CSCI 210: Computer Architecture
Lecture 17: Arithmetic Logic Unit

Stephen Checkoway

Slides from Cynthia Taylor

CS History: Mohamed M. Atalla

- Born in 1924 in Egypt
- Invented the MOSFET (metal-oxide semiconductor field-effect transistor) with Dawon Kahng in 1960
- First truly compact transistor
- MOS transistors are the fundamental building blocks of today's electronics
- Most manufactured device in history
  - 13 sextillion MOS transistors manufactured as of 2018
- Went on to start a cybersecurity company, invented the "Atalla box" which secured most ATMs in the past

Arithmetic and Logical Unit (ALU)

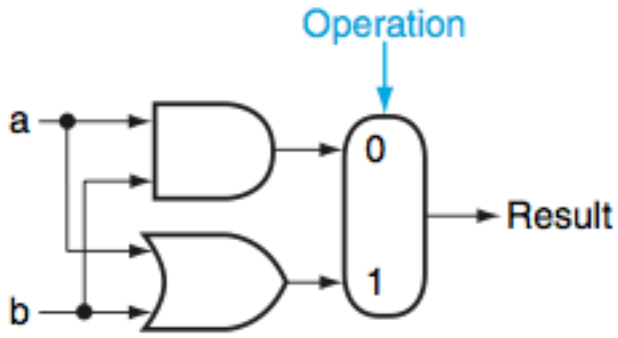
- Need to use digital logic to build a unit that can do basic computation – math, logical operations, etc.

- Needs to be 32 bits wide, since MIPS has 32 bit words.
  - Build out of 1-bit ALUs

Our ALU will support the following operations:

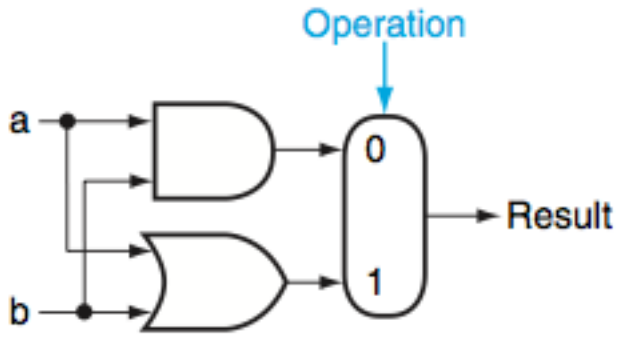
- Add
- Sub
- And
- Or
- Nor
- Nand
- Set less than

1-bit ALU: AND and OR



- Inputs go to both AND and OR
- Multiplexer selects AND or OR function for output

If a = 0, b = 1, and operation = 1, what is Result?



A. 0

B. 1

C. Impossible to say without additional information

1-bit Binary Addition

0 + 0 = 0

0 + 1 = 1

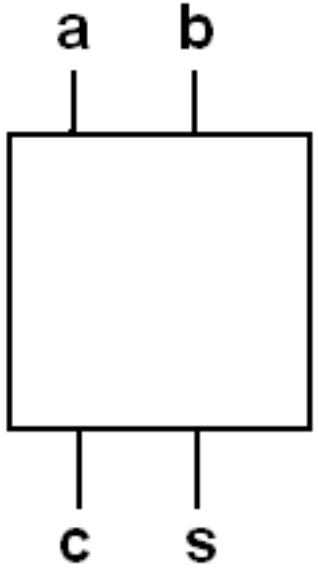
1 + 0 = 1

1 + 1 = 10

Need to account for two output bits!

Half Adder

- Inputs a, b
- Outputs sum and carry out
- Sum is the 1-bit result of adding a and b
- Carry out is the carry in the normal sense



Below is the truth table for the SUM output of a half adder. What is the Boolean algebra function that will give us this truth table?

a	b	Sum
0	0	0
0	1	1
1	0	1
1	1	0

A. a OR b

B. a XOR b

C. a AND b

D. a NOR b

E. None of the above

Below is the truth table for the CARRY output of a half adder. What is the Boolean algebra function that will give us this truth table?

a	b	Carry out
0	0	0
0	1	0
1	0	0
1	1	1

A. a OR b

B. a XOR b

C. a AND b

D. a NOR b

E. None of the above

Binary Addition with Arbitrary Number of Bits

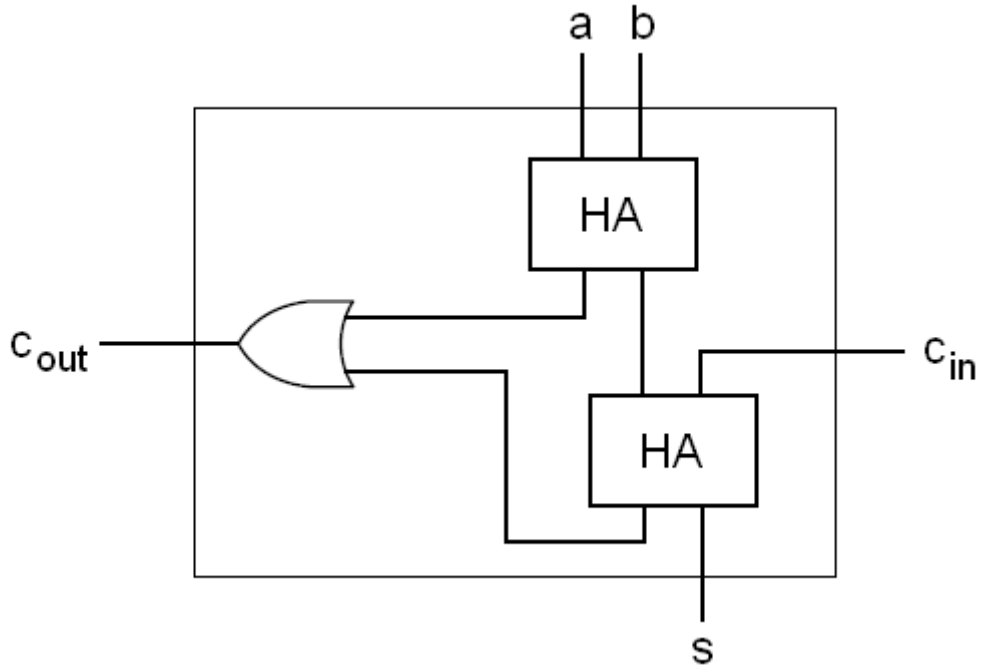
- Just like regular, grade school addition
  - Make sure we carry a 1 to the next digit when needed
- Now we need to be able to account for the carry-in from the next least-significant bit
- Example: 7+5

Addition

- We’re going to chain together thirty-two 1-bit “full adders”
- Each full adder has
  - 3 inputs: a, b, and carry in (which is the carry out of the previous bit)
  - 2 outputs: sum and carry out

```
 1 1 1
 0111
 0101
 ----
 1100
```

Full Adder from Half Adders



Build a full adder from two half adders

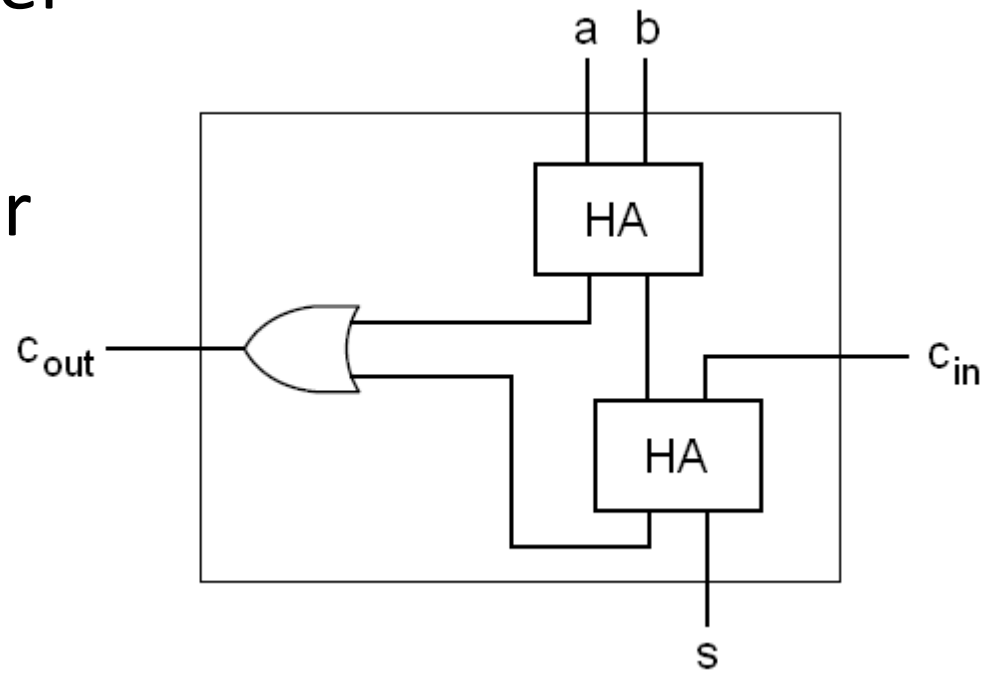
What if both half adders have carry-out?

A. We will get the wrong answer

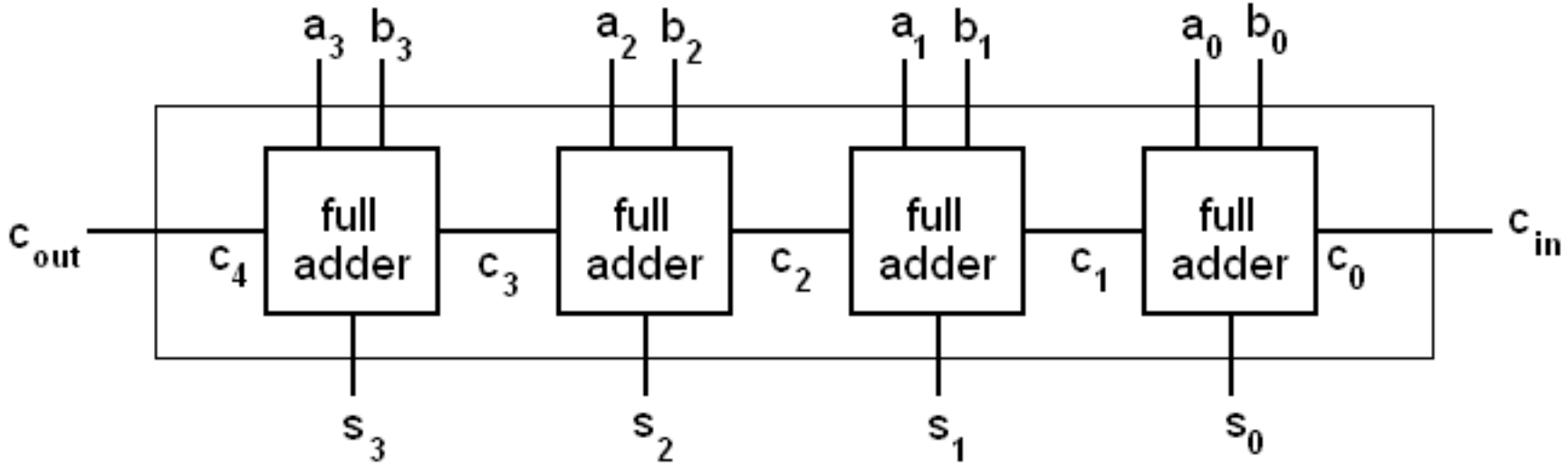
B. We will ignore it; the answer will still be correct

C. That will never happen

D. None of the above



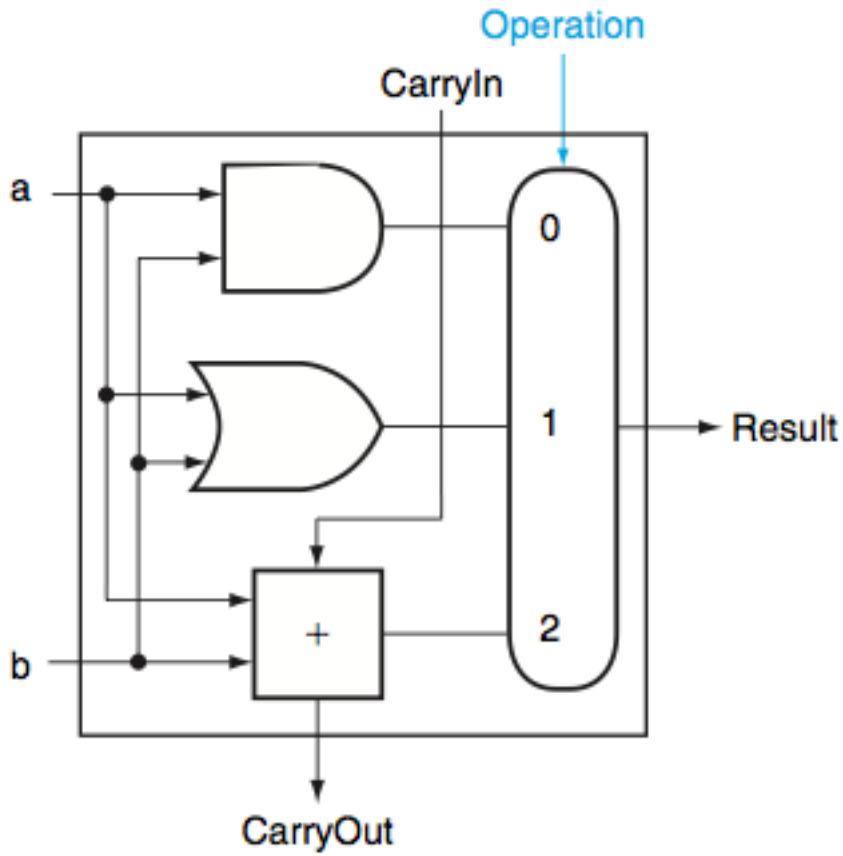
Ripple-Carry Adder



- Create adder for an arbitrary number of bits simply by connecting carry-out from adder n-1 to the carry-in for adder n

- Carry bit “ripples” up

1-bit ALU



Subtraction: a − b

- Just add negative version of b!

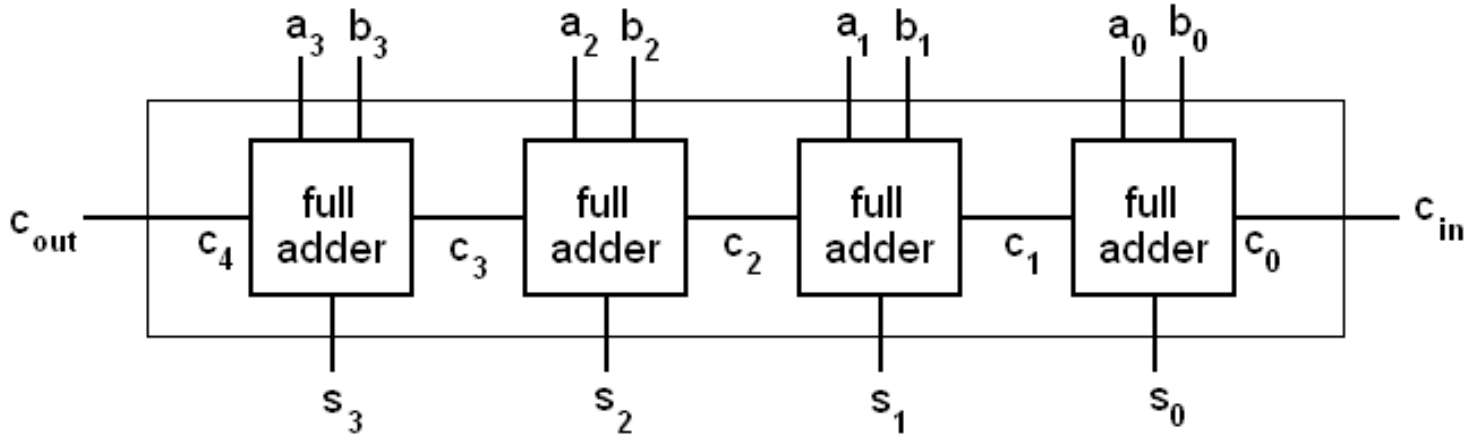
- To negate operand, take its two's compliment
  - Invert each bit
  - Add one

We can use a NOT gate to invert the input. To add one to the input, we should

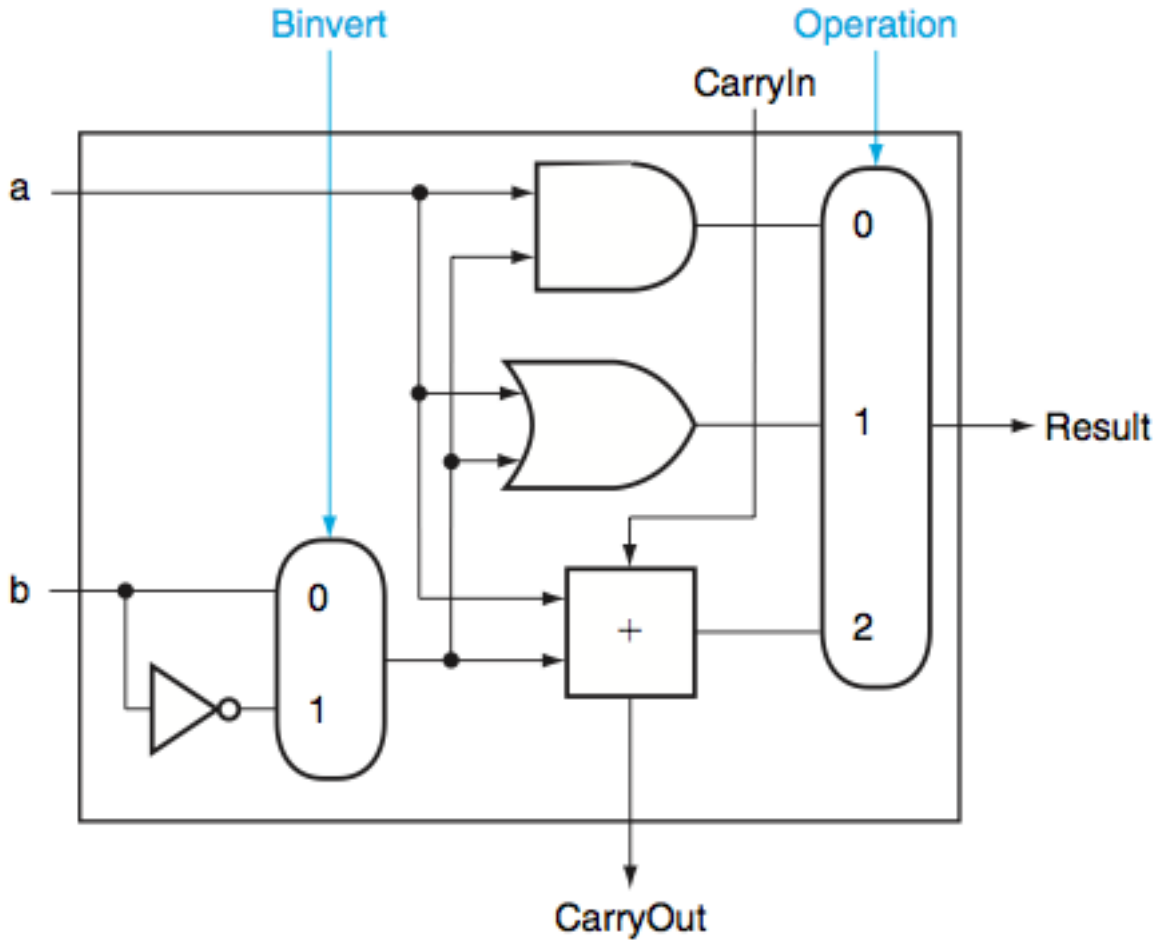
A. Set the carry-in for the least significant bit to 1

B. Add a new “subtract” input to each full adder that we set to 1 for subtraction and 0 for addition

C. Do something else

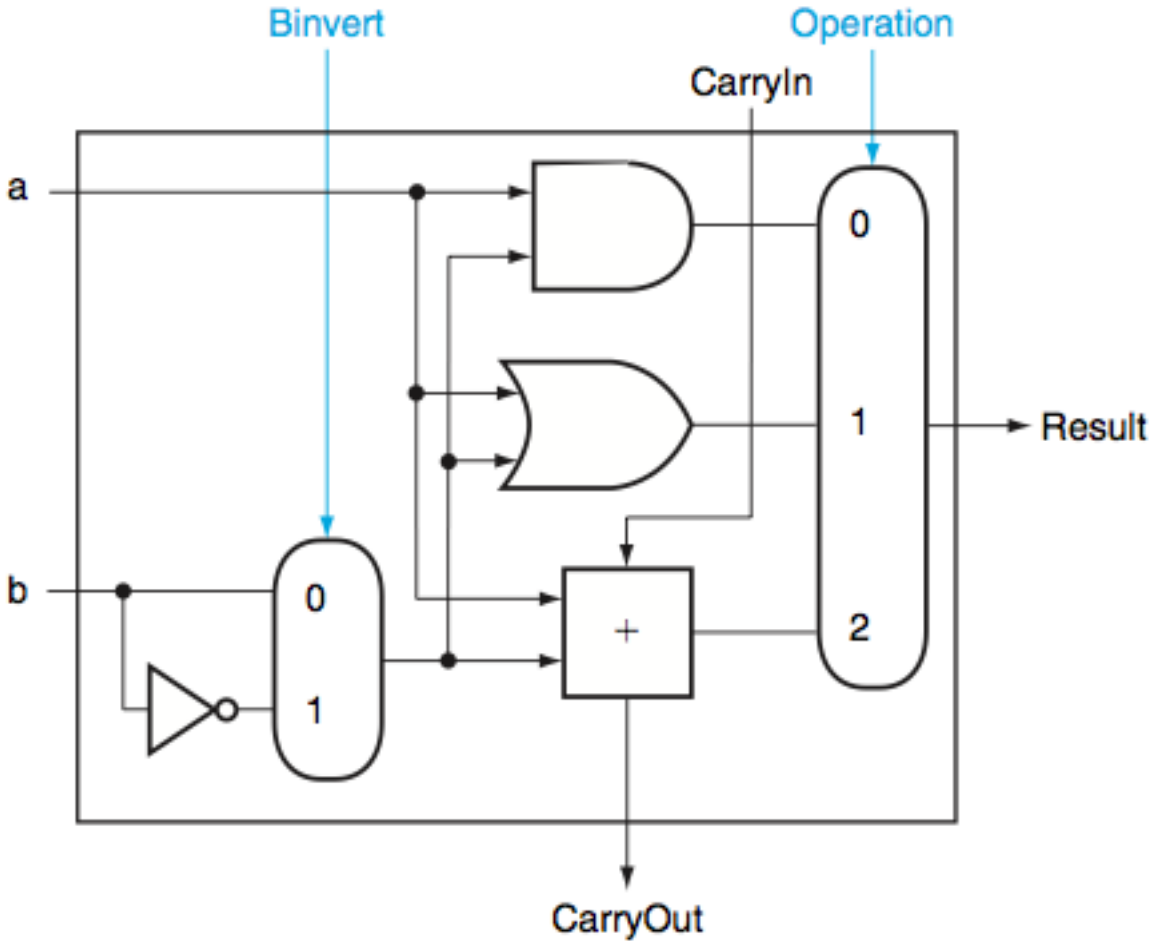


1-bit ALU with Subtraction



This 1-bit ALU supports 4 operations (and, or, add, sub). How many bits are required in the operation input signal, and *why* do we need that many?

A. 1
B. 2
C. 3
D. 4



Adding NOR

- Want to add NOR functionality

- De Morgan's Law
  - $\overline{(A+B)} = \bar{A}\,\bar{B}$

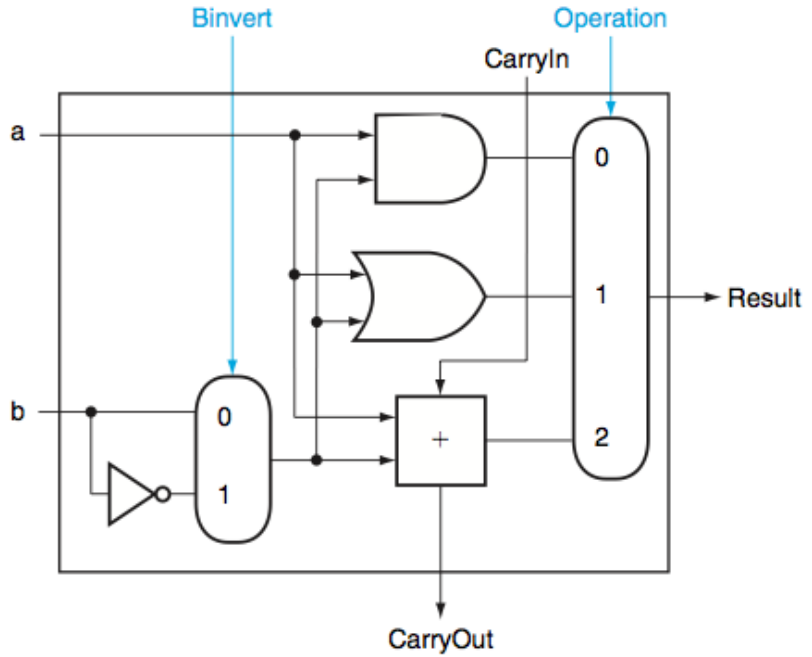
To add NOR to the ALU, we need to add

A. Nothing

B. The ability to invert A

C. A NOR gate

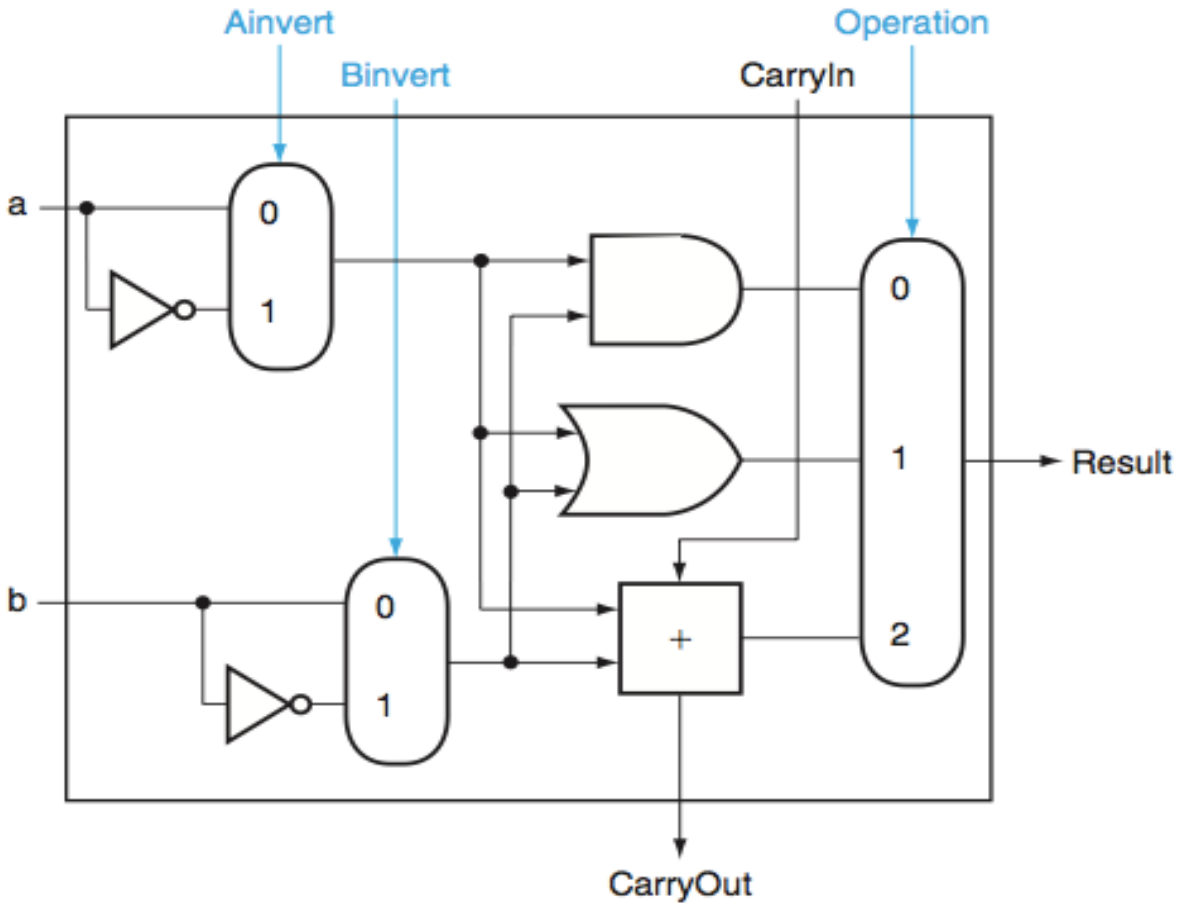
D. Something else



De Morgan's Law

$\overline{(A+B)} = \bar{A}\,\bar{B}$

1-bit ALU with NOR

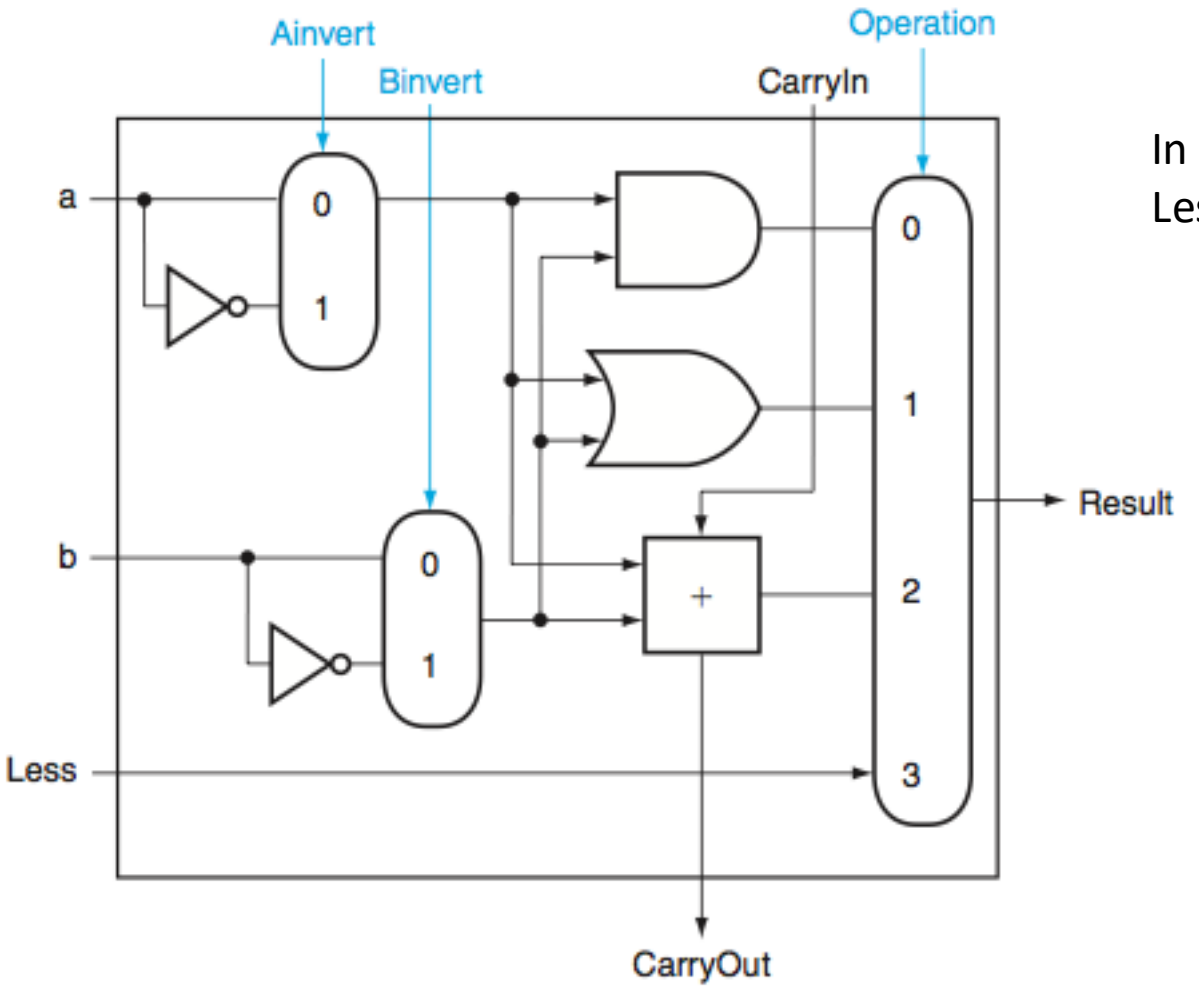


Adding slt

- `slt rd, rs, rt`
  - rd = 1 if rs < rt, and 0 otherwise

- Only sets least significant bit
  - All other bits are 0

1-bit ALU: Add new input for slt

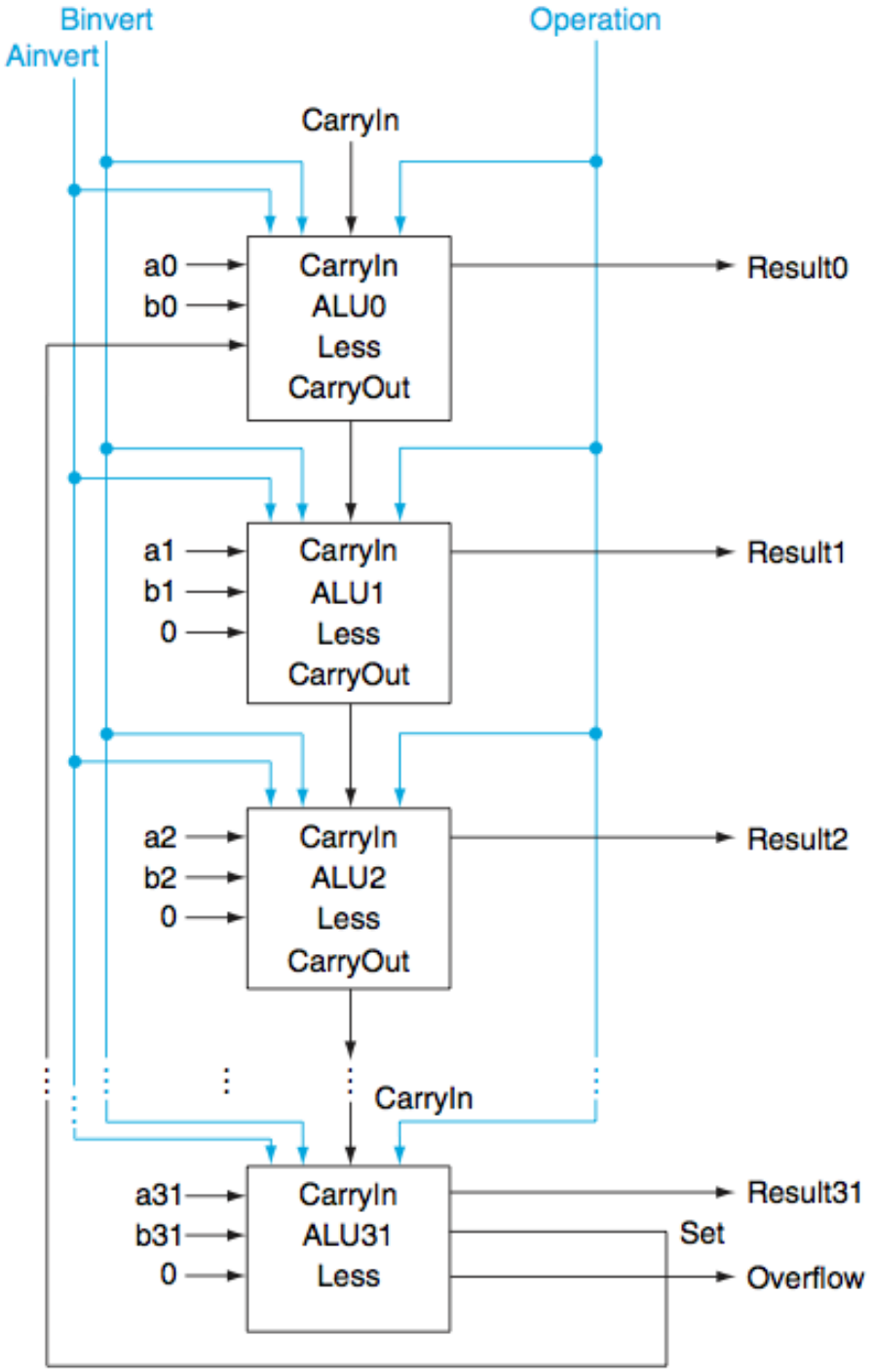


In all but the least significant bit,
Less will always be 0

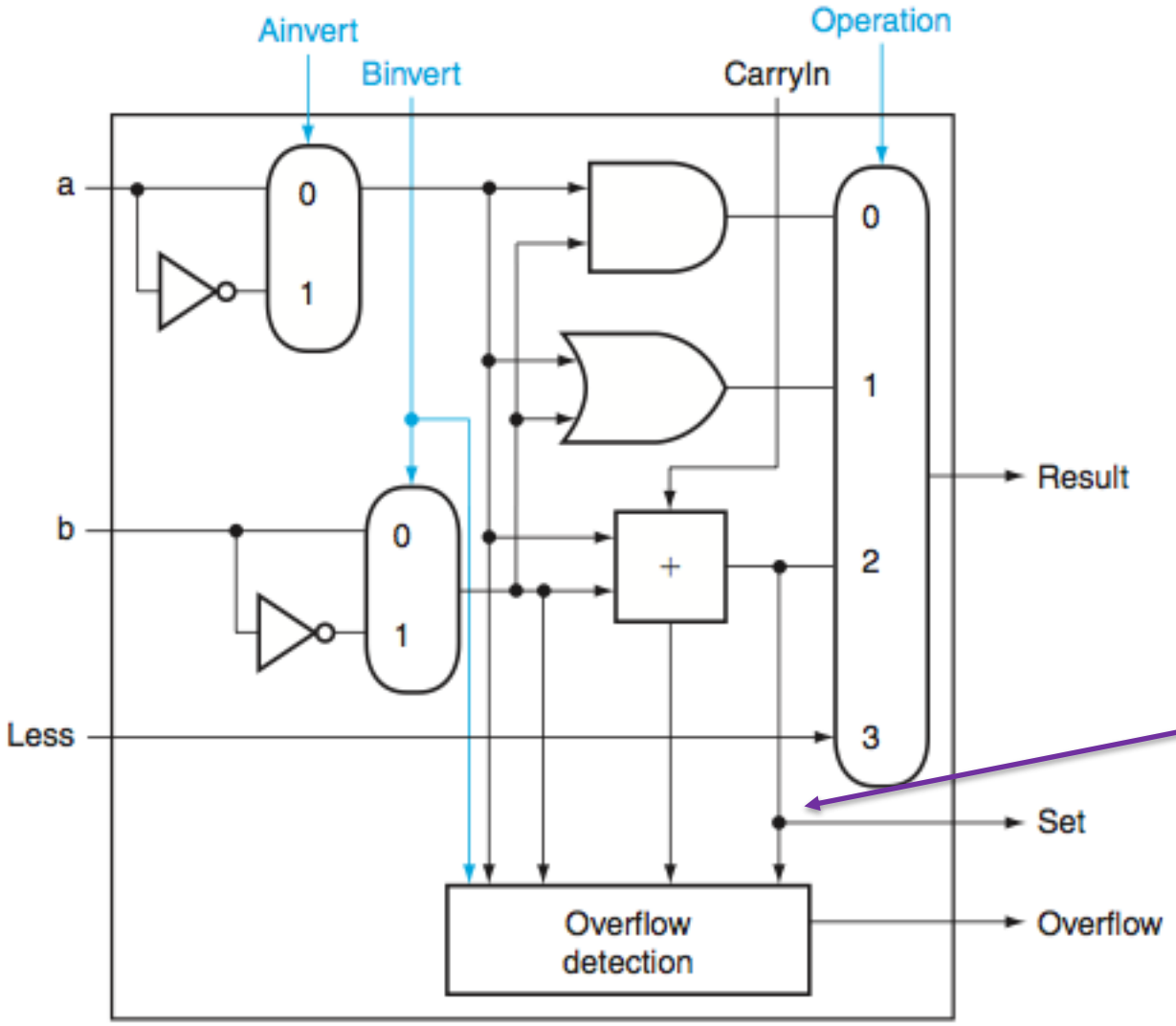
How do we tell if a < b?

- Subtract b from a
- If a – b < 0, then a < b
- We can check this by checking the most significant bit
  - MSB = 1, a < b

- Problem: Output is at most significant bit, we need it at least significant bit
- Solution: Special ALU for Most Significant Bit, with output for SLT
- Connect Set output to Less input for least significant bit



1-bit ALU for the Most Significant Bit



This doesn't always work! You'll fix it in problem set 6

Recall: Overflow

- If we add two n-bit numbers, we may end up with a number we can only represent in at least n+1 bits

- Hardware can detect this

a and b have different signs. Will adding them ever result in overflow?

A. Yes

B. No

Adding overflow detection to add

- If a and b have different MSBs, then there is no overflow
- If a and b have the same MSB, then
  - If the output MSB is different from the input MSBs, then overflow occurred
- Another way to check for overflow: If the carry into the MSB differs from the carry out of the MSB, then overflow occurs

Overflow if carry into MSB ≠ carry out of MSB (All numbers in binary!)

- Set up: We're adding two numbers let a and b be the MSB of the two inputs to add. Four cases to consider
  1. Carry in but no carry out: a + b + 1 ≤ 1 (because no carry out) so a = b = 0 but MSB of result is **1** (because 0 + 0 + 1 = 0**1**): **Overflow**
  2. No carry in but carry out: a + b + 0 > 1 (because carry out) so a = b = 1 but MSB of result is **0** (because 1 + 1 + 0 = 1**0**): **Overflow**
  3. Carry in and carry out: a + b + 1 > 1. If a ≠ b, then no overflow. If a = b, then a = b = 1 so MSB of result is **1** (because 1 + 1 + 1 = 1**1**): **No overflow**
  4. No carry in, no carry out: a + b + 0 ≤ 1. If a ≠ b, then no overflow. If a = b, then a = b = 0 so MSB of result is **0** (because 0 + 0 + 0 = 0**0**): **No overflow**

To check if the Carry_in is different from the Carry_out, check if

A. Carry_in AND Carry_out == 0

B. Carry_in OR Carry_out == 1

C. Carry_in NOR Carry_out == 0

D. Carry_in XOR Carry_out == 1

E. None of the above

Reading

- Next lecture: Clocks, Latches and Flip flops
  - 3.6

- Problem set 5
  - Due Friday

- Lab 4
  - Due Monday