Computers:
A Look Behind The Curtain

Sam Wolfson

CSE 120, Winter 2020

Administrivia

- Assignments
  - Controlling Elli due tonight
  - Portfolio Update 2 due next Wednesday (Feb 26)
  - Tic-Tac-Toe (last programming assignment until your final project!) due next Thursday (Feb 27)
- Looking ahead...
  - Final project design document due next Friday (Feb 28)
  - Living Computers Museum report due Mar 2
- Guest lecture next Monday: HCI

Quiz Recap

3. (6 points) Loops & Arrays

   I've written a (partially-complete) function `prod` that calculates and returns the *product* of all the elements in the array `arr`. Complete this function by filling in the blanks.

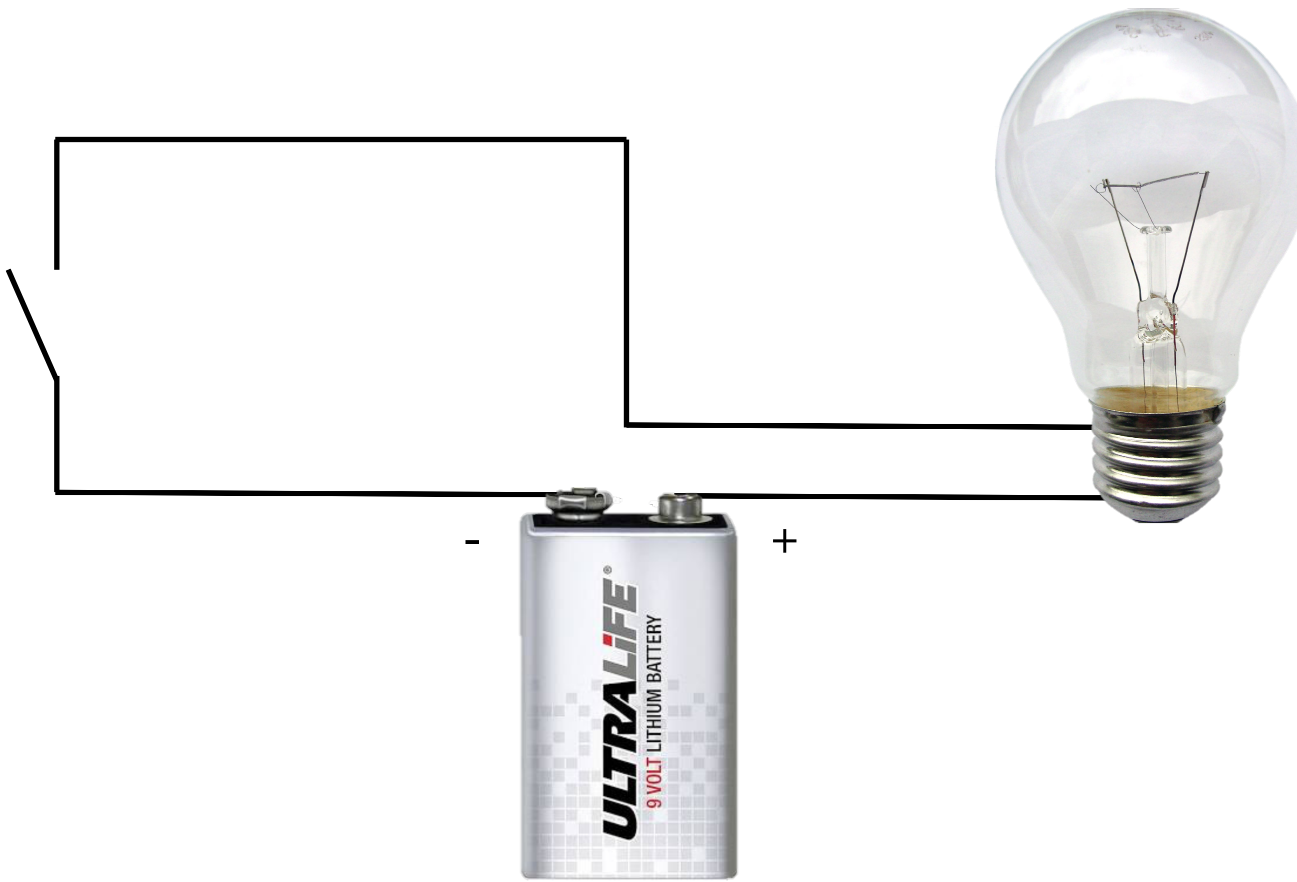
```
______________ prod(int[] arr) {
     int index = ______________;
     int product = ______________;

     while ( index < ______________ ) {
          product = ____________________________;
          index = ______________;
     }

     return ______________;
}
```

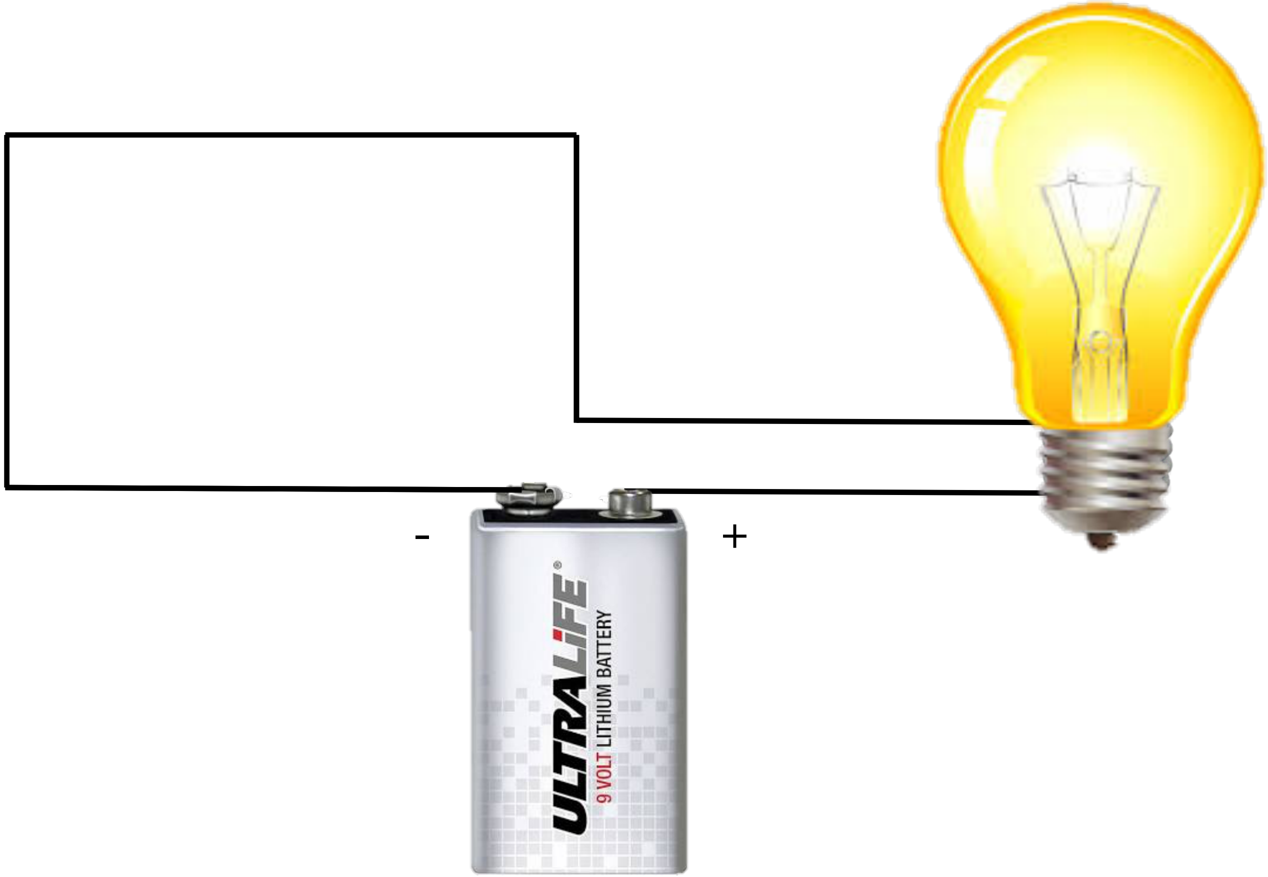
A Light Switch

The switch interrupts the circuit when it is off



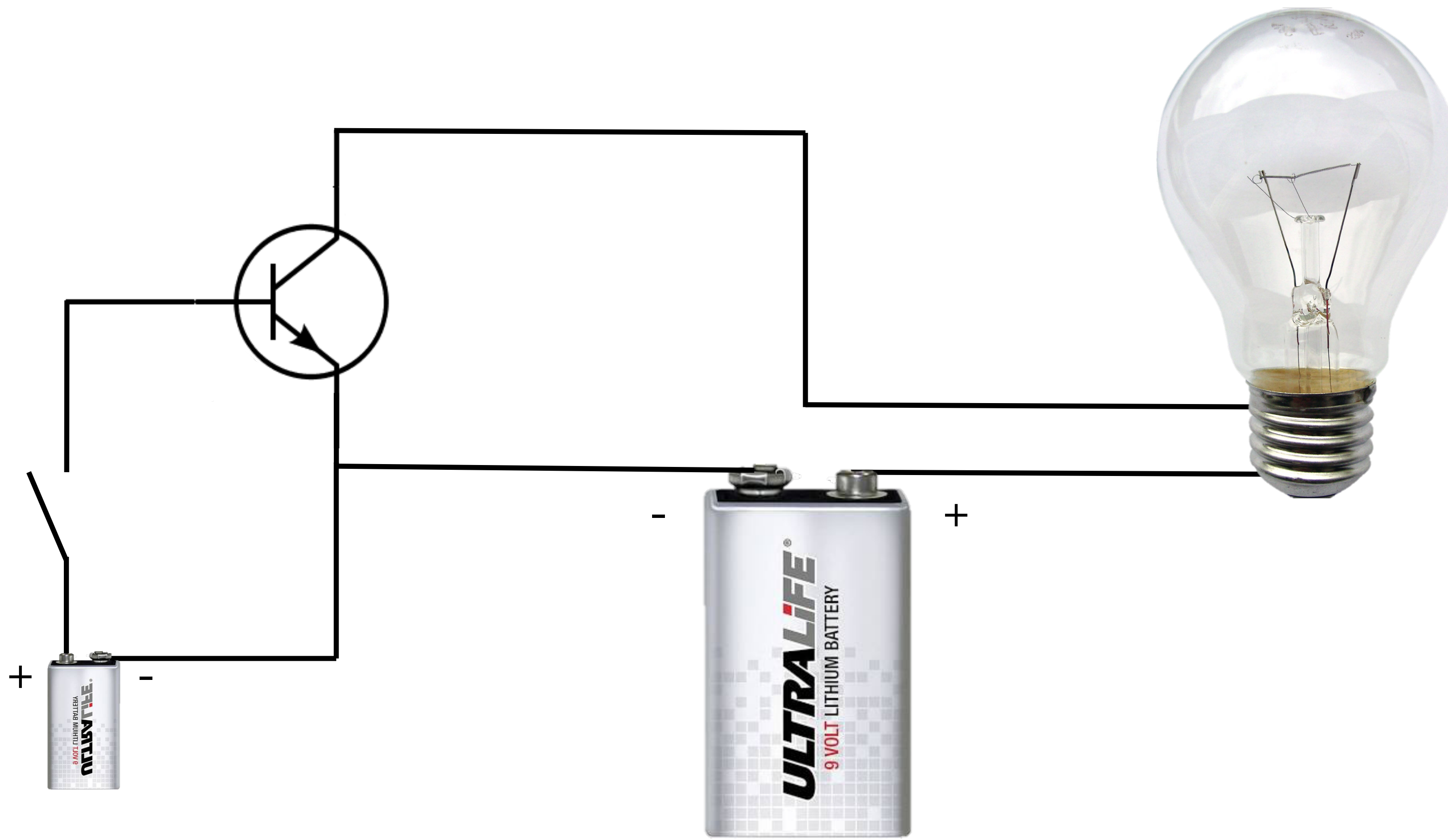
A Light Switch

...and completes the circuit when it is on



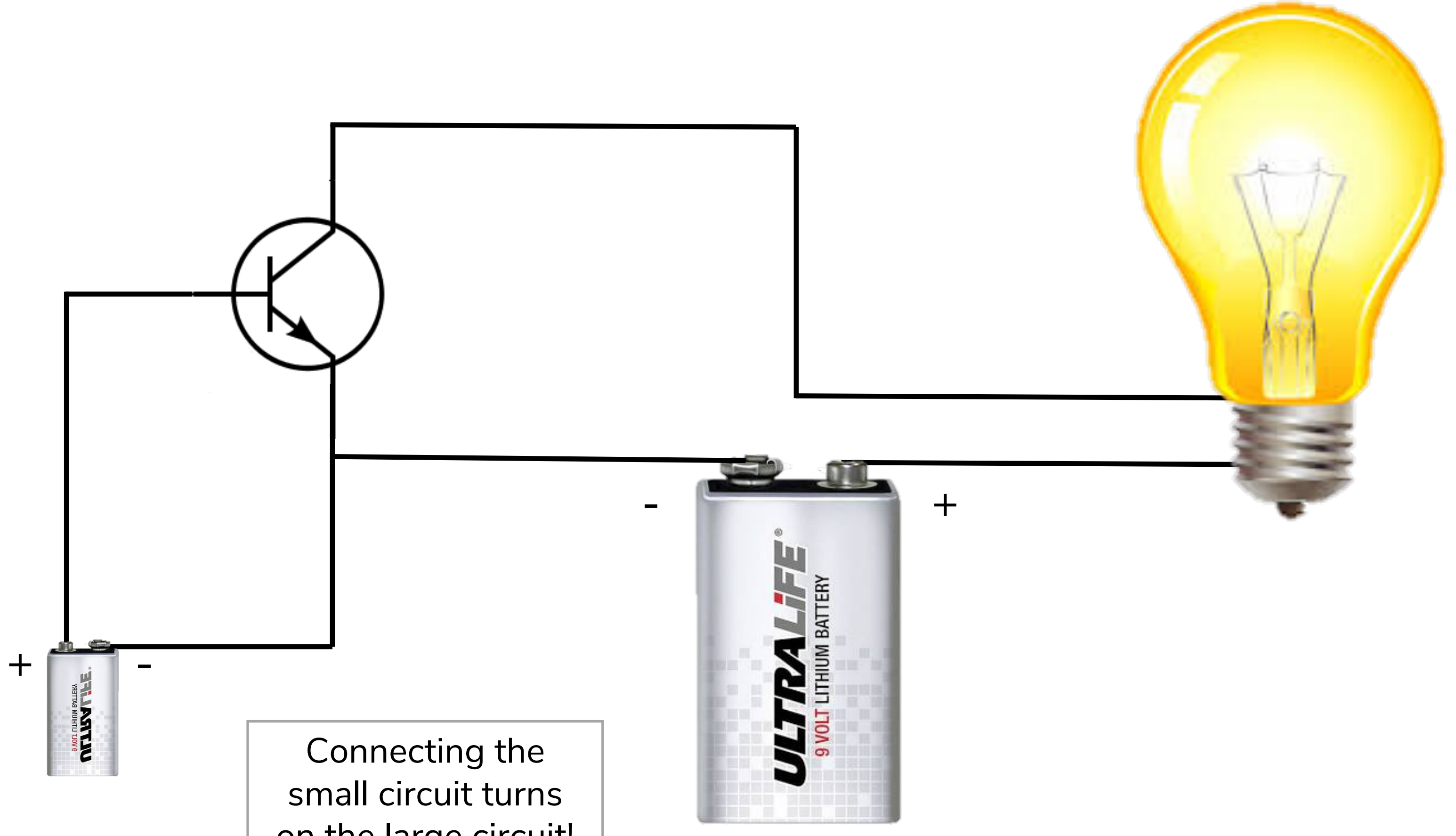
A Transistor

...is just like a switch (but controlled by electricity)!



A Transistor

...is just like a switch (but controlled by electricity)!



Connecting the small circuit turns on the large circuit!

Transistors

- **Idea**: use a small amount of electricity to control a (possibly larger) amount of electricity
    - Example: amplifiers

- In computers: use circuits to control other circuits!

Building Logic With Transistors

- In Processing: can compare `boolean` values

```
A && B          A || B          !A
```

- In hardware:
  - `false` means the circuit is off
  - `true` means the circuit is on

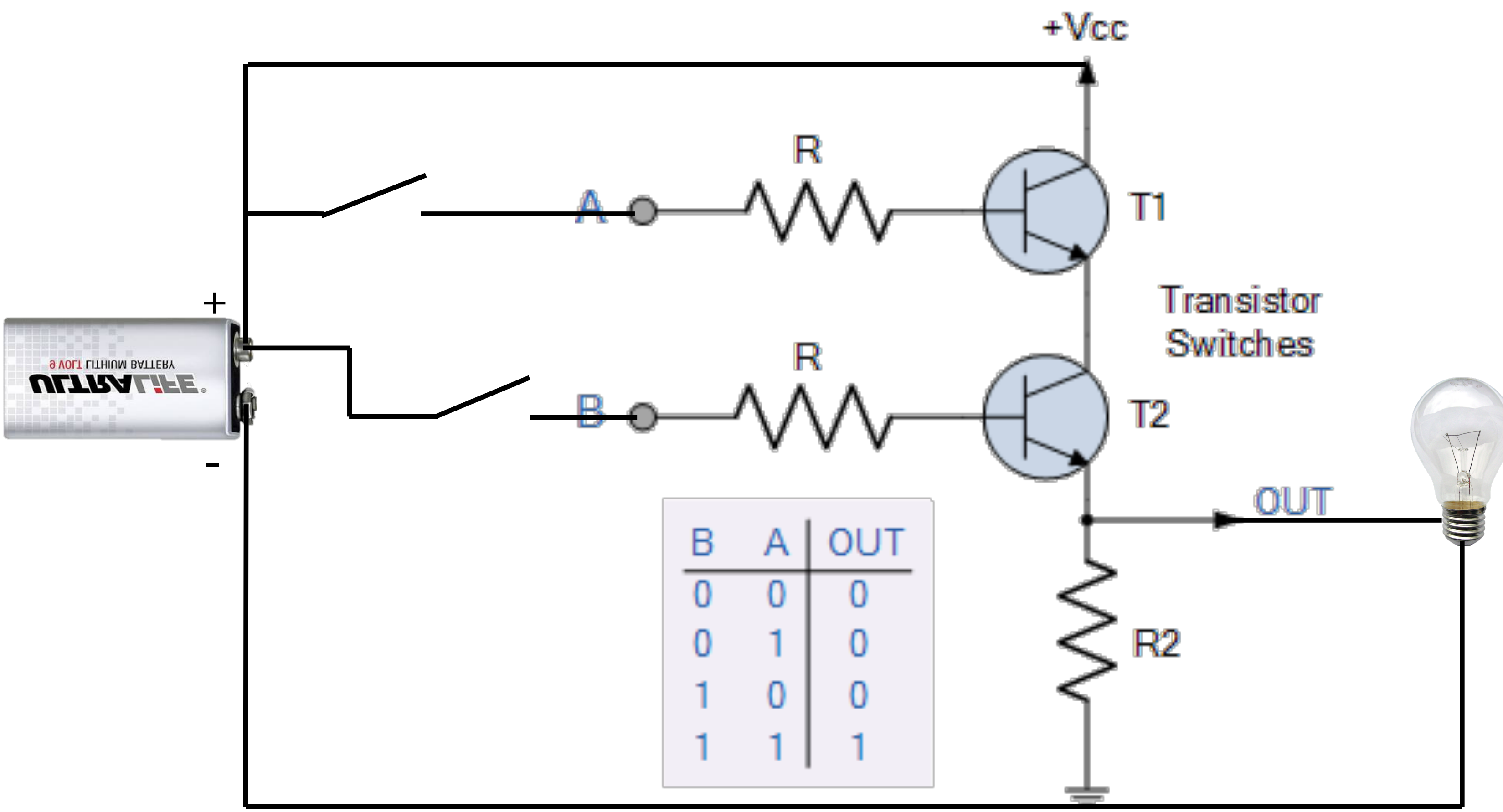
- How to implement comparison with transistors?

AND gate

false: circuit off
true: circuit on

Goal: `OUT = A && B`

voltage



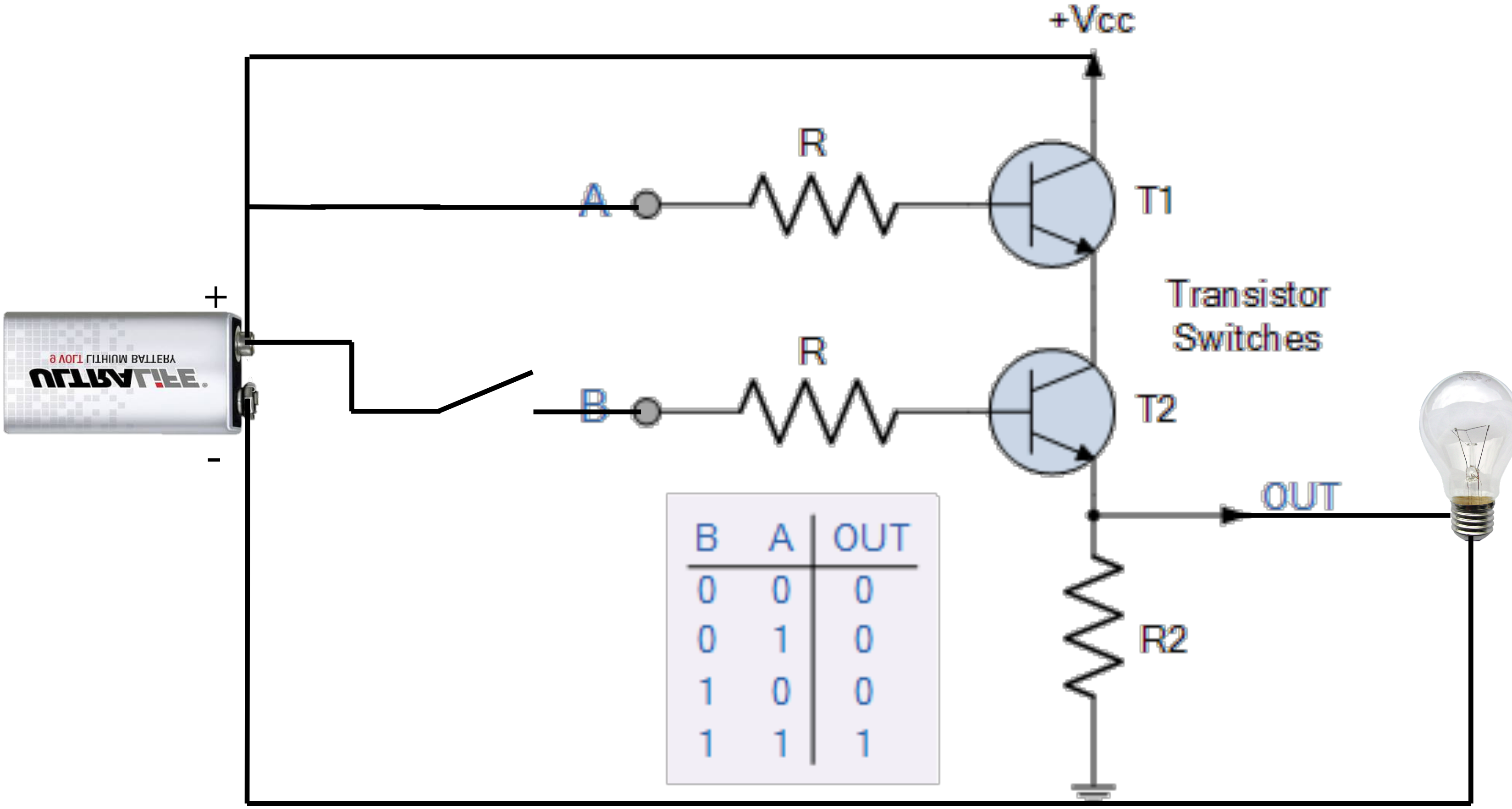
B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1

ground

AND gate

`false`: circuit off
`true`: circuit on

Goal: `OUT = A && B`



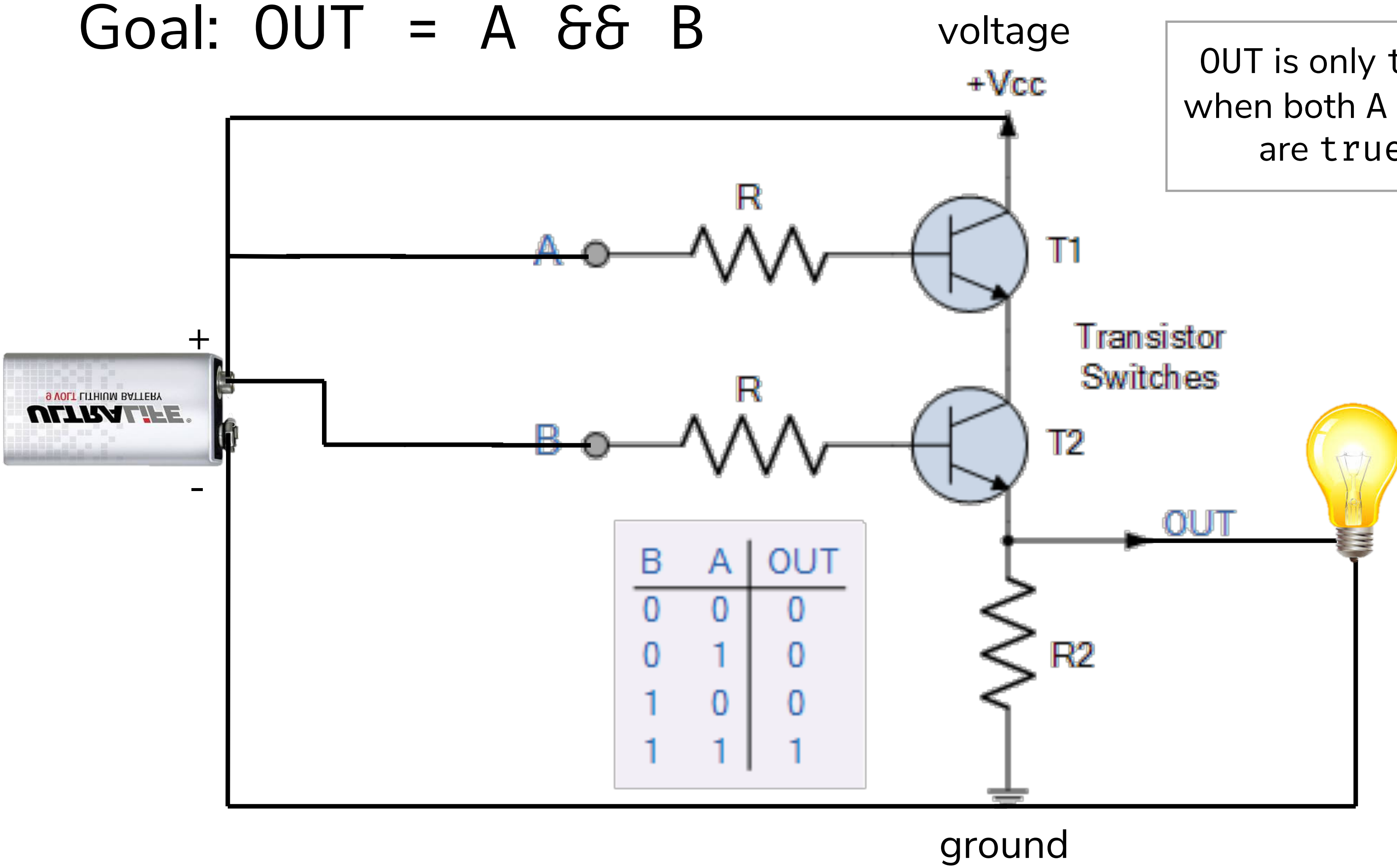
B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1

AND gate

`false`: circuit off
`true`: circuit on

Goal: `OUT = A && B`

OUT is only `true` when both A and B are `true`!



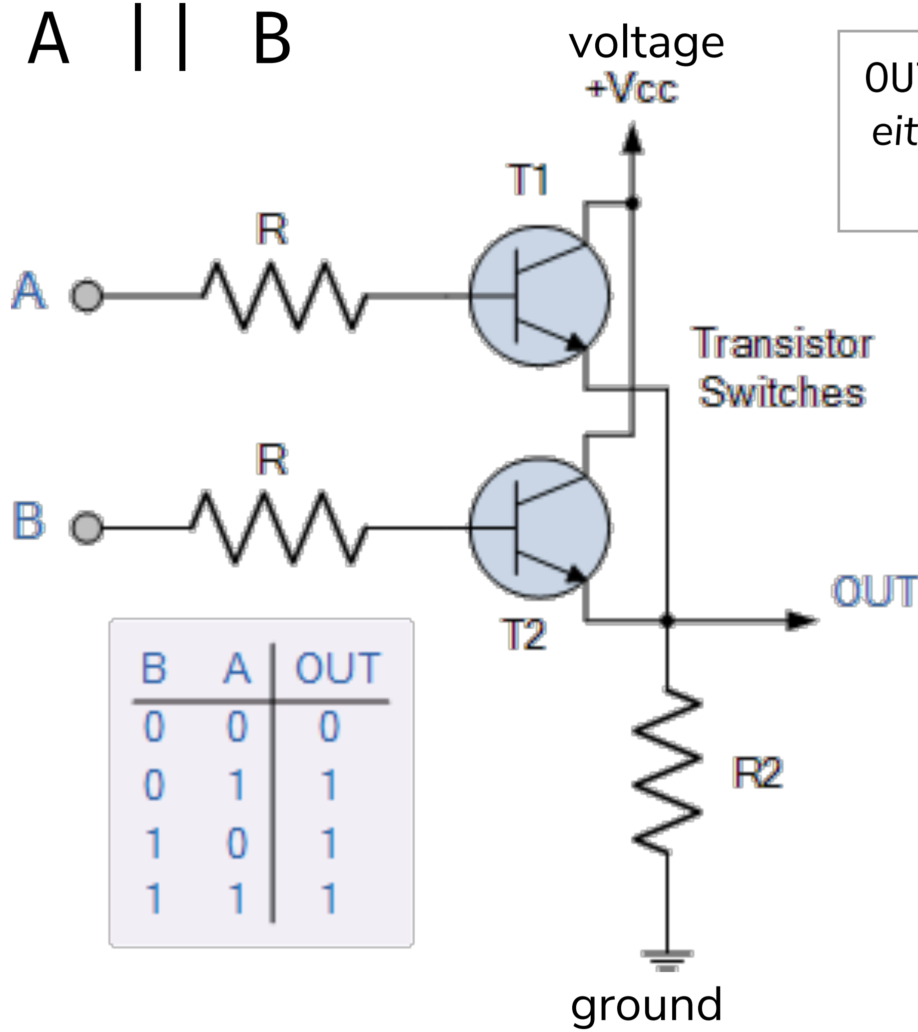
B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1

OR gate

false: circuit off
true: circuit on

Goal: OUT = A || B

OUT is true when *either* A and B are true!



B	A	OUT
0	0	0
0	1	1
1	0	1
1	1	1

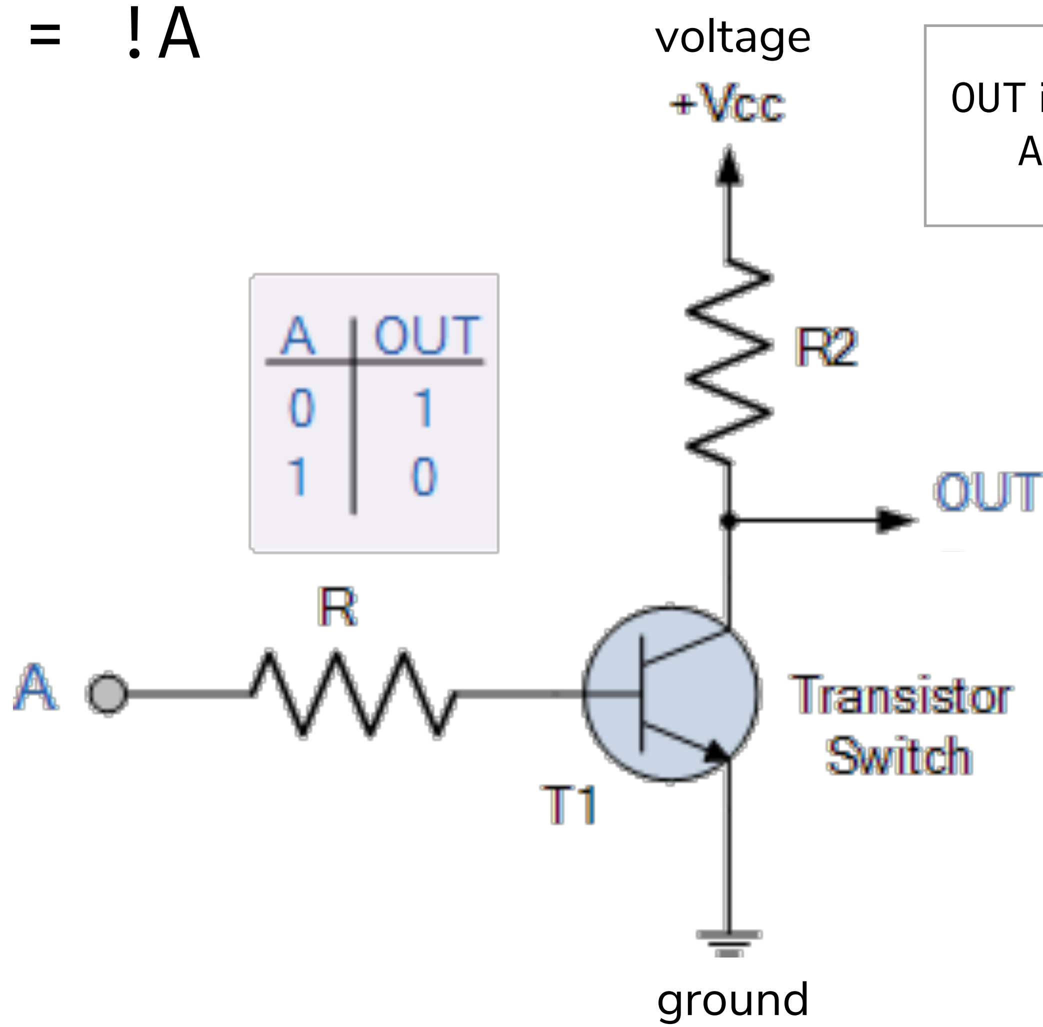
NOT gate

false: circuit off
true: circuit on

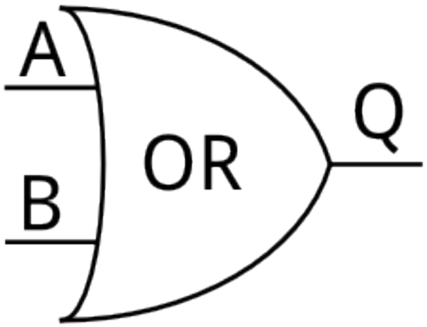
Goal: `OUT = !A`

OUT is true when A is false!

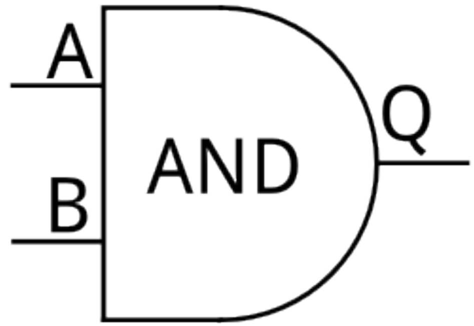
A	OUT
0	1
1	0



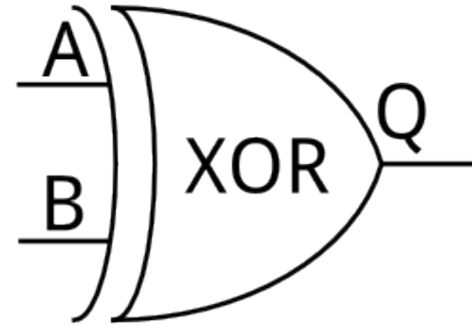
Gates Galore!



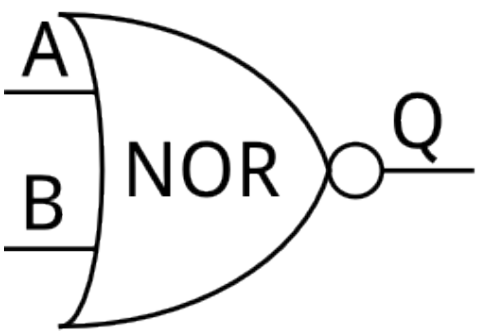
`Q = A || B`



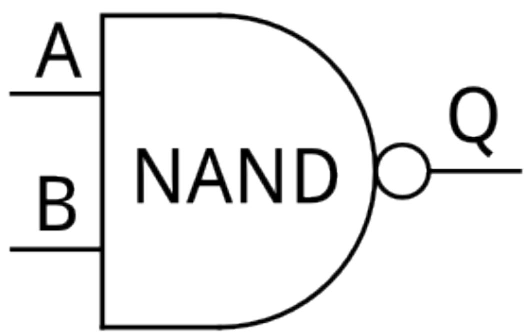
`Q = A && B`



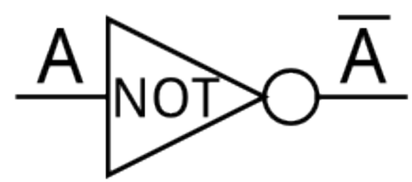
`Q = A ^ B`



`Q = !(A || B)`



`Q = !(A && B)`

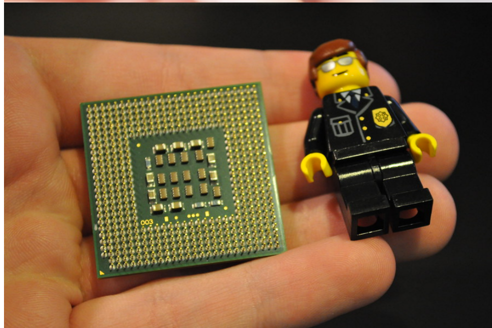


`Q = !A`

Gates can be combined...

- To build more complex circuits
  - Addition, subtraction, multiplication, comparison, etc.

- The CPU in your computer contains **billions** of transistors arranged into these circuits
  - Performs these operations billions of times *per second*



- How do we tell the CPU what to do?
  - Could switch wires on and off, but...

Computer Instructions

- We can feed certain instructions into a computer and retrieve the results.
- But what does an instruction look like?
  How do we know which one to use?
- Like all other data on a computer, instructions are just binary! (literally translated to electricity on wires)
  - Example: the number `0x83` tells computers with Intel processors to add two numbers together.
- An executable file (program) contains the binary encoding of all its instructions and data.
  - Example: `.exe` files on Windows

Instructions Are Limited

- The number and types of instructions that a CPU can perform is **always** limited.
  - Can't change the circuits after the CPU is built!
- Example: with Lightbot, you could only perform a certain number of actions:

- The instructions that a specific type of computer can understand are defined by the Instruction Set Architecture (ISA).
  - The CPU and other hardware are designed to understand **only** these predefined instructions.

Types of Instructions

- What kinds of operations do you think would be useful for a computer to support?
  - Talk with your neighbor!
    - Shut down the computer
    - Arithmetic
    - User input
    - Taking pictures
    - Internet access

Types of Instructions

- Arithmetic operations

```
c = a + b;      z = x * y;      i = h && j;
```

- Control flow: what should we do next?
  - Normally, instructions are executed sequentially. However, we can use control flow instructions to:
    - **Jump** to function calls
    - **Possibly jump** on conditional branches
    - **Possibly jump** in loops

```
int i = 0;
while (i < 3) {
    i = i + 1;
}
```

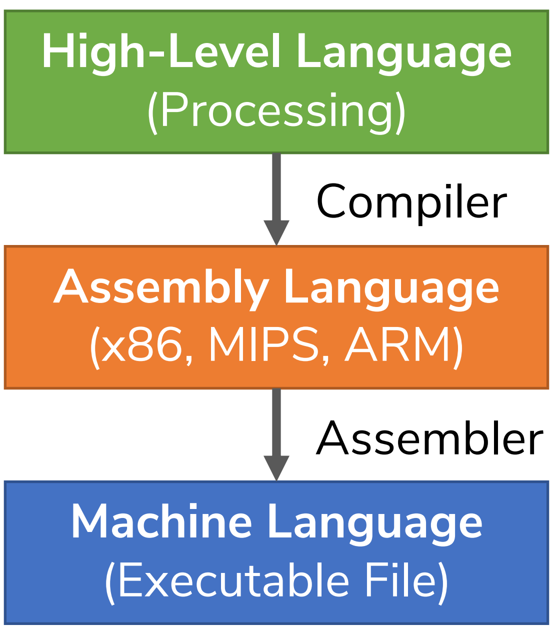
- Transfer data between CPU and memory
  - **Load** data from memory into CPU
  - **Store** data from CPU into memory

Aside: Memory

- We need somewhere to store information
  - Instructions for the computer to execute
  - Data (e.g., variables, files, etc.)
- Treat memory like a single, massive array
  - Each entry is the same size (1 byte)
  - Each entry has an index (**address**) and a value (**data**)
- If instructions need to reference data stored in memory, they can look it up using the address
  - Just like indexing into an array

Generating Instructions

- We need to specify complex tasks using only simple actions provided by instructions
  - Luckily, this is done for us – by other programs!



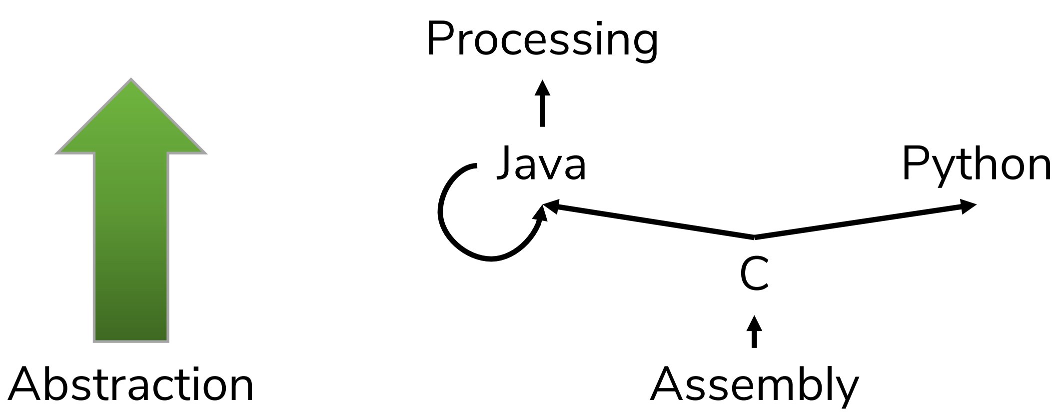

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
mov (%rsp), %edx
mov (%rsp,4), %ecx
mov %edx, (%rsp,4)
mov %ecx, (%rsp)
```

```
0000 1001 1100 0110 1010 1111 0101
1000 1010 1111 0101 1000 0000 1001
1100 0110 1100 0110 1010 1111 0101
```

Bootstrapping

- But wait – if we use another program to compiler our program, how was that program compiled?
  - Who compiles the compiler?
- The first compilers were written directly in binary.
- Bootstrapping: we can use simple languages to create increasingly complex ones.

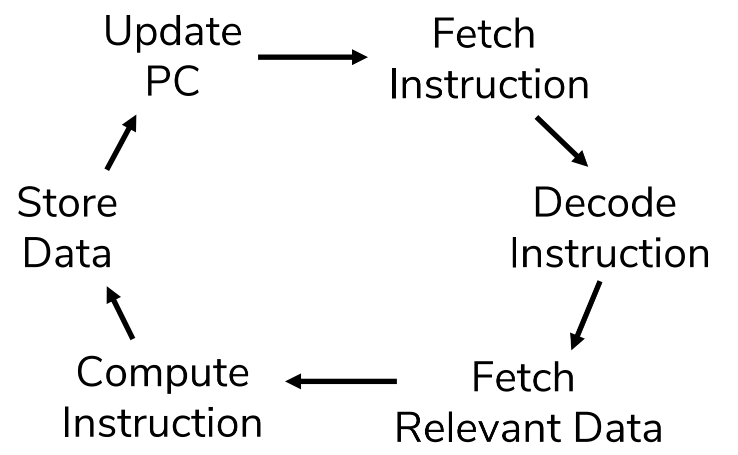


Instruction Execution

- The agent (in this case, the CPU) follows instructions **flawlessly** and **mindlessly**.
  - Identical inputs → identical results
- The **program counter (PC)** contains the memory address of the current instruction.
  - So the CPU knows what to execute
  - Updated after each instruction is executed, sometimes jumping around based on the program's **control flow**.

Fetch-Execute Cycle

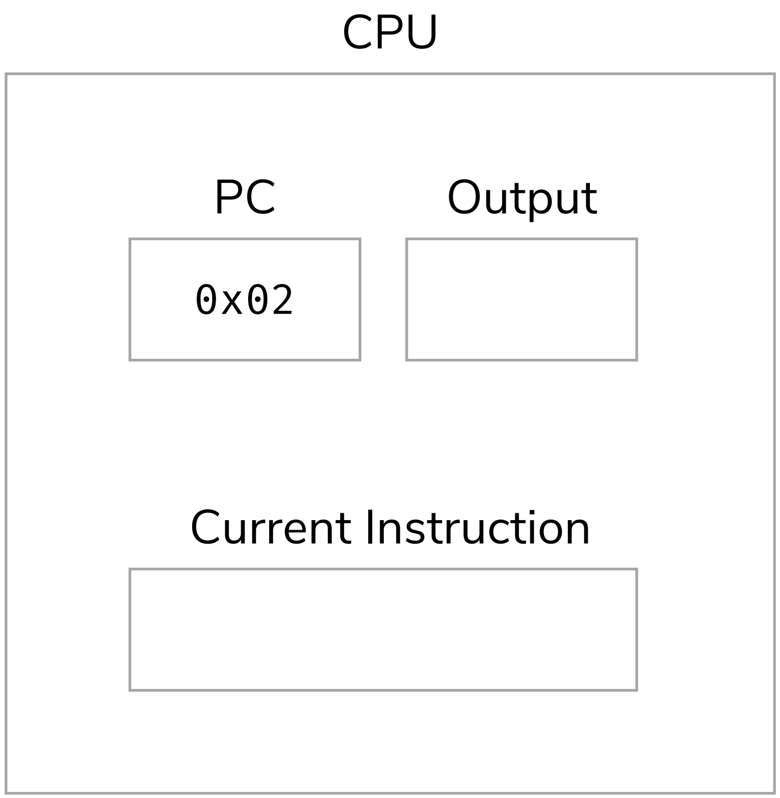
- The most basic operation of a computer is to continually perform the following cycle:
  - **Fetch** the next instruction (read from memory).
  - **Execute** the instruction based on its purpose and data.
- **Execute** portion broken down into:
  - Instruction decode
  - Data fetch
  - Instruction computation
  - Store result



Fetch-Execute Cycle (Worksheet)

Memory

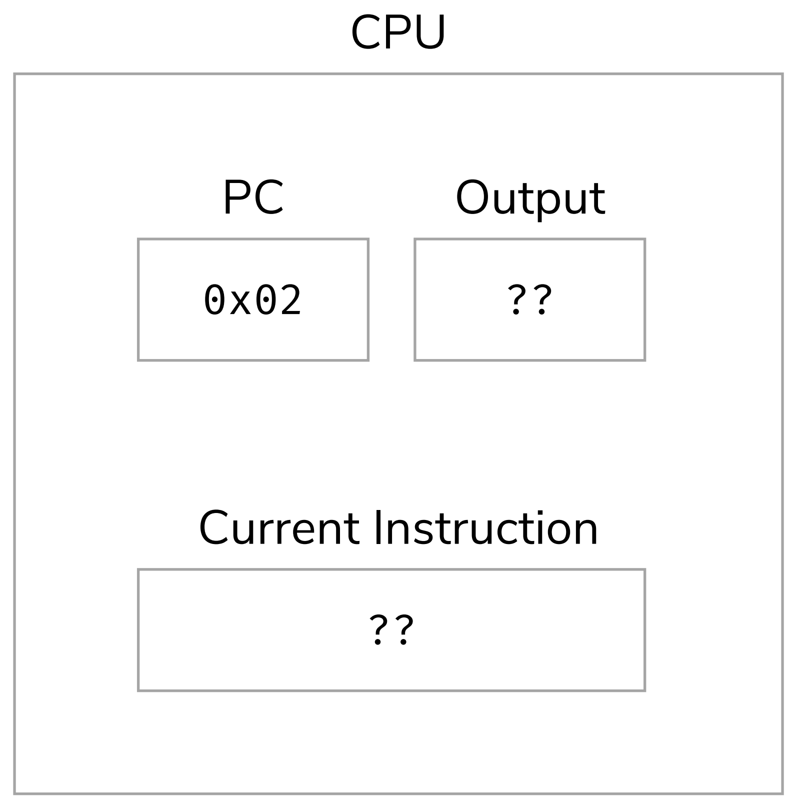
Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01



Fetch-Execute Cycle

Memory

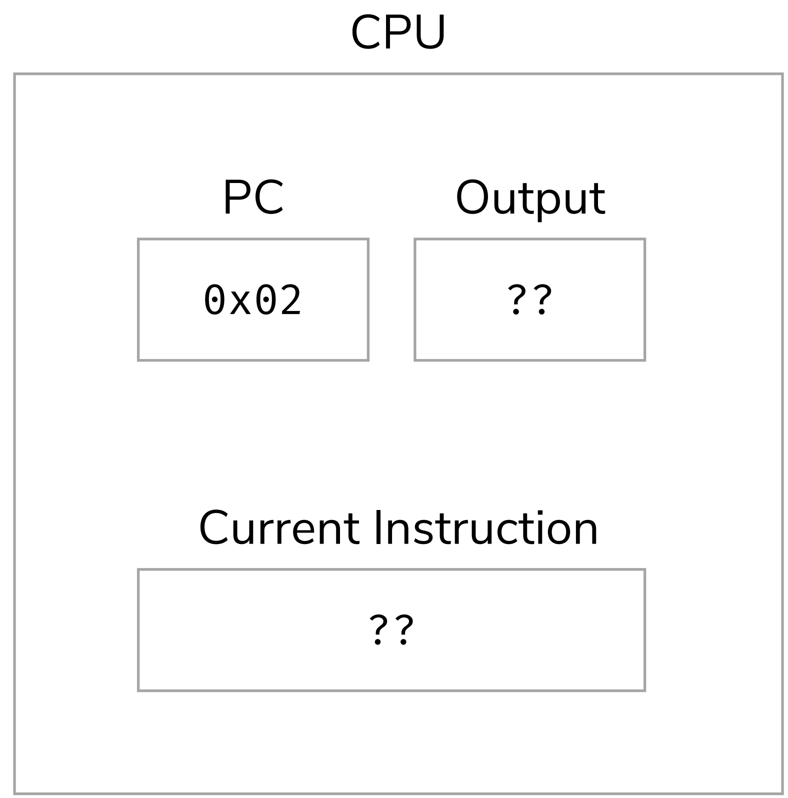
Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01



Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01

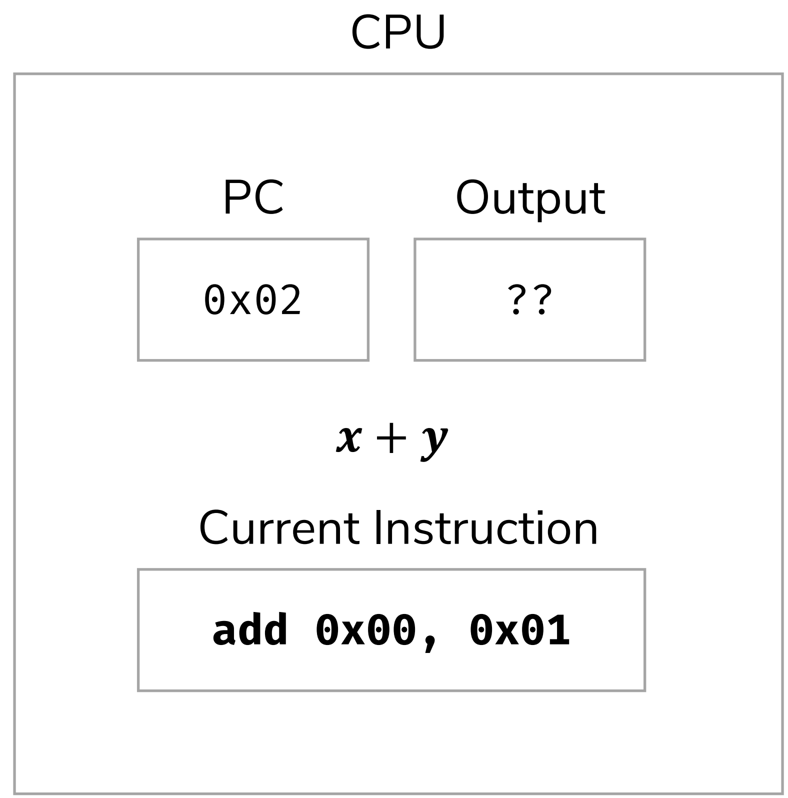


The Program Counter points to the address `0x02` in memory.

Fetch-Execute Cycle

Memory

Address	Value
`0x00`	`12`
`0x01`	`6`
`0x02`	`add 0x00, 0x01`
`0x03`	`store 0x01`

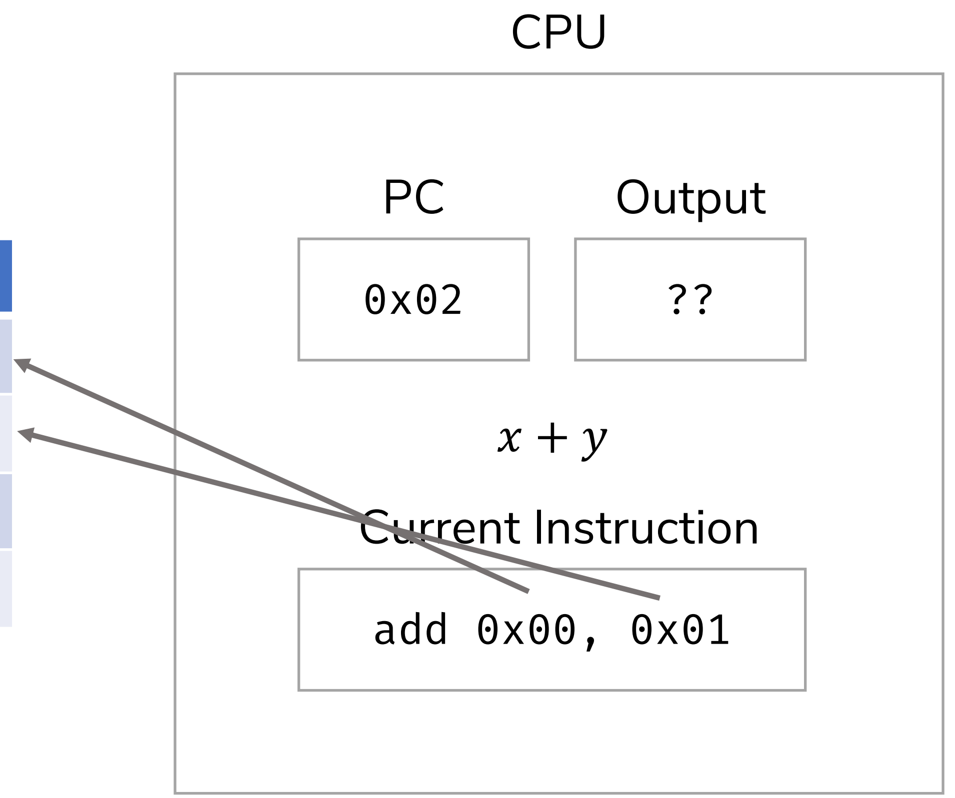


Fetch the instruction.

Fetch-Execute Cycle

Memory

Address	Value
`0x00`	`12`
`0x01`	`6`
`0x02`	`add 0x00, 0x01`
`0x03`	`store 0x01`

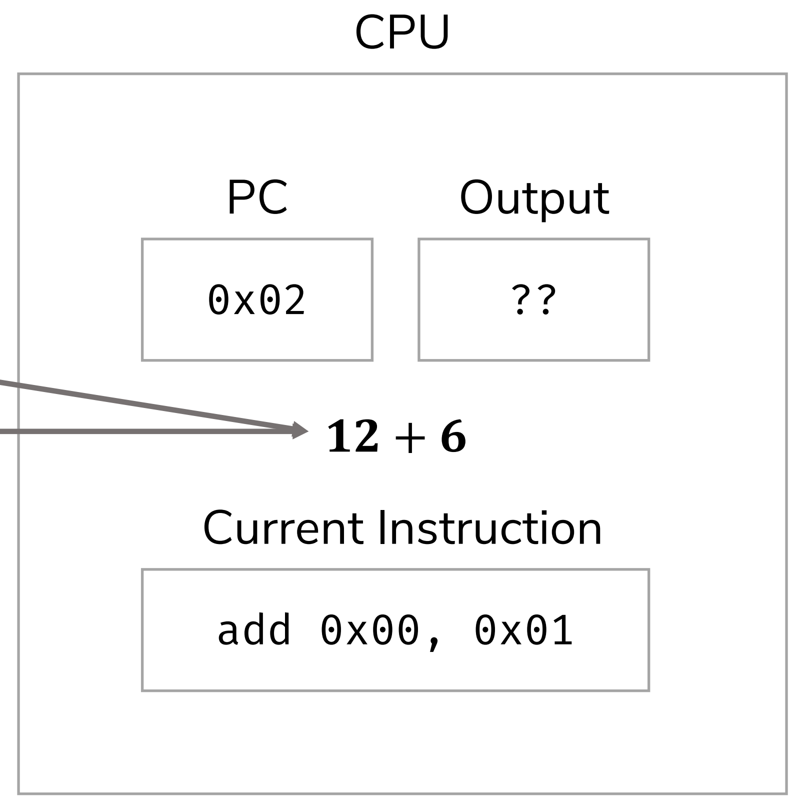


Decode the instruction.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01



Fetch the relevant data from memory.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01

CPU

PC: 0x02

Output: ??

$12 + 6 = \mathbf{18}$

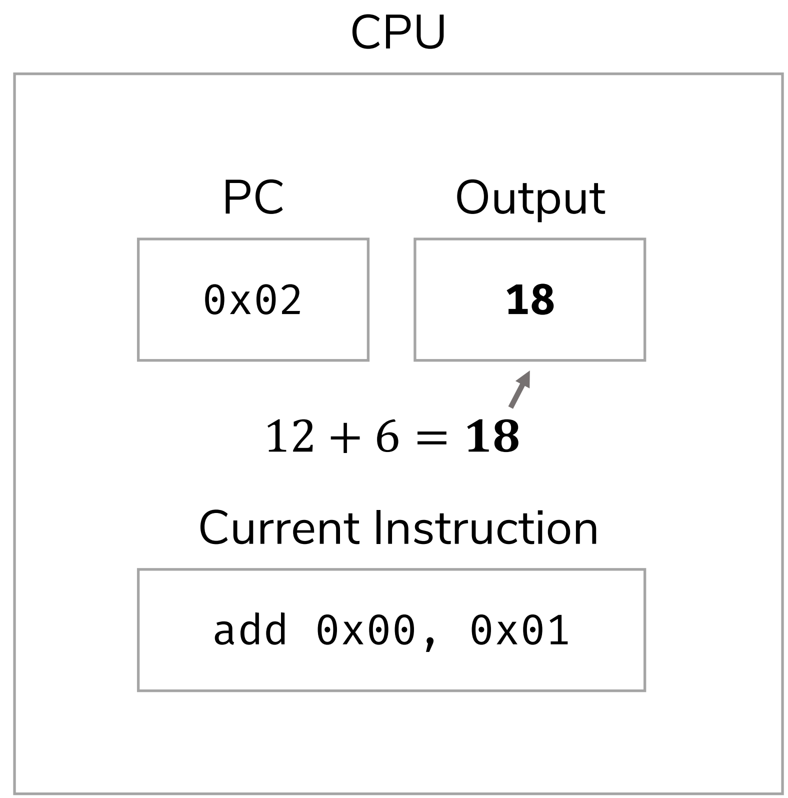
Current Instruction: add 0x00, 0x01

Compute the result...

Fetch-Execute Cycle

Memory

Address	Value
`0x00`	`12`
`0x01`	`6`
`0x02`	`add 0x00, 0x01`
`0x03`	`store 0x01`

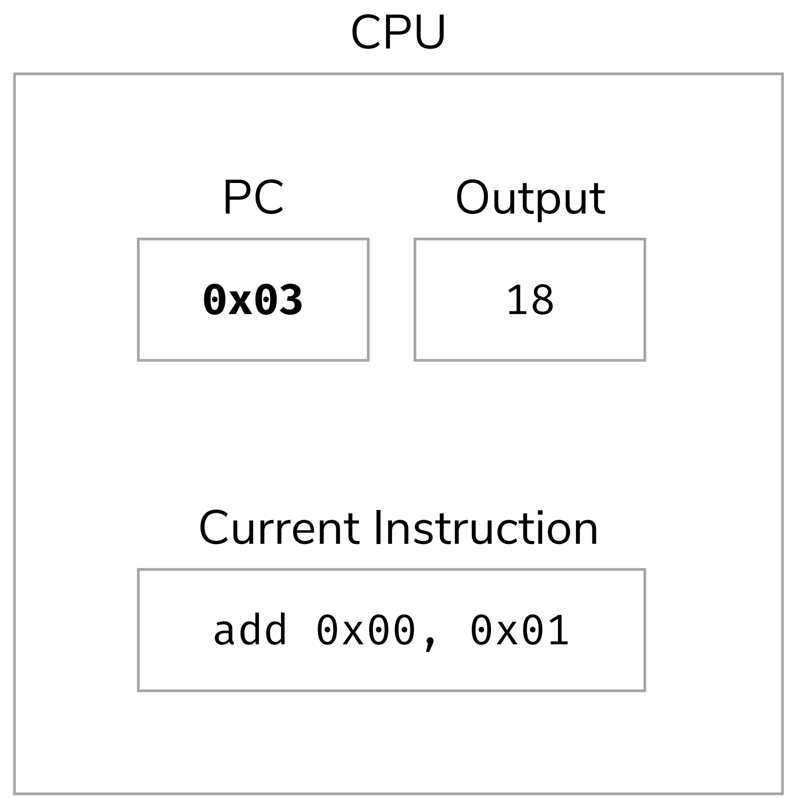


...and place it in temporary storage.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01



Now, advance the Program Counter to point to the next instruction.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01

CPU

PC

0x03

Output

18

Current Instruction

add 0x00, 0x01

Now, advance the Program Counter to point to the next instruction.

Fetch-Execute Cycle

Memory

Address	Value
`0x00`	`12`
`0x01`	`6`
`0x02`	`add 0x00, 0x01`
`0x03`	`store 0x01`

CPU

PC

`0x03`

Output

`18`

Current Instruction

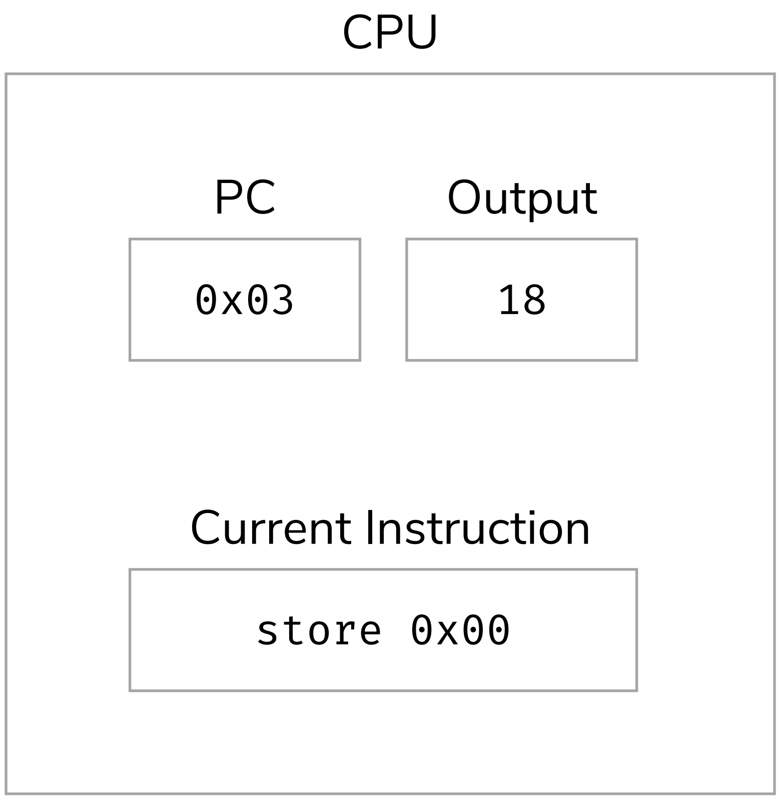
`store 0x01`

Fetch the instruction into the CPU.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01

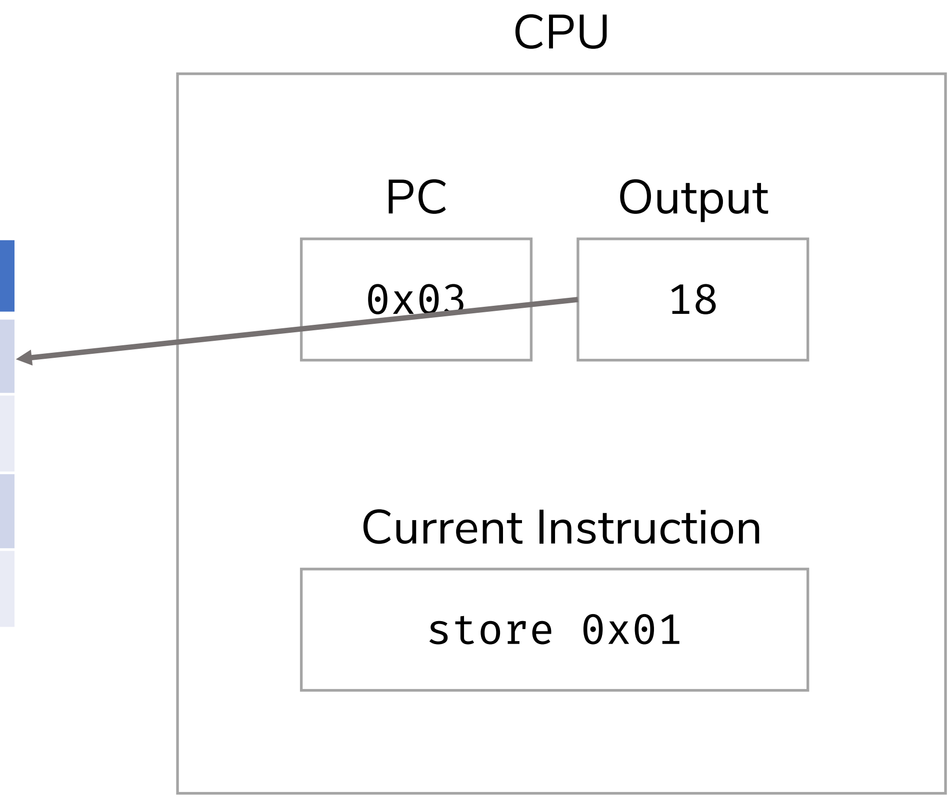


Decode the instruction: “store the output value into memory at `0x00`.”

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01

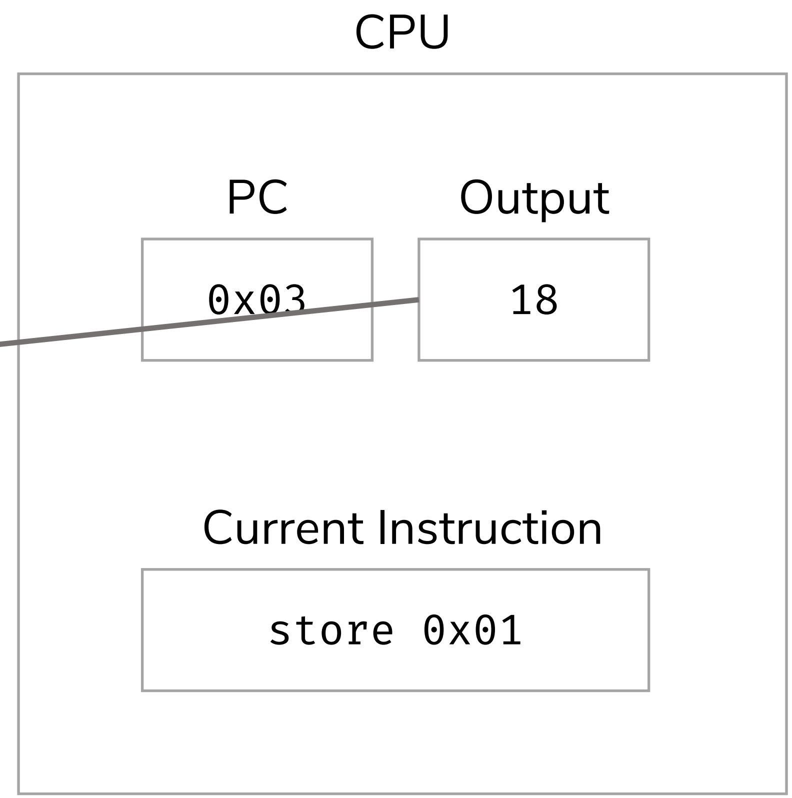


Execute the instruction.

Fetch-Execute Cycle

Memory

Address	Value
0x00	**18**
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01

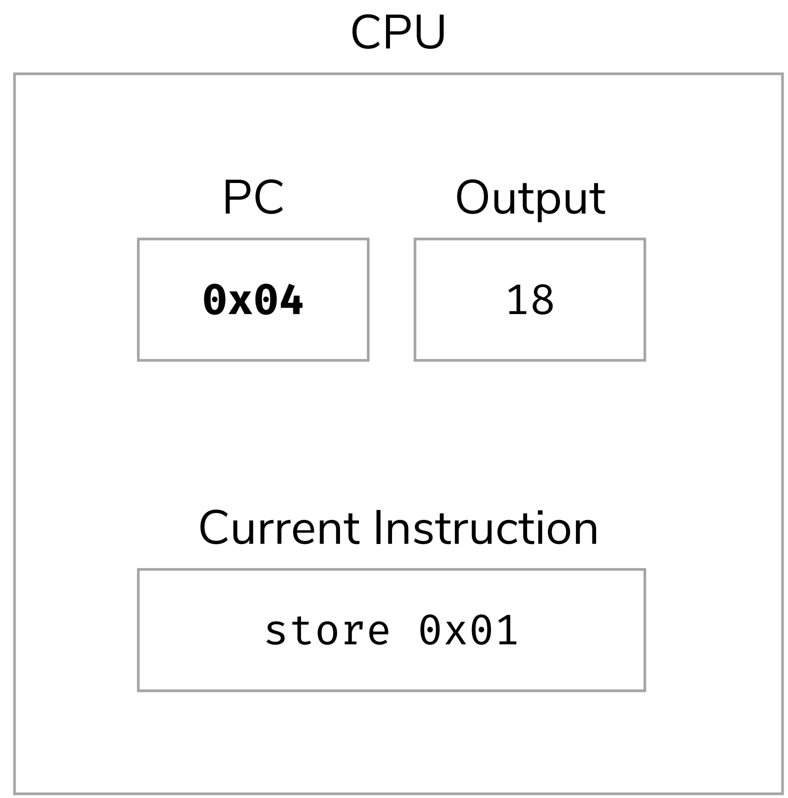


Execute the instruction.

Fetch-Execute Cycle

Memory

Address	Value
0x00	**18**
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x01



And so on, and so forth...

Clock Rate

- The speed at which your computer can perform the Fetch-Execute cycle.
  - Must ensure that the clock rate is slow enough to accommodate the **slowest instruction.**
- Clock rate is usually given in Hertz. $1\ hertz = \frac{1\ instruction}{second}$
  - Example: $2\ Ghz = 2 * 10^9\ Hz = 2\ \text{billion}\ \frac{\textbf{instructions}}{\textbf{second}}$
- However, clock rate is often **not** a good indicator of speed
  - Modern CPUs spend a lot of their time idle, waiting for data from memory, disk drives, networks, etc.

Example: Running Processing

- The Processing environment compiles your code into machine language (0s and 1s, which becomes electricity on wires in the CPU)
- Memory is automatically set aside for the program's instructions, variables, and data.
- Starting from the beginning of your program (in the case of Processing, the `setup()` function) the computer will continuously perform the Fetch-Execute cycle.
  - It will continue executing until the end of the program is reached, or it encounters an error.