Lecture 19: Computers

Pay no attention to the man behind the curtain.

Samuel Wolfson // CSE 120, Winter 2018

Final Project Administrivia

- Show off all the cool skillz you've learned this quarter!
- Three parts:
  - Design Document due **Saturday, Feb 24**
    - Includes project name and storyboard
  - Project Update due **Thursday, March 1 in lab**
  - Final Submission due **Friday, March 9**
- Single program, done alone or with a partner
  - Should be much more complex than Creativity Assignment
  - Must include at least 3 hand-created assets

What we're doing

- **What components are inside of a computer?**
- How do we tell the hardware what to do?
- How are our instructions executed?

Hard Drive

Processor

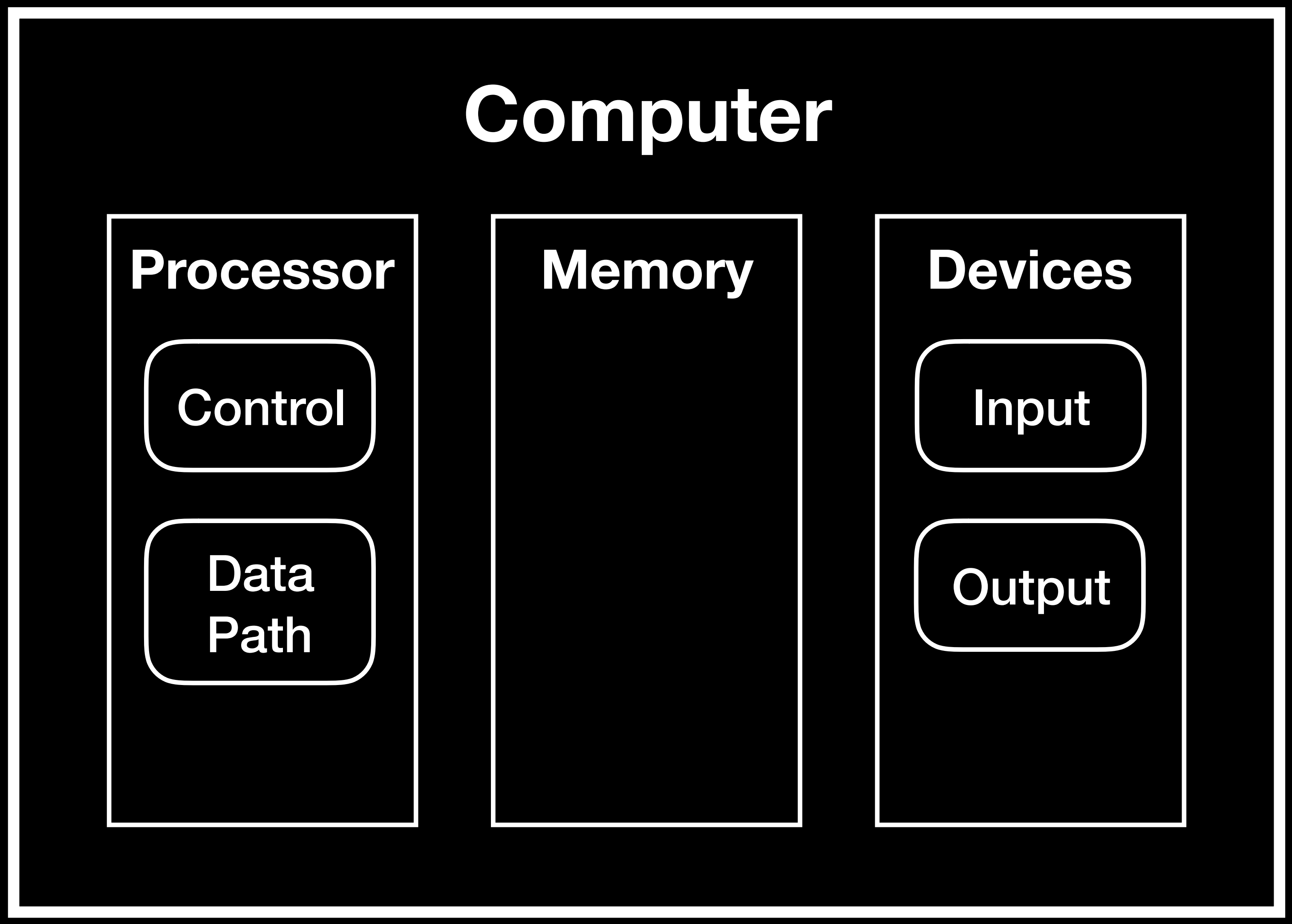
What are different computer components that you've heard of?

Motherboard

Memory

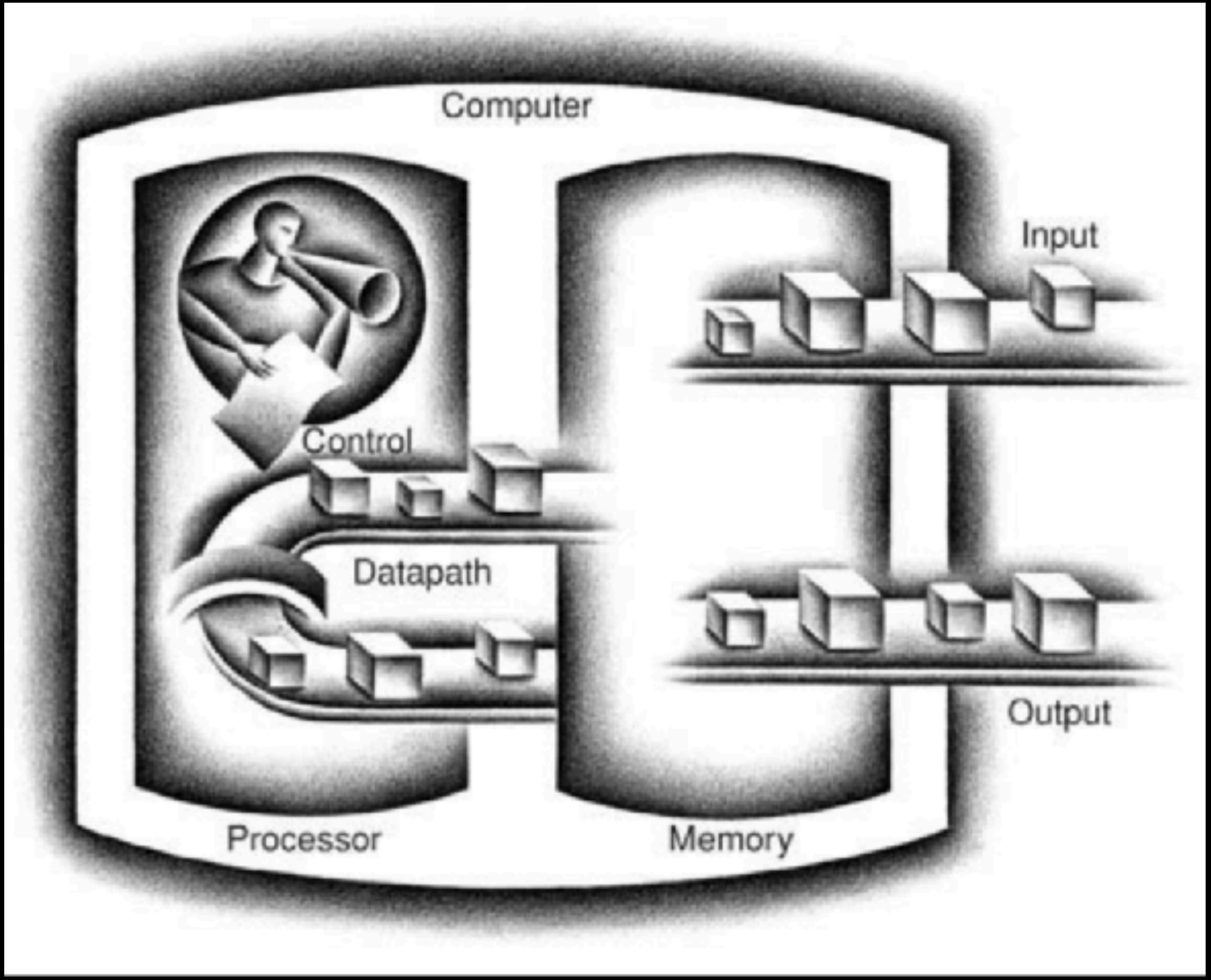
Five Logical "Components" of a Computer

- Control
- Data path
- Memory
- Input
- Output



Data "flows" through each of these components in a computer.

How does data "flow" through a computer?



Input

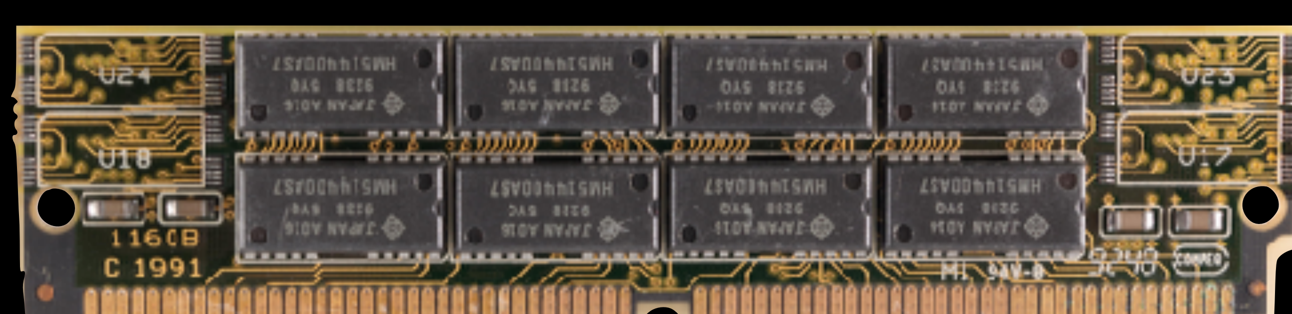
- Devices that send information **into** the computer
- Examples we've seen
  - Mouse
  - Keyboard
  - Ethernet / WiFi
- Other Examples
  - Microphone
  - Disk (read)

Output

- Devices that receive information sent **out of** the computer
- Examples we've seen
  - Monitor / graphics card
  - Ethernet / WiFi
- Other Examples
  - Speakers
  - Printers
  - Disk (write)

Memory

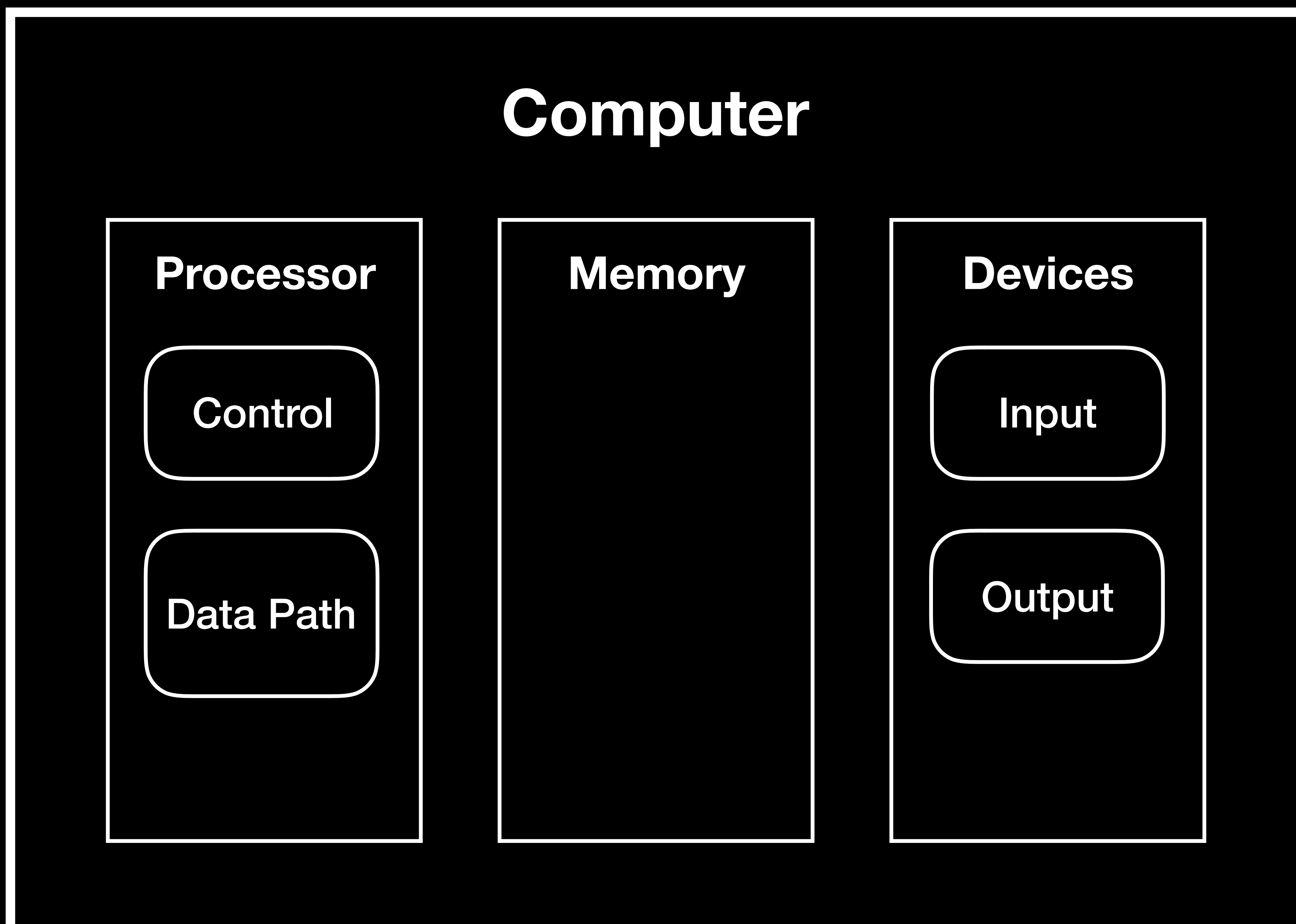
- Used for **temporary** data storage
- Much faster than hard drives, but forgets everything when power is lost!
  - **Volatile** storage
- Permanent storage goes to the disk instead
  - **Nonvolatile** storage



Central Processing Unit (CPU)

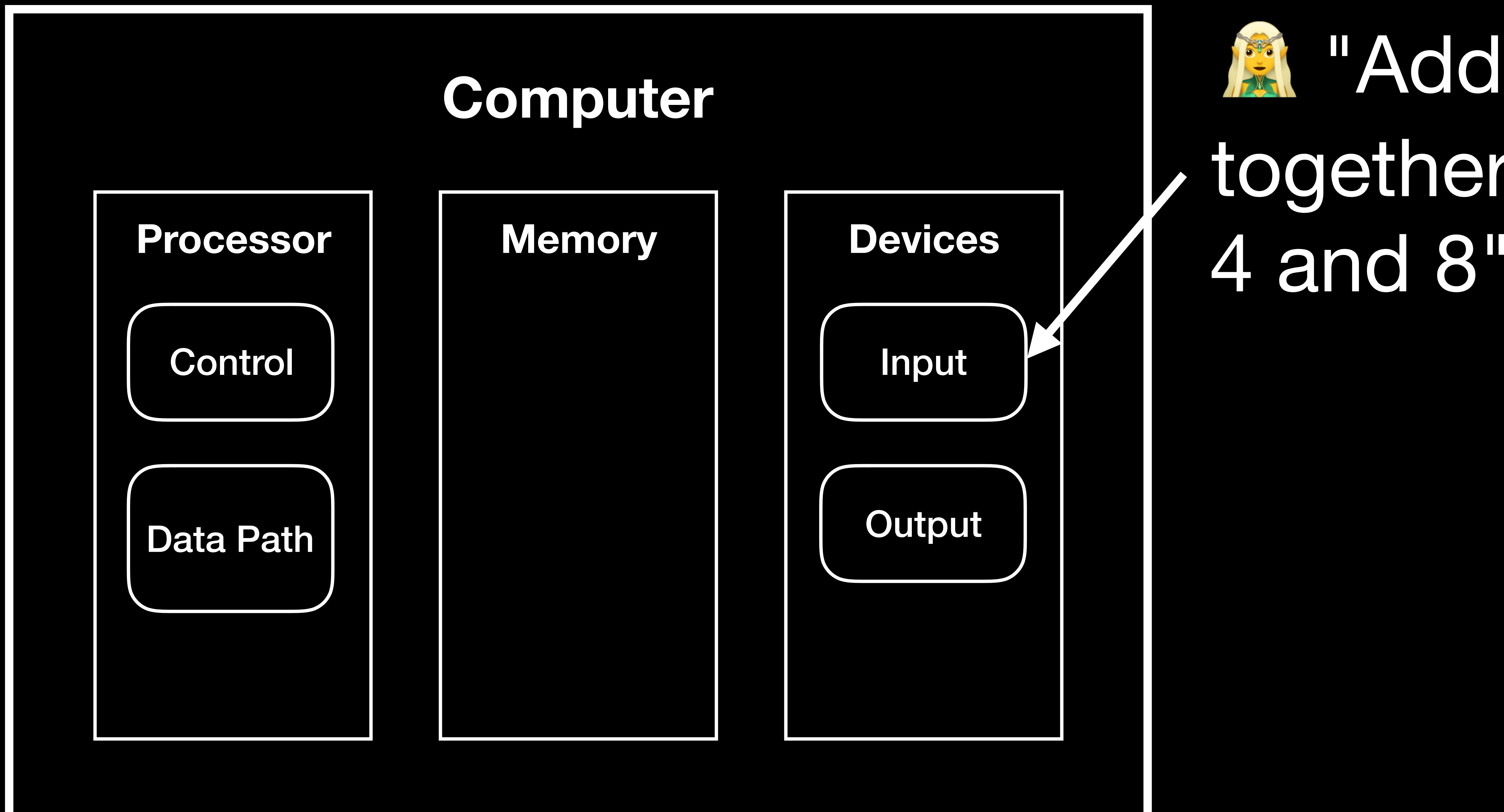
- "Brain" of the computer -- contains circuitry that carries out all instructions given as input.
  - Data path: circuits that **hold** and **process** data.
    - Arithmetic operations (addition, subtraction, etc.)
  - Control: tells the data path components **how** they should handle certain instructions.

Data Flow Example



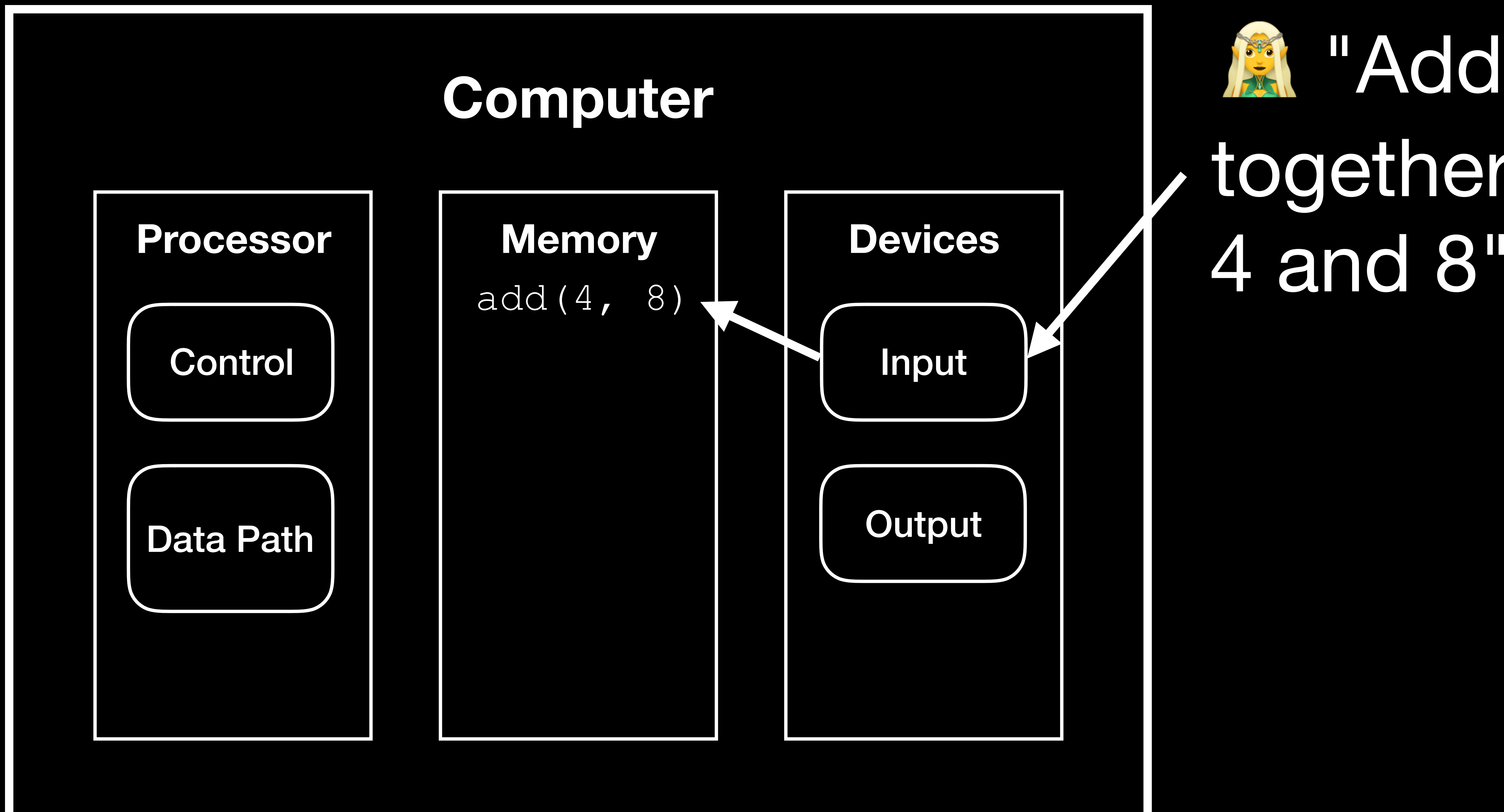
🧝‍♀️ "Add together 4 and 8"

Data Flow Example



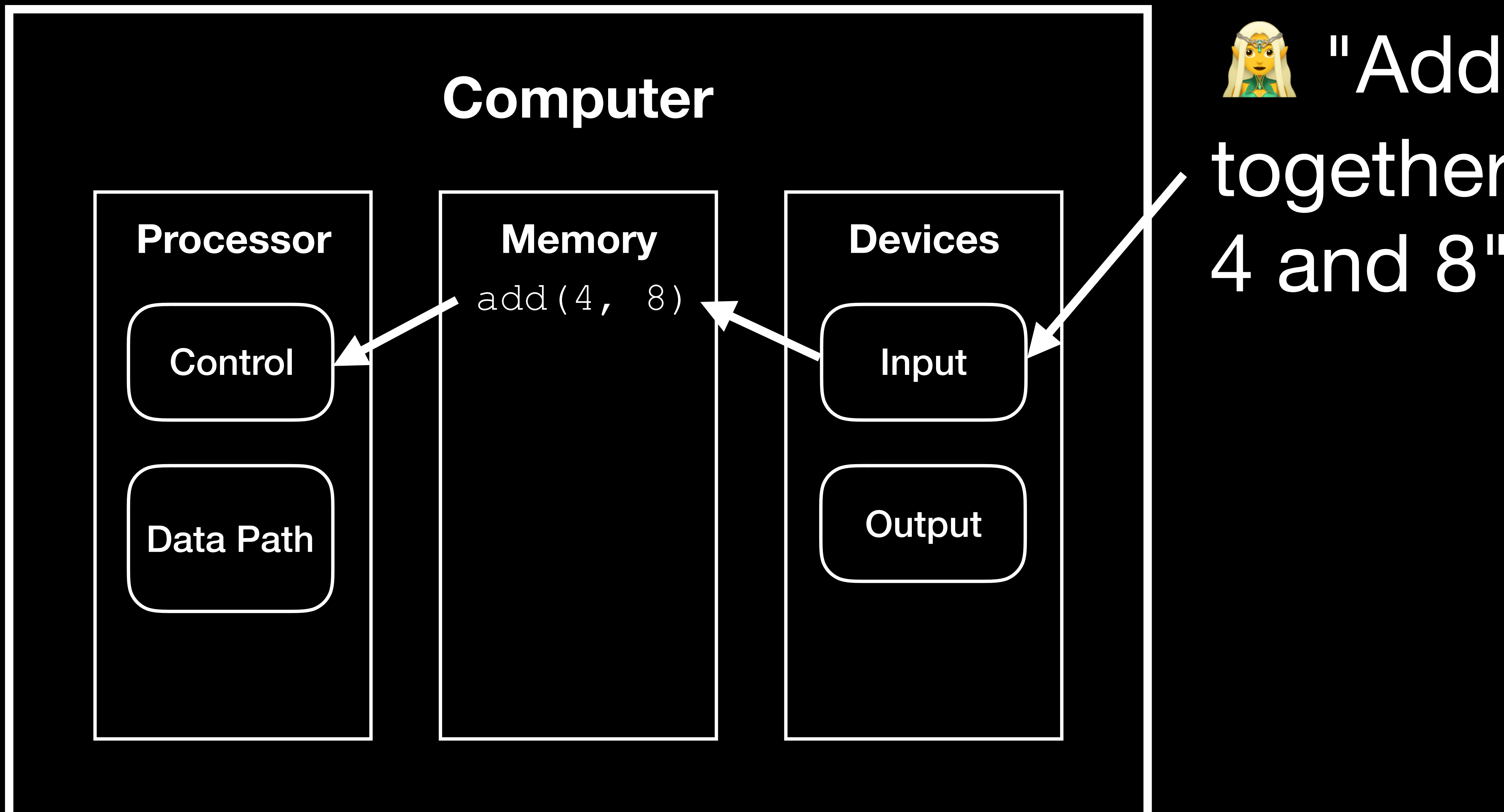
User inputs some command via keyboard, mouse, etc.

Data Flow Example



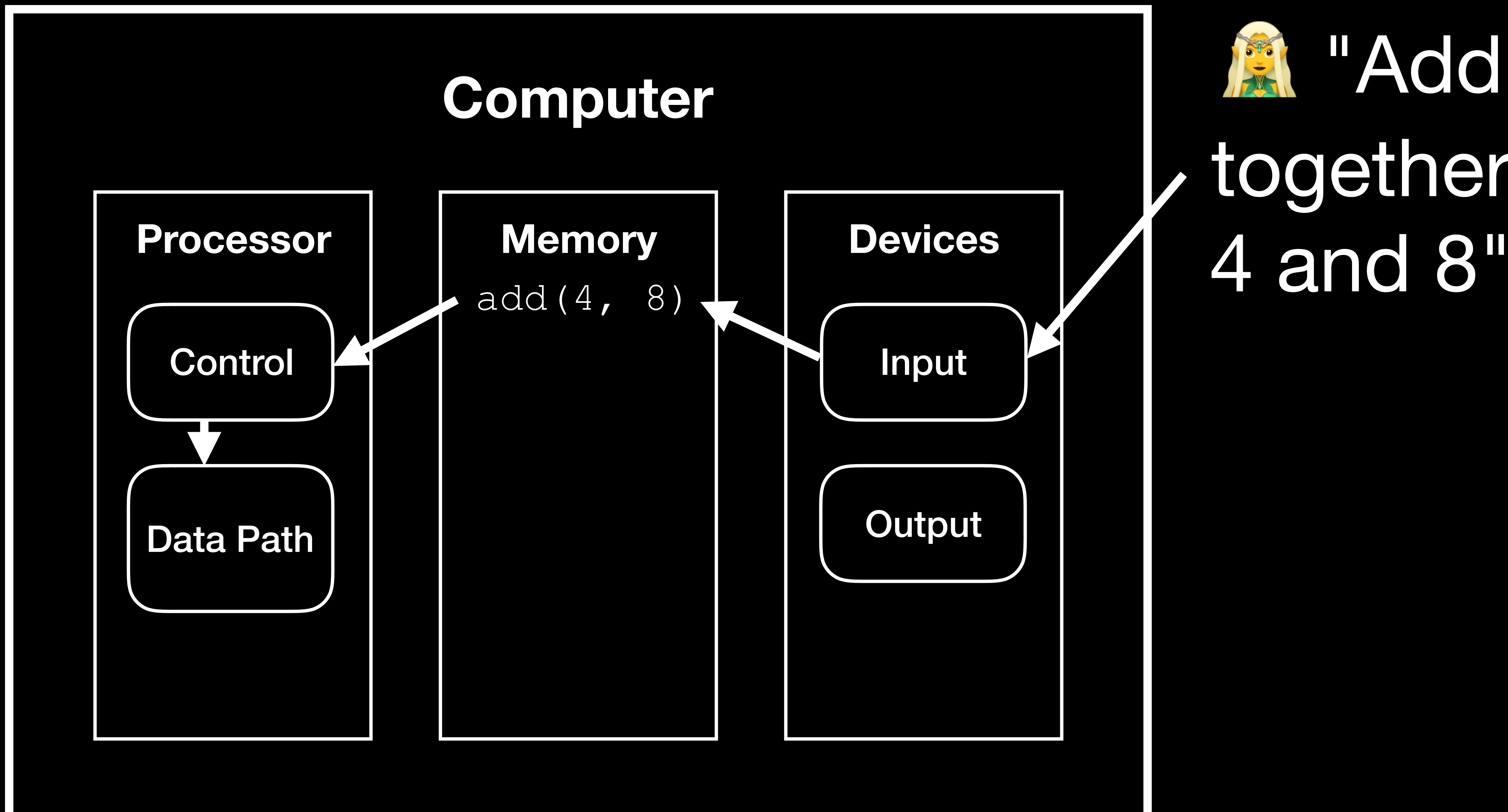
Command is translated into bits, and stored in computer's memory.

Data Flow Example



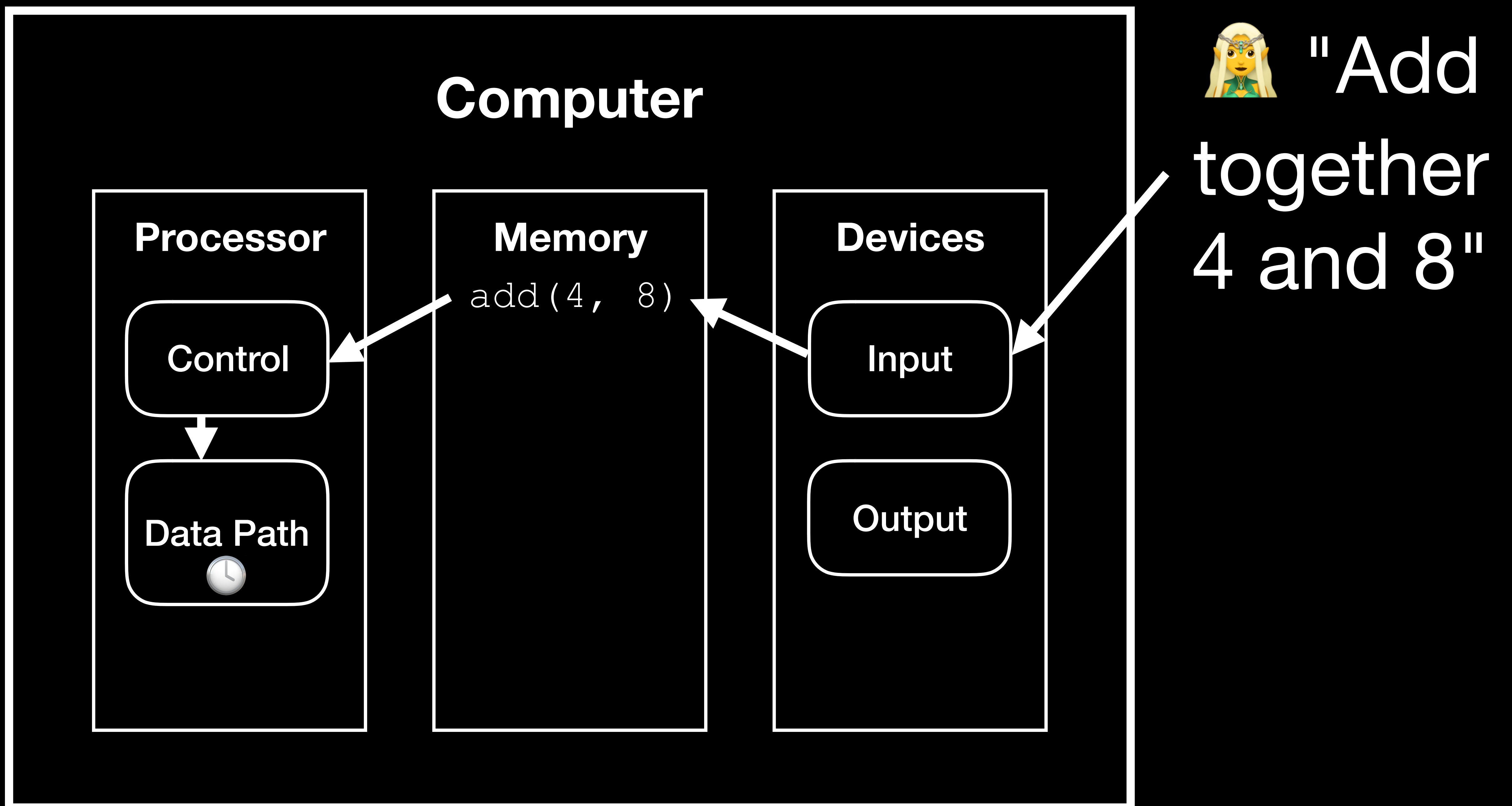
The control unit of the CPU reads data from memory into the CPU.

Data Flow Example



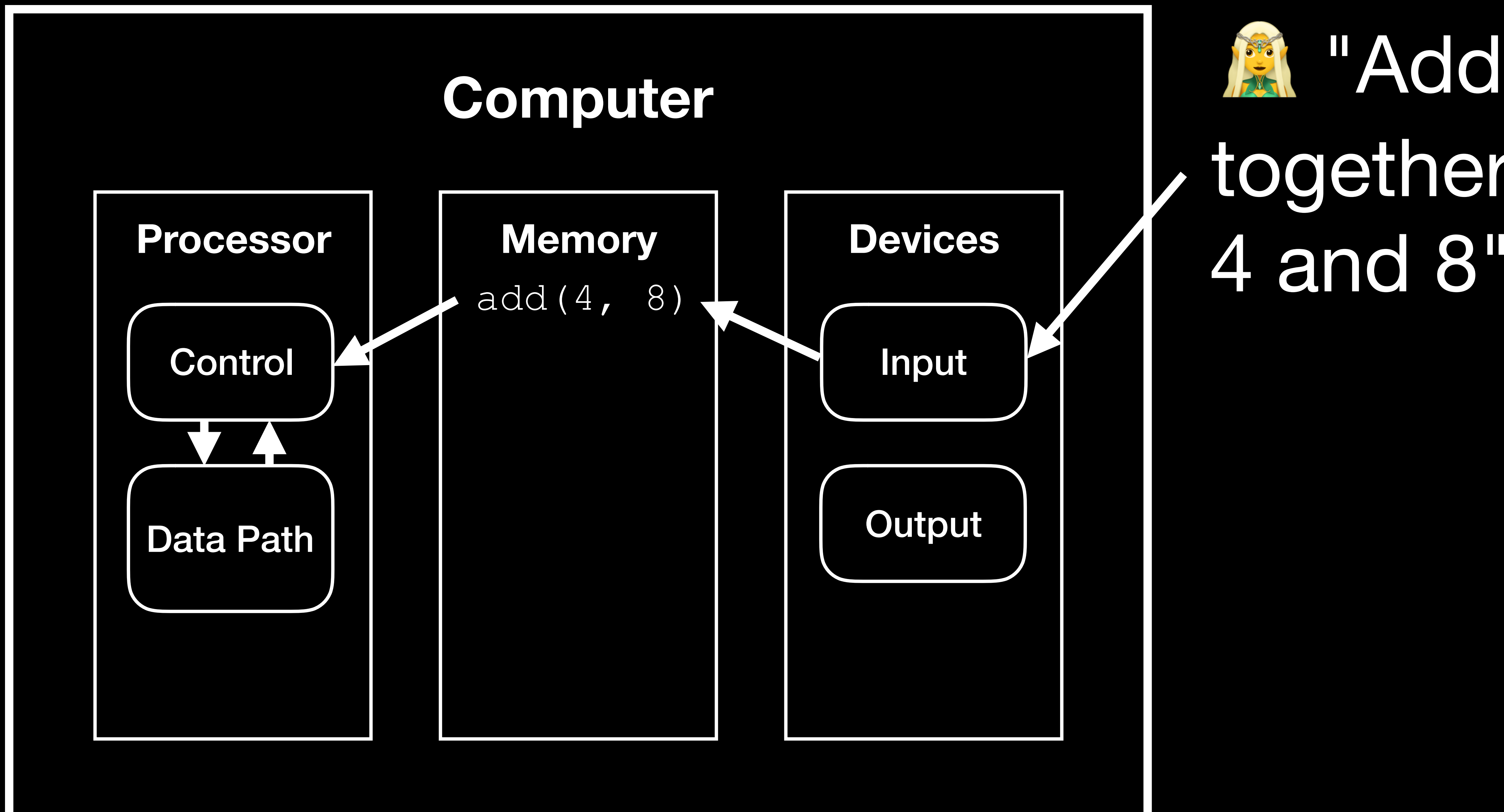
The control unit determines that addition is performed by the arithmetic & logic unit.

Data Flow Example



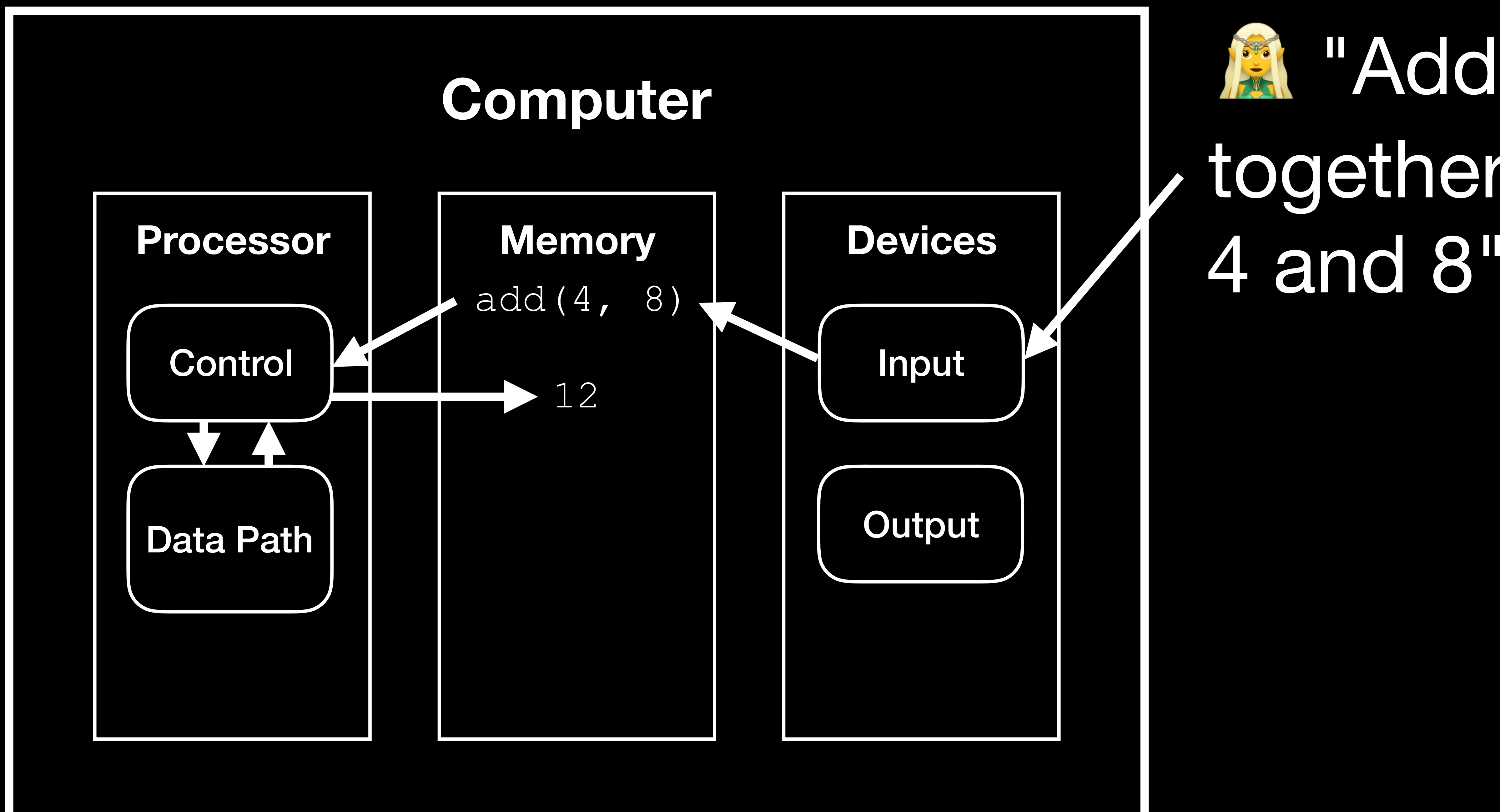
The ALU, which is part of the data path, performs the operation.

Data Flow Example



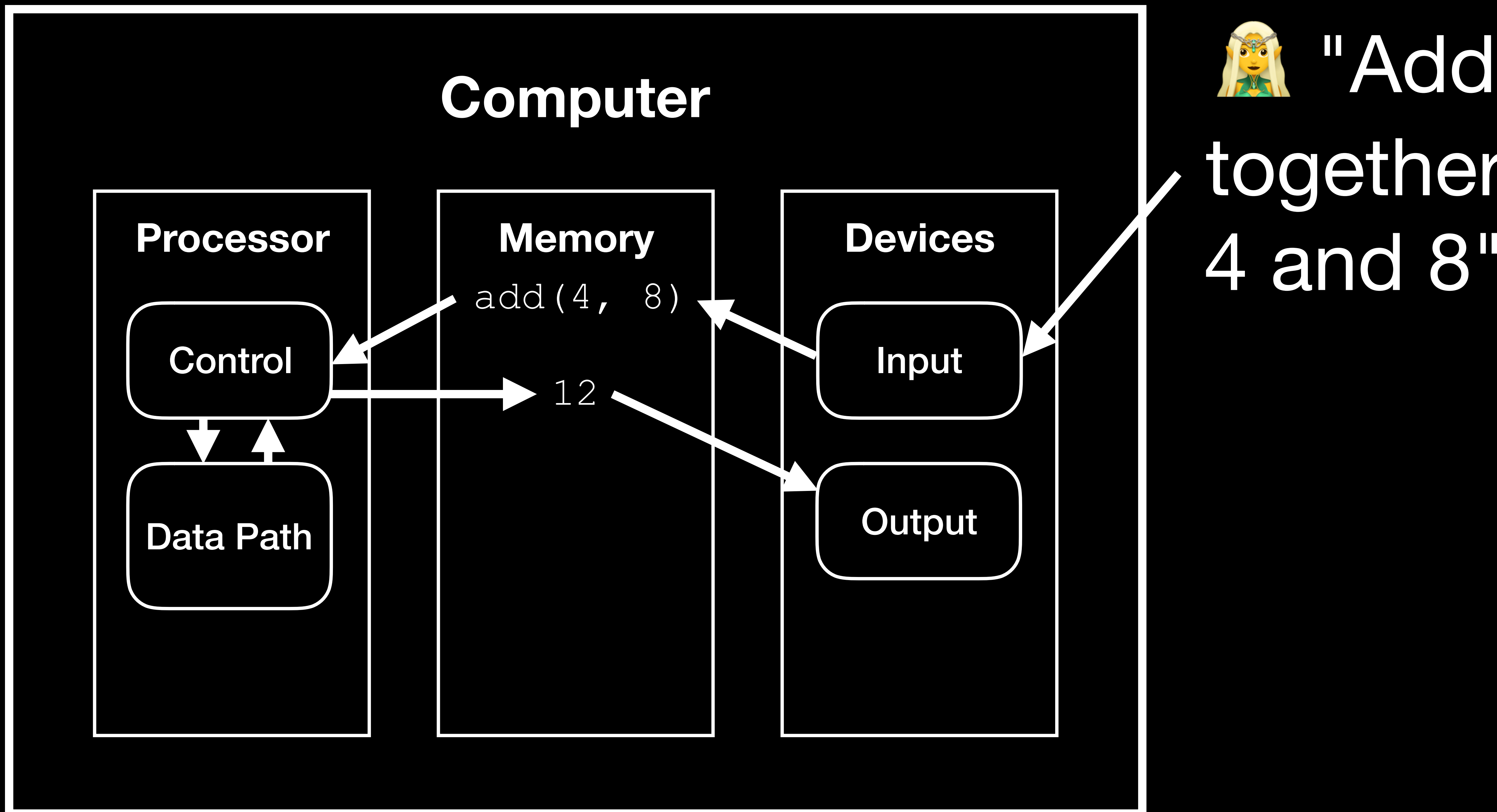
Once the operation is complete, the control unit takes the result...

Data Flow Example



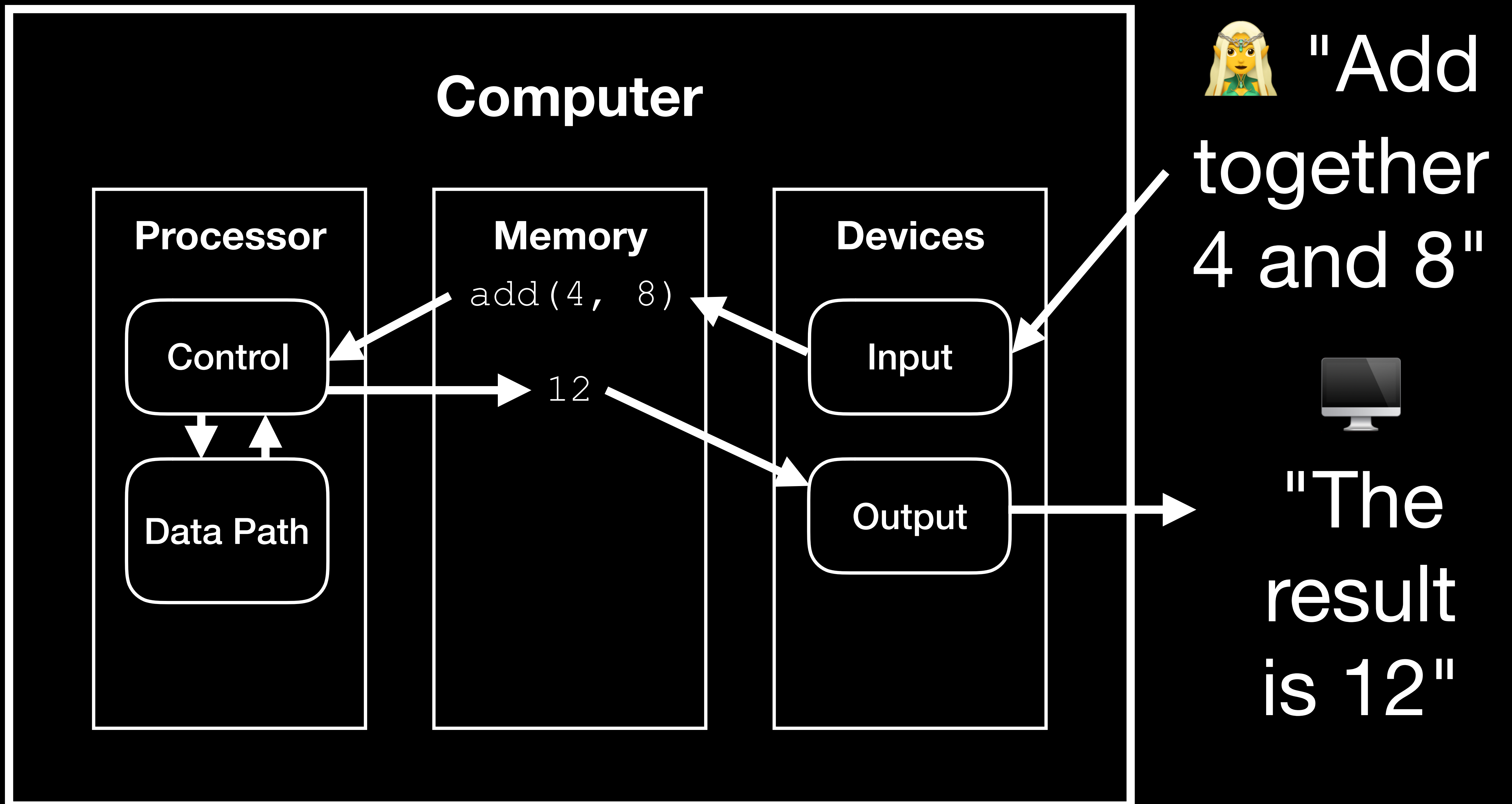
...and stores it back into memory.

Data Flow Example



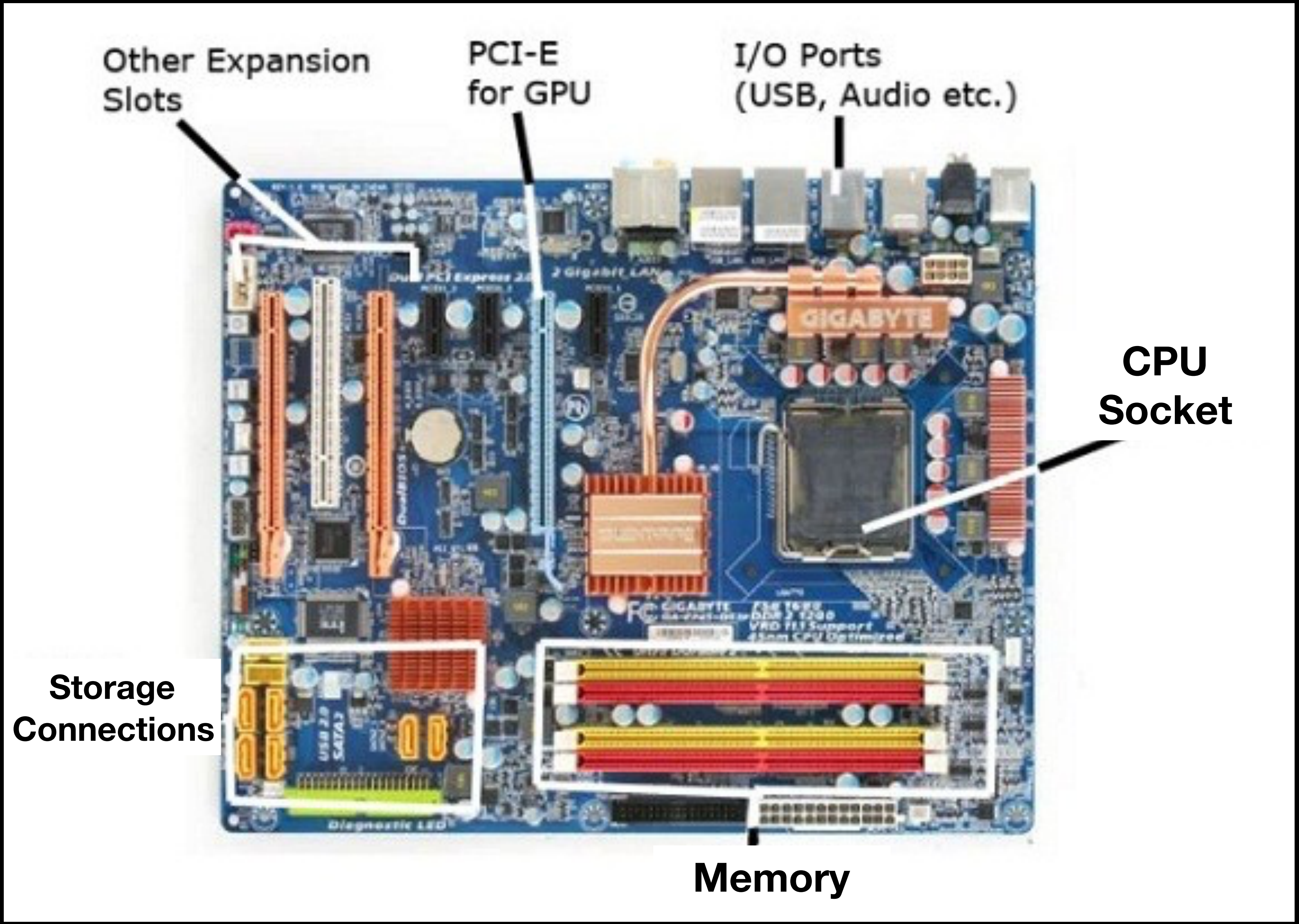
Finally, the result is written to an output device...

Data Flow Example



...and returned to the user.

Physical View



Operating Systems

- Everything that we've talked about up to this point is **hardware.**
- **Operating systems** tie hardware and software together.
  - Provide **abstraction** for user programs
  - "Just another program," but has special privileges
    - Can read data owned by other programs
  - Heavily responsible for the security of the data and the programs in your computer

There are lots of operating systems.

What we're doing

- What components are inside of a computer?
- **How do we tell the hardware what to do?**
- How are our instructions executed?

Computer Instructions

- We can feed certain instructions into a computer, and retrieve the results.
- But what does an instruction actually look like? How do we know which one to use?
- Like all other data on a computer, instructions are just binary!
  - 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.
  - Example: with LightBot, you could only perform a certain number of actions:

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

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 the CPU and memory
  - **Load** data from memory into CPU
  - **Store** data from CPU into memory

Memory (reprise)

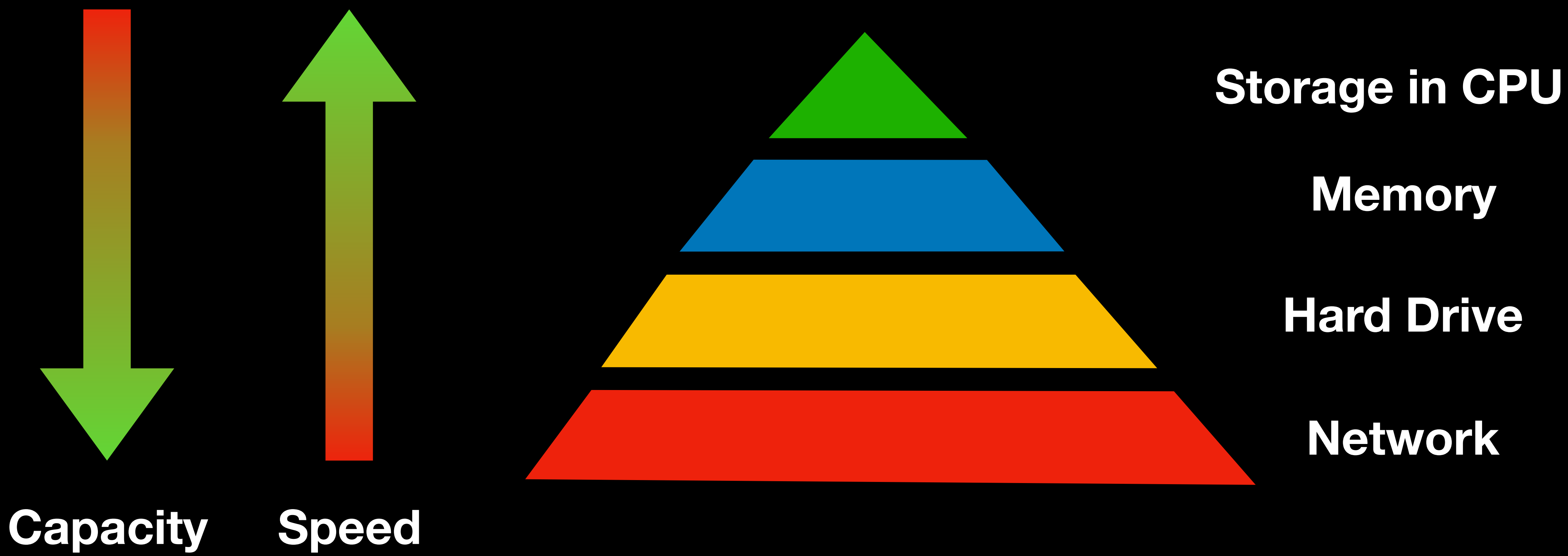
- We can treat memory like a single, massive array.
  - Each memory entry is the same size (1 byte).
  - Each memory entry has an index (the **address**) and a value (the **data**).
- If our instructions need to reference data that is stored in memory, they can look it up using the memory address.

Address	0x00	0x01	0x02	0x03	0x04	...	0xFF...FF
Value	0xDE	0xAD	0xBE	0xEF	0xCA		0xFE

Example: suppose the `load` instruction loads a value from memory into the CPU. What does `load 0x04` return?

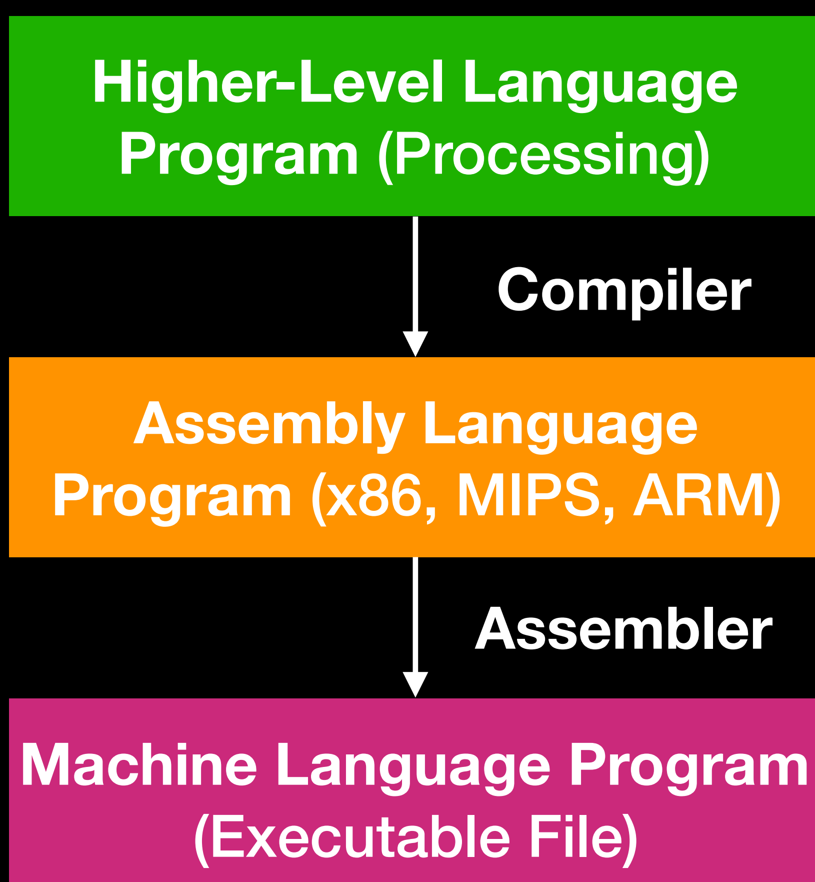
Where Are The Instructions?

- When a program is running, the instructions for that program are stored in **memory**.
  - It's much faster to read instructions from memory than it is to read them from a file on the disk.



Generating Instructions

- We need to figure out how we can specify complex tasks using simple actions.
  - Luckily, this is usually done for us -- by other programs!



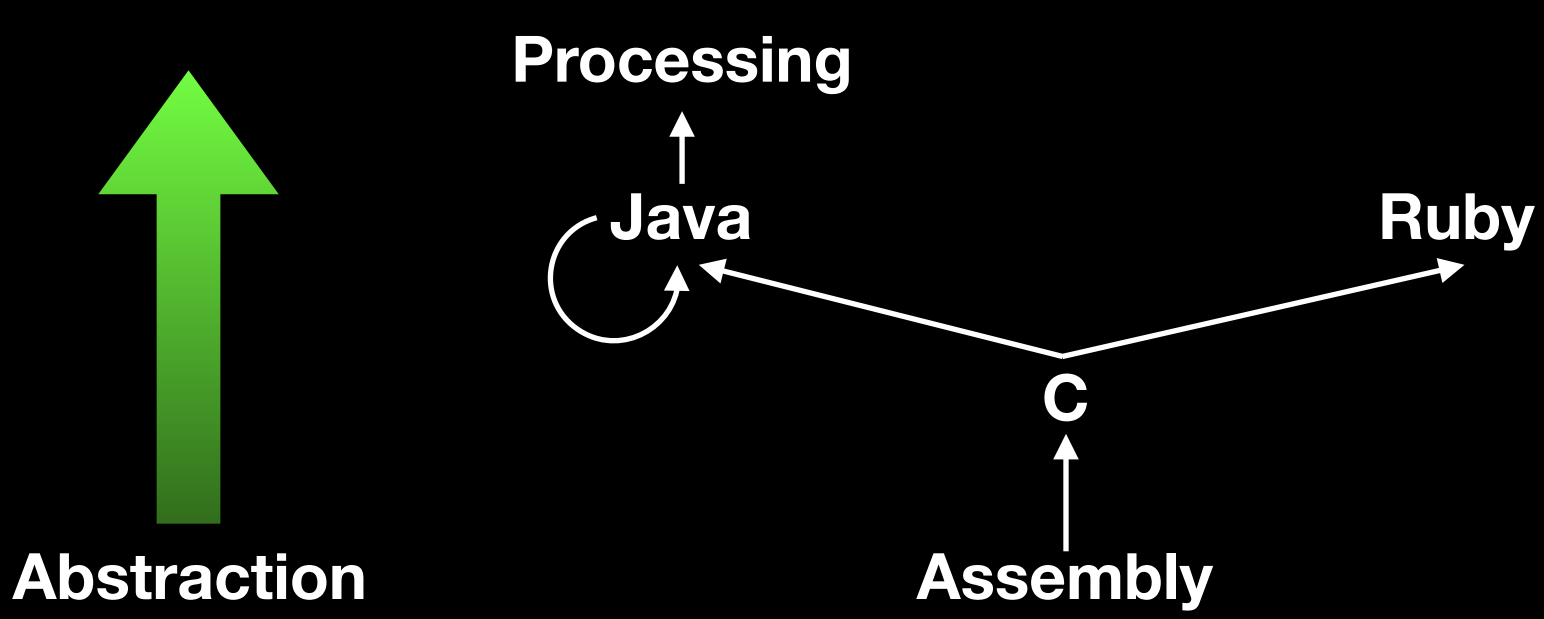

```
temp   = v[k];
v[k]  = v[k+1];
v[k+1] = temp;
```

```
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
1000 0000 1001 0101 1000 0000 1001
```

Bootstrapping

- But wait -- if we can use another program to compile our program, how was that program compiled?
  - Who compiles the compiler?
- The first compilers were written directly in binary.
- Bootstrapping means we use simpler languages to create increasingly complex and abstract languages.



What we're doing

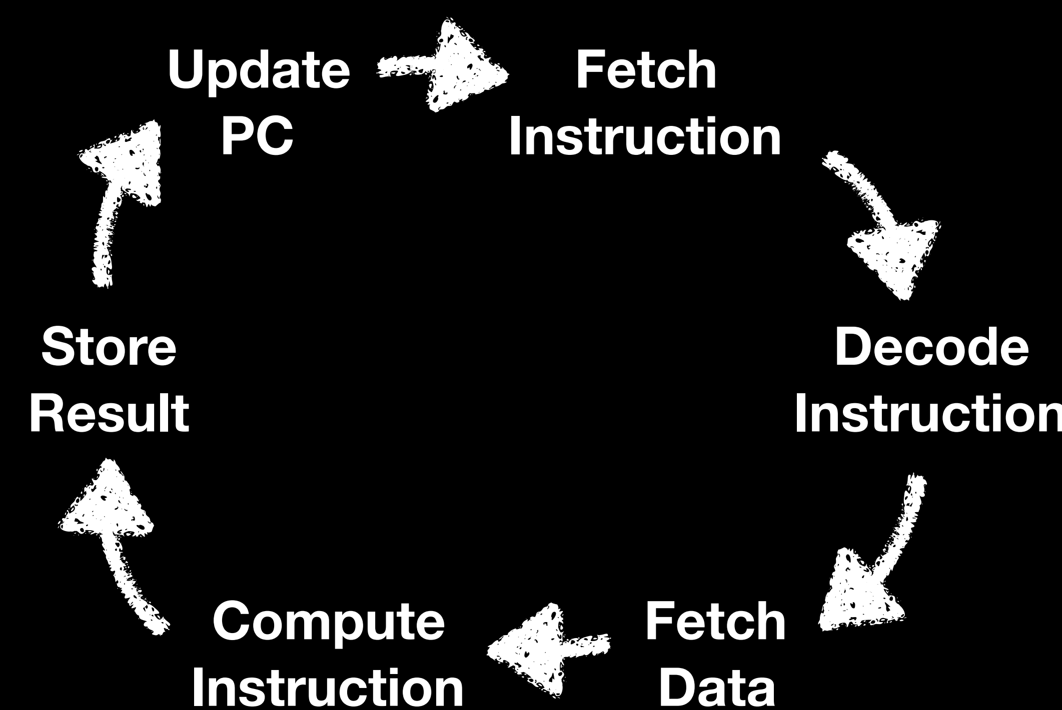
- What components are inside of a computer?
- How do we tell the hardware what to do?
- **How are our instructions executed?**

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

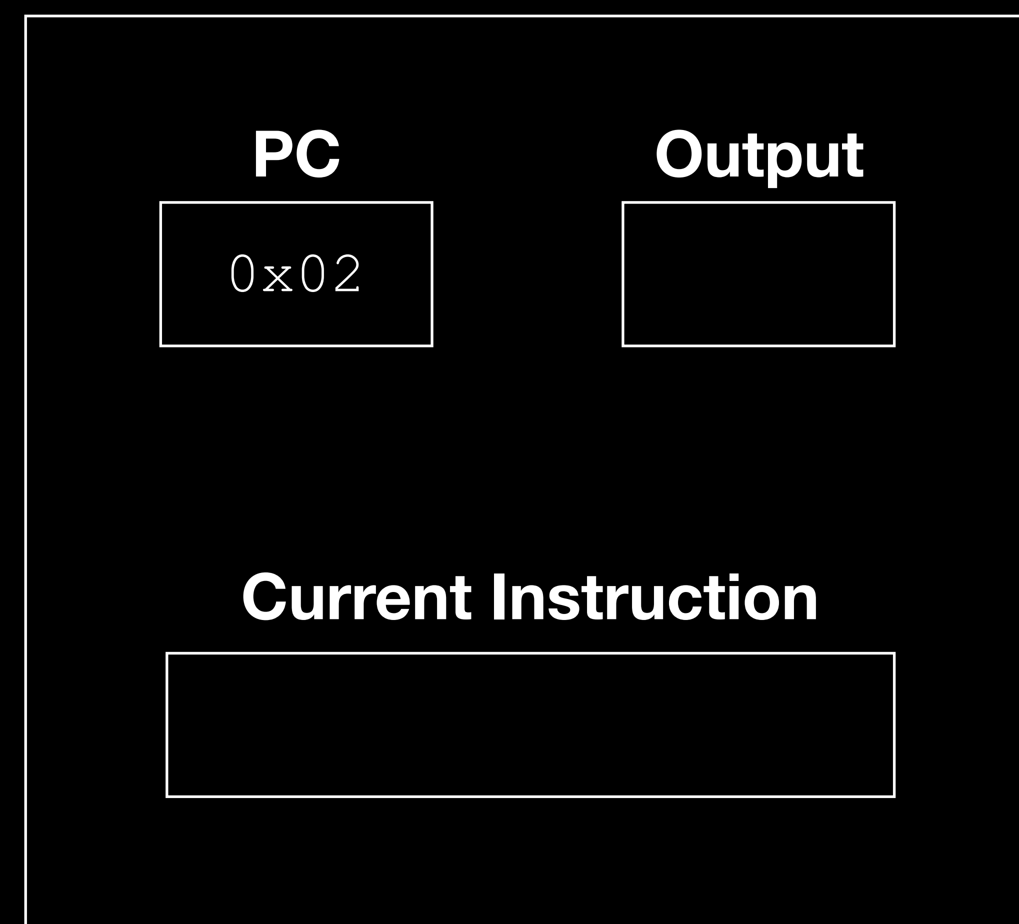


Fetch-Execute Cycle

Memory

Address	Value
`0x00`	`12`
`0x01`	`6`
`0x02`	`add 0x00, 0x01`
`0x03`	`store 0x1`

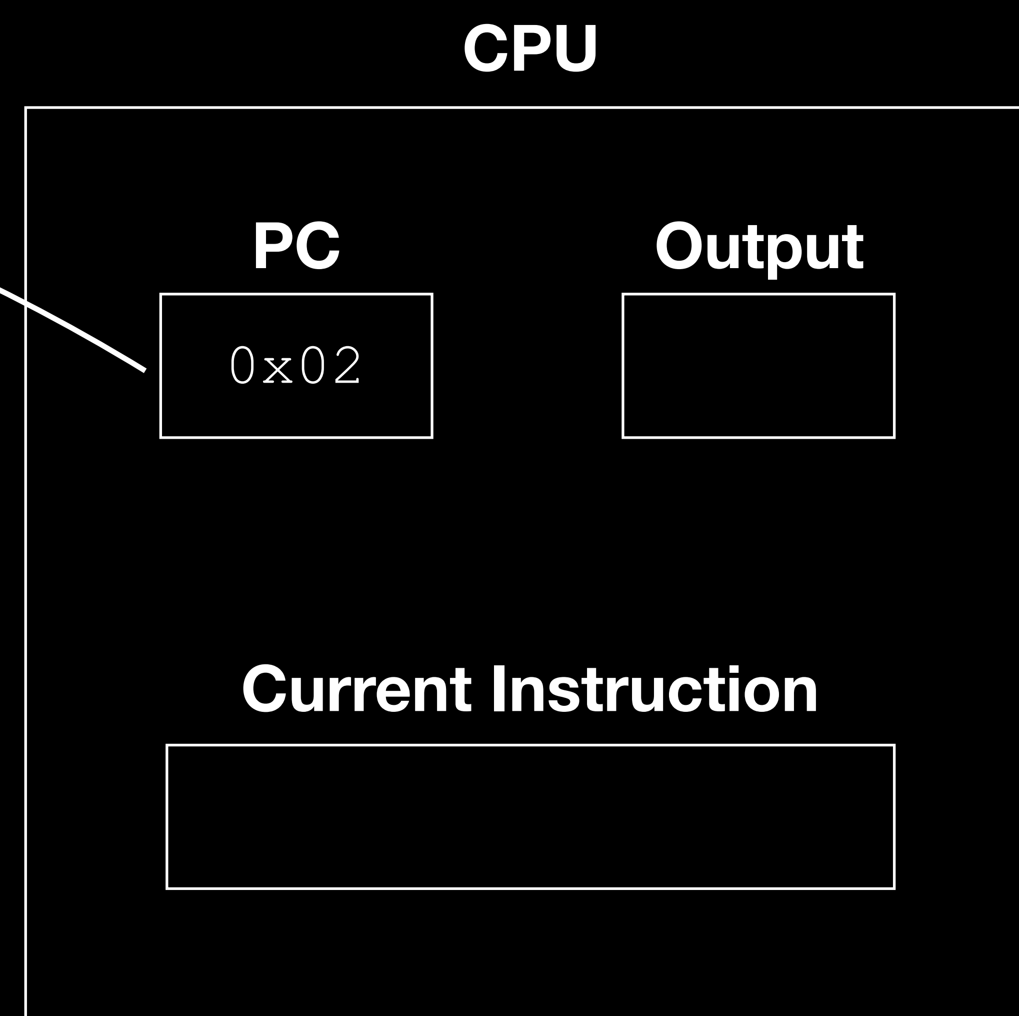
CPU



Fetch-Execute Cycle

Memory

Address	Value
`0x00`	`12`
`0x01`	`6`
`0x02`	`add 0x00, 0x01`
`0x03`	`store 0x1`

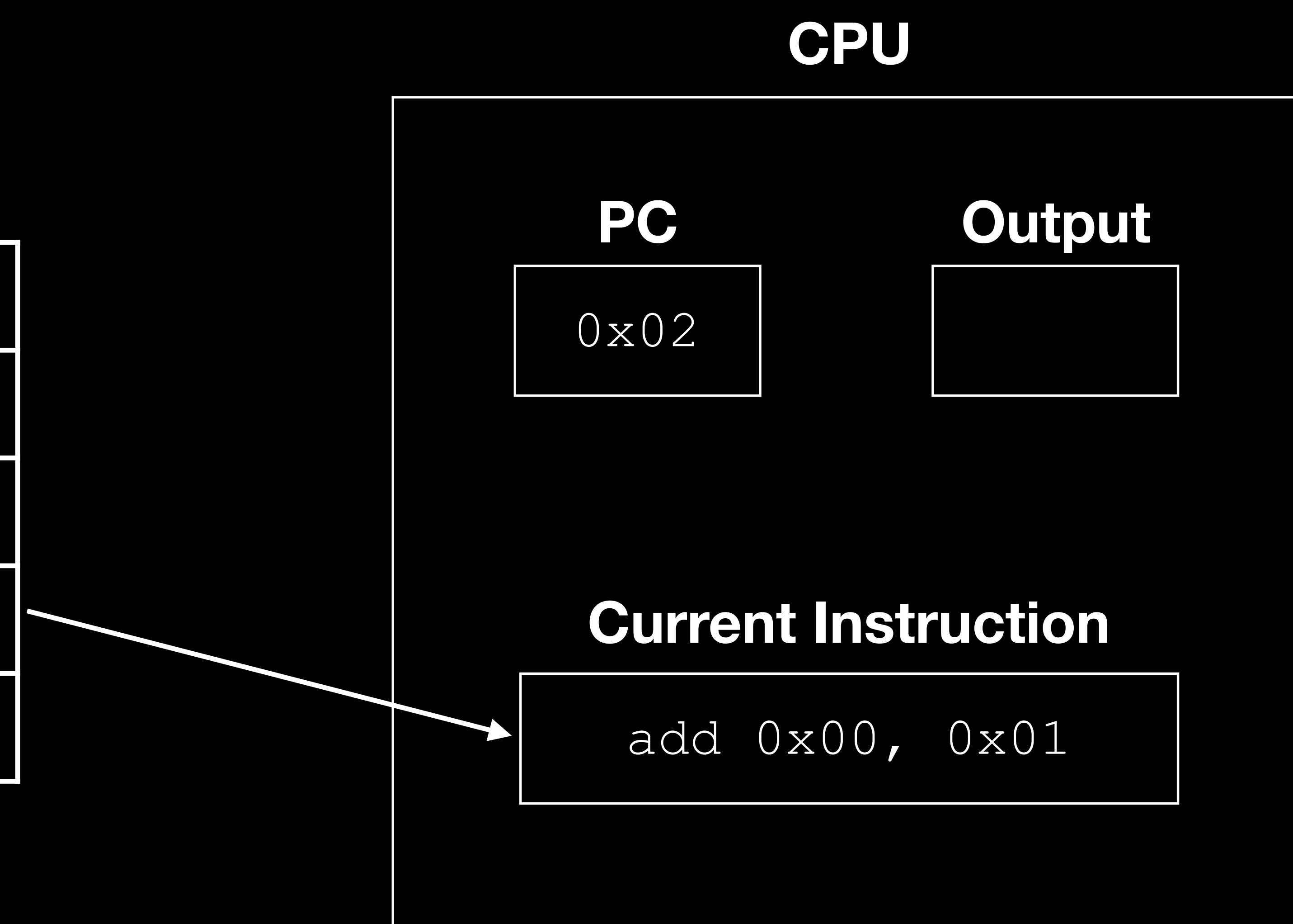


The Program Counter points to the address `0x02` in memory.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1

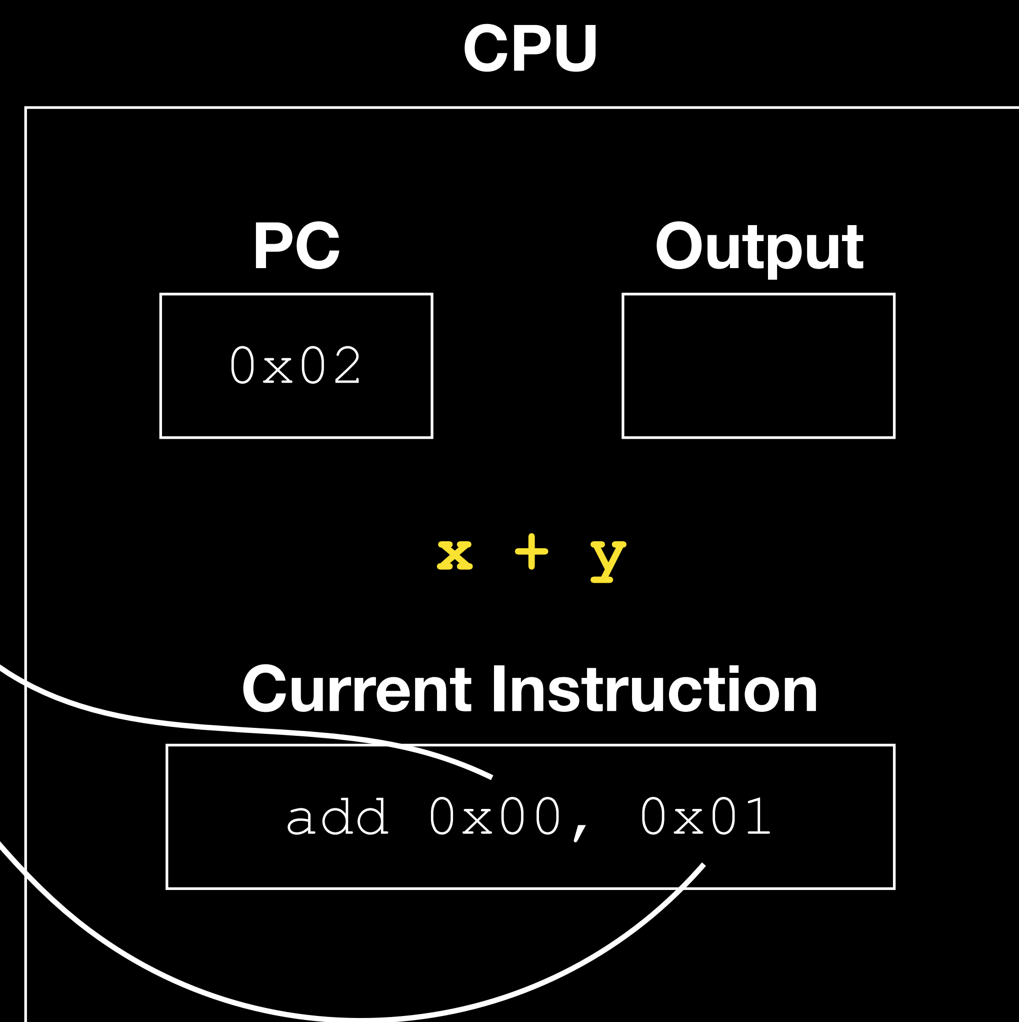


Fetch the instruction into the CPU.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1

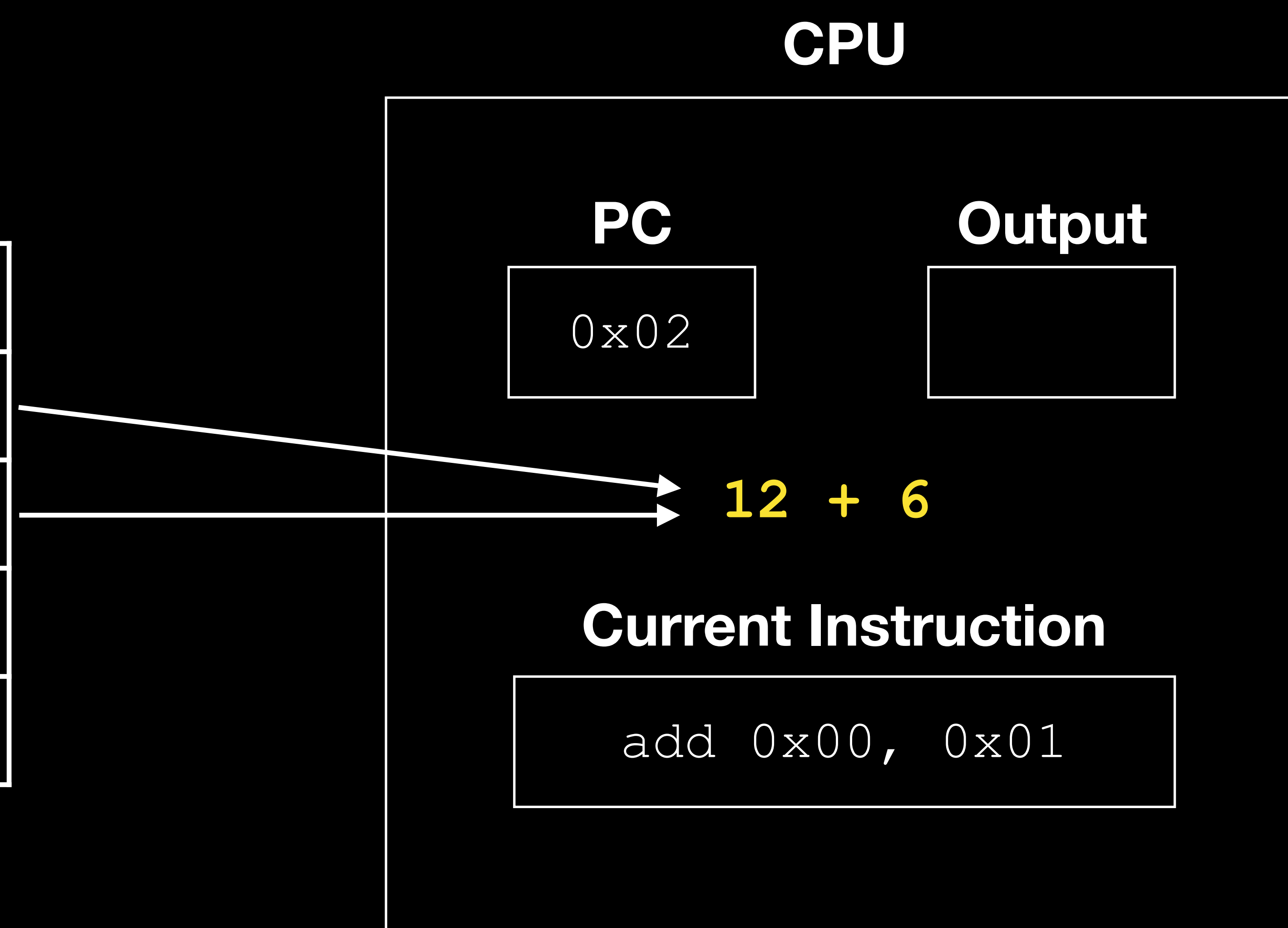


Decode what the instruction means.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1



Fetch the necessary data from memory.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1

CPU

PC

0x02

Output

12 + 6 = 18

Current Instruction

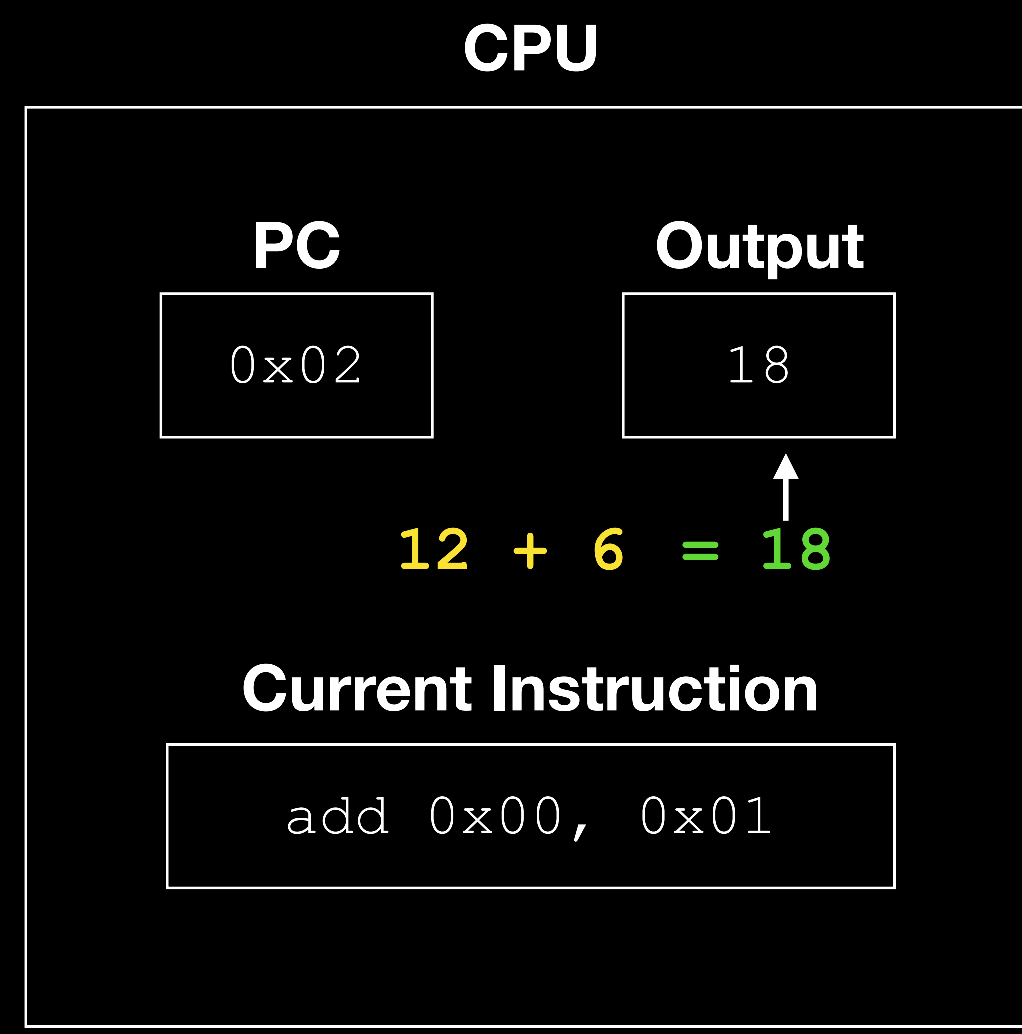
add 0x00, 0x01

Compute the result of the instruction.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1



And store it into temporary storage.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1

CPU

PC	Output
0x03	18

Current Instruction

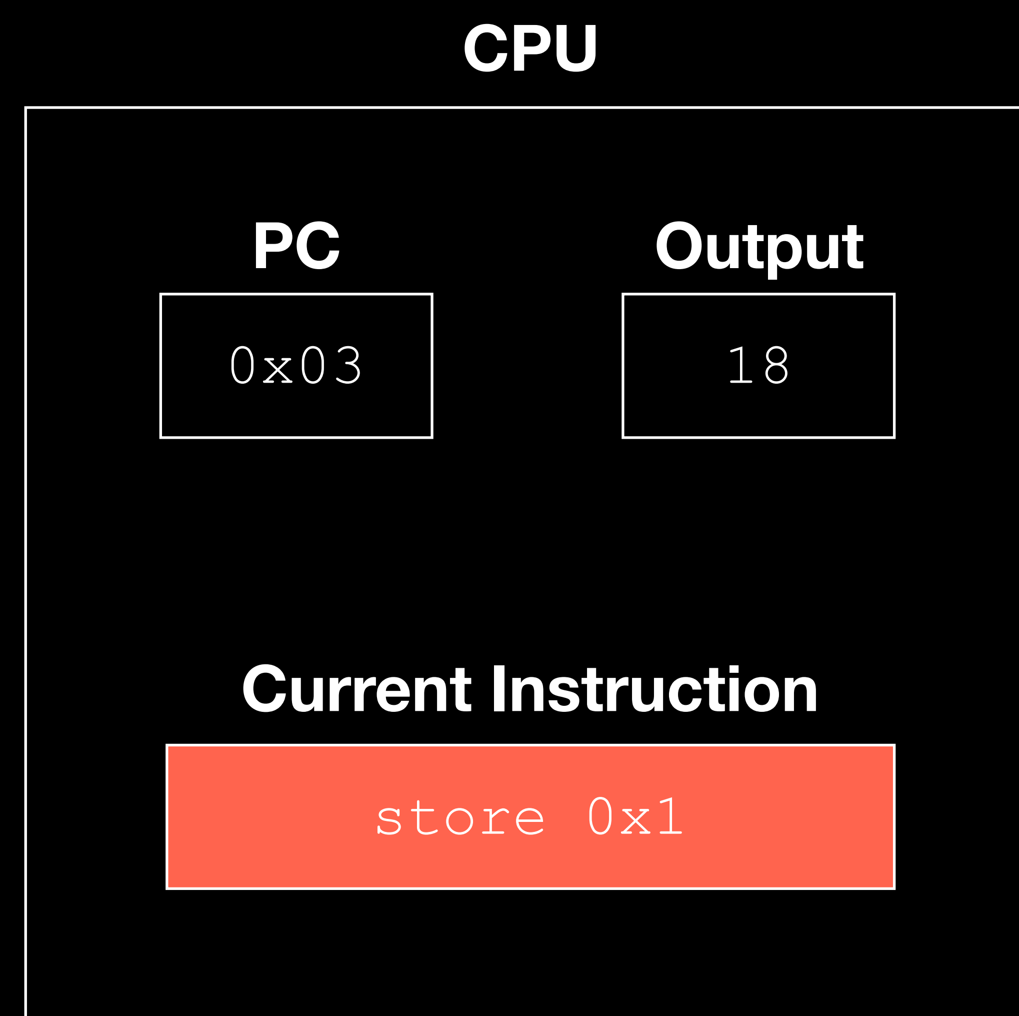
add 0x00, 0x01

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 0x1



Fetch the next instruction into the CPU.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1

CPU

PC

0x03

Output

18

Current Instruction

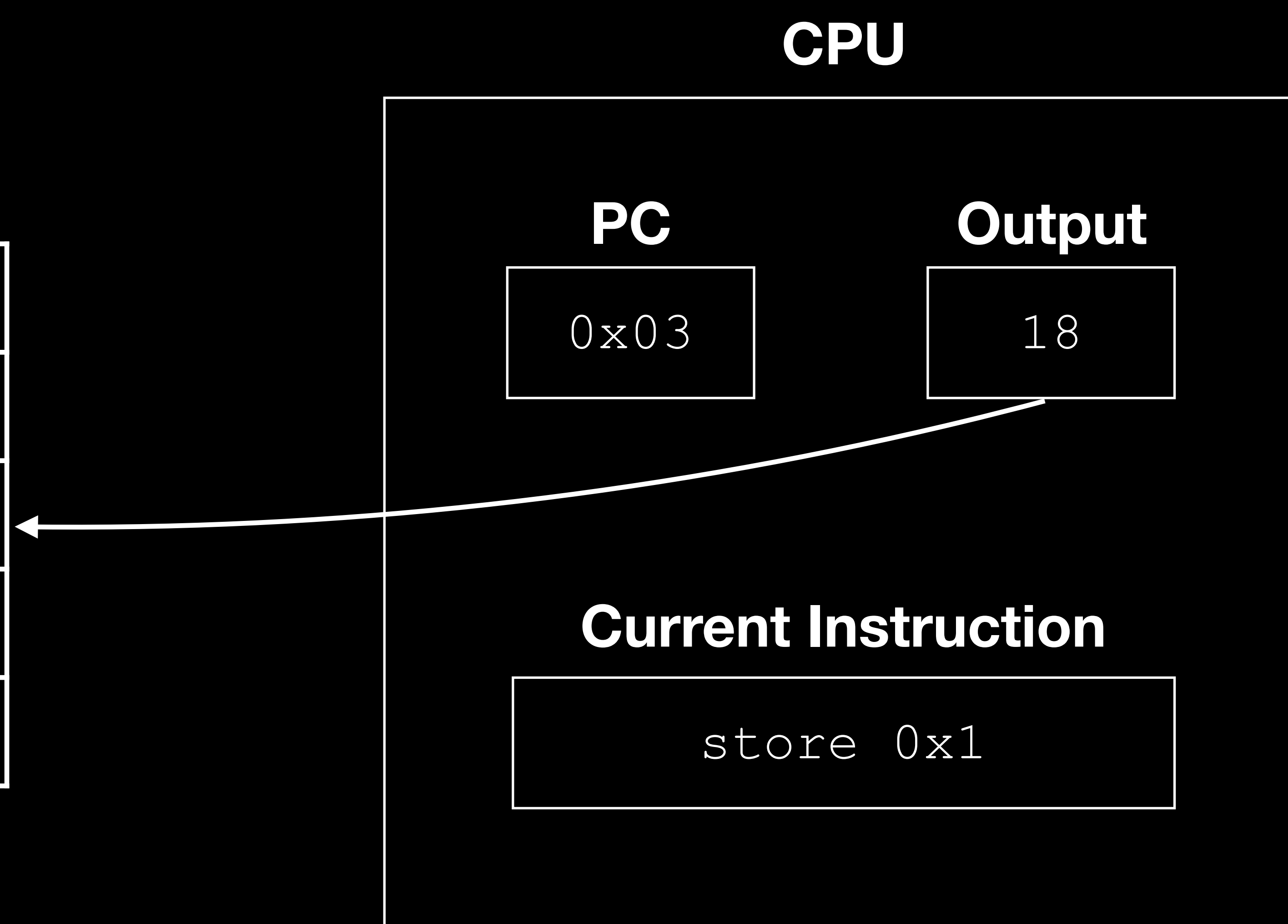
store 0x1

Decode the instruction: "store the output value into memory at `0x01`."

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	6
0x02	add 0x00, 0x01
0x03	store 0x1

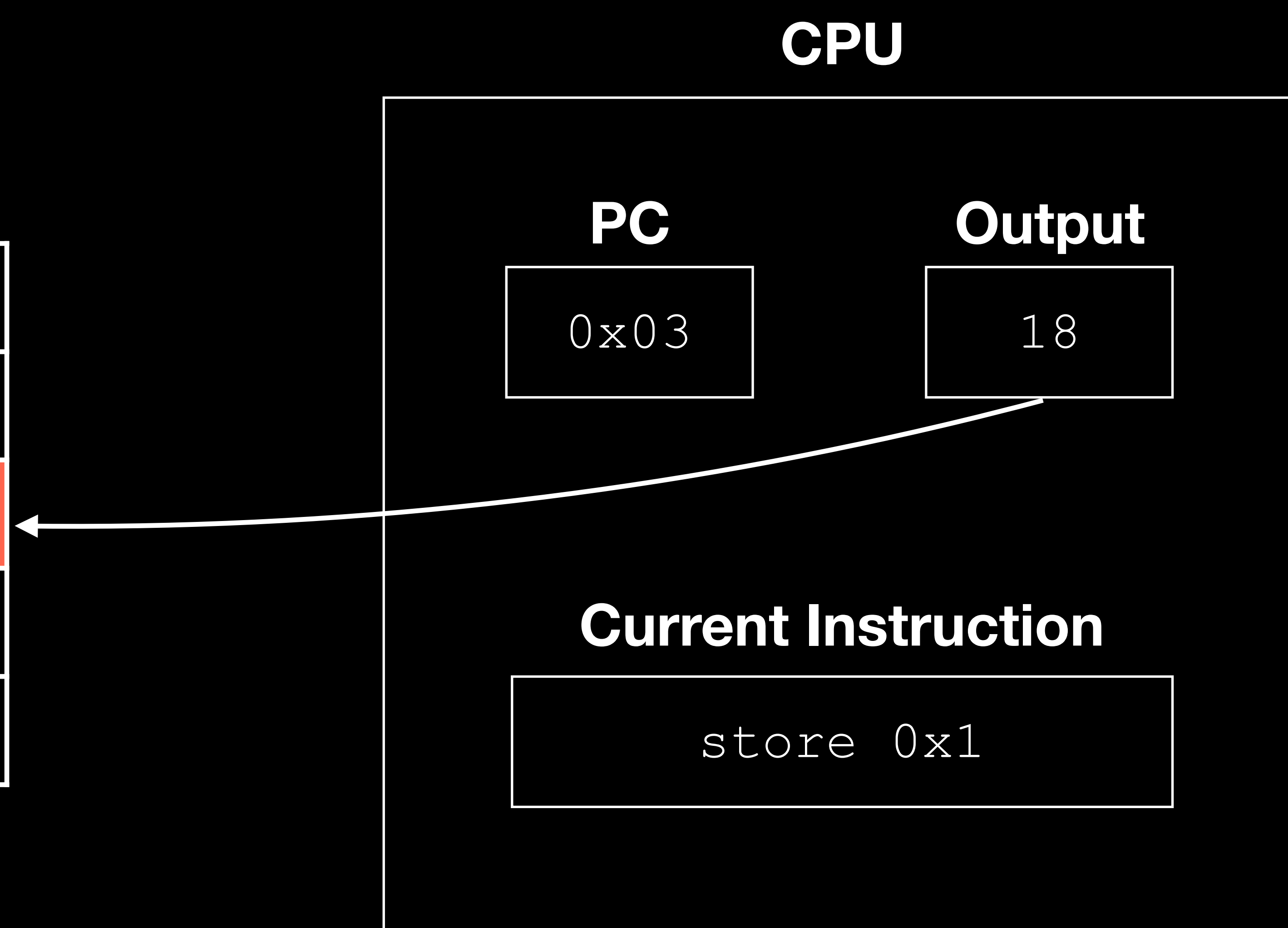


Execute the instruction.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	18
0x02	add 0x00, 0x01
0x03	store 0x1

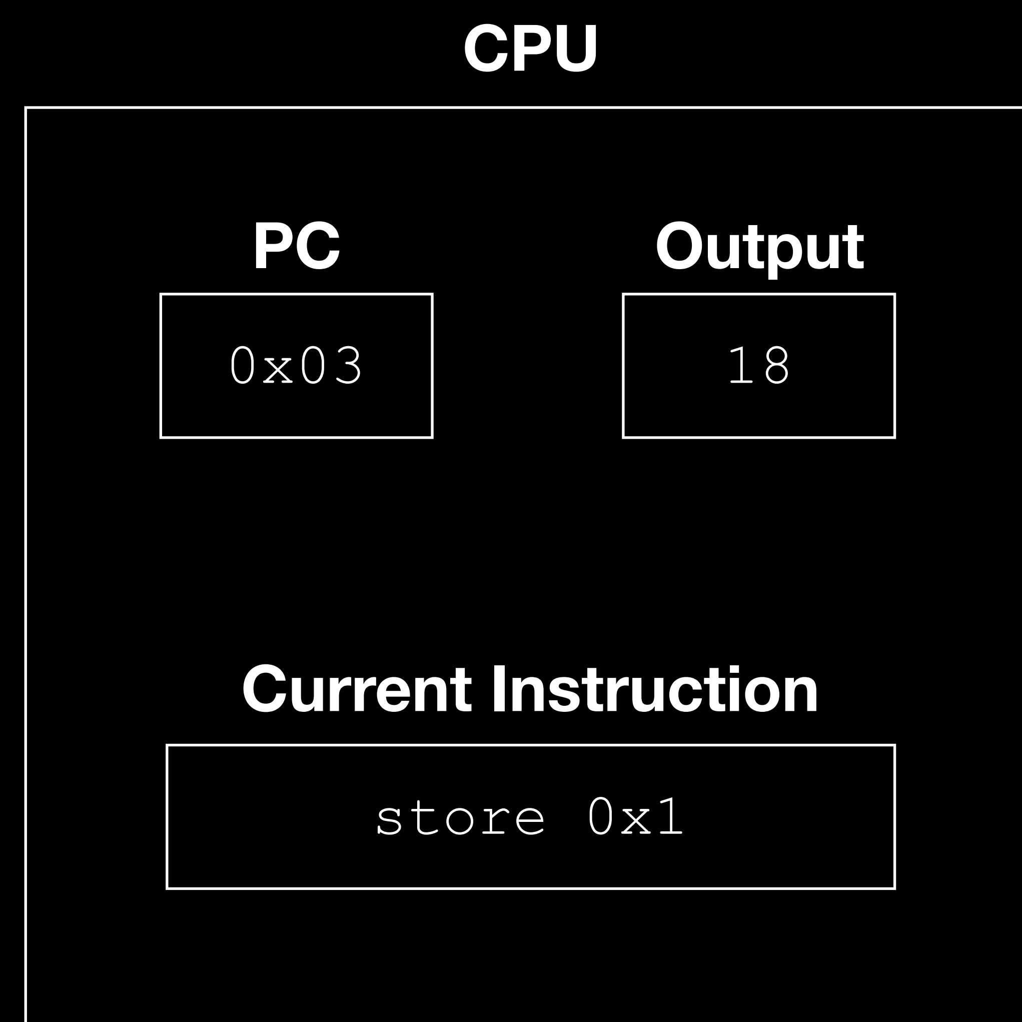


Execute the instruction.

Fetch-Execute Cycle

Memory

Address	Value
0x00	12
0x01	18
0x02	add 0x00, 0x01
0x03	store 0x1



And so on, and so forth...

Clock Rate

- The rate at which your CPU can perform the Fetch-Execute Cycle.
  - Must ensure that the clock speed is slow enough to accommodate the **slowest** instruction.
- Clock rate is usually given in Hertz. $1 \text{ hertz} = \frac{1 \text{ instruction}}{\text{second}}$
  - Example: $2 \text{ GHz} = 2 * 10^9 \text{ Hz}$

$$= 2 \text{ billion instructions per second}$$

- However, clock rate is often **not** a good indicator of speed
  - Modern CPUs spend lots of their time idle, waiting for data from memory, disk drives, networks, etc.

Example: Running a Processing Program

- The Processing environment compiles your code into machine language (`0`s and `1`s)
- 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.

Summary

- What components are inside of a computer?
  - Input, Output, Memory, CPU
- How do we tell the hardware what to do?
  - CPUs understand a limited set of instructions
  - We need to translate our abstract code (in Processing, Java, etc.) to code in the CPU's instruction set
- How are our instructions executed?
  - Program Counter keeps track of the current instruction
  - Instructions executed using Fetch-Execute Cycle
  - The CPU sometimes jumps around based on control flow.