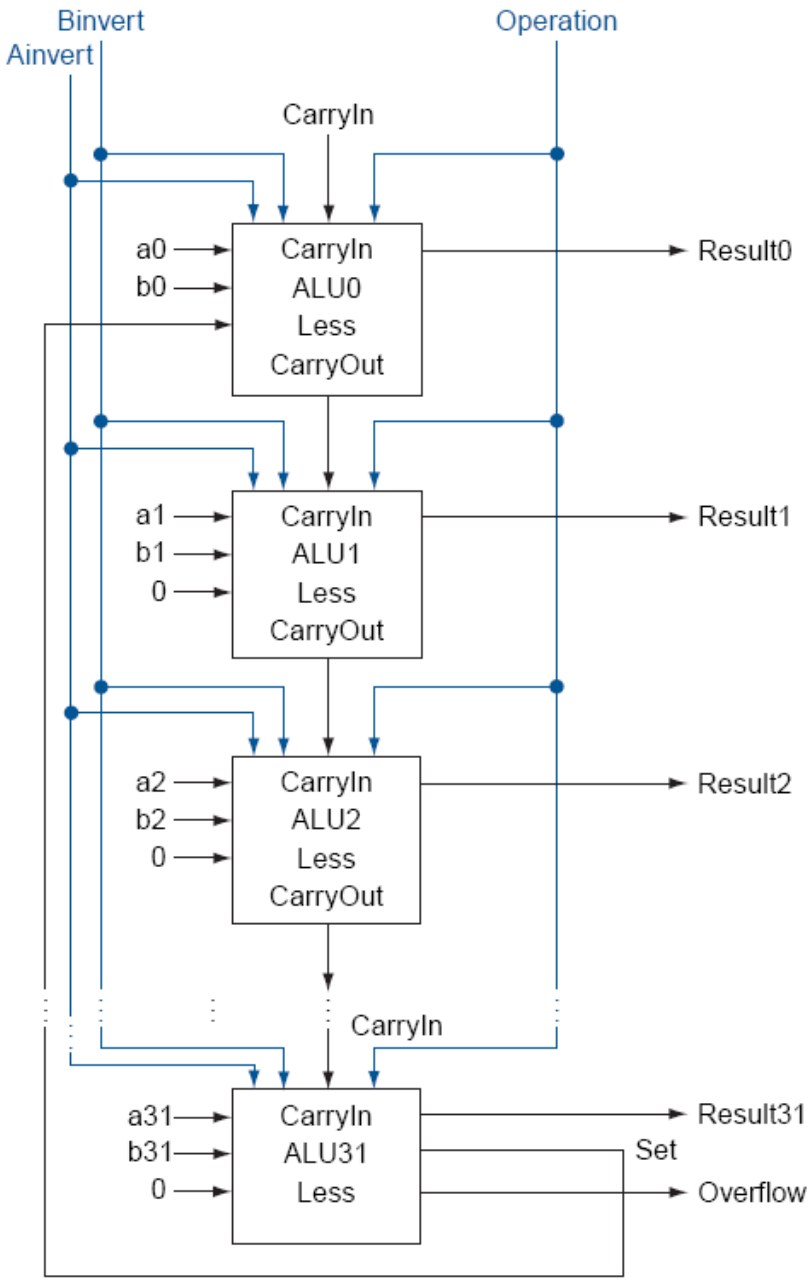
- The LSB of the ALU is 1 if a < b, and 0 otherwise.
- The MSB of a – b is 1 (negative) when a < b, and 0 otherwise



The 1-bit ALU for MSB.

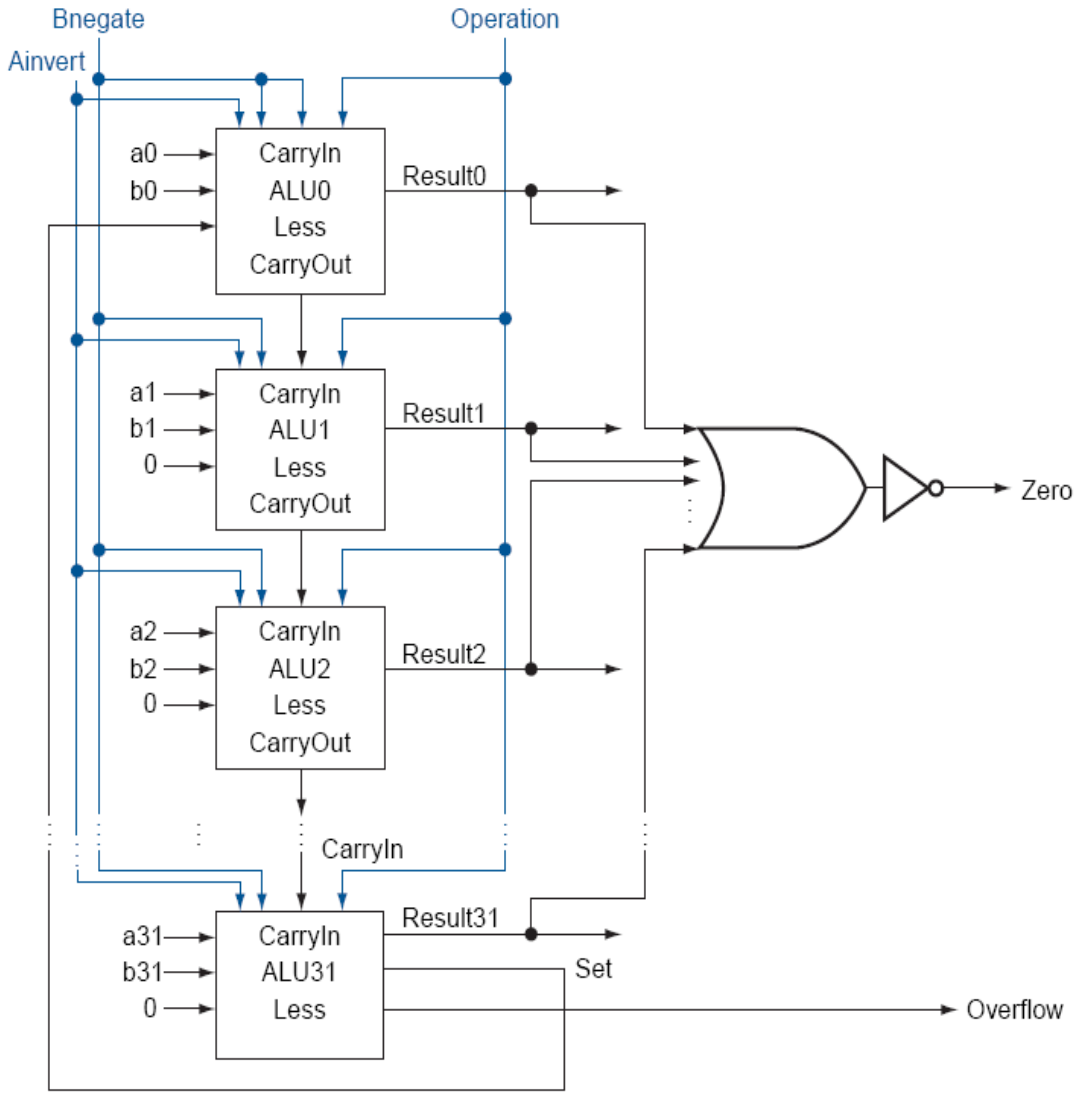
Set on less than

- The Set of the MSB is used as the *Less* input of the LSB.
- The *Carryin* of the LST is set to 1 for the subtraction.



Testing Zero

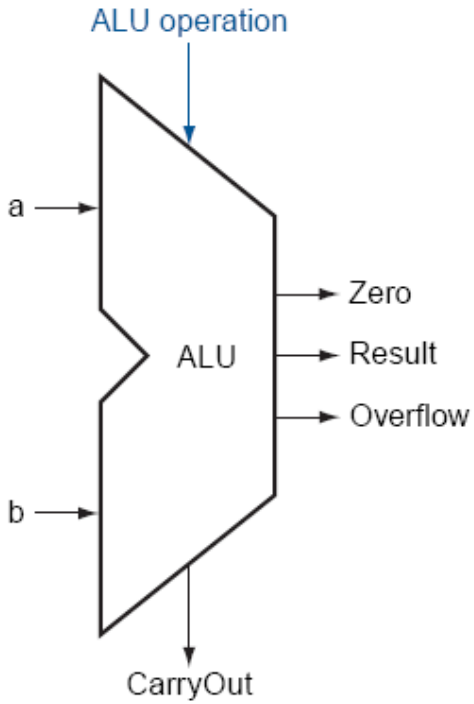
- This is to support conditional branch instructions.



The ALU

We can think of the combination of the 1-bit Ainvert line, the 1-bit Binvert line, and the 2-bit Operation lines as 4-bit control lines for the ALU.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR



The MIPS ALU in Verilog

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```