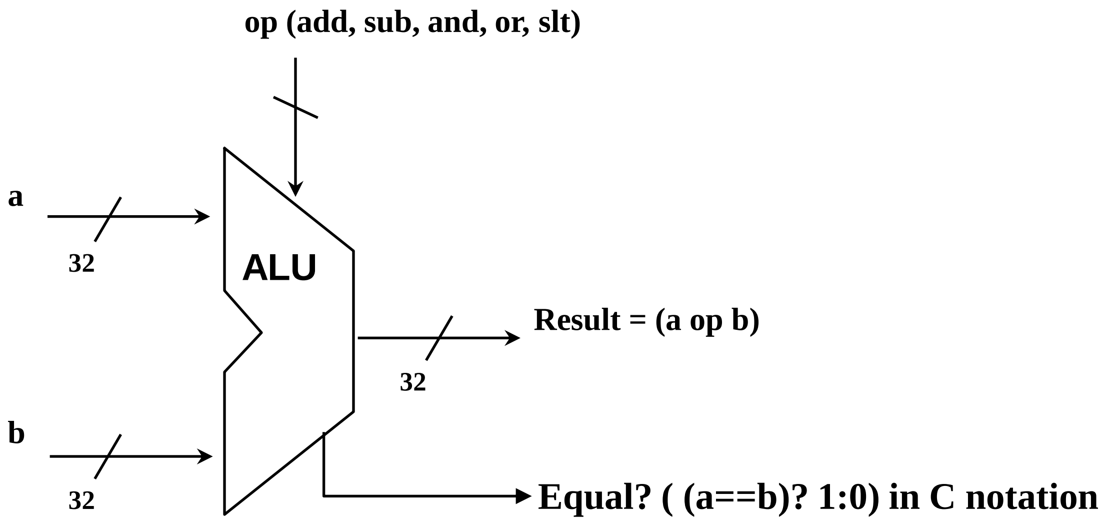
Chapter 4

Building a simple ALU to support MIPS arithmetic instructions

Extra material (Appendix C.5 under Oncourse:resources)

Introduction

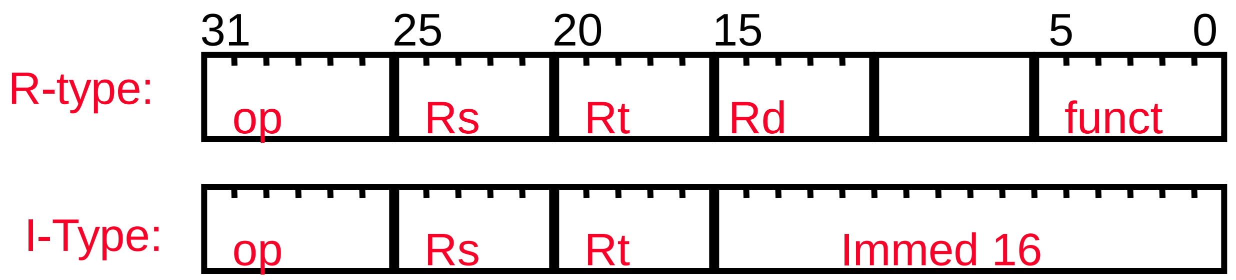
- What's up ahead:
  - Implementing the Processor Architecture
  - Design simple ALU to support MIPS arithmetic/logic instructions


op (add, sub, and, or, slt)
a
32
ALU
Result = (a op b)
32
b
32
Equal? ((a==b)? 1:0) in C notation

MIPS ALU requirements

- add, addu, sub,   subu, addi, addiu
  - 2's complement adder/sub with overflow detection
- and, or, andi, ori, xor, xori, nor
  - Logical and, logical or, xor, nor
- slt, sltu, slti, sltiu (set less than)
  - ➔ 2's complement adder with inverter, check sign bit of result
- ALU we will design supports these

MIPS arithmetic instruction format



Type	op	funct
addi	10	xx
addiu	11	xx
slti	12	xx
sltiu	13	xx
andi	14	xx
ori	15	xx
xori	16	xx
lui	17	xx

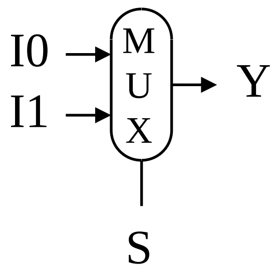
Type	op	funct
add	00	40
addu	00	41
sub	00	42
subu	00	43
and	00	44
or	00	45
xor	00	46
nor	00	47

Type	op	funct
	00	50
	00	51
slt	00	52
sltu	00	53

Octal values

Design trick

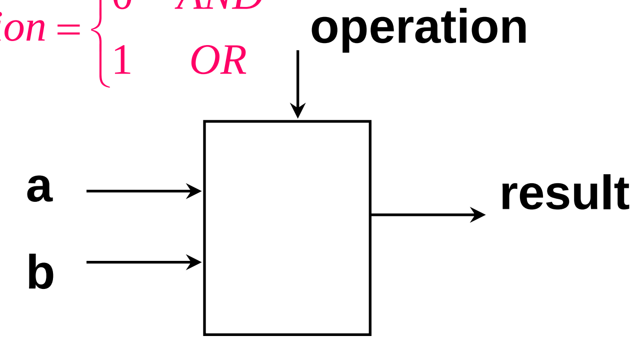
- take pieces you know and try to put them together.
  - We will design the processor one bit at a time.
  - We will design the modules of the processor and put them together.
  - Multiplexer: one component used to put modules together
- 2-to-1 Multiplexor (MUX)
  - Y = S ? I1 : I0



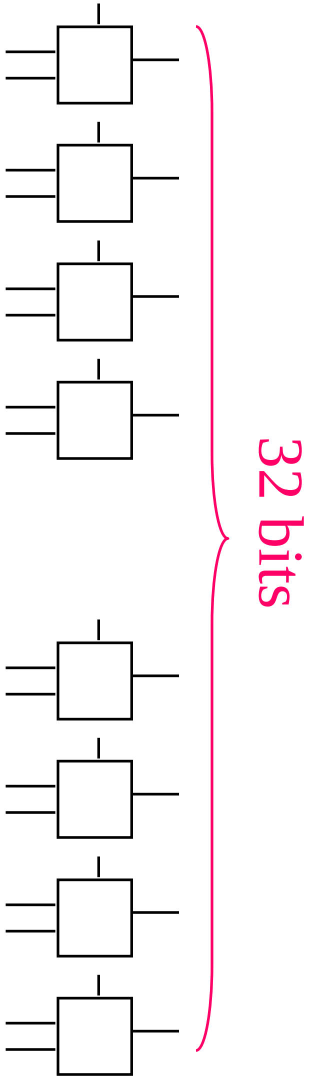
An ALU (arithmetic logic unit)

- Let's build an ALU to support the `and` and `or` instructions
  - we'll just build a 1 bit ALU, and use 32 of them

$$operation = \begin{cases} 0 & AND \\ 1 & OR \end{cases}$$



	op	a	b	res
AND	0	0	0	0
	0	0	1	0
	0	1	0	0
	0	1	1	1
OR	1	0	0	0
	1	0	1	1
	1	1	0	1
	1	1	1	1

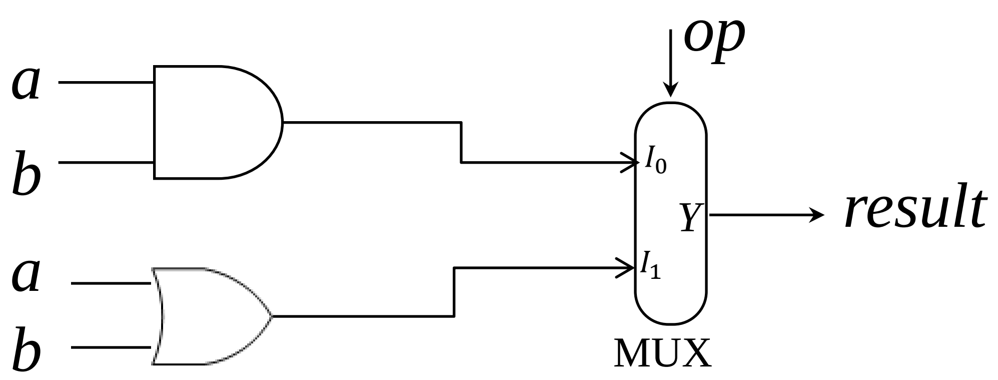


- Possible Implementation (sum-of-products):

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

Op would be the select (S) bit for MUX

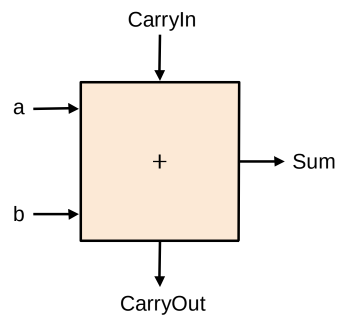
Partial ALU supporting and and or



op	$result$
0	$I_0 = a \cdot b$
1	$I_1 = a + b$

Different Implementations

- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



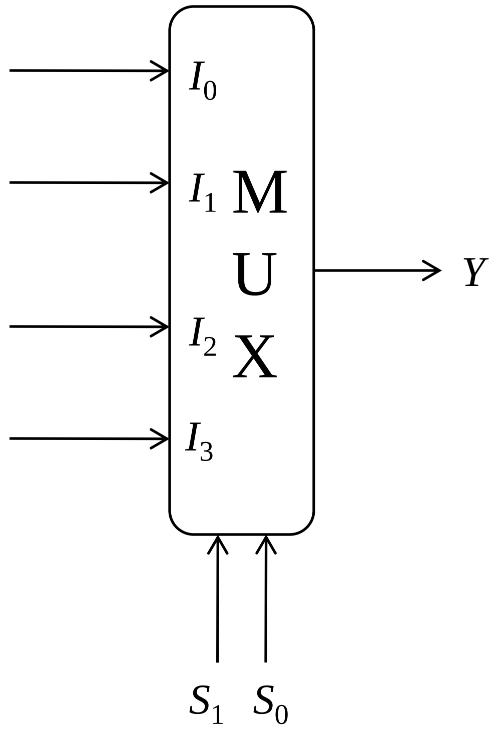
$$c_{out} = a\,b + a\,c_{in} + b\,c_{in}$$
$$sum = a \oplus b \oplus c_{in}$$

Full Adder

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Larger multiplexors

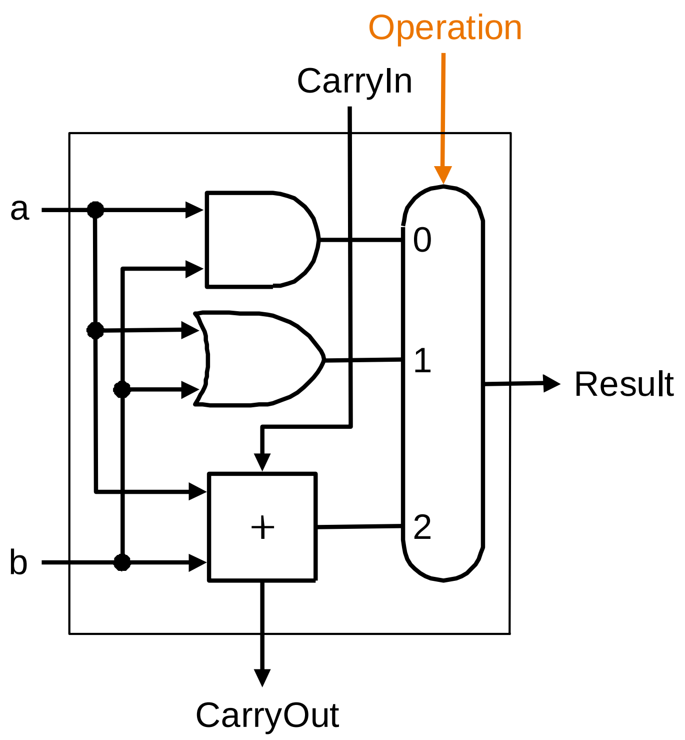
- **4-to-1 MUX : 2 select bits; selects one of 4 inputs**



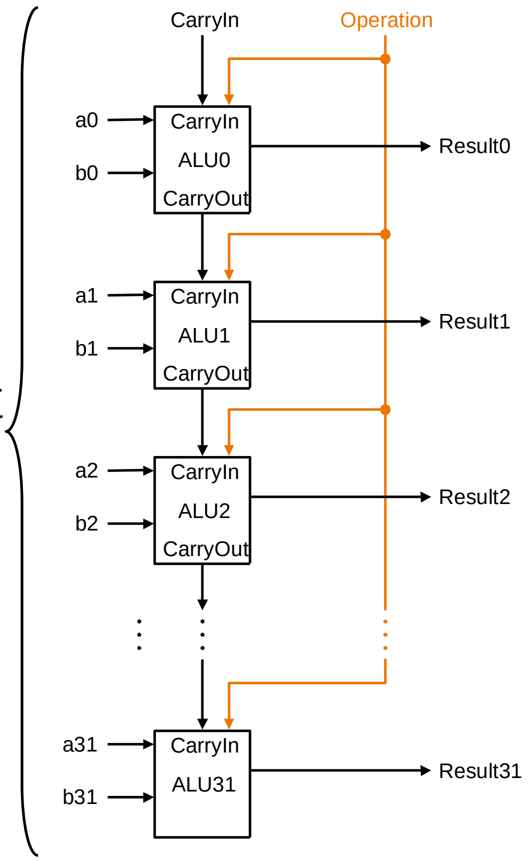
$S_1\ S_0$	Y
00	I_0
01	I_1
10	I_2
11	I_3

Building a 32 bit ALU

Opcode = 0 ➔ AND
Opcode = 1 ➔ OR
Opcode = 2 ➔ ADD



32-bit ALU



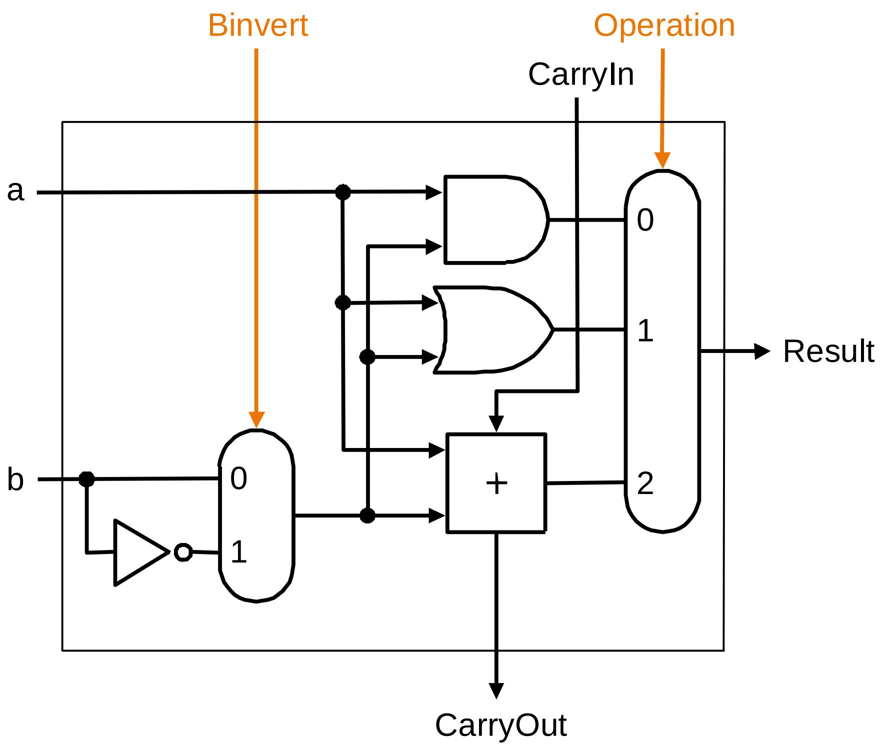
What about subtraction (a – b) ?

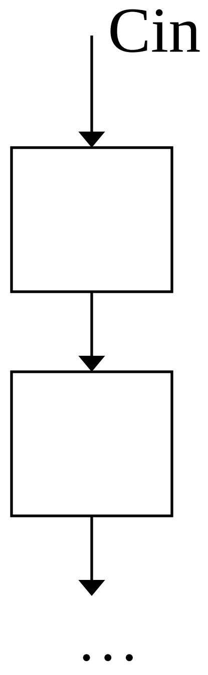
- **Two's complement approch: just negate b and add.**
- **How do we negate (not invert)?**

Carry-in = Binvert for LSB

Call it bnegate (1 control bit)

- **A very clever solution:**



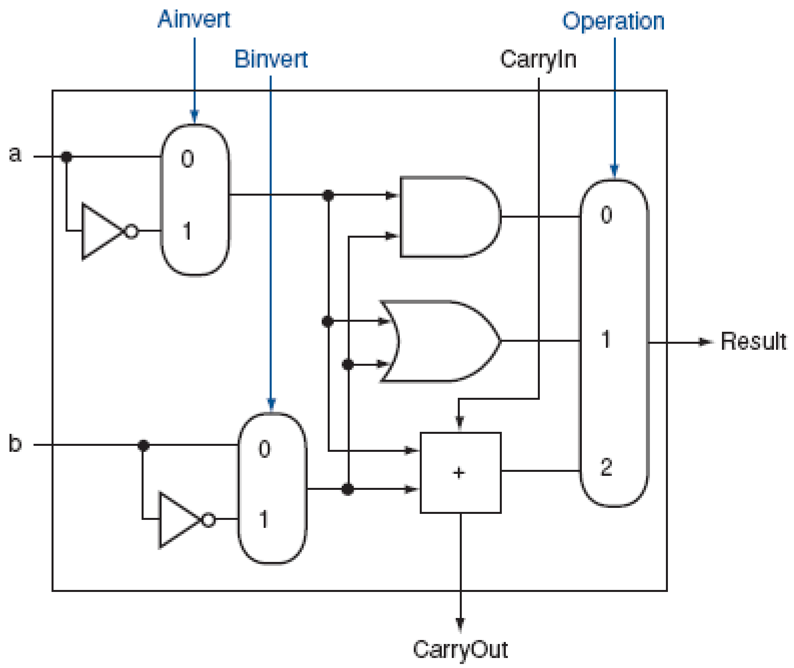


NOR operation

- **We can add NOR operation by using DeMorgan's law:**

$$(a+b)' = a' \cdot b'$$

- **We can easily add this using the existing AND gate with inverters to the inputs:**



To implement NOR
Ainvert=1
Binvert=1
Op=0 (and)

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**
  - **use subtraction: (a-b) < 0 implies a < b**
- **Need to support test for equality (beq $t5, $t6, $t7)**
  - **use subtraction: (a-b) = 0 implies a = b**

Supporting slt

- **Can we figure out the idea?**

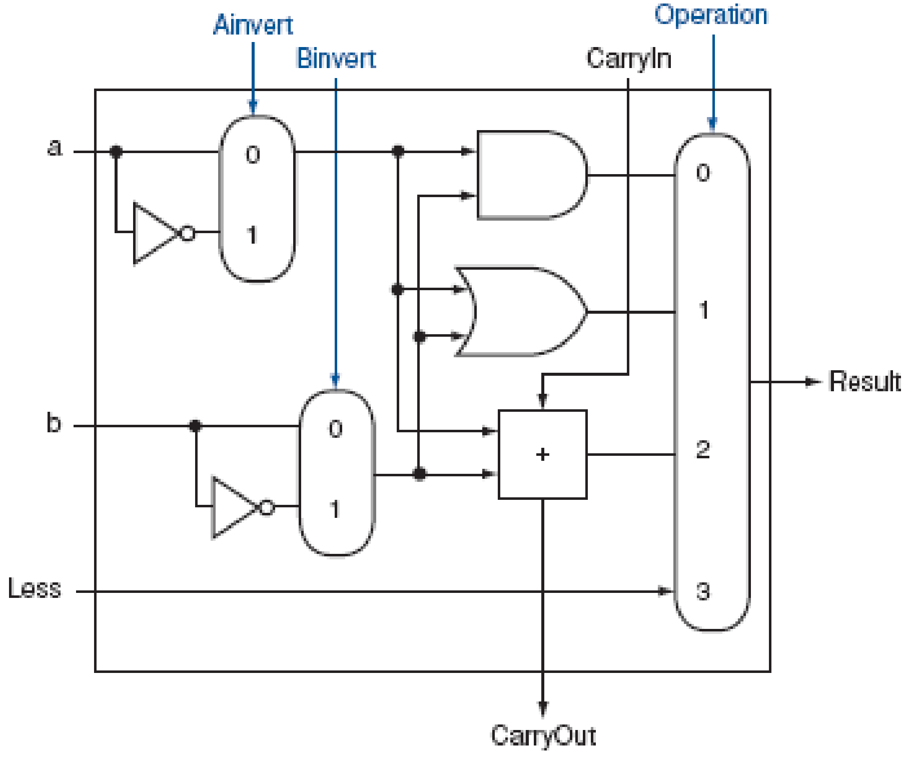
LSB of the word is set to 1 if less=1; all the other bits are always set to 0   Op for slt=3

$$Less = \begin{cases} 1 & \text{if } a < b \\ 0 & \text{if } a \geq b \end{cases}$$

$$= \begin{cases} 1 & \text{if } (a\text{-}b) < 0 \\ 0 & \text{if } (a\text{-}b) \geq 0 \end{cases}$$

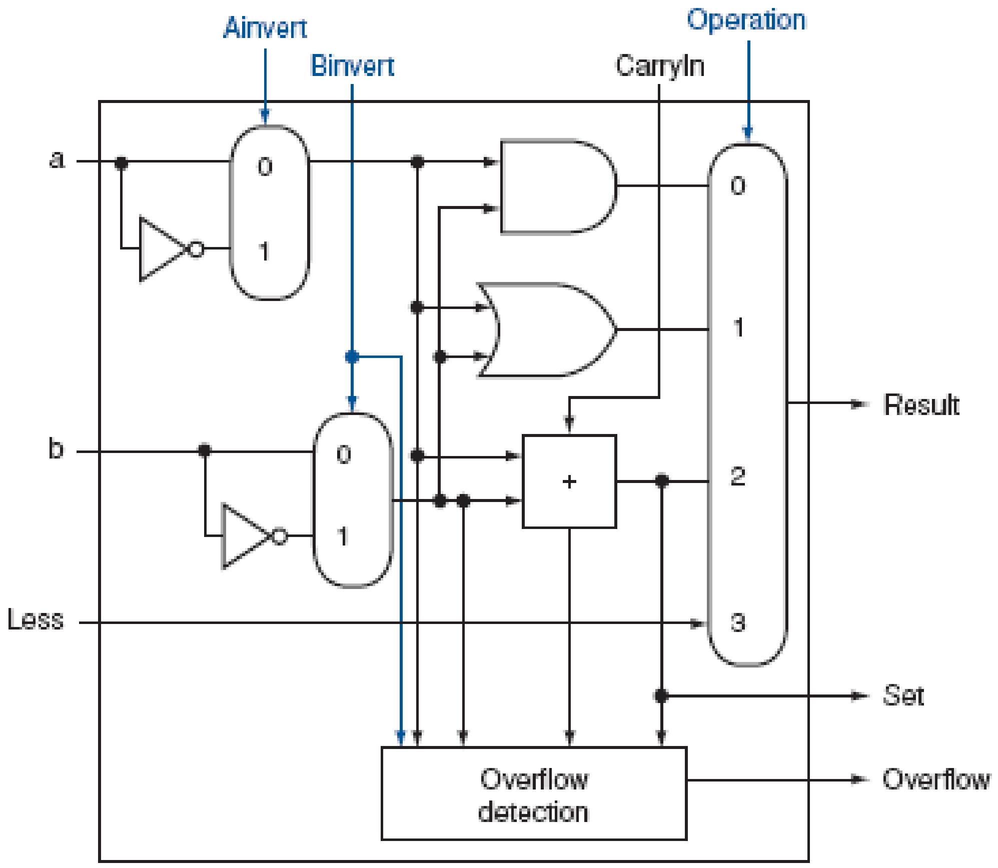
↑

Sign bit



Supporting overflow detection

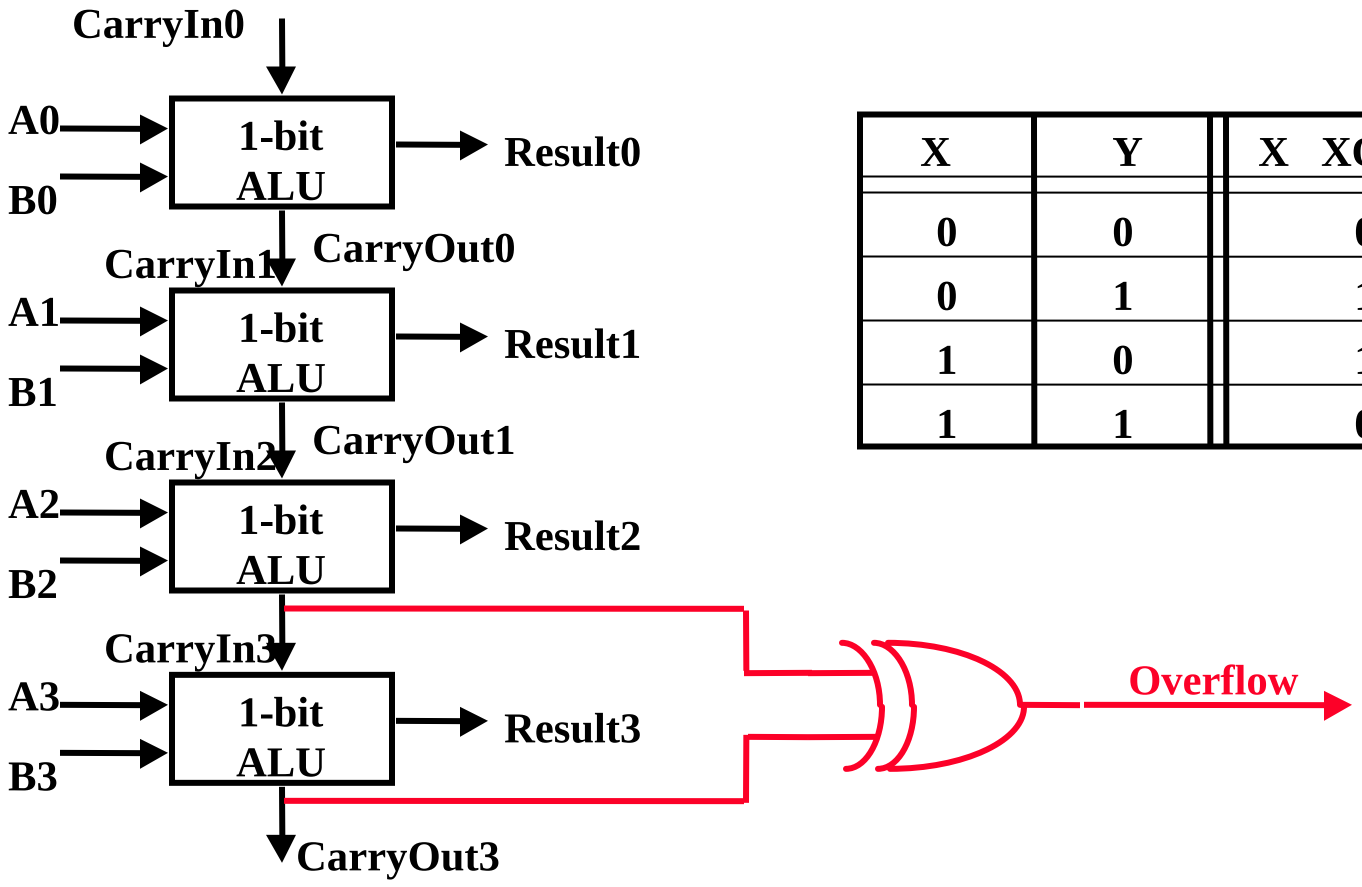
- **Since we need a special ALU hardware for the MSB, we might as well add the overflow logic.**



Overflow circuitry
is done only for MSB

Overflow Detection Logic

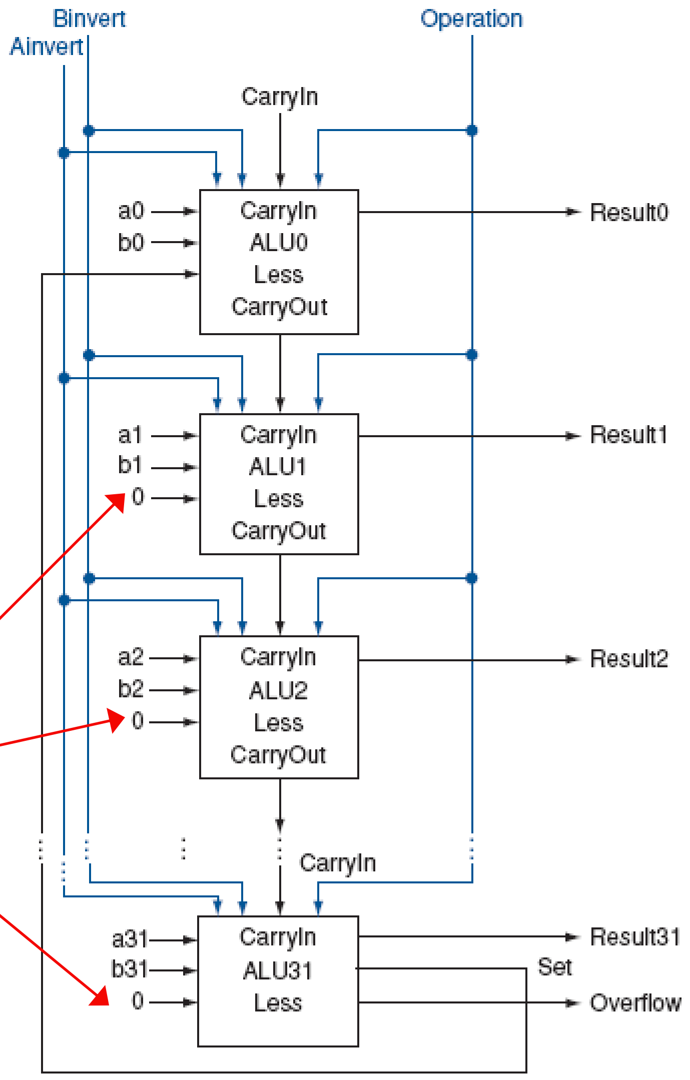
- Remember: Overflow = Carry into MSB ⊕ Carry out of MSB
  - For a n-bit ALU: Overflow = CarryIn[n - 1] XOR CarryOut[n - 1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Supporting slt

LSB bit gets fed from the Set result of the MSB which is the sign bit of (a-b)



All less bits are always = 0 except LSB

ALU hardware for bit 32 is special that includes overflow detection

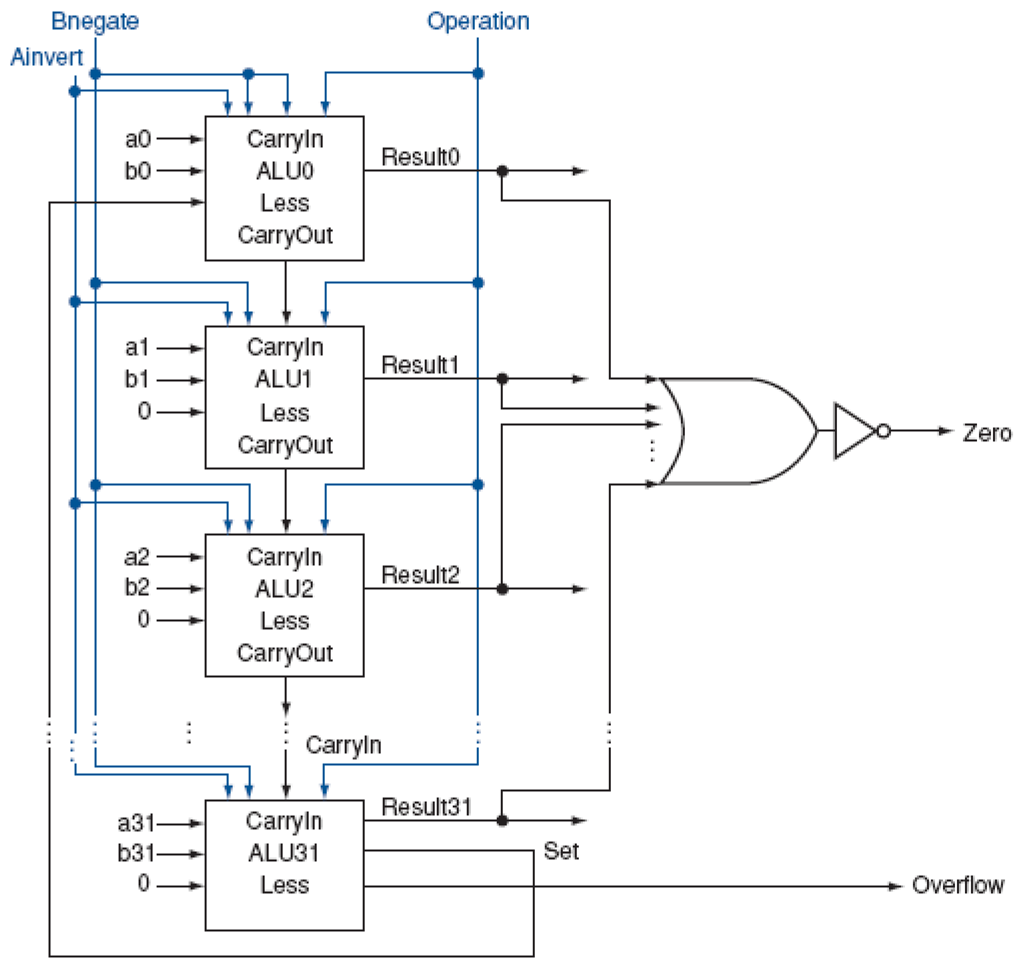
Test for equality

- **Notice control lines:**

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = nor
```

•*Note: zero is a 1 when the result is zero!*
That is, all 32 bits are=0

Bit 1 = ainvert
Bit 2 = bnegate
Bits 3,4= op



ALU in textbook: extended to include nor

- **4-bit op to ALU instead of 3 bits.**
- **ALU used for**
  - **Load/Store: F = add**
  - **Branch: F = subtract**
  - **R-type: F depends on funct field**

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

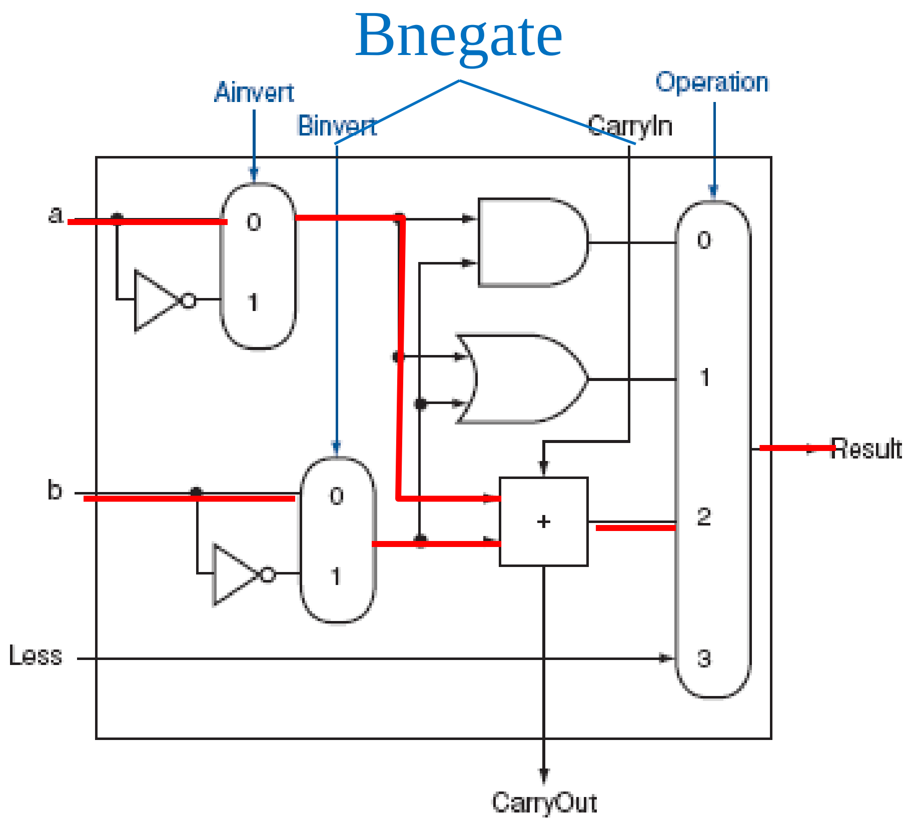
Example settings

- **Control bits during add**

  Ainvert=0

  Bnegate=Binvert=Carry

  Operation=10



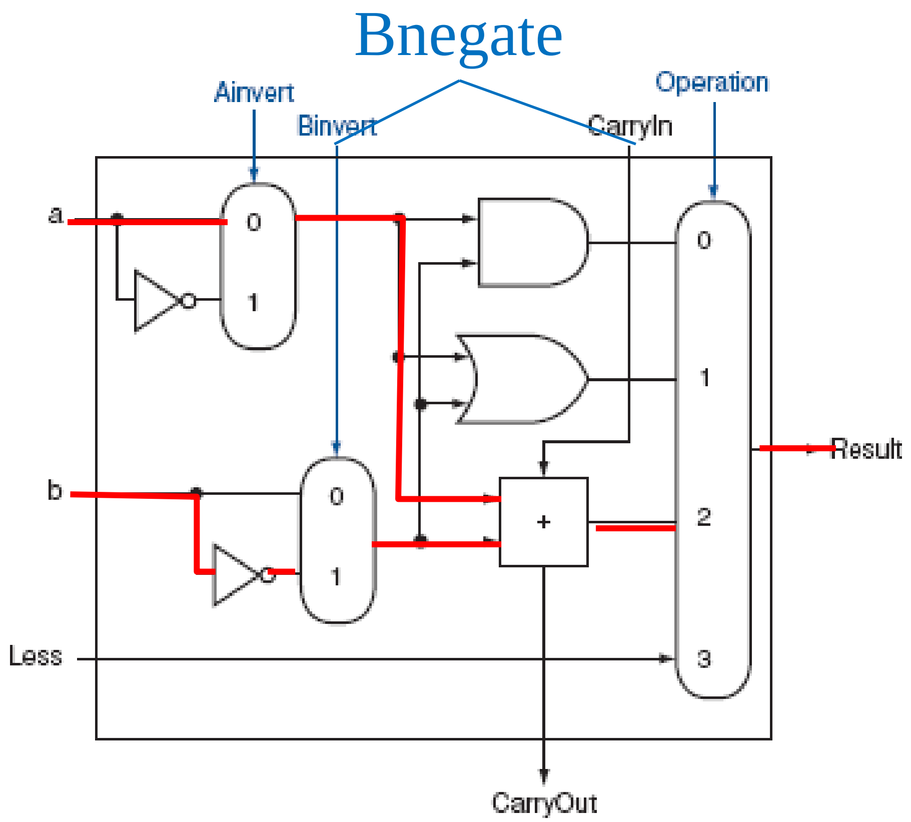
Example settings

- **Control bits during subtract**

  Ainvert=0

  Bnegate=Binvert=Carry

  Operation=10



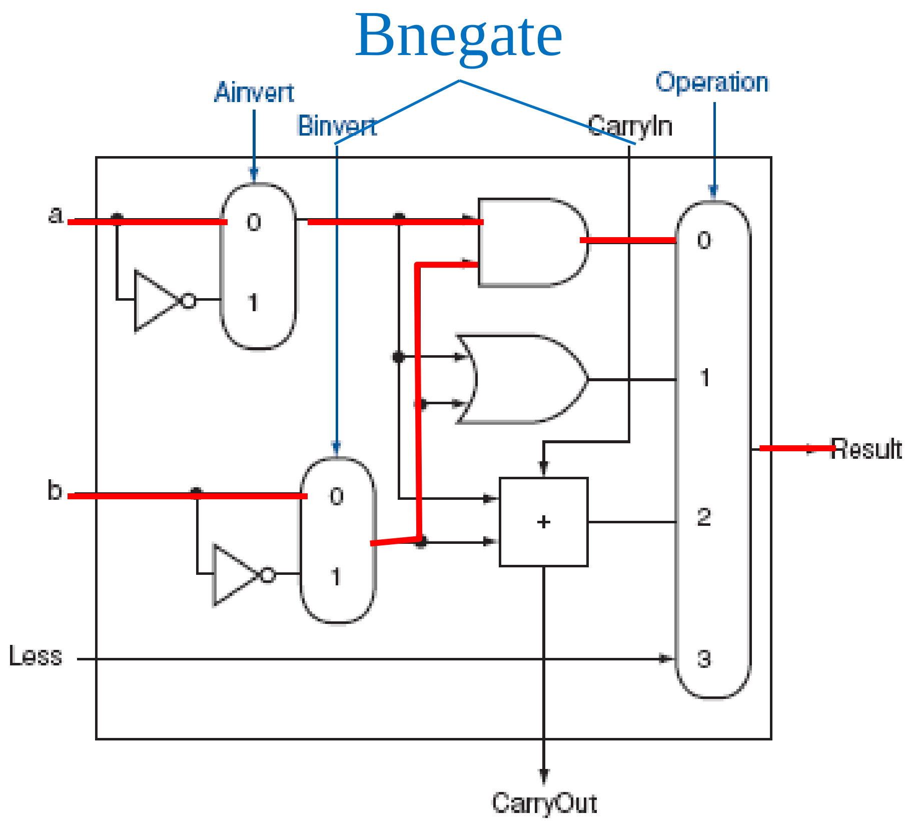
Example settings

- **Control bits during AND**

  Ainvert=0

  Bnegate=Binvert=Carry

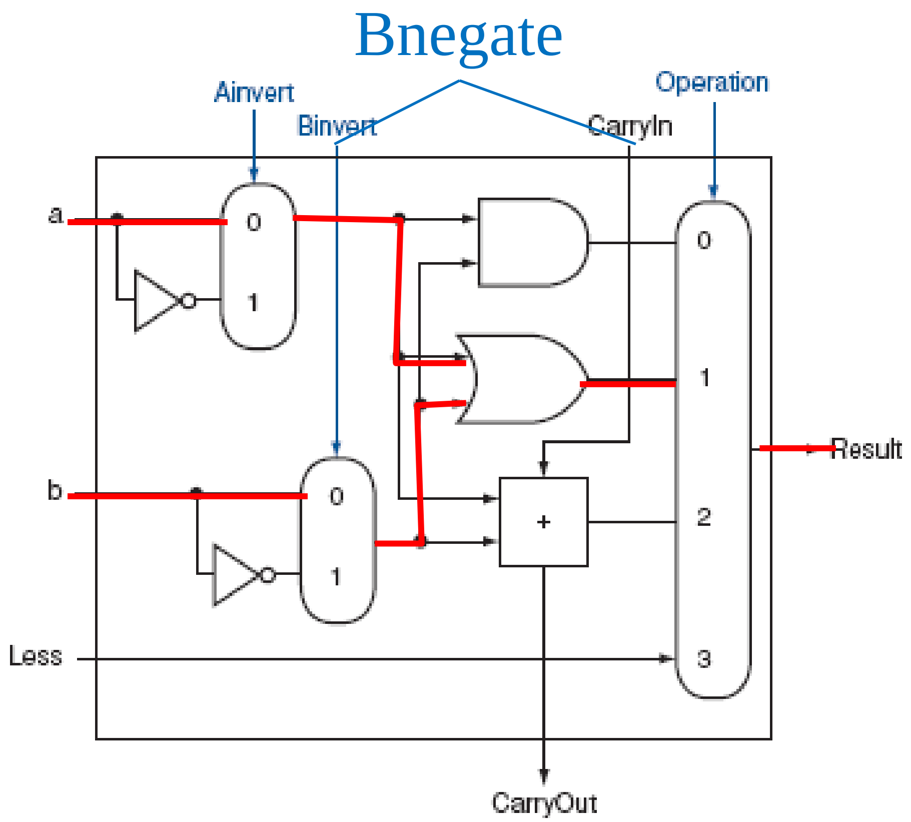
  Operation=00



Example settings

- **Control bits during OR**

  Ainvert=0
  Bnegate=Binvert=Carry
  Operation=01



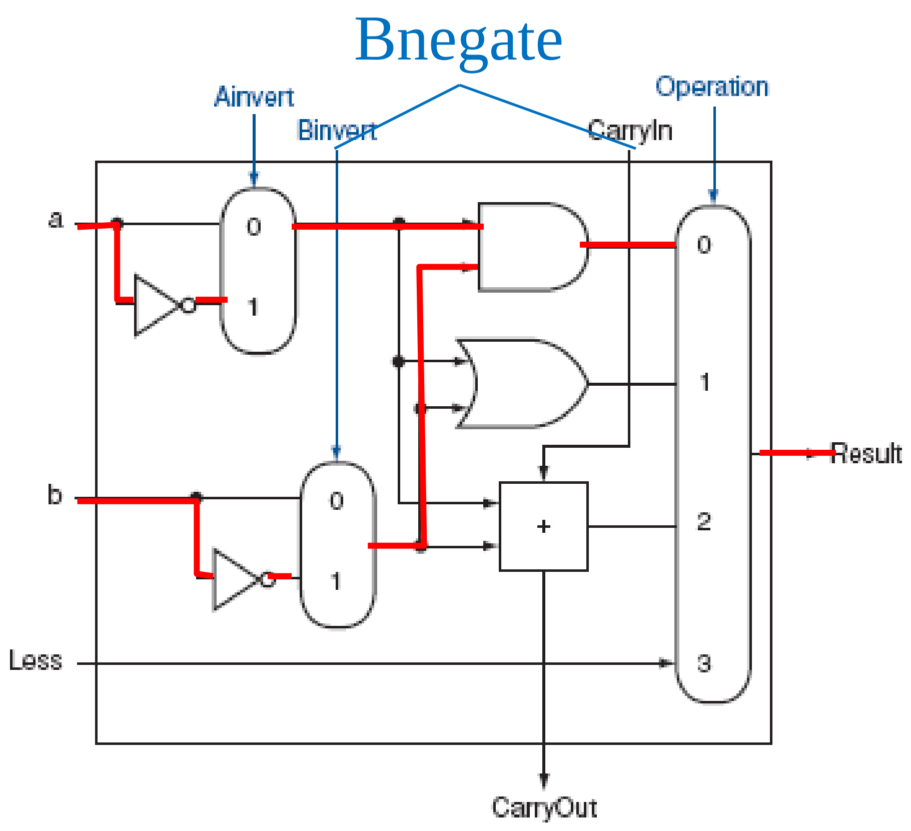
Example settings

- **Control bits during NOR**

  Ainvert=1

  Bnegate=Binvert=Carry

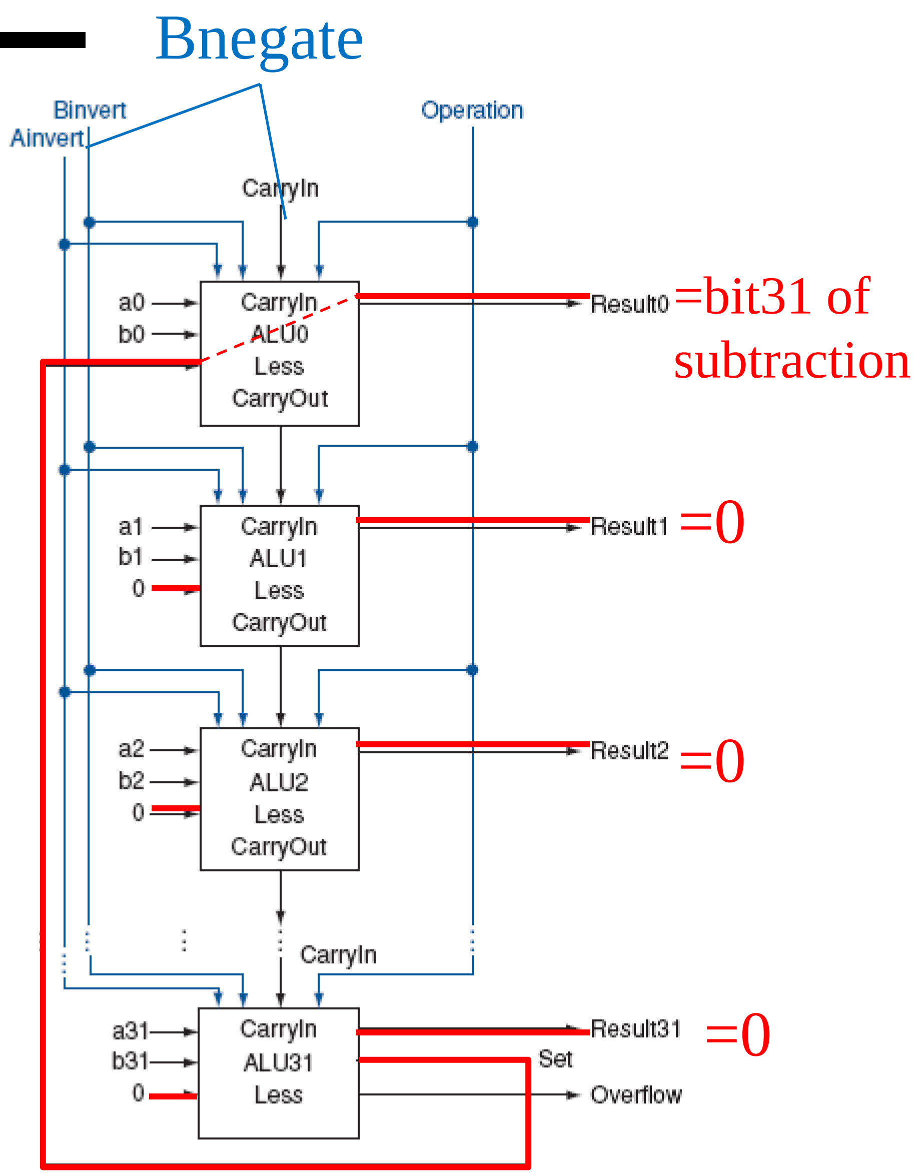
  Operation=00



Example settings

- **Control bits during slt**

  Ainvert=0 ← subtract
  Bnegate=CarryIn=Binvert=1
  Operation=11 } Select input 3 of mux



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):
    - E.g.., Booth's algorithm for multiplication