Chapter 3

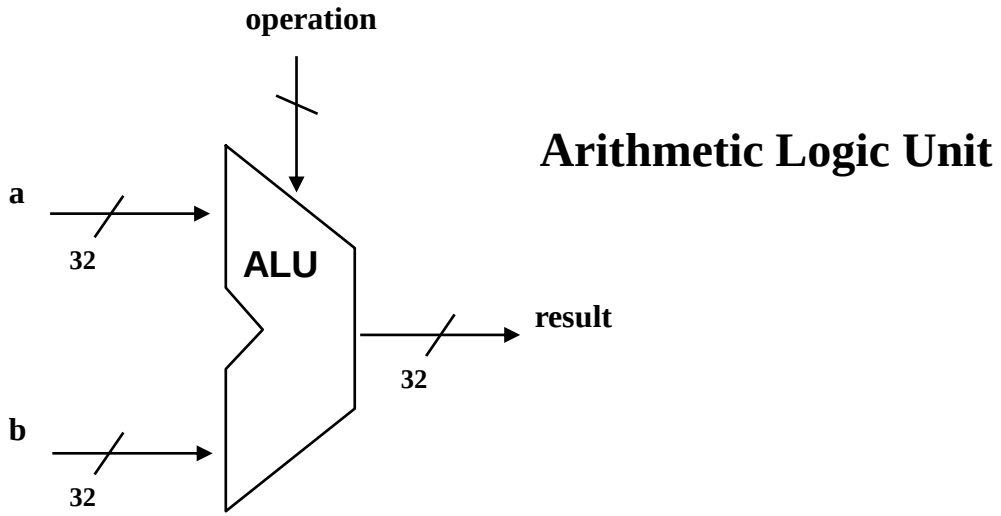
Arithmetic for Computers

Concepts Introduced in Chapter 3

- two's complement integer representation and operations on this representation
- IEEE Floating-Point Representation
- basic logic operations and hardware building blocks
- description of a simple 32-bit ALU
- floating-point representation
- multiplication and division

Arithmetic

- **Where we've been:**
  - **Abstractions:**
    Instruction Set Architecture
    Assembly Language and Machine Language
- **What's up ahead:**
  - **Implementing the Architecture**



Binary Representations

- **32 bit *unsigned* numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
...
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Binary Representations - MIPS

- **Two's complement representation can represent both positive and negative values.**
- **32 bit *signed* numbers:**

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = + 2ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten   maxint
1000 0000 0000 0000 0000 0000 0000 0000two = – 2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = – 2,147,483,647ten   minint
1000 0000 0000 0000 0000 0000 0000 0010two = – 2,147,483,646ten
...
1111 1111 1111 1111 1111 1111 1111 1101two = – 3ten
1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```

Two's Complement Operations

- **Negating a two's complement number: *invert all bits and add 1***
  - **remember: "negate" and "invert" are quite different!**

- **Example 1:** $3_{10} = 0000000000\ 0000000000\ 0000000000\ 11_2$

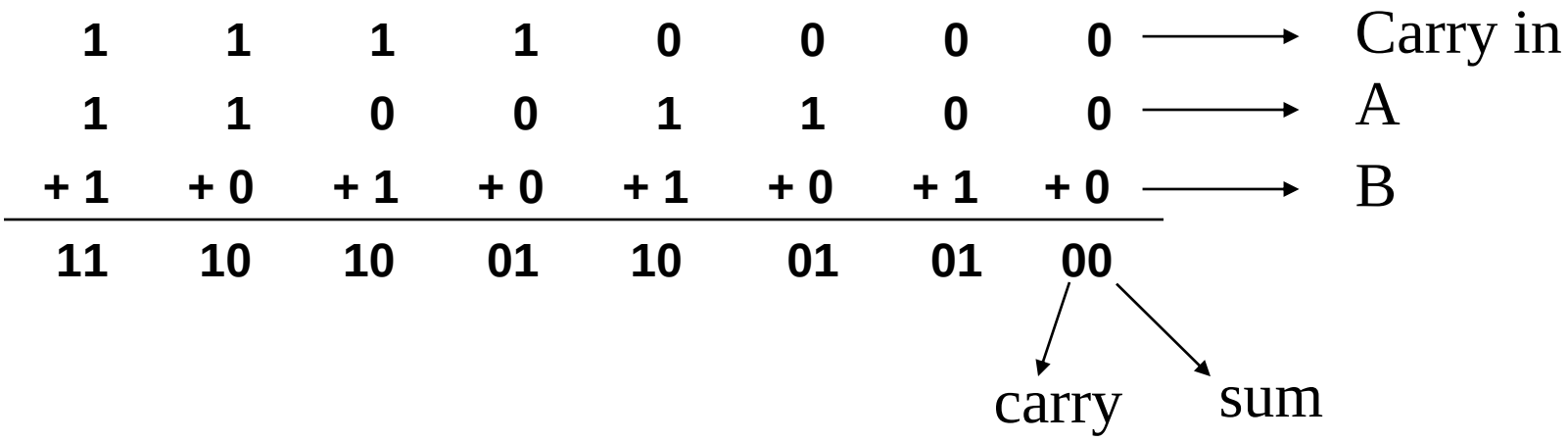
$$-3_{10} = 1111111111\ 1111111111\ 1111111111\ 00_2 + 1$$
$$= 1111111111\ 1111111111\ 1111111111\ 01_2$$

- **Example 2:** $-3_{10} = 1111111111\ 1111111111\ 1111111111\ 01_2$

$$3_{10} = 0000000000\ 0000000000\ 0000000000\ 10_2 + 1$$
$$= 0000000000\ 0000000000\ 0000000000\ 11_2$$

Examples: Addition with carries

- **1-bit addition with carry: Just like in grade school**



Example:

```
   01111000   (carries)
   00101101   (45)
+  00011110   (30)
   --------   ----
   01001011   (75)
```

Examples: Addition & Subtraction

- **Addition:**

```
  00000101   (5)        00000101    (5)
+ 00000100   (4)      + 00001011   (11)
  00001001   (9)        00010000   (16)
```

- **Subtraction: Two's complement operations easy**
  - **subtraction using addition of negative numbers**

    Note: x-y = x+(-y)

2-5 = 2+(-5) = -3

```
  00000010   (2ten)
+ 11111011   (-5ten)
  11111101   (-3ten)
```

7-6 = 7+(-6) = 1

```
  00000111   (7ten)
+ 11111010   (-6ten)
1 00000001   (1ten)
```

ignore this bit

Overflow

- **Overflow: result too large for finite computer word**
  - **e.g., adding two n-bit numbers does not yield an n-bit number**
  - **e.g., adding two positive integers yield a negative integer**

```
  0111 (+7)
+ 1001 (-7)
-----------
1 0000  (0)
```

hardware overflow

```
 0111 (+7)
+0110 (+6)
----------
 1101 (-3)
```

sign overflow

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative (add)
  - or, adding two negatives gives a positive (add)
  - or, subtract a negative from a positive and get a negative (sub)
  - or, subtract a positive from a negative and get a positive (sub)

Overflow Conditions

	Operation	Operand A	Operand B	Overflow when result is
add	A+B	>=0	>=0	<0
add	A+B	<0	<0	>=0
sub	A-B	>=0	<0	<0
sub	A-B	<0	>=0	>=0

Overflow Conditions (in Truth Tables)

- For A+B (add):

A	B	Result (R)	Overflow?
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

A'B'R+ABR'

- For A-B (sub):

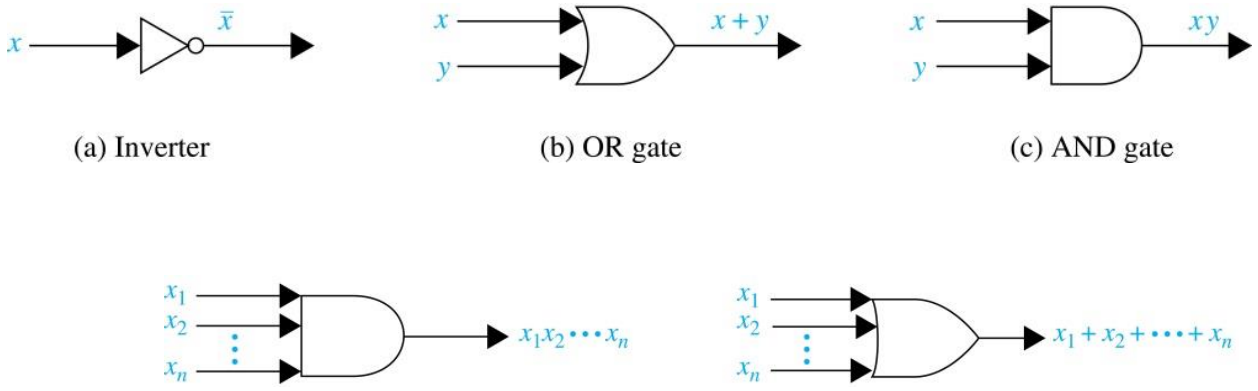
A	B	Result (R)	Overflow?
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

A'BR+AB'R'

Logic Gates

- **In digital circuits, we use**
  - **high voltage (2 volts - 5 volts) to represent 1 (true)**
  - **low voltage (0 volts - 0.8 volts) to represent 0 (false)**

- **By using high/low voltages, we are able to design digital gates:**



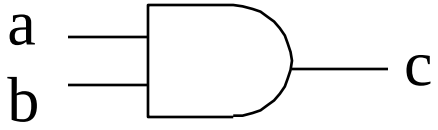
(a) Inverter (b) OR gate (c) AND gate

Boolean Algebra & Gates (See Appendix B)

- **And :**

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

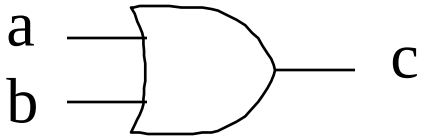
Notation: a•b (or just ab)



- **Or:**

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

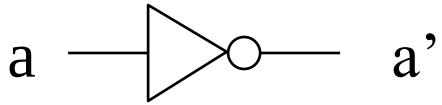
Notation: a+b



- **Inverter (not):**

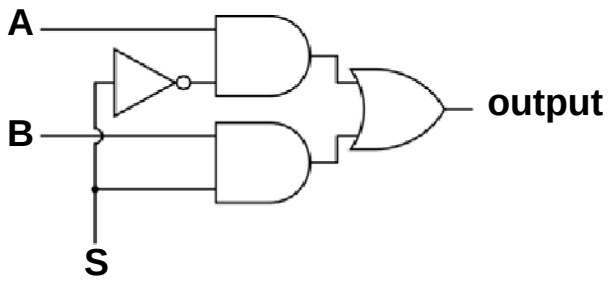
a	a'
0	1
1	0

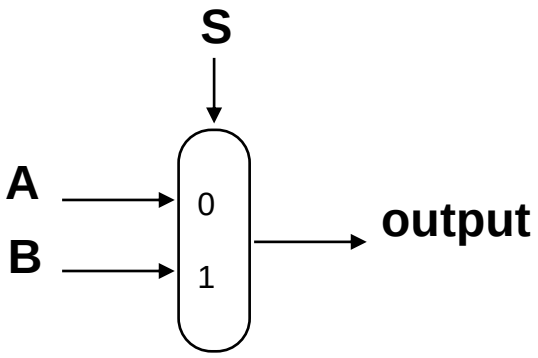
Notation: a' (or $\overline{a}$)



Review: The Multiplexer

- **Selects one of the inputs to be the output, based on a control input**



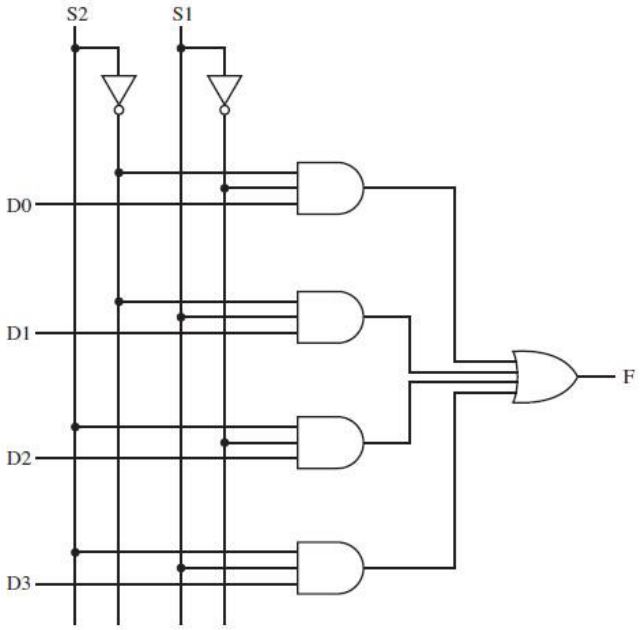


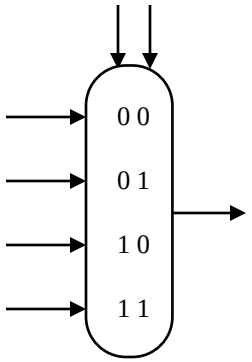
s	output
0	A
1	B

(We will build our ALU using a MUX)

Review: The Multiplexer

Multiplexer:
(4-input)

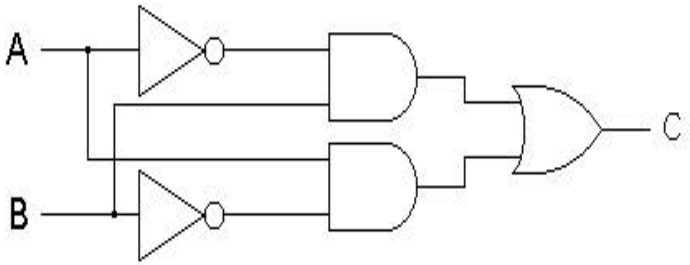


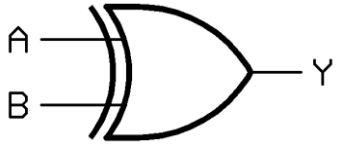


Input S2	Input S1	Output F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Review: Boolean Algebra & Gates

XOR Gate:

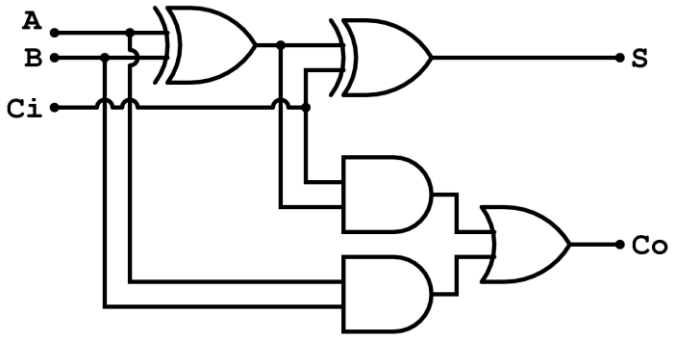


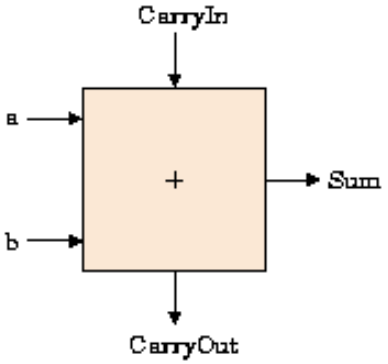


Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Review: Boolean Algebra & Gates

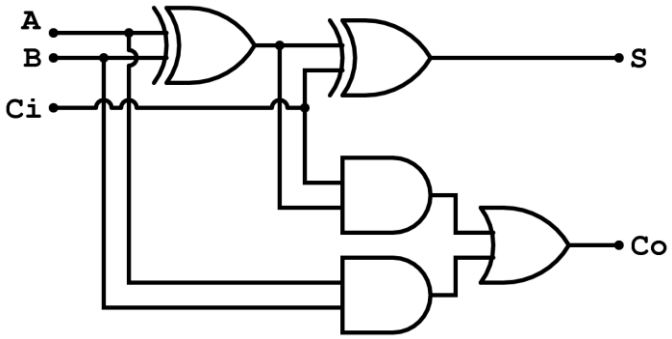
Adder:

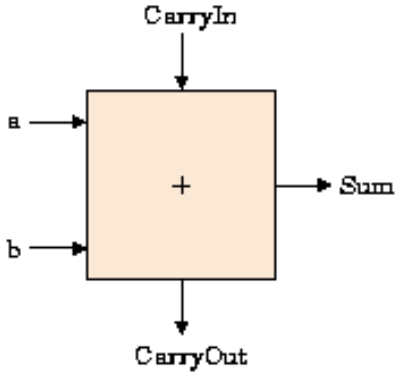




A	B	Ci (C-in)	Co (C-out)	S (Sum)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A 1-bit adder



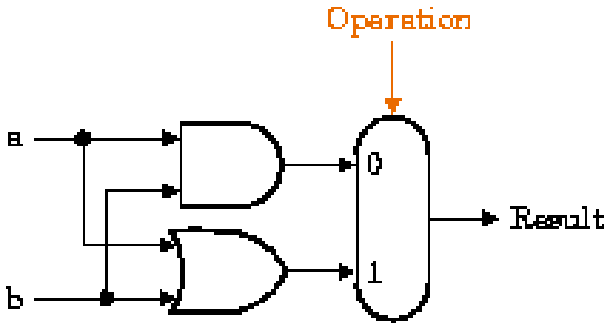


Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0+0+0=00_{two}$
0	0	1	0	1	$0+0+1=01_{two}$
0	1	0	0	1	$0+1+0=01_{two}$
0	1	1	1	0	$0+1+1=10_{two}$
1	0	0	0	1	$1+0+0=01_{two}$
1	0	1	1	0	$1+0+1=10_{two}$
1	1	0	1	0	$1+1+0=10_{two}$
1	1	1	1	1	$1+1+1=11_{two}$

$$c_{out} = a\ b + c_{in}\ (a \text{ xor } b)$$
$$= a\ b + a\ c_{in} + b\ c_{in}$$

$$sum = a \text{ xor } b \text{ xor } c_{in}$$

The 1-bit logical unit for AND and OR



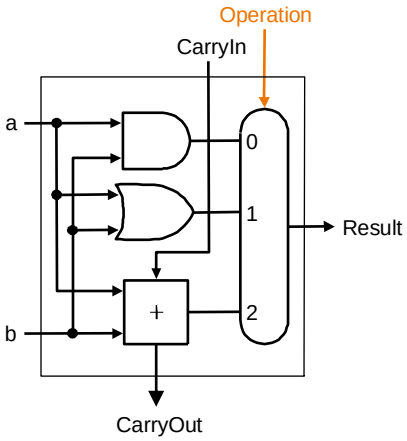
a	b	op	result
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1

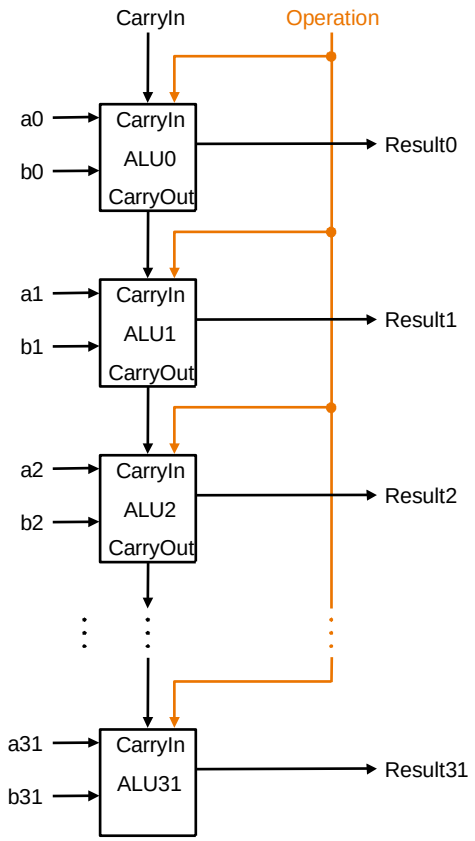
AND

a	b	op	result
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

OR

Building a 32 bit ALU(Arithmetic Logic Unit)

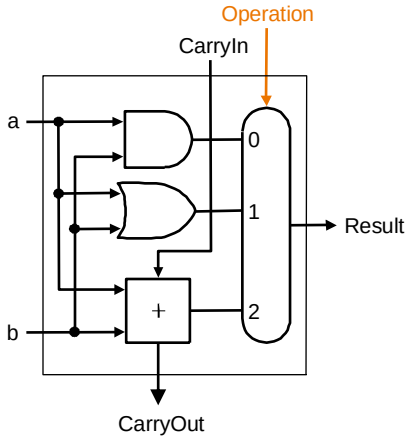


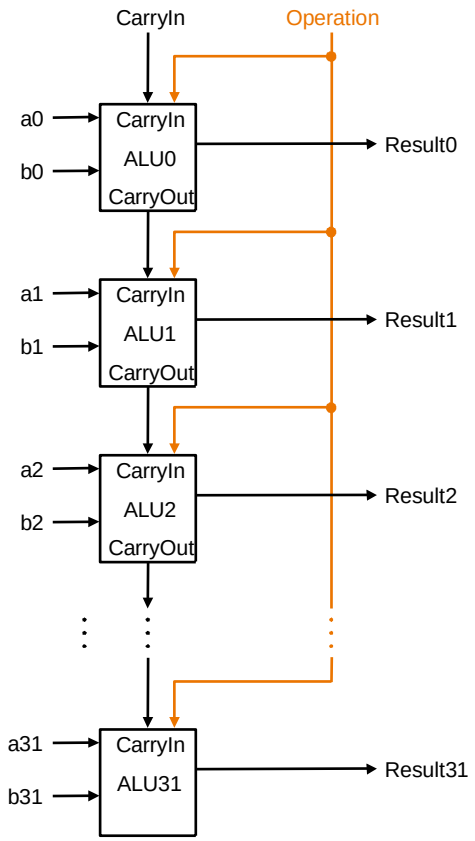


1-bit ALU **32-bit ALU**

Building a 32 bit ALU(and, or, add)

Operation = 00 (and)
01 (or)
10 (add)
CarryIn = 0





1-bit ALU **32-bit ALU**

What about subtraction (a – b) ?

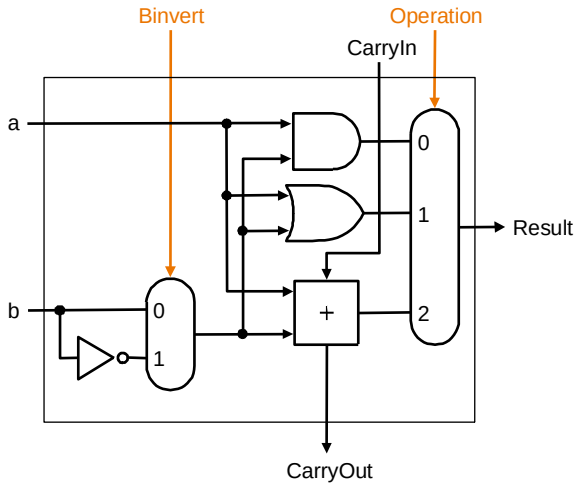
- **Two's complement approach: just negate b and add 1.**

  a-b = a+(-b) = a+(2's complement of -b) = a+(invert(b)+1)

- **How do we negate?**

- **A very clever solution:**

Operation = 10
Binvert = 1
CarryIn = 1



Tailoring the ALU to the MIPS

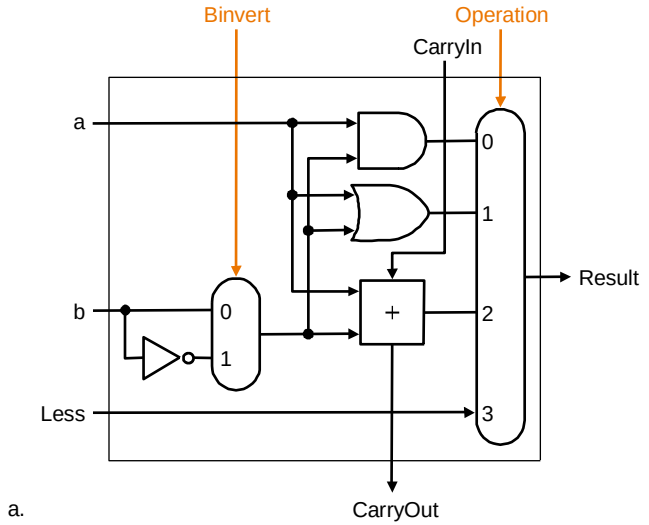
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - slt $t0, $t1, $t2 => use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq/bne)
  - beq $t1, $t2, label => use subtraction: (a-b) = 0 implies a = b
  - bne $t1, $t2, label => use subtraction: (a-b) != 0 implies a != b

  (note: bne is the opposite of beq)

Supporting slt

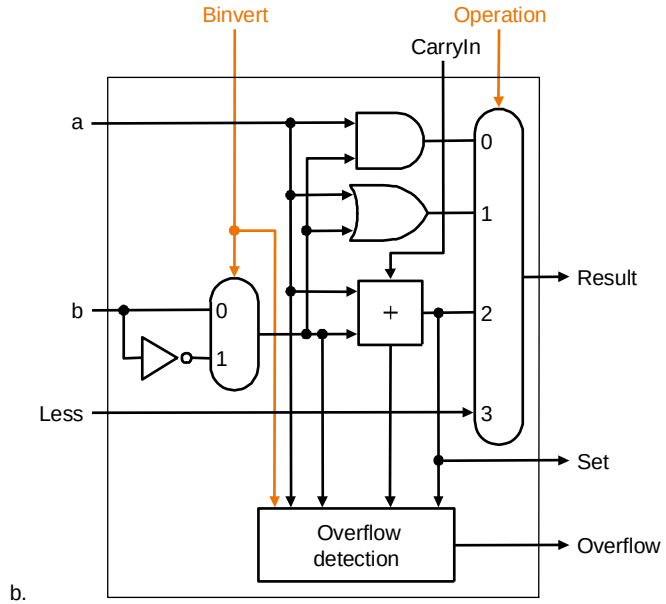
- Can we figure out the idea?

1-bit ALU for bit 0 to bit 30



a.

1-bit ALU for bit 31

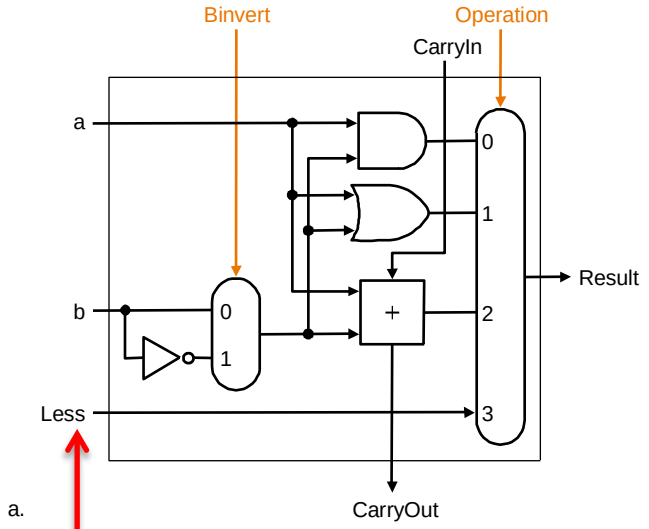


b.

Supporting slt

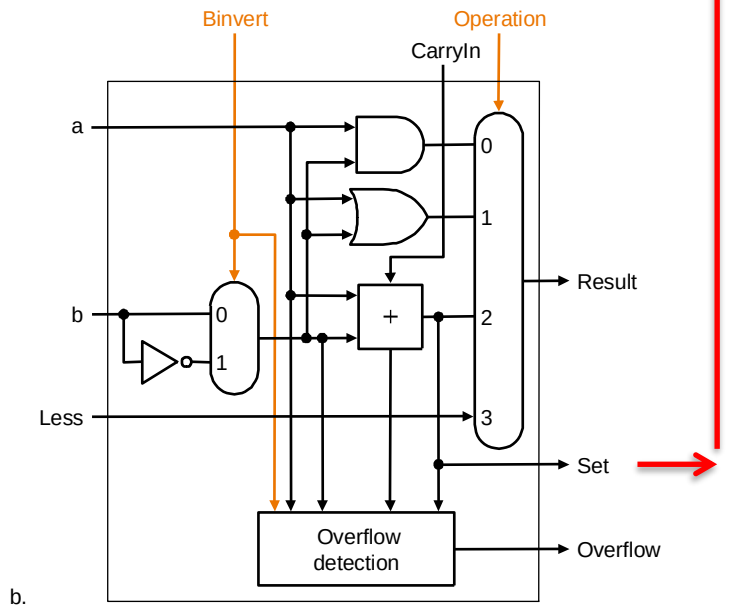
- Can we figure out the idea?

1-bit ALU for bit 0



Operation = 11
Binvert = 1
CarryIn (bit 0) = 1

1-bit ALU for bit 31



Overflow Detection

- For A+B (add): A'B'R+ABR' (see slide #12)
- For A-B (sub): A'BR+AB'R' (see slide #12)

- How do we know if it's "add" or "sub"?

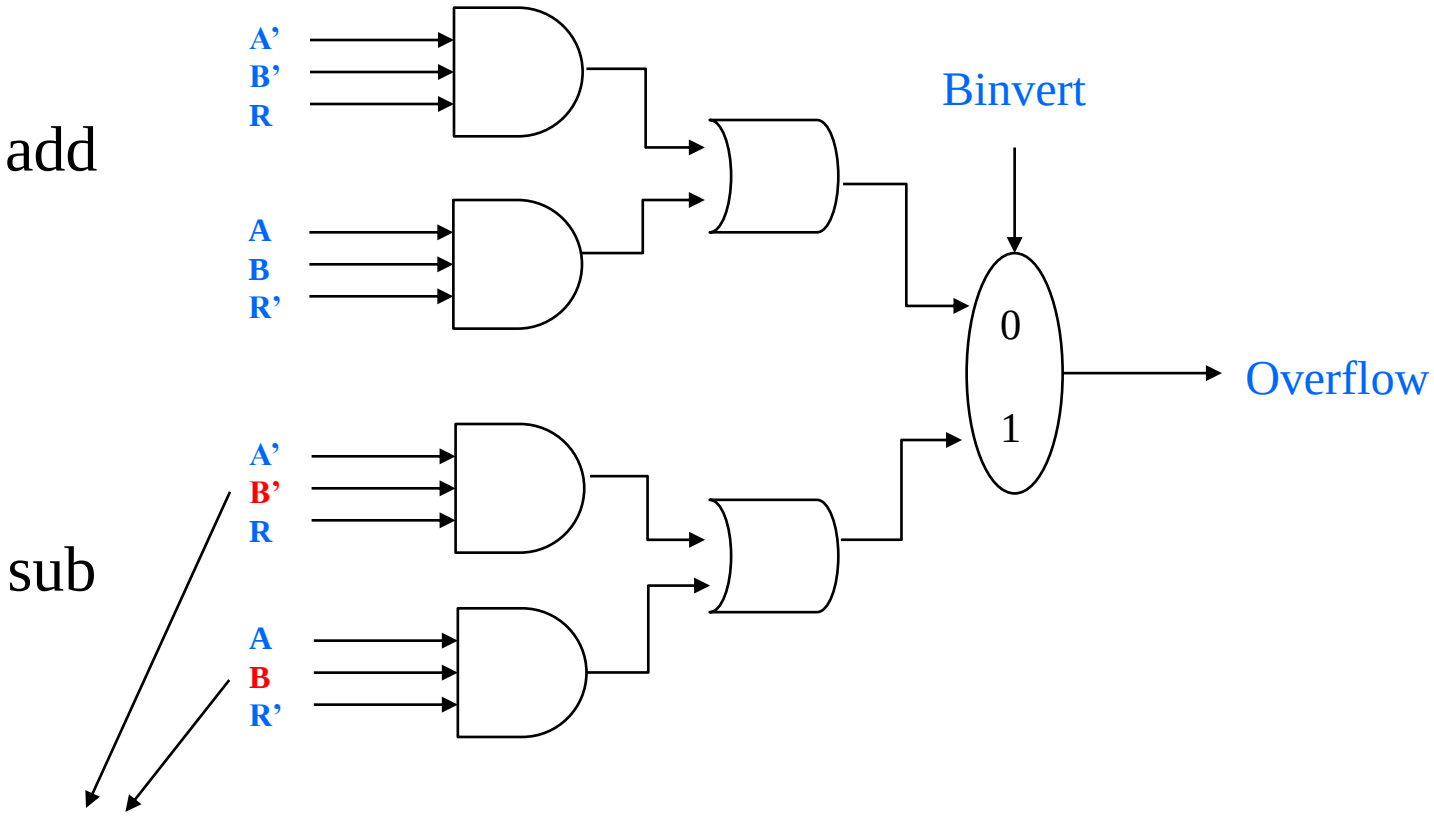
  Binvert = 0 (add)

  Binvert = 1 (sub)

- Can you draw this Overflow Detection logic circuit?

Overflow Detection

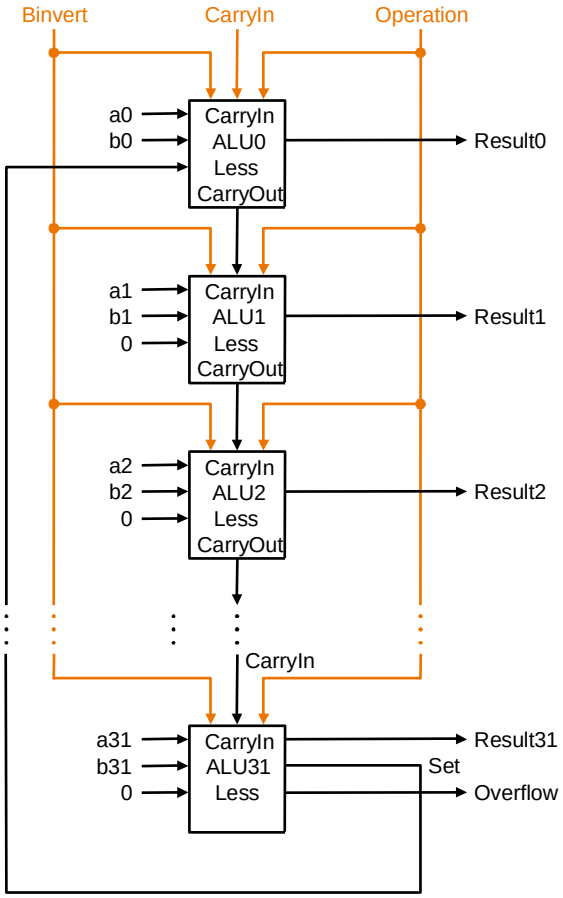
- **Overflow Detection logic circuit**



*For sub: Binvert = 1 means "not B" has already been set

32-bit MIPS ALU

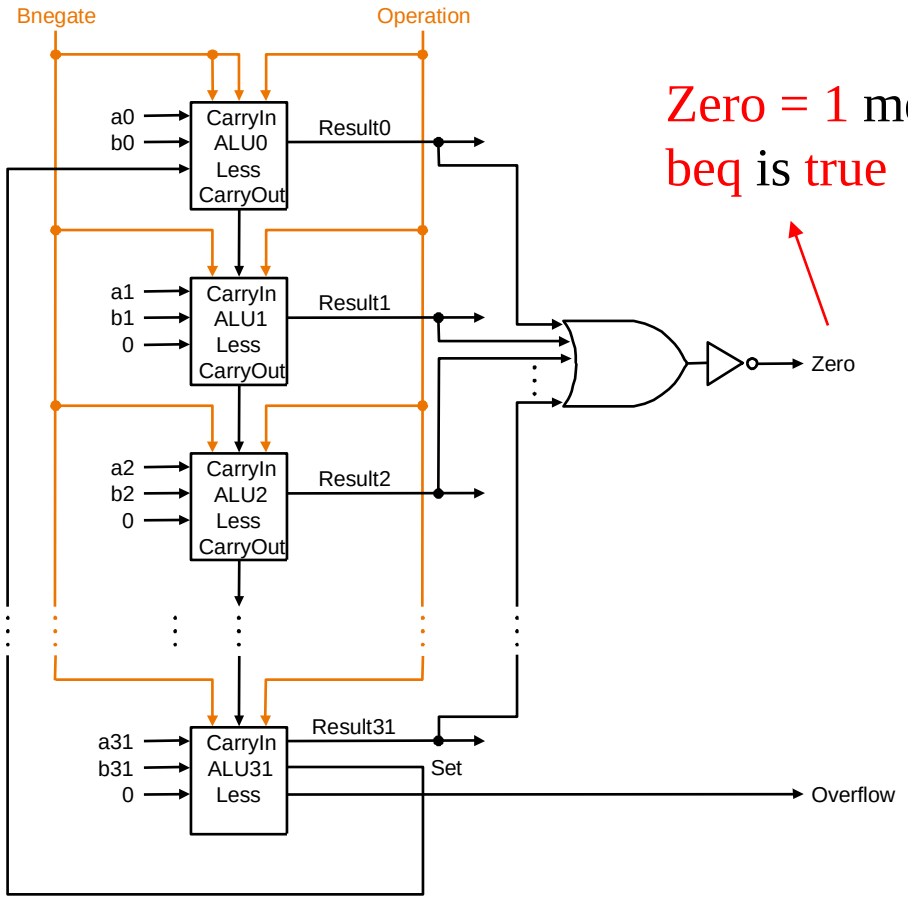
Since Binvert and CarryIn have the same signs, we can combine them together (Bnegate – see next slide).



Supporting beq/bne

***Just use sub:**

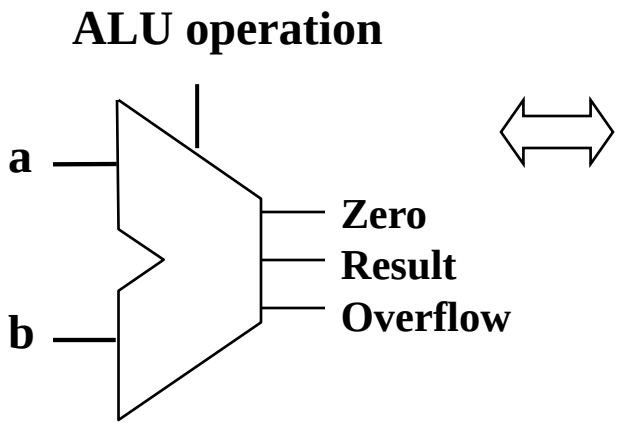
Operation = 10
Bnegate = 1

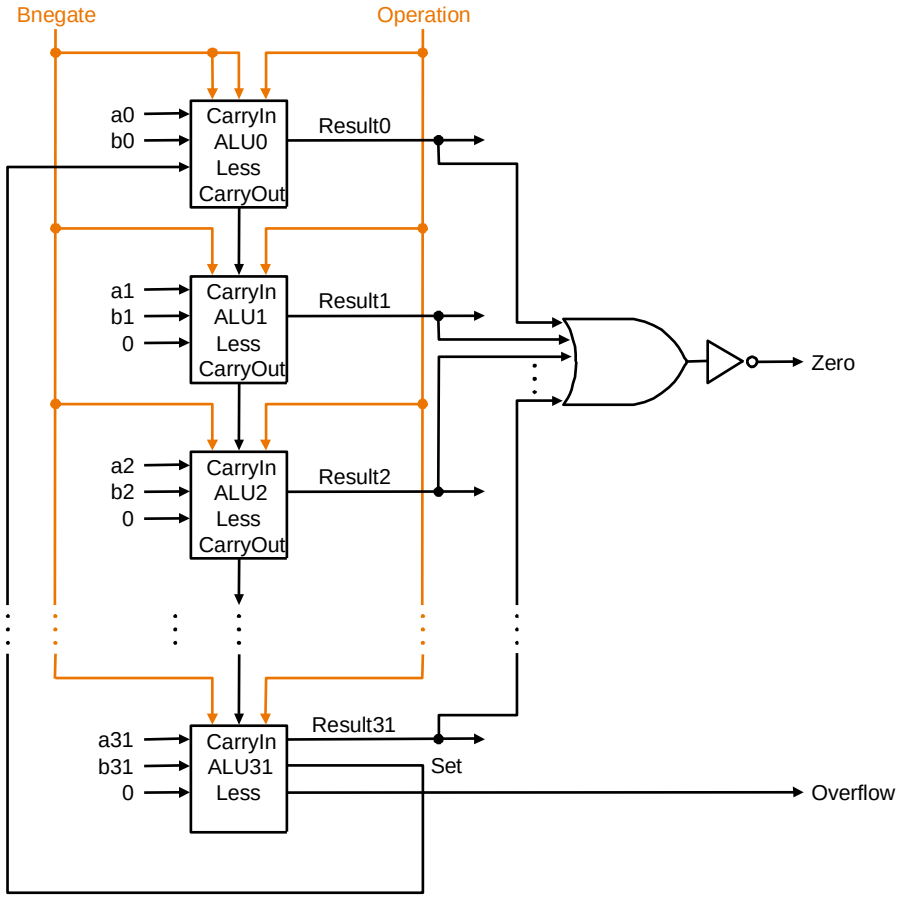


32-bit MIPS ALU

- **ALU control lines:**

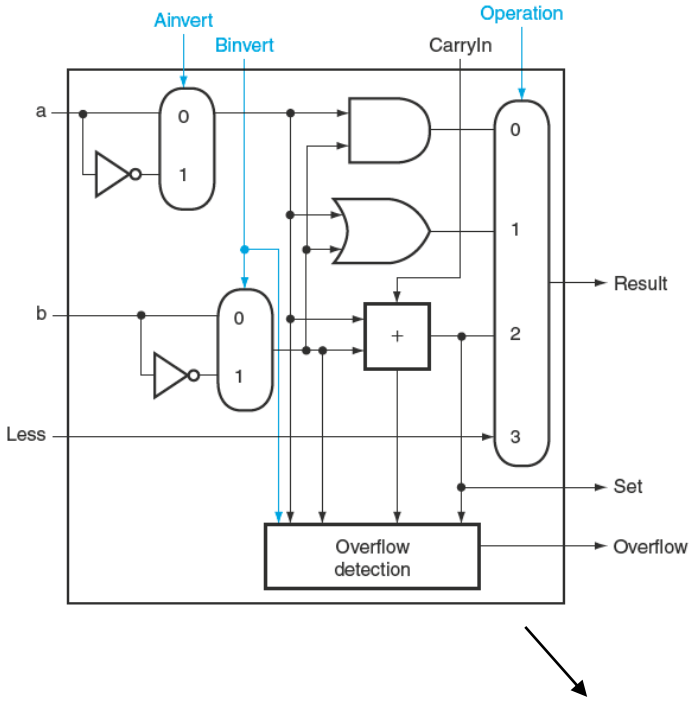
```
000 = and
001 = or
010 = add
110 = subtract(beq/bne)
111 = slt
```



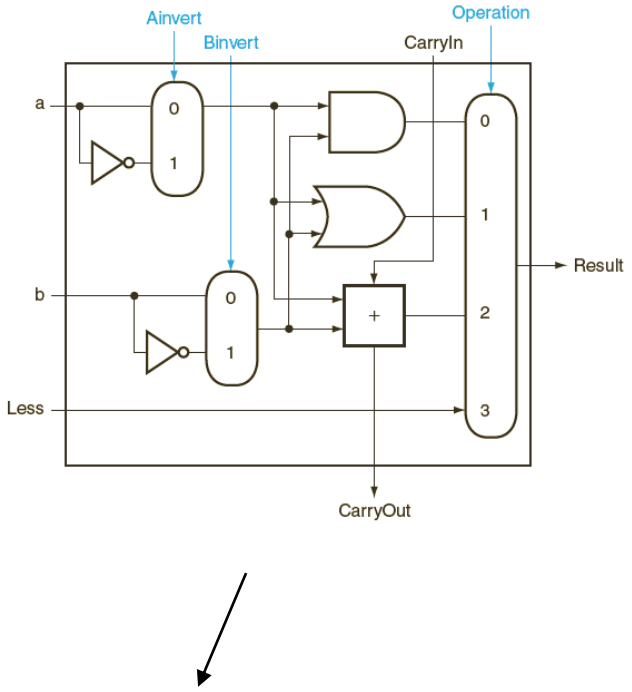


What about (a nor b) ?

- To do: NOT $t0 => use: nor $t0, $t0, $zero
- (a nor b) = not (a or b) = (not a) and (not b) ---> DeMorgan's law



bit 31

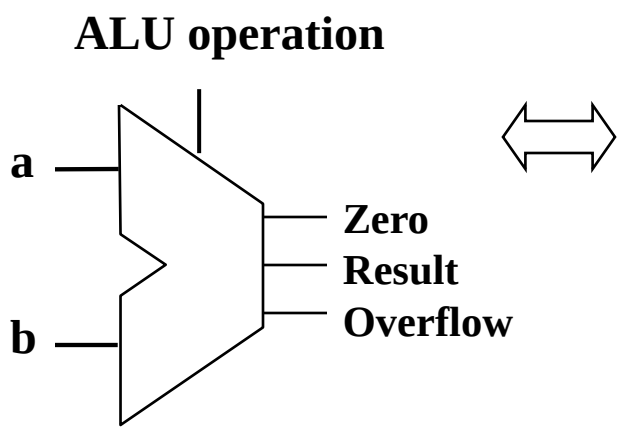


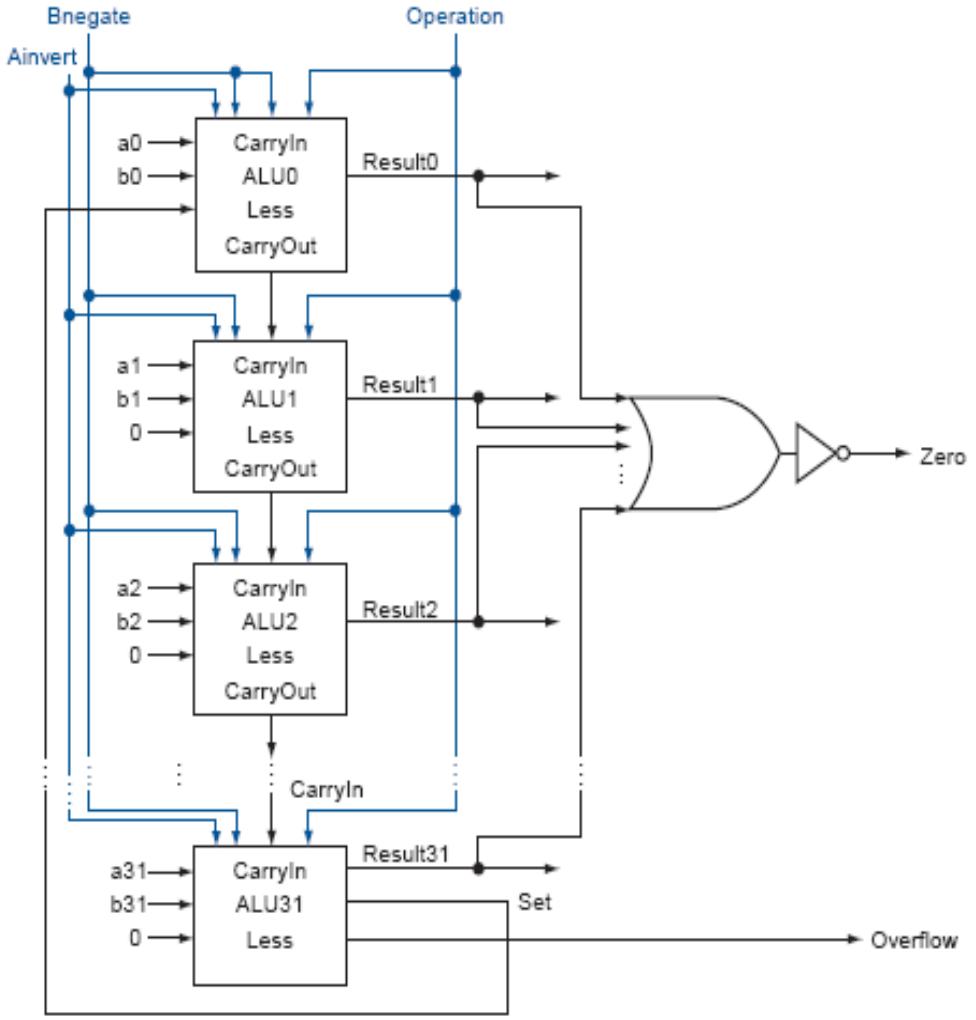
bit 0 to bit 30

Final Version: 32-bit MIPS ALU

- **ALU control lines:**

```
0000 = and
0001 = or
0010 = add
0110 = subtract(beq/bne)
0111 = slt
1100 = nor
```





Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication