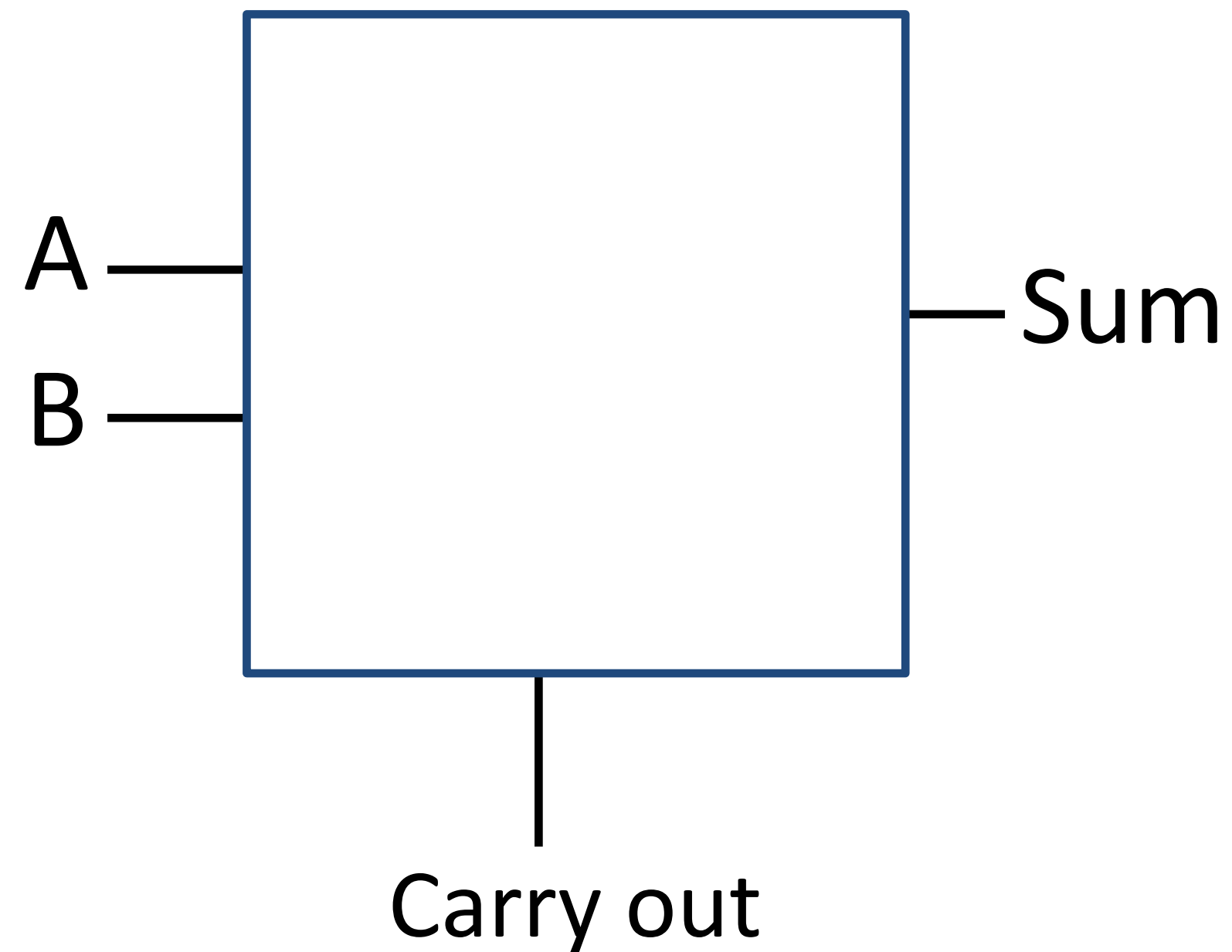# Arithmetic Logic

adders

Arithmetic Logic Unit
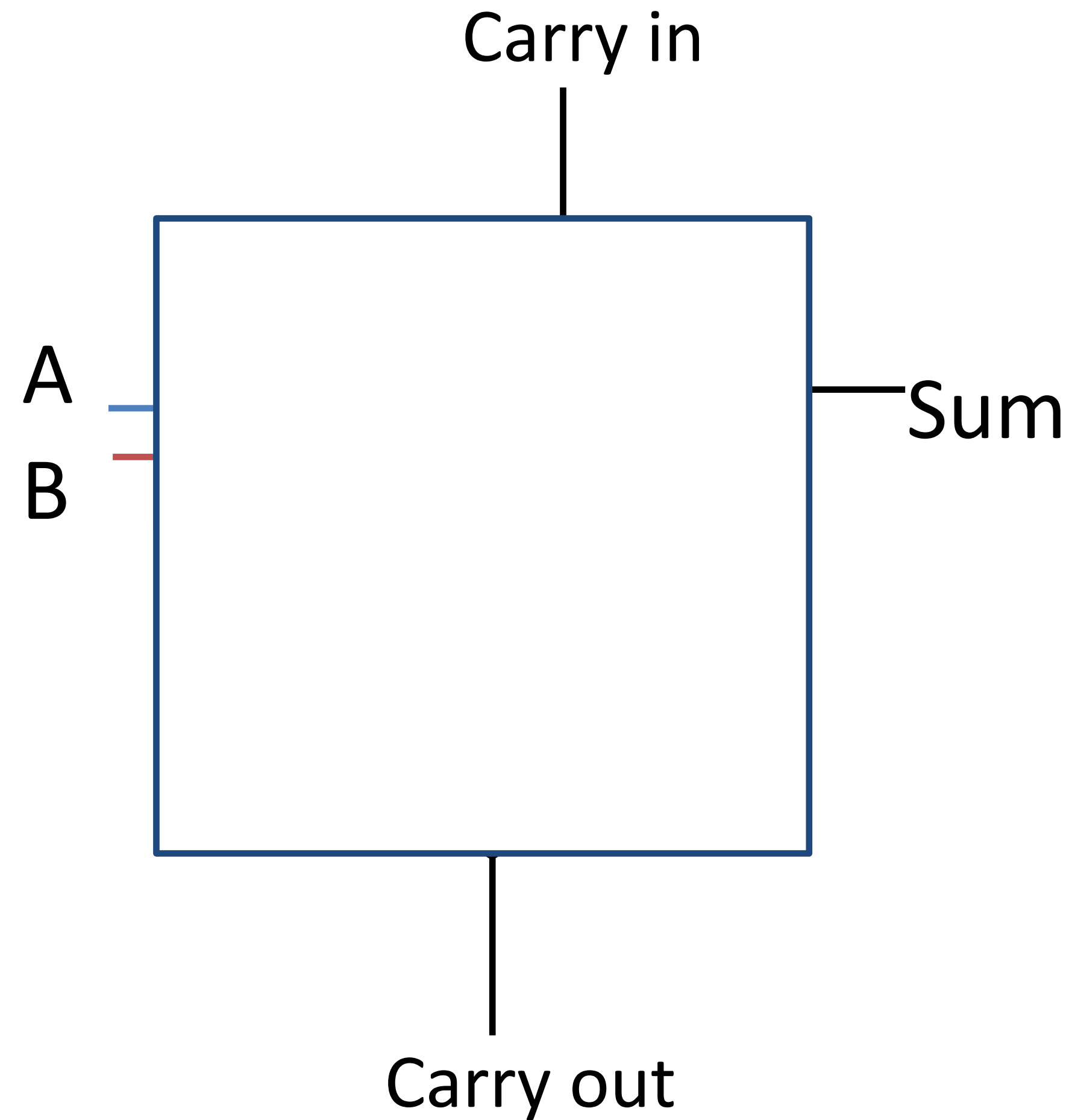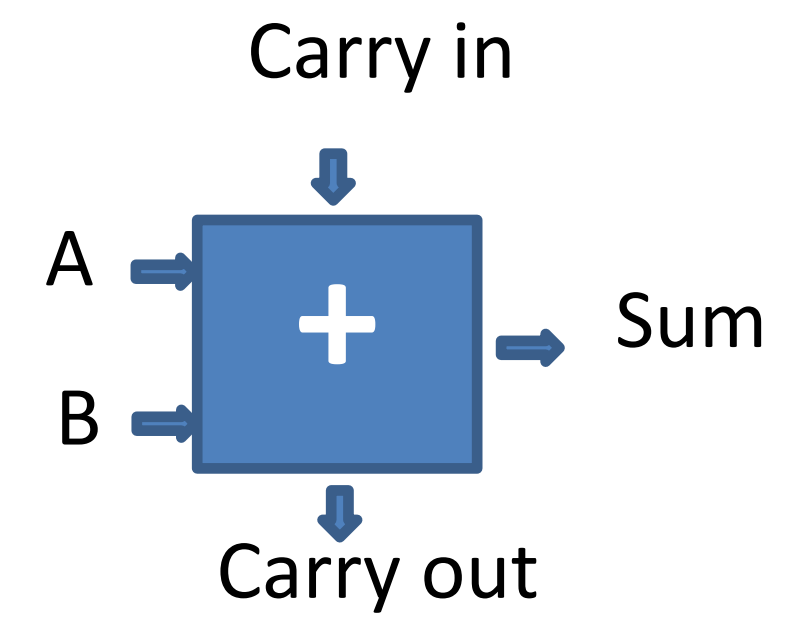
Addition: 1-bit *half* adder

A →
+ → Sum
B →
↓
Carry out

A ——
B ——

Sum
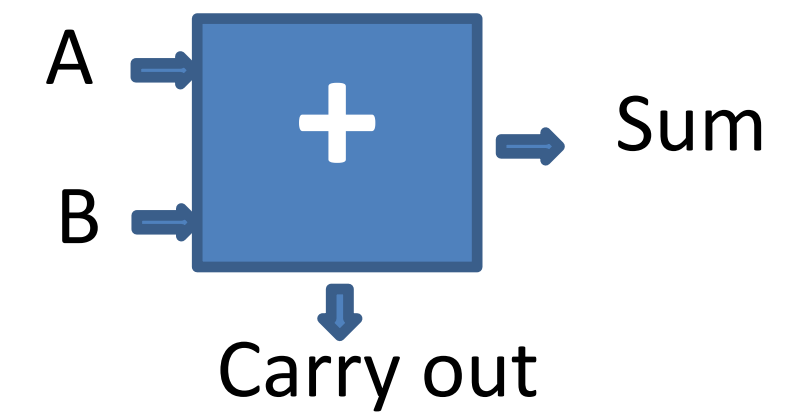
Carry out

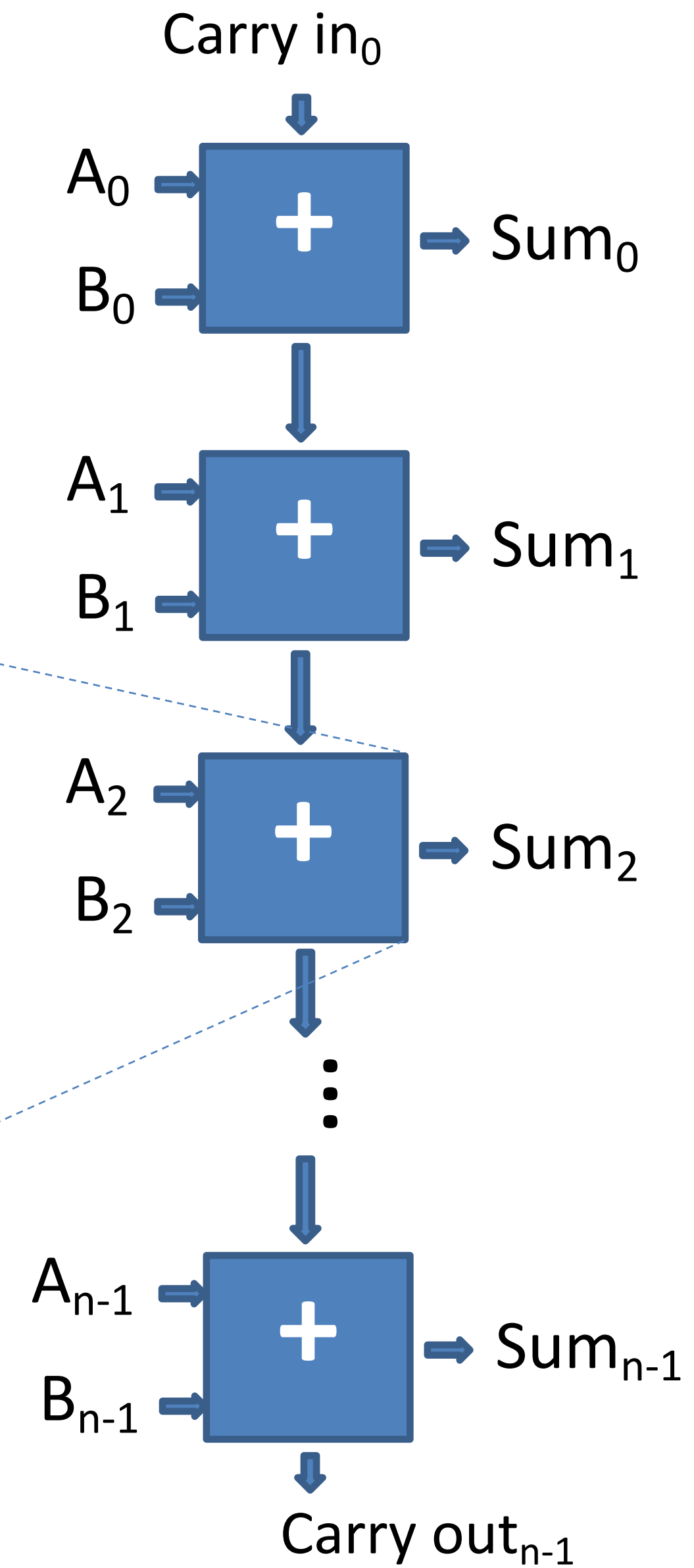| A	B	Carry Out	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Hint: the smallest solution uses 2 gates from: AND, OR, XOR, NOT, NAND, NOR

Addition: 1-bit *full* adder

Carry in

A → + → Sum

B

Carry out

Carry in

A

Sum

B

Carry out

Carry in	A	B	Carry Out	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

Addition: *n*-bit *ripple-carry* adder

Carry in

A

B

Carry out

Carry in$_0$

A$_0$

B$_0$

Sum$_0$

A$_1$

B$_1$

Sum$_1$

A$_2$

B$_2$

Sum$_2$

A$_{n-1}$

B$_{n-1}$

Sum$_{n-1}$

Carry out$_{n-1}$
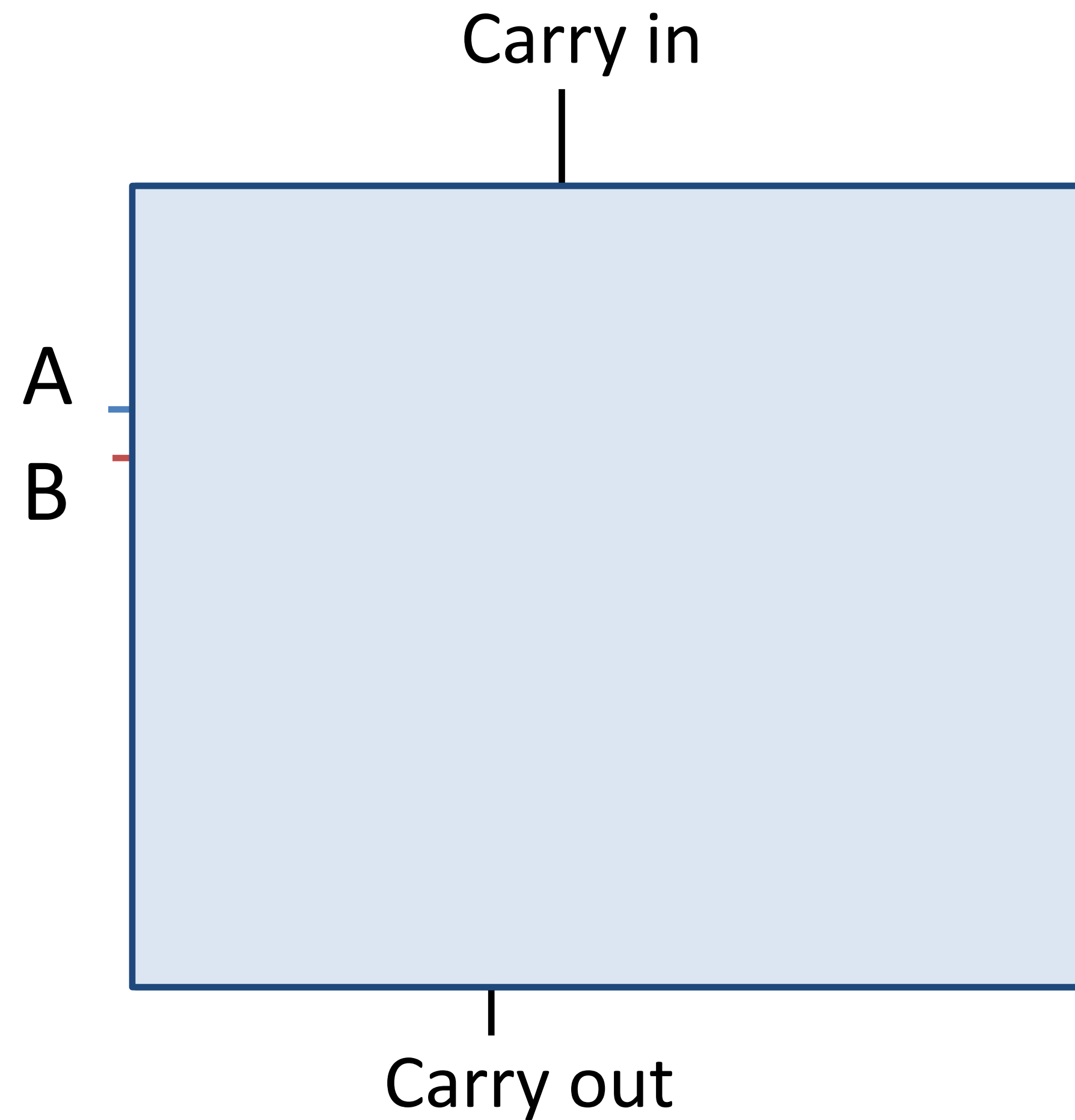
There are faster, more complicated ways too...

4

Processor Components

Abstraction!

4 Instruction Fetch and Decode

2 Registers

1 ALU

3 Memory

Arithmetic Logic Unit (ALU)

a few bits → **Operation**

Operand A →

words

Operand B →

ALU → **Result**

word

Condition Codes ← *a few bits*
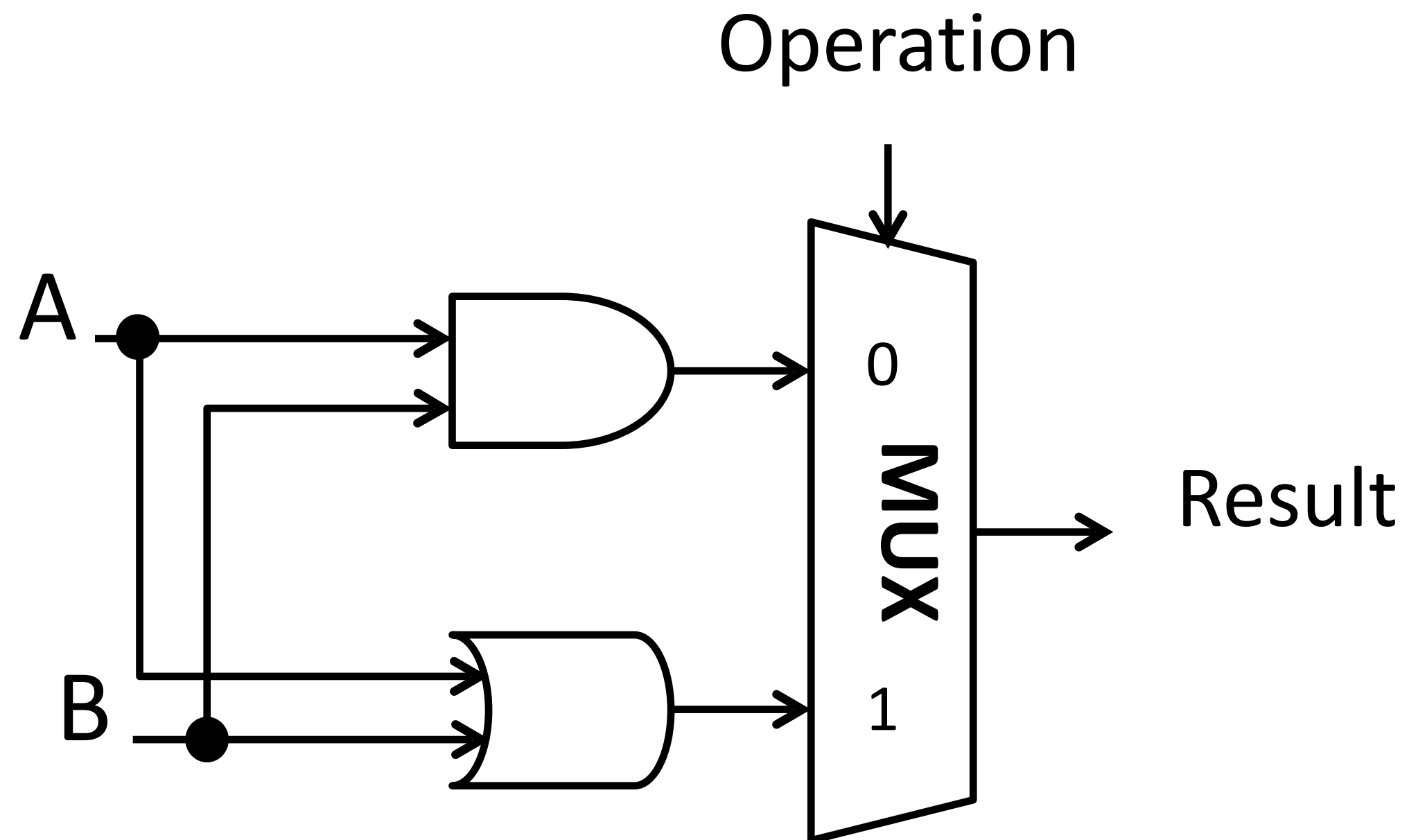(sign, overflow, carry-out, zero)
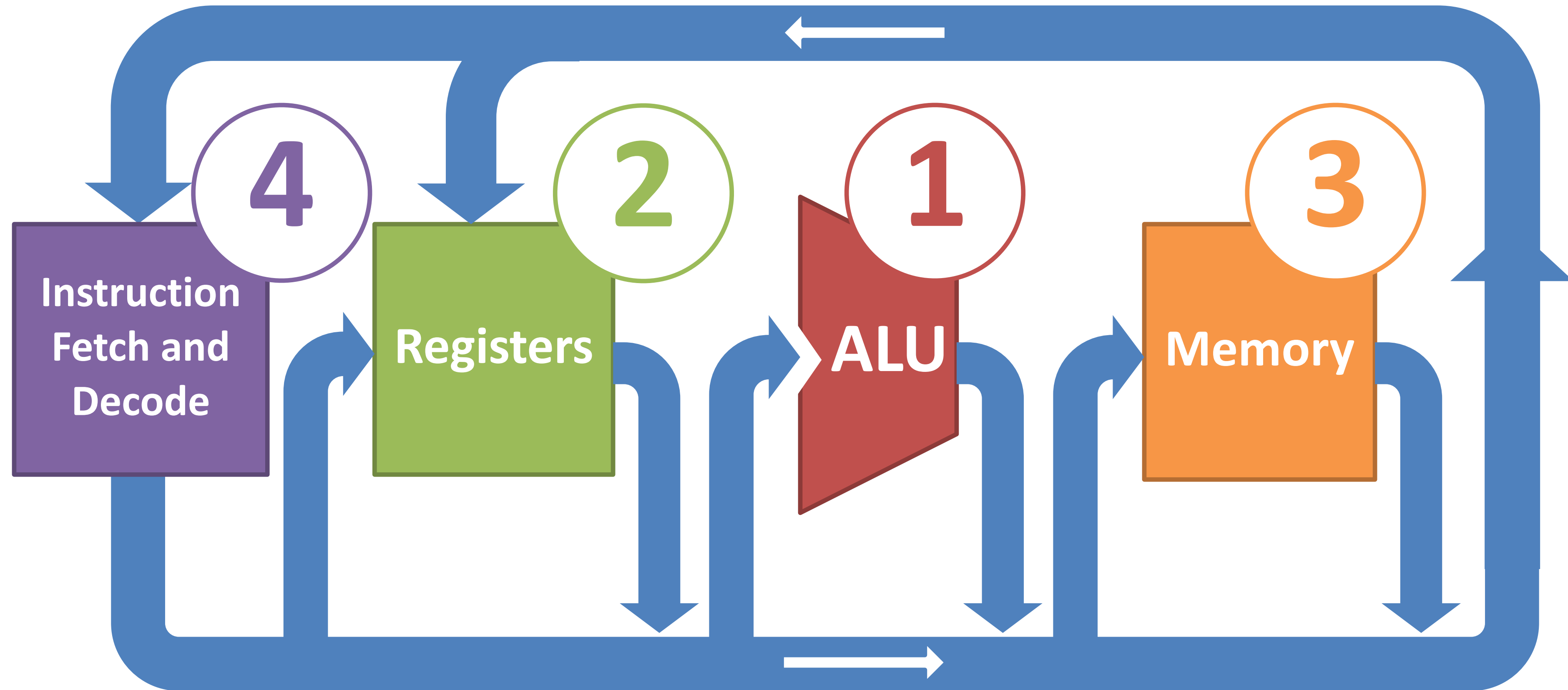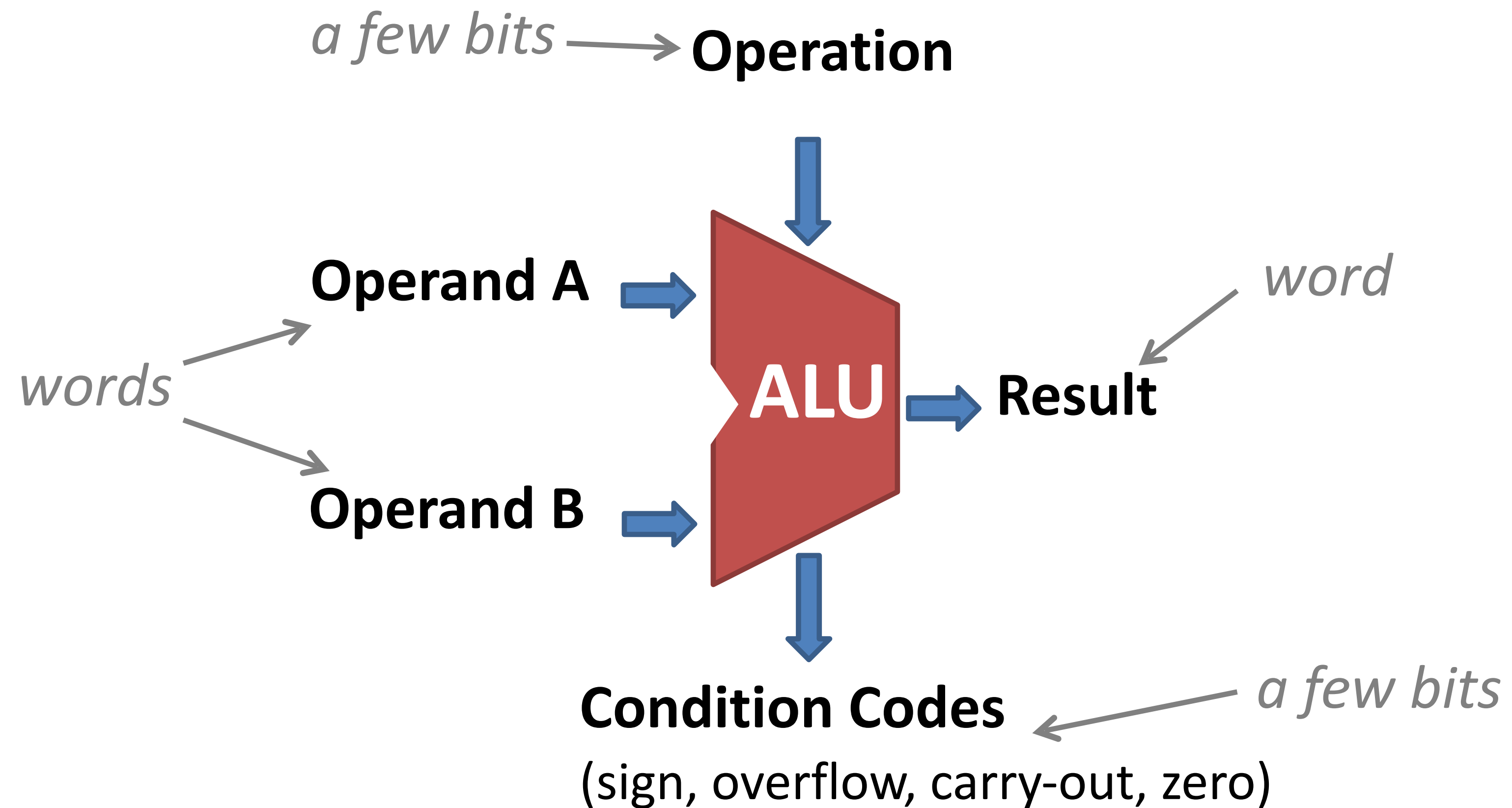
Hardware unit for arithmetic and bitwise operations.

1-bit ALU for bitwise operations

We will use *n* 1-bit ALUs to build an n-bit ALU.

Each bit *i* in the result is computed from the corresponding bit *i* in the two inputs.
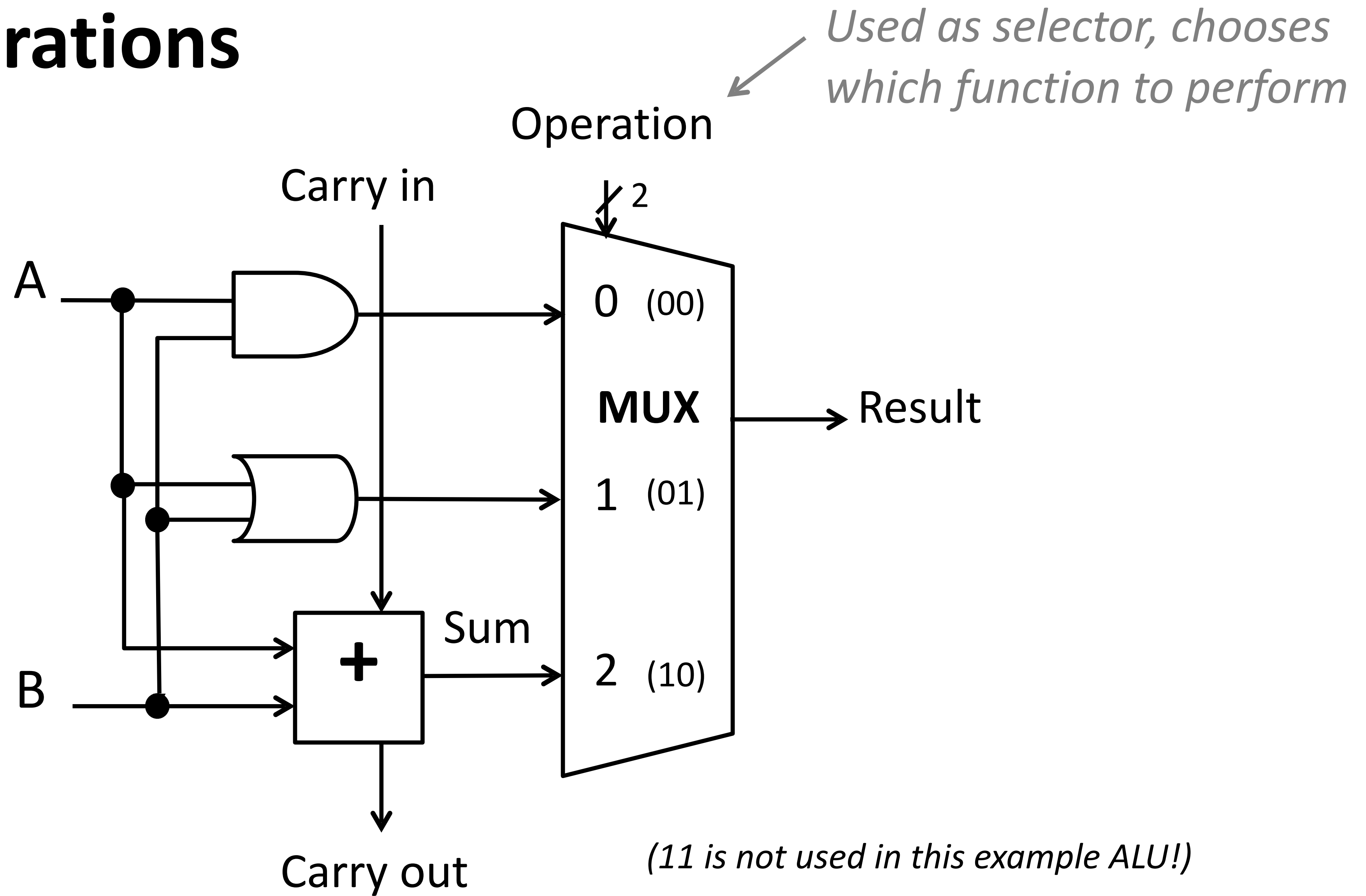
An example (simplified) 1-bit ALU

Op	A	B	Result
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

ex

1-bit ALU: 3 operations

Operation

Carry in

A

B

Carry in

MUX

0 (00)

1 (01)

2 (10)

Result

Sum

+

Carry out

(11 is not used in this example ALU!)

8

n-bit ripple carry adder

Carry in

A₀, B₀ → ➕ → Sum₀

A₁, B₁ → ➕ → Sum₁

A₂, B₂ → ➕ → Sum₂

⋮

Aₙ₋₁, Bₙ₋₁ → ➕ → Sumₙ₋₁

Carry out

n-bit ALU

Carry in
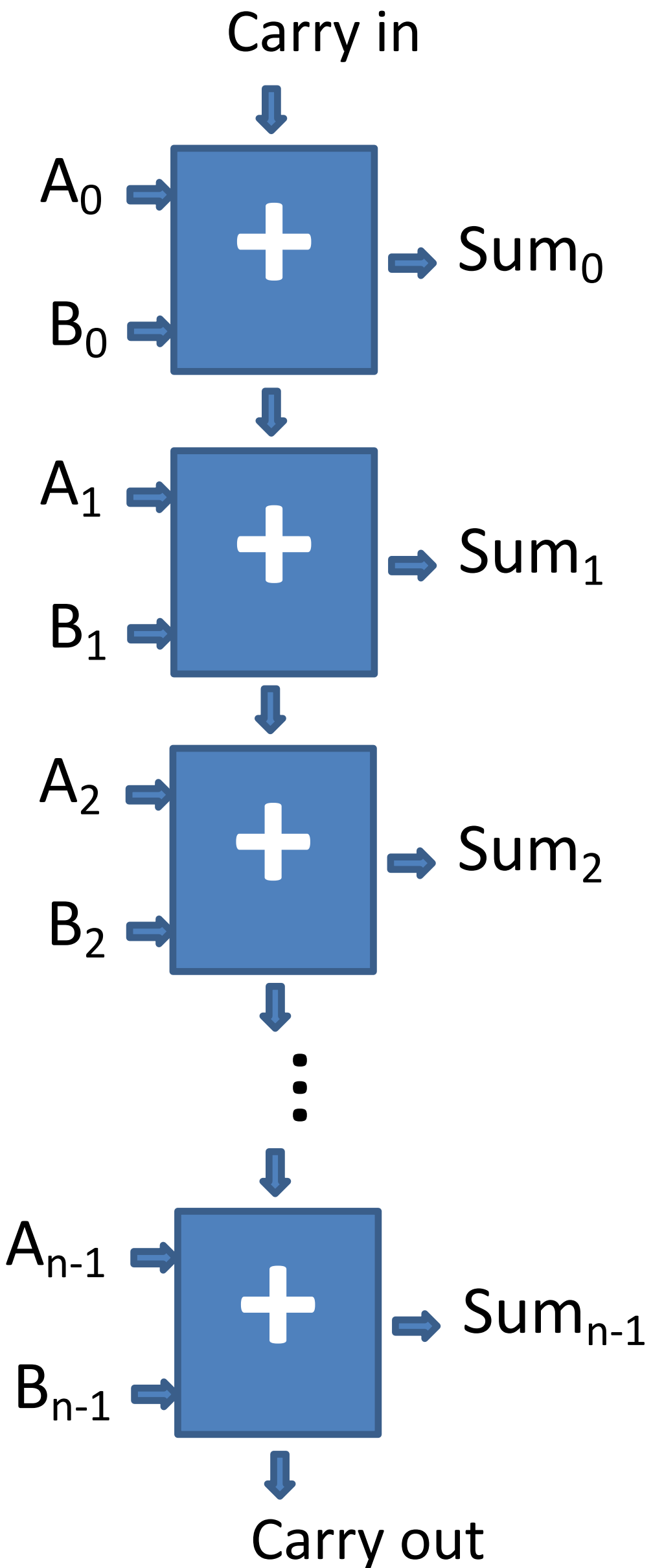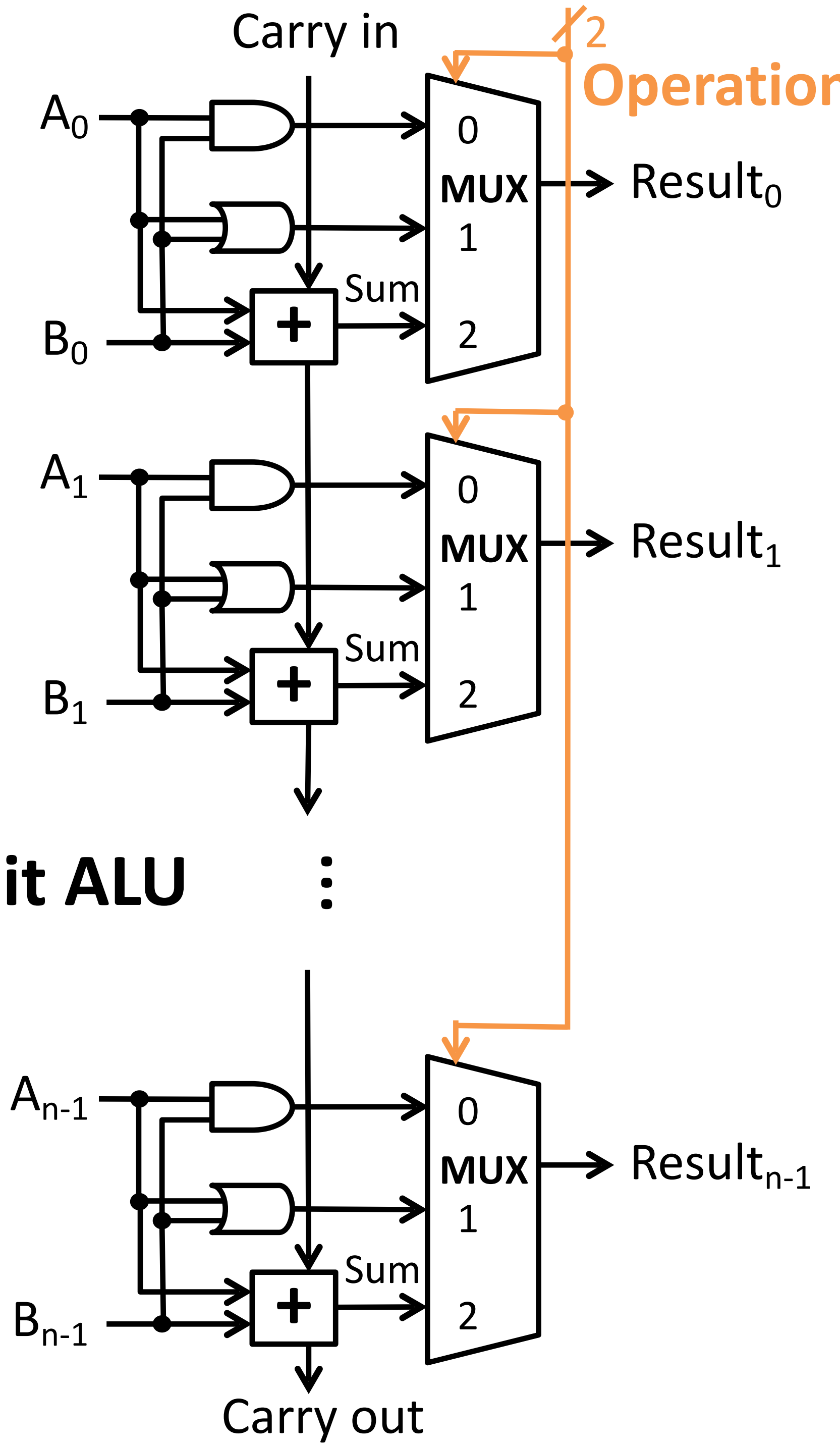
Operation ← *Use the same selector for every 1-bit ALU*

A₀ → MUX → Result₀

B₀ → Sum

A₁ → MUX → Result₁

B₁ → Sum

⋮

Aₙ₋₁ → MUX → Resultₙ₋₁

Bₙ₋₁ → Sum

Carry out

Controlling the ALU

ALU control lines	Function
00	AND
01	OR
10	add
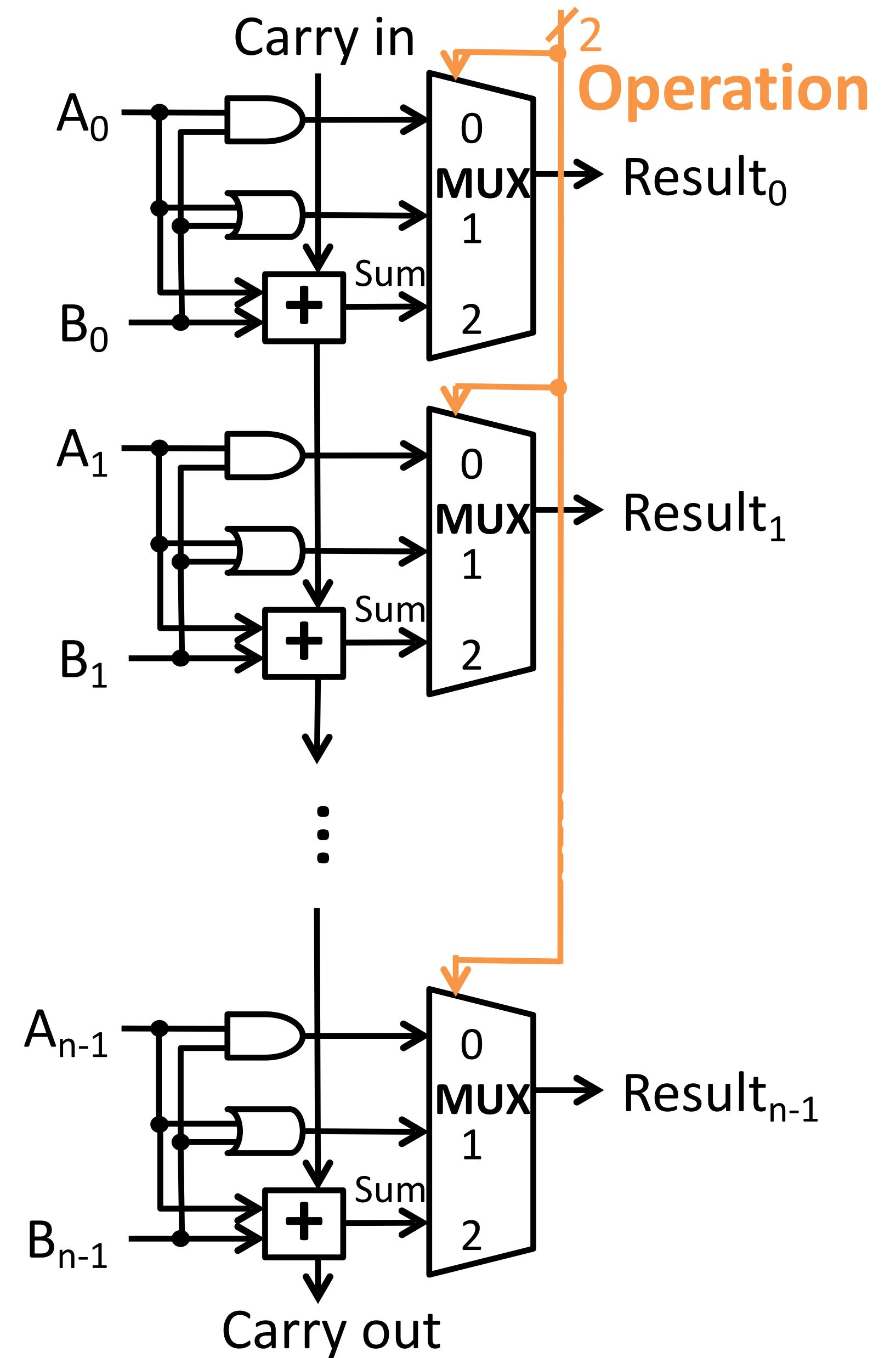
Include subtraction

How can we control ALU inputs
or add minimal new logic

to **also compute A−B**?

Recall:

```
A − B = A + (−B)
      = A + (~B + 1)
```

Plan:

Feed bitwise-not B into the adder
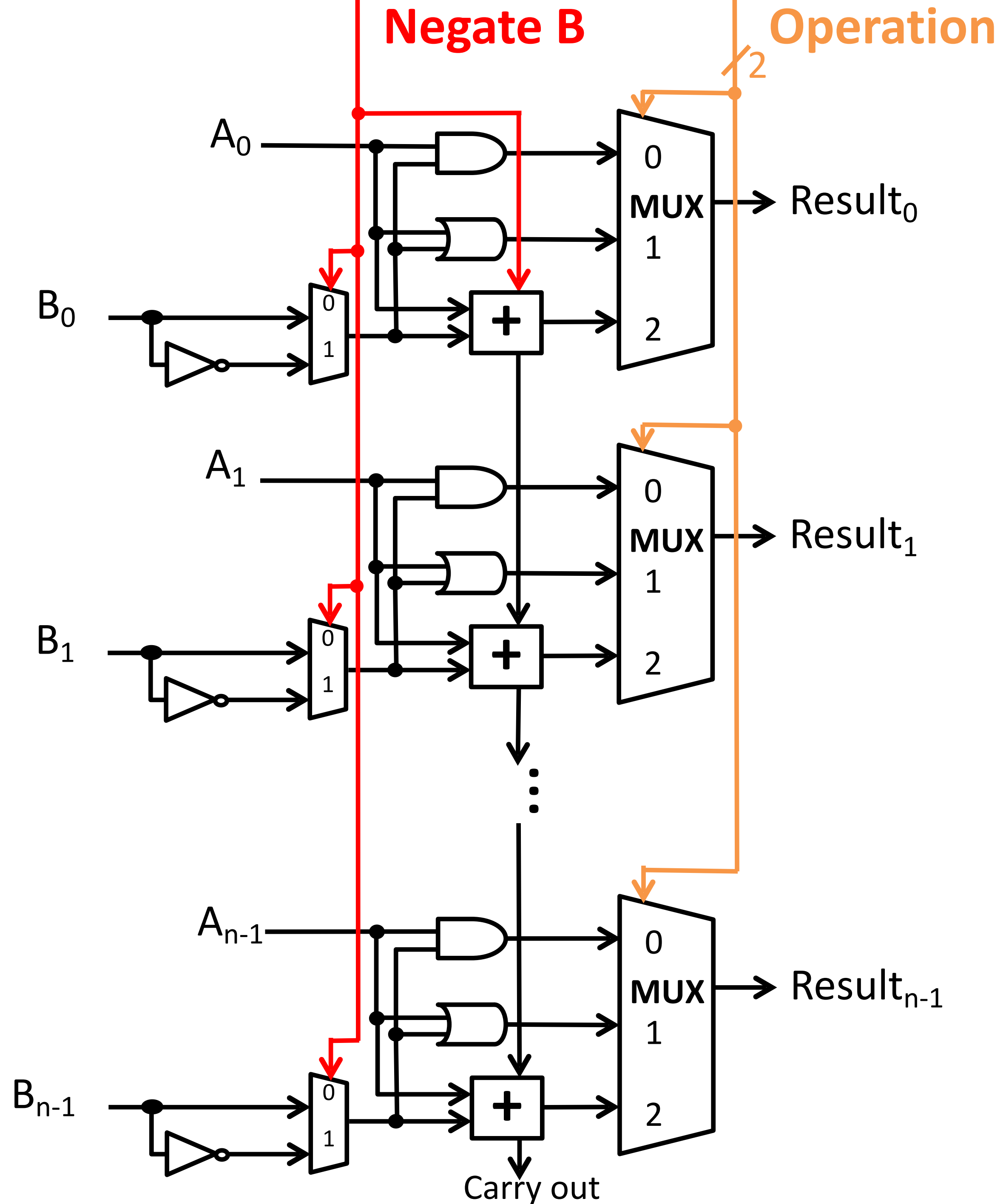
Add an extra 1: how?
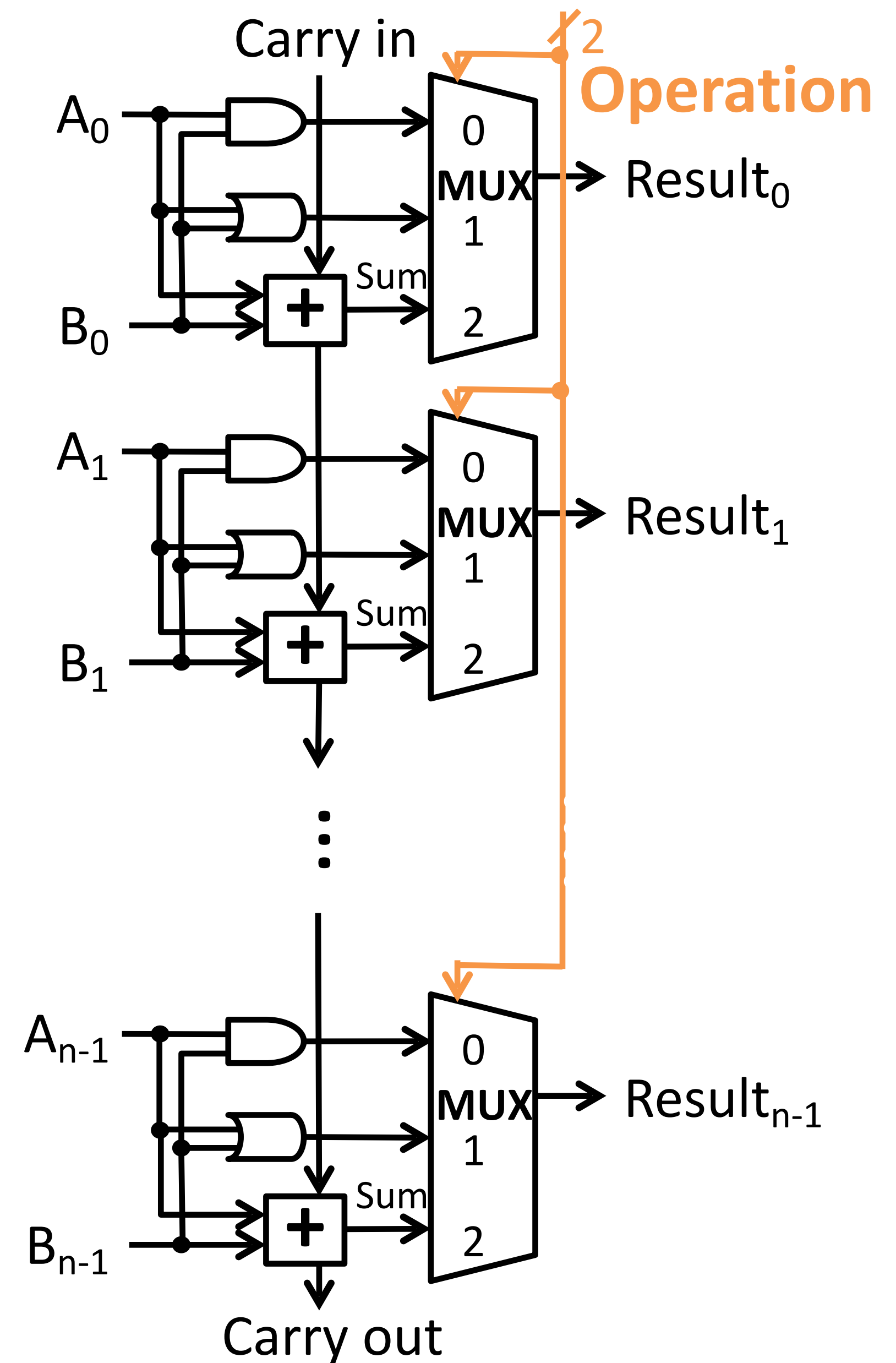
Include subtraction

Plan to compute A−B:

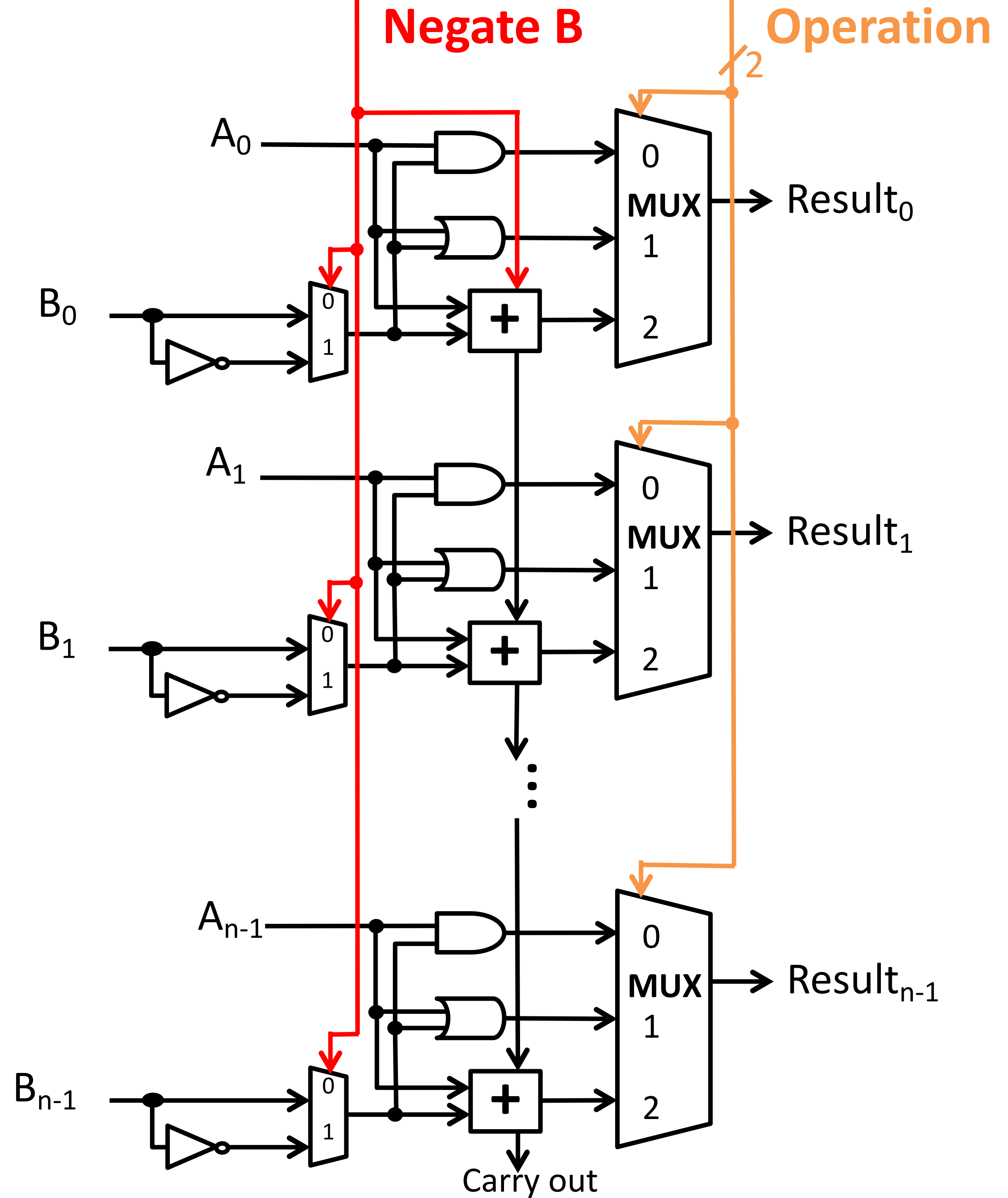1. Feed bitwise-not B into the adder
2. Add an extra 1

Key insight:

The *same* selector bit (0 or 1) can be used for both!

1. Feed the selector into a new 2:1 mux to choose B or ~B
2. Feed the selector in as the carry in to the least significant bit

Include subtraction

ALU control lines	Function
000	AND
001	OR
010	add
110	subtract
...	...

Negate B   Operation

A_0 B_0 A_1 B_1 A_{n-1} B_{n-1}

MUX   Result$_0$

MUX   Result$_1$

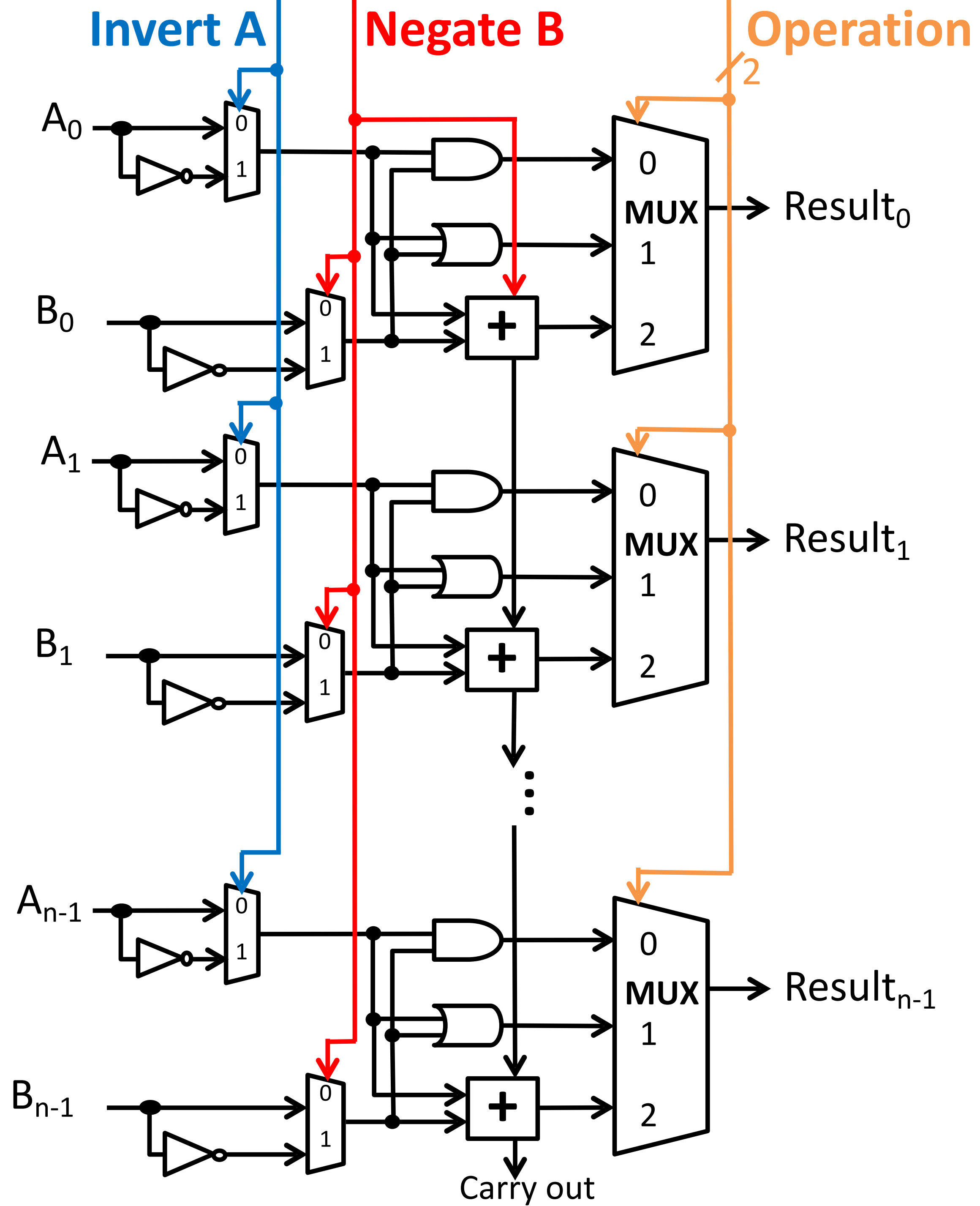MUX   Result$_{n-1}$

Carry out

A NAND B

A NOR B

A<B

A==B
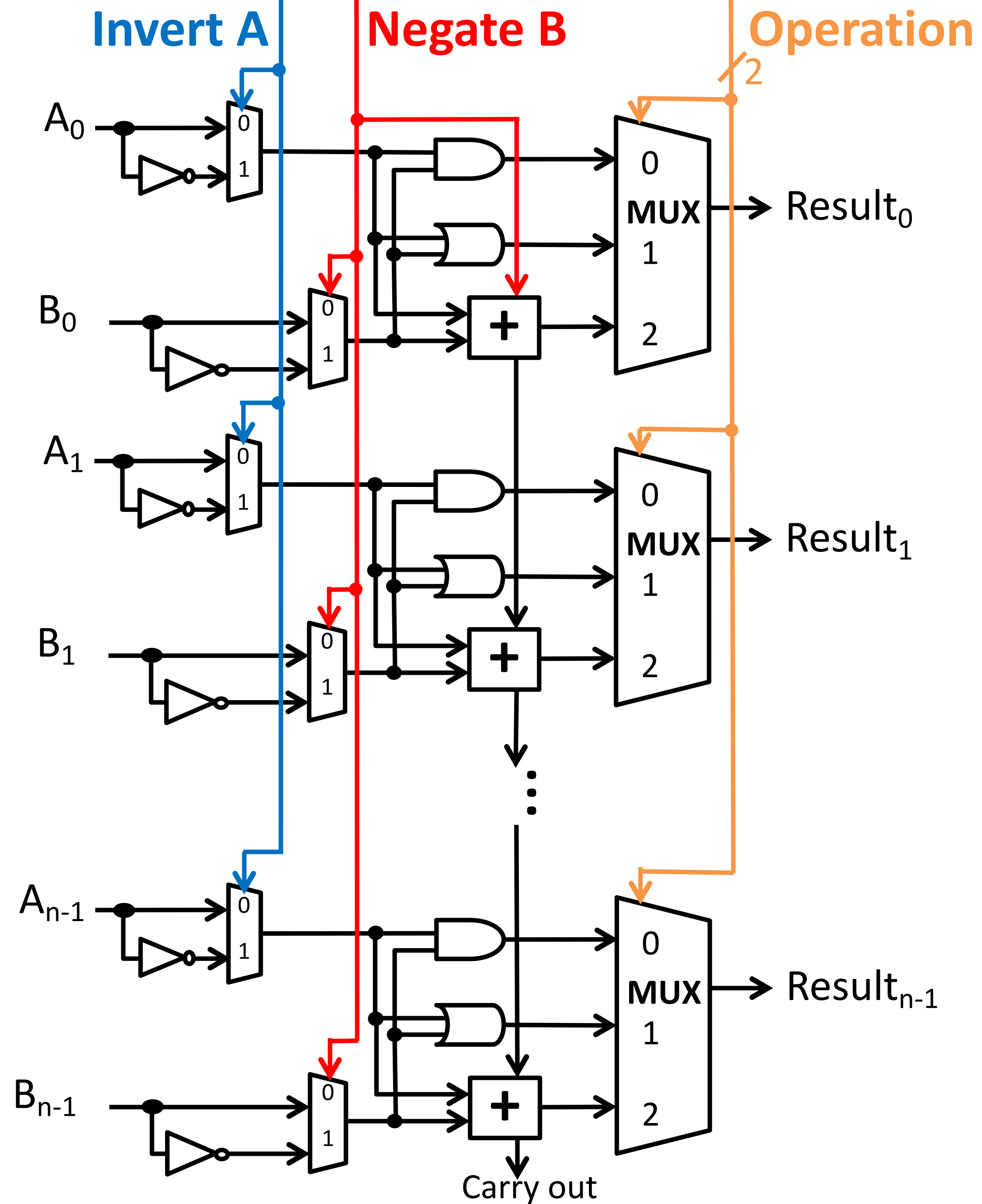
How can we control ALU inputs or add minimal new logic to compute each?

You will implement these in the Circuits Assignment!

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
????	NAND
????	NOR
????	less than
????	equals

You will implement these in the Circuits Assignment!

Controlling the ALU

Abstraction!

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
...	...

Control Lines

Operand A →

ALU → **Result**

Operand B →

Condition Codes

How many different functions (operations) could this ALU theoretically perform?

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
...	...
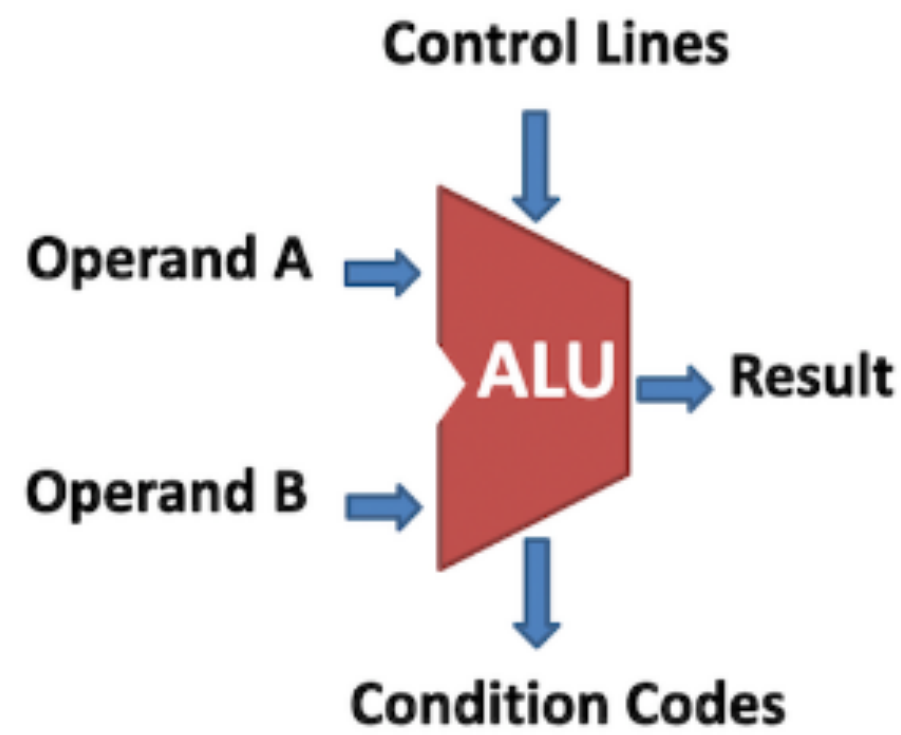
4

8

16

32

None of the above

ALU conditions (additional outputs)

Extra ALU outputs
describing properties of result.

Zero Flag:
1 if result is 00...0 else 0

Sign Flag:
1 if result is negative else 0

Carry Flag:
1 if carry out else 0

(Signed) **Overflow Flag:**
1 if signed overflow else 0

You will implement these in the Circuits Assignment!

Carry in

Operation

A_0

B_0

MUX → $Result_0$

Sum

A_1

B_1

MUX → $Result_1$

Sum

A_{n-1}

B_{n-1}

MUX → $Result_{n-1}$

Sum

Carry out