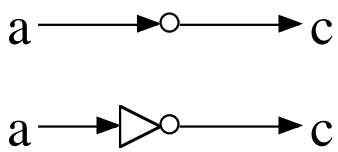
Gates and Boolean Algebra

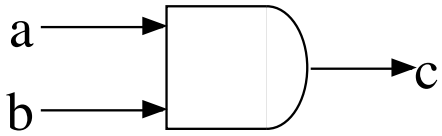
Read: Appendix C (4th edition) or Appendix B (5th edition), sections 1, 2, and 3 (partial) — pages 1 to 13.

- NOT — unary operator, written as a bar over the variable.
  - Example: $\overline{a}$, true (1) if a is 0.
  - NOT gate ($c = \overline{a}$).
  - There are two different drawings used for NOT.

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

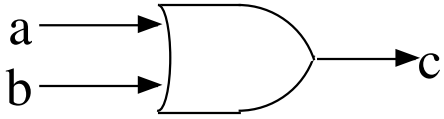


- AND — written as •
  - Example: A • B, true (1) only if both A and B are 1.
  - AND gate (c = a AND b, c = a • b).



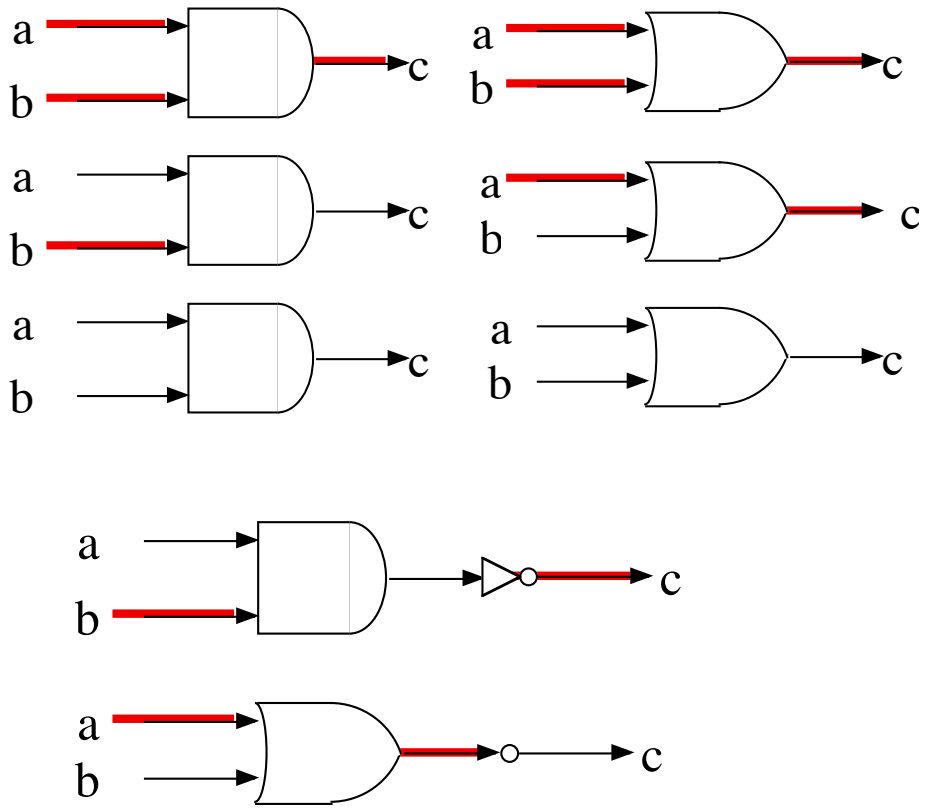
a	b	c = a • b
0	0	0
0	1	0
1	0	0
1	1	1

- OR — written as +
  - Example: A + B, true (1) if at least one of A or B is 1.
  - OR gate (c = a OR b, c = a + b)

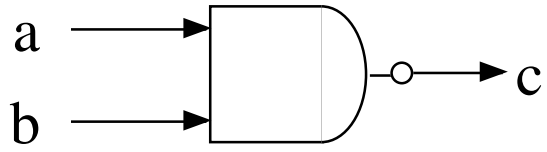


a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

- Some examples:

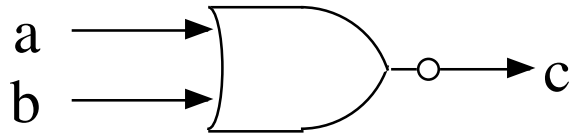


- NAND — not and:



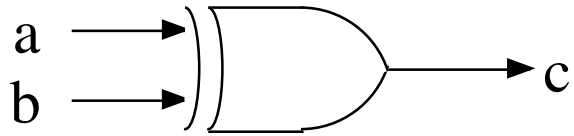
a	b	c = a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

- NOR — not or:



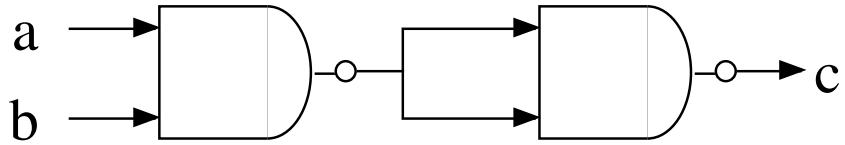
a	b	c = a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

- XOR — exclusive or:



a	b	c = a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

- It is possible to implement all the other types of gates from NAND gates, or from NOR gates.
  - Simplifies the underlying structure
  - For example, an AND gate can be constructed from two NAND gates:



Binary Addition

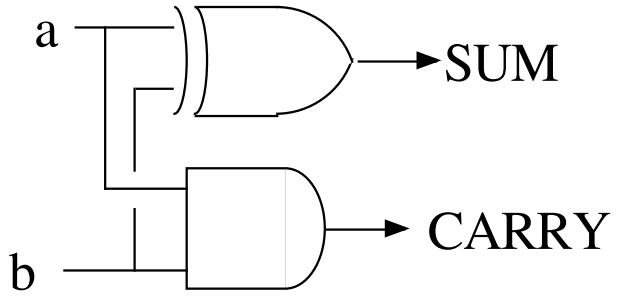
Read: Appendix C (4th edition) or Appendix B (5th edition), section 5 — pages 26 to 35.

Binary Addition, first attempt:

- Need to compute a + b to get the sum, and the carry:

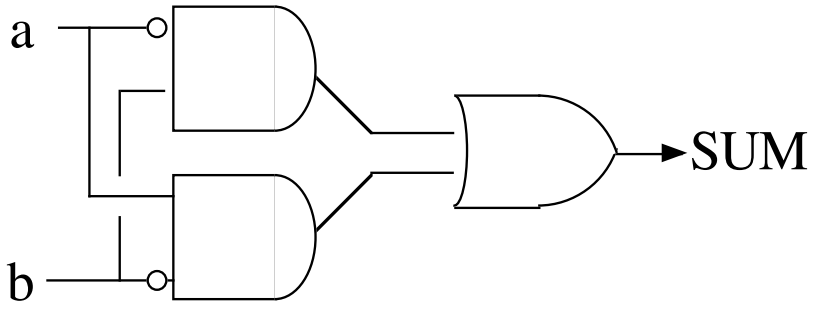
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Here is one solution using one XOR and one AND gate:



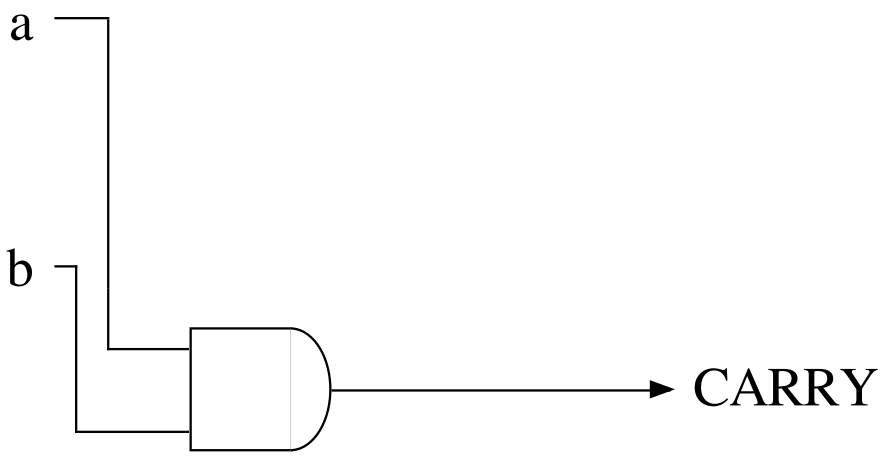
Binary Addition, first attempt (continued):

- Here is a second solution. This one starts by computing the Sum bit. It uses two AND gates, one OR gate, and two NOT gates (do you see the NOT gates?):



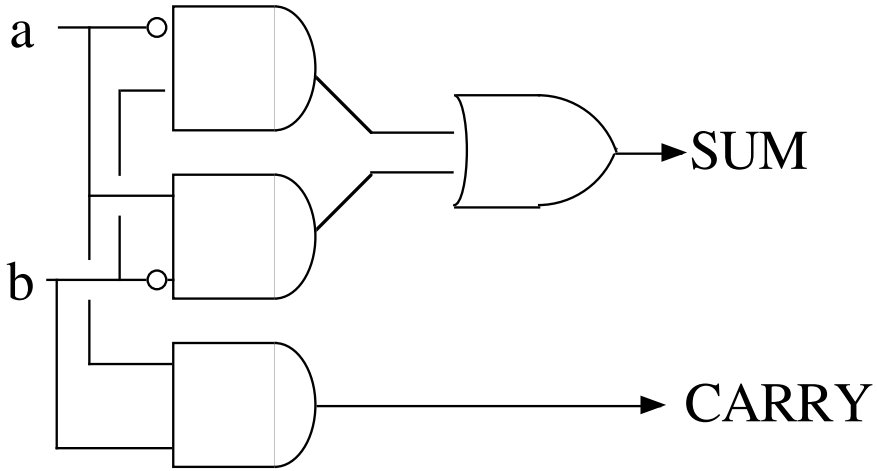
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Then, to get the Carry bit:

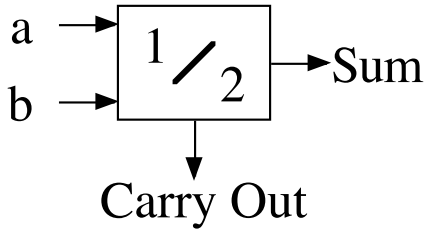


Binary Addition, first attempt (continued):

- Put the two parts from the previous slide together:



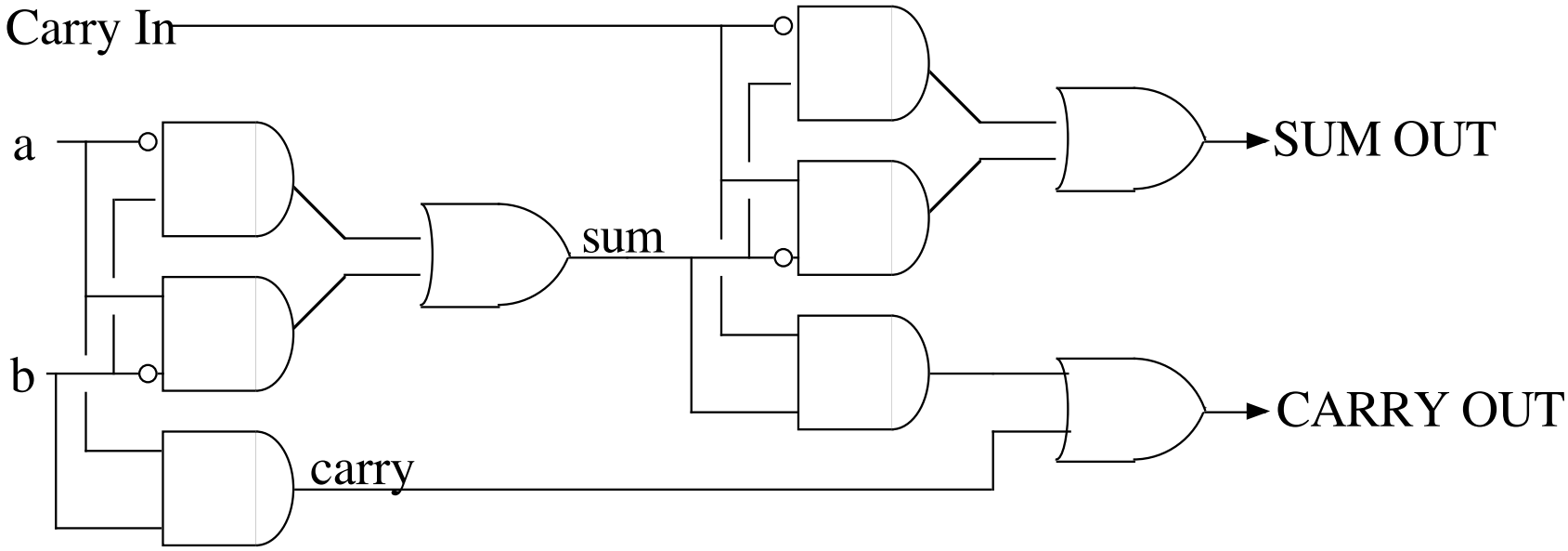
  - Either of these two solutions is a *Half Adder.*
    - It adds the two binary digits producing the correct sum and carry results.
    - To simplify drawing the picture, we use:



Binary Addition, first attempt (continued):

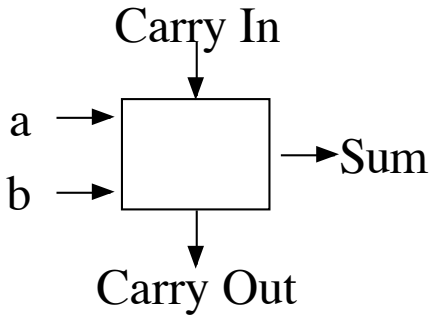
```
 1101
+0011
```

Inputs			Outputs	
A	B	Carry In	Carry Out	Sum
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

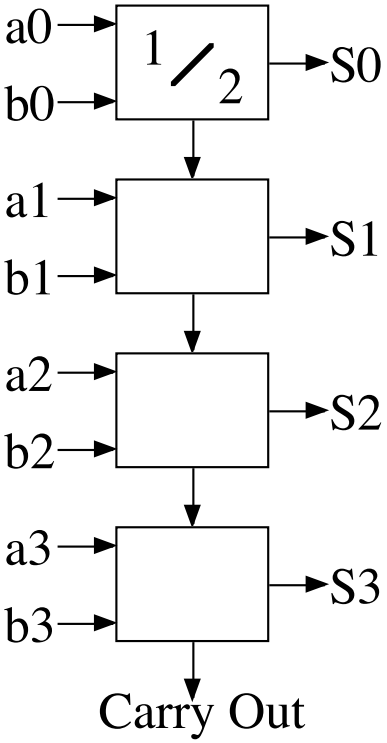


Binary Addition, first attempt (continued):

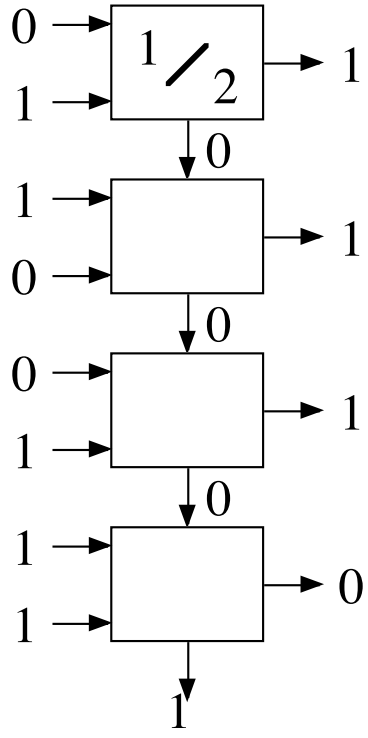
- Again, we use a simplified drawing to represent the *full adder*:



- We can hook Full Adder's together to add more than one pair of bits:




```
   1
  a = 1010
+ b = 1101
  s = 0111, carry out = 1
```



Binary Addition, first attempt (continued):

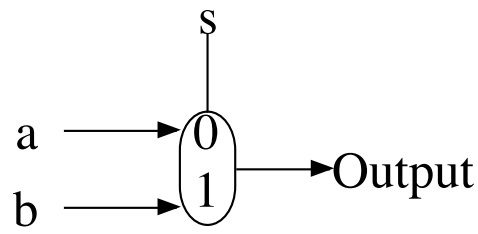
- This is called a Ripple Adder.
  - Note how the Carry has to "ripple" through for each bit.
  - Implies that each bit cannot compute its sum until all the previous bits have finished.
  - This can be extended to any number of bits: 8, 16, 32, 64, …
  - Ripple adders are vvveeerrryyy sssllloooowww.
    - We will see a much better solution later.

```
  1001 0110 0110 0111
+ 1011 1001 1001 1100

 1001 0110   0110 0111
+1011 1001 + 1001 1100
```

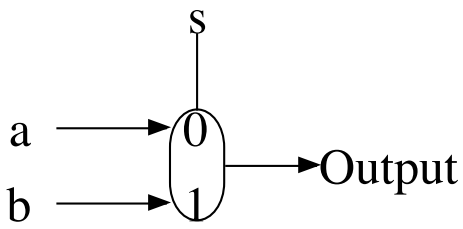
Multiplexor (Selector).

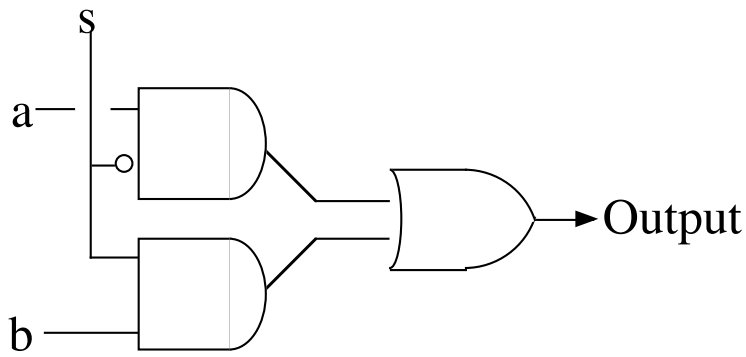
- The output equals one of the inputs as determined by the *selector* or *control* input.

a	b	s	Output
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1



- Construction of a two-input multiplexor:





Summary, thus far:

- There are three key types of logic gate: AND, OR, NOT.
- All three can be implemented in terms of NAND gates, or in terms of NOR gates.
- Logic gates can be assembled to form circuits.
    - We have already covered how to create *half-adders*, *full-adders*, and *multiplexors* using AND, OR, and NOT gates.
- Such circuits can be used to store and manipulate binary information.

Boolean Algebra:

Reading: Section C.2 (4th edition), Section B.2 (5th edition)

- Reminder:
  - OR — written as +
    - Example: A + B, true (1) if at least one of A or B is 1.
  - AND — written as •
    - Example: A • B, true (1) only if both A and B are 1.
  - NOT — unary operator, written as a bar over the variable.
    - Example: $\overline{A}$, true (1) if A is 0.
- Useful laws of Boolean Algebra (see page C-6, 4th edition)
  - Identity law: A + 0 = A and A • 1 = A.
  - Zero and One laws: A + 1 = 1 and A • 0 = 0.
  - Inverse laws: $A + \overline{A} = 1$ and $A \bullet \overline{A} = 0$.
  - Commutative laws: A + B = B + A and A • B = B • A.
  - Associative laws: A + (B + C) = (A + B) + C and A • (B • C) = (A • B) • C.
  - Distributive laws: A • (B + C) = (A • B) + (A • C) and A + (B • C) = (A + B) • (A + C)

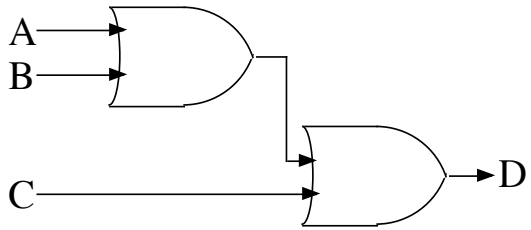
Boolean Algebra (continued):

- Problem: Construct a logic circuit for a device with three inputs: A, B, and C. The circuit will have three outputs: D, E, and F.
  - Output D as true (1) if at least one input is true. $D = A + B + C$
  - Output E as true (1) if exactly two inputs are true. $E = (\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C})$
  - Output F as true (1) if all three inputs are true. $F = A \bullet B \bullet C$
- The truth table for D, E, and F:

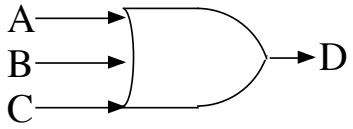
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Boolean Algebra (continued):

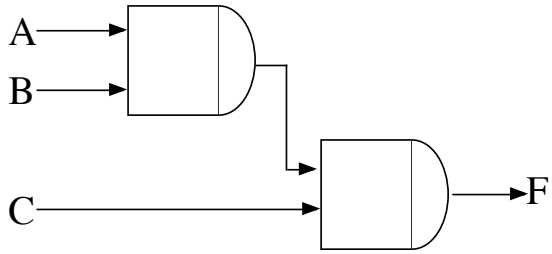
- D = A + B + C



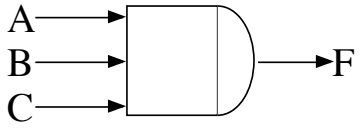
In the future, we will represent multi-input OR gates as:



- F = A • B • C

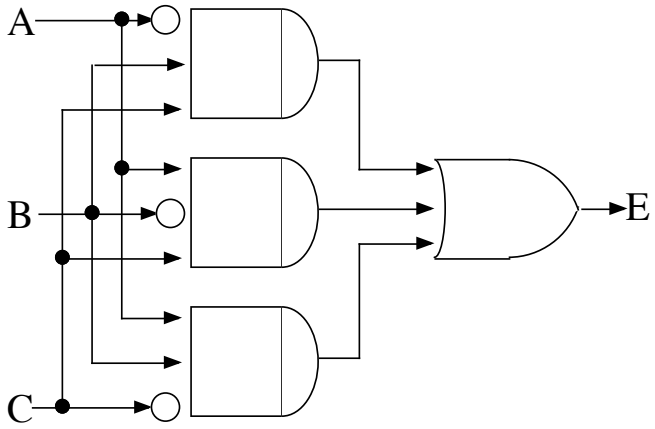


In the future, we will represent multi-input AND gates as:



Boolean Algebra (continued):

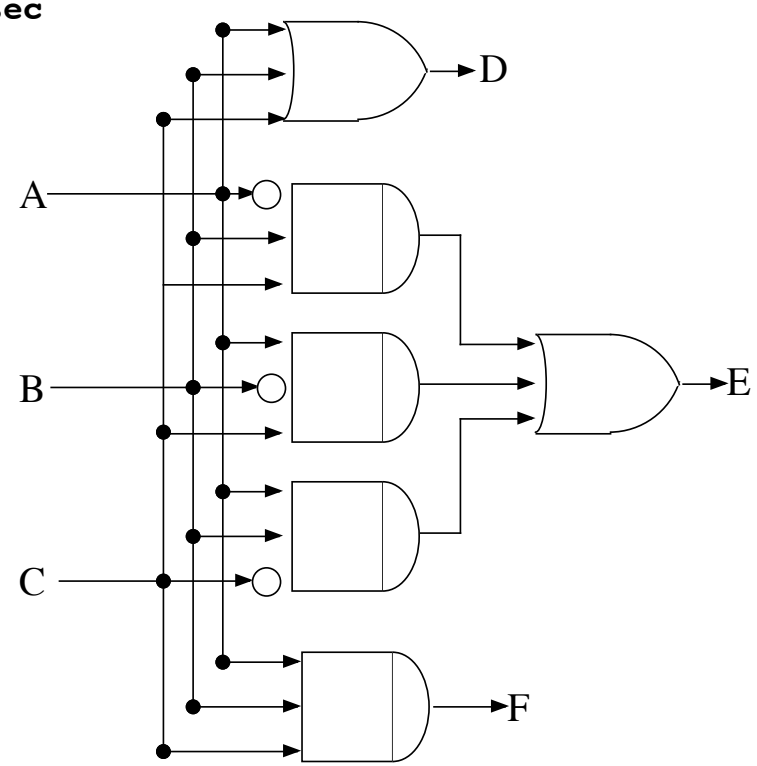
- Using the three-input OR and AND gates, we can construct the circuit for E:
  - $E = (\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C})$



- How to construct one circuit for computing all three outputs: D, E, F?
  - Combine the three circuits we have already built!
  - The only hard part is drawing the lines for A, B, and C to show where they cross over each other :-).
- Using multi-input AND and OR gates simplifies the drawing of circuits.
  - Importance of abstraction: Once we know how to construct the interior details, we no longer show those details!

Boolean Algebra (continued):

V = Voltage = energy of one electron
i = Current = # of electrons/sec
amp = ampere
Power = iV



Two wires that cross but do **not** touch.

Two wires that **are** connected.

Constructing an Arithmetic Logic Unit

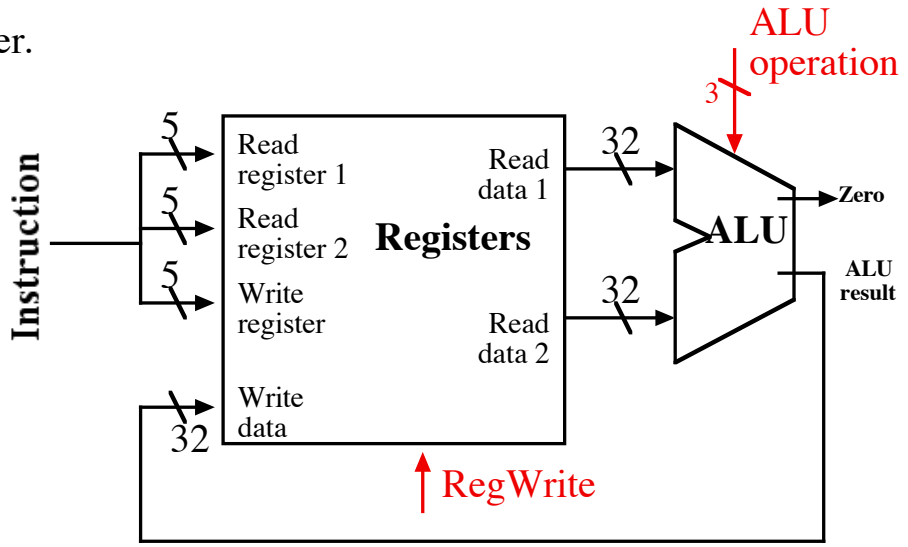
ALU Overview:

Reading: Section C-5 (4th edition) or Section B-5 (5th edition) — pages 26 to 35.

- The ALU that we will construct will:
  - Take two 32-bit input values, A and B.
  - Output one 32-bit output value C, and a single additional bit Z.
  - Compute the following operations:
    - Bit-wise Logical OR: C = A | B
    - Bit-wise Logical AND: C = A & B
    - Addition: C = A + B
    - Subtraction: C = A - B
    - Set Less Than: if (A < B) then C = 1 else C = 0
    - If Equal: if (A == B) then Z = 1 else Z = 0

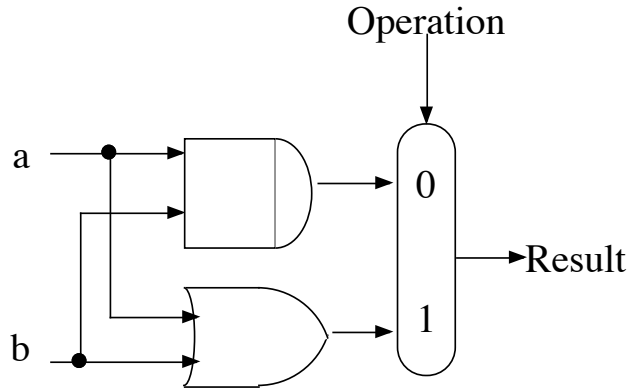
ALU Overview (continued):

- Where the ALU fits into the CPU:
  - The ALU is one part of the CPU.
    - ALU handles arithmetic (add, subtract) and logical (and, or) operations.
    - Inputs to the ALU come from registers.
    - Result of the ALU (generally) goes to a register.
  - 15 bits of an instruction specify the registers:
    - 2 source registers (values to read).
    - 1 destination register (value to write).
  - ALU operation needs 3 bits to control the ALU.
    - Will be a total of 5 operations.



ALU Logical operations:

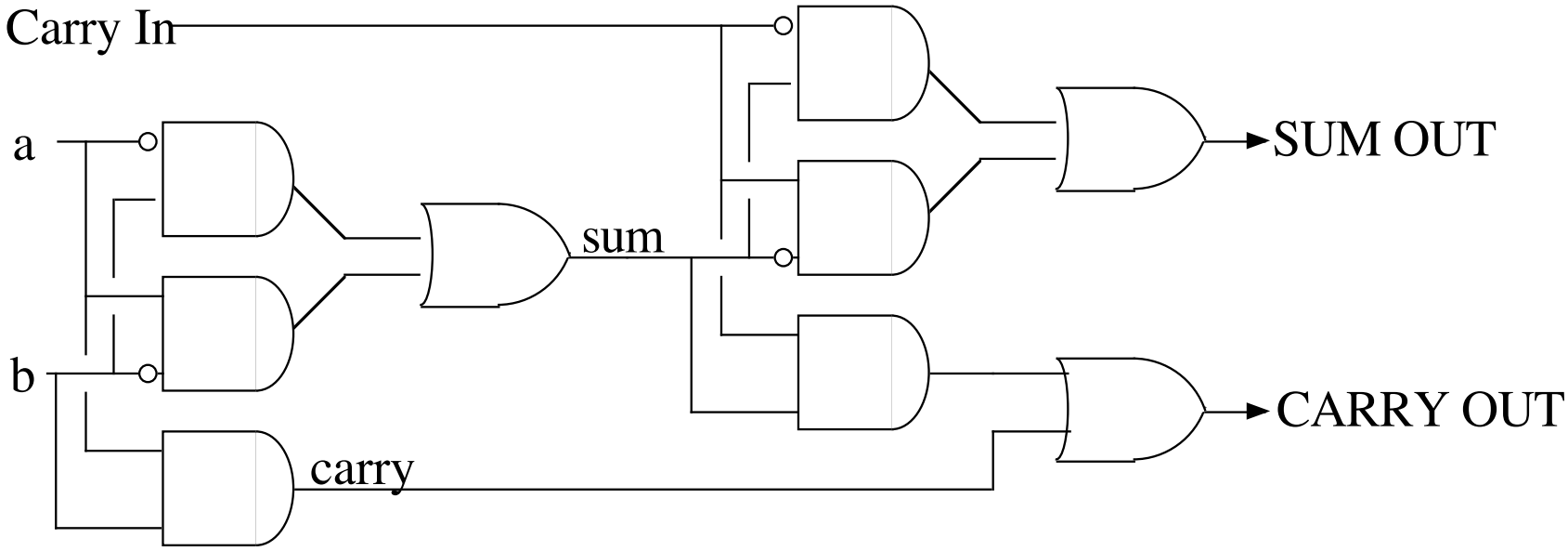
- Need the ability to do AND and OR operation on each pair of bits.
  - One bit from operand A; one bit from operand B.
- Need a multiplexor to choose the desired result.
  - Both operations computed each time.
  - Need to choose only one result.
- To compute a 32-bit result from two 32-bit operands:
  - Replicate the unit shown at right 32 times!



ALU Arithmetic operations:

- From Slide 9:
  - Combined two half-adders, plus one OR gate.
  - Six AND gates, three OR gates, four NOT gates.
  - 13 gates total. 6 gates on the longest path
- There are other ways to approach this problem.

Inputs			Outputs	
A	B	Carry In	Carry Out	Sum
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

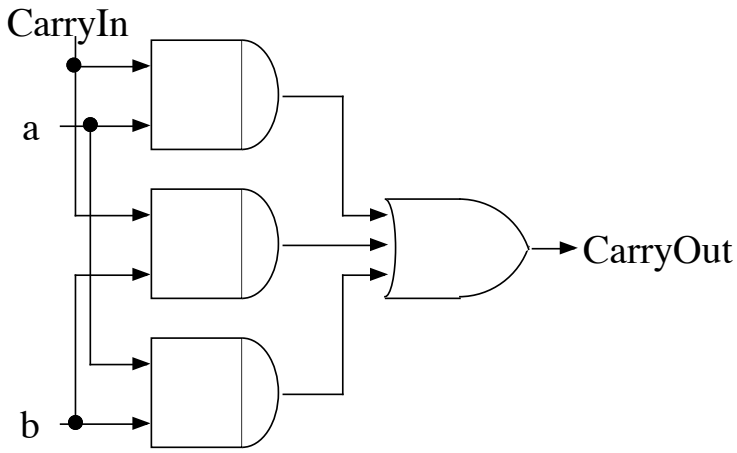


ALU Arithmetic operations (continued):

- When is Carry Out true?
  - If two of the inputs are one, or if all three are 1.

$$\text{CarryOut} = (a \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \text{CarryIn}) + (a \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

  - Could use a 3-input AND gate for each of the four terms.
    - One input to each of the first three AND gates would be negated.
  - Instead, we can optimize and use 2-input AND gates, leaving out the negated term each time.
  - Why does this work?
    - Leaving out the negate in the 3 AND gates results in true at all 3 AND gates when all 3 inputs are true.



Three AND gates.
One 3-input OR gate.

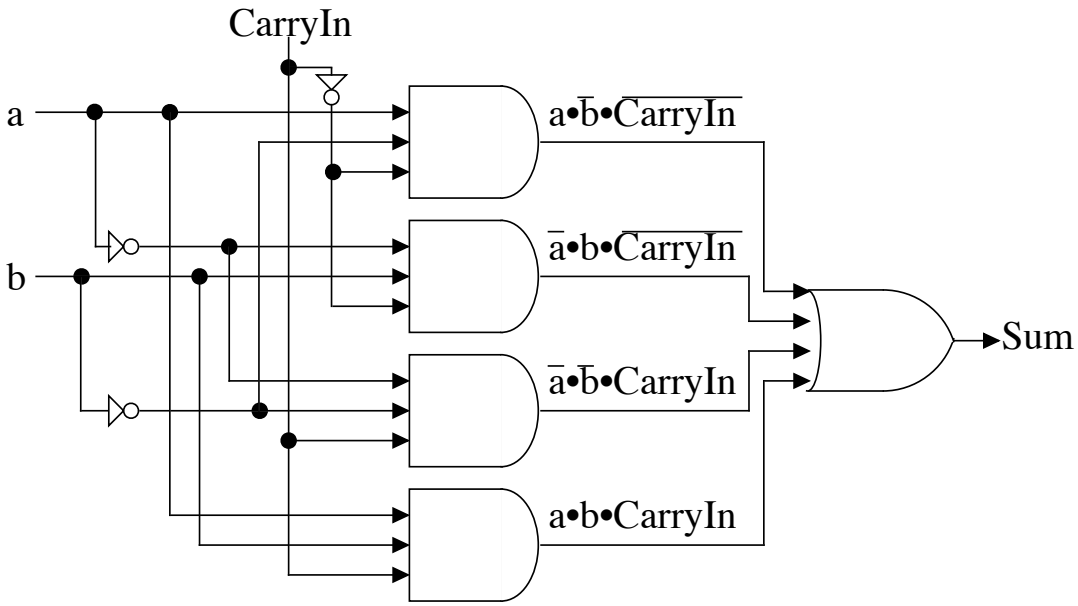
Total: 3 AND + 2 OR gates
= 5 gates

ALU Arithmetic operations (continued):

- When is Sum true (1)?
  - If exactly one of the three inputs (a, b, CarryIn) is true, or if all three are ture:

$$\text{Sum} = (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

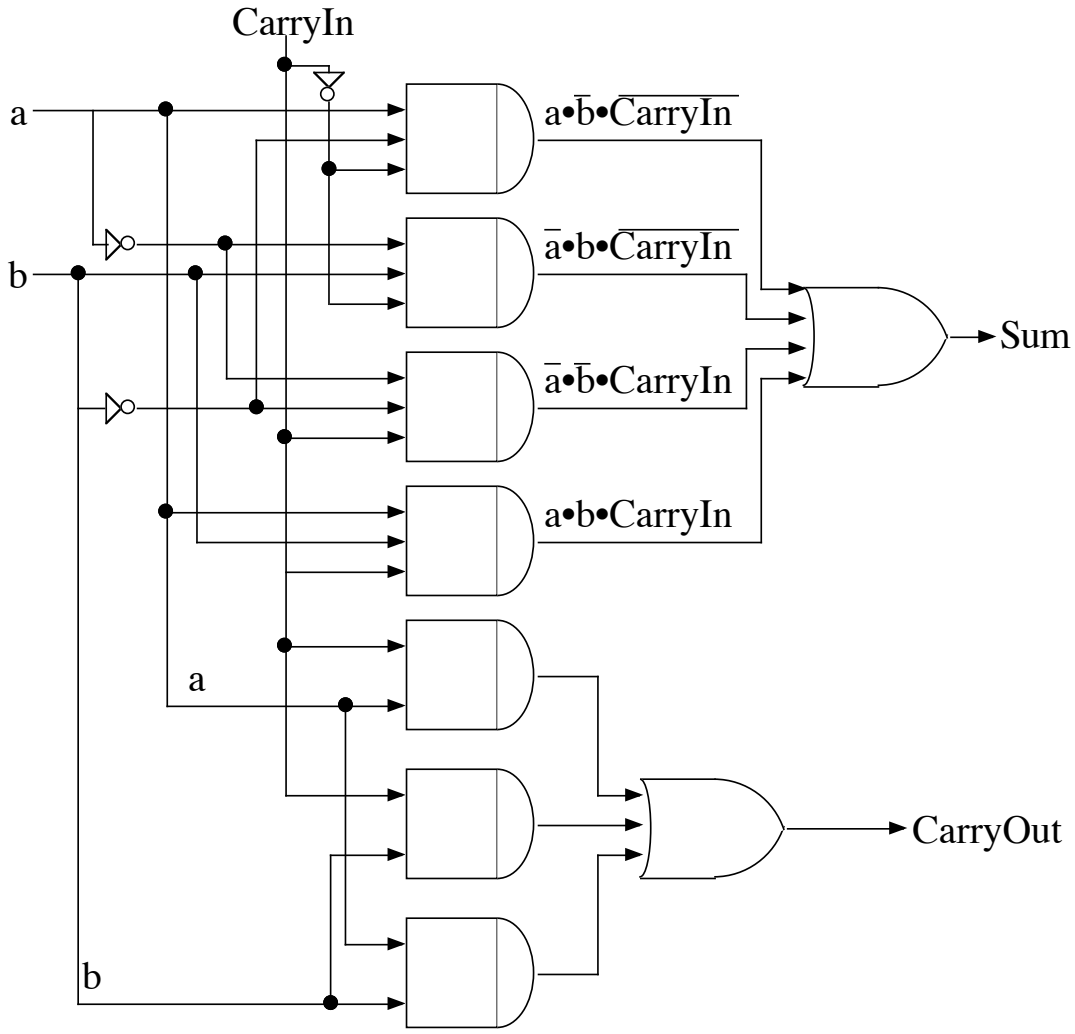
    - Four AND gates (3 inputs each), one OR gate (4 inputs), three NOT gates:
      - Total: 4 * 2 AND gates + 3 OR gates + 3 NOT gates = 14 gates



Inputs			Outputs	
A	B	Carry In	Carry Out	Sum
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

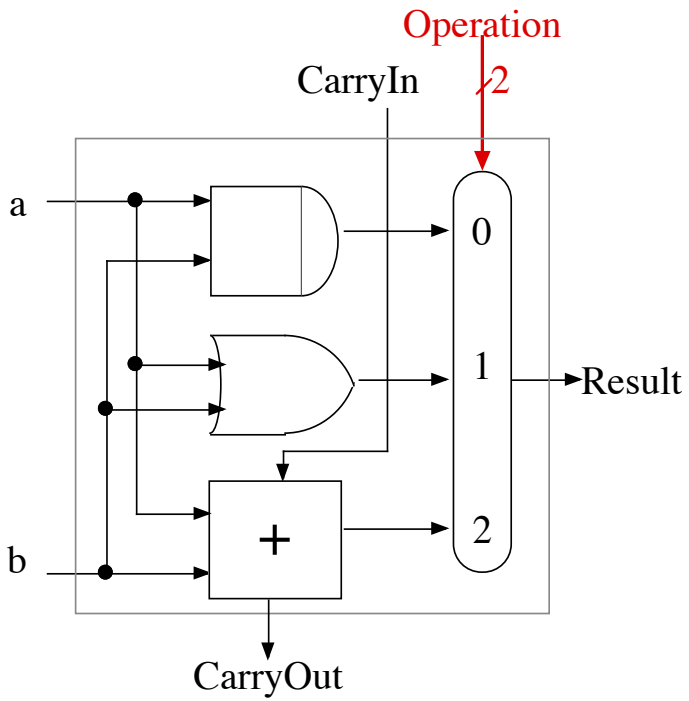
ALU Arithmetic operations (continued):

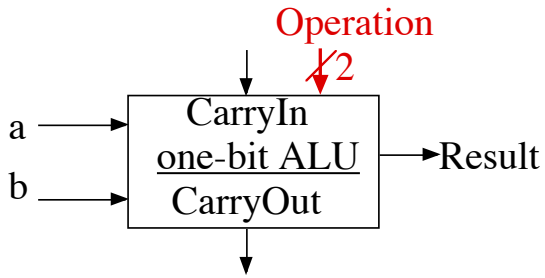
- Combined result of Sum and Carry bits.
  - Eleven AND gates.
  - Five OR gates.
  - Three NOT gates.
  - 19 gates total
- Earlier adder version used 13 gates.
- Where is the advantage?
  - We can compute the Sum bit *and* the CarryOut bit simultaneously.
  - Was done one after the other in the earlier version.



One-bit ALU: supports AND, OR, ADD:

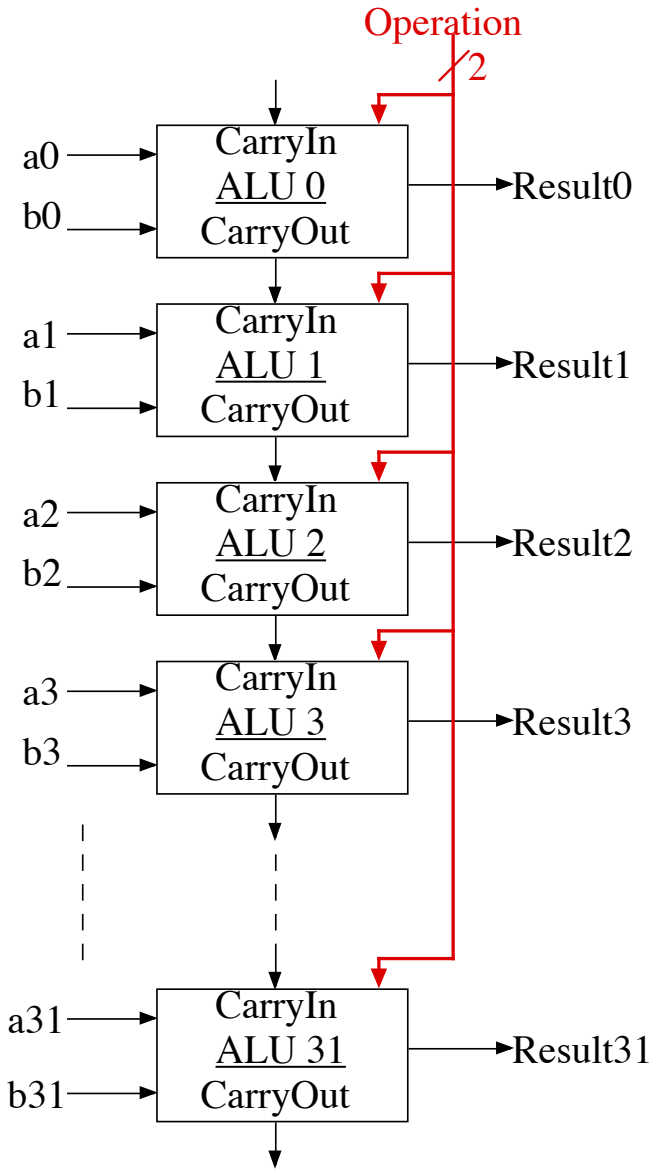
- Outputs:
  - 0: Result of AND.
  - 1: Result of OR.
  - 2: Result of ADD.
    - Includes CarryOut
- Note: All results are computed every time.
  - The Multiplexor allows the unit to send only one result to output.





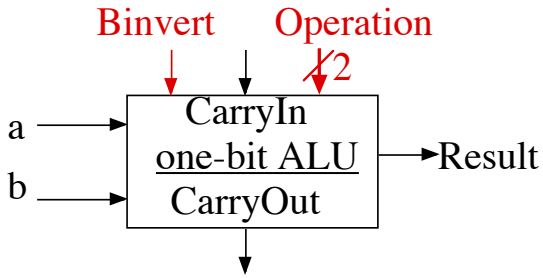
Combine one-bit ALU's to create 32 bit ALU:

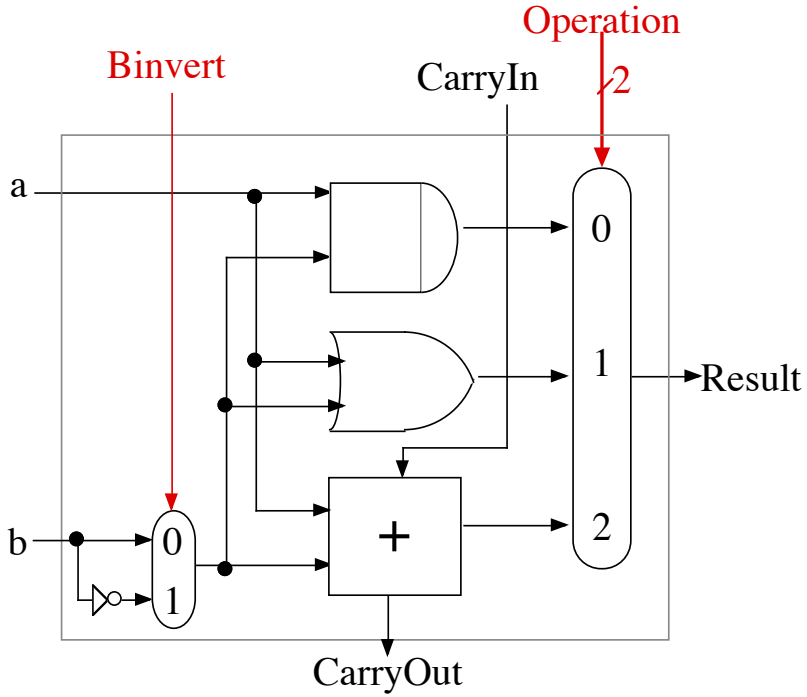
- Called a *Ripple Adder*.
- Figure B.5.7, page B-30.
- Drawback: Takes time to "ripple" the carry out bit through all 32 adders.



Subtraction:

- Two's complement approach: Find the two's complement of b, then add.
  - Two parts:
    - Invert step.
    - Add 1 step.
- Invert step is easy:

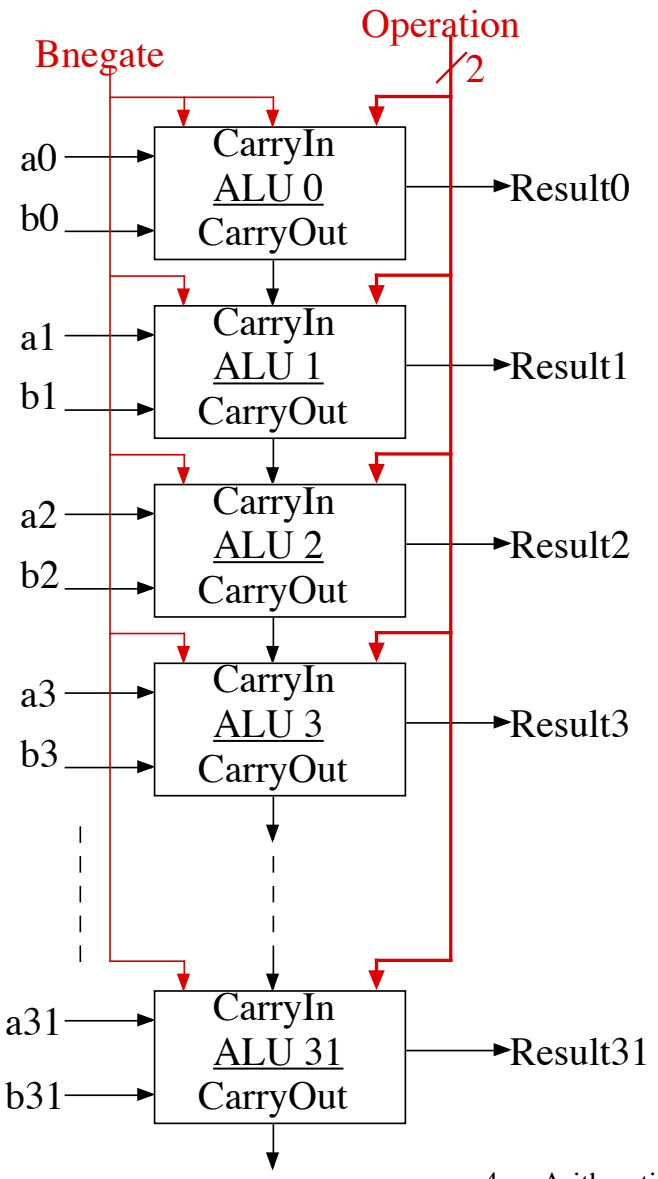




Subtraction (continued):

- Add 1 step (to get two's complement of input b):
  - Put 1 as the CarryIn to ALU 0.
  - Do this by connecting the Binvert control line to CarryIn for ALU 0.
- Since Binvert is now really Bnegate, change its name.
- Thus, to do subtraction:
  - Set Bnegate to 1.
  - Set Operation to ADD.

$$a - b = a + (-b)$$



Set-less-than:

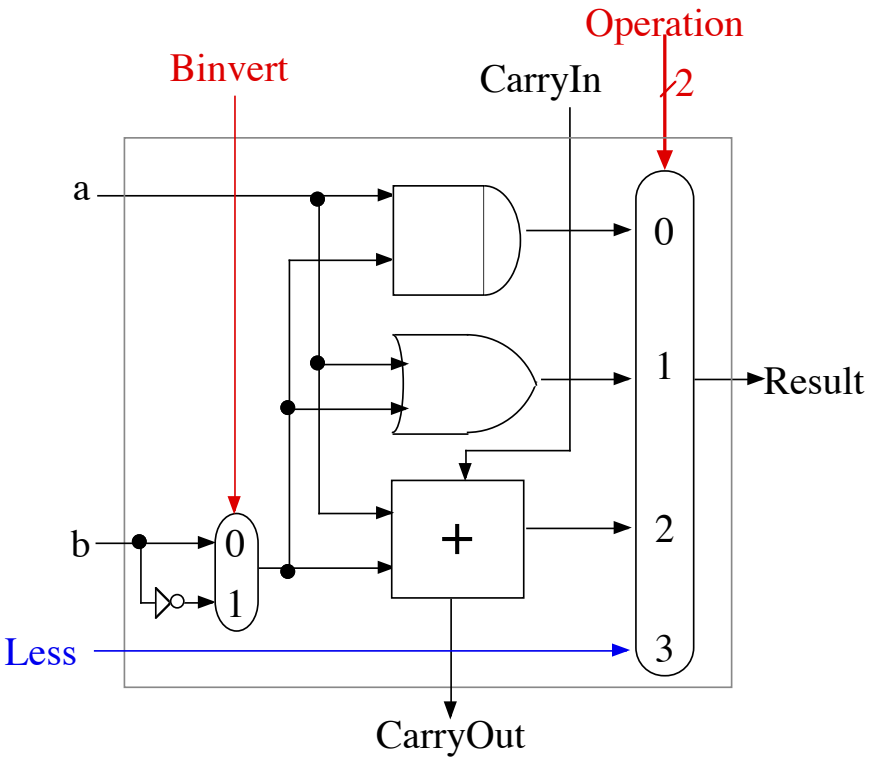
```
slt  $t3, $s4, $s7
```

- **`slt`** is an arithmetic instruction:
  - Produces 1 if **`a < b`**, 0 otherwise.
  - Uses subtraction: **`(a - b) < 0`** implies **`a < b`**.
  - **`slt`** is 1 when **`(a - b)`** is negative.
    - Use the sign bit to determine if **`(a - b)`** is negative!
  - Result: bits 1 to 31 are always 0.
  - Result: bit 0 is either 0 or 1.
    - But, bit 31 is the sign bit.
    - Need to connect bit 31 to bit 0.

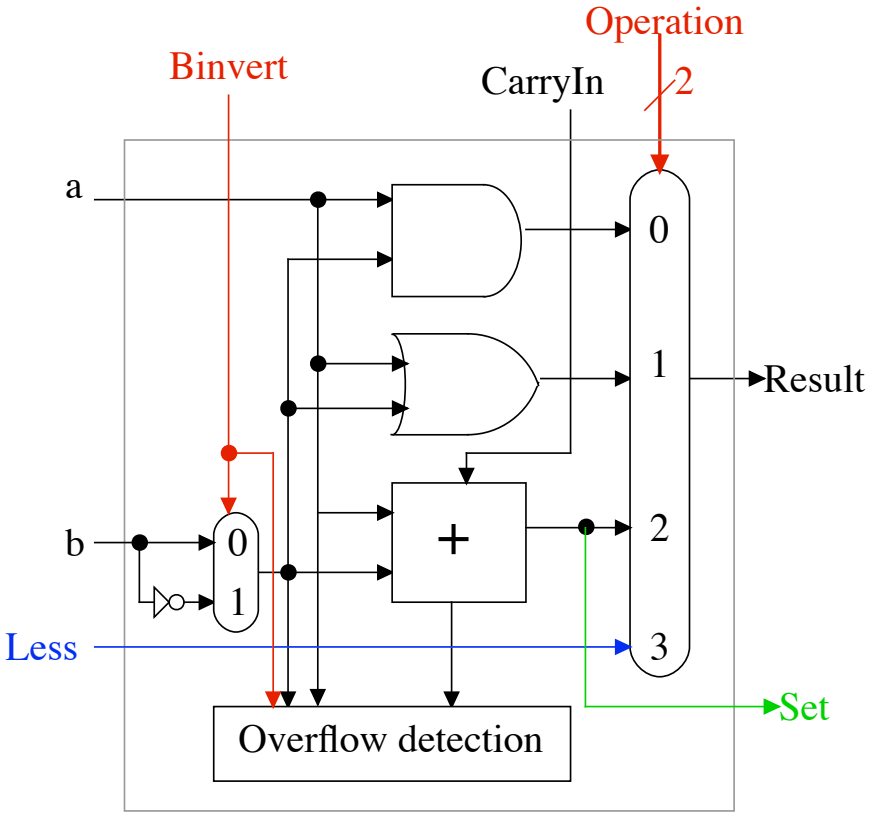
```
slt  $t3, $s4, $s7

($s4 - $s7)
if   bit 31 == 1  then
    $t3 = 1      0000 0000 0000 0000  0000 0000 0000 0001
else
    $t3 = 0      0000 0000 0000 0000  0000 0000 0000 0000
```

Set-less-than (continued):

31 instances of this 1-bit ALU.
Bits 0 to 30.



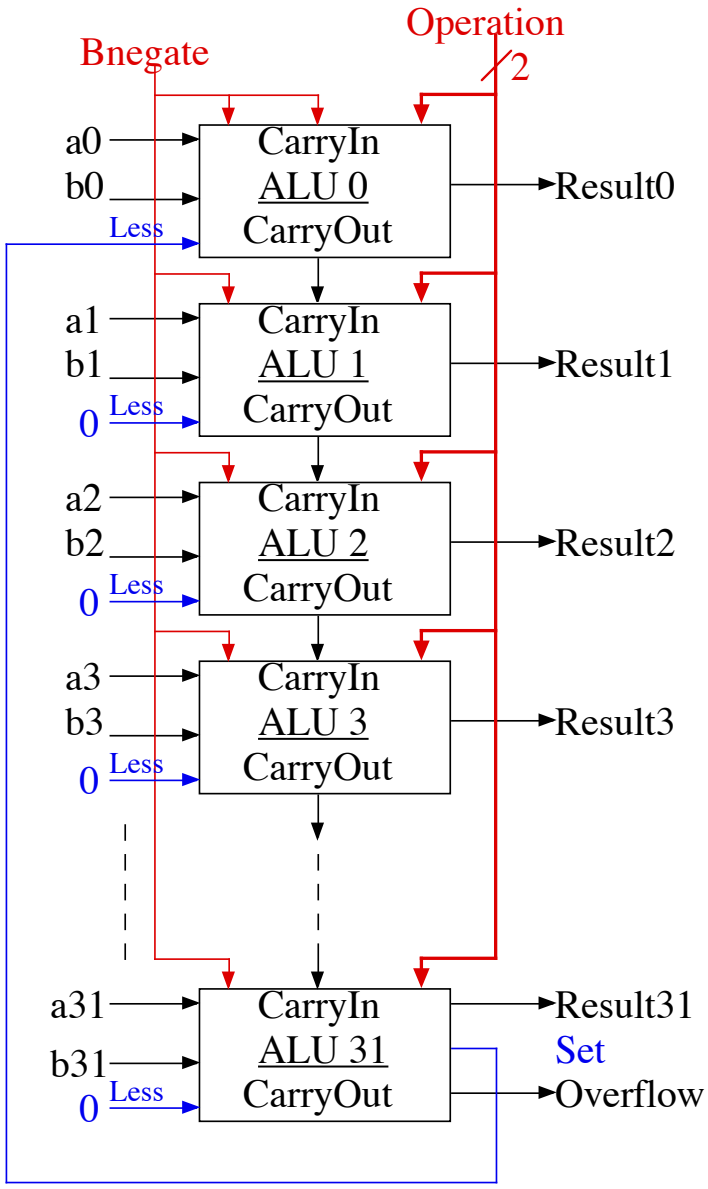
1 instance of this 1-bit ALU.
Bit 31 (most significant bit)



Set-less-than (continued):

- ALU 0 thru ALU 30 are identical units.
- ALU 31 contains overflow detection.
- The Set result for ALU 31 tells us if `(a - b) < 0`.
  - Remember: bit 31 is the sign bit.
  - Connect 0 to Less input for bits 1 to 31.
  - Connect bit 31's Set output to the Less input for bit 0.
    - Allows ALU to return:

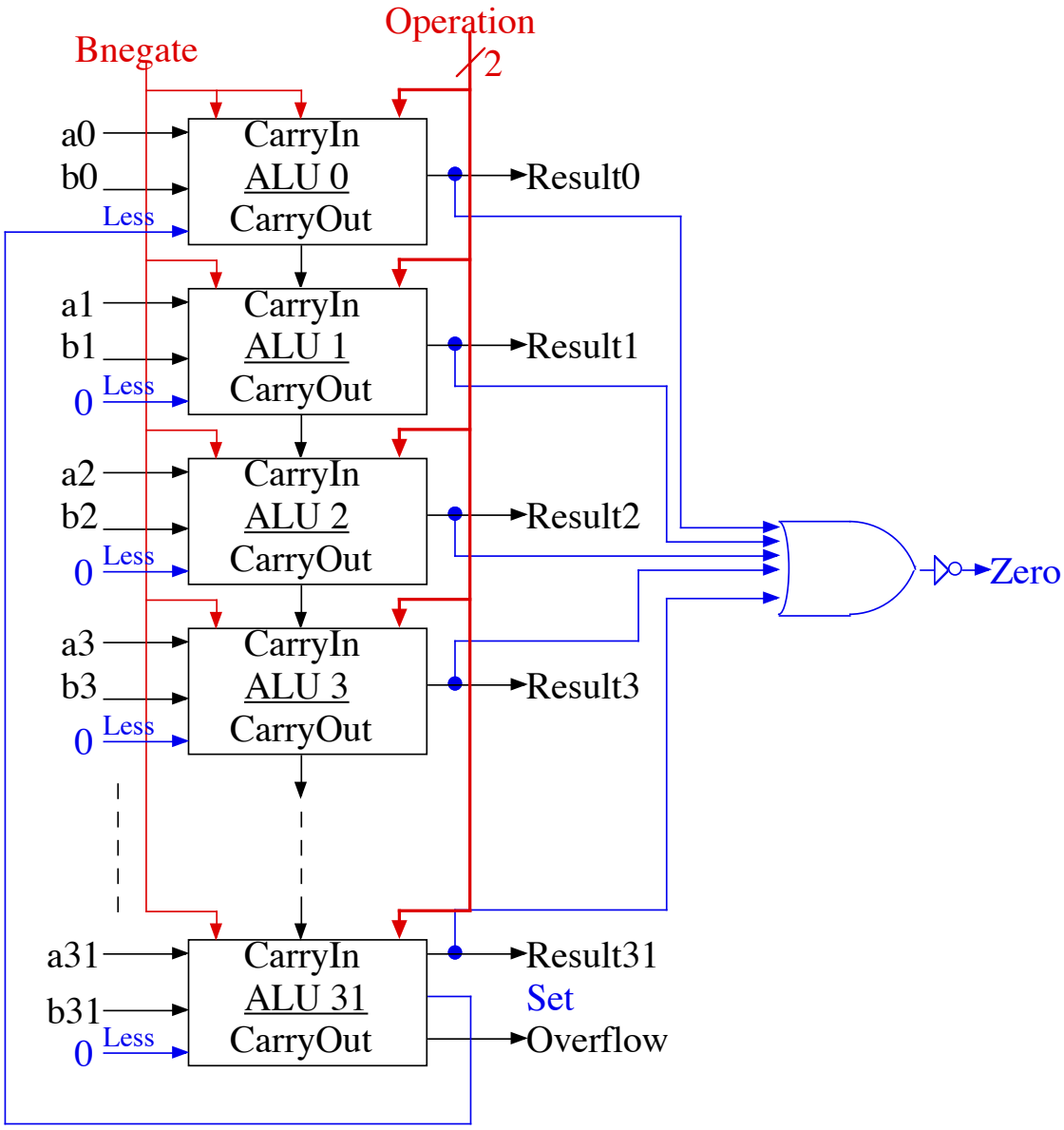
      0000 0000 0000 0000 0000 0000 0000 0001



Test for Equality:

- Use subtraction:
  - (a - b) == 0 implies a == b.

- Why does Zero need the NOT gate?



Control Functions for the ALU:

- Bnegate line:
  - Turned on (1) for sub, slt.
  - Turned off (0) for add, and, or.
- Operation:
  - Four functions: and, or, add, slt.
    - Add with Bnegate turned on gives subtraction.
- Five operations: add, sub, and, or, slt.
  - Three control lines:
    - Bnegate (1 line).
    - Operation (2 lines).
- Three wires can support up to 8 operations.
  - We only have 5.
  - Leaves 3 for future expansion.

ALU control lines		Function
Bnegate	Operation	
0	0 0	and
0	0 1	or
0	1 0	add
1	1 0	subtract
1	1 1	set on less than

The 3 "missing" operations: 1 00, 1 01, 0 11

ALU Summary:

- 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 (fewer is faster).
  - The speed of a circuit is affected by the number of gates in series.
    - On the "critical path", or
    - "deepest level of logic"
  - Our primary focus: Comprehension
    - However, we will take note of
      - Clever changes to organization can improve performance
        - This is similar to finding better algorithms in software.

Faster Addition: Carry Lookahead

Reading: Section C.6 (4th edition), Section B.6 (5th edition).

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition (or to skin a cat)?
  - Two extremes: ripple carry and sum-of-products.
- What is known at the beginning of the problem?
  - a0, a1, a2, …, a31
  - b0, b1, b2, …, b31
  - c0
    - We do <u>not</u> know c1, c2, c3, …, c31. Need to compute these.
- Basic idea: What is known, and not known, in each equation?

$$c_1 = b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0$$

$$c_2 = b_1 \cdot c_1 + a_1 \cdot c_1 + a_1 \cdot b_1$$

$$c_3 = b_2 \cdot c_2 + a_2 \cdot c_2 + a_2 \cdot b_2$$

$$c_4 = b_3 \cdot c_3 + a_3 \cdot c_3 + a_3 \cdot b_3$$

$$c_7 = b_6 \cdot c_6 + a_6 \cdot c_6 + a_6 \cdot b_6$$

```
  ???? ???
  1011 0111
+ 1001 1101
```

Carry Lookahead (continued):

- Sum-of-products: Can you see the ripple?

$c_1 = b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0$ Needs 3 AND gates of 2 inputs each and a 3-input OR gate. (Total = 5 gates)

$c_2 = b_1 \cdot c_1 + a_1 \cdot c_1 + a_1 \cdot b_1$

$= b_1 \cdot (b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0) + a_1 \cdot (b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0) + a_1 \cdot b_1$

$= b_1 \cdot b_0 \cdot c_0 + b_1 \cdot a_0 \cdot c_0 + b_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_0 \cdot c_0 + a_1 \cdot a_0 \cdot c_0 + a_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_1$

Needs 7 AND gates of up to 3 inputs each, plus a 7-input OR gate. (Total = 19 gates)

$c_3 = b_2 \cdot c_2 + a_2 \cdot c_2 + a_2 \cdot b_2$

$= b_2 \cdot (b_1 \cdot b_0 \cdot c_0 + b_1 \cdot a_0 \cdot c_0 + b_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_0 \cdot c_0 + a_1 \cdot a_0 \cdot c_0 + a_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_1)$

$+ a_2 \cdot (b_1 \cdot b_0 \cdot c_0 + b_1 \cdot a_0 \cdot c_0 + b_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_0 \cdot c_0 + a_1 \cdot a_0 \cdot c_0 + a_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_1) + a_2 \cdot b_2$

$= b_2 \cdot b_1 \cdot b_0 \cdot c_0 + b_2 \cdot b_1 \cdot a_0 \cdot c_0 + b_2 \cdot b_1 \cdot a_0 \cdot b_0 + b_2 \cdot a_1 \cdot b_0 \cdot c_0 + b_2 \cdot a_1 \cdot a_0 \cdot c_0 + b_2 \cdot a_1 \cdot a_0 \cdot b_0 + b_2 \cdot a_1 \cdot b_1$

$+ a_2 \cdot b_1 \cdot b_0 \cdot c_0 + a_2 \cdot b_1 \cdot a_0 \cdot c_0 + a_2 \cdot b_1 \cdot a_0 \cdot b_0 + a_2 \cdot a_1 \cdot b_0 \cdot c_0 + a_2 \cdot a_1 \cdot a_0 \cdot c_0 + a_2 \cdot a_1 \cdot a_0 \cdot b_0$

$+ a_2 \cdot a_1 \cdot b_1 + a_2 \cdot b_2$

Needs 15 AND gates of up to 4 inputs each, plus a 15-input OR gate. (Total = 41 + 14 = 55 gates)

$c_4 = b_3 \cdot c_3 + a_3 \cdot c_3 + a_3 \cdot b_3$

etc.

- Expensive! Why?

Carry Lookahead (continued):

- $c_1 = g_0 + p_0 \cdot c_0 = a0 \cdot b0 + (a0+b0) \cdot c0$

  $= a_0 \cdot b_0 + a_0 \cdot c_0 + b_0 \cdot c_0$

  Note: **all** of these are known at the start of the problem.

$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0)$

$= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$ *Equation A*

$= a_1 \cdot b_1 + (a_1 + b_1) \cdot a_0 \cdot b_0 + (a_1 + b_1) \cdot (a_0 + b_0) \cdot c_0$

$= a_1 \cdot b_1 + a_1 \cdot a_0 \cdot b_0 + b_1 \cdot a_0 \cdot b_0 +$

$a_1 \cdot (a_0 + b_0) \cdot c_0 + b_1 \cdot (a_0 + b_0) \cdot c_0$

$= a_1 \cdot b_1 + a_1 \cdot a_0 \cdot b_0 + b_1 \cdot a_0 \cdot b_0 + a_1 \cdot a_0 \cdot c_0 +$

$a_1 \cdot b_0 \cdot c_0 + b_1 \cdot a_0 \cdot c_0 + b_1 \cdot b_0 \cdot c_0$

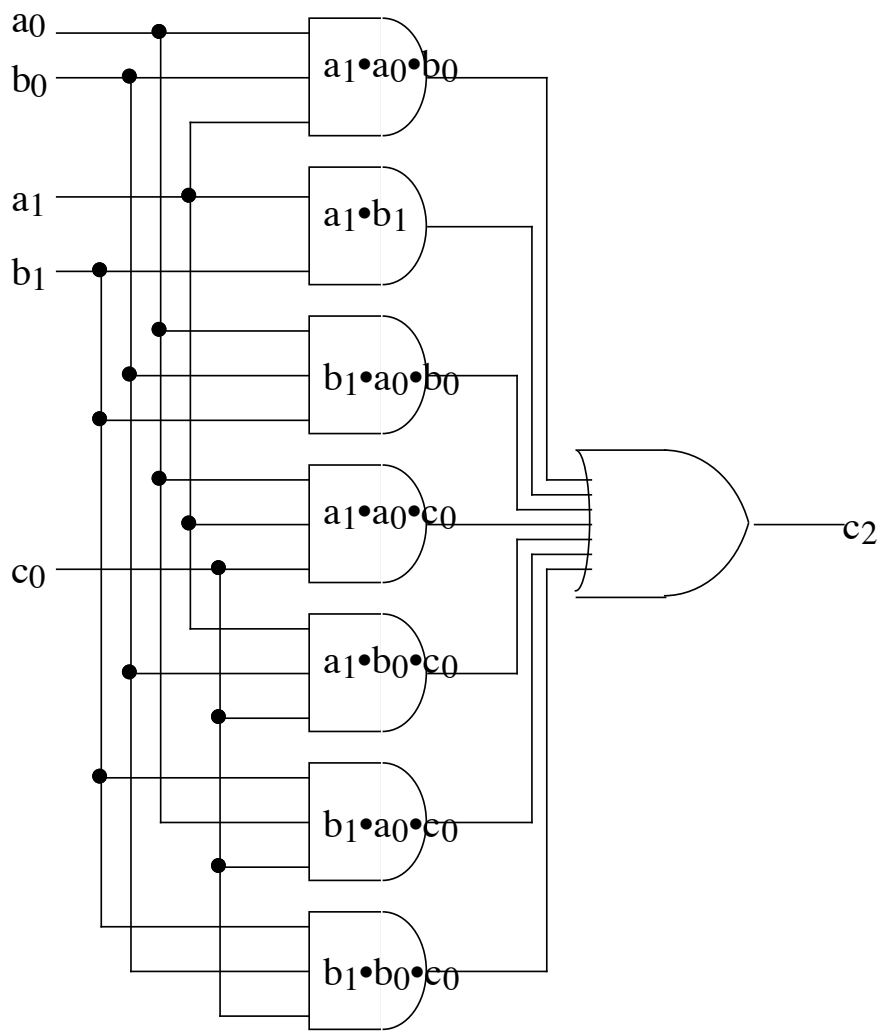
7 AND gates of up to 3 inputs each.

Total number of 2-input gates:

AND gates: 13

OR gates: 6

Total: 19



Carry Lookahead (continued):

- It is possible to continue the sum-of-products technique.
  - But, it gets expensive -- exponential growth in the number of gates needed at each stage.
- Here is an "in-between" approach:
- Motivation:
  - If we did not know the value of carry-in, what could we do?
  - When would we always generate a carry out? $g_i = a_i \cdot b_i$
  - When would we propagate the carry out? $p_i = a_i + b_i$
    - Example:

```
g:      0 0 1 0   0 0 1 0
p:      1 0 1 1   1 1 1 1
carry: 0 1 1 1   1 1 0 0
a:      1 0 1 1   0 0 1 1
b:    + 0 0 1 0   1 1 1 0
result: 1 1 1 0   0 0 0 1
```

$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$

$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

- $c_1 = g_0 + p_0 \cdot c_0 = g_0 + (a_0 + b_0) c_0$

  $= a_0 \cdot b_0 + a_0 \cdot c_0 + b_0 \cdot c_0$ Note: **all** of these items are known at the start of the problem.

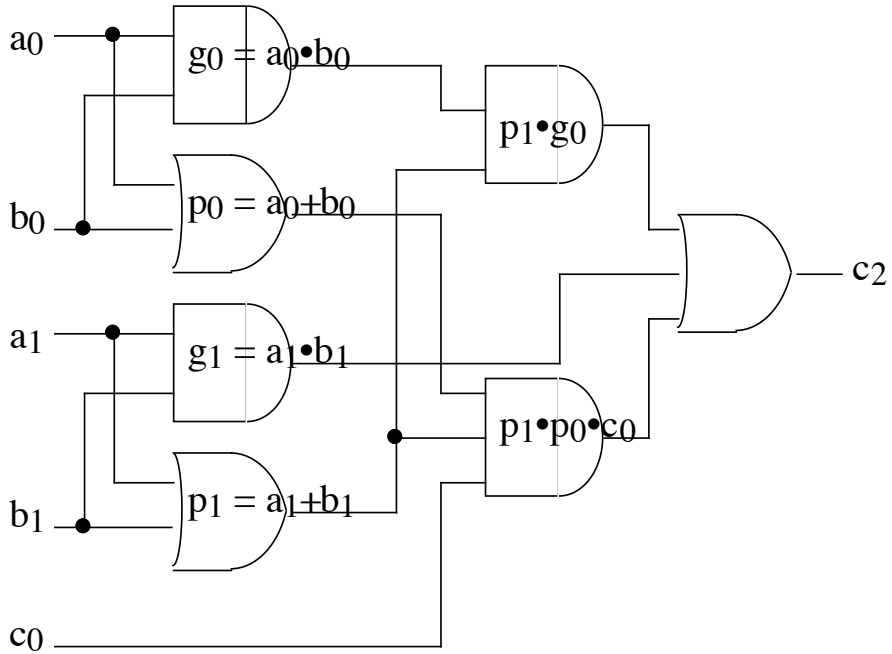
Carry Lookahead (continued):

- But, can optimize by treating **g0**, **p0**, **g1**, **p1** as logical units.

$$c_2 = g_1 + p_1 \bullet c_1 = g_1 + p_1 \bullet (g_0 + p_0 \bullet c_0)$$

$$= g_1 + p_1 \bullet g_0 + p_1 \bullet p_0 \bullet c_0 \qquad \textit{Equation A}$$

- Equation A becomes:
  - 4 AND gates:
    - two input gates: g_0, g_1, $p_1 \bullet g_0$
    - three input gate: $p_1 \bullet p_0 \bullet c_0$
  - 3 OR gates:
    - two input gates: **p0**, **p1**
    - three input gate: c_2
- By comparison, the figure on the previous slide, in effect, had g_0 twice.
  - Appears only once here.



Total number of 2-input gates:

AND gates: 5

OR gates: 4

Total: 9

Carry Lookahead (continued):

- The gate count is not as many as stated by the previous slide:

  Total number of 2-input gates:

  AND gates: 5

  OR gates: 4

  Total: 9

- To implement Carry Lookahead, we only need to add:

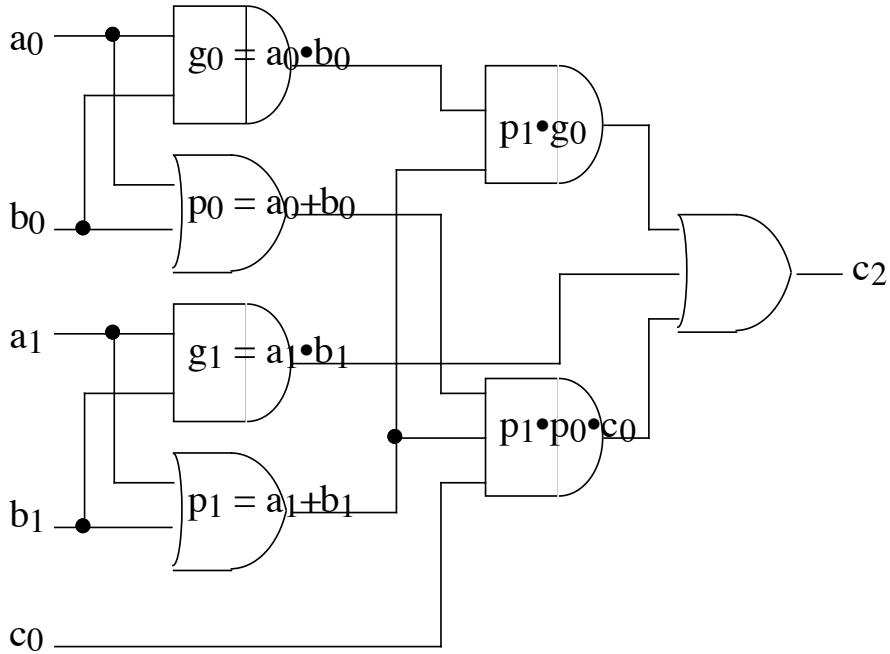
  Total number of 2-input gates:

  AND gates: 3

  OR gates: 2

  Total: 5

- Why?



Carry Lookahead (continued):

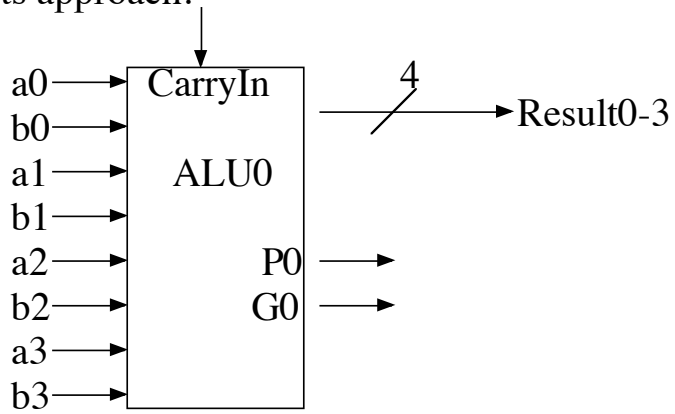
- Continuing, we can compute c3:

$$c_3 = g_2 + p_2 \bullet c_2 = g_2 + p_2 \bullet (g_1 + p_1 \bullet g_0 + p_1 \bullet p_0 \bullet c_0)$$

$$= g_2 + p_2 \bullet g_1 + p_2 \bullet p_1 \bullet g_0 + p_2 \bullet p_1 \bullet p_0 \bullet c_0$$

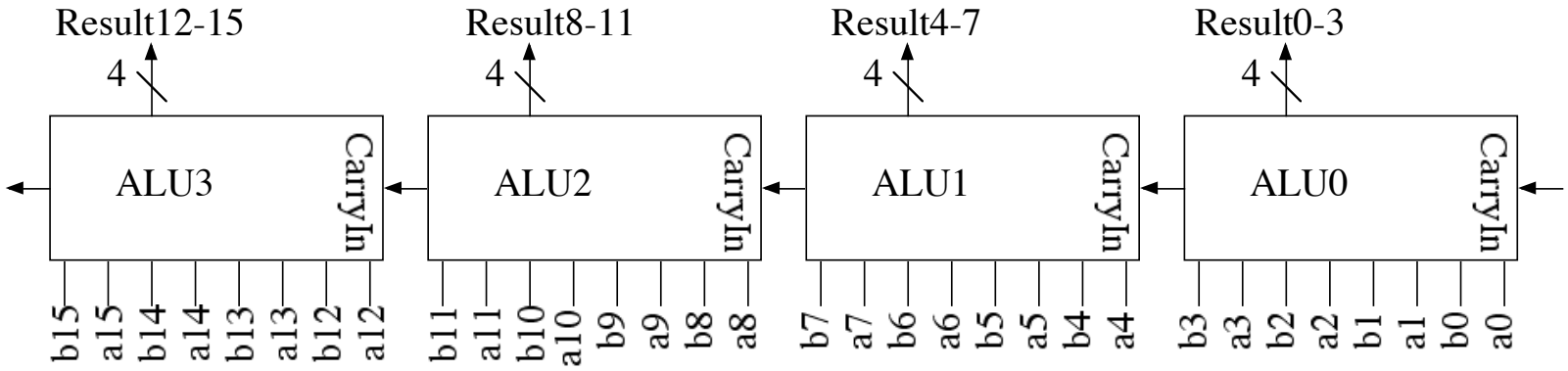
  - 4 AND gates of (up to) 4 inputs each, plus 3 AND (g_0, g_1, g_2), plus 3 OR (p_0, p_1, p_2) gates of 2 inputs each.
- ditto for c_4.

- Better!
  - More complicated than the ripple adder.
  - But, the growth in complexity is not as great as the sum-of-products approach.
  - Compromise between performance and complexity.
- Called a *Carry Lookahead Adder*, or CLA.
- There is a practical limit to the number of inputs to a CLA.
  - We will assume 4-bits.



"Super" Carry Lookahead:

- Another level of abstraction is needed:
- Consider a 16-bit adder, constructed using 4-bit CLA's.
  - Use 4 CLA's to handle adding the 16-bit values 4 bits at a time.
  - Need to compute the carry from one 4-bit CLA to the next.
    - Could do this using a ripple add.
    - An improvement over the original ripple adder (why?).
    - We can do better.



"Super" Carry Lookahead (continued):

- "Super" bits to the rescue!
- Compute "super" propagate bits:

$$P0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

$$P1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$

$$P2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$

$$P3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$

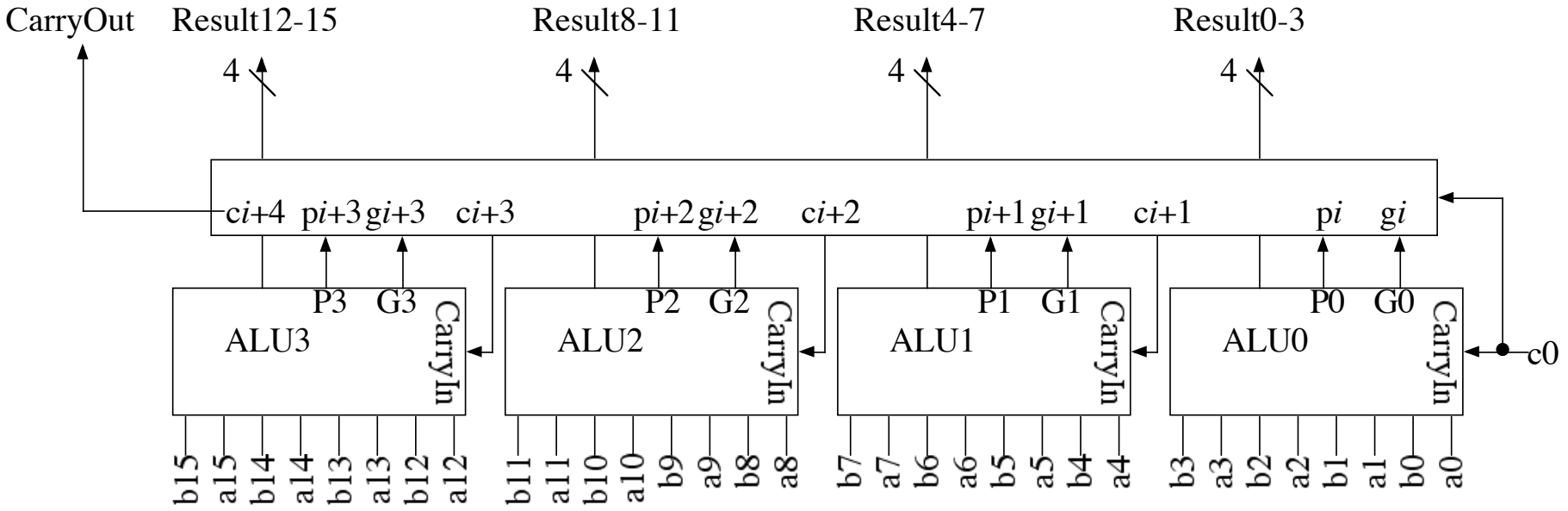
- Compute "super" generate bits:

$$G0 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$

$$G1 = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4$$

$$G2 = g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8$$

$$G3 = g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}$$



"Super" Carry Lookahead (continued):

- Example from pages C-44 to C-46 (4th edition):
- Determine the **gi**, **pi**, **Pi**, and **Gi** values of these two 16-bit numbers:

  a: 0001 1010 0011 0011$_{two}$
  b: 1110 0101 1110 1011$_{two}$

  Also, what is **CarryOut15** (**C4**)?

- Solution:

  a: 0001 1010 0011 0011
  b: 1110 0101 1110 1011

  gi: 0000 0000 0010 0011 determined from ai • bi
  pi: 1111 1111 1111 1011 determined from ai + bi

- Compute the "super" propagates:

  P0 = p3 • p2 • p1 • p0 = 1 • 0 • 1 • 1 = 0
  P1 = p7 • p6 • p5 • p4 = 1 • 1 • 1 • 1 = 1
  P2 = p11 • p10 • p9 • p8 = 1 • 1 • 1 • 1 = 1
  P3 = p15 • p14 • p13 • p12 = 1 • 1 • 1 • 1 = 1

- Compute the "super" generates:

  G0 = g_3 + p_3•g_2 + p_3•p_2•g_1 + p_3•p_2•p_1•g_0 = 0 + 1•0 + 1•0•1 + 1•0•1•1 = 0+0+0+0 = 0
  G1 = g_7 + p_7•g_6 + p_7•p_6•g_5 + p_7•p_6•p_5•g_4 = 0 + 1•0 + 1•1•1 + 1•1•1•0 = 0+0+1+0 = 1
  G2 = g_{11} + p_{11}•g_{10} + p_{11}•p_{10}•g_9 + p_{11}•p_{10}•p_9•g_8 = 0 + 1•0 + 1•1•0 + 1•1•1•0 = 0+0+0+0 = 0
  G3 = g_{15} + p_{15}•g_{14} + p_{15}•p_{14}•g_{13} + p_{15}•p_{14}•p_{13}•g_{12} = 0 + 1•0 + 1•1•0 + 1•1•1•0 = 0+0+0+0 = 0

"Super" Carry Lookahead (continued):

- From the previous slide:

```
P0 = 0   P1 = 1   P2 = 1   P3 = 1
G0 = 0   G1 = 1   G2 = 0   G3 = 0
```

- Which gives:

```
C1 = G0 + P0•c0
   =  0 +  0•0
   =  0 +   0                                          = 0 = C1
C2 = G1 + P1•G0 + P1•P0•c0
   =  1 +  1•0  +  1•1•0
   =  1 +   0   +    0                                 = 1 = C2
C3 = G2 + P2•G1 + P2•P1•G0 + P2•P1•P0•c0
   =  0 +  1•1  +  1•1•0   +   1•1•0•0
   =  0 +   1   +    0     +      0                    = 1 = C3
C4 = G3 + P3•G2 + P3•P2•G1 + P3•P2•P1•G0 + P3•P2•P1•P0•c0
   =  0 +  1•0  +  1•1•1   +  1•1•1•0    +  1•1•1•0•0
   =  0 +   0   +    1     +     0       +      0      = 1 = C4
```

- Note: All of the C's can be calculated at the *same* time!
- The sequence then becomes:
  - Calculate all of the **`P's`** and **`G's`** at the same time.
  - Calculate all of the **`C's`** at the same time. Now, the carry-in to each CLA is known.
  - Calculate all of the result bits at the same time.

"Super" Carry Lookahead (continued):

```
  0110 1001 1110 0010
+ 1001 1011 0100 0011

p 1111 1011 1110 0011
g 0000 1001 0100 0010
P  1    0    0    0
G  0    1    1    0
1    1    1    0        The carry-in to each 4-bit adder
  0110 1001 1110 0010
+ 1001 1011 0100 0011
```

"Super" Carry Lookahead (continued):

- Suppose we don't know the bits of the two numbers, but we do know all the **p** and **g** bits:

```
  xxxx xxxx xxxx xxxx
? xxxx xxxx xxxx xxxx

p 1111 1011 1110 0011
g 0000 1001 0100 0010
P  1    0    0    0
G  0    1    1    0
1 1111  111 1  0  1??
  xxxx 10x1 x1x0 001x
- xxxx 10x1 x1x0 001x
  0000 0101 0010 01??
  0000 0101 0010 0101  addition, no overflow
  0000 0101 0010 0110  subtraction, maybe overflow, maybe not
```

Multiplication

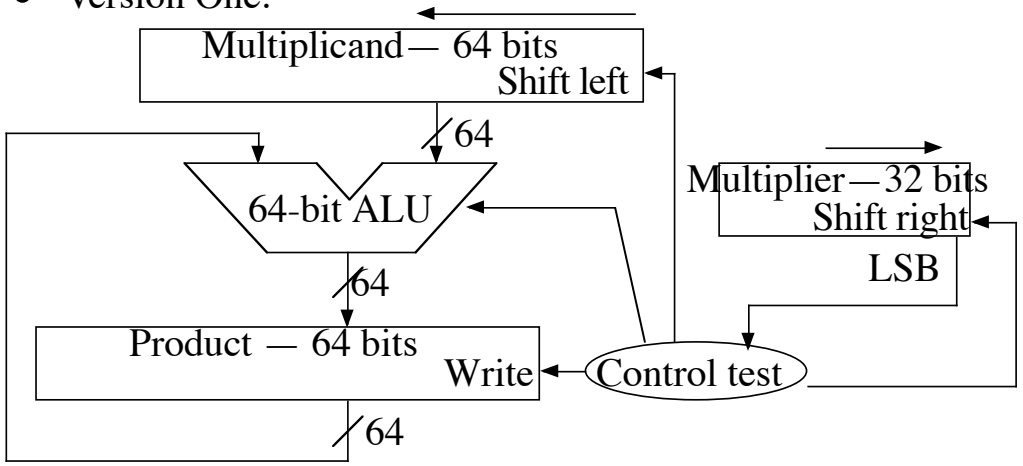
Reading: Section 3.3: pages 230-236 (4th edition), pages 183-188 (5th edition).

- More complicated than addition.
  - Accomplished with shifting and addition.
- Takes more time and requires more area on the CPU (that is, more transistors).
- Will look at 3 versions, all based on the grade school multiplication algorithm:

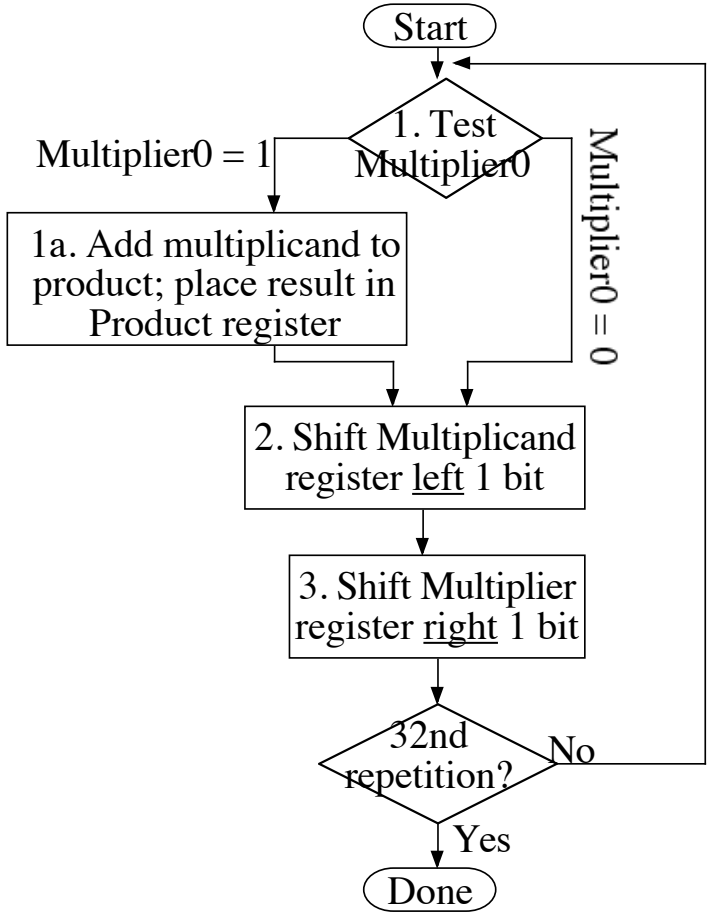
```
          01100111
        * 01001101
----------------
          01100111
         00000000
        01100111
       01100111
      00000000
     00000000
    01100111
   00000000
----------------
 0001111011111011
```

Multiplication (continued):

- Version One:

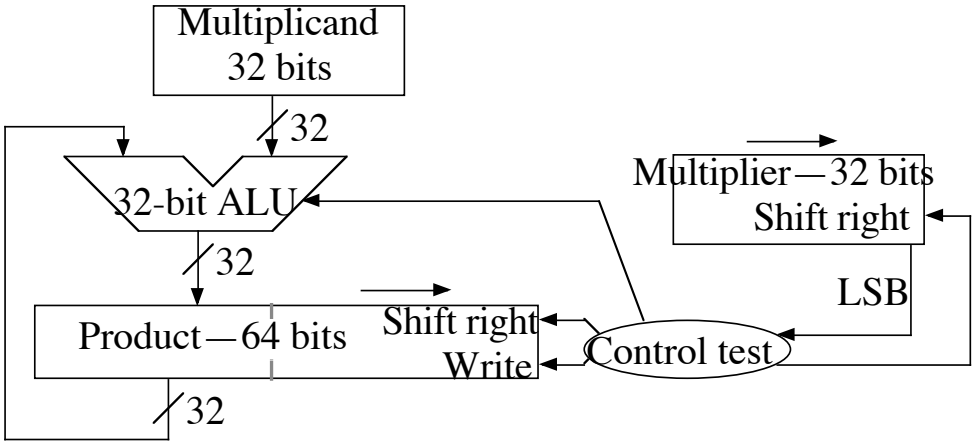


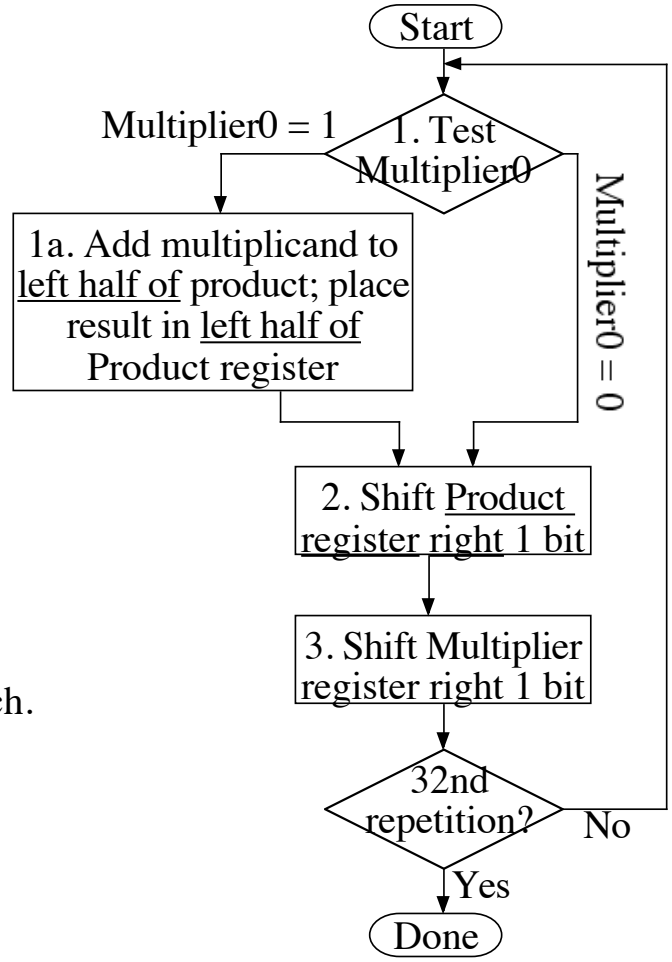
- Multiplicand:
  - Stored in right half (right 32-bits) of register.
  - Shifted left one bit on each iteration.
- Multiplier:
  - Shifted right one bit.
  - Look at least-significant bit only on each iteration.
- Product:
  - Potentially a 64-bit answer.



Multiplication (continued):

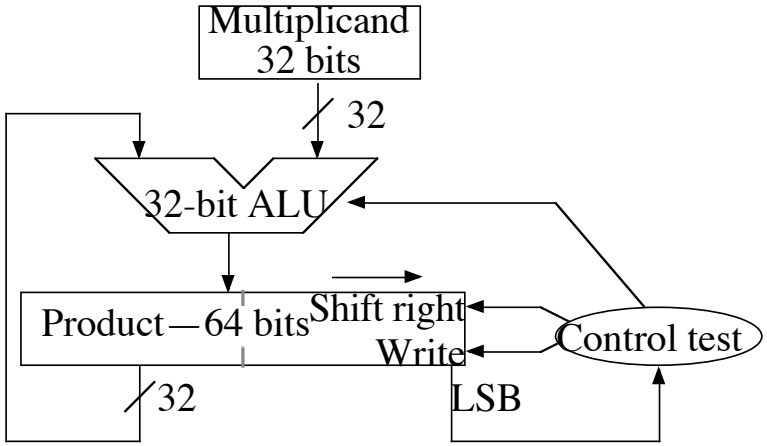
- Version Two:

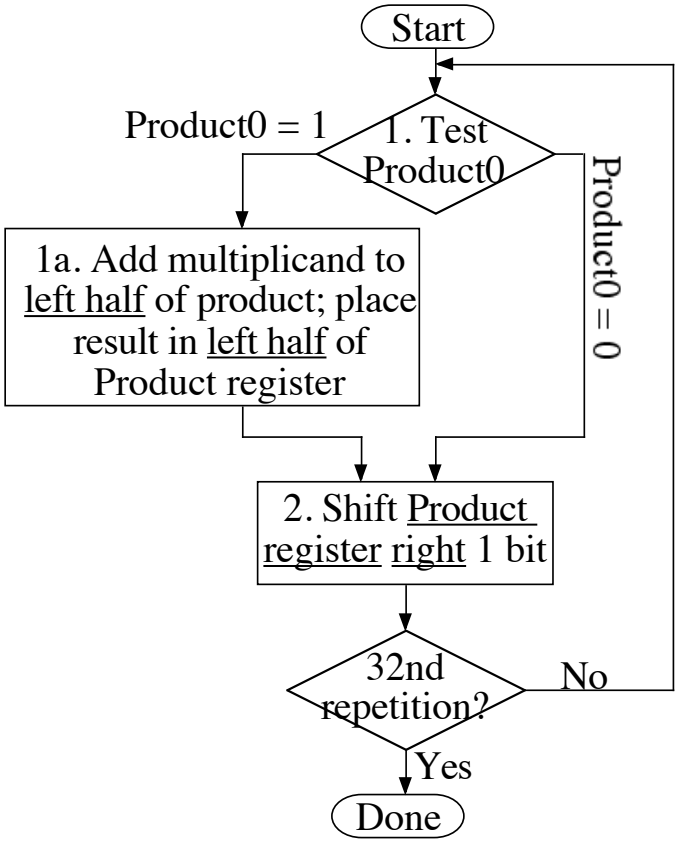




- Multiplicand register, ALU, and Multiplier register are now 32-bits each.
- Product is now shifted right.

Multiplication (continued):





- Separate Multiplier register is gone.
- Put Multiplier initially in the right-half of the product register.
  - As the product is computed and shifted right, the multiplier will shift right as well.
  - Right-most bit (LSB) of Product register is the bit tested in the first if.
  - Step 3 is gone.

Multiplication in MIPS:

Reading: Spim Appendix, page 53.

- Two versions:
  - Signed multiplication: **`mult`**

```
mult   $s1, $s2   # multiply signed $s1 * $s2
```

  - Unsigned multiplication: **`multu`**

```
multu  $s3, $s4   # multiply unsigned $s3 * $s4
```

- Note: No destination register!!
  - The product goes into two special-purpose registers (not part of the 32 general-purpose registers).
    - **`lo`** holds the lower 32-bits of the product.
    - **`hi`** holds the upper 32-bits of the product.
  - Use **`mflo`** (move from **`lo`**) and **`mfhi`** (move from **`hi`**)

```
mflo   $s1        # move 32 bit value from lo to $s1
mfhi   $s2        # move 32 bit value from hi to $s2
```

- Notes:
  - The programmer (that would be you!) must test for overflow — not done by **`mult`**.
  - Pseudo-instructions are available in MIPS that specify a destination register: **`mul`**, **`mulo`**, **`mulou`**.

Division:

- Will skip Section 3.4 (4th and 5th editions), Division.
- But, take note of MIPS division instructions:
  - "Divide in MIPS"
    - Pages 241-242 (4th edition).
  - Page 52 in Spim Appendix: **`div`**, **`divu`** instructions.
- Analogous to multiply.
- Register **`lo`** holds the quotient.
- Register **`hi`** holds the remainder.

Summary:

- Computer arithmetic is constrained by limited precision.
  - Limited range of values for integers and floating-point.
  - Floating-point further limited by fractional part.
- Bit patterns have no inherent meaning but standards do exist.
  - Two's complement.
  - IEEE 754 floating-point.
- Computer instructions determine "meaning" of the bit patterns.
- Performance and accuracy are important; thus, there are many complexities in real machines.
- Basics of Gates and Boolean Algebra.
- Basics of integer ALU.