Computer Arithmetic

Subtraction

- Normal Binary subtraction :

Binary Subtraction

- 0 – 0 = 0
- 1 – 0 = 1
- 1 – 1 = 0
- 0 – 1 = 1 (Borrow 1)

- **<u>Binary Subtraction thorough 2's complement:</u>**

The operation is carried out by means of the following steps:

(i) At first, 2's complement of the subtrahend is found.

(ii) Then it is added to the minuend.

(iii) If the final carry over of the sum is 1, it is dropped and the result is positive.

(iv) If there is no carry over, the two's complement of the sum will be the result and it is negative.

Range of number represented by 2's complement = $(-2^{n-1}$ to $2^{n-1} - 1)$

Subtraction

32 bit

7−6

7+(−6) 2's

	0000 0000 0000 0000 0000 0000 0000 0111_{two}	$= 7_{ten}$
-	0000 0000 0000 0000 0000 0000 0000 0110_{two}	$= 6_{ten}$
=	0000 0000 0000 0000 0000 0000 0000 0001_{two}	$= 1_{ten}$

or via addition using the two's complement representation of −6:

	0000 0000 0000 0000 0000 0000 0000 0111_{two}	$= 7_{ten}$
+	1111 1111 1111 1111 1111 1111 1111 1010_{two}	$= -6_{ten}$
=	0000 0000 0000 0000 0000 0000 0000 0001_{two}	$= 1_{ten}$

-4 & -5 -4 +5
(1

- Overflow if result is out of range

  - subtracting two +ve or two -ve operands = no overflow

    (when the signs of the operands are the same, overflow cannot occur. To see this, remember that c- a = c + (-a) because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up by adding operands of different signs)

  - subtract +ve from –ve operands; overflow if sign bit is 0 (answer is positive)

  - subtract -ve from +ve operands; overflow if sign bit is 1 (answer is negative)

- Overflow in 2’s complement:

  - Overflows occur when we add two numbers with the same sign (both positive or both negative) and the result has the opposite sign.

  -When adding numbers in two's complement, if the carry-out and the carry-on into the most significant bit (sign bit) are different that means an overflow has occurred.

- Subtract 1 from -8 through 2's complement :

  (add -8 and -1 =-9)

```
-8 + (-1) = -9

 1000  (carry)
  1000 (-8)
+ 1111 (-1)
------
 0111 (+7) OVERFLOW!
```

The carry-out is 1 and the carry-on to sign bit (MSB) is 0.

- Subtract -1 from 7 through 2’s complement :

  (add 7 and 1 = 8)

```
+7 + 1 = +8

 0111  (carry)
  0111 (+7)
+ 0001 (+1)
------
  1000 (-8) OVERFLOW!
```

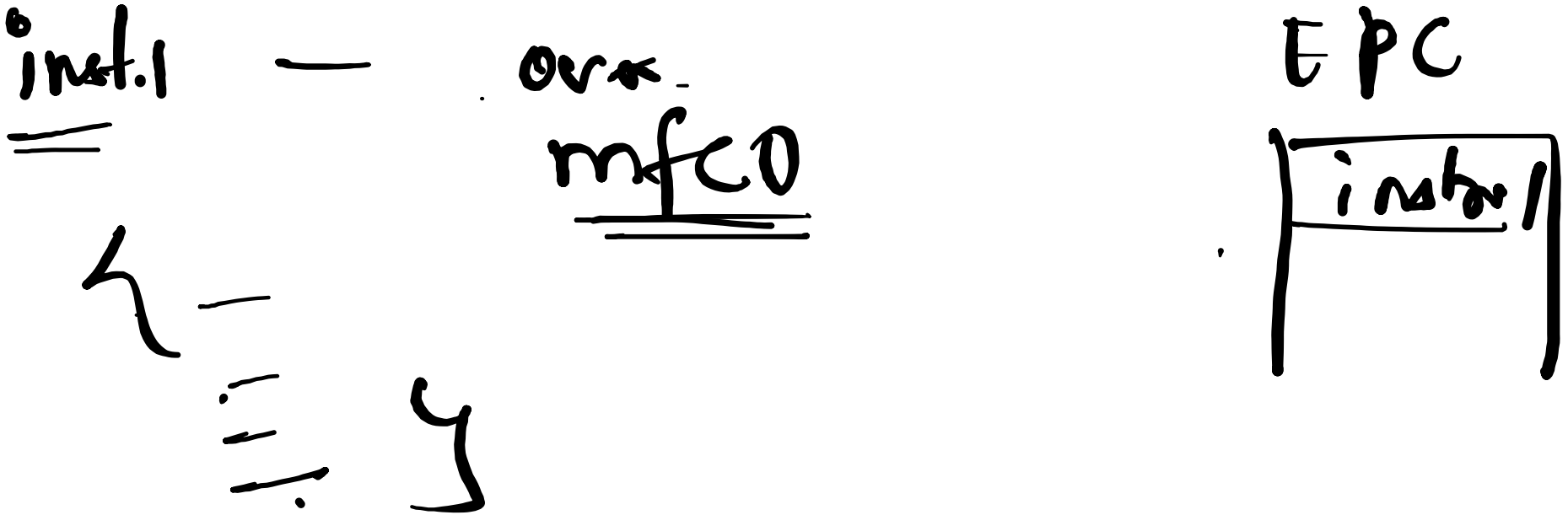
The carry-out is 0 and the carry-on to sign bit (MSB) is 1.

Handling overflows in MIPS

- The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:
    - Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.
    - Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do not cause exceptions on overflow.

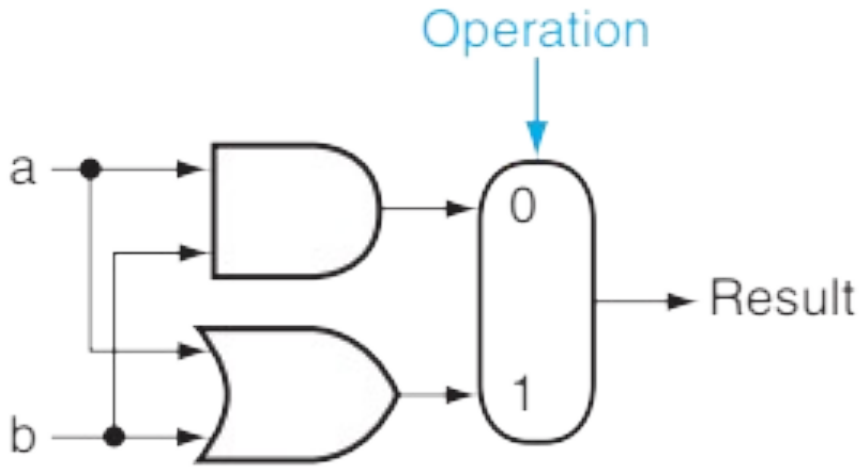
- The computer designer must decide how to handle arithmetic overflows. Although some languages like C and Java ignore integer overflow, languages like Ada and Fortran require that the program be notified.
- The programmer or the programming environment must then decide what to do when overflow occurs.
- MIPS detects overflow with an exception, also called an interrupt on many computers.
- An exception or interrupt is essentially an unscheduled procedure call that disrupts program execution and used to detect overflow.
- The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception.
- The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

- MIPS includes a register called the exception program counter (EPC) to contain the address of the instruction that caused the exception. The instruction move from system control (mfc0) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.



A 1 bit ALU

- Logical operations : The 1-bit logical unit for AND and OR looks like the figure below:

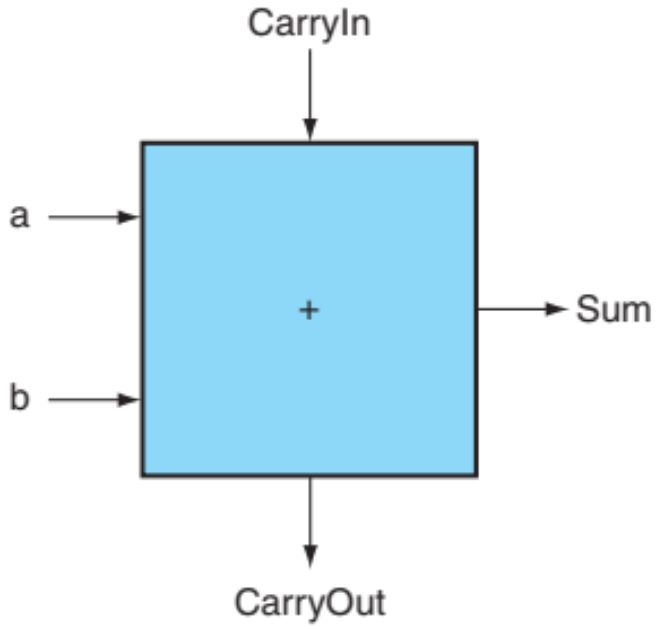


- The multiplexor on the right then selects a AND b or a OR b, depending on whether the value of Operation is 0 or 1.
- The line that controls the multiplexor is shown in color to distinguish it from the lines containing data.

Adder (or full adder)

- Addition Operations :

The next function to include is addition. An adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called CarryOut. Since the CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called CarryIn.



- Truth table for 1 bit Full adder :

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

- Logic expression for Sum :

Sum = A' B' C-IN + A' B C-IN' + A B' C-IN' + A B C-IN

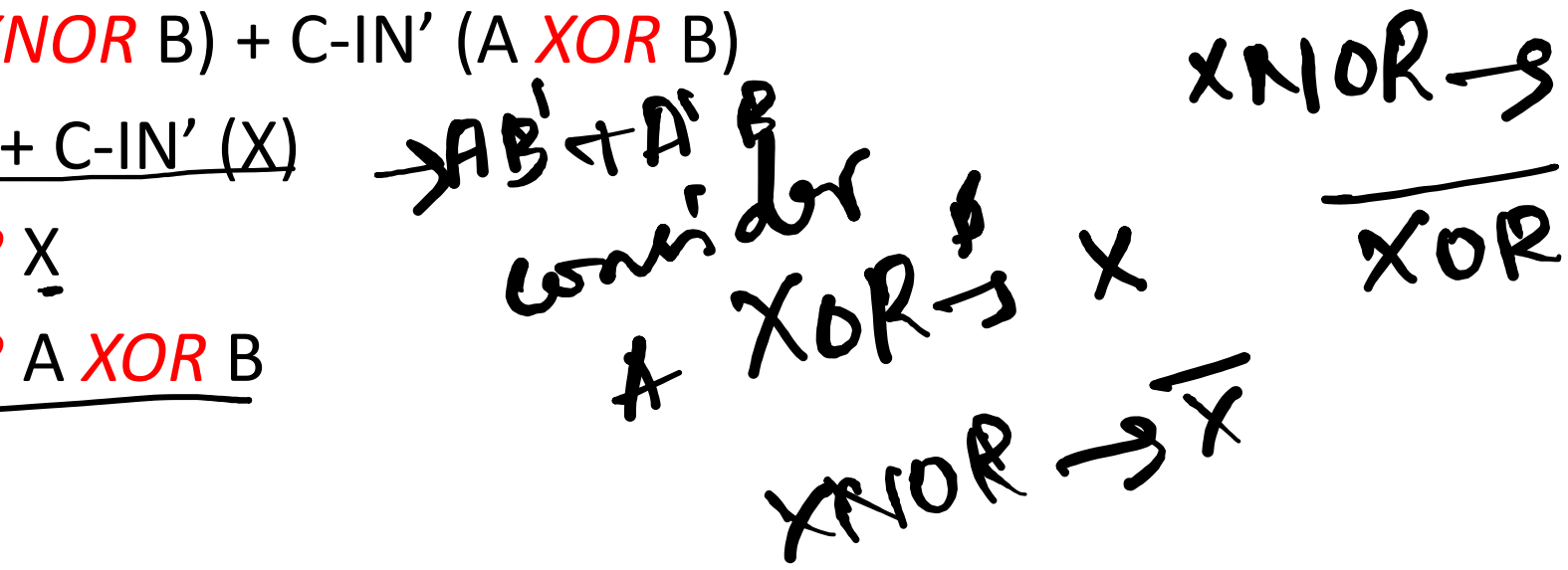
= C-IN (A' B' + A B) + C-IN' (A' B + A B')

= C-IN (A *XNOR* B) + C-IN' (A *XOR* B)

= C-IN (X') + C-IN' (X)

= C-IN *XOR* X

= C-IN *XOR* A *XOR* B



- Logical expression for Carry -out

Carry-out = A' B C-IN + A B' C-IN + A B C-IN' + A B C-IN

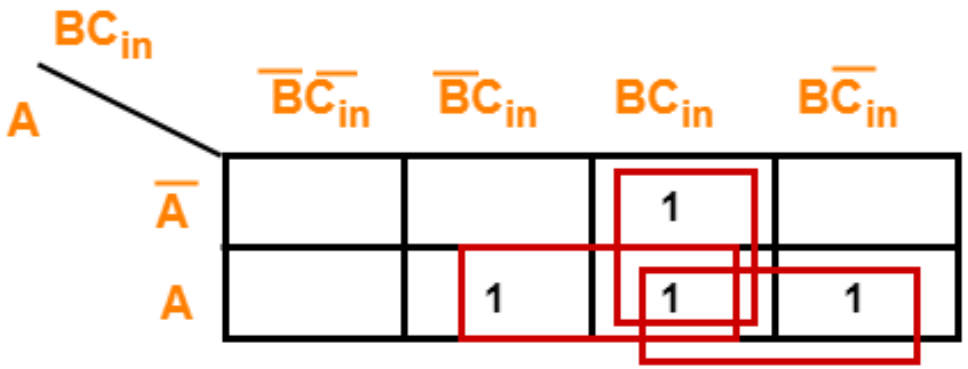
= A' B C-IN + A B' C-IN + AB(C-IN' + C-IN)

= A' B C-IN + A B' C-IN + AB

= C-IN (A'B + AB') + AB

= C-IN (A *XOR* B) + AB

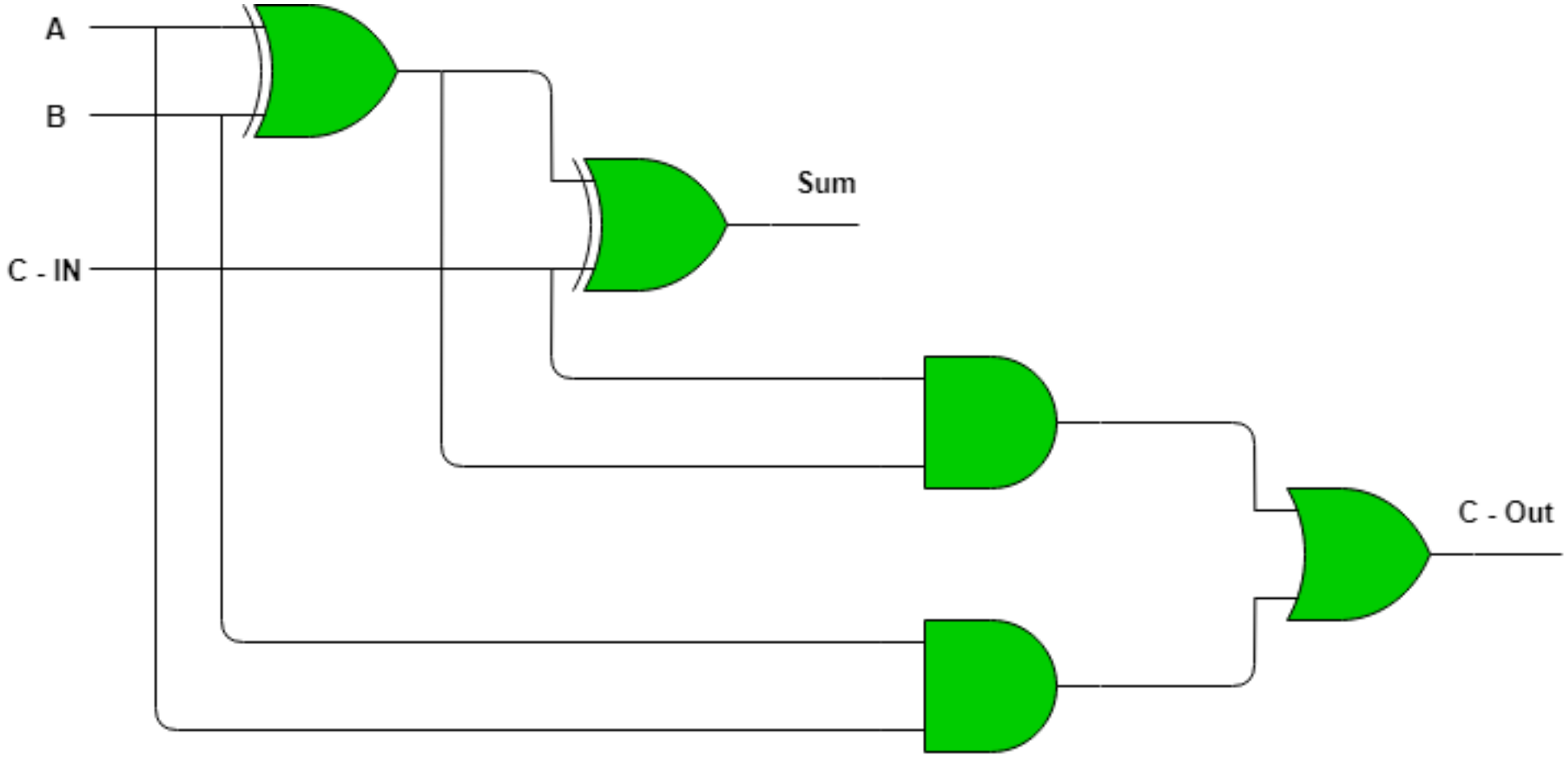
Method 2:



$C_{out} = AB + BC_{in} + C_{in}A$

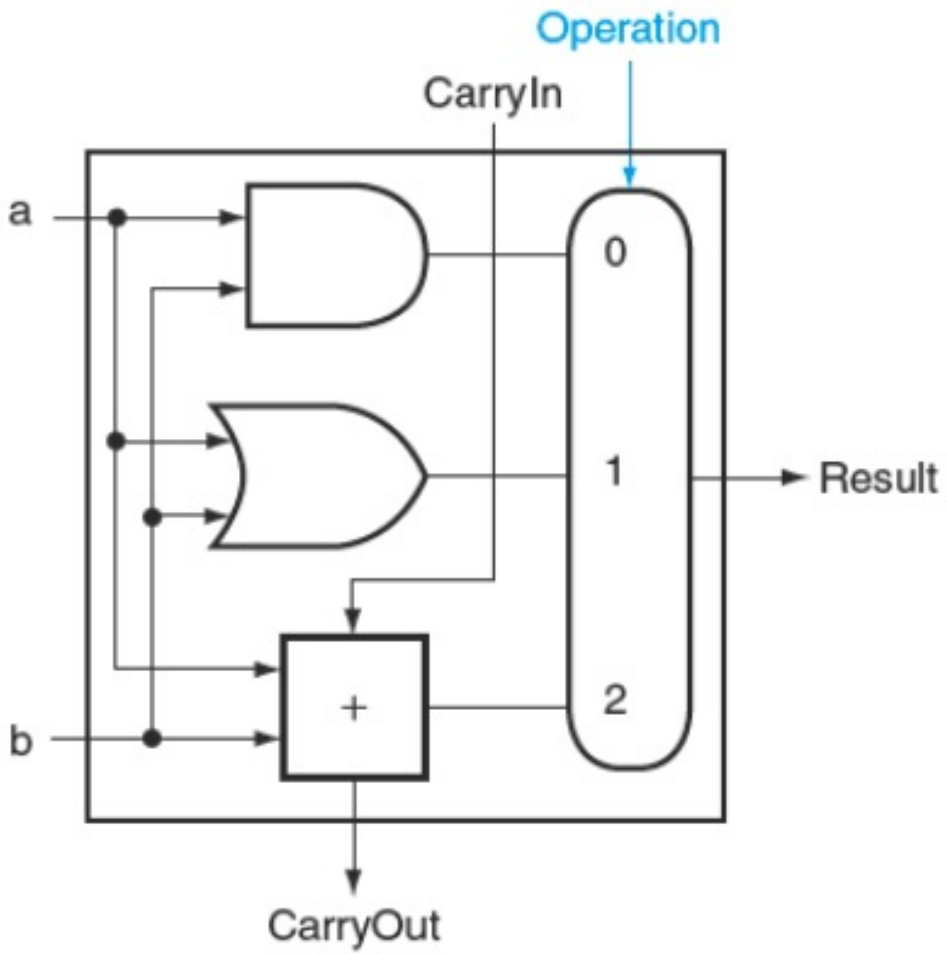
Cout →
1. AND, OR
2. AND, XOR, OR

- Circuit diagram for Full adder



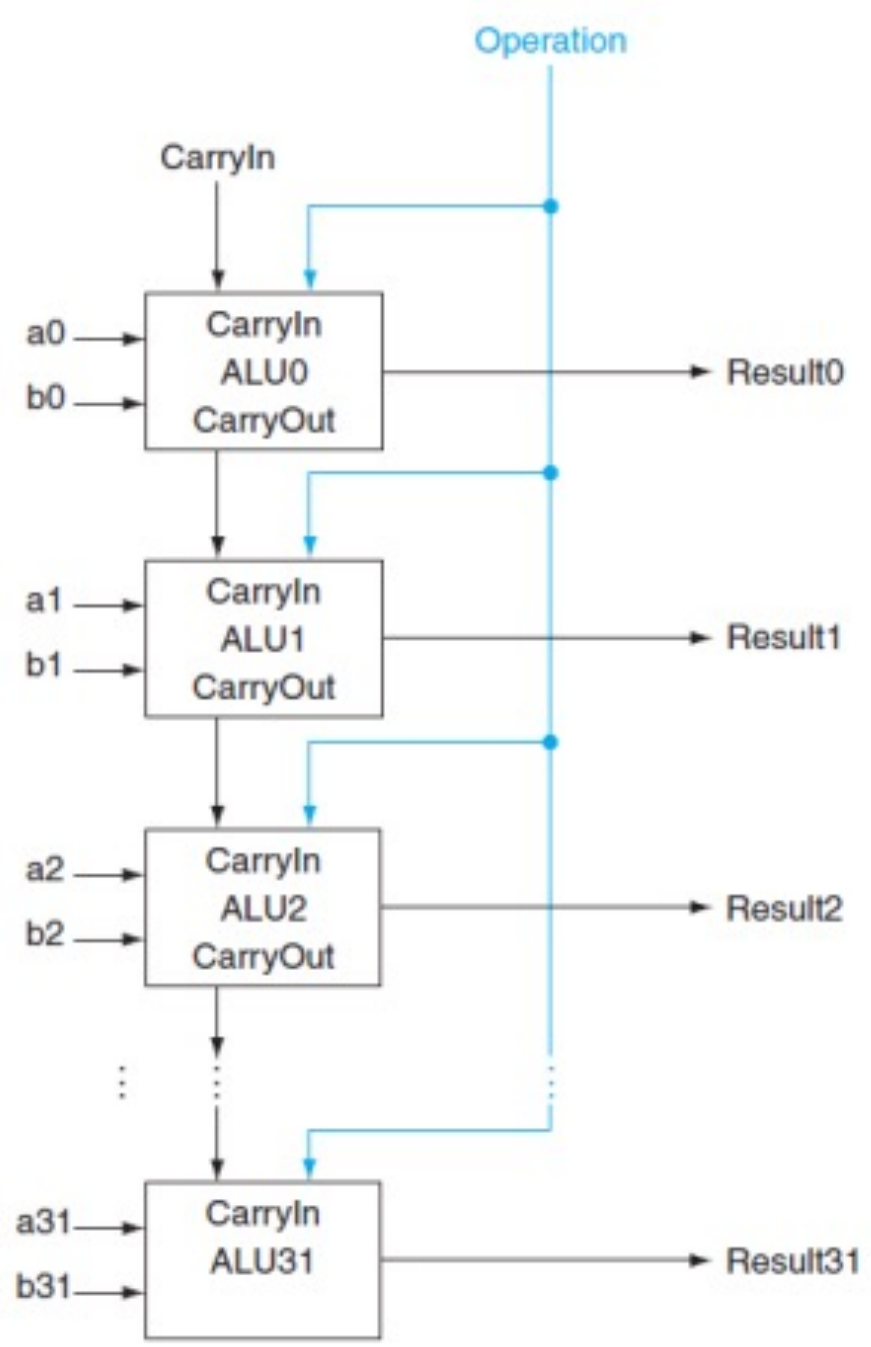
1 bit ALU (ADD, AND, OR)

- Sometimes designers also want the ALU to perform a few more simple operations, such as generating 0.
- The easiest way to add an operation is to expand the multiplexor controlled by the Operation line and, for this example, to connect 0 directly to the new input of that expanded multiplexor.



32 bit ALU

- Now that we have completed the 1-bit ALU, the full 32-bit ALU is created by connecting adjacent “black boxes.”
- Just as a single stone can cause ripples to radiate to the shores of a quiet lake, a single carry out of the least significant bit (Result0) can ripple all the way through the adder, causing a carry out of the most significant bit (Result31).
- Hence, the adder created by directly linking the carries of 1-bit adders is called a ripple carry adder.



1 bit ALU (ADD,SUB, AND, OR)

- Subtraction is the same as adding the negative version of an operand, and this is how adders perform subtraction.
- The shortcut for negating a two's complement number is to invert each bit (sometimes called the one's complement) and then add 1.
- To invert each bit, we simply add a 2:1 multiplexor that chooses between b and b