WELLESLEY

Arithmetic Logic

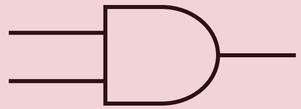
adders

Arithmetic Logic Unit

Motivation: how do we go from code to gates?

```
int count_odds(int array[10]) {
    int count = 0;
    for (int i = 0; i < 10; i++) {
        count += array[i] & 0x1;
    }
    return count;
}
```

AND

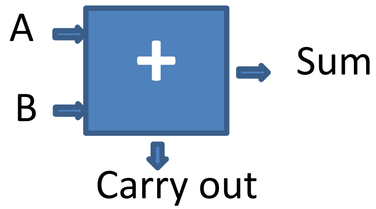


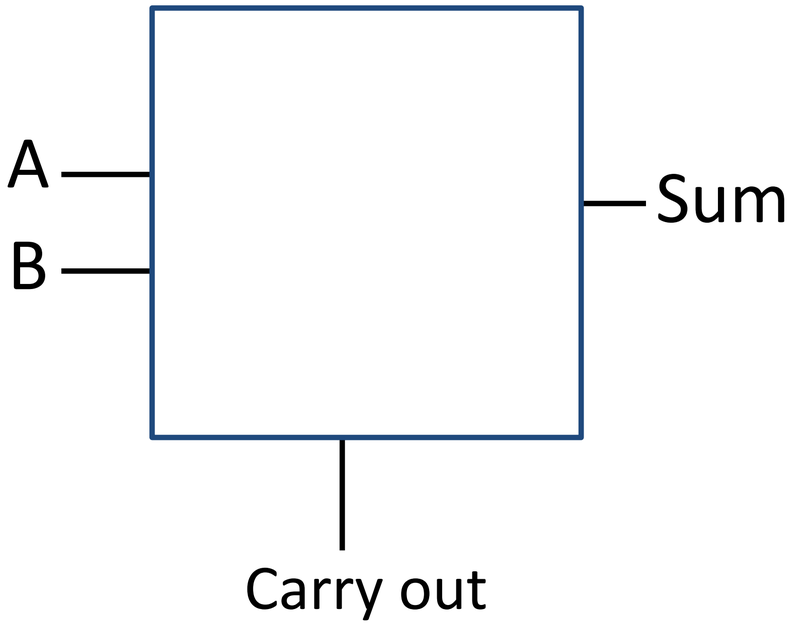
ADD

???

Addition: 1-bit ***half*** adder

ex

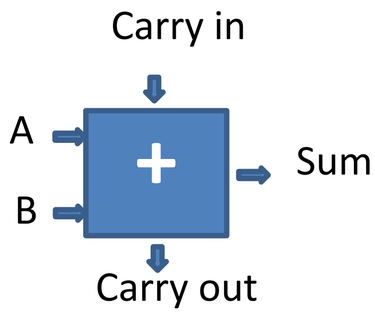


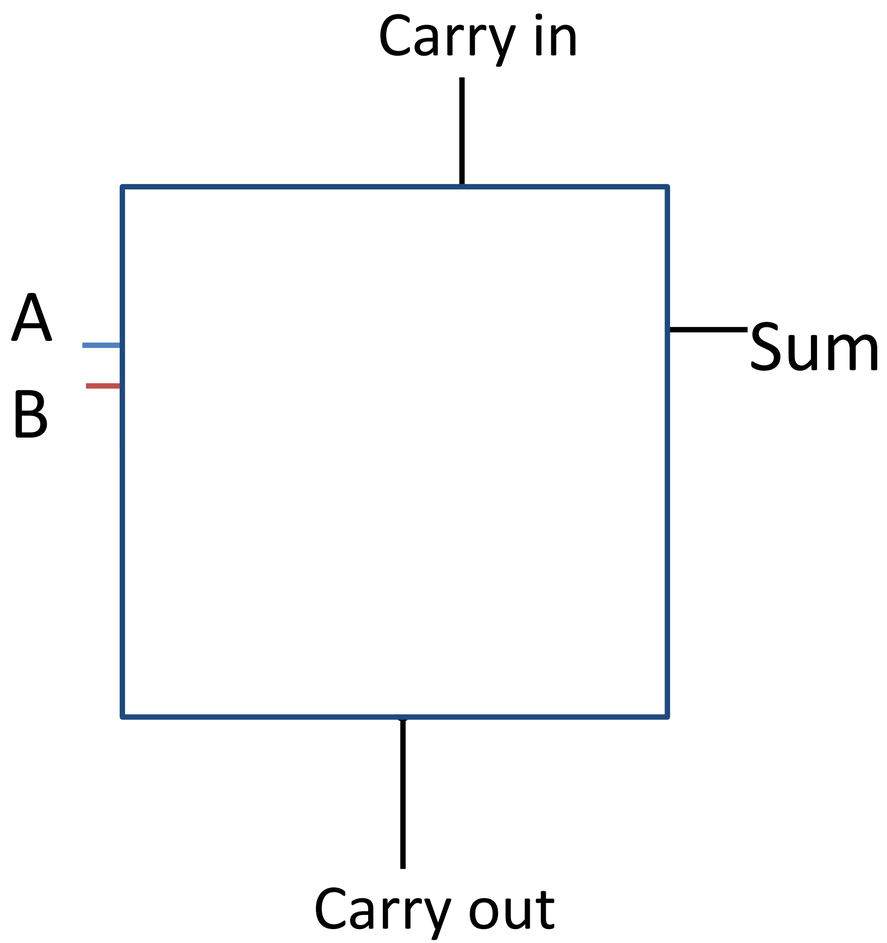


A	B	Carry Out	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

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

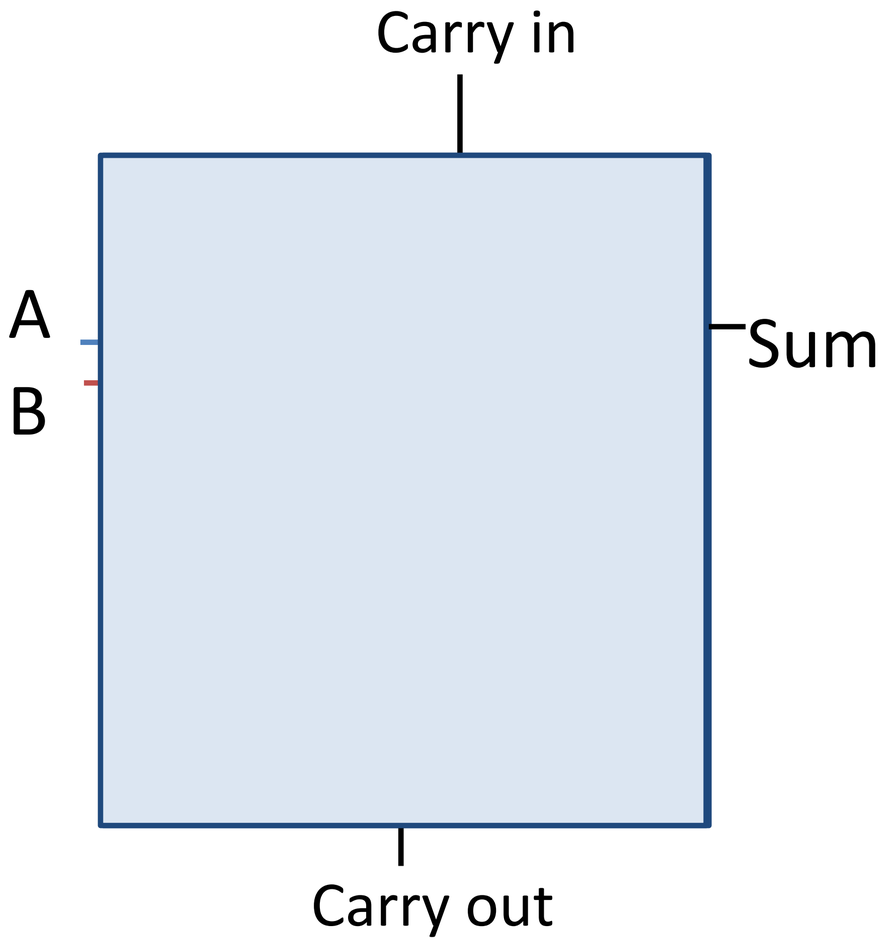
Addition: 1-bit ***full*** adder

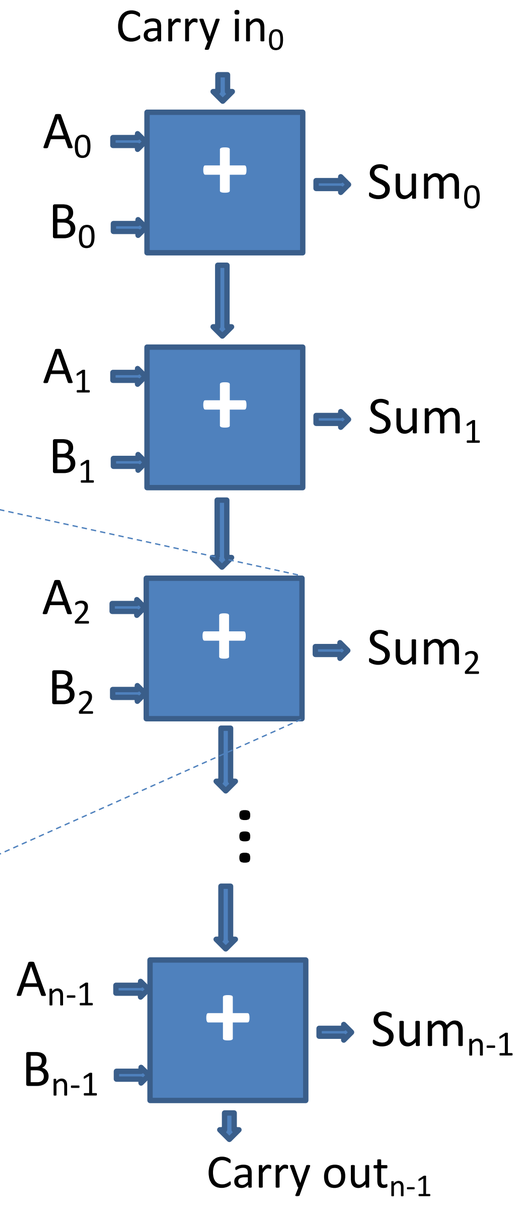




Carry in	A	B	Carry Out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

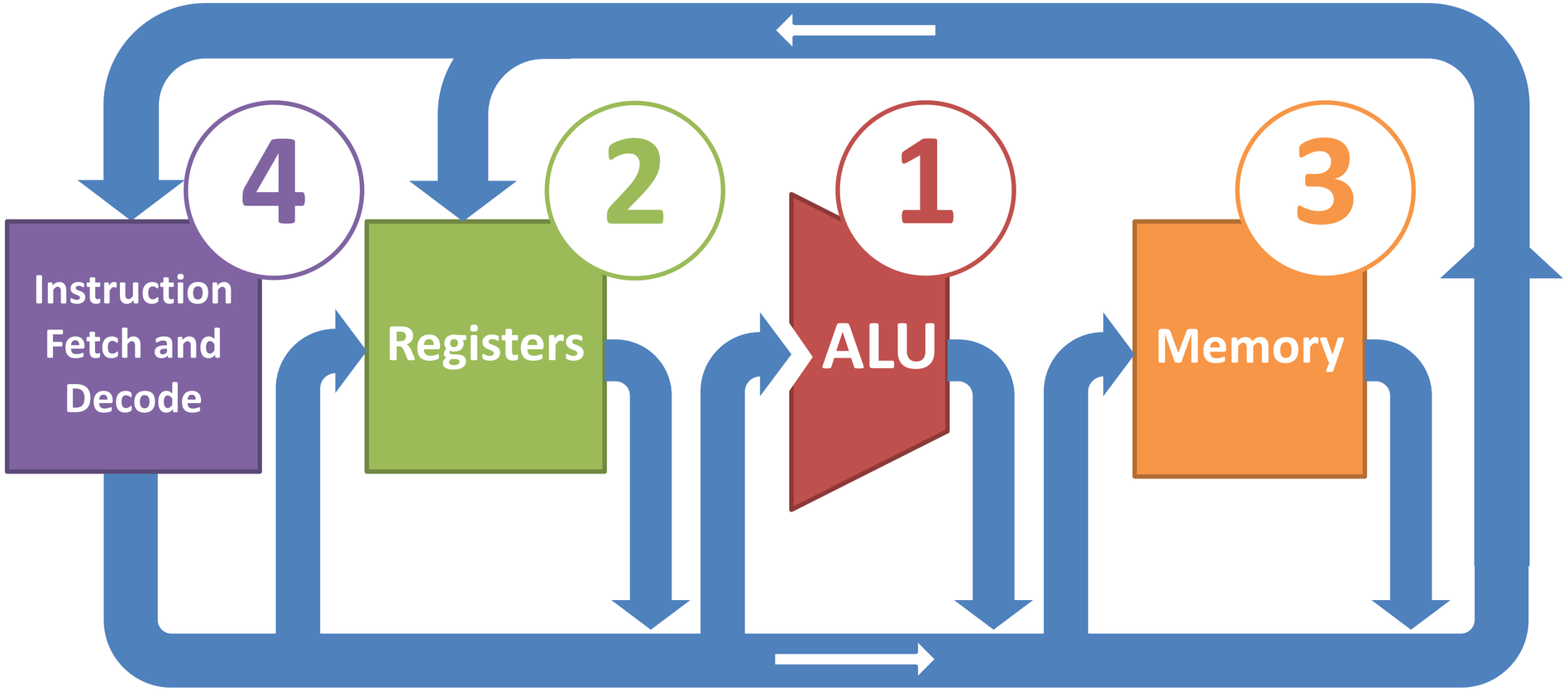




There are faster, more complicated ways too…

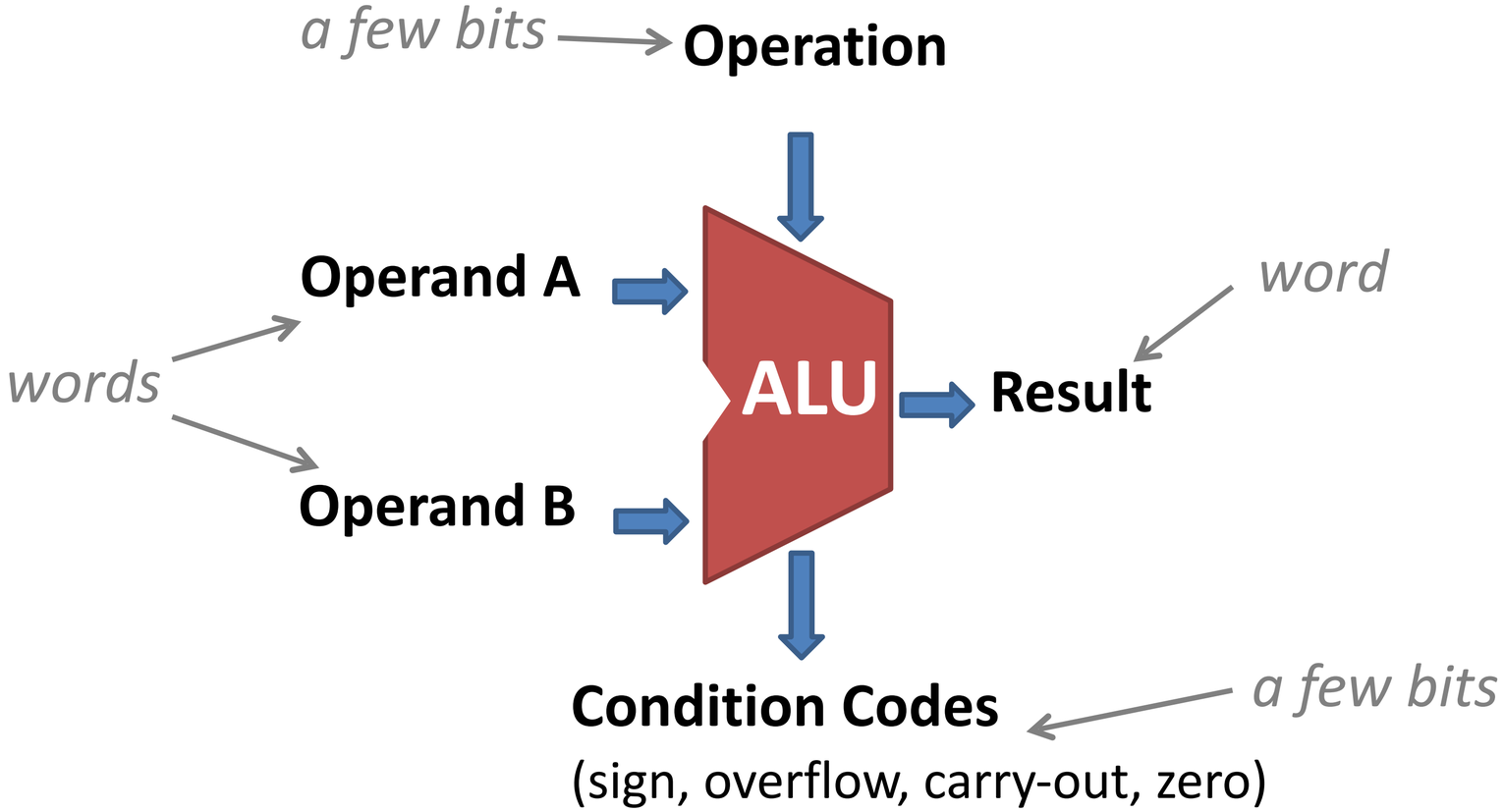
Processor Components

Abstraction!



Arithmetic Logic Unit (ALU)

1



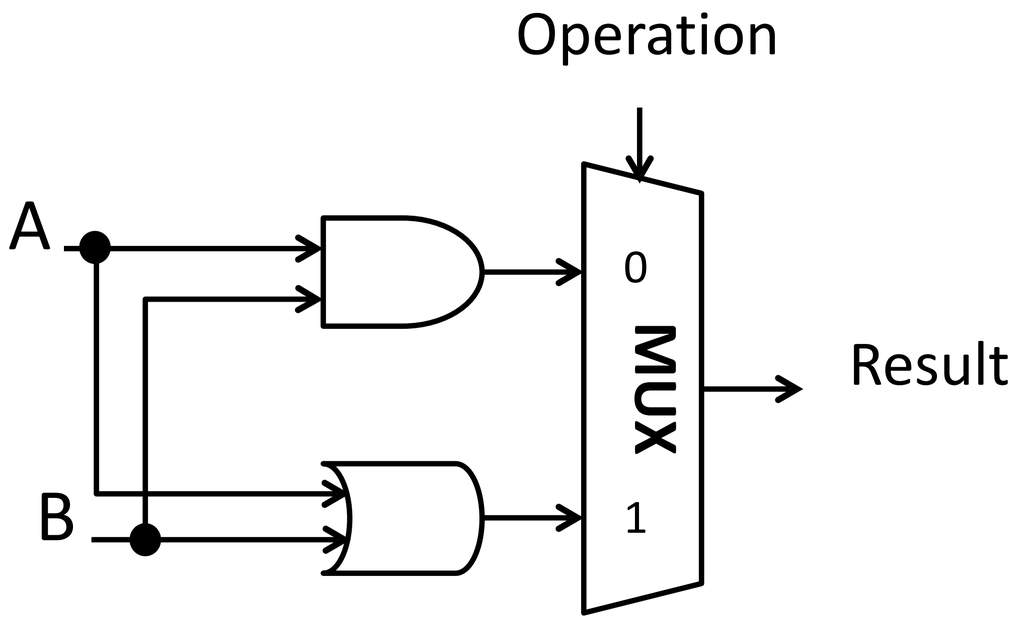
Hardware unit for arithmetic and bitwise operations.

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

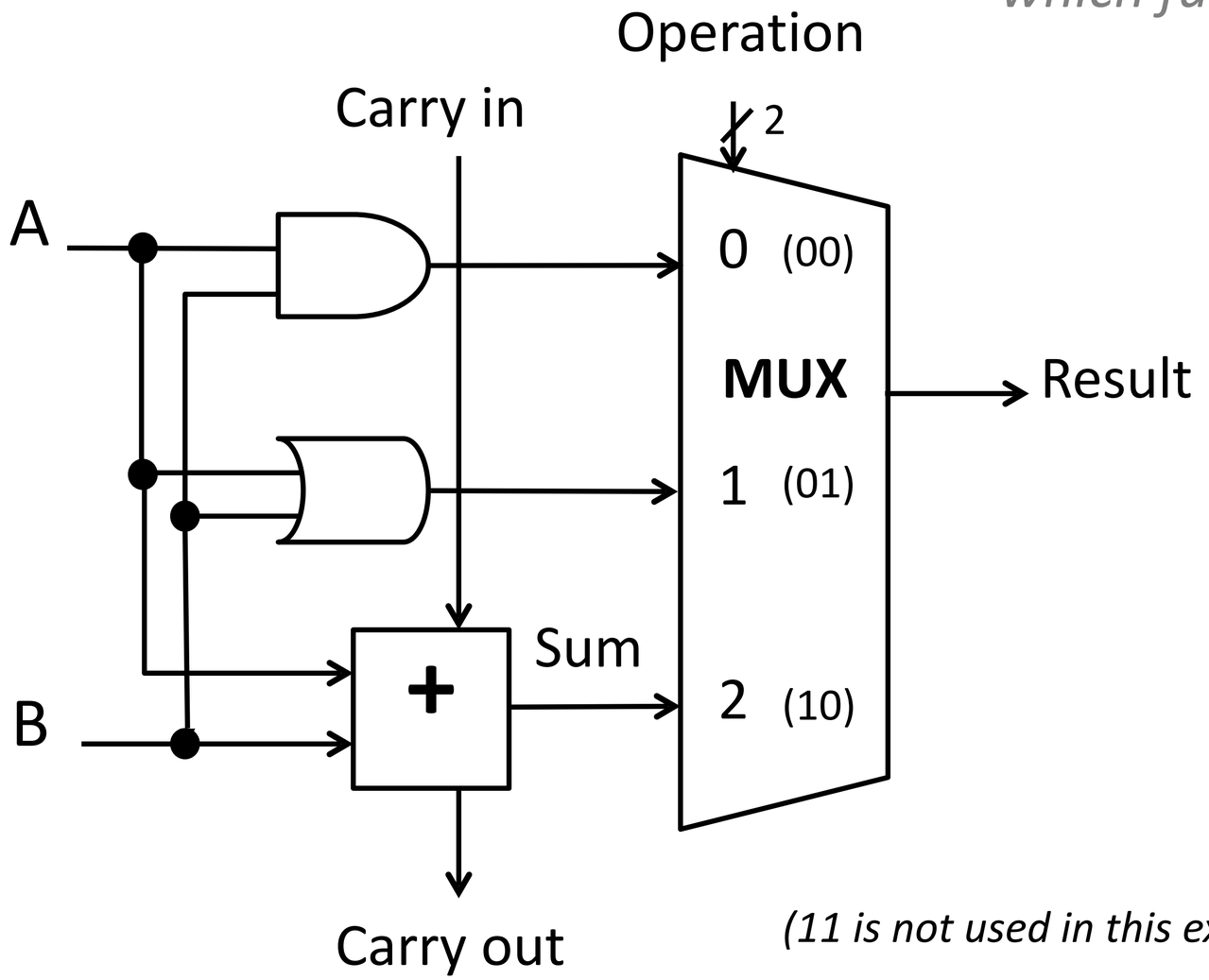


Op	A	B	Result
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

ex

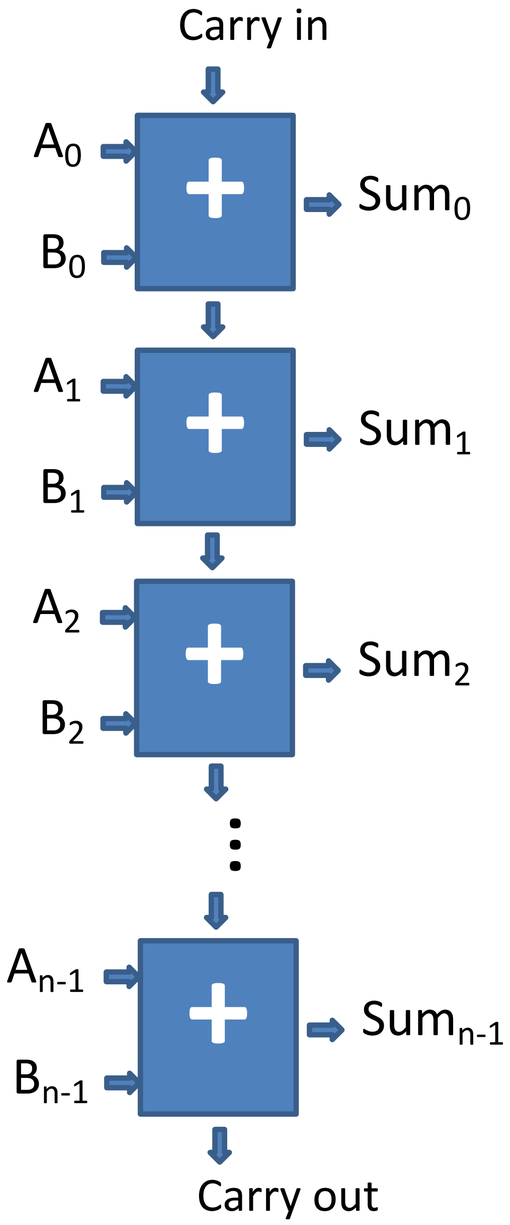
1-bit ALU: 3 operations

Used as selector, chooses which function to perform

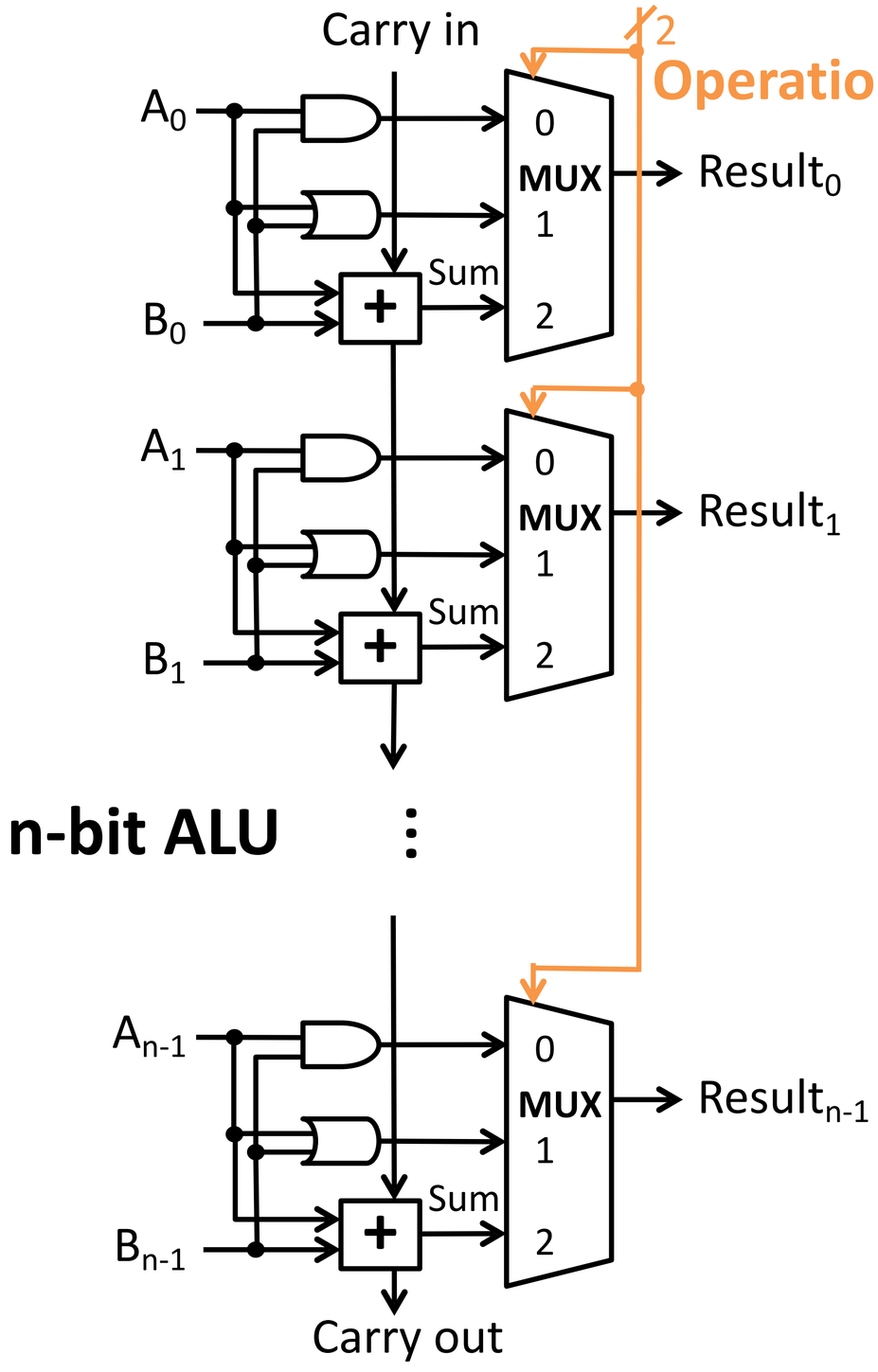


(11 is not used in this example ALU!)

n-bit ripple carry adder



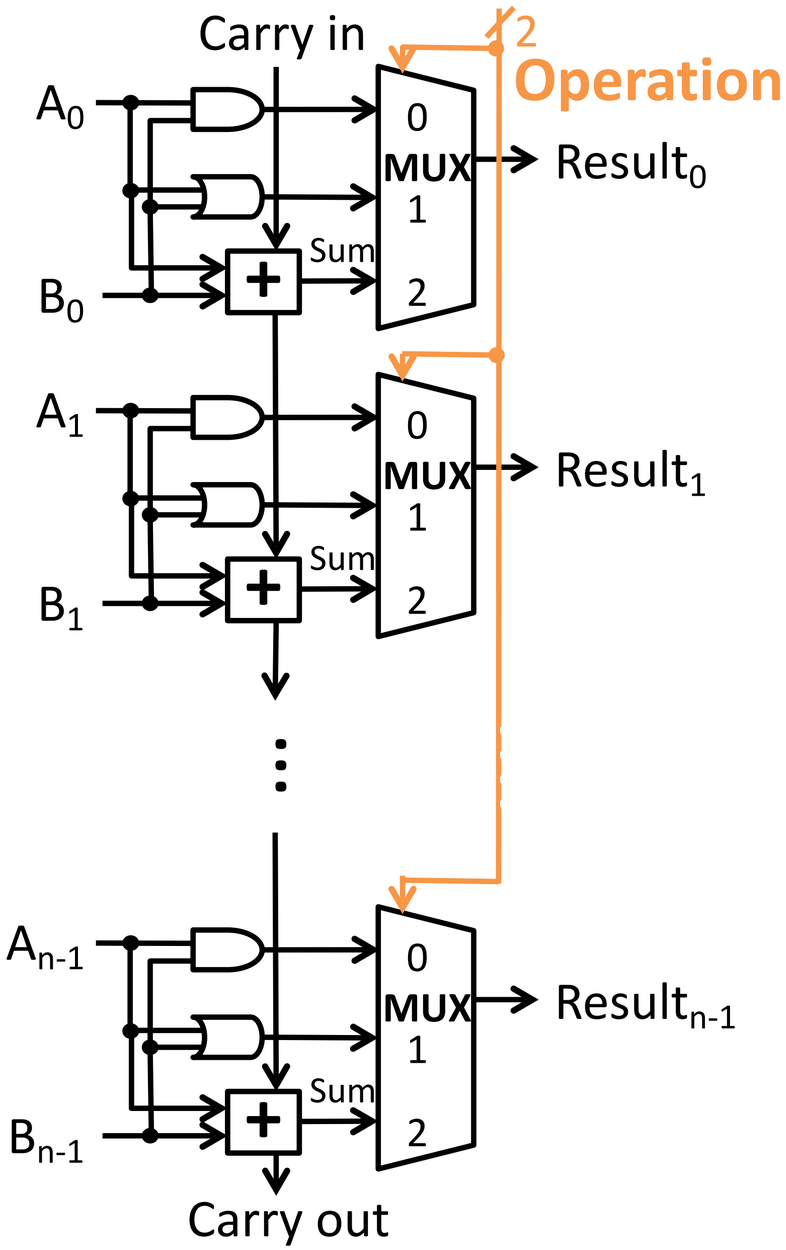
n-bit ALU



Use the same selector for every 1-bit ALU

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`**?

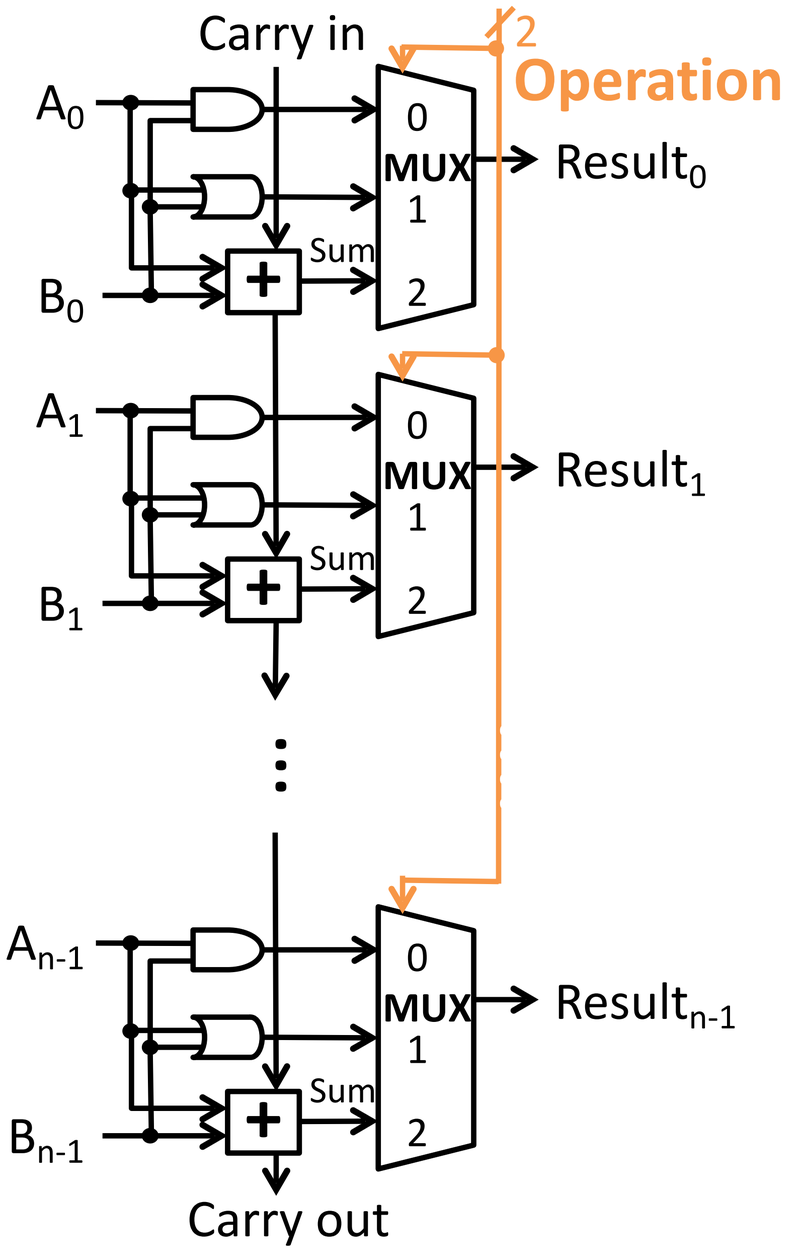
Recall:

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

Plan:

Feed bitwise-not B into the adder

Add an extra 1: how?



Include subtraction

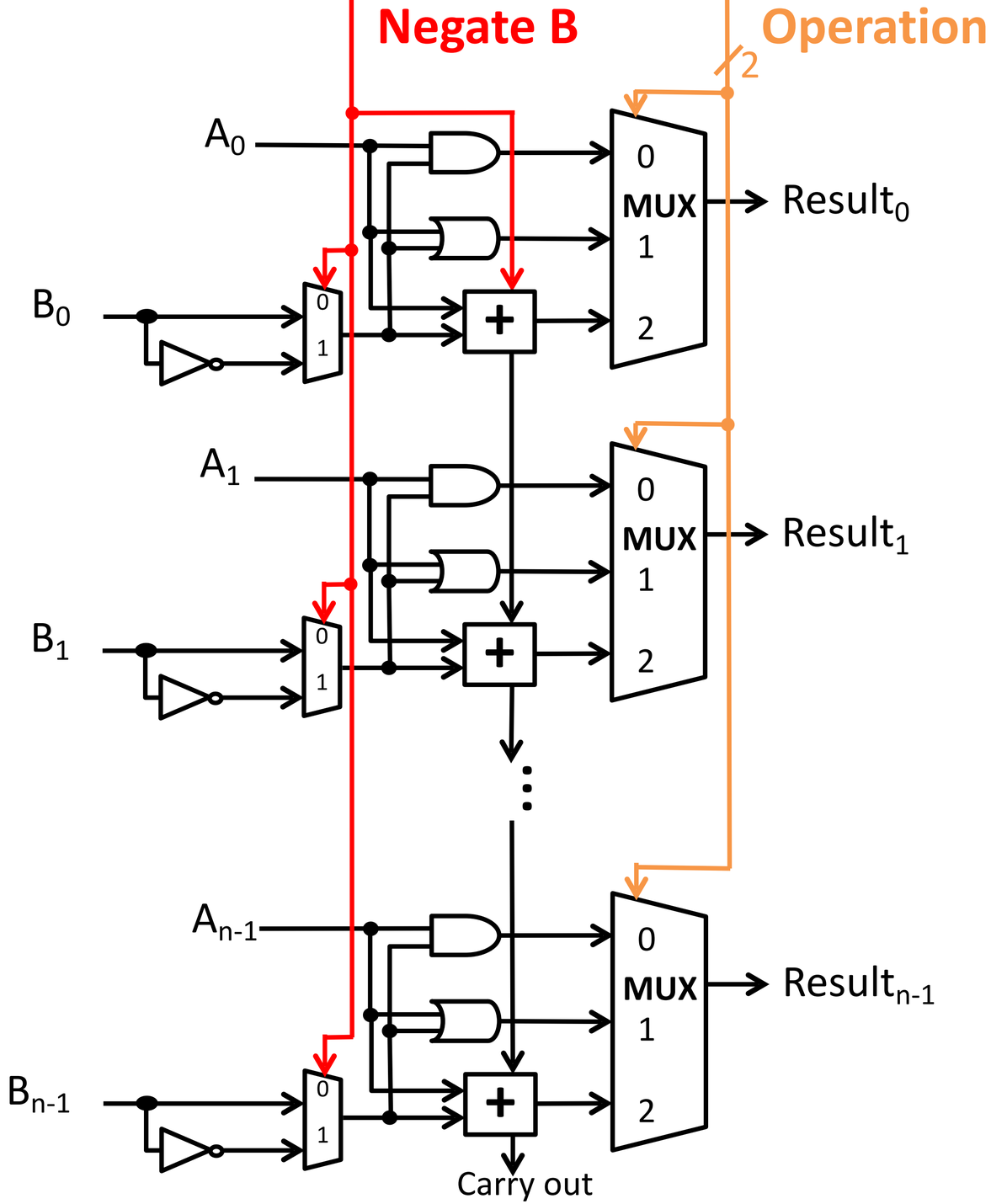
Plan to compute `A–B`:

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

Key insight:

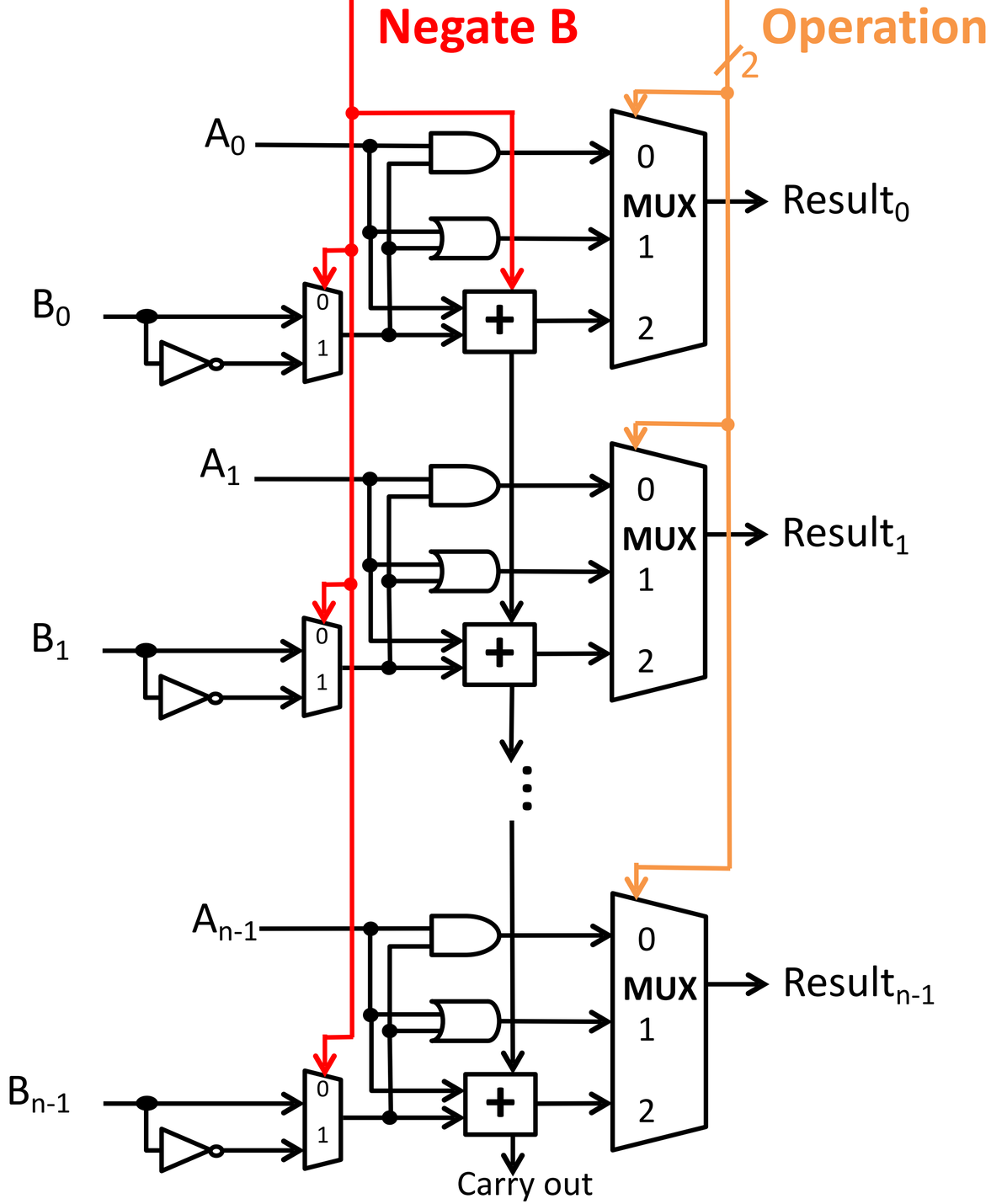
The *same* selector bit (0 or 1) can be used for both!

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
…	…



A NAND B

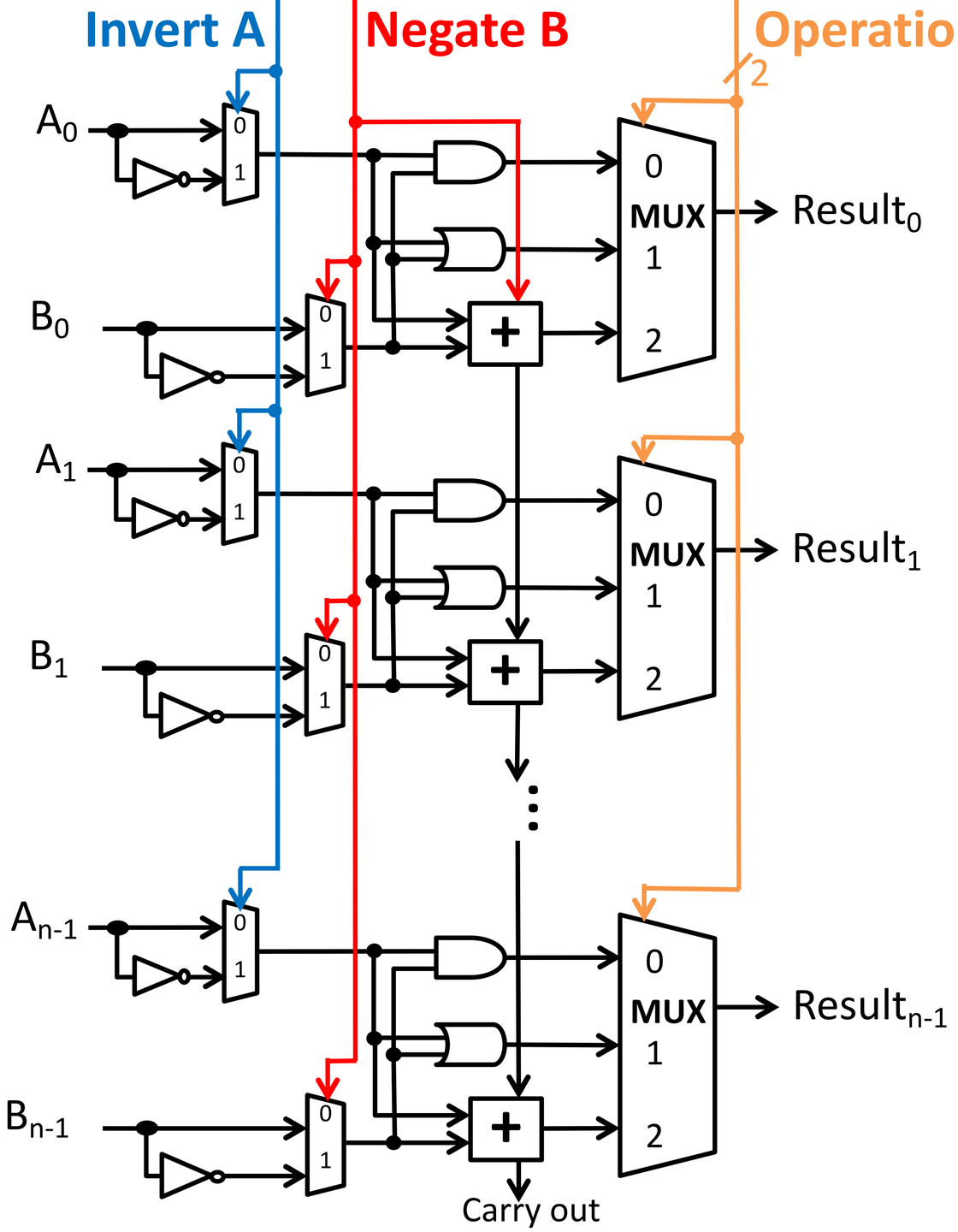
A NOR B

A<B

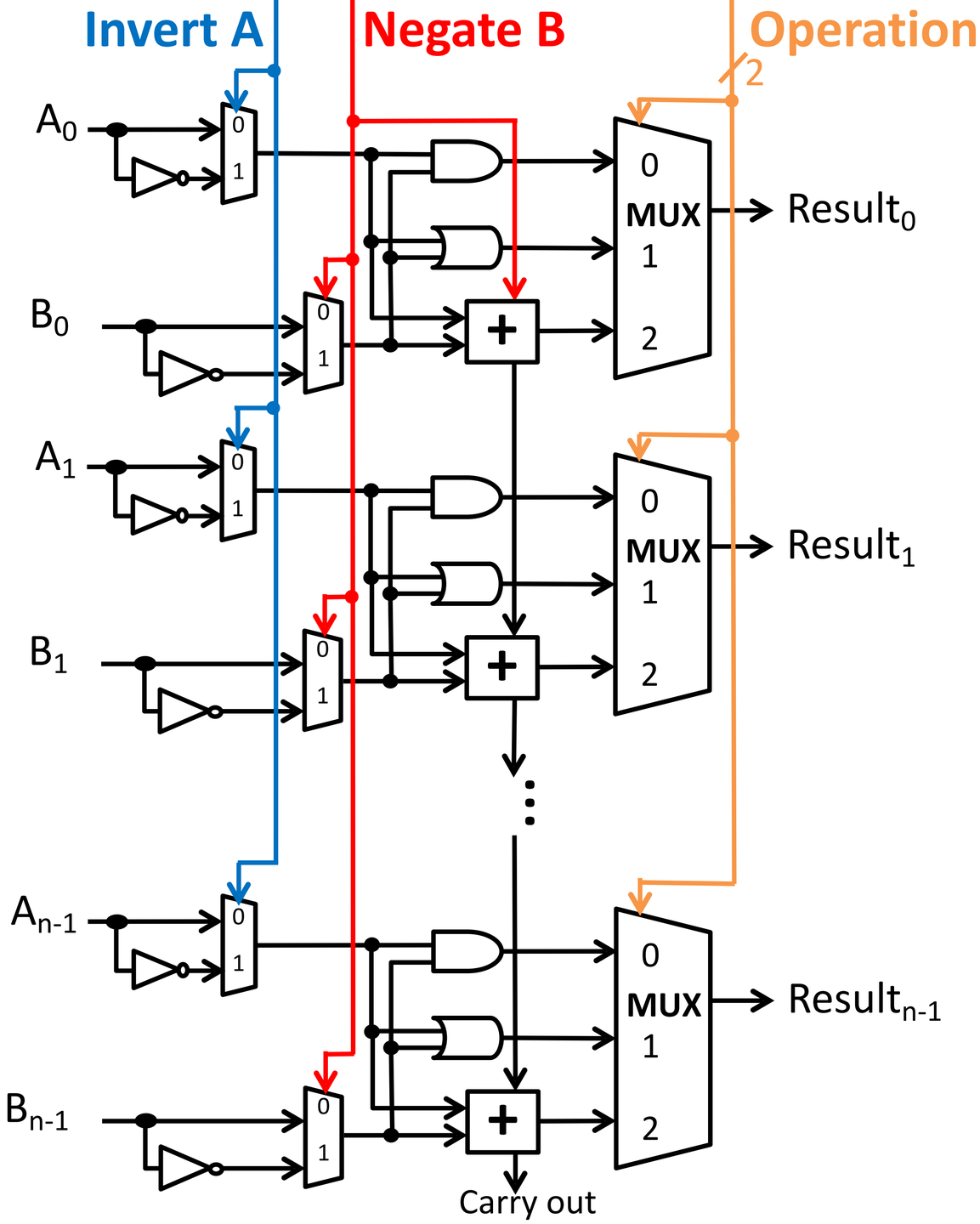
A==B

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

You will implement some of these in the Arch Assignment!

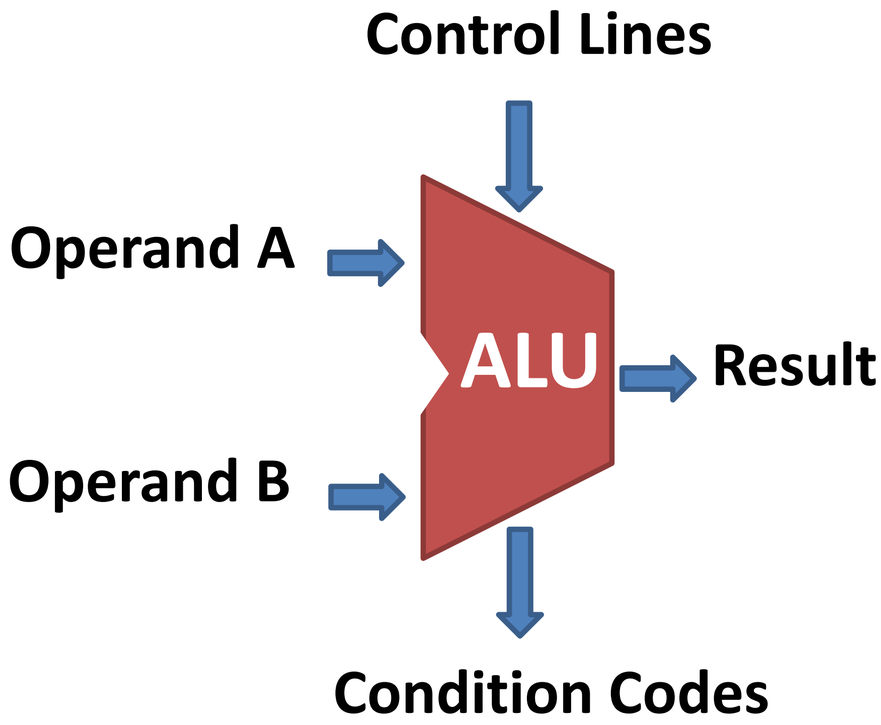


ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
????	NAND
????	NOR
????	less than
????	equals



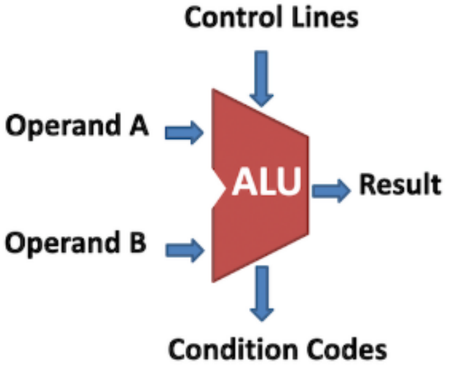
Controlling the ALU

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



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

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



4

8

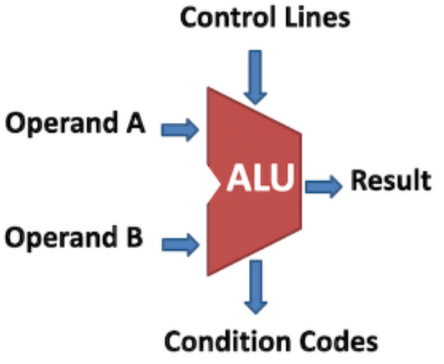
16

32

None of the above

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

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



4
0%

8
0%

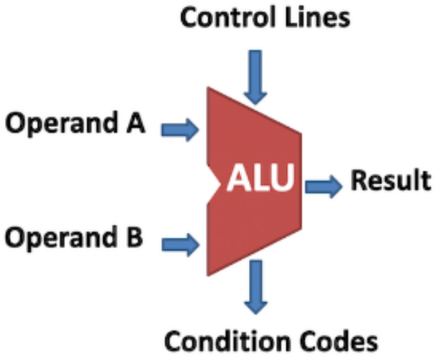
16
0%

32
0%

None of the above
0%

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

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



4
0%

8
0%

16
0%

32
0%

None of the above
0%

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 Arch Assignment!