CS 31: Introduction to Computer Systems

06: Computer Architecture

02-06-2025

SWARTHMORE COLLEGE

Reading Quiz

- Note the red border!
- 1 minute per question
- No talking, no laptops, phones during the quiz

<u>Check your frequency:</u>

- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:

1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA
   vote status light will indicate success

What we will learn this week

1. Introduction to C
   - Data organization and strings
   - Functions

2. Computer Architecture
   - Machine memory models
   - Digital signals
   - Logic gates

Functions: Specifying Types

Need to specify the return type of the function, and the type of each parameter:

```
<return type> <func name> ( <param list> ) {
    // declare local variables first
    // then function statements
    return <expression>;
}

// my_function takes 2 int values and returns an int
int my_function(int x, int y) {
  int result;
  result = x;
  if(y > x) {
    result = y+5;
  }
  return result*2;
}
```

Compiler will yell at you if you try to pass the wrong type!

Passing Arrays

- An array argument's value is its base address

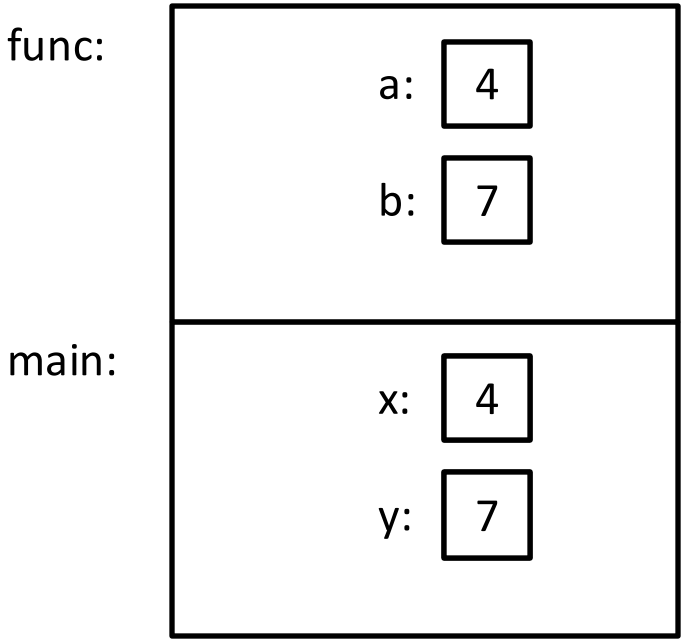
```
int main(void){
  int values[10];
  foo(values, 10);
}
void foo(int arr[], int n){
  arr[2] = 6;
}
```

array base address

Function Arguments

- Arguments are **passed by value**
  - The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main(void) {
    int x, y;  // declare two integers
    x = 4;
    y = 7;
    y = func(x, y);
    printf(“%d, %d”, x, y);
}
```



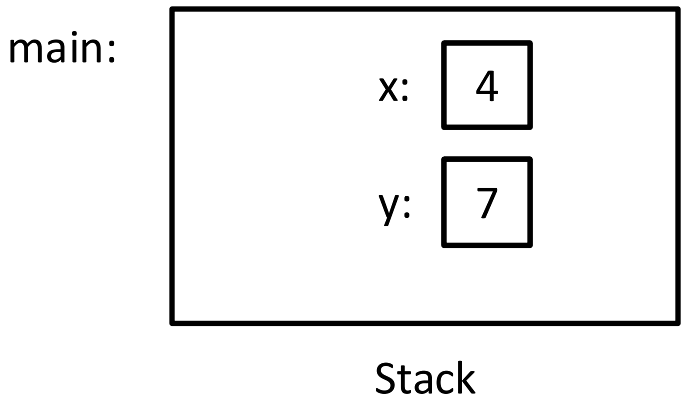
Stack

Function Arguments

- Arguments are **passed by value**
  - The function gets a separate copy of the passed variable

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main(void) {
    int x, y;  // declare two integers
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

It doesn't matter what func does with a and b. The value of x in main doesn't change.

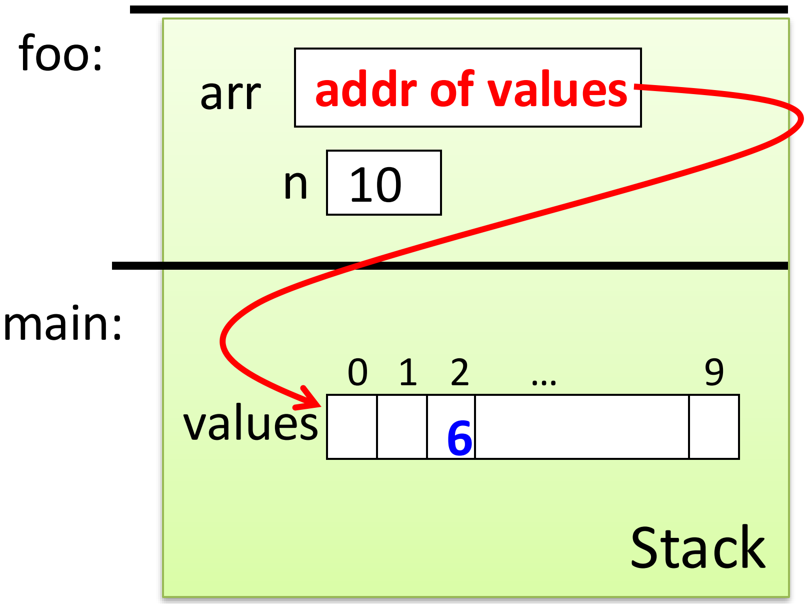


Passing Arrays

- An array argument's value is its base address

```
int main(void){
   int values[10];
   foo(values, 10);
}
void foo(int arr[], int n){
   arr[2] = 6;
}
```

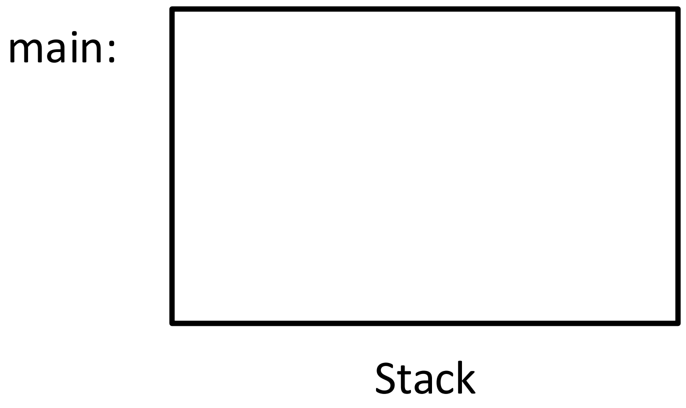
array base address



Function Arguments

- Arguments can be pointers!
  - The function gets the address of the passed variable!

```
void func(int *a) {
    *a = *a + 5;
}

int main(void) {
    int x = 4;

    func(&x);
    printf("%d", x);
}
```

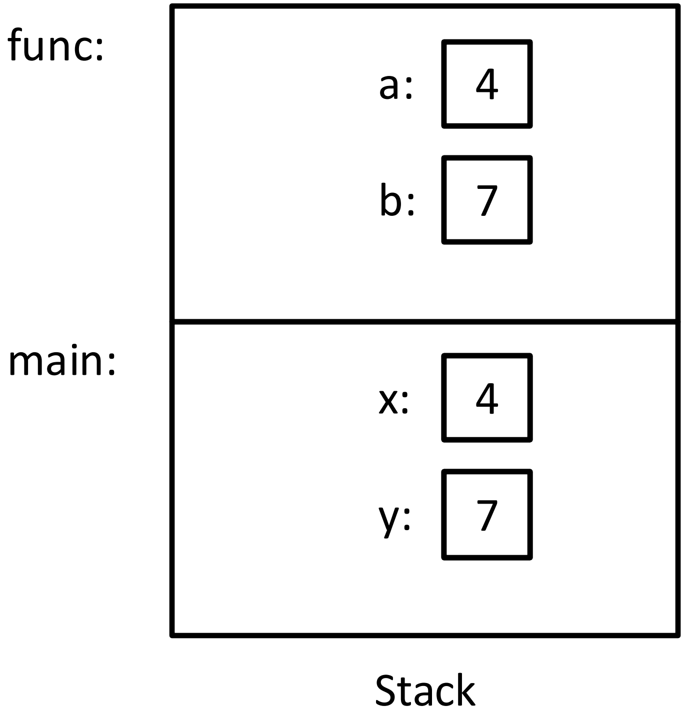


Function Arguments: passed by value

- Arguments are **passed by value**
  - The function gets a separate copy of the passed variable

It doesn't matter what func does with a and b. The value of x in main doesn't change.

```
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main(void) {
    int x, y;   // declare two integers
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

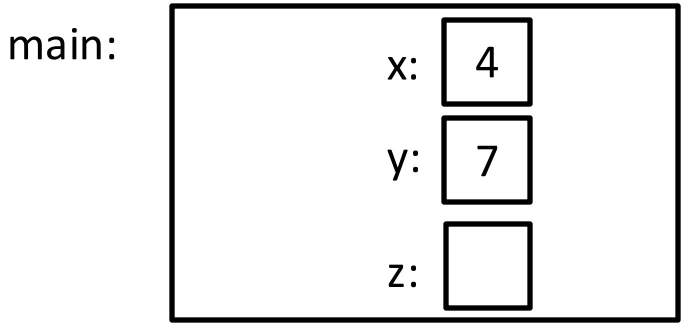


Function Arguments: passed by memory address

- Arguments are **passed by memory address**
  - The function gets the address of the passed variable

```c
int func(?, ?)
{
    //computations adding 4 to x
    return //x-y ;
}

int main(void) {
    int x, y, z;
    x = 4;
    y = 7;
    z = func(&x, &y);
    printf("%d, %d", x, y);
}
```

what data type are a and b? they can't be integers. Recall we have passed in a memory address in the function call.



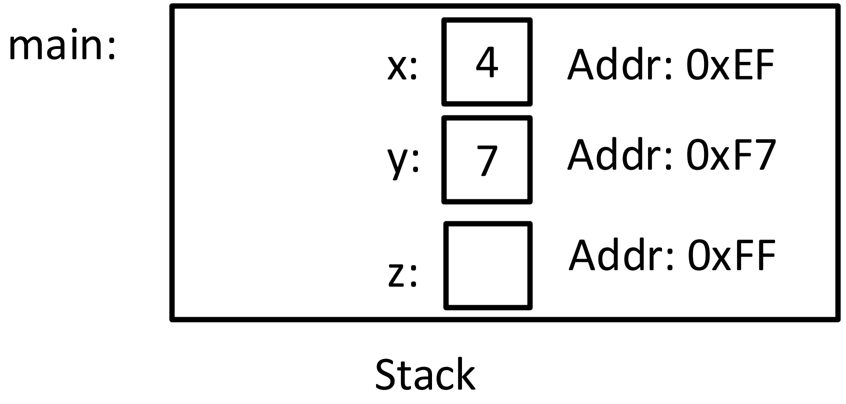
Stack

Function Arguments: passed by memory address

- Arguments are **passed by memory address**
  - The function gets the address of the passed variable

```
int func(<memory addressss of int> a, <memory addressss of int> b) {
     //computations adding 4 to x
    return //x-y ;
}

int main(void) {
    int x, y, z;  // declare three integers
    x = 4;
    y = 7;
    z = func(&x, &y);
    printf(“%d, %d”, x, y);
}
```

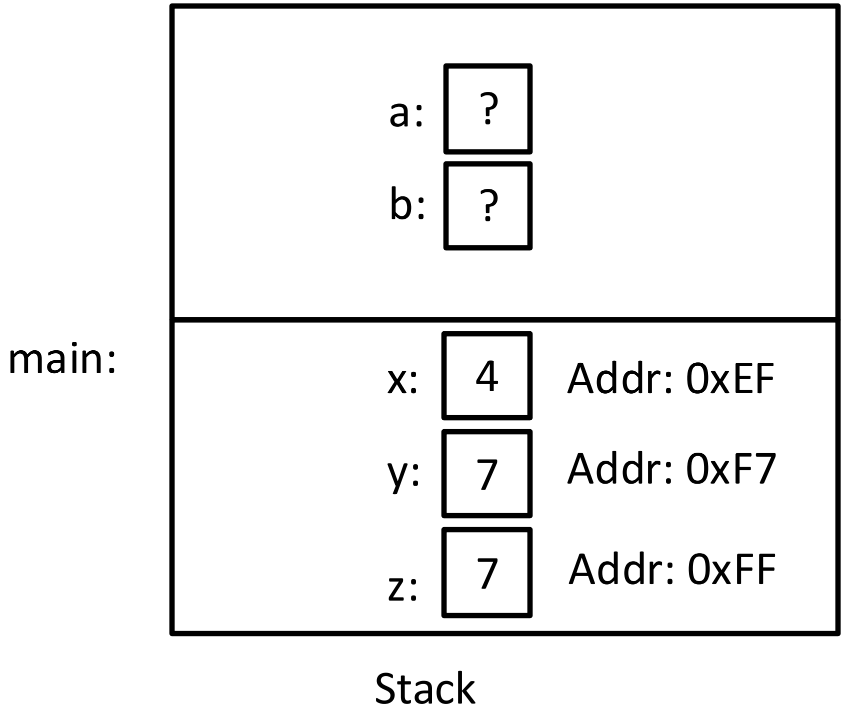
what data type are a and b? they can’t be an integers. they are type **memory addresses of an int**



Function Arguments: passed by memory address

- Arguments are **passed by memory address**
  - The function gets the address of the passed variable

```
int func(<memory address of int> a, <memory address of int> b) {
     //computations adding 4 to x
    return //x-y ;
}

int main(void) {
    int x, y, z;  // declare three integers
    x = 4;
    y = 7;
    z = func(&x, &y);
    printf("%d, %d", x, y);
}
```

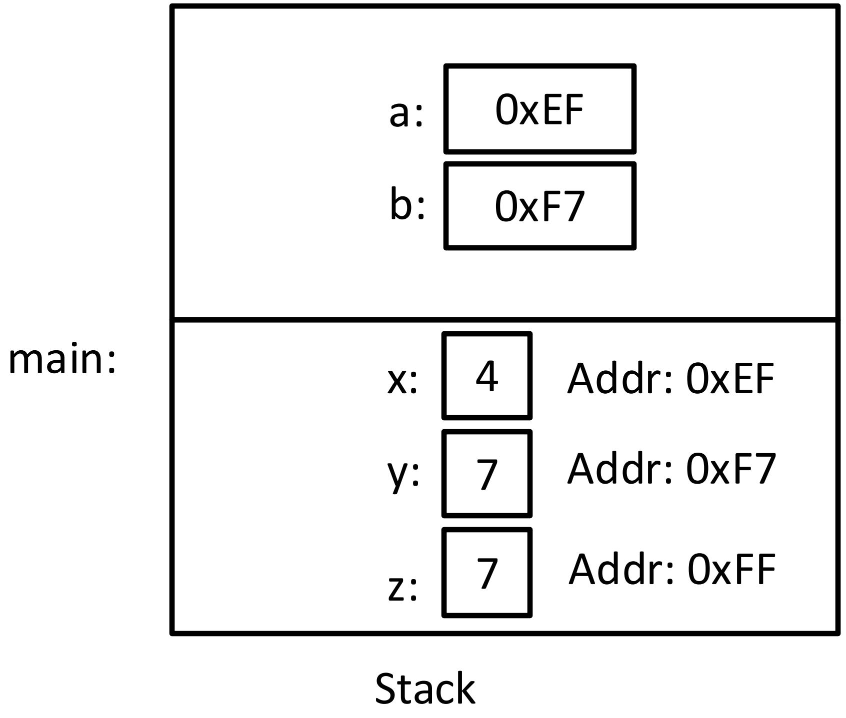


Function Arguments: passed by memory address

- Arguments are **passed by memory address**
  - The function gets the address of the passed variable

In func, we want to add 4 to x. How do we do that?

```
int func(<memory address of int> a, <memory address of int> b) {
     //computations adding 4 to x
    return //x-y ;
}

int main(void) {
    int x, y, z;  // declare three integers
    x = 4;
    y = 7;
    z = func(&x, &y);
    printf("%d, %d", x, y);
}
```

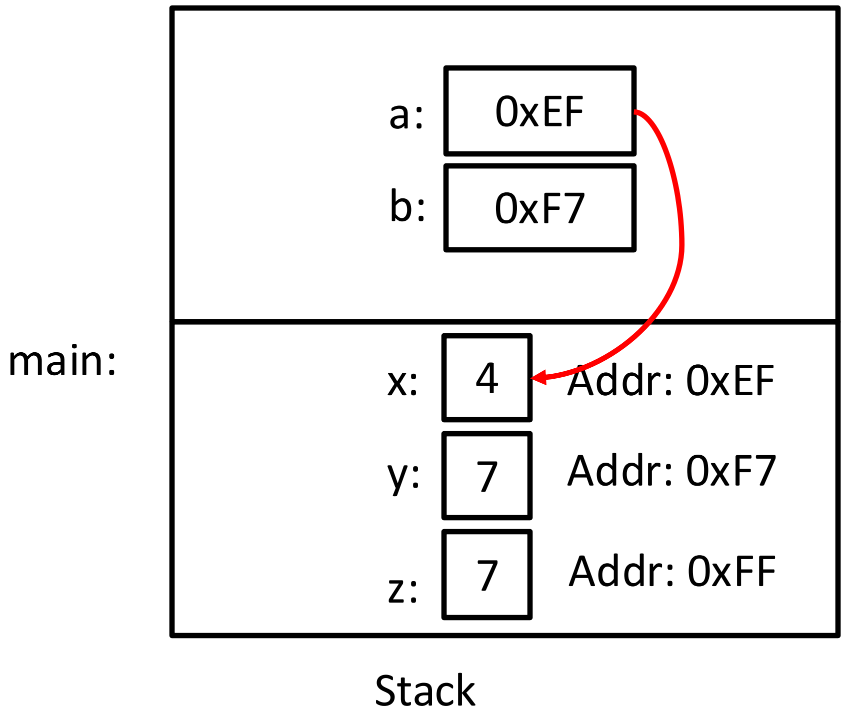


Function Arguments: passed by memory address

- Arguments are **passed by memory address**
  - The function gets the address of the passed variable

We need the **value at the memory address of a.**

```
int func(<memory address of int> a, <memory address of int> b) {
     //computations adding 4 to x
    return //x-y ;
}

int main(void) {
    int x, y, z;  // declare three integers
    x = 4;
    y = 7;
    z = func(&x, &y);
    printf("%d, %d", x, y);
}
```

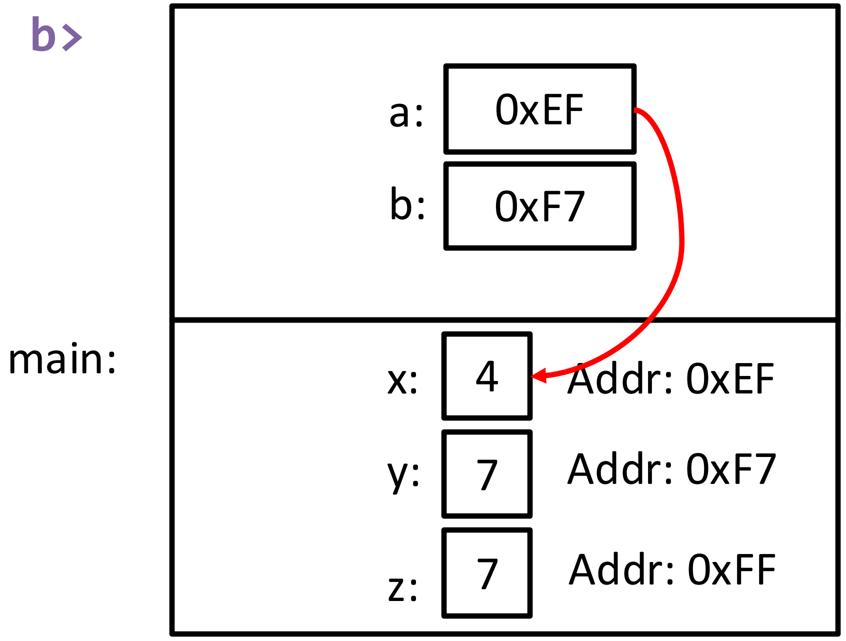


Function Arguments: passed by memory address

- Arguments are **passed by memory address**
  - The function gets the address of the passed variable

We need the **value at the memory address of a** and **the value at the memory address of b**

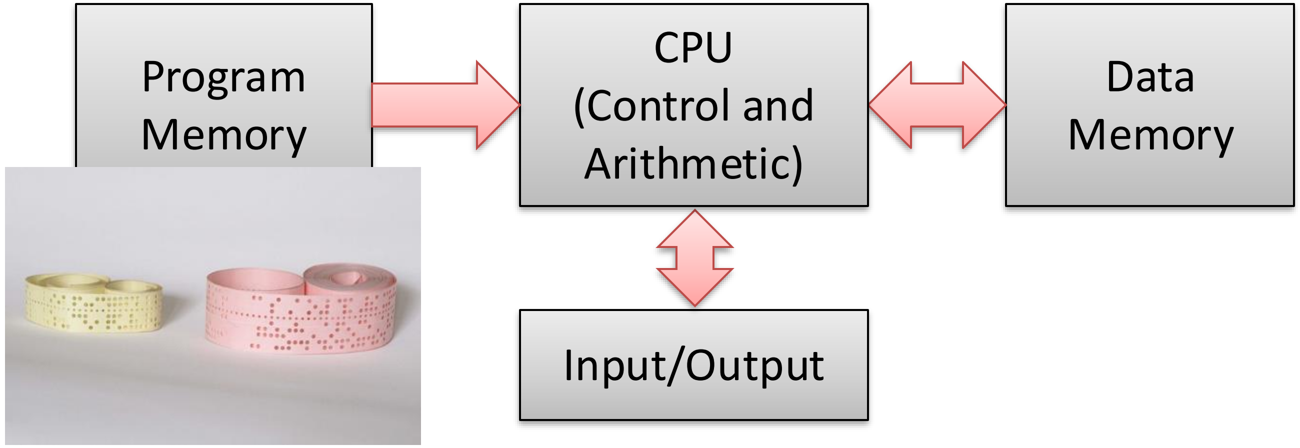
```
int func(<memory address of int> a, <memory address of int> b) {
     <value at addr a> += 5;
    return <value at addr a> - <value ad addr b>
}

int main(void) {
    int x, y, z;  // declare three integers
    x = 4;
    y = 7;
    z = func(&x, &y);
    printf("%d, %d", x, y);
}
```



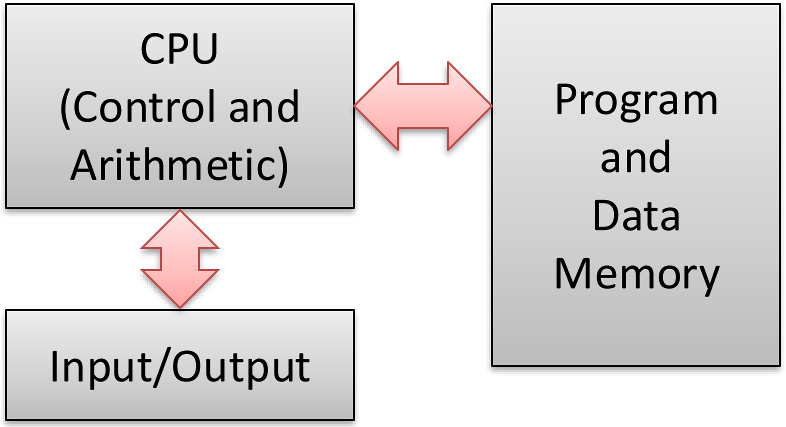
Stack

Hardware Models (1940's)

- Harvard Architecture:



- Von Neumann Architecture:



Von Neumann Architecture 1945

Computer is a generic computing machine

- Can be used to compute anything that is computable
- Based on Alan Turing's Universal Turing Machine

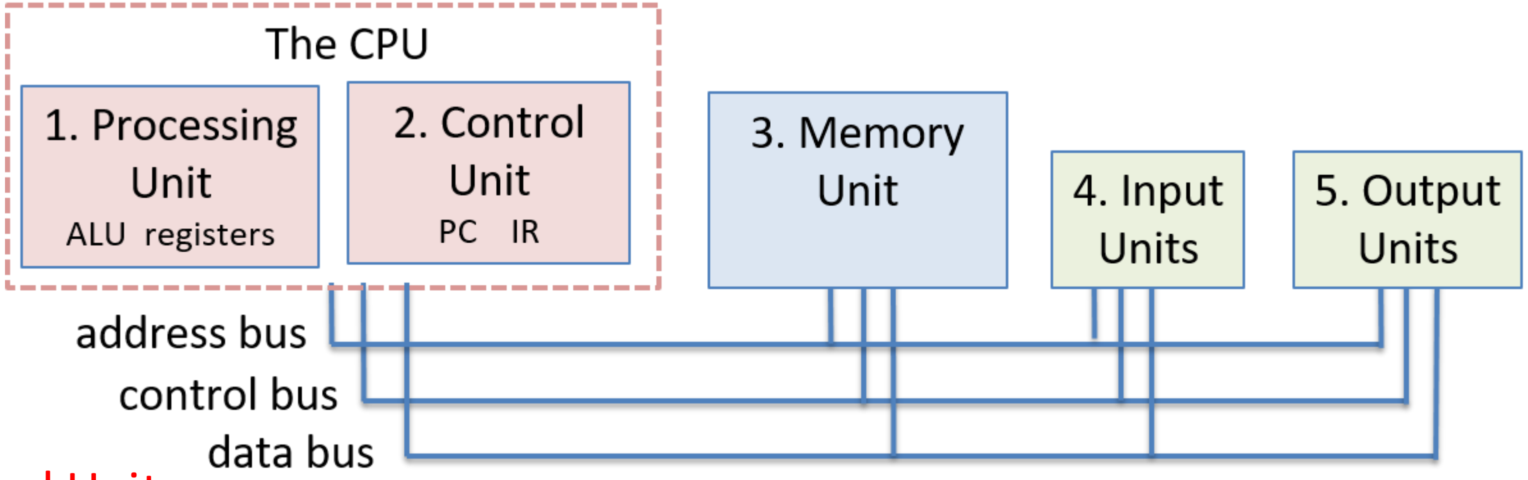
Uses a stored program model

- both program & data loaded into computer memory
- No distinction between data & instructions in memory
    - Earlier computers used fixed program encoded on machine, data loaded and run by fixed program

All modern computers based on the Von Neumann model

Von Neumann Model

5 units connected by buses (wires) to communicate



Processing & Control Units:

- implement CPU \execute program instructions on program data

Memory: stores program instructions and data

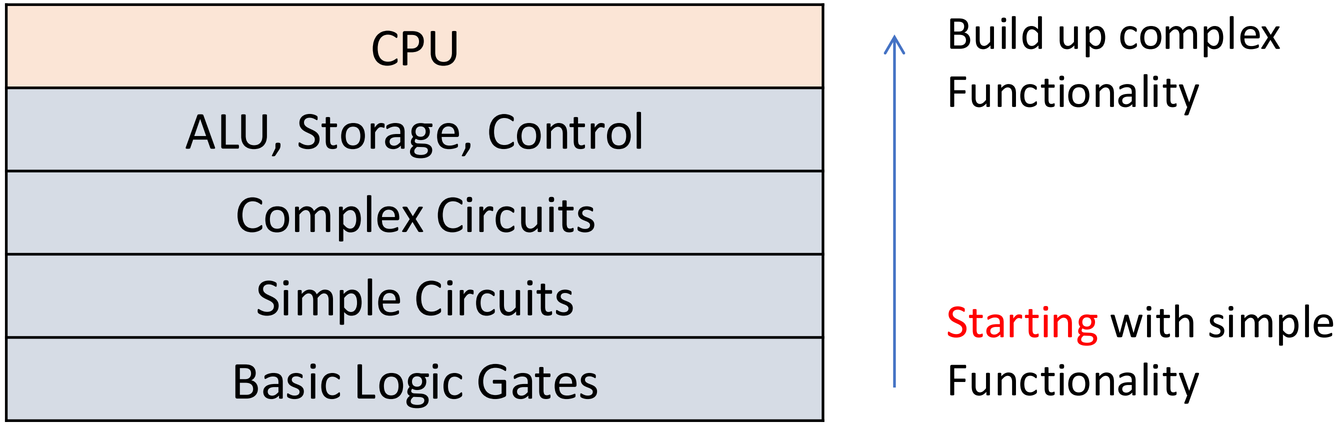
- memory is addressable: addr 0, 1, 2, ...

Input, Output: interface to compute

- trigger actions: load program, initiate execution, ...
- display/store results: to terminal, save to disk, ...

Our Goal: Build a CPU (model)

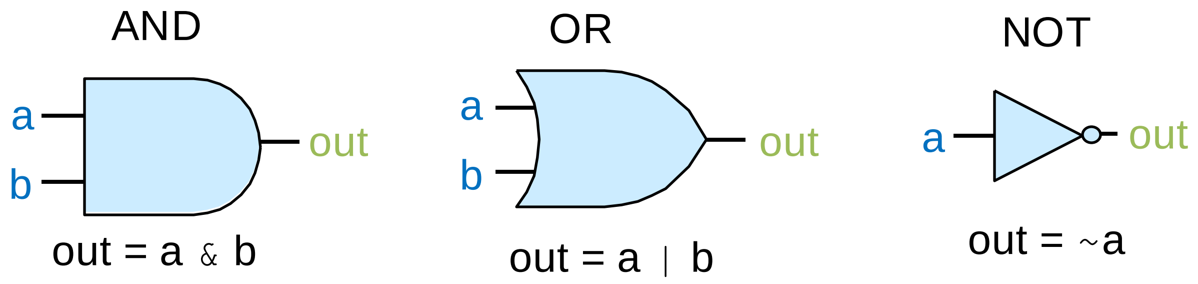
Start with very simple functionality, and add complexity



Logic Gates

Input: Boolean value(s) (high and low voltages for 1 and 0)

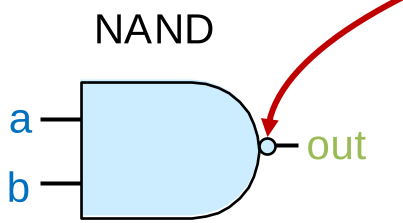
Output: Boolean value result of Boolean function
Always present, but may change when input changes

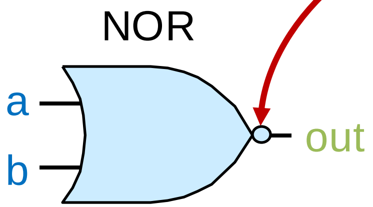


A	B	A & B	A | B	~A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

More Logic Gates

Note the circle on the output. This circle means bitwise "not" (flip bits).

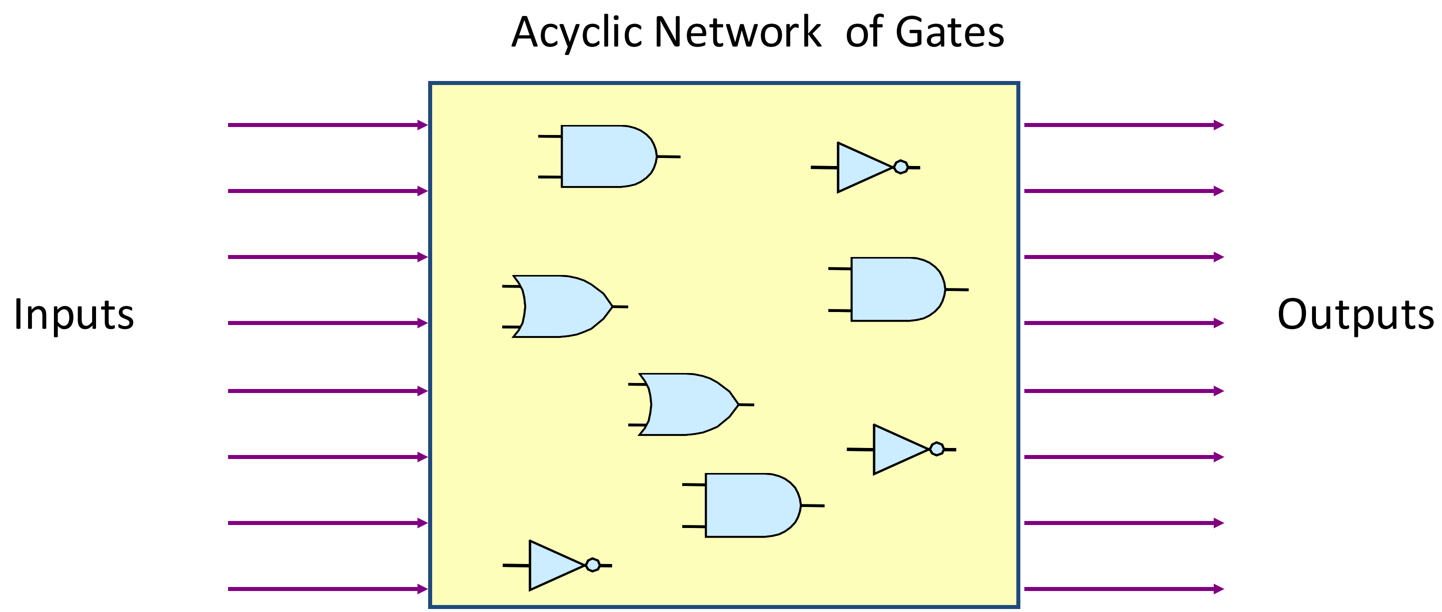




A	B	A NAND B	A NOR B
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Combinational Logic Circuits

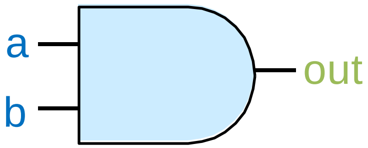
- Build up higher level processor functionality from basic gates



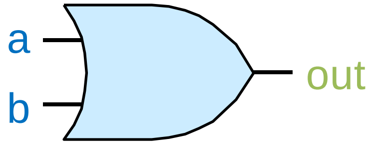
- Outputs are boolean functions of inputs
- Outputs continuously respond to changes to inputs

What does this circuit output?

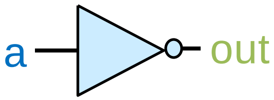
AND

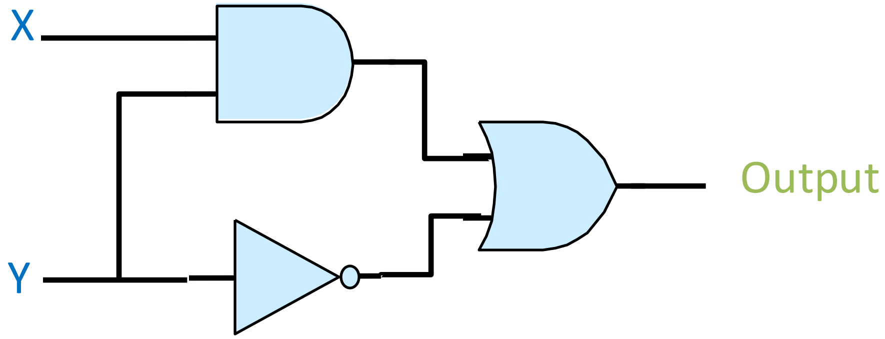


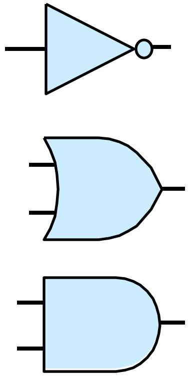
OR



NOT





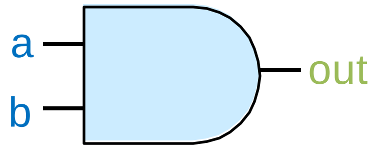


Clicker Choices

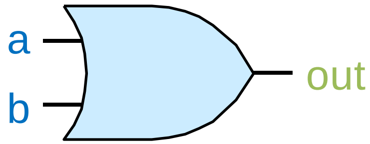
X	Y	Out_A	Out_B	Out_C	Out_D	Out_E
0	0	0	1	0	1	0
0	1	0	1	0	0	1
1	0	1	0	1	1	1
1	1	0	0	1	1	0

What does this circuit output?

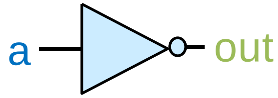
AND

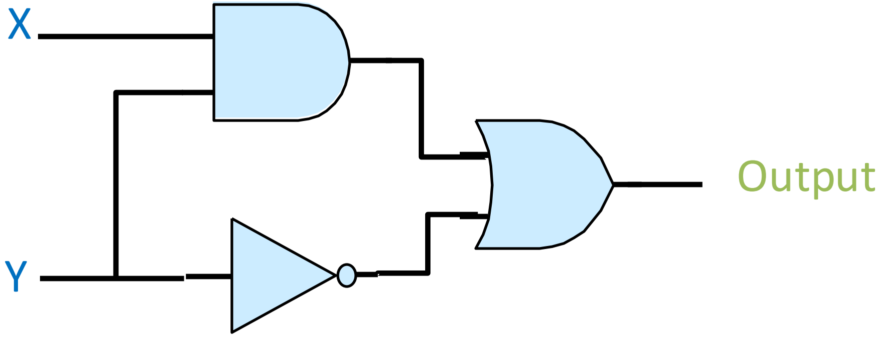


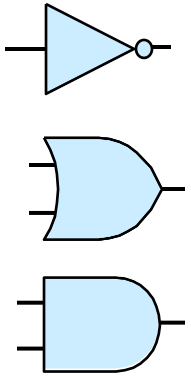
OR



NOT







Clicker Choices

X	Y	Out_A	Out_B	Out_C	Out_D	Out_E
0	0	0	1	0	1	0
0	1	0	1	0	0	1
1	0	1	0	1	1	1
1	1	0	0	1	1	0

Building more interesting circuits…

- Build-up XOR from basic gates (AND, OR, NOT)

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

- Q: When is A^B ==1?

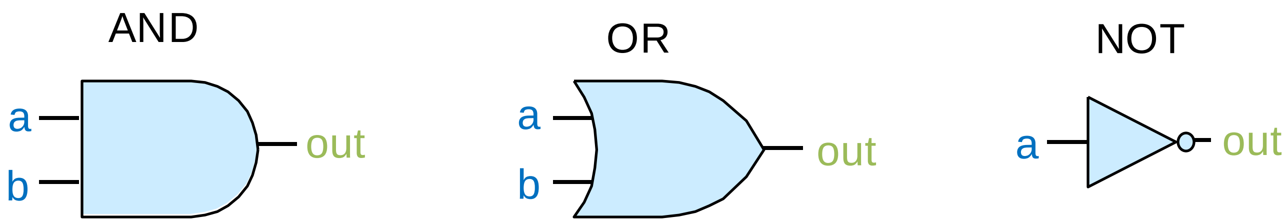
Building more interesting circuits…

- Build-up XOR from basic gates (AND, OR, NOT)

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

- Q: When is A^B ==1?

Which of these is an XOR circuit?

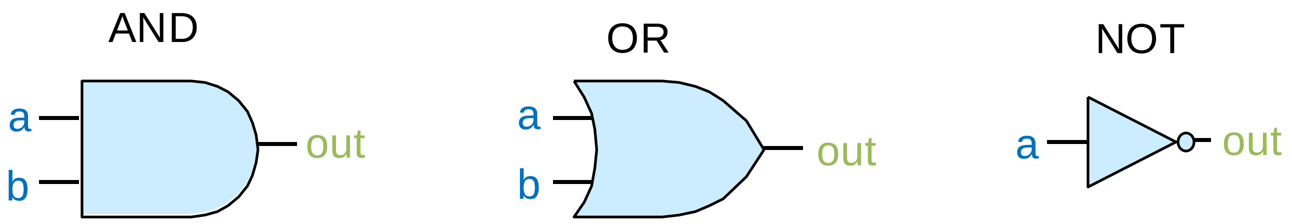


General strategy:

1. Determine truth table (given inputs)
2. Find rows with output = 1
   - express these in terms of input values A, B combined with AND, NOT
   - then, combine each row expression with OR
3. Translate expression to a circuit

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Which of these is an XOR circuit?

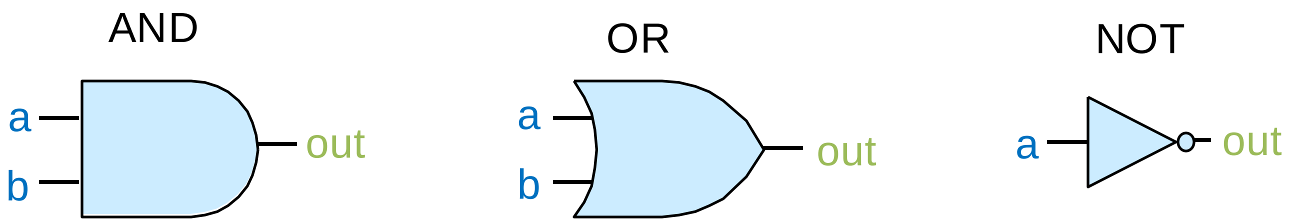


Draw an XOR circuit using AND, OR, and NOT gates.

I'll show you the clicker options after you've had some time.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Which of these is an XOR circuit?



General strategy:

1. Determine truth table (given inputs)
2. Find rows with output = 1
   - express these in terms of input values A, B combined with AND, NOT
   - then, combine each row expression with OR
3. Translate expression to a circuit

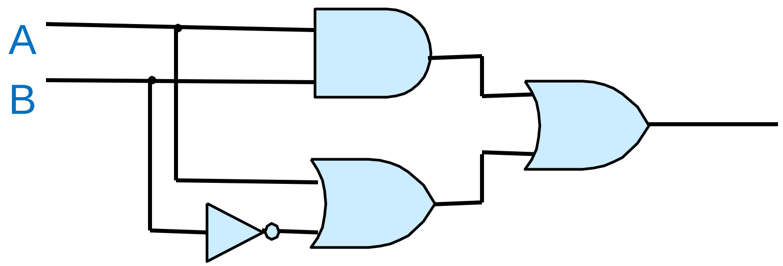
A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Use `A^B == (~A & B) | (A & ~B)`

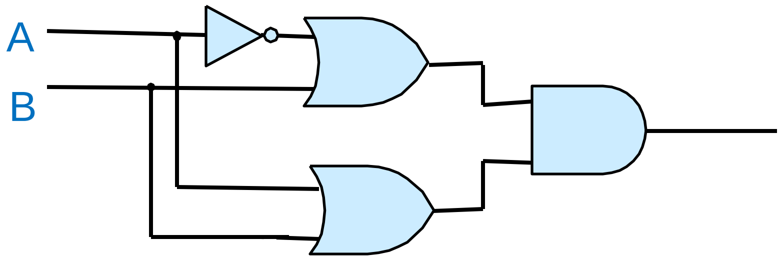
Which of these is an XOR circuit?

Use `A^B == (~A & B) | (A & ~B)`

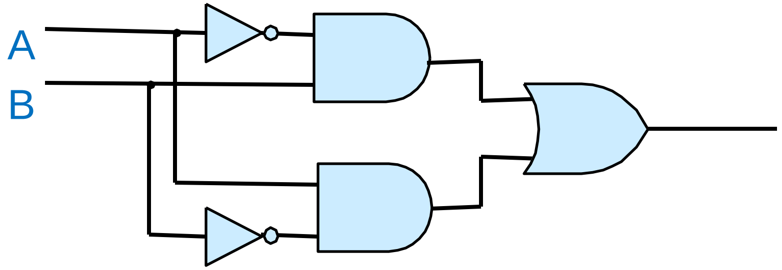
A:



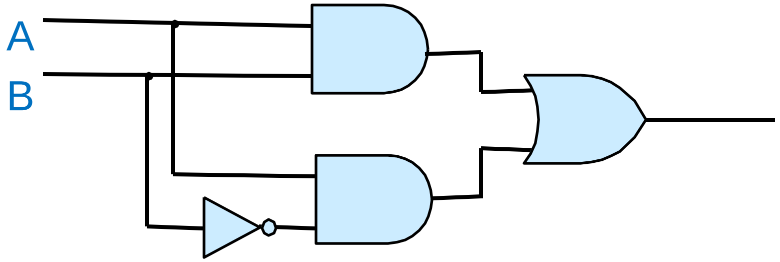
B:



C:



D:

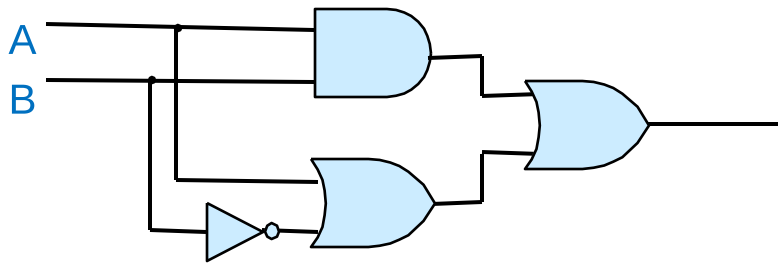


E: None of these are XOR.

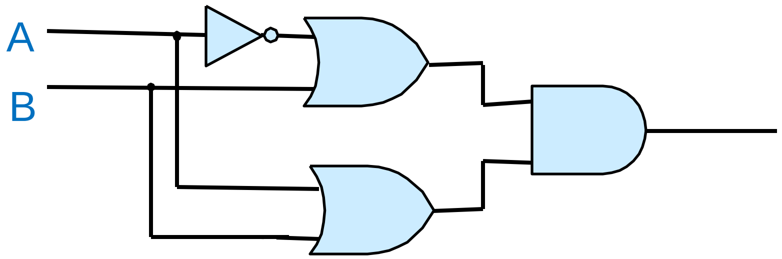
Which of these is an XOR circuit?

Use `A^B == (~A & B) | (A & ~B)`

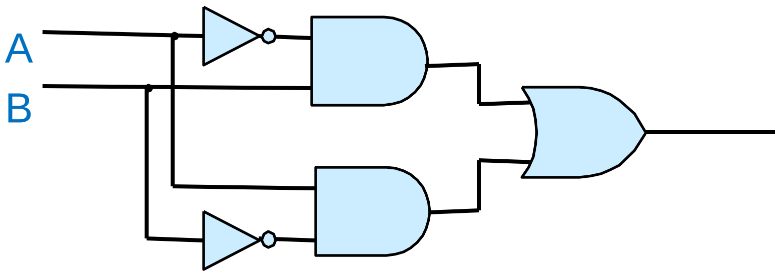
A:



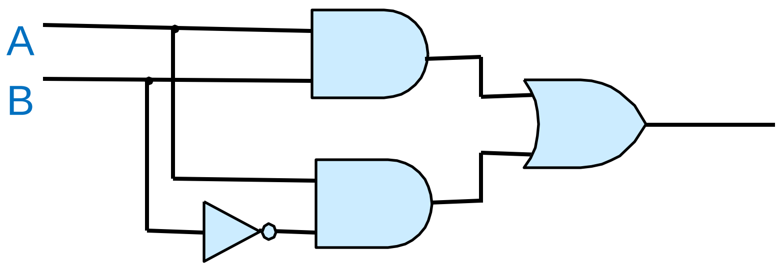
B:



C:



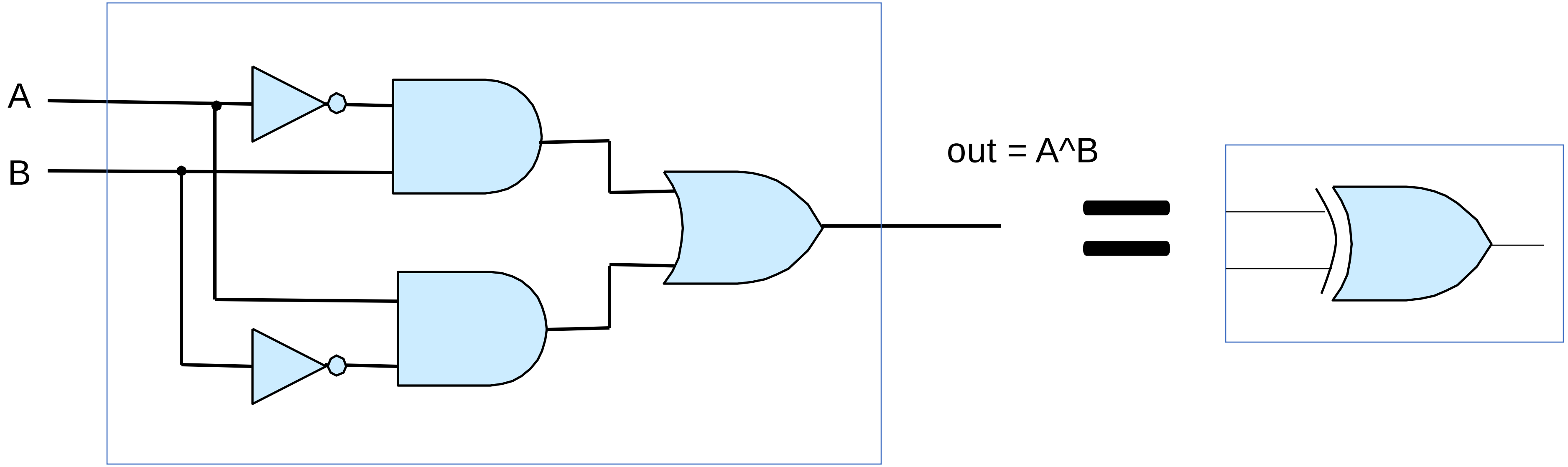
D:



E: None of these are XOR.

XOR Circuit: Abstraction

```
A^B  == (~A & B) | (A & ~B)
```

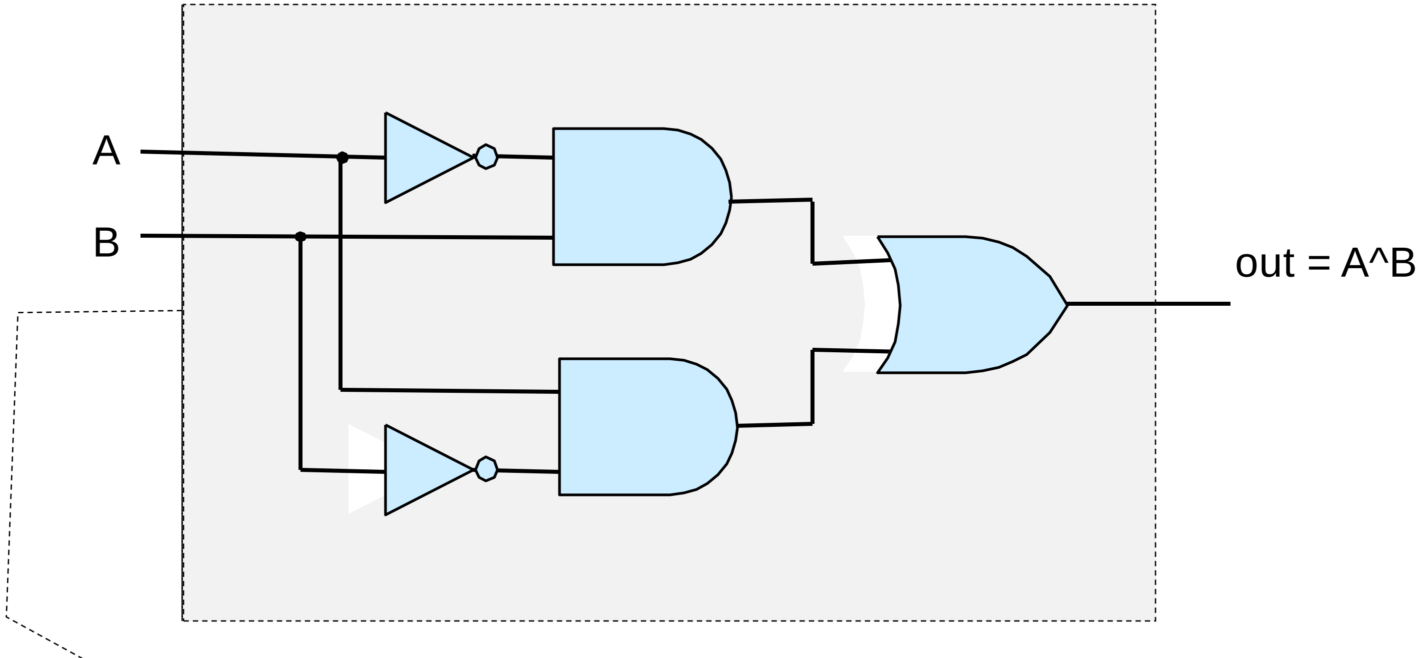


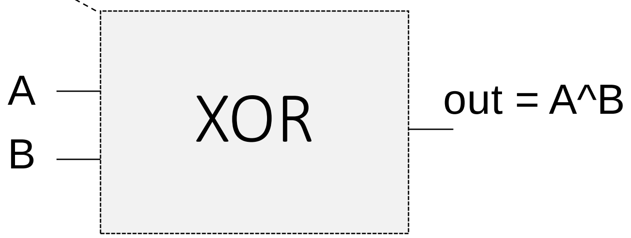

```
A:0 B:0 A^B:
A:0 B:1 A^B:
```

```
A:1 B:0 A^B:
A:1 B:1 A^B:
```

XOR Circuit: Abstraction!

```
A^B  == (~A & B) | (A & ~B)
```





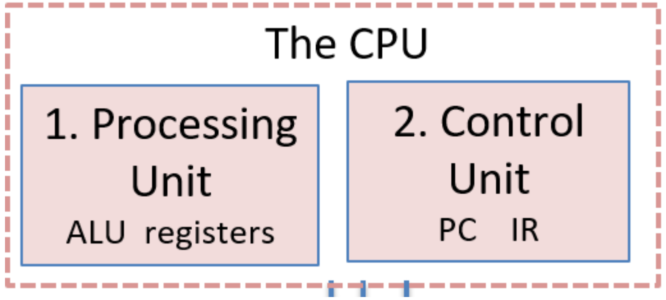
Treat XOR Circuit as a building block for other circuits!

Abstraction!

- Hide away the complex internals of ***how*** the system functions, and focus on ***what*** functionality we expect. I.e., the guaranteed output of a system given the set of allowed inputs, and treating the functionality of the system as a black box.

- What are examples of abstractions you have experienced in daily life?

Recall Goal: Build a CPU (model)



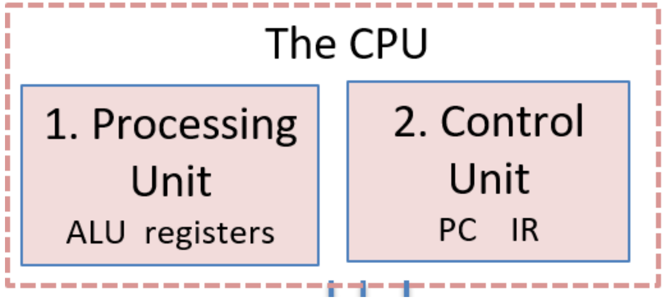
Three main classifications of hardware circuits:

1. ALU: implement arithmetic & logic functionality
   – Example: adder circuit to add two values together

2. Storage: to store binary values
   – Example: set of CPU registers ("register file") to store temporary values

3. Control: support/coordinate instruction execution
   – Example: circuitry to fetch the next instruction from memory and decode it

Recall Goal: Build a CPU (model)



<u>Three main classifications of hardware circuits:</u>

1. ALU: implement arithmetic & logic functionality
   – Example: adder circuit to add two values together

Start with ALU components (e.g., adder circuit, bitwise operator circuits)

Combine component circuits into ALU!

Digital Circuits - Building a CPU

Start with building an ALU:

1. Individual components from basic logic gates

   Adder, Subtractor, Bit shifter, Bit-wise OR, ...

2. Combine them together into ALU!

ALU
Full Adder, Shifter, …
1-bit circuits
Basic Logic Gates

Arithmetic Circuits

- 1 bit adder: A+B
- Two outputs:
  1. Obvious one: the sum
  2. Other one: ??

A	B	Sum (A + B)	C_{out}
0	0		
0	1		
1	0		
1	1		

Arithmetic Circuits

- 1 bit adder: A+B
- Two outputs:
  1. Obvious one: the sum
  2. Other one: ??

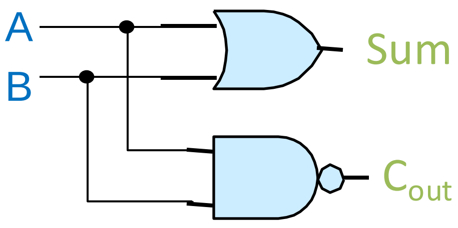
A	B	Sum (A + B)	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Which of these circuits is a one-bit adder?

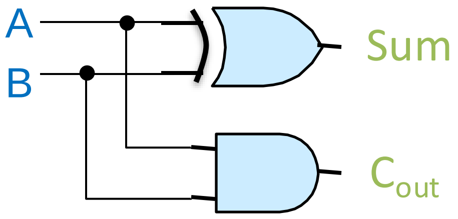
A	B	Sum (A + B)	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

A	B	Sum (A + B)	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

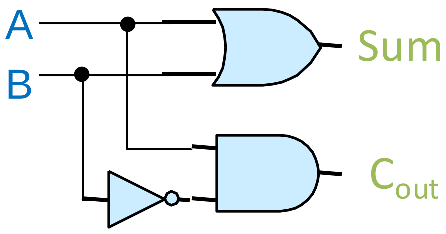
A:



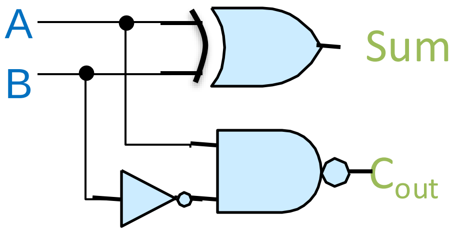
B:



C:



D:

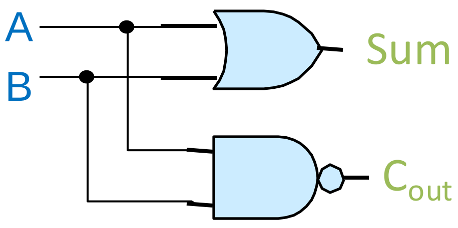


Which of these circuits is a one-bit adder?

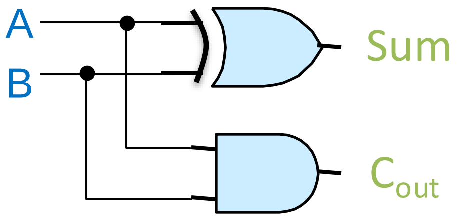
A	B	Sum (A + B)	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

A	B	Sum (A + B)	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

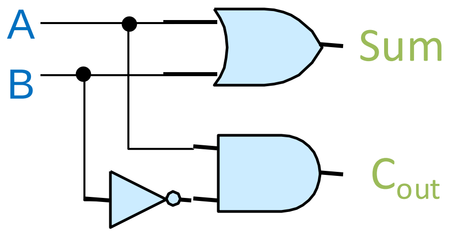
A:



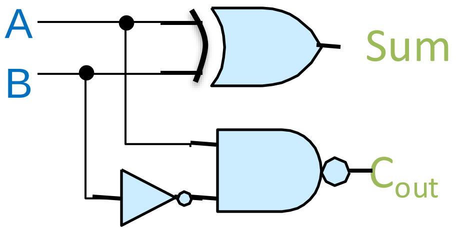
B:



C:



D:



More than one bit addition?

- When adding, sometimes have *carry in* too

```
    0011010
+   0001111
    -------
```

More than one bit?

- When adding, sometimes have *carry in* too

```
     1111
    0011010
+   0001111
```

Write Boolean expressions for $Sum = 1$ and $C_{out} = 1$

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

When is Sum 1?

When is C_{out} 1?

Write Boolean expressions for

Sum = 1 and C_{out} = 1

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

When is Sum 1?

$\sim C_{in}$ & (A^B) | C_{in} & ~(A^B) == (C_{in} ^ (A^B))

When is C_{out} 1?

Write Boolean expressions for

Sum = 1 and C_{out} = 1

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

When is Sum 1?

$\sim C_{in}$ & (A^B) | C_{in} & ~(A^B) == (C_{in} ^ (A^B))

When is C_{out} 1?

(A & B) | ((A^B) & C_{in})

Write Boolean expressions for $Sum = 1$ and $C_{out} = 1$

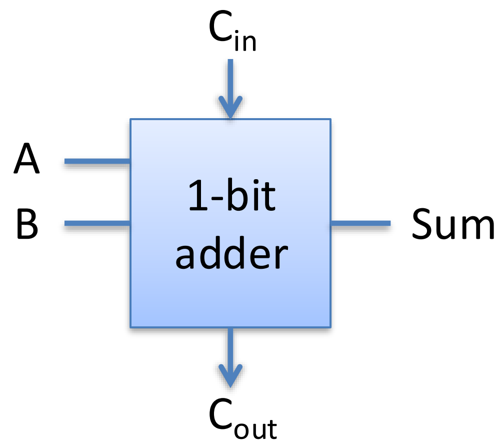
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

When is Sum 1?

~C_{in} & (A^B) | C_{in} & ~(A^B) == (C_{in} ^ (A^B))

When is C_{out} 1?

(A & B) | ((A^B) & C_{in})

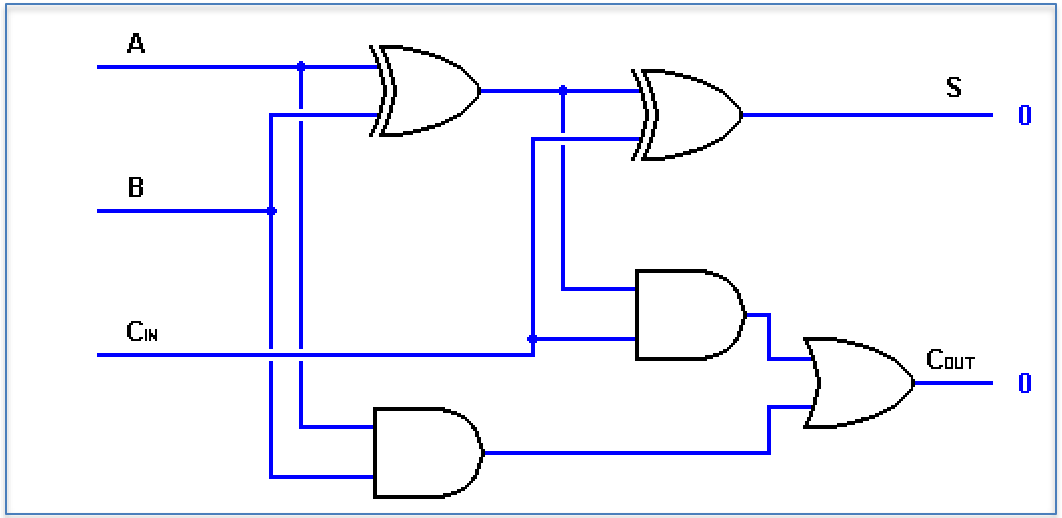


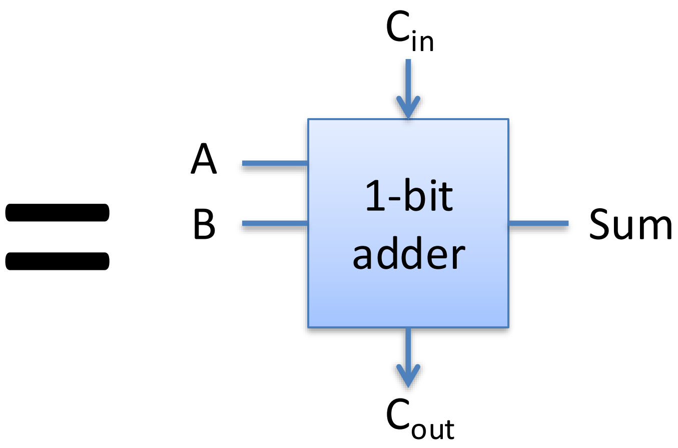
One-bit (full) adder

- Need to include:

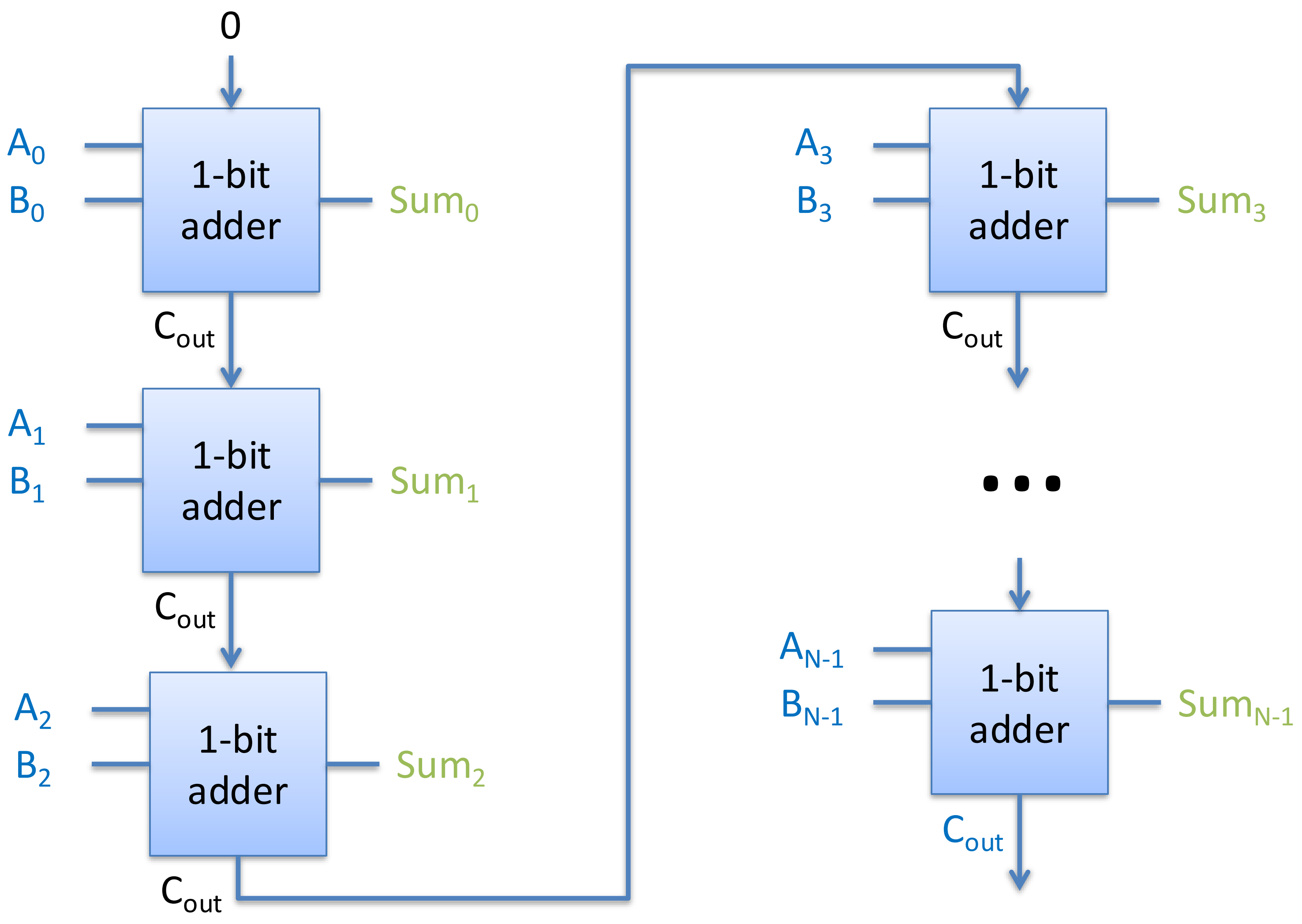
  carry-in and carry-out

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



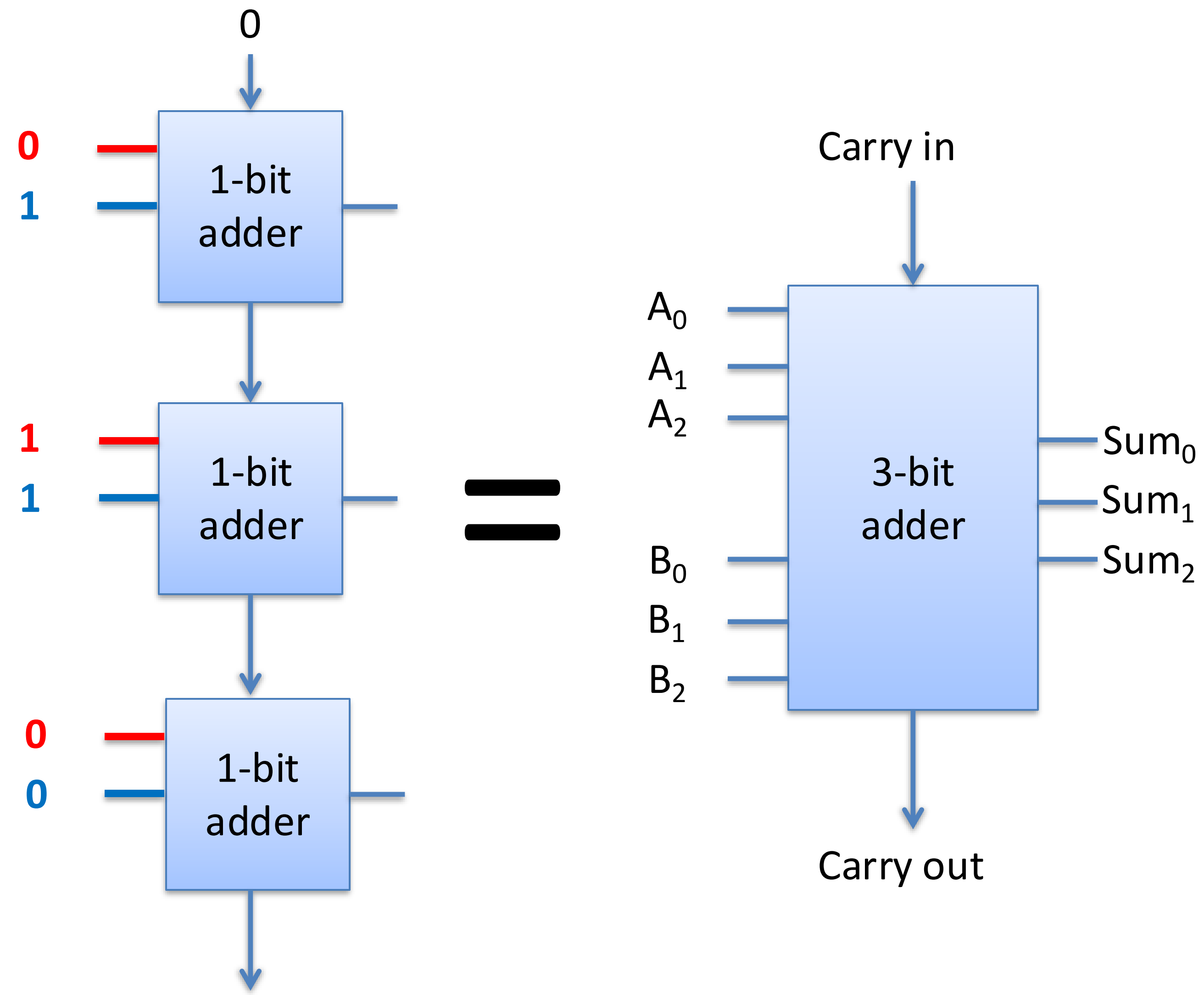


Multi-bit Adder (Ripple-carry Adder)



Three-bit Adder (Ripple-carry Adder)

010
+ 011



Arithmetic Logic Unit (ALU)

- One component that knows how to manipulate bits in multiple ways
  - Addition
  - Subtraction
  - Multiplication / Division
  - Bitwise AND, OR, NOT, etc.

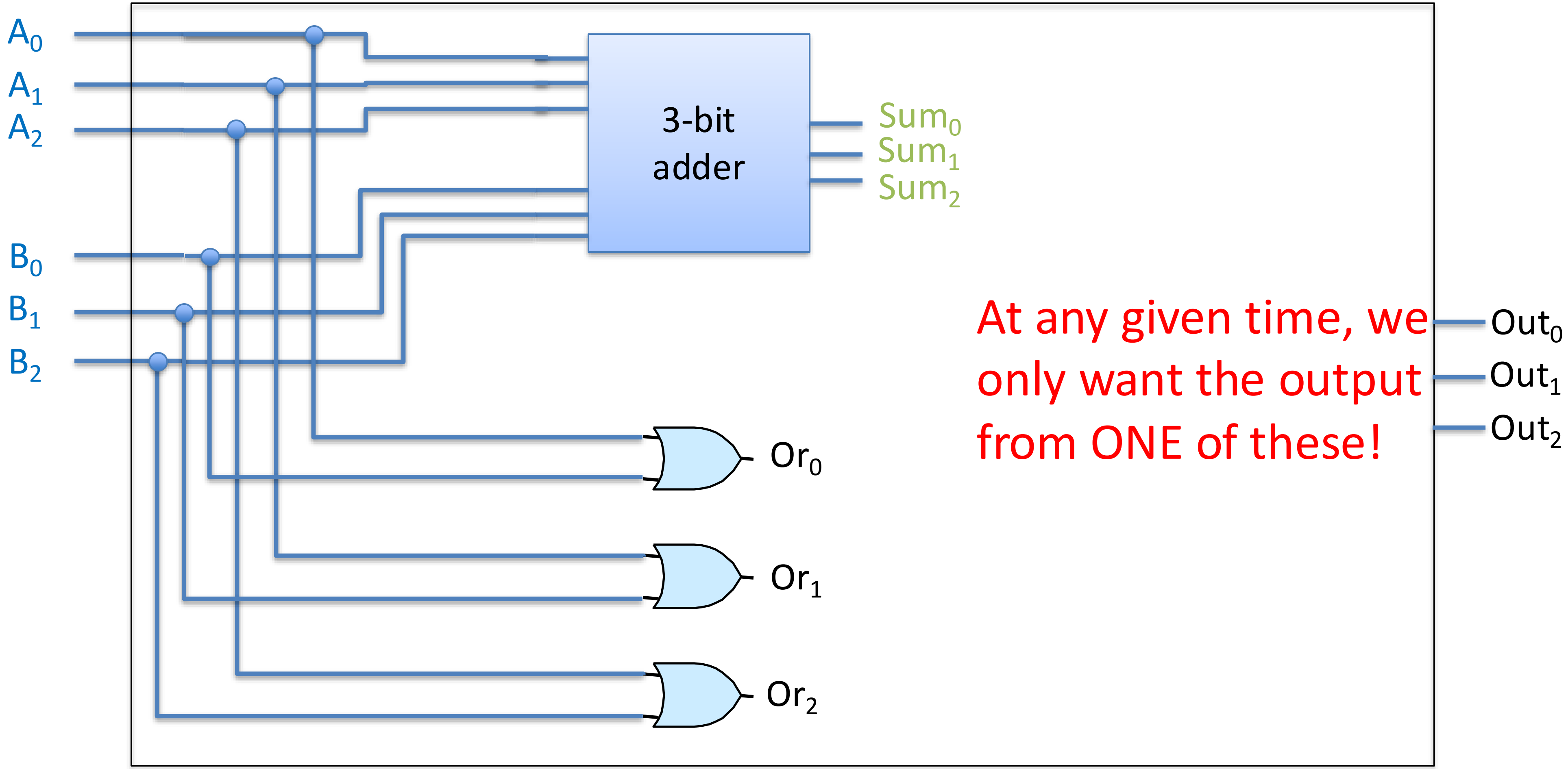
- Built by combining components
  - Take advantage of sharing HW when possible (e.g., subtraction using adder)

Simple 3-bit ALU: Add and bitwise OR

3-bit inputs
A and B:

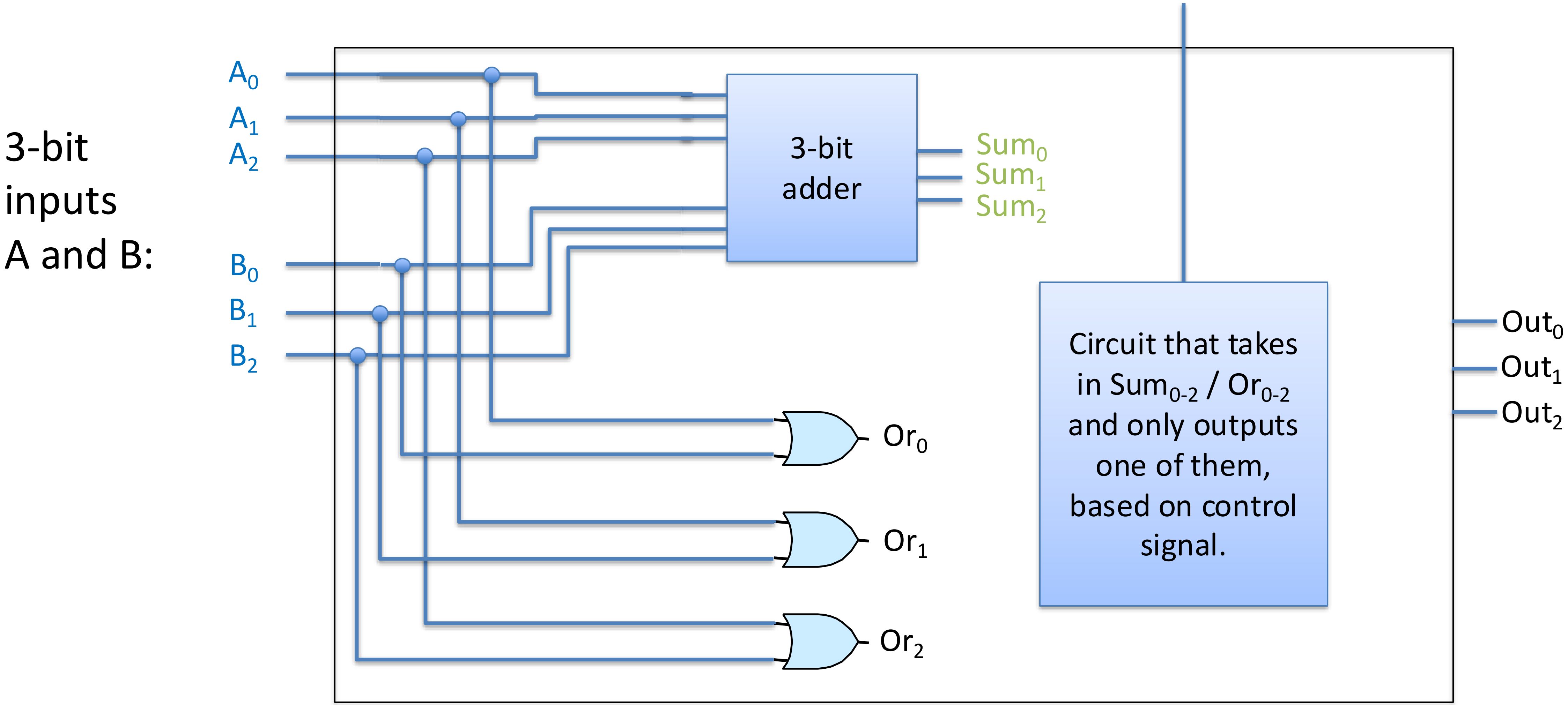
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs A and B:



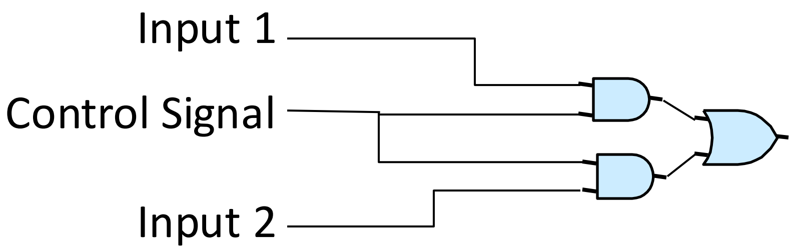
Simple 3-bit ALU: Add and bitwise OR

Extra input: control signal to select Sum vs. OR

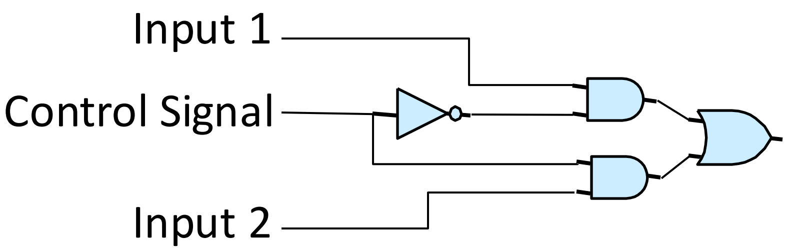


Which of these circuits lets us select between two inputs?

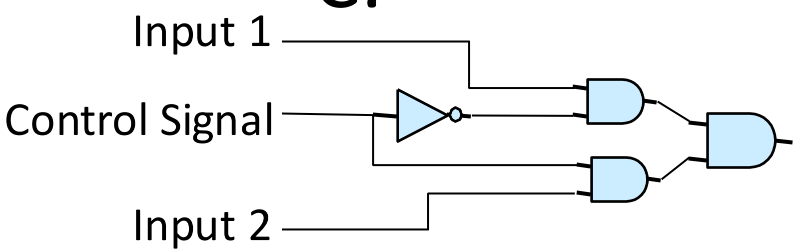
A:



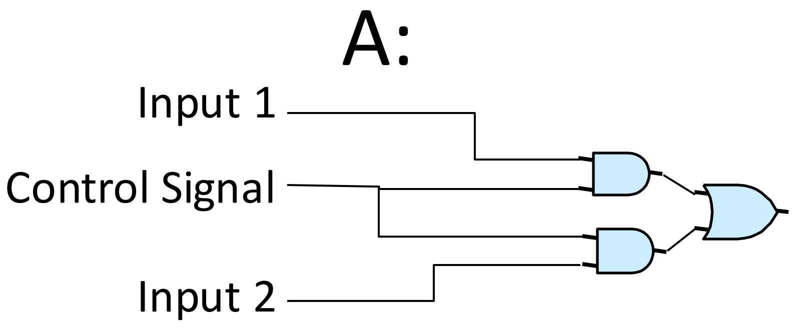
B:

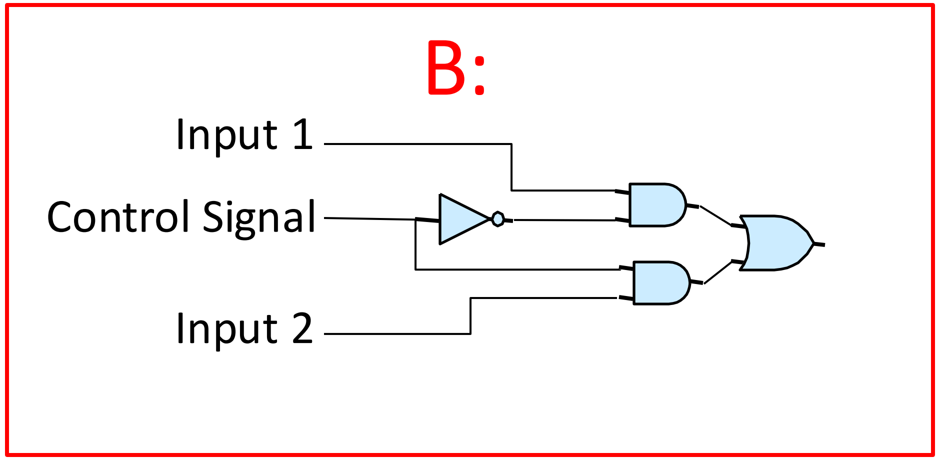


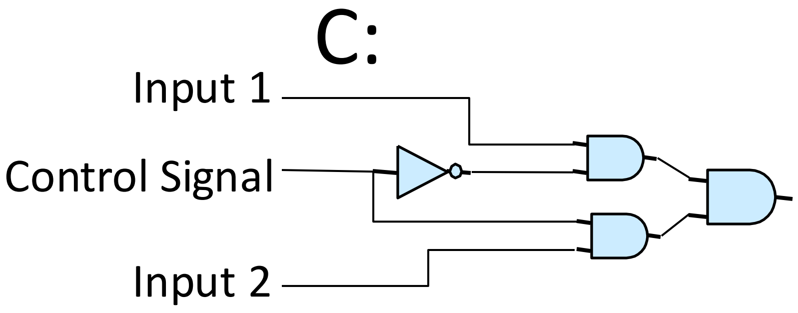
C:



Which of these circuits lets us select between two inputs?



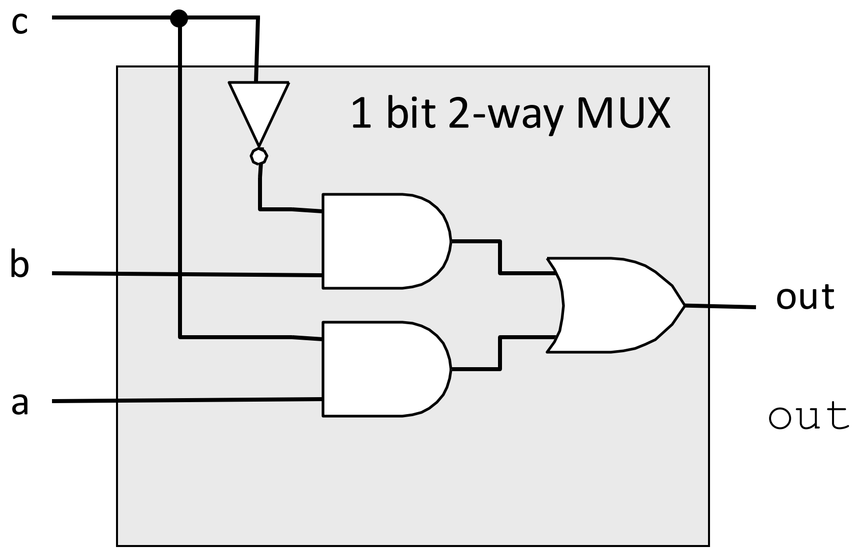




Multiplexor: Chooses an input value

Inputs: 2^N data inputs, N signal bits

Output: is one of the 2^N input values

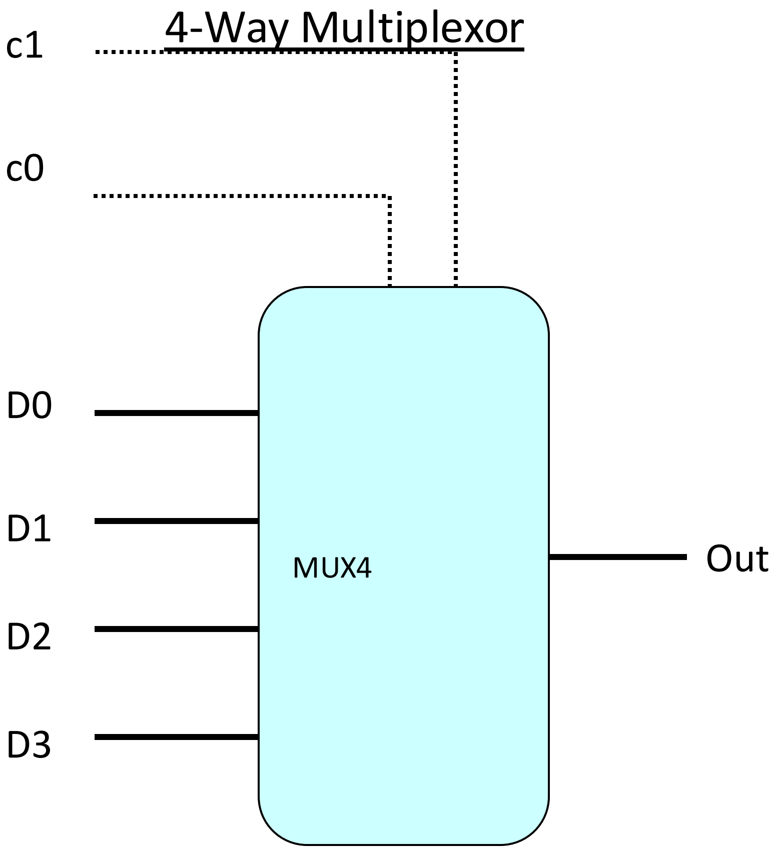


`out = (c & a)|(~c &b)`

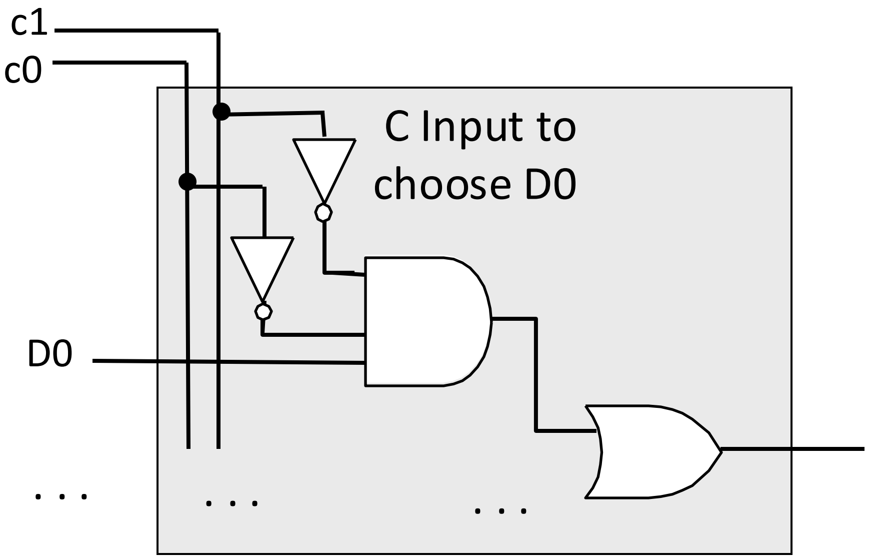
- Control signal c, chooses the input for output
  - When c is 1: choose a, when c is 0: choose b

N-Way Multiplexor

Choose one of N inputs, need $\log_2 N$ select bits

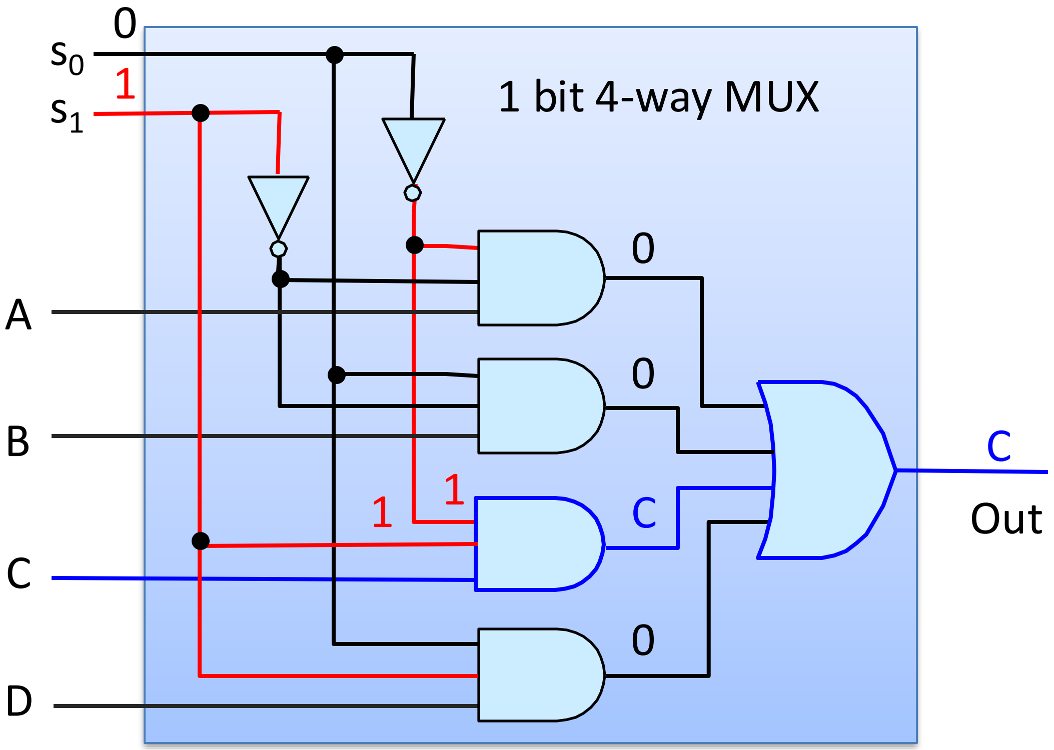


c_1	c_2	Output
0	0	D0
0	1	D1
1	0	D2
1	1	D3

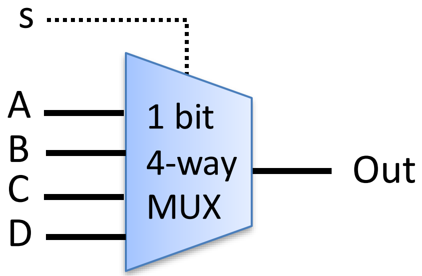


Example 1-bit, 4-way MUX

- When select input is 2 (0b10): C chosen as output

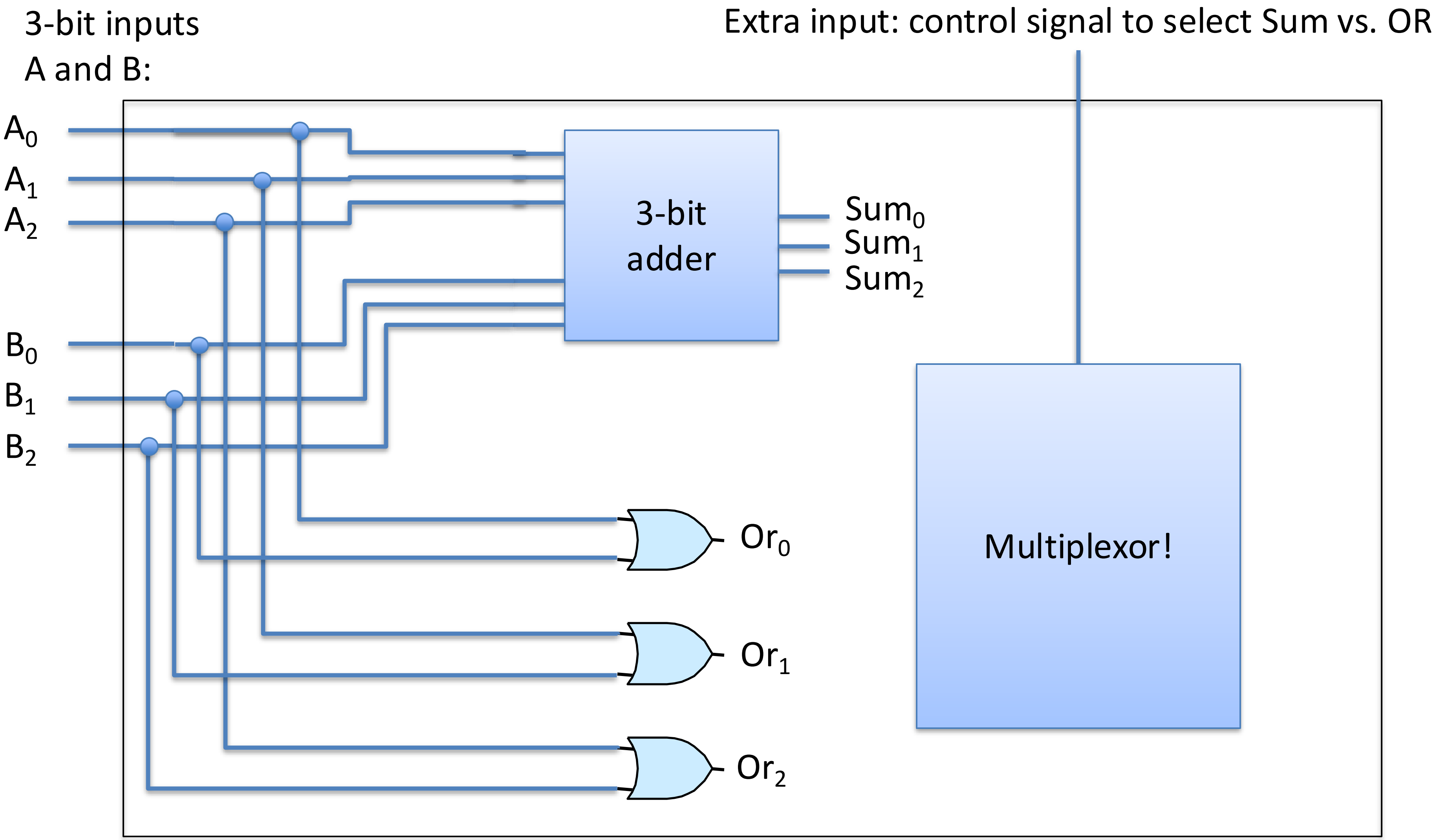


=

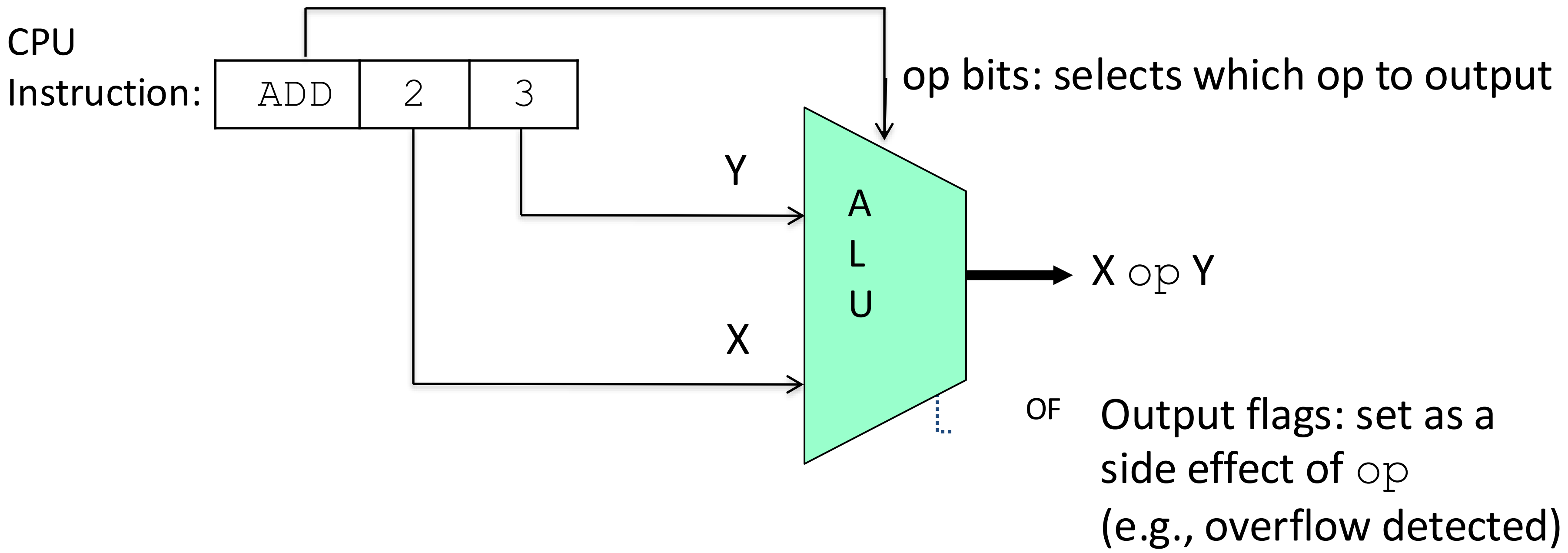


S	Out
0	A
1	B
2	C
3	D

Simple 3-bit ALU: Add and bitwise OR

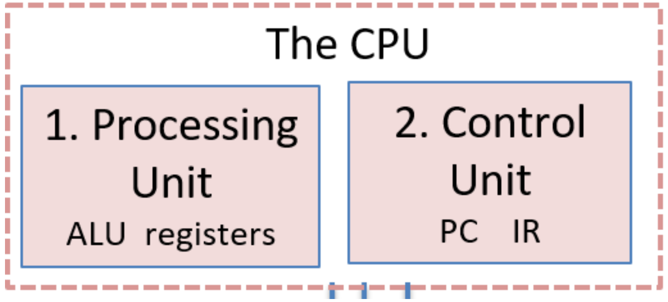


ALU: Arithmetic Logic Unit



- Arithmetic and logic circuits: ADD, SUB, NOT, ...
- Control circuits: use op bits to select output
- Circuits around ALU:
  - Select input values X and Y from instruction or register
  - Select op bits from instruction to feed into ALU
  - Feed output somewhere

Goal: Build a CPU (model)



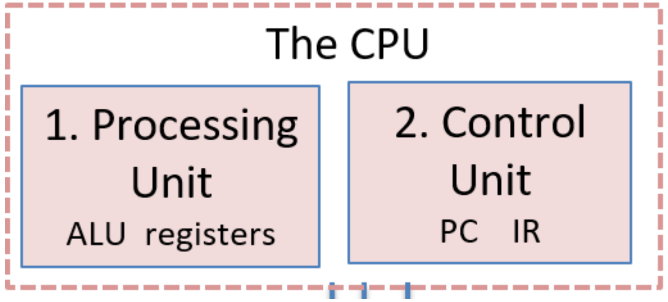
<u>Three main classifications of hardware circuits:</u>

1. ALU: implement arithmetic & logic functionality
   – Example: adder circuit to add two values together

2. Storage: to store binary values
   – Example: set of CPU registers ("register file") to store temporary values

3. Control: support/coordinate instruction execution
   – Example: circuitry to fetch the next instruction from memory and decode it

Goal: Build a CPU (model)



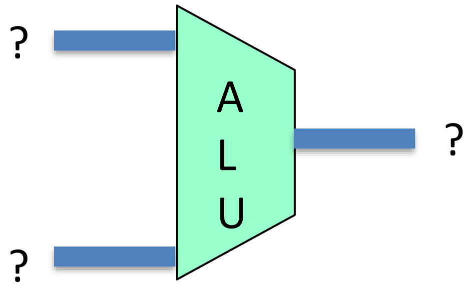
<u>Three main classifications of hardware circuits:</u>

2. Storage: to store binary values
   – Example: set of CPU registers (“register file”) to store temporary values

Give the CPU a “scratch space” to perform calculations and keep track of the state its in.

CPU so far…

- We can perform arithmetic!
- Storage questions:
  – Where to the ALU input values come from?
  – Where do we store the result?
  – What does this “register” thing mean?



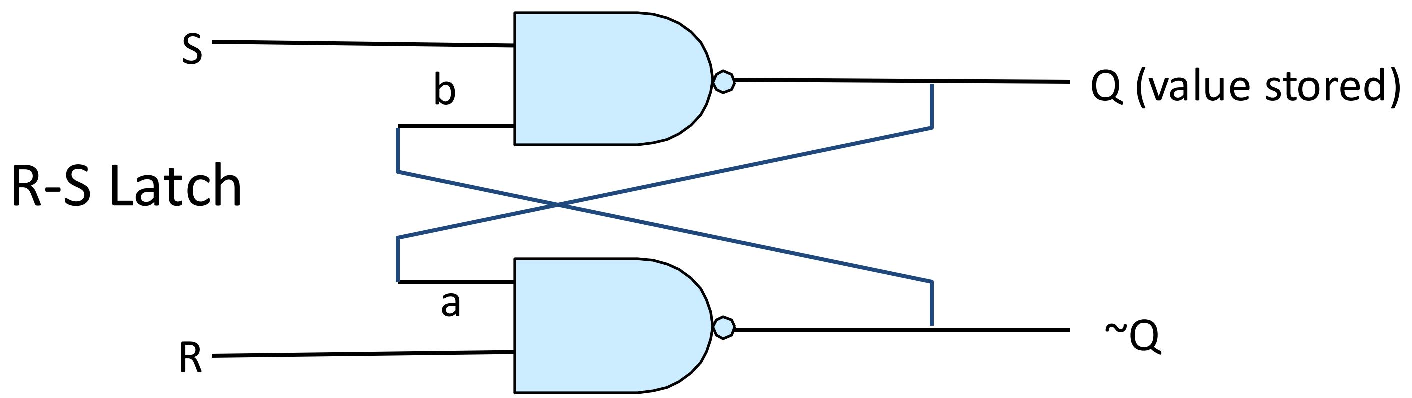
Memory Circuit Goals: Starting Small

- Store a 0 or 1
- Retrieve the 0 or 1 value on demand (read)
- Set the 0 or 1 value on demand (write)

R-S Latch: Stores Value Q

When R and S are both 1: Maintain a value

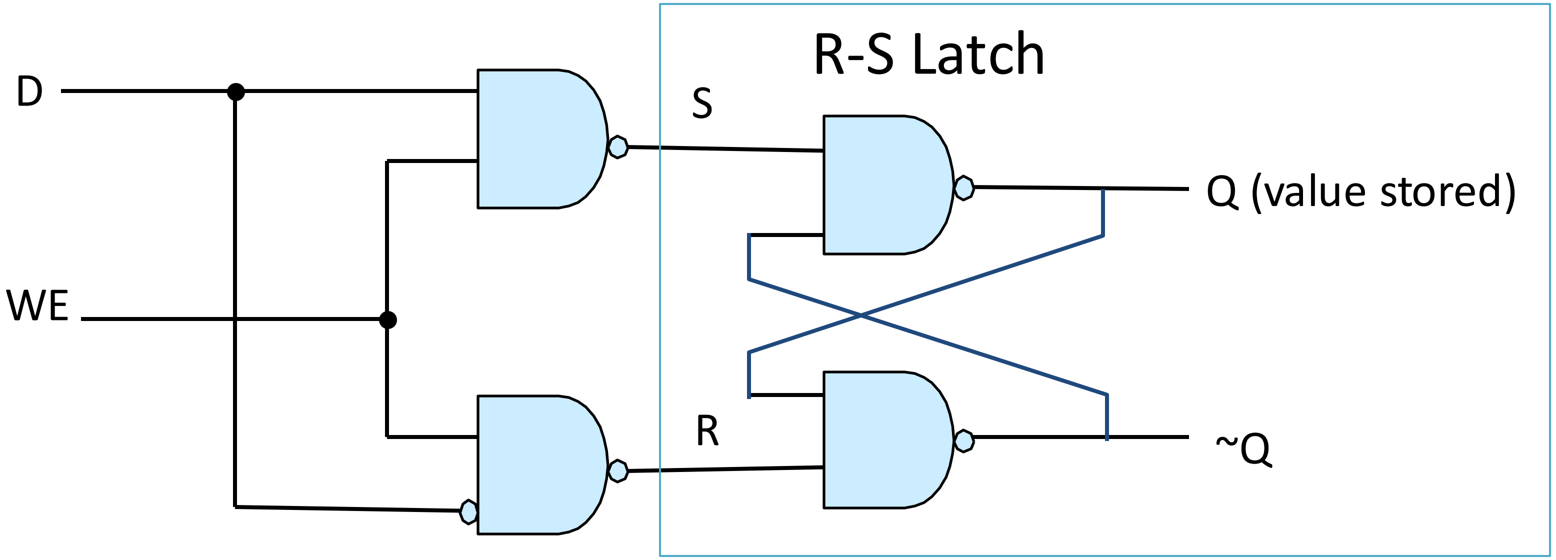
R and S are never both simultaneously 0



- To write a new value:
  - Set S to 0 momentarily (R stays at 1): to write a 1
  - Set R to 0 momentarily (S stays at 1): to write a 0

Gated D Latch

Controls S-R latch writing, ensures S & R never both 0

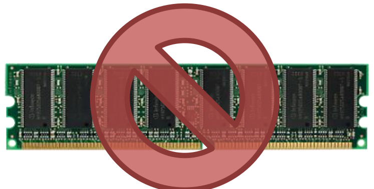


D: into top NAND, ~D into bottom NAND

WE: write-enabled, when set, latch is set to value of D

Latches used in registers (up next) and SRAM (caches, later)

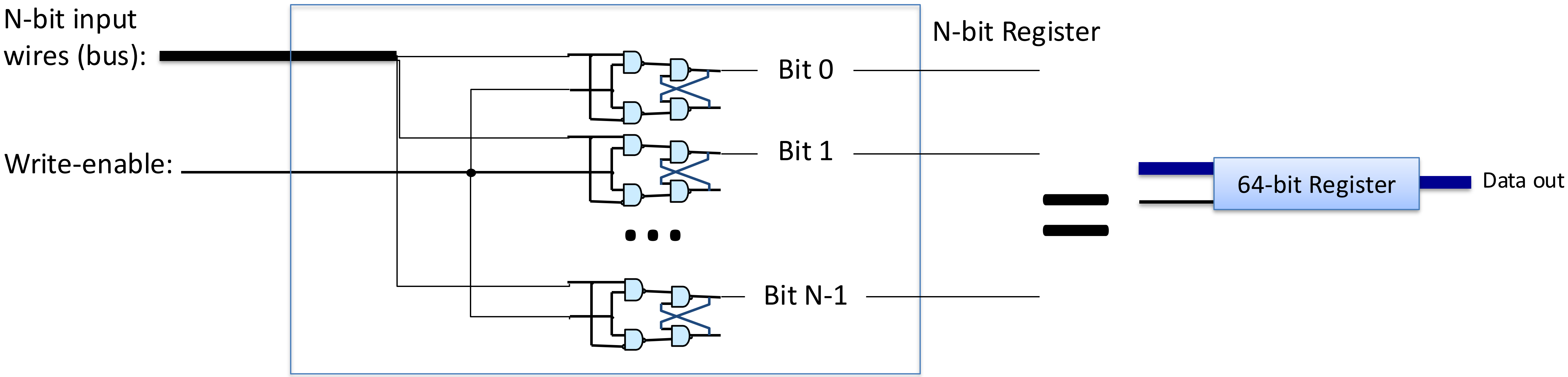
Fast, not very dense, expensive

DRAM: capacitor-based: 

An N-bit Register

- Fixed-size storage (8-bit, 32-bit, 64-bit, etc.)

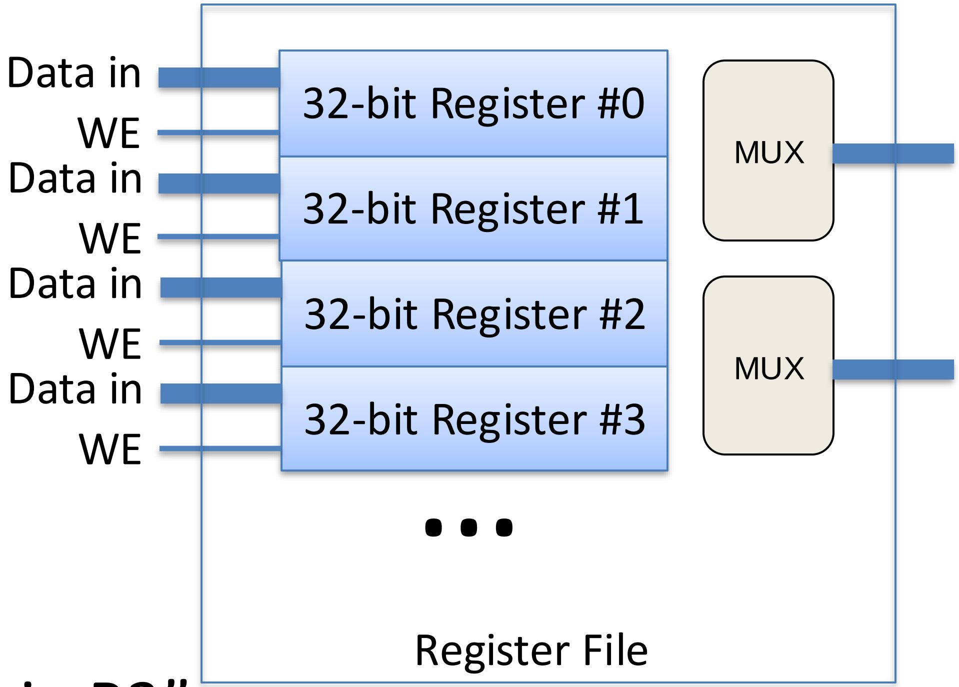
- Gated D latch lets us store one bit
  - Connect N of them to the same write-enable wire!



“Register file”

- A set of registers for the CPU to store temporary values.

- This is (finally) something you will interact with!



- Instructions of form:
  - “add R1 + R2, store result in R3”

Memory Circuit Summary

- Lots of abstraction going on here!
  - Gates hide the details of transistors.
  - Build R-S Latches out of gates to store one bit.
  - Combining multiple latches gives us N-bit register.
  - Grouping N-bit registers gives us register file.

- Register file's simple interface:
  - Read R_x's value, use for calculation
  - Write R_y's value to store result