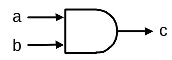
Hardware implementation

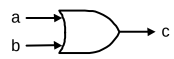
Review: Basic Hardware

1. AND gate (c = a . b)



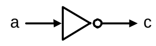
a	b	c = a . b
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate (c = a + b)



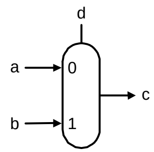
a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)



a	$c = \bar{a}$
0	1
1	0

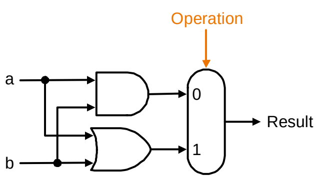
4. Multiplexor
   (if d = = 0, c = a;
   else c = b)

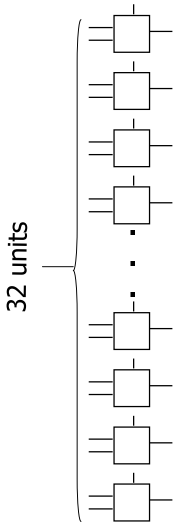


d	c
0	a
1	b

Implementation with a Multiplexor

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

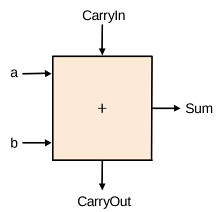




- Lets build our ALU using a MUX (multiplexor):

Implementations

- Not easy to decide the *best* way to implement something
  - do not want too many inputs to a single gate
  - do not want to have to go through too many gates (= levels)
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



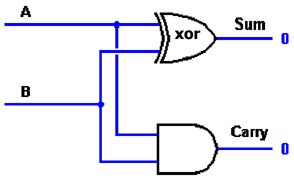
$$c_{out} = a.b + a.c_{in} + b.c_{in}$$

$$sum = a.\bar{b}.\bar{c}_{in} + \bar{a}.b.\bar{c}_{in} + \bar{a}.\bar{b}.c_{in} + a.b.c_{in}$$
$$= a \oplus b \oplus c_{in}$$

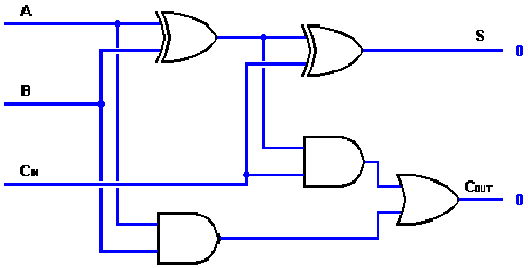
exclusive or (xor)

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

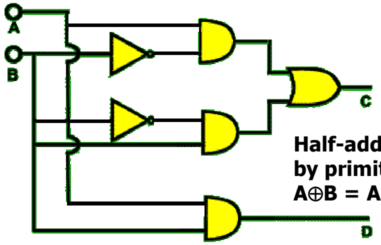
1-bit Adder Logic



Half-adder with one xor gate

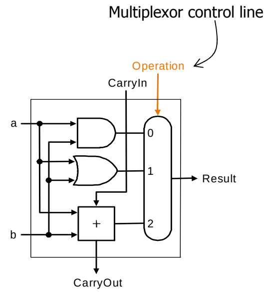


Full-adder from 2 half-adders and an or gate

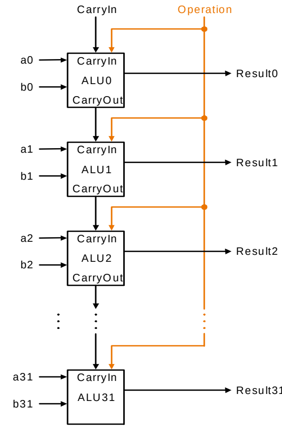


Half-adder with the xor gate replaced by primitive gates using the equation $A \oplus B = A.\bar{B} + \bar{A}.B$

Building a 32-bit ALU



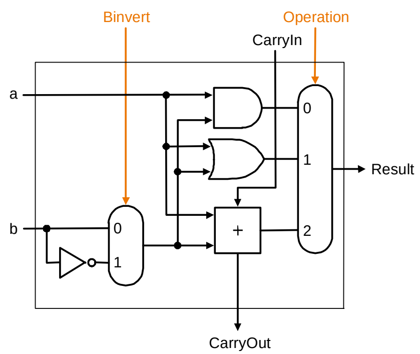
1-bit ALU for AND, OR and add



Ripple-Carry Logic for 32-bit ALU

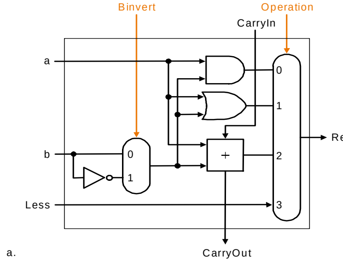
What about Subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- How do we negate?
  – recall *negation shortcut* : invert each bit of b and set CarryIn to *least significant bit* (ALU0) to 1



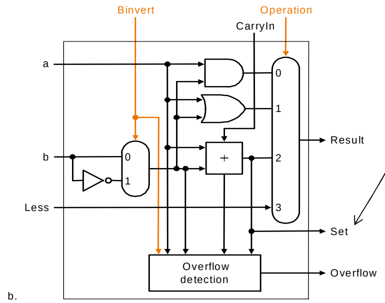
Tailoring the ALU to MIPS: Test for Less-than and Equality

- Need to support the *set-on-less-than* instruction
  – e.g., `slt $t0, $t3, $t4`
  – remember: `slt` is an *R-type instruction* that produces 1 if rs < rt and 0 otherwise
  – idea is to use subtraction: rs < rt ⇔ rs – rt < 0. Recall msb of negative number is 1
  – two cases after subtraction rs – rt:
    - <u>if no overflow</u> then rs < rt ⇔ most significant bit of rs – rt = 1
    - <u>if overflow</u> then rs < rt ⇔ most significant bit of rs – rt = 0
  – set bit is sent from ALU31 to ALU0 as the *Less* bit at ALU0; all other Less bits are hardwired 0; so Less is the 32-bit result of `slt`

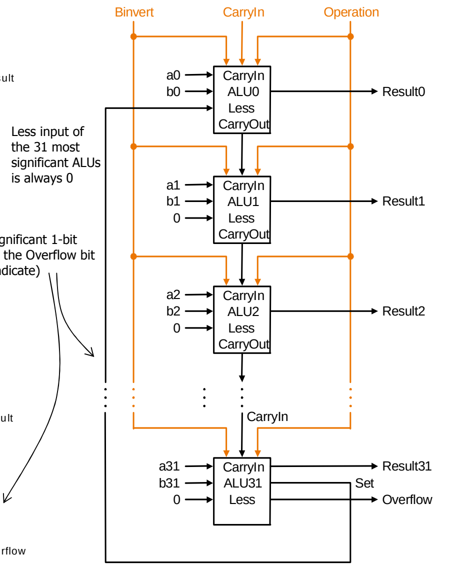


1- bit ALU for the 31 least significant bits

Extra set bit, to be routed to the Less input of the least significant 1-bit ALU, is computed from the most significant Result bit and the Overflow bit (it is *not* the output of the adder as the figure seems to indicate)



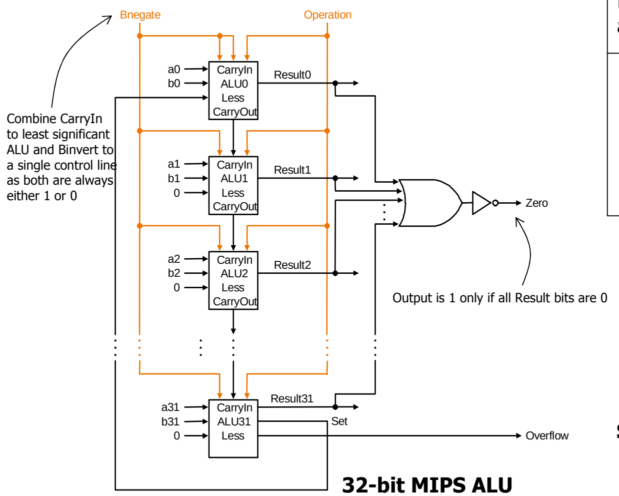
1-bit ALU for the most significant bit



32-bit ALU from 31 copies of ALU at top left and 1 copy of ALU at bottom left in the most significant position

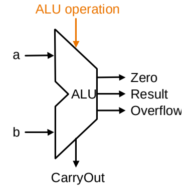
Tailoring the ALU to MIPS: Test for Less-than and Equality

- What about logic for the *overflow bit* ?
  - overflow bit = carry *in to* msb ⊕ carry *out of* msb
  - logic for overflow detection therefore can be put in to ALU31
- Need to support *test for equality*
  - e.g., `beq $t5, $t6, $t7`
  - use subtraction: rs - rt = 0 ⇔ rs = rt



32-bit MIPS ALU

ALU control lines		
Bneg-ate	**Oper-ation**	**Func-tion**
0	**00**	**and**
0	**01**	**or**
0	**10**	**add**
1	**10**	**sub**
1	**11**	**slt**



Symbol representing ALU

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 gates are always working
  - speed of a gate depends number of inputs (fan-in) to the gate
  - speed of a circuit depends on number of gates in series (particularly, on the *critical path* to the deepest level of logic)
- Speed of MIPS operations
  - clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at examples for addition, multiplication and division