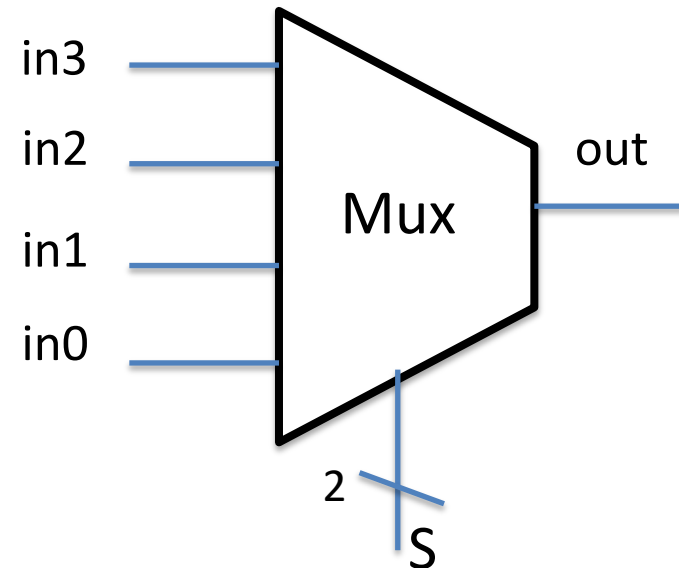
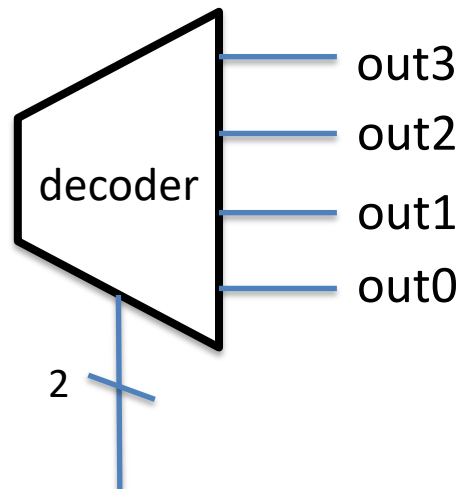


Building an ALU

Jinyang Li

What we've learnt so far

- Basic logic design
 - Logic circuits == Boolean expressions
- How to build a combinatorial logic circuit
 - Specify the truth table
 - Output is the sum of products (implemented in PLA, programmable logic array)
- Common CL
 - Decoder
 - Multiplexer

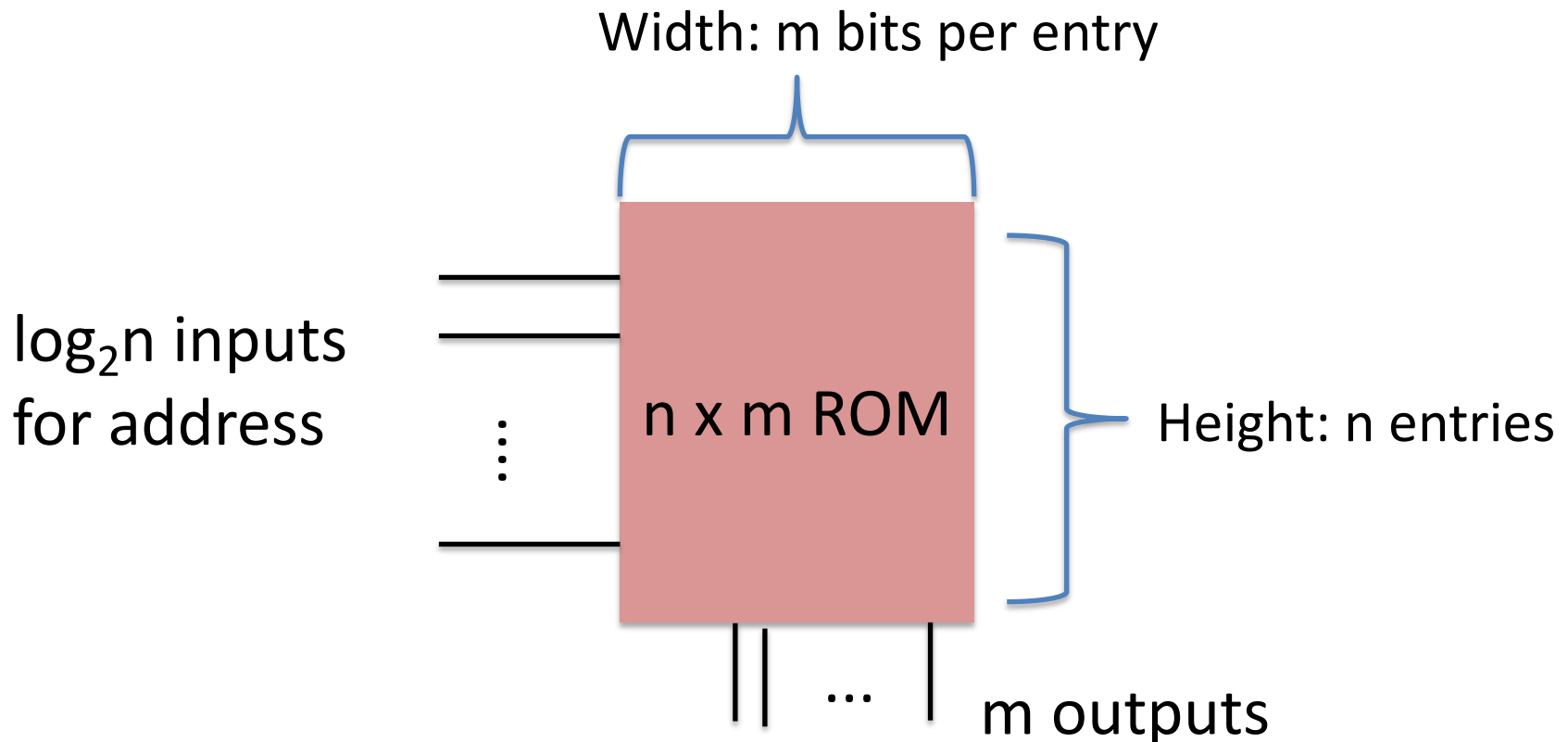


Lesson plan

- ROM (another way to implement CL)
- ALU
 - Logical ops: AND/OR
 - Arithmetic ops: addition, subtraction...

ROM (read-only memory)

- A combinatorial component for storing (fixed) data
- Programmed in the factory or field



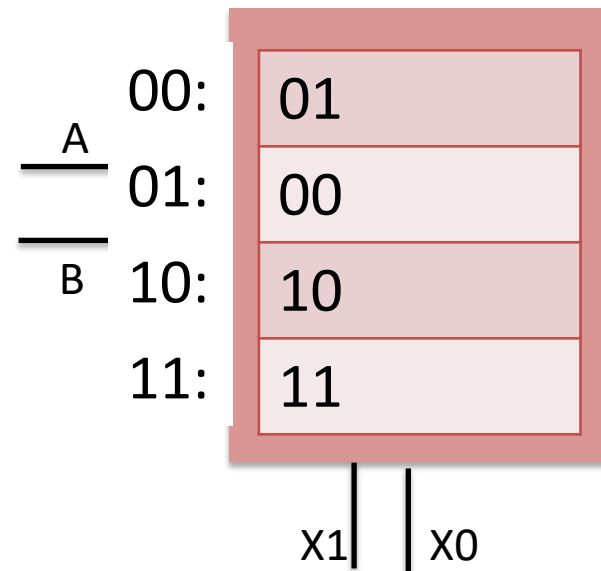
ROM (read-only memory)

- A $n \times m$ ROM can store the truth table for m functions defined on $\log_2 n$ variables.

$$X_1 = A$$

$$X_0 = \bar{A} \cdot \bar{B} + A \cdot B$$

A	B	X1	X0
0	0	0	1
0	1	0	0
1	0	1	0
1	1	1	1

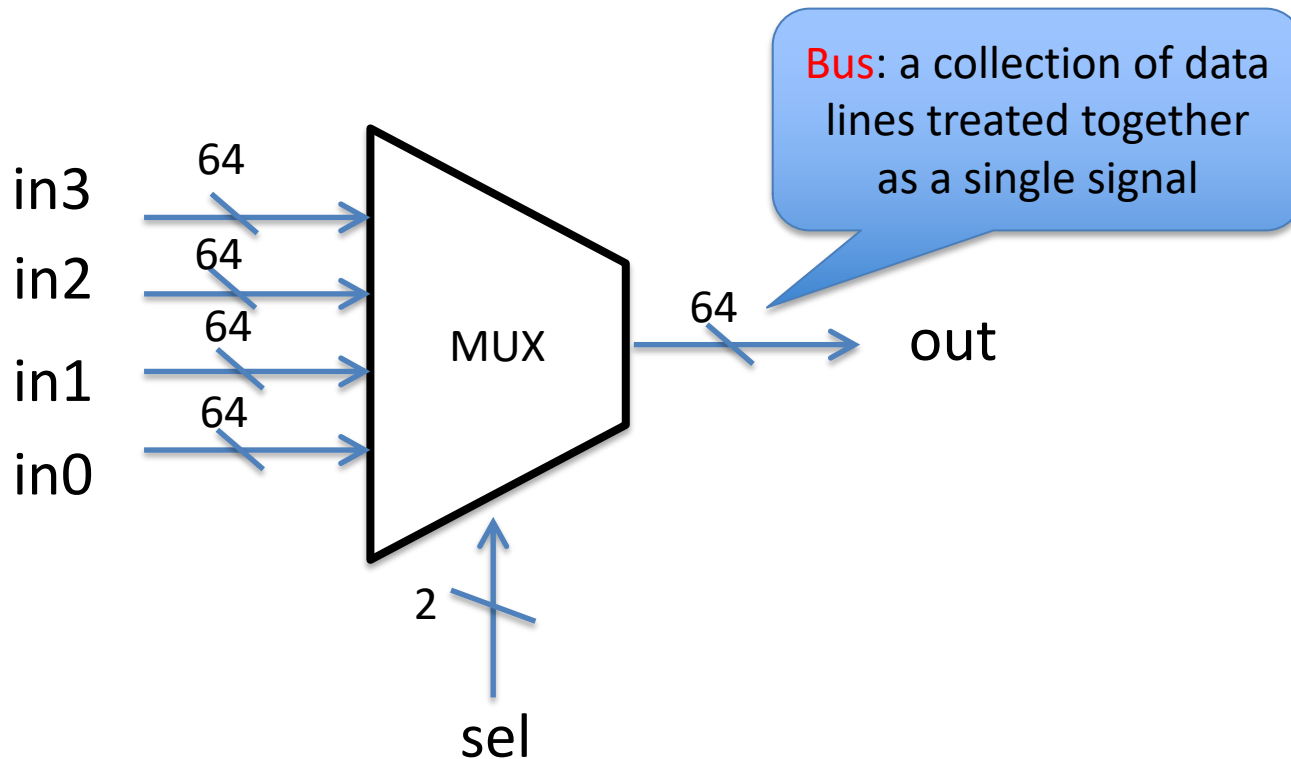


ROM (read-only memory)

- Both ROM and PLA can impl. boolean functions
- ROM is not as efficient for sparse functions
 - # of entries grows exponentially with inputs
- ROM is easier to change if function changes

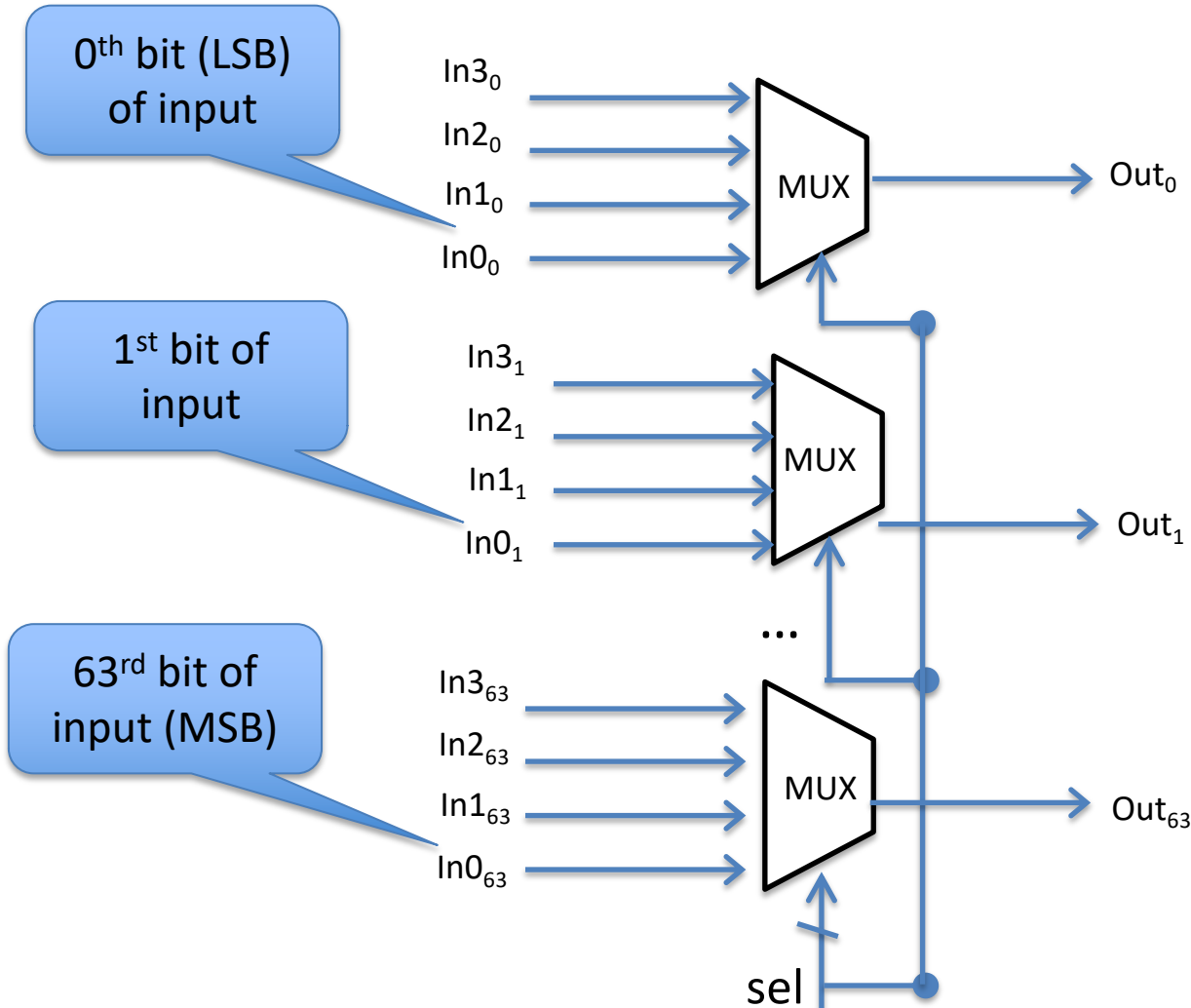
Array of logic elements

- So far, our circuits work on 1-bit inputs/outputs
- How to build circuits with n-bit inputs/outputs?



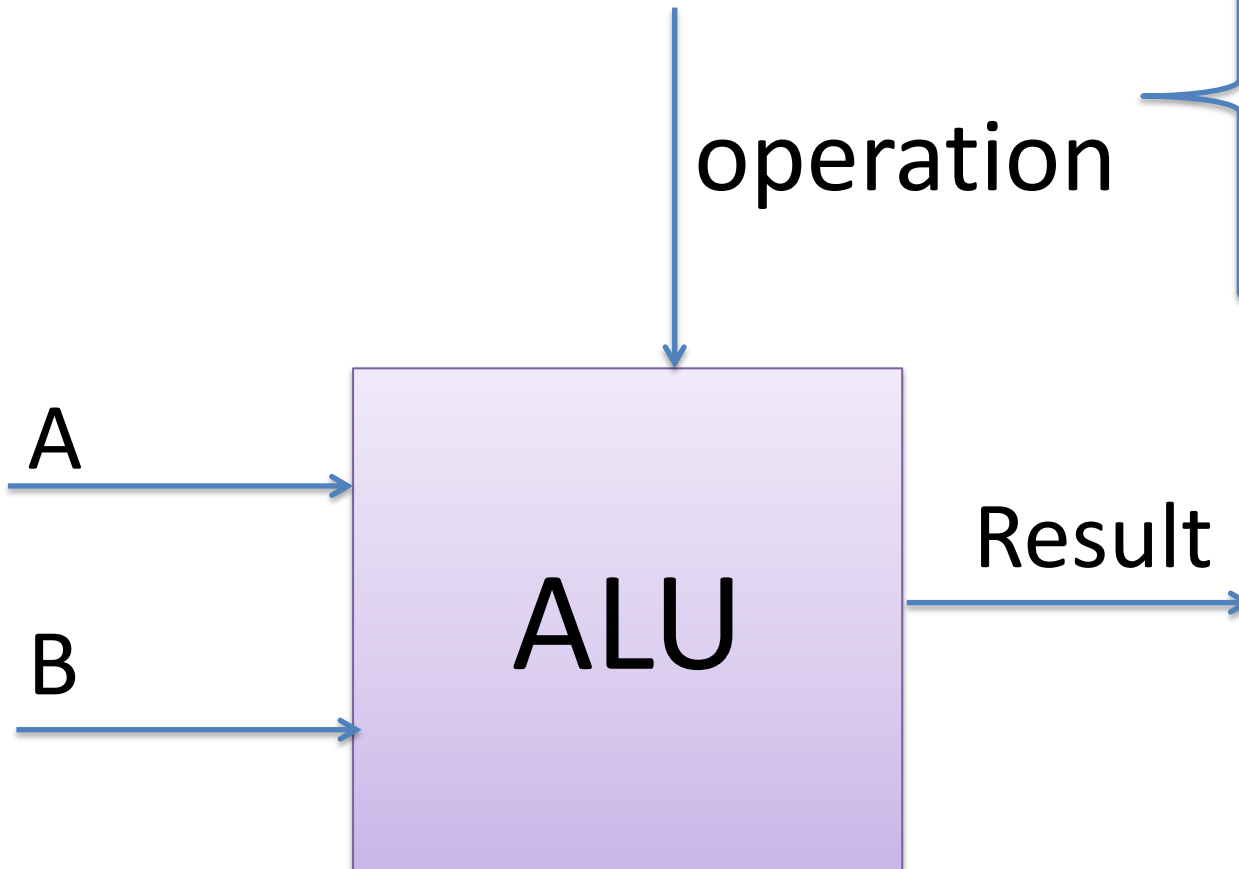
Array of logic elements

- 64-bit multiplexor: an array of 64 1-bit multiplexors



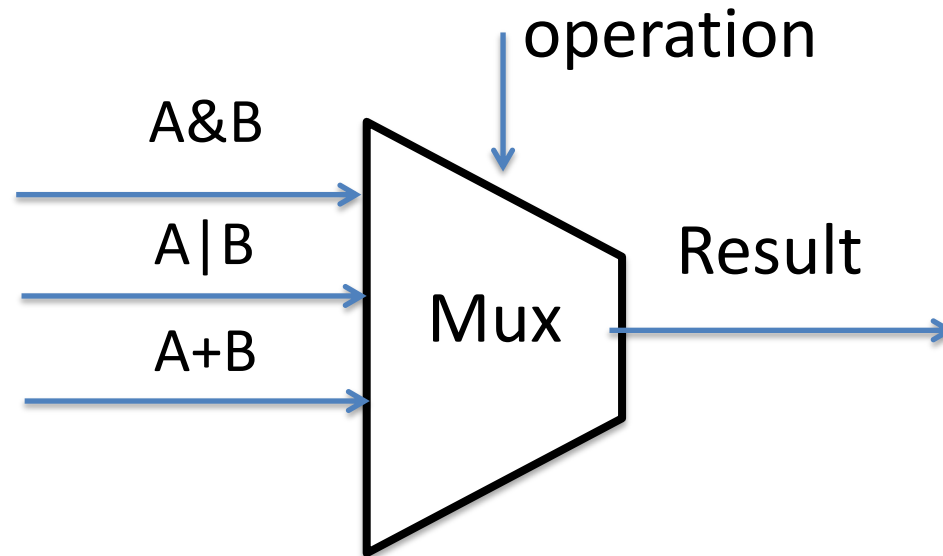
ALU overview

Example

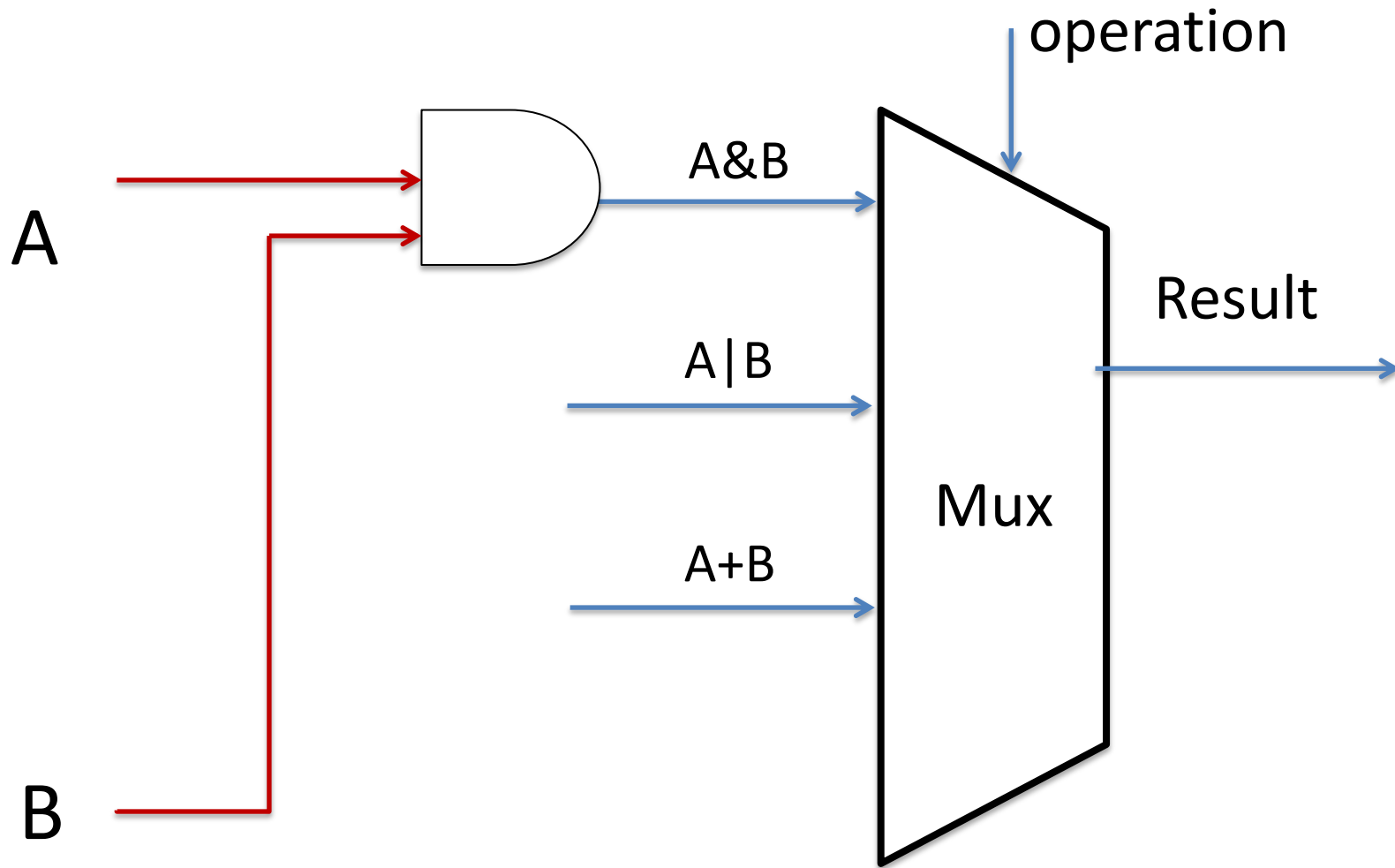


Op	
00	A & B
01	A B
10	A + B

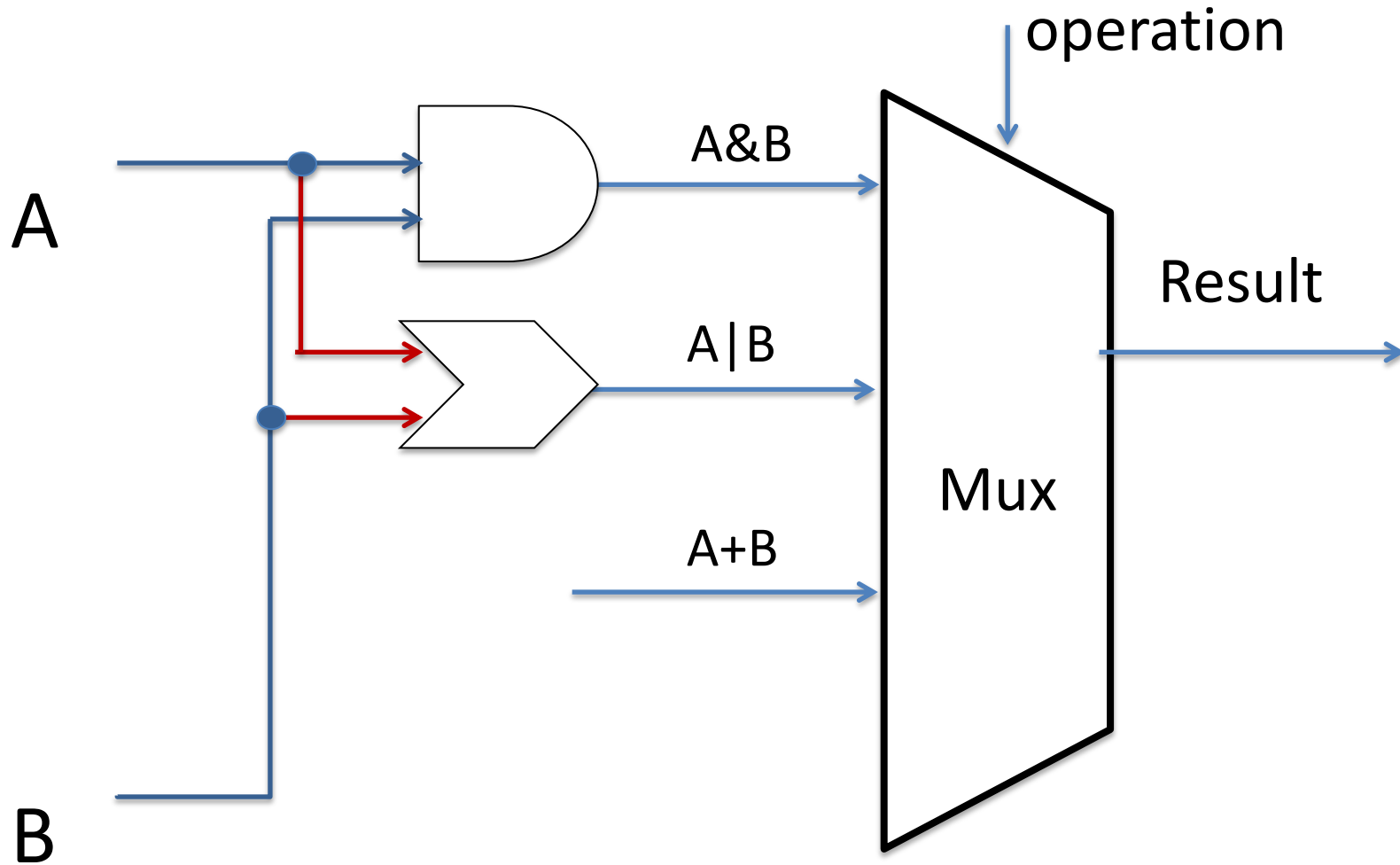
Implementing ALU: AND



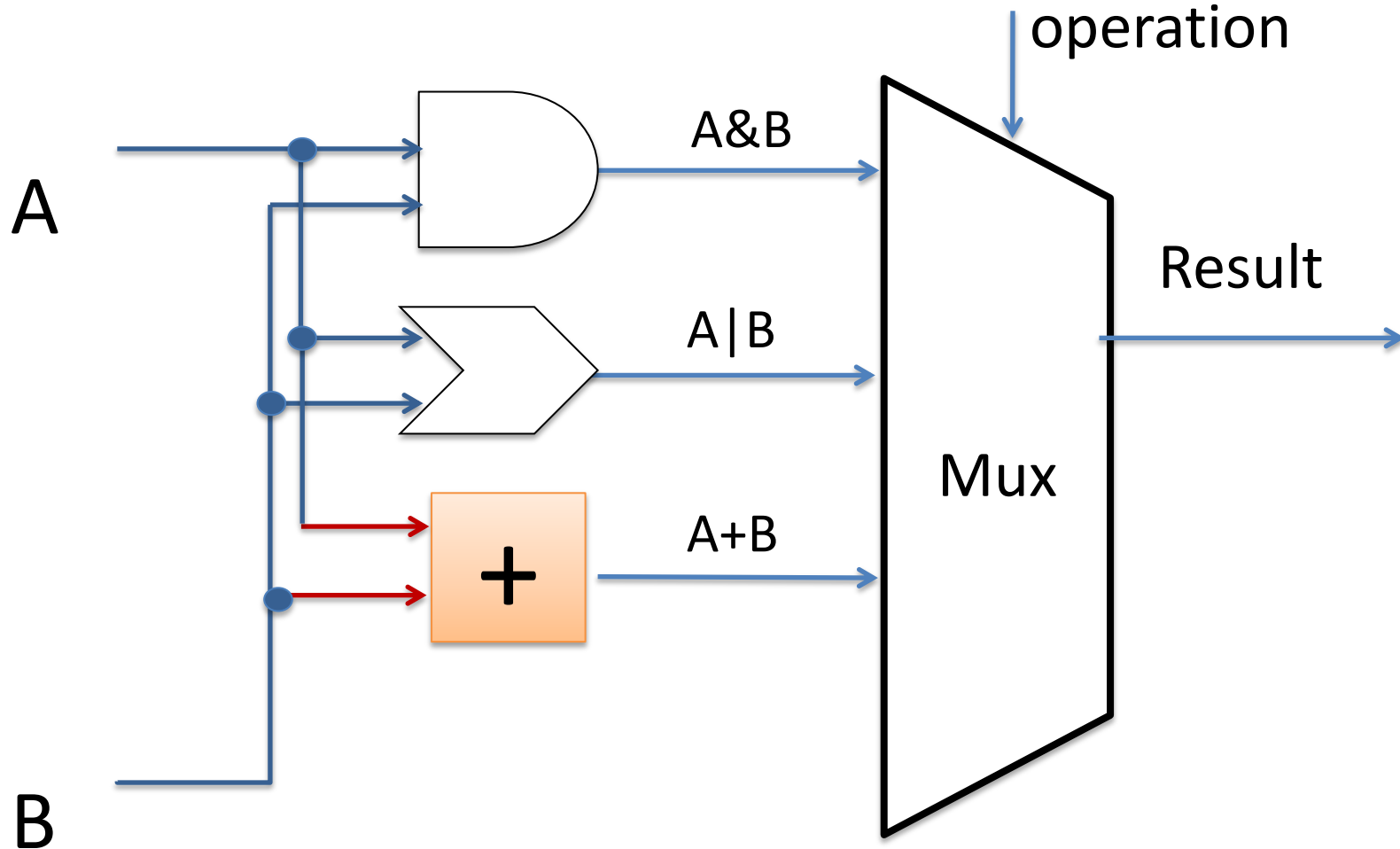
Implementing ALU: AND



Implementing ALU: OR

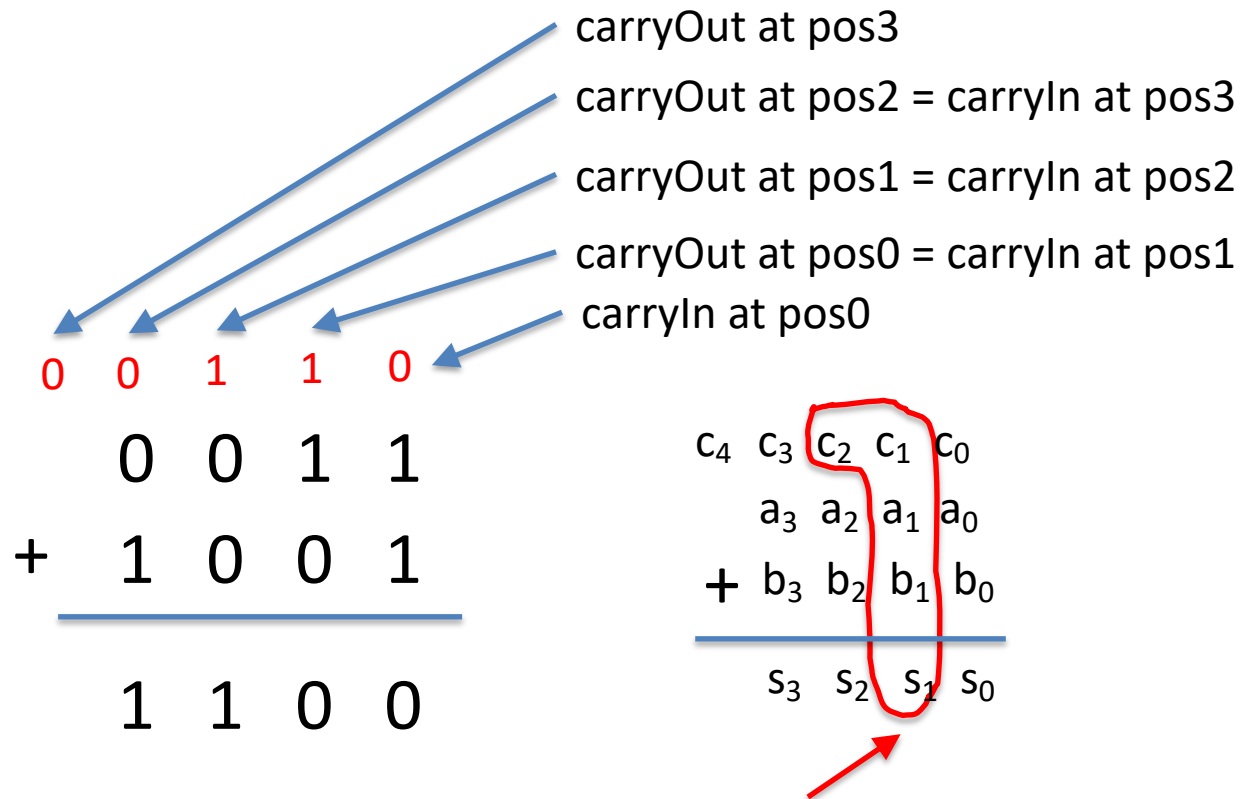


Implementing ALU: adder



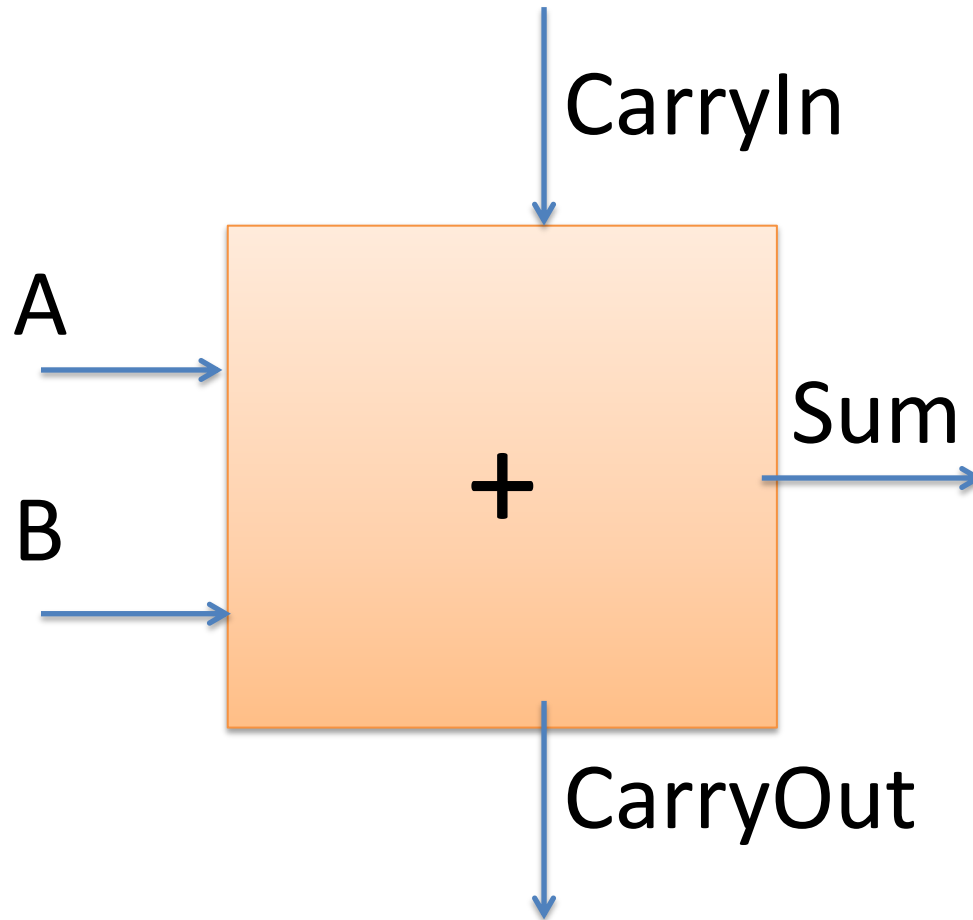
Implementing the adder: 1-bit adder

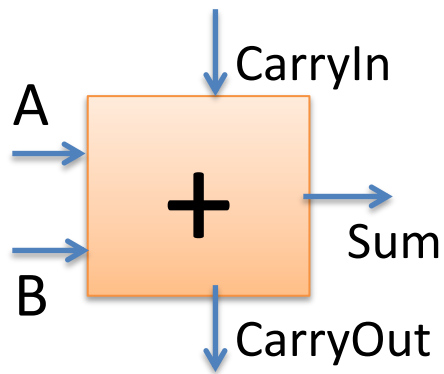
- Recall how base-2 addition works



At each bit position (e.g. pos1), take as inputs carryIn(c_1), a_1 , b_1 , and compute sum (s_1), carryout(c_2). Feed c_2 to the next bit position as carryIn.

1-bit adder





1-bit adder

Inputs			Outputs	
a	b	CarryIn	CarryOut	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

Brute force PLA

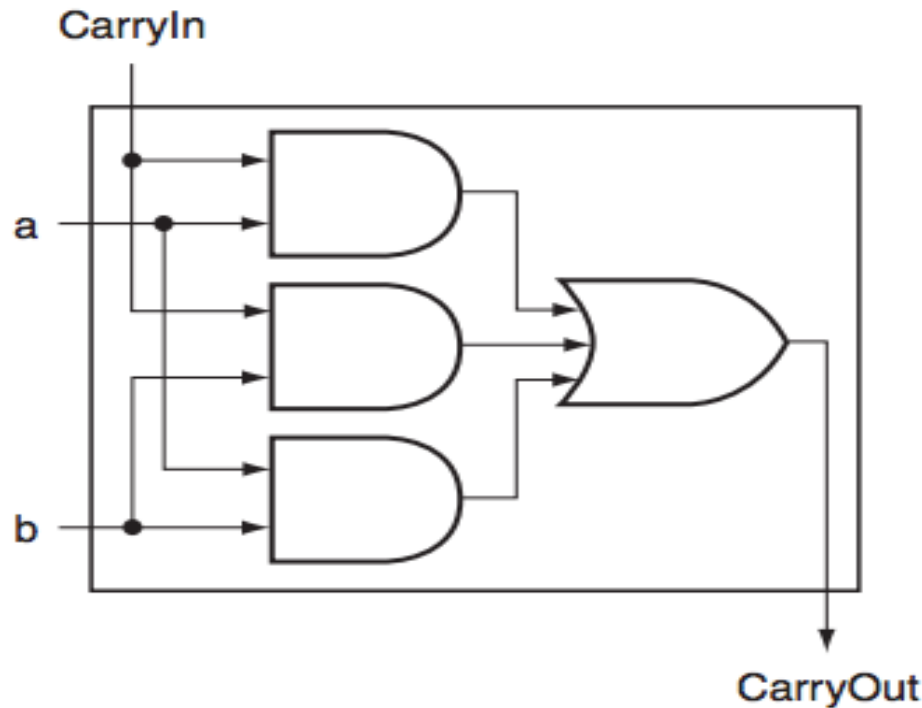
1-bit adder: computing CarryOut

a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

Rows where CarryOut is 1

1-bit adder: computing CarryOut

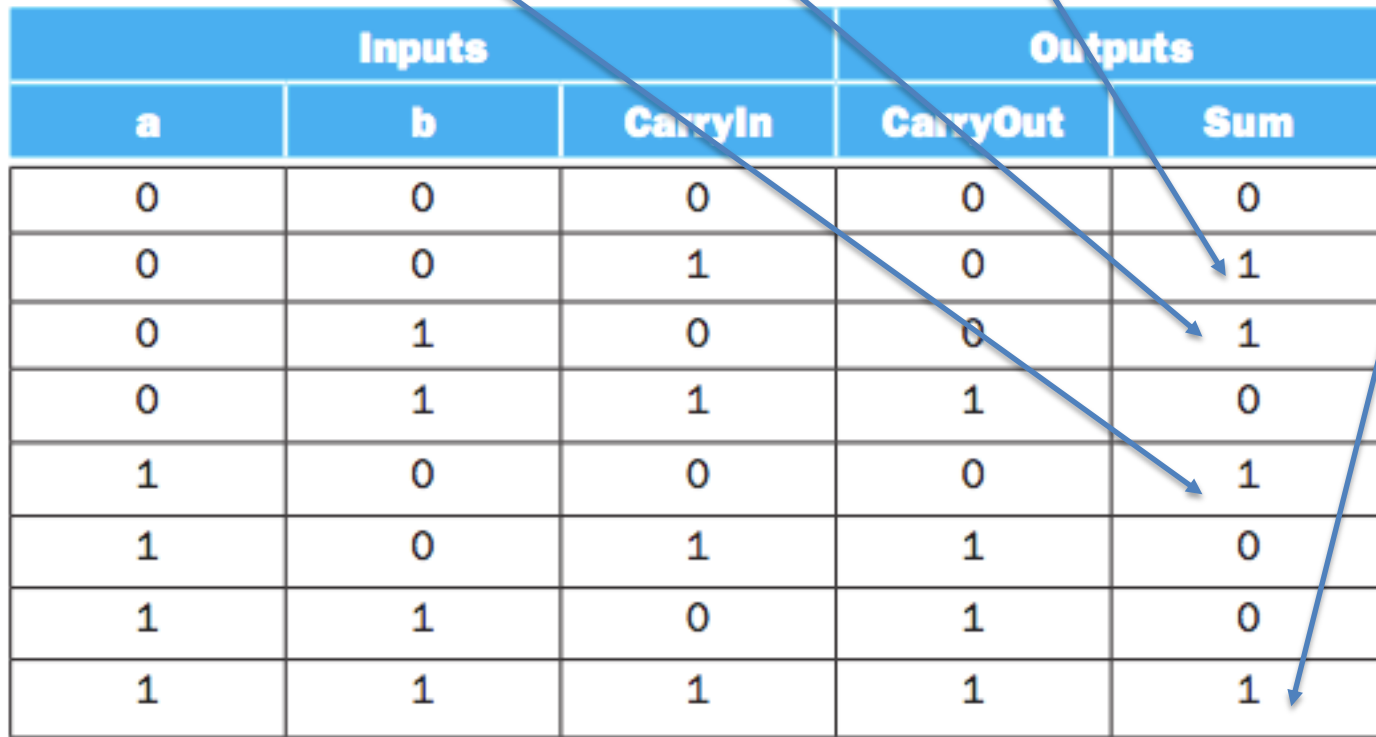
$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$



1-bit adder: computing Sum

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

Inputs			Outputs	
a	b	CarryIn	CarryOut	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

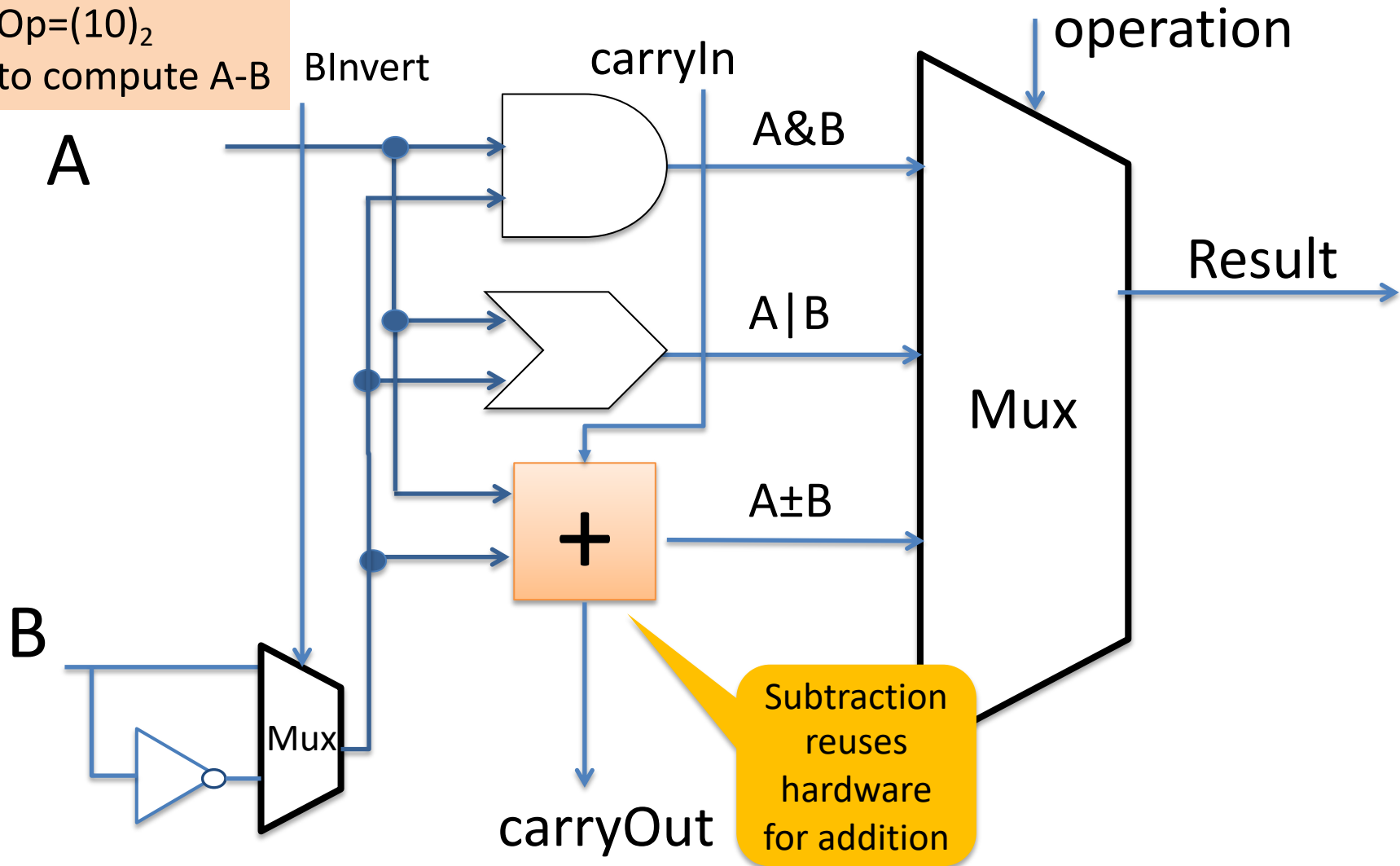


Subtraction

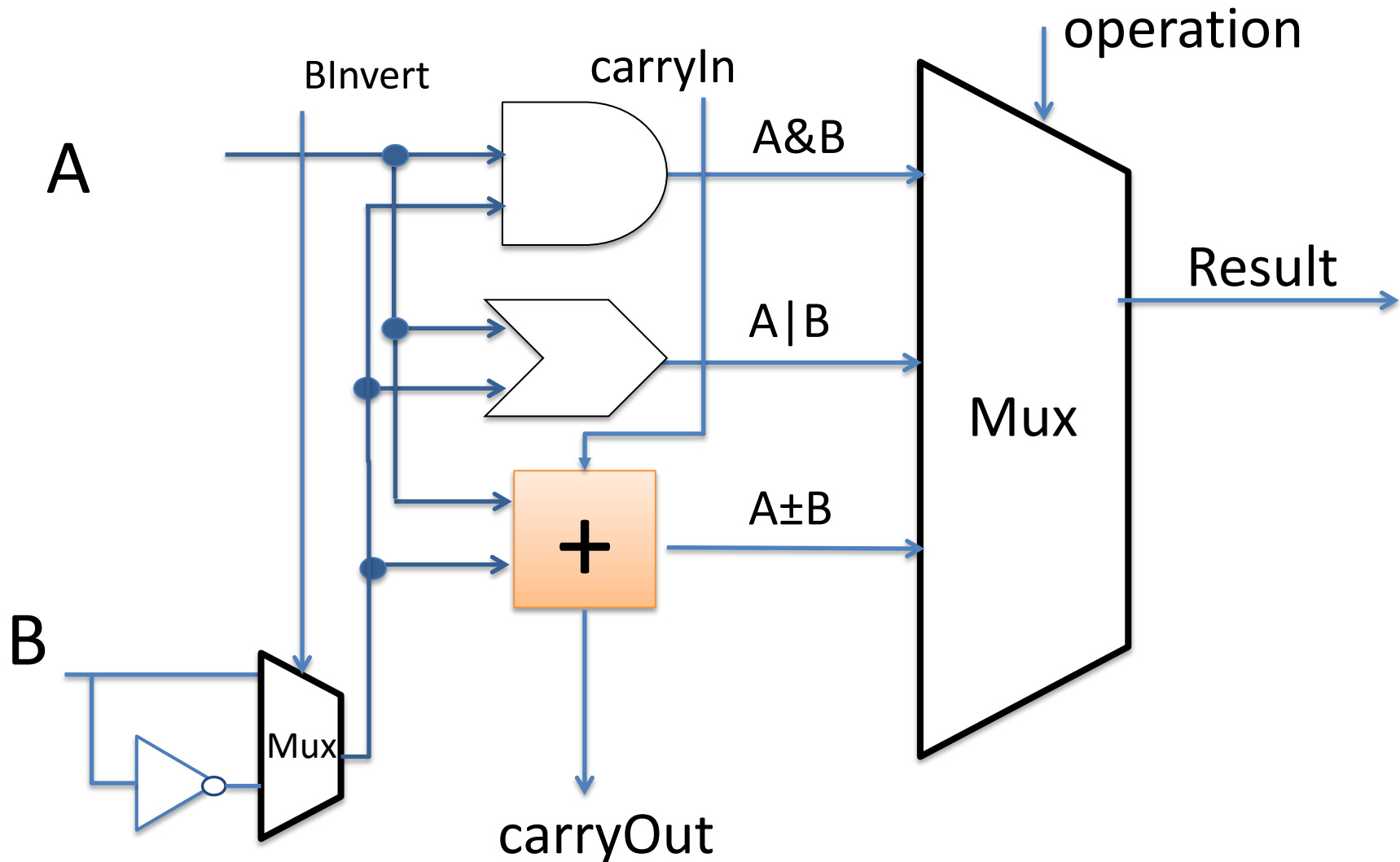
- Idea: $a - b = a + (-b)$
- How to calculate 2's complement?

Set Binvert=1
carryIn=1
Op=(10)₂
to compute A-B

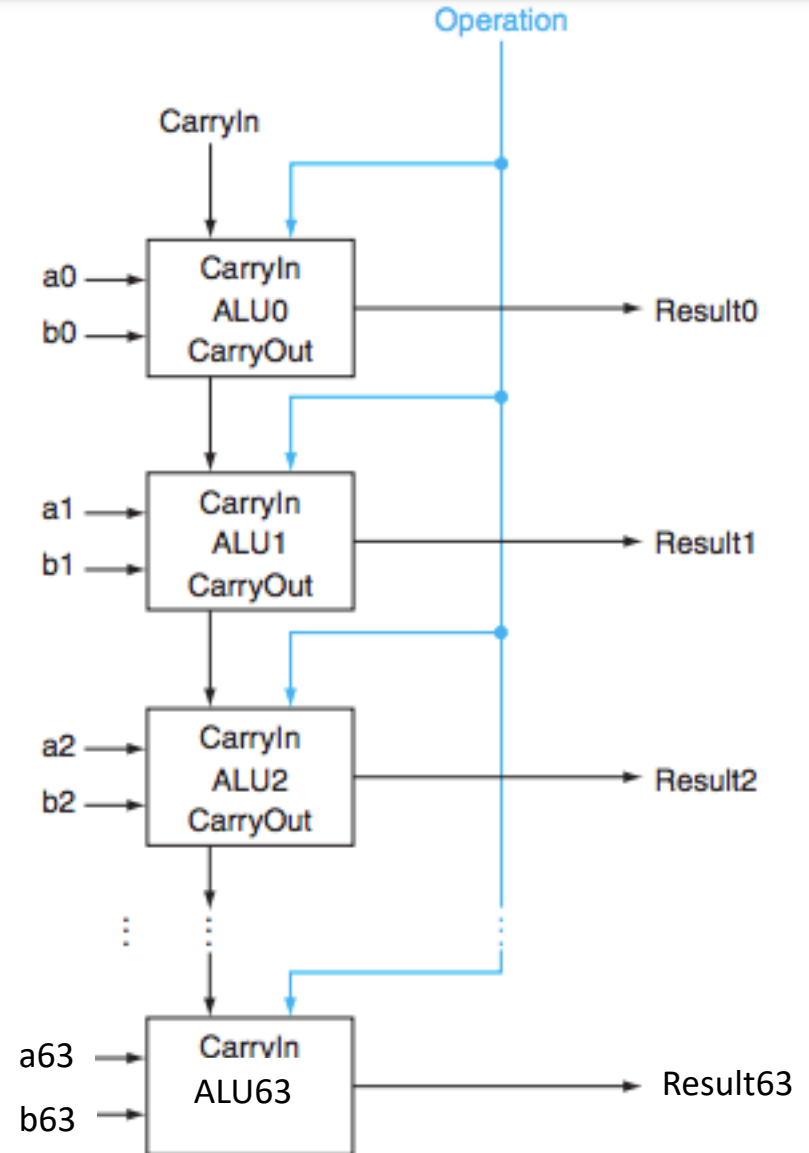
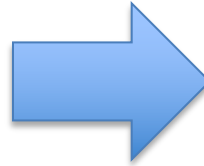
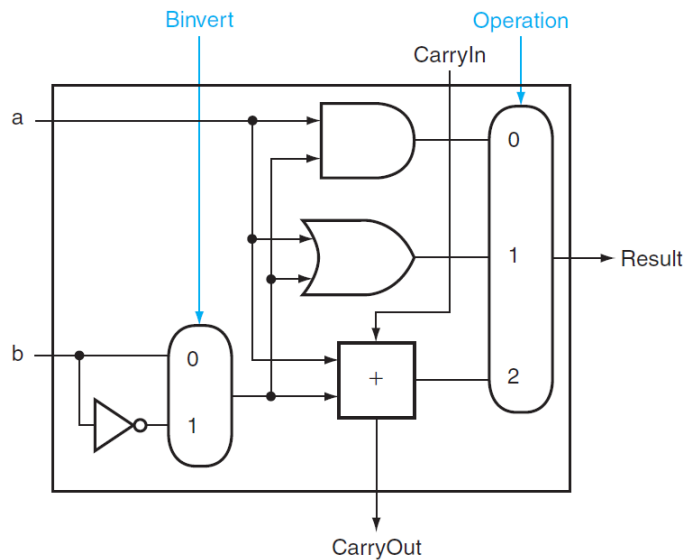
Subtraction in 1-bit ALU



Extend 1-bit ALU to 64-bit



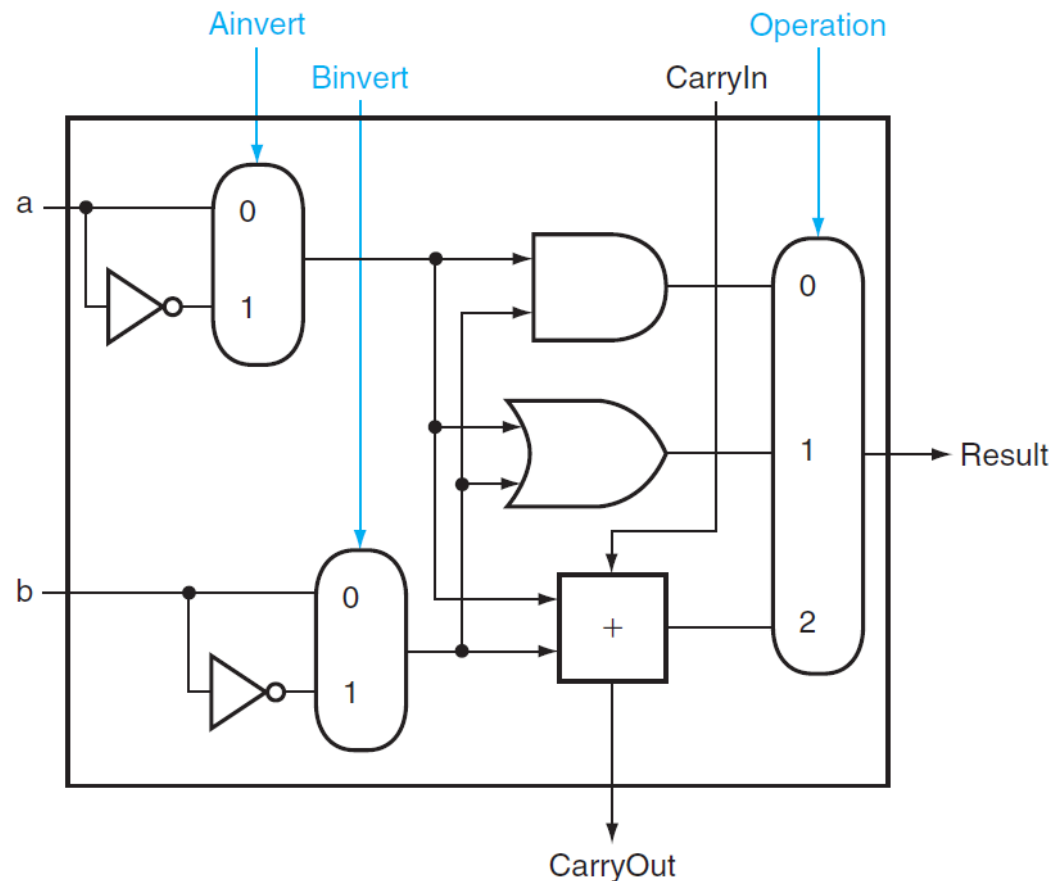
Extend 1-bit ALU to 64-bit ALU



Extend ALU to include NOR

$$\overline{A+B} = \overline{A} \cdot \overline{B}$$

Set Binvert=1, Ainvert=1
Op=(00)₂
to compute a NOR b

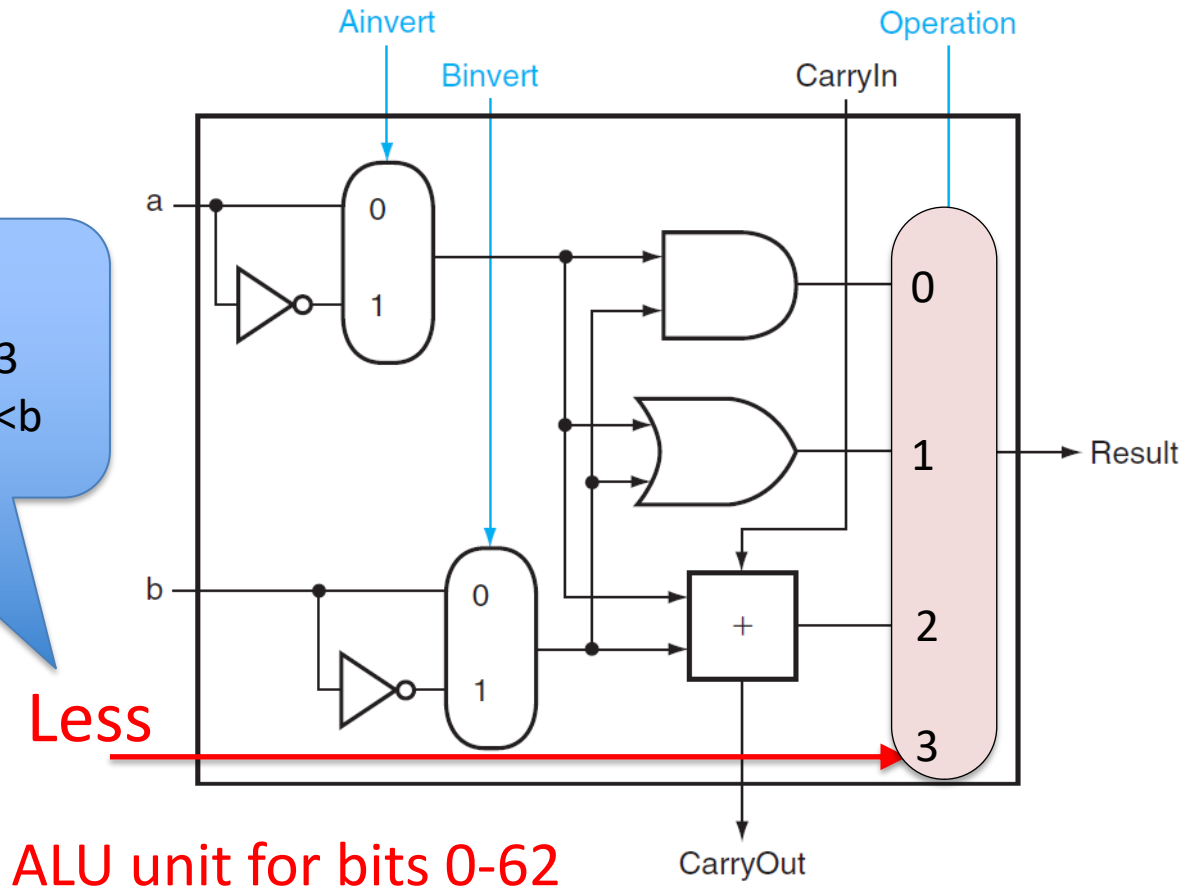


Extend ALU to include slt

- RISC-V slt (set-less-than) instruction
 - Result = $(A < B) ? 1 : 0$ Signed
 - X86 equivalent: `cmpq %rbx,%rax setl %rcx`

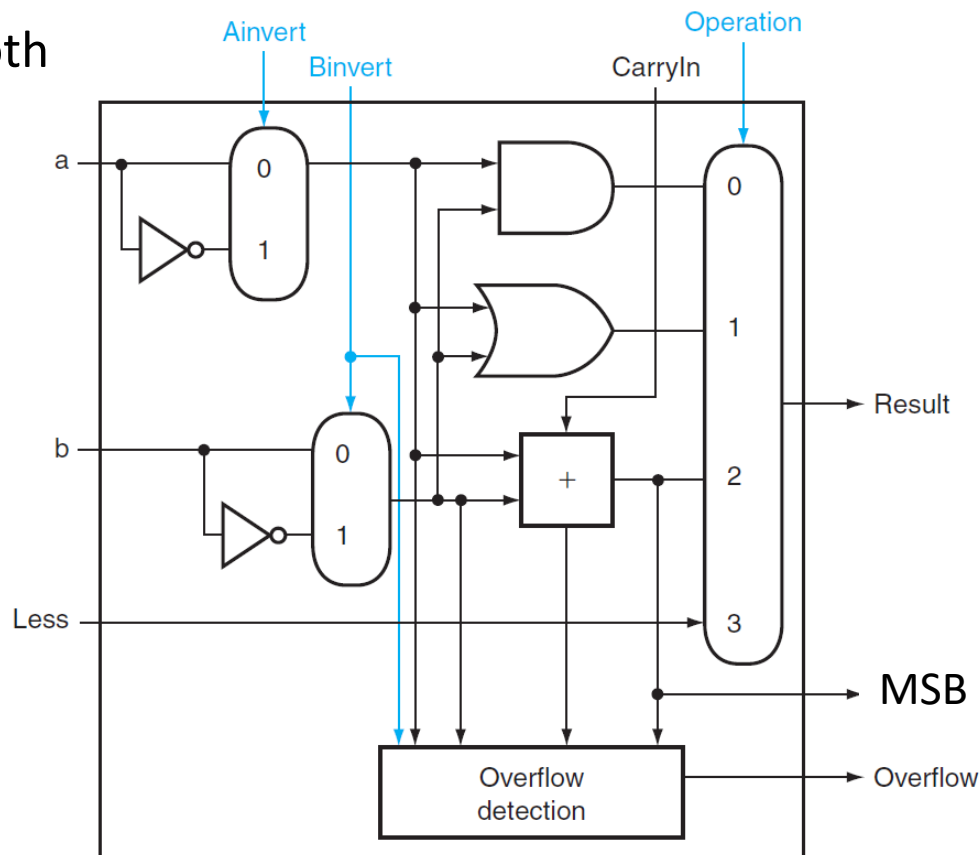
New input “Less”:

- always 0 for bits1:63
- Set to 1 for bit0 if $a < b$

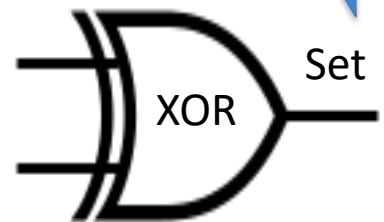


Extend ALU to include slt

- $A < B$ iff:
 - (A-B) is negative (MSB is 1)
 - (A-B) overflowed
 - But not both

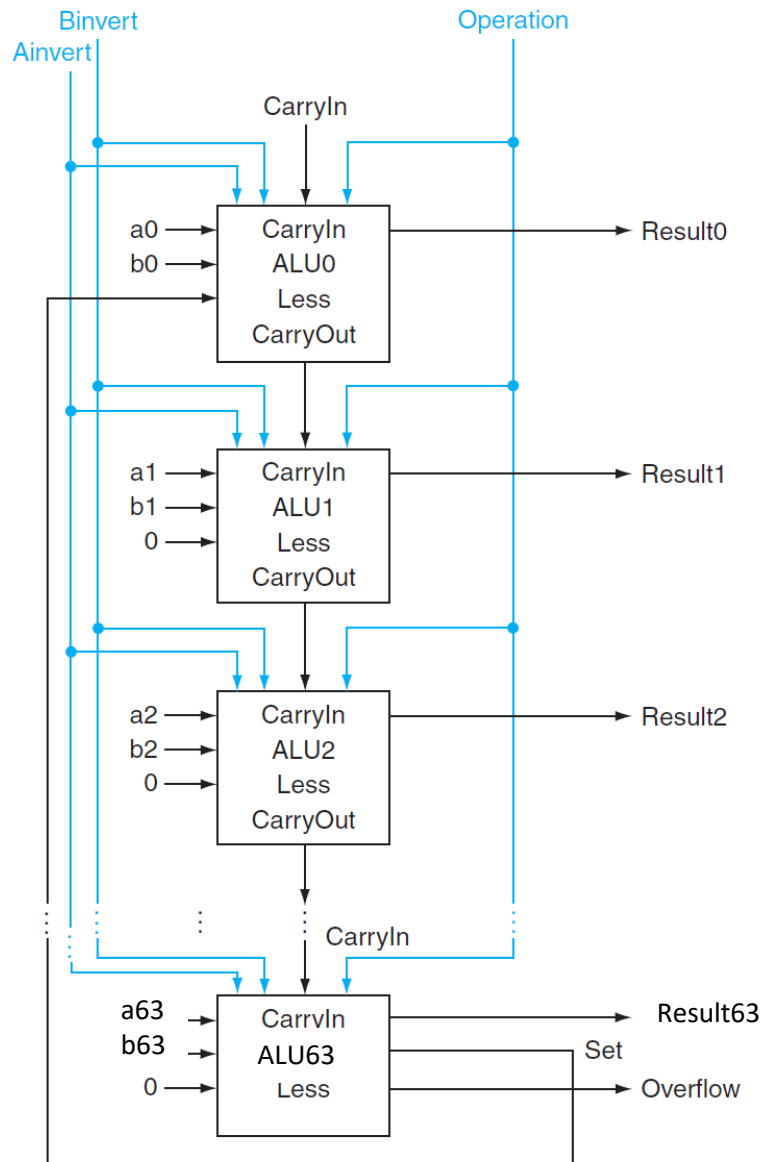


Wire to “Less”
input of the
least significant
bit ALU

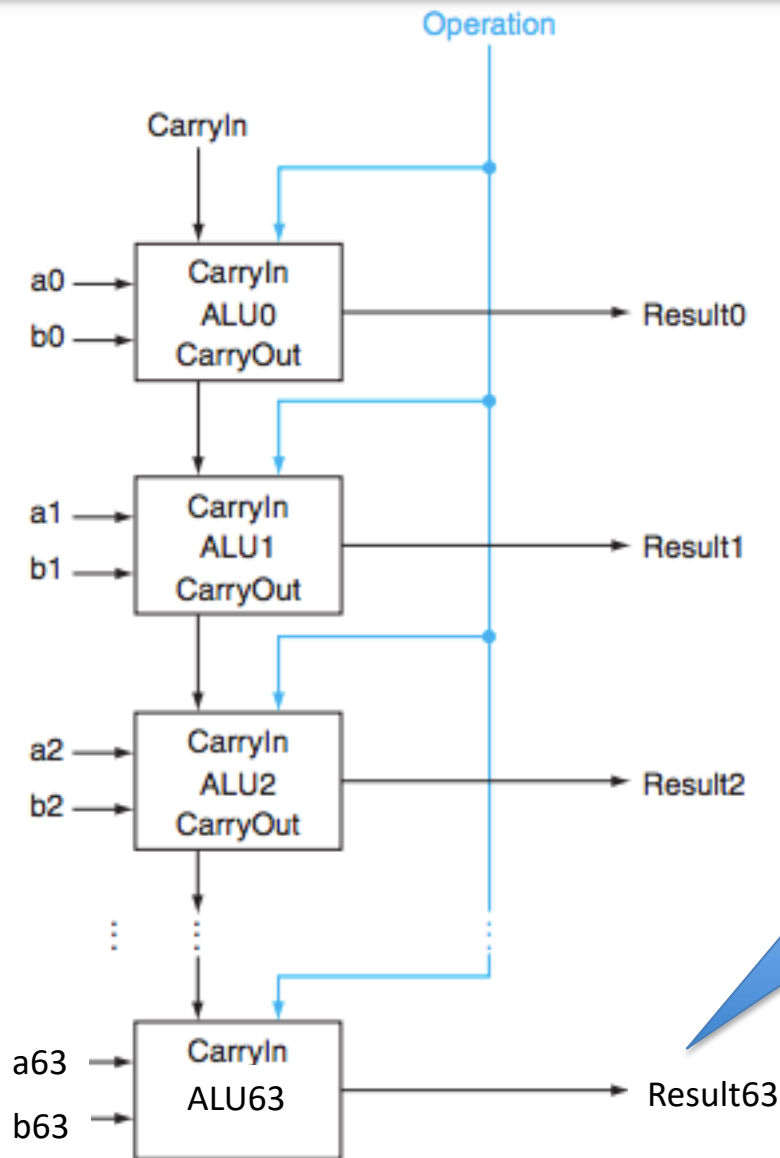


ALU unit for MSB(bit63)

Extend ALU to include slt



Downside of ripple carry?



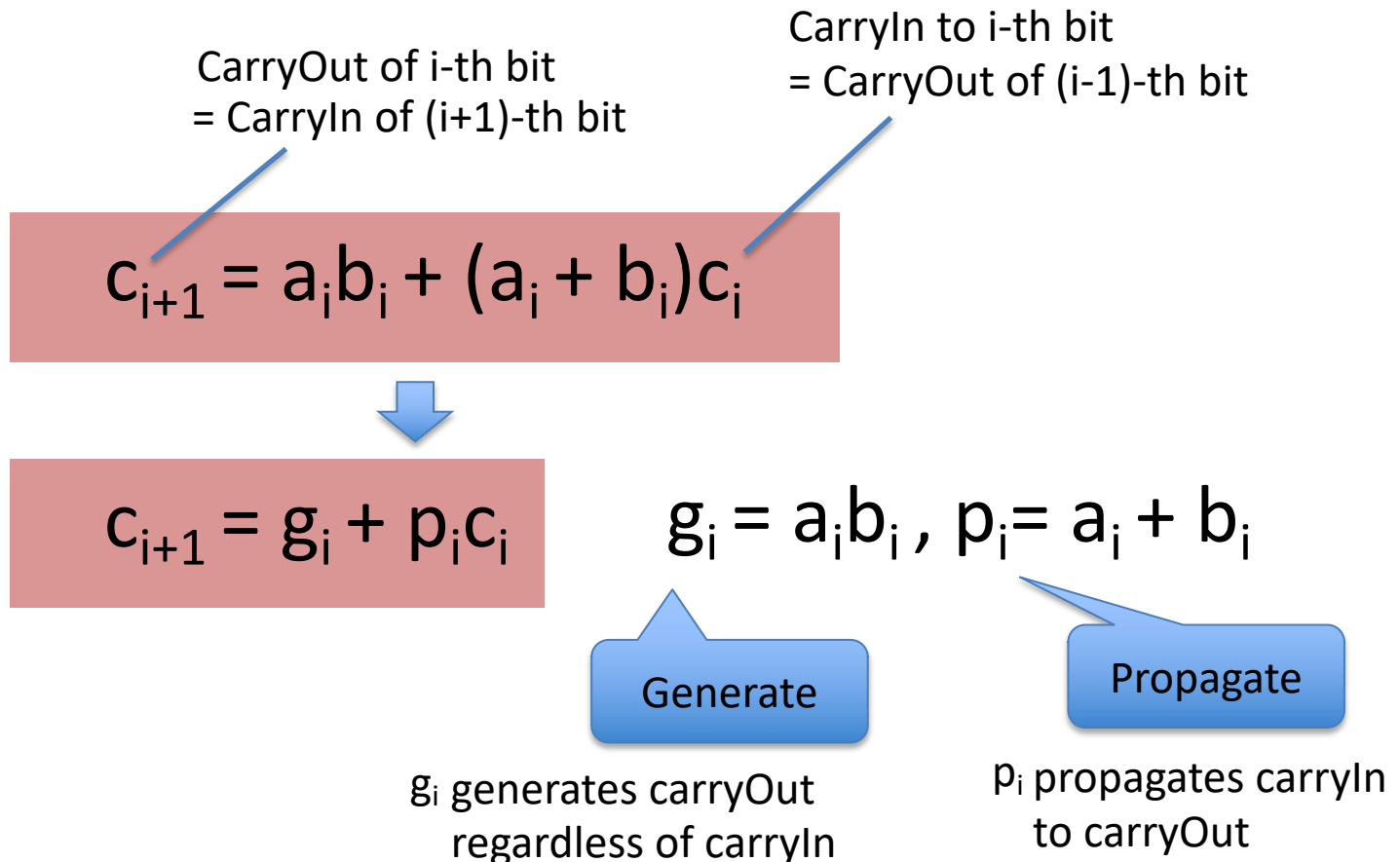
Must wait for sequential evaluation of all 64 1-bit adders

In search of a faster adder

- Ripple carry:
 - Delay: 64, Gate count: $64 * c$
- Brute-force (truth table->PLA)
 - Delay: 2, Gate count: $O(2^{64+64})$
- Clever designs in between?
- Idea #1: (Carry lookahead) compute multiple carry-bits at a time

Faster adder: carry lookahead

- Idea #1: (Carry lookahead) compute multiple carry-bits at a time



Faster adder: carry lookahead

- Idea #1: (Carry lookahead) compute multiple carry-bits at a time

Computing all carry-bits of a 4-bit adder:

$$c_1 = g_0 + (p_0 \cdot c_0)$$

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

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

$$c_4 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\ + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0)$$

Delay? 3 4-bit ripple carry
delay: $2 * 4$

Faster adder: carry lookahead

- Idea #1: (Carry lookahead) compute multiple carry-bits at a time

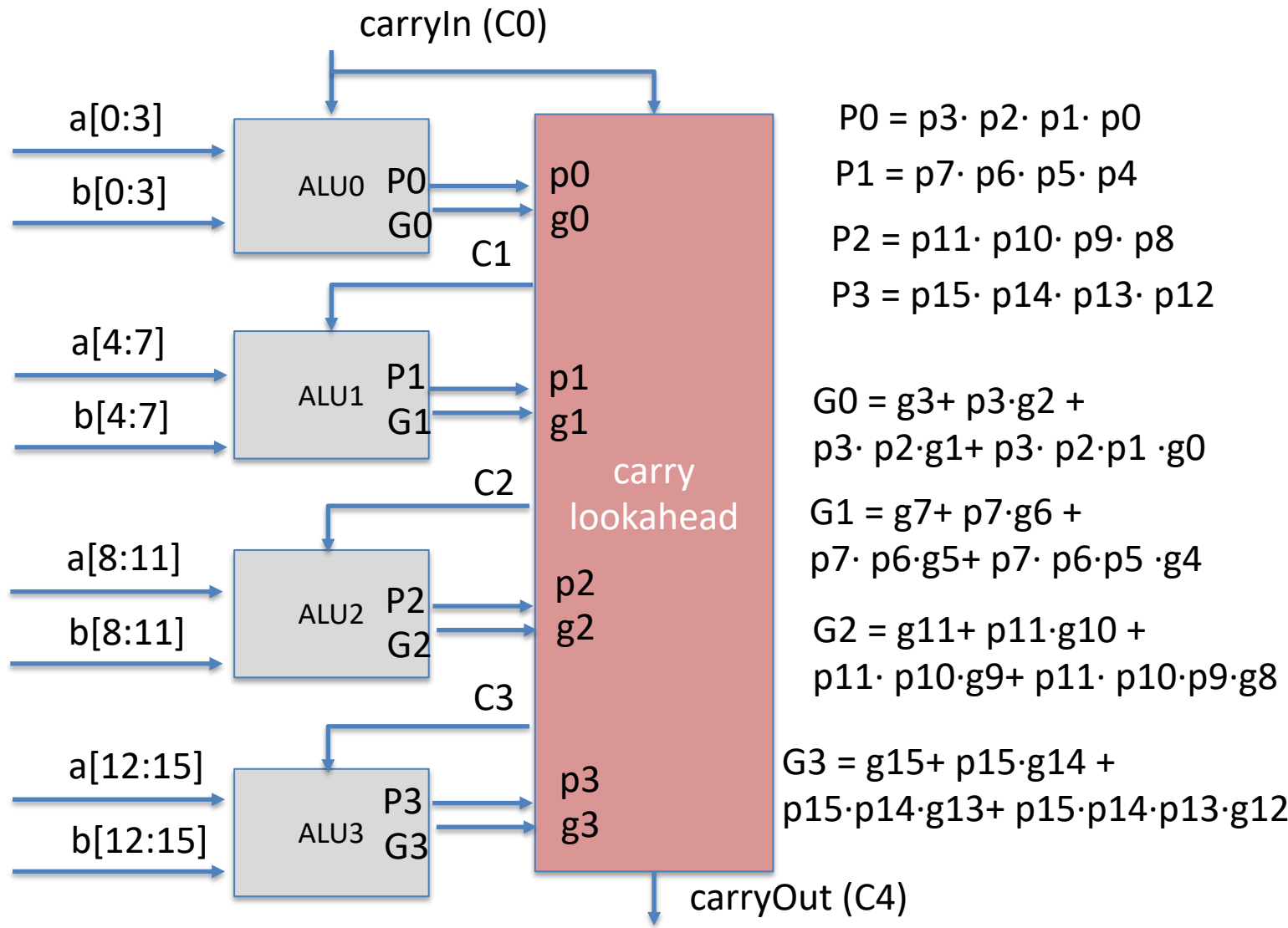
Computing all result bits in a 4-bit adder:

$$S_i = a_i \cdot \bar{b}_i \cdot \bar{c}_i + \bar{a}_i \cdot b_i \cdot \bar{c}_i + \bar{a}_i \cdot \bar{b}_i \cdot c_i + a_i \cdot b_i \cdot c_i$$

for $i = 0, 1, 2, 3$

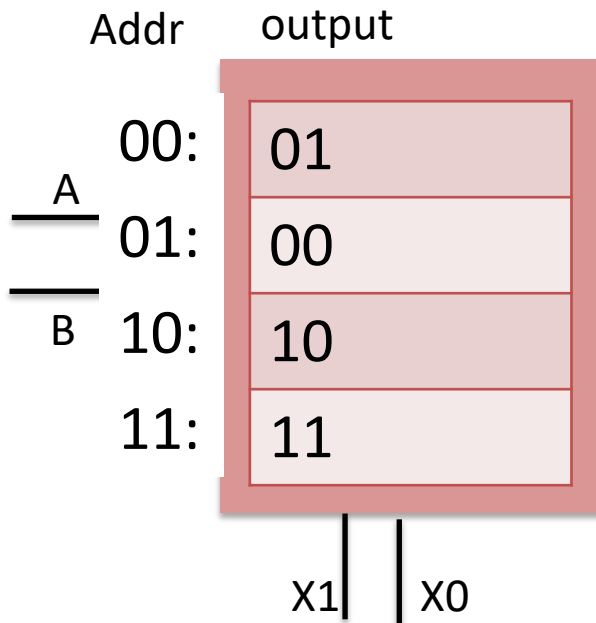
Faster adder: carry lookahead

- Build a 16-bit adder with carry-ahead 4-bit adders



Summary

- ROM



- ALU

