ICDS Spring 2025

# Computer Architecture

## How a program being executed

# Recap: Data in the memory

- Objects (programs/data) are stored in the memory (e.g., in RAM cells)
- Memory cells have unique addresses
- Create a variable → associated it with an address that stores the object
    - immutable → the memory cell of the data is locked; modification is not allowed
    - mutable → the memory cell is unlocked; you may change the values in the cell

```
In [1]: a = 1

In [2]: id(a)
Out[2]: 4311873888

In [3]: a = 2

In [4]: id(a)
Out[4]: 4311873920
```

`a` is immutable; change its value virtually associates it with another cell.

- `id()`: the built-in function to show the memory address of the variable.

```
In [6]: a = [1, 2, 3, [4, 5, 6]]

In [7]: b = a[:]

In [8]: b[3][0] = "Four"

In [9]: a
Out[9]: [1, 2, 3, ['Four', 5, 6]]

In [10]: id(a[3])
Out[10]: 140511199555392

In [11]: id(b[3])
Out[11]: 140511199555392
```

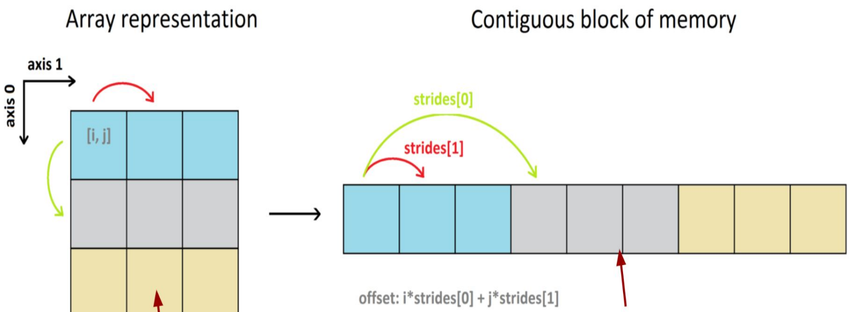
A variable (e.g, list) is a container of addresses

- `a[3]` is the address of the list `[4, 5, 6]`.

Shallow copy a only makes a copy the elements in `a` which are addresses, so, `b[3]` and `a[3]`refer to the same list. → change elements in `b[3]` will change elements in `a[3]`.

# Recap: Locality

- Computers "guess" what data will be used in the next step and load them into cache.
- They do it by following two principles of locality. (temporal/spatial locality)




This is how the array looks like in the view of Python users.

This is how the array being stored in the computer memory.

<u>So, indexing the elements along the rows has better spatial locality.</u>

```python
import numpy as np

##Create an numpy array
a = np.array([5, 4, 3, 2, 1])

# do something with the 3rd element;
# Which element(s) will be loaded into
cache together with a[2]?
print(a[2])
```

Q: Which elements are **most likely** to be loaded into cache together with a[2]?

# Exercise: Locality

Q: Which elements are **most likely** to be loaded into cache together with a[2]?

A. *a[0]*
B. *a[1]*
C. *1*
D. *2*

```python
import numpy as np

##Create an numpy array
a = np.array([5, 4, 3, 2, 1])

# do something with the 3rd element;
# Which element(s) will be loaded into
cache together with a[2]?
print(a[2])
```

Scan to answer!

# Agenda

- Computer architecture
- Cycle of program execution
  - Vole: a simple computer
  - Machine language
  - Running codes in high/low-level programming languages
- Appendix

# Computer architecture

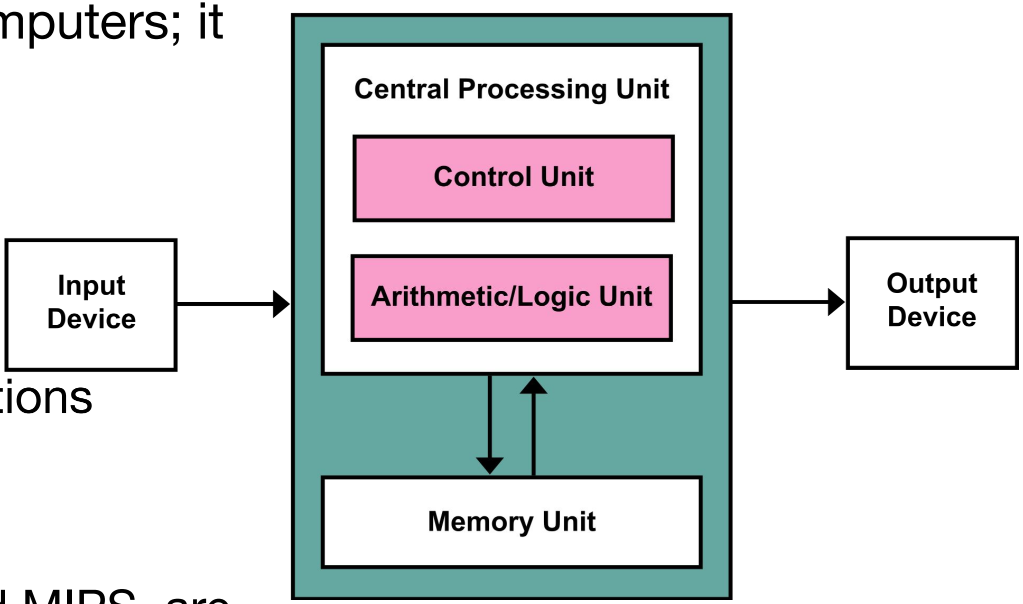
It is a blueprint for building a computer, it tells

- what circuit blocks (primitive components) the machine needs, and
- how these components can work together
    - e.g., how to coordinate the CPU and the memory

# von Neumann architecture

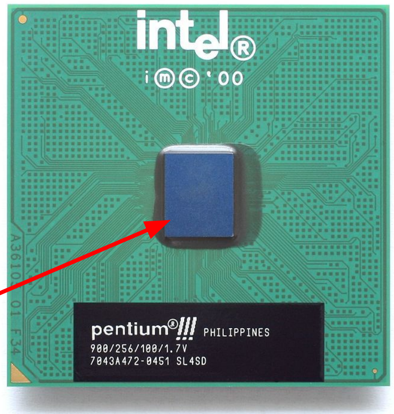
- The mainstream of modern computers; it has,
- A central processor unit
  - Control unit
  - ALU
  - Registers (memory in CPU)
- Memory stores data and instructions
- Input and output devices

Modern processors, such as x86 and MIPS, are examples of the von Neumann architecture.

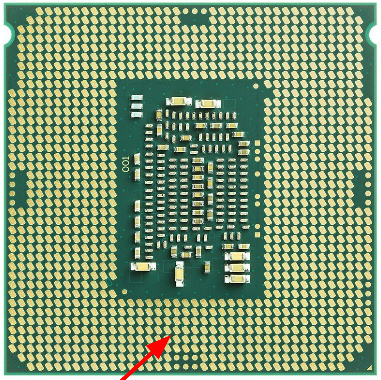



The schematic is from from [Wikipedia](Wikipedia).

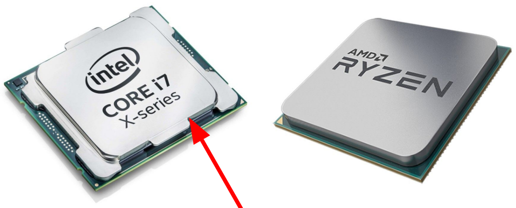
# Modern CPU




Silicon
Die



Pins that connects the
CPU to the motherboard



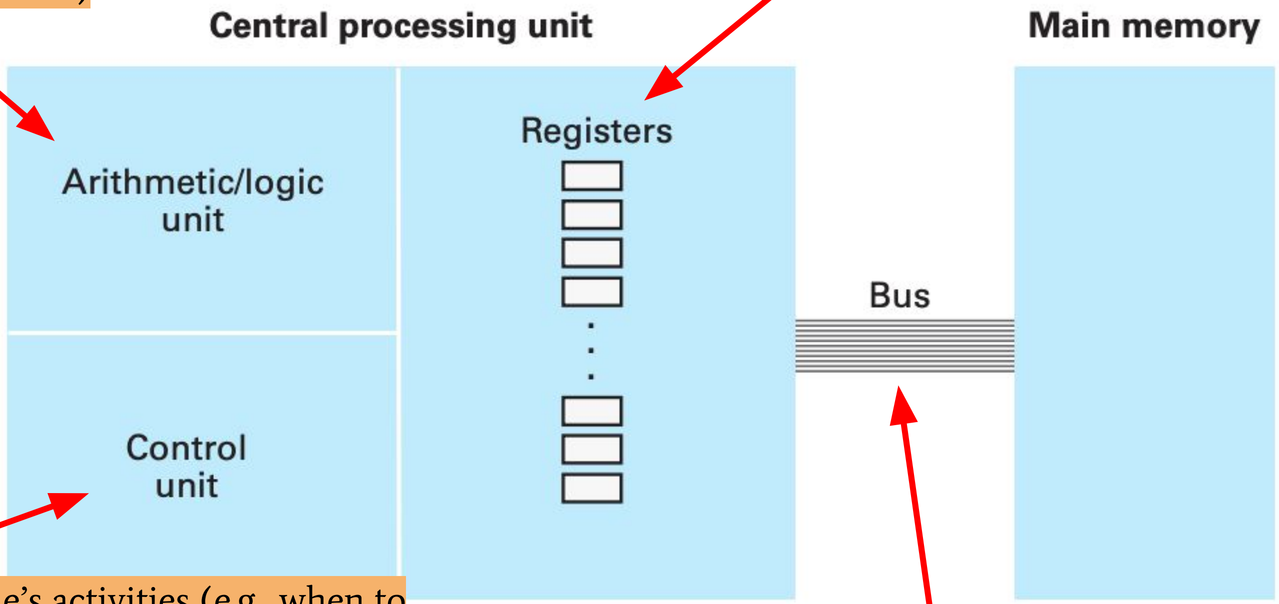

The lid over the silicon die is called
IHS (integrated heater spreader)

How a cpu made.

# The heart of a computer

temporarily storing data/instructions

Processing data (e.g., + or -)



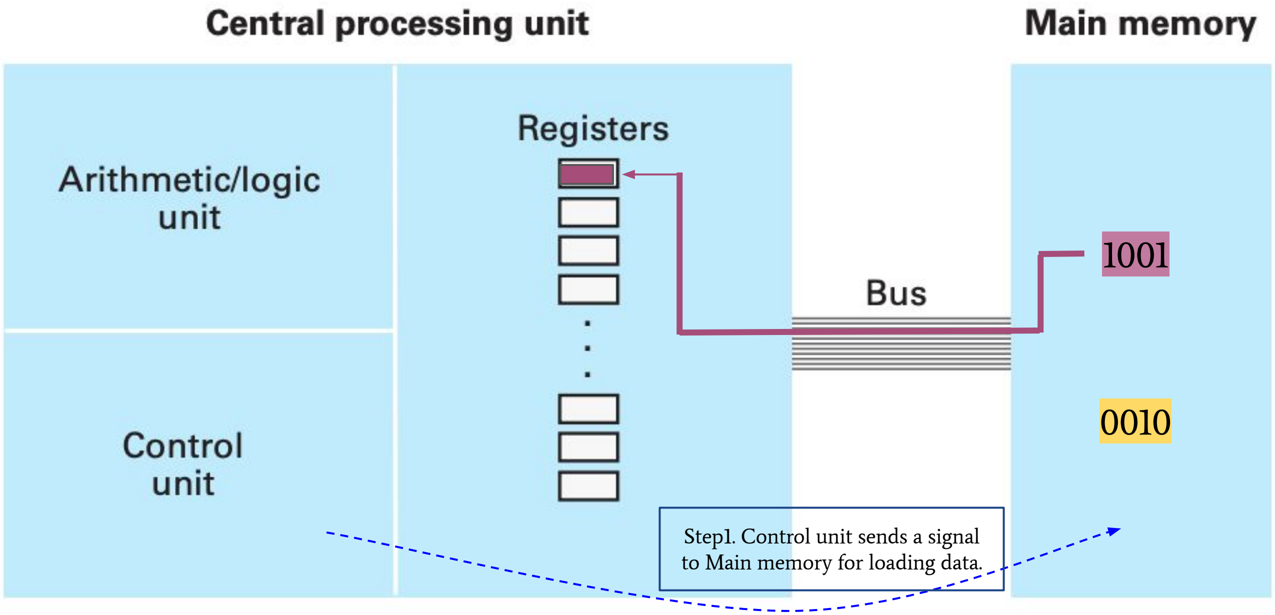

Managing the machine's activities (e.g., when to load/write/, how to process)

transferring data between CPU and Main memory

# How a CPU adds two numbers

To add two values (e.g., $9_{10}$, and $2_{10}$) stored in Main memory.

Step 1. Load 1001 from Main memory into a register

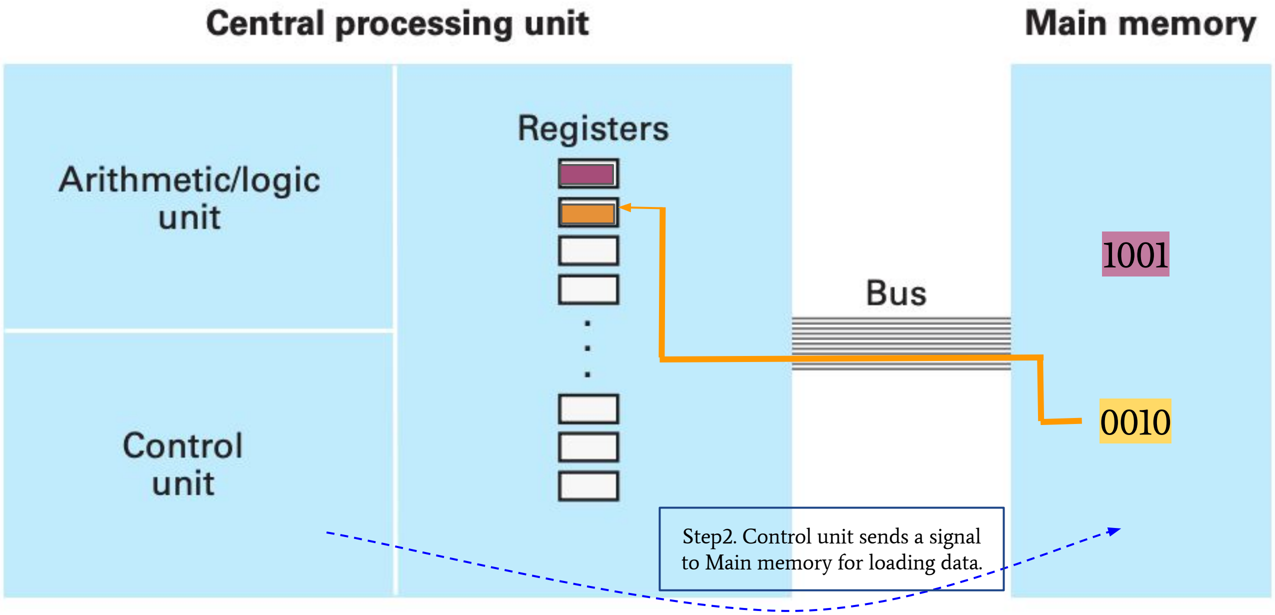


# How CPU add two numbers

To add two values (e.g., $9_{10}$, and $2_{10}$) stored in Main memory.

Step 1. Load 1001 from Main memory into a register

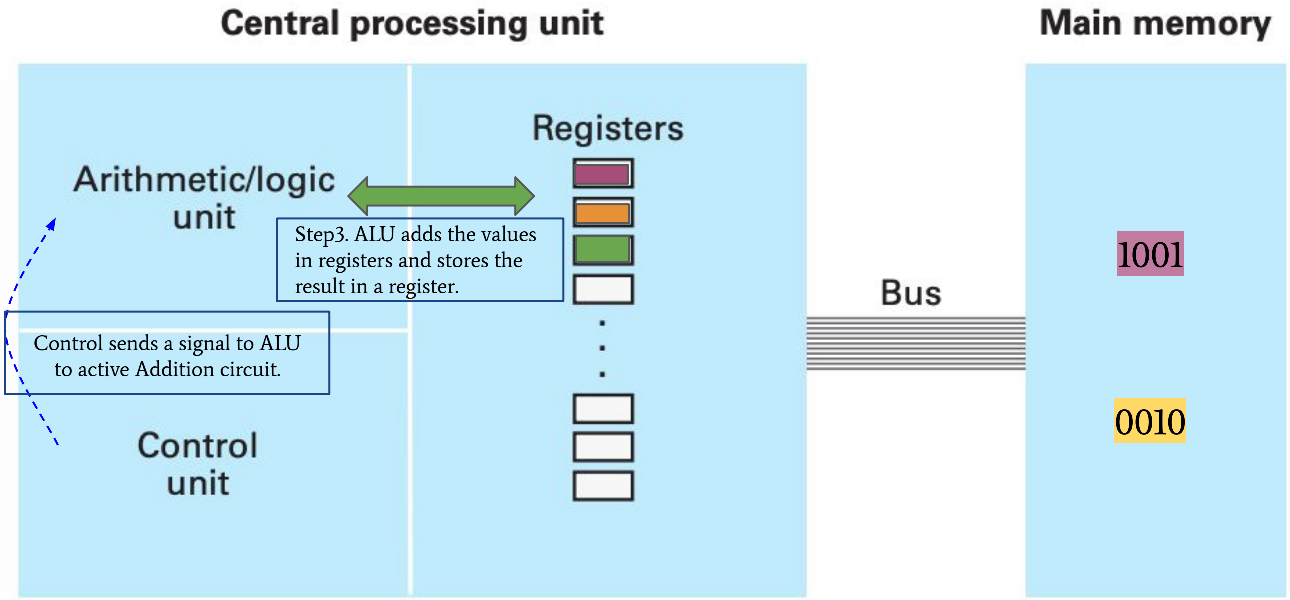
Step 2. Load 0010 from Main memory into another register



# How CPU add two numbers

To add two binary values (e.g., $9_{10}$, and $2_{10}$) stored in Main memory.

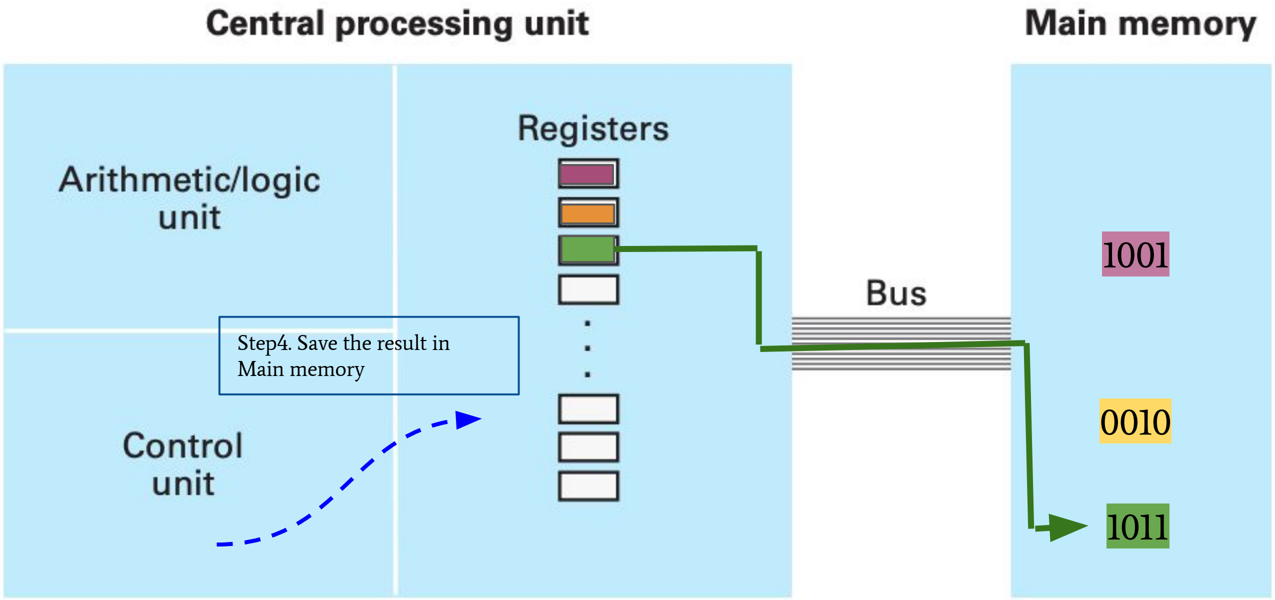
Step 1. Load two values from Main memory into two  registers
Step 2. Load 0010 from Main memory into another register
Step 3. ALU operates on the registers; the result is temporary stored in a register.



# How CPU add two numbers

To add two binary values (e.g., 1001, 0010) stored in Main memory.

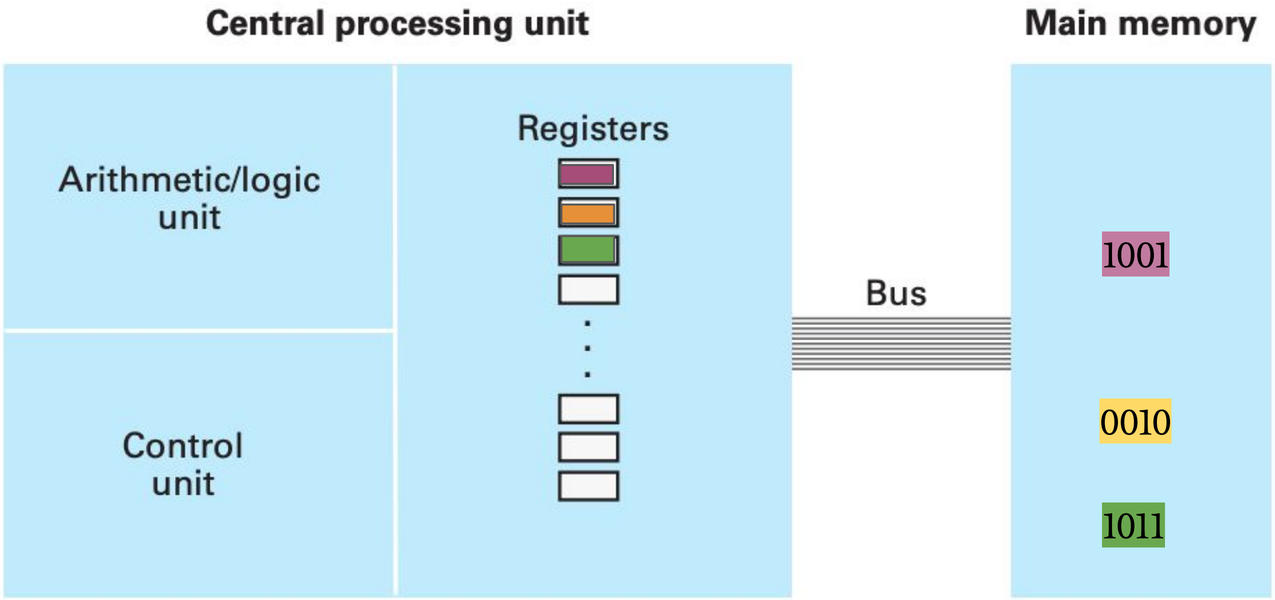
Step 1. Load two values from Main memory into two  registers
Step 2. Load 0010 from Main memory into a register
Step 3. ALU operates on the registers; the result is temporary stored in a register
Step 4. Store the result in Main memory



# How CPU add two numbers

To add two binary values (e.g., 1001, 0010) stored in Main memory.

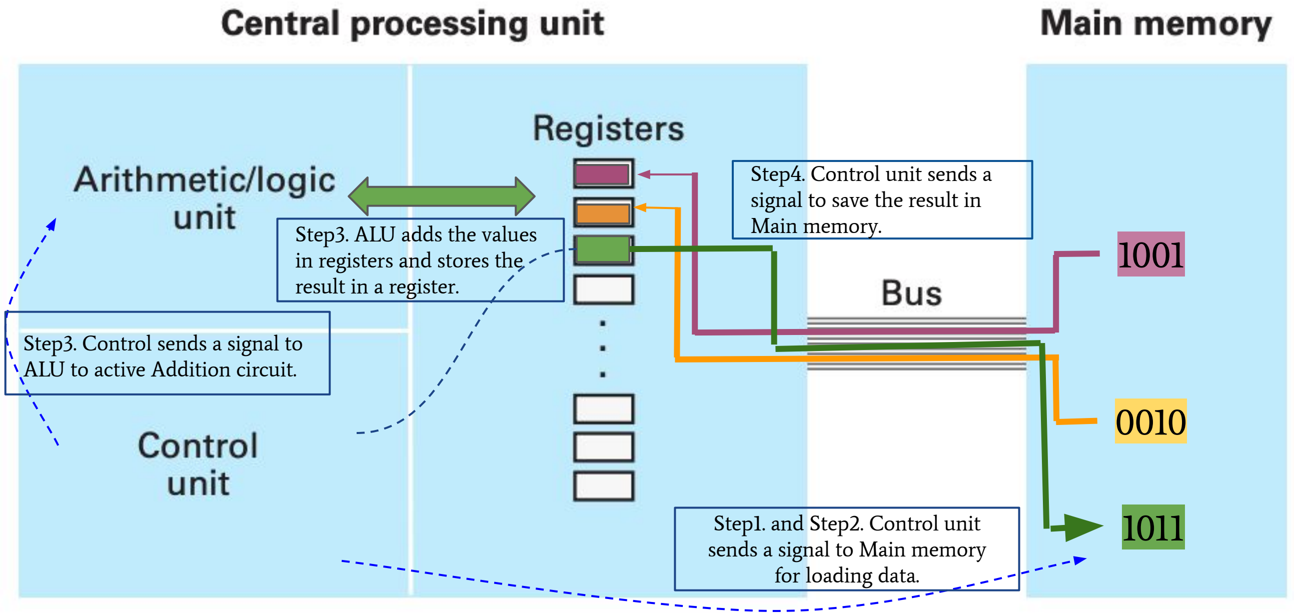
Step 1. Load two values from Main memory into two  registers
Step 2. Active the addition circuitry in ALU
Step 3. ALU operates on the registers; the result is temporary stored in a register
Step 4. Store the result in Main memory
Step 5. Stop



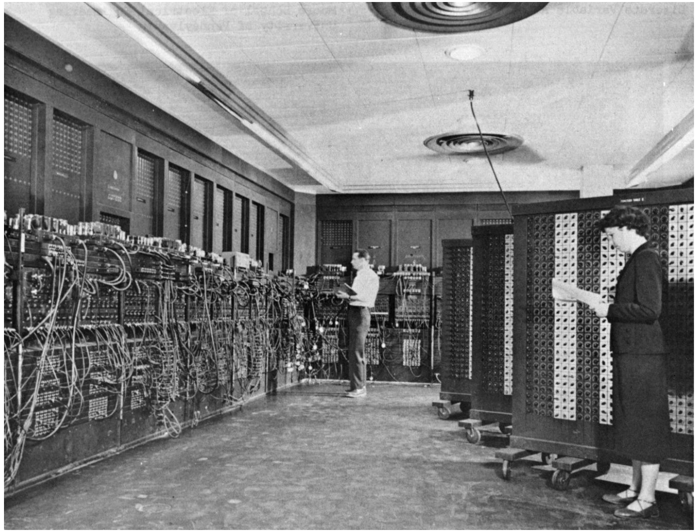
# How CPU add two numbers (overview)

Adding two binary values (e.g., 1001, 0010) stored in Main memory.

1. Load 1001 from Main memory into a register
2. Load 0010 from Main memory into a register
3. ALU operates on the registers; the result is temporary stored in a register
4. Store the result in Main memory
5. Stop



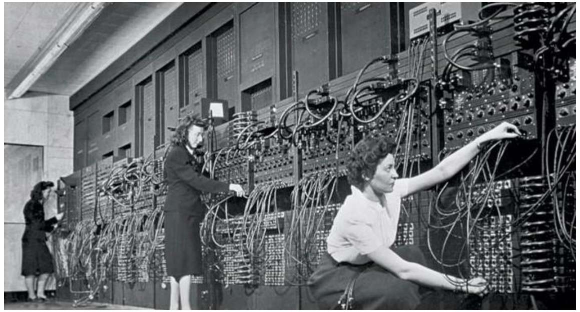
# Fixed-program computers (like electricity calculators)



The figure is ENIAC, the first electronic general-purpose computer. People had to change the connection among circuit blocks to reprogram it.

In the early days, human played the role of control unit.

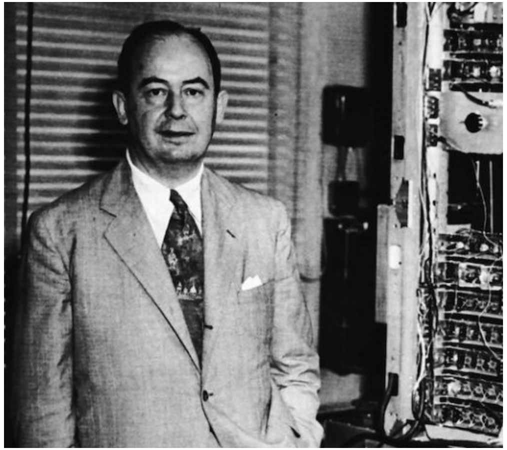
- To program was to reconnect the circuit blocks for different tasks.
- Reprogramming took days



# Stored-program computers

Instructions can be stored in main memory, like data!

- They can be represented as binary strings, so, having the same storage format as data

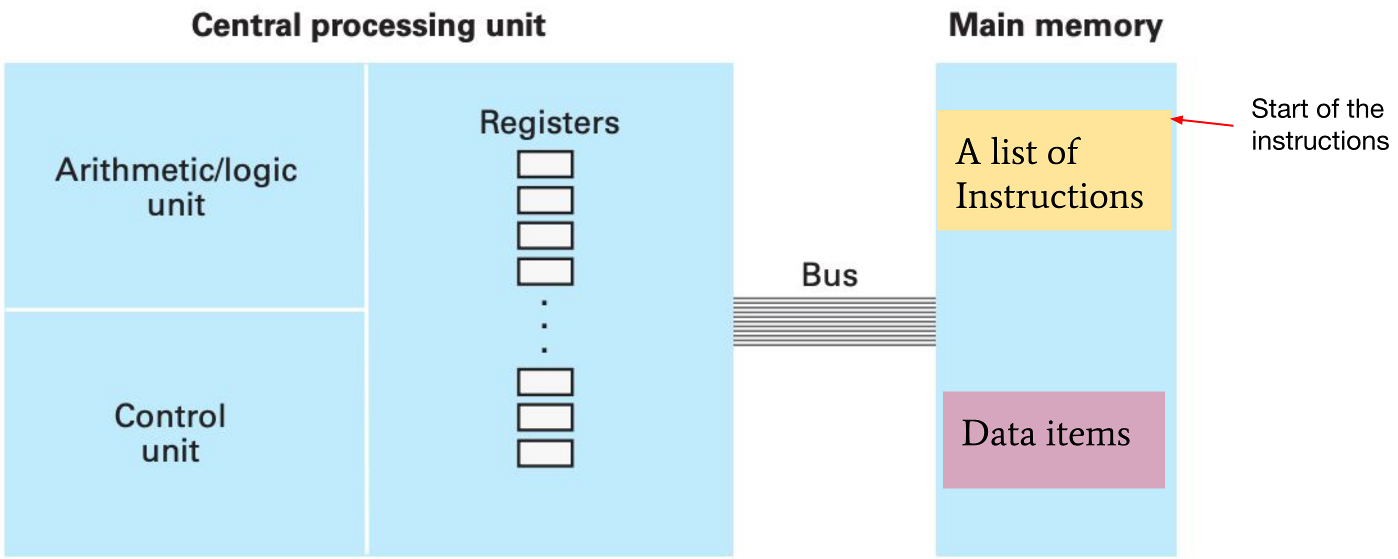


John von Neumann,
1903-1957

Almost all modern computers are stored-program computers.

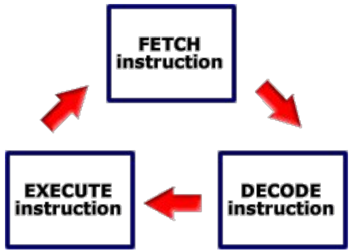
- They store data and instructions together in main memory.
- The stored-program concept was introduced by von Neumann

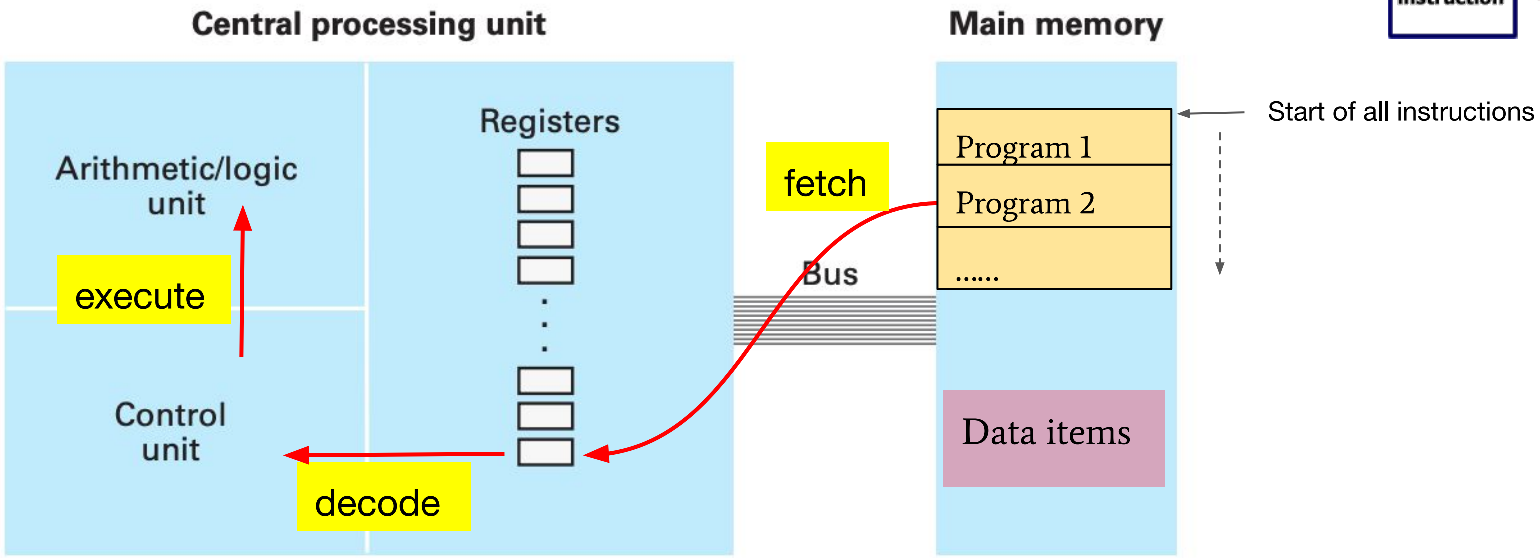
# Stored-program computers




- Instructions and data are both stored in the main memory.
- Instructions are listed in a list (like an address book)
- CPU loads one instruction at a time from the list and run it ⇒ it repeats this action until all instructions are done

# The cycle of execution




By default, the CPU will fetch instructions from the start of the list, following a top-down order. Of course, this is not enough.

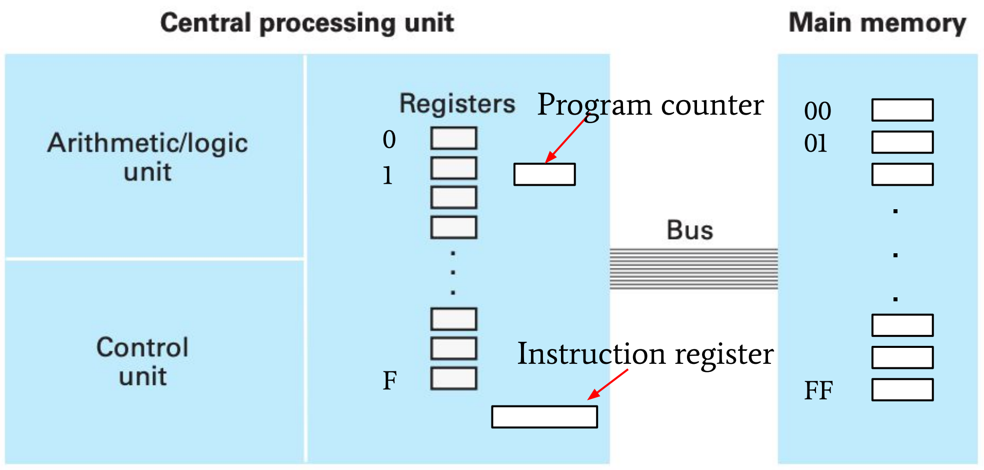
- How can we control the order of execution?
- We often use two special registers to solve it.

# Vole: a simplified computer

Now, let's look at a very simple computer, namely, Vole.

Vole's Architecture

- 16 8-bit general-purpose registers
- 256 main memory 8-bit cells
- one instruction register (16-bit)
- one program counter (8-bit)




Other specifications:

- The registers are labeled with 0 through F (hexadecimal, i.e., 0-15 in decimal)
- The address of the memory cell are from 00 to FF (i.e., 0-255 in decimal)
- a machine instruction of Vole has 16 bits

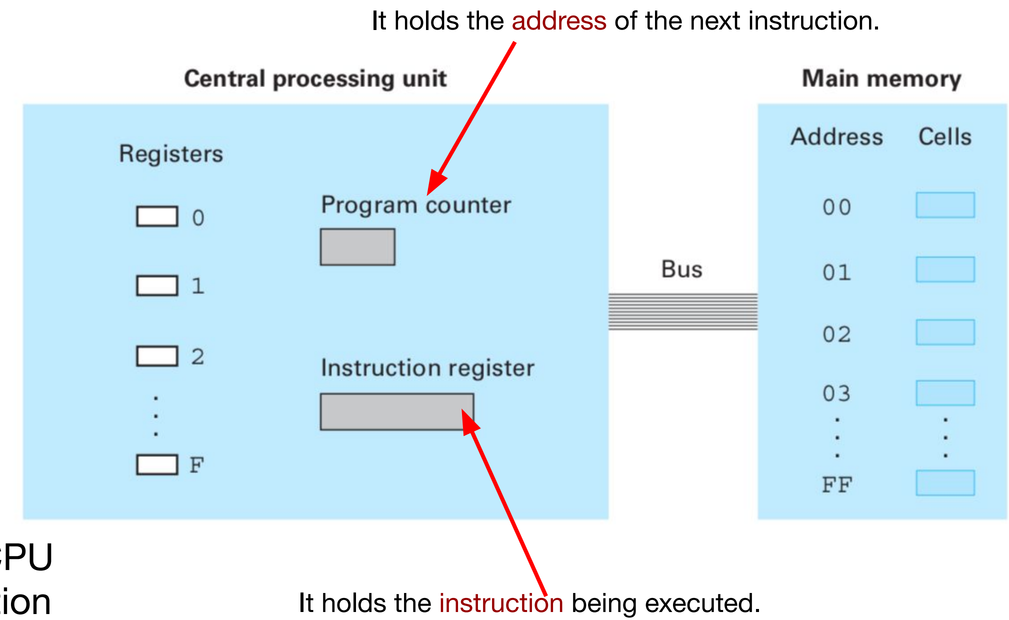
# Two special-purpose registers in the CPU

- Program counter (**PC**): holding the **address** of the next instruction

- Instruction register (**IR**): holding the **instruction** being executed

During the fetch step, CPU loads the instruction from the address in PC, then, increments the address (recall the address is a binary) so that it becomes the address of the next instruction.

The instruction is loaded into the IR where the CPU decodes the instruction (associating the instruction to some concrete operations on circuits).



# Machine language

In stored-program computers, instructions are represented in binary strings.

- Machine language is a set of binary patterns that represent instructions

# Instructions

Machine instructions should follow some **pre-defined format** so that CPUs can “understand” them.

- A machine instruction contains two parts: op-code and operand
  - The op-code: the operation to take. (e.g., load/store)
  - The operand: the information needed by the op-code. (e.g., where to load/store)

| Op-code | Operand | | |
|---|---|---|---|
| 0011 | 0101 | 1010 | 0111 |

An example of a 16-bit instruction.

- Different designs of processors may have different instruction formats.
  - Machine language of a PC is different to that of an Apple M1. (PC uses x86; M1 uses ARM64)

# Two philosophies of machine languages

- RISC (reduced instructions set computer)
  - CPUs are designed to execute a minimal set of instructions, e.g., it may have A + B but no A x B.
  - Running fast and less expensive
  - Apple silicon M1 is RISC
- CISC (complex instructions set computer)
  - CPUs can execute many complex, redundant instructions
    - e.g., it has A + B and A x B.
  - Convenient for programming
  - x86 is CISC

By the way, if you would like to have a taste of a compact programming language, you may have a look at this. (Don't be astonished by its name, it reproduces the pain for programming with RICS machine languages ).

# The instruction repertoire

Machine's instructions can be categorized into 3 groups:

- Data transfer
  - Instructions for copying/moving data from one location to another
  - e.g., load, store, I/O instructions
- Arithmetic/Logic
  - Instructions for activities of ALU.
  - e.g, +, -, Boolean operations (AND, OR, XOR)
- Control
  - Instructions for directing the execution of the program
  - e.g., halt, jump

# Machine language of Vole

We set, from left to right,

- The 1st hexadecimal digits (4 bits) represents the op-code
- The next 3 hexadecimal digits (12 bits) are operands

The format of a Vole's instruction

| Op-code | Operand | | |
|---|---|---|---|
| 0011 | 0101 | 1010 | 0111 |

Representing binaries with hexadecimals

| Op-code | Operand | | |
|---|---|---|---|
| 0x3 | 0x5 | 0xA | 0x7 |

We often use hexadecimal notations which is more concise than binaries.
Note: The "0x" prefix indicates the number is hexadecimal.

# Machine language of Vole

| Op-code | Operand | Description | Example |
|---|---|---|---|
| 1 | RXY | LOAD the register R with the bit pattern found in the memory cell whose address is XY | 14A3 would cause the contents of the memory cell located at address A3 to be placed in register 4. |
| 2 | RXY | LOAD the register R with the bit pattern XY | 20A3 would cause the value A3 to be placed in register 0. |
| 3 | RXY | STORE the bit pattern found in register R in the memory cell whose address is XY | 35B1 would cause the contents of register 5 to be placed in the memory cell whose address is B1 |
| 4 | 0RS | MOVE the bit pattern found in Register R to Register S | 40A4 would cause the contents of register A to be copied into register 4. |
| 5 | RST | ADD the bit patterns in registers S and T as though they were two's complement representations and leave the result in register R. | 5726 would cause the binary values in register 2 and 6 to be added and the sum placed in register 7. |
| 6 | RST | ADD the bit patterns in registers S and T as though they were two's floating-point notations and leave the result in register R. | 634E would cause the values in register 4 and E to be added as floating-point values and the result to be placed in register 3. |

R, S, and T:
3 different registers

X, and Y:
Hexadecimal digits

| 3 | 5 | B | 1 |
|---|---|---|---|
| STORE | register 5 | Memory address B1 | |

To save contents in register 5 to B1 of the main memory

# Machine language of Vole (continued)

| Op-code | Operand | Description | Example |
| --- | --- | --- | --- |
| 7 | RST | OR the bit patterns in register S and T and place the result in register R. | 7CB4 would cause the result of ORing the contents of registers B and 4 to be placed in register C. |
| 8 | RST | AND the bit patterns in register S and T and place the result in register R. | 8045 would cause the result of ANDing the contents of registers 4 and 5 to be placed in register 0. |
| 9 | RST | XOR the bit patterns in register S and T and place the result in register R. | 95F3 would cause the result of XORing the contents of registers F and 3 to be placed in register 5. |
| A | R0X | ROTATE the bit pattern in register R ond bit to the right X times. Each time place the bit that started at the low-order end at the high-order end. | A403 would cause the contents of register 4 to be rotated 3 bits to the right in a circular fashion. |
| B | RXY | JUMP to the instruction located in the memory cell at address XY if the bit pattern in register R is equal to the bit pattern in register 0. Otherwise, continue with the normal sequence of execution. (The jump is implemented by copying XY into the program counter during the execute phase.) | B43C would first compare the contents of register 4 with the contents of register 0. If the two were equal, the pattern 3C would be placed in the program counter so that the next instruction executed would be the one located at the memory address 3C. Otherwise, nothing would be done and program execution would continue in its normal sequence. |
| C | 000 | HALT execution. | C000 would cause program execution to stop. |

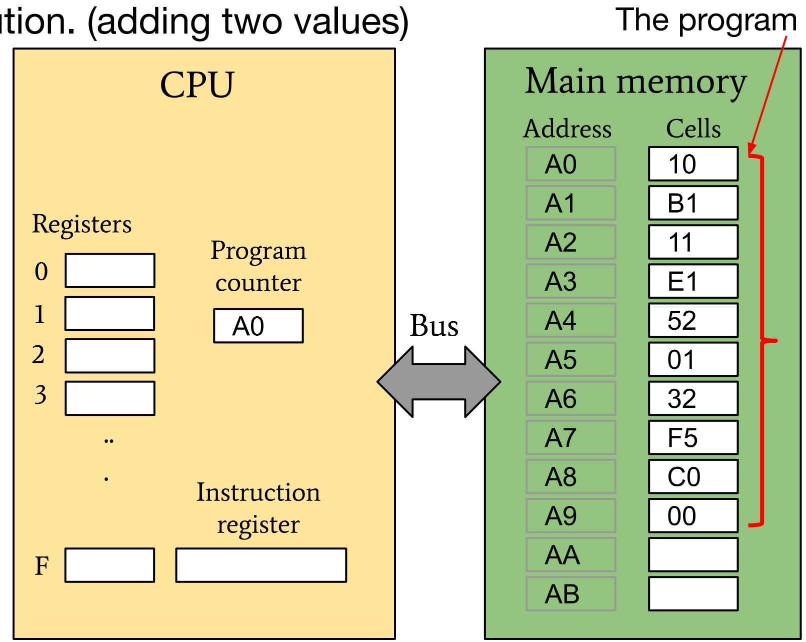
R, S, and T:
3 different registers

X, and Y:
Hexadecimal digits

# Example: adding two values

The program is stored in main memory and ready for execution. (adding two values)

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32F5 | Store the contents of register 2 at address F5 |
| C000 | Halt |

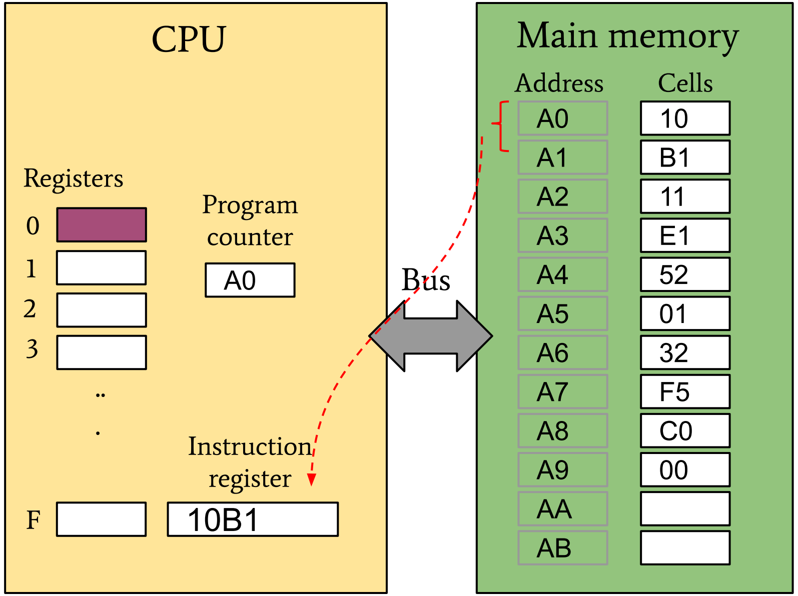



The first instruction is loaded from A0, because Program counter is set to be A0 at the beginning.

# Example of execution

The program is stored in main memory and ready for execution. (adding two values)

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32F5 | Store the contents of register 2 at address F5 |
| C000 | Halt |

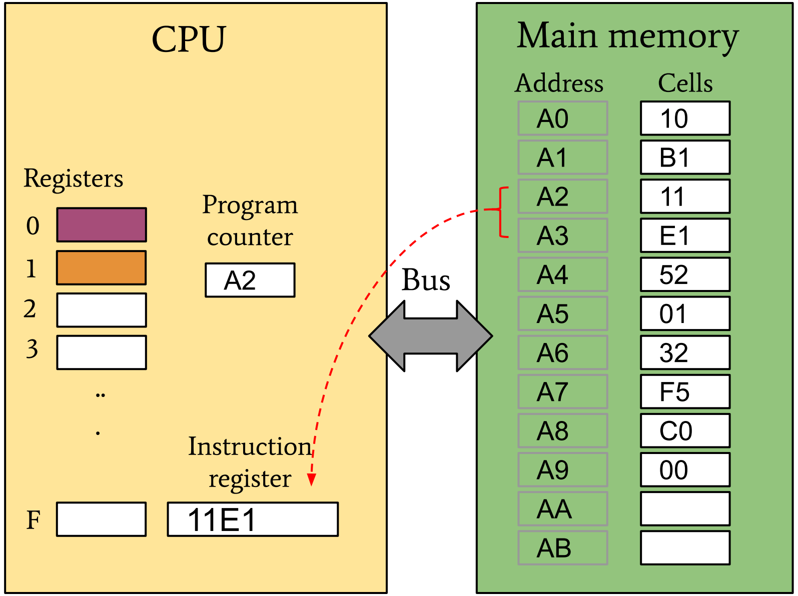



Loading from A0: Since each instruction in Vole has 2 bytes long, the CPU fetches two memory cells from the main memory and **increments** the **PC by 2** (2 is the default value of this machine) so that the next fetch will start from A2.

# Example of execution

The program is stored in main memory and ready for execution. (adding two values)

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32F5 | Store the contents of register 2 at address F5 |
| C000 | Halt |

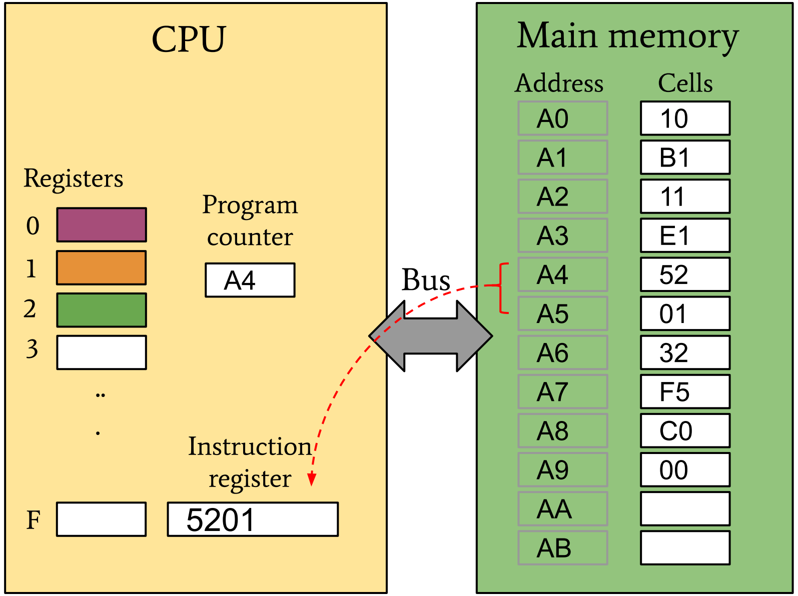



Loading from A2, incrementing PC by two to A4

# Example of execution

The program is stored in main memory and ready for execution. (adding two values)

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32F5 | Store the contents of register 2 at address F5 |
| C000 | Halt |

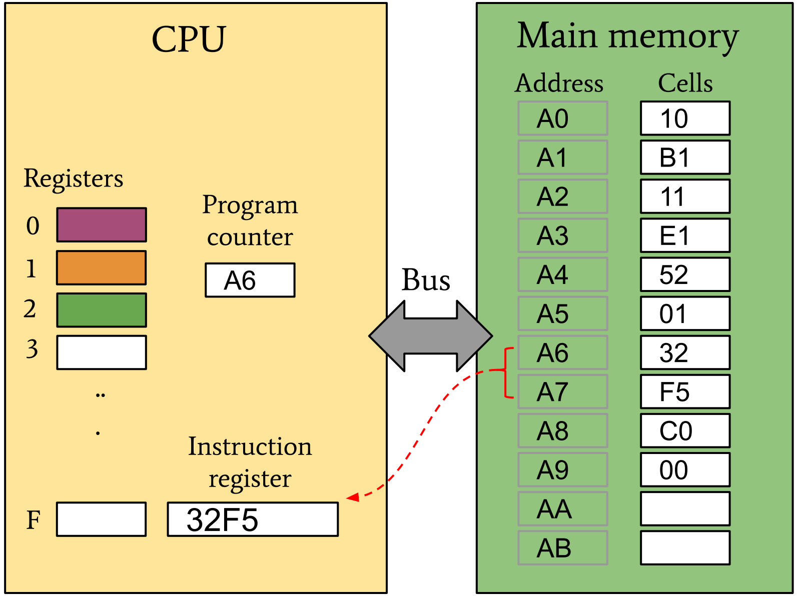



Loading from A4, incrementing PC by two to A6.

# Example of execution

The program is stored in main memory and ready for execution. (adding two values)

| Encoded instructions | Translation |
| --- | --- |
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32F5 | Store the contents of register 2 at address F5 |
| C000 | Halt |

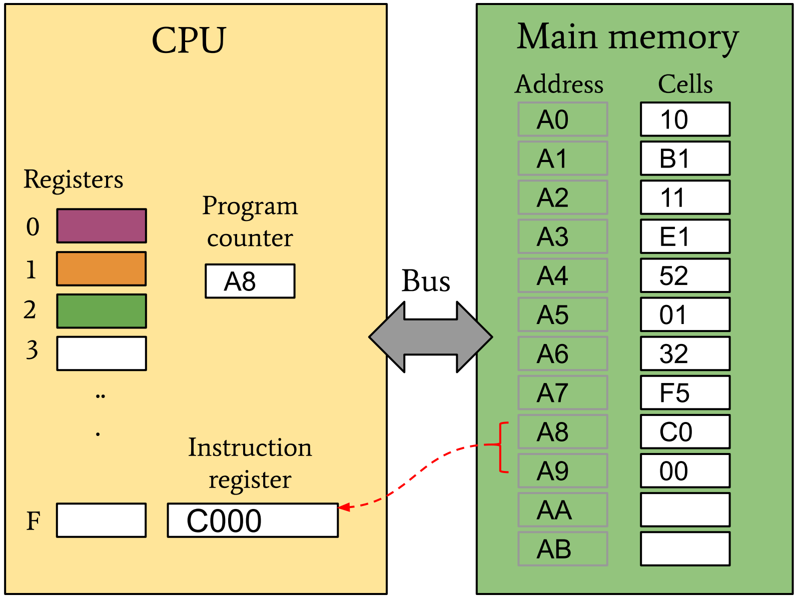



Loading from A6, incrementing  PC by two to A8

# Example of execution

The program is stored in main memory and ready for execution. (adding two values)

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32F5 | Store the contents of register 2 at address F5 |
| C000 | Halt |




Loading from A8, incrementing PC by two to AA; the program ends.

# Example: run if/else in Vole

Now, let's see the execution of some Python script in Vole.

1. The if/else statements (i.e., branch)

Assume the value of m is stored at **B1**, and the value of n is stored at **E1**.

```
m = 3
n = 3

if m==n:
    m = m+n
else:
    m = n
```

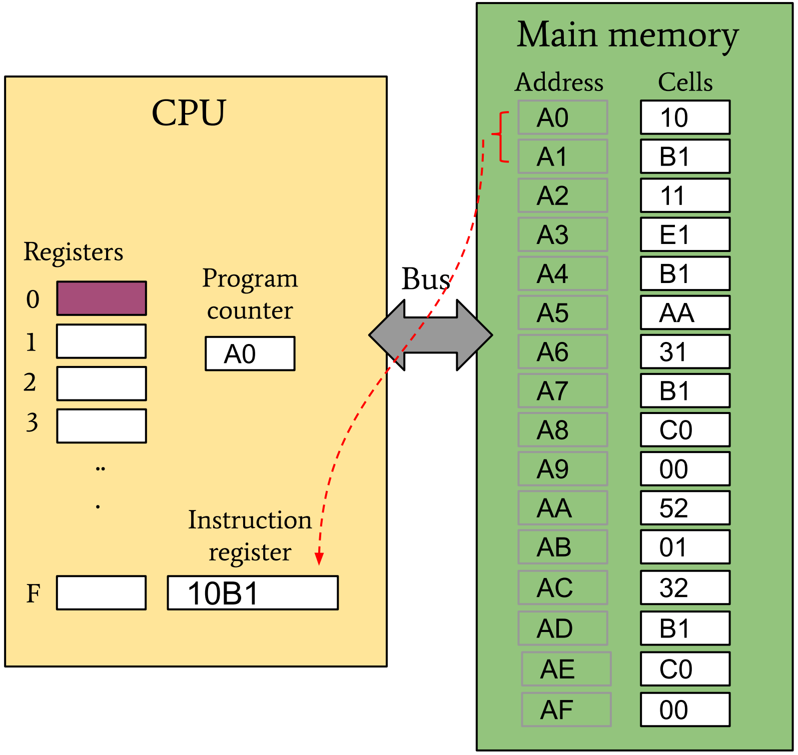
We carry out the if/else by using JUMP.

| B | R | X | Y |
|---|---|---|---|
| JUMP | Register R | Memory address XY | |

Put **XY** into the **PC** if the value in **Register R** is equal to **Register 0.**

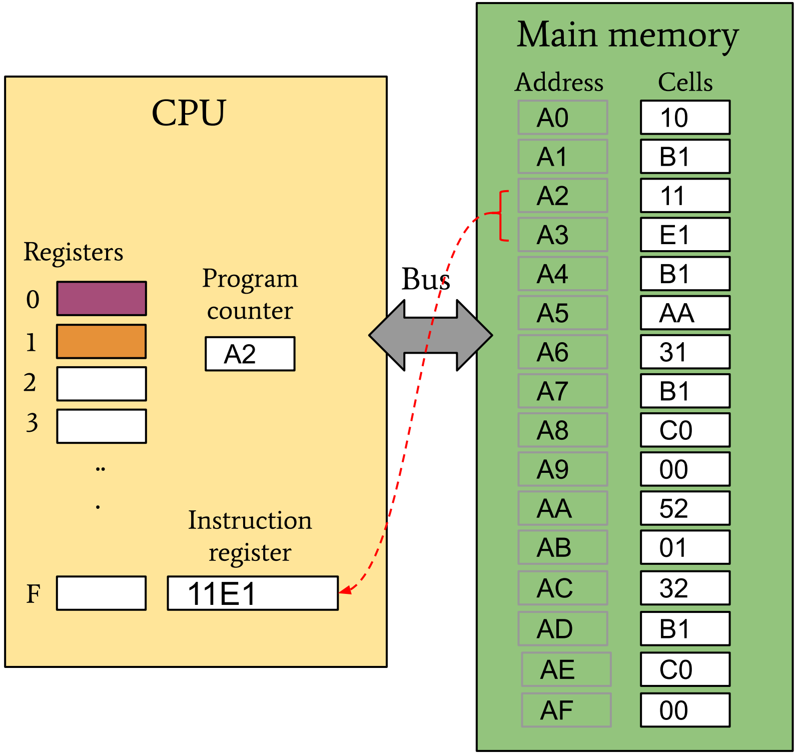
# The if/else statement

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| B1AA | If Reg.1 is equal to Reg. 0, jump to AA |
| 31B1 | Store the contents of Reg.1 at address B1 |
| C000 | Halt |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32B1 | Store the contents of Reg. 2 at address B1 |
| C000 | Halt |



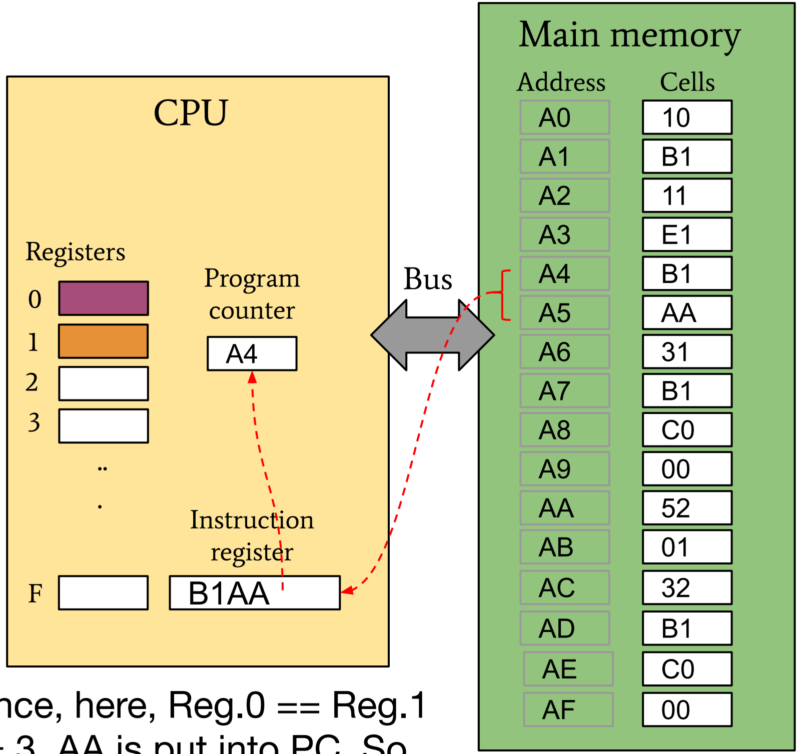
# The if/else statement

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| B1AA | If Reg.1 is equal to Reg. 0, jump to AA |
| 31B1 | Store the contents of Reg.1 at address B1 |
| C000 | Halt |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32B1 | Store the contents of Reg. 2 at address B1 |
| C000 | Halt |



# The if/else statement

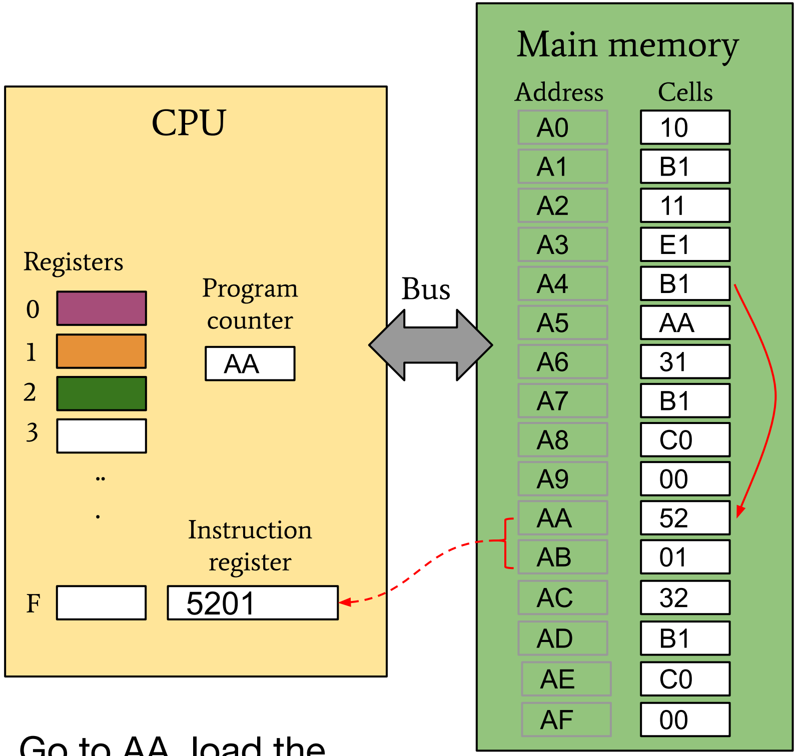
| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| B1AA | If Reg.1 is equal to Reg. 0, jump to AA |
| 31B1 | Store the contents of Reg.1 at address B1 |
| C000 | Halt |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32B1 | Store the contents of Reg. 2 at address B1 |
| C000 | Halt |




Since, here, Reg.0 == Reg.1 == 3, AA is put into PC. So, next instruction is from AA.

# The if/else statement

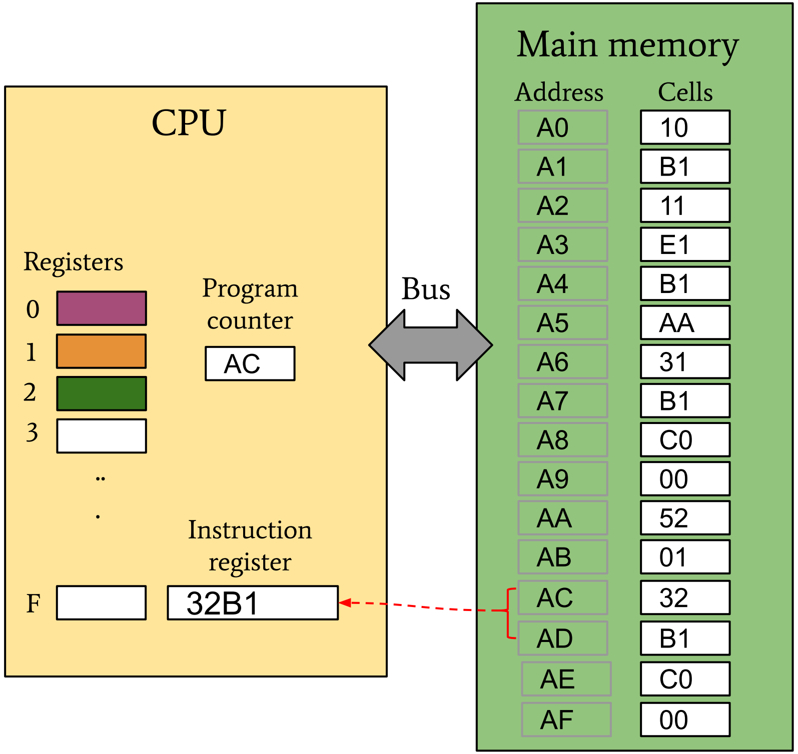
| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| B1AA | If Reg.1 is equal to Reg. 0, jump to AA |
| 31B1 | Store the contents of Reg.1 at address B1 |
| C000 | Halt |
| 5201 | Add contents of Reg. 0 and Reg. 1; leave the result in Reg. 2 |
| 32B1 | Store the contents of Reg. 2 at address B1 |
| C000 | Halt |




Go to AA, load the instruction into IR

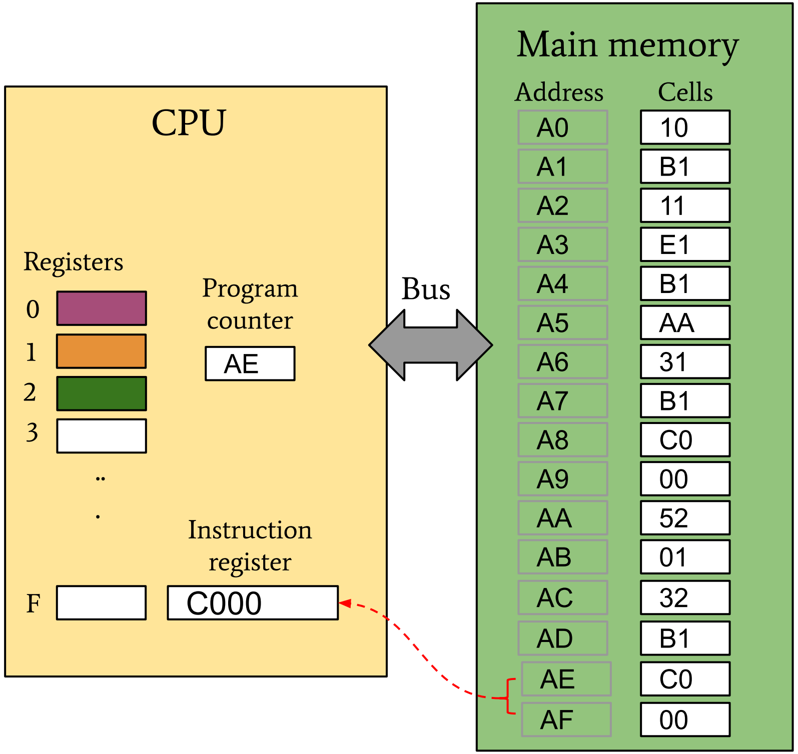
# The if/else statement

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| B1A8 | If Reg.1 is equal to Reg. 0, jump to AA |
| 31B1 | Store the contents of Reg.1 at address B1 |
| C000 | Halt |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32B1 | Store the contents of Reg. 2 at address B1 |
| C000 | Halt |

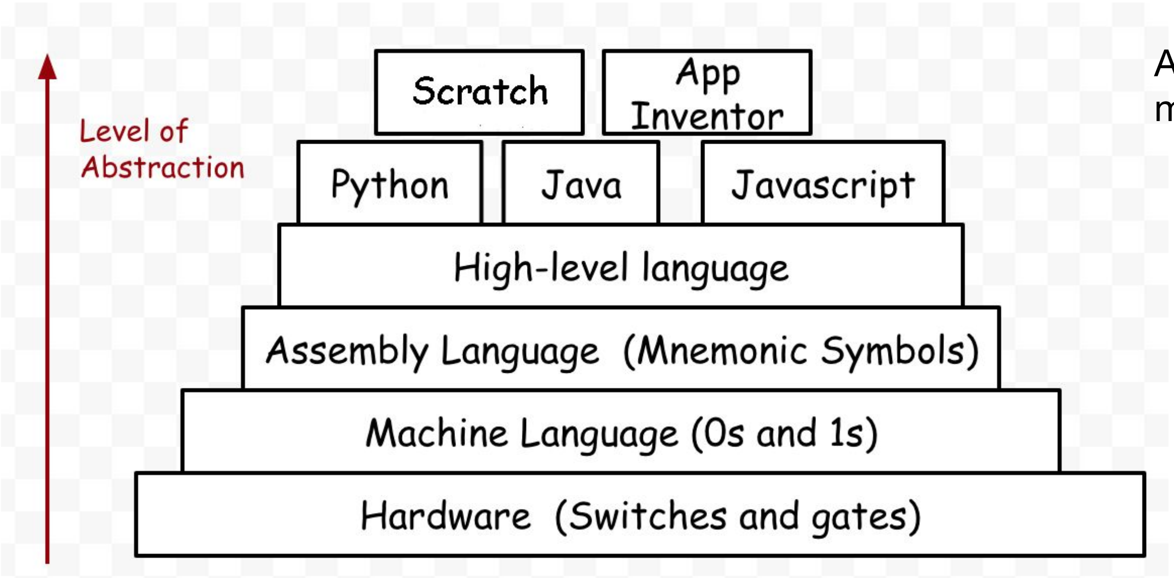


# The if/else statement

| Encoded instructions | Translation |
|---|---|
| 10B1 | Load register 0 from address B1 |
| 11E1 | Load register 1 from address E1 |
| B1A8 | If Reg.1 is equal to Reg. 0, jump to AA |
| 31B1 | Store the contents of Reg.1 at address B1 |
| C000 | Halt |
| 5201 | Add contents of register 0 and 1; leave the result in register 2 |
| 32B1 | Store the contents of Reg. 2 at address B1 |
| C000 | Halt |



# Levels of languages




Abstraction: a powerful tool for managing complexity.

- representing complex systems, processes, or data in a simplified or more manageable form, while hiding unnecessary details or complexities

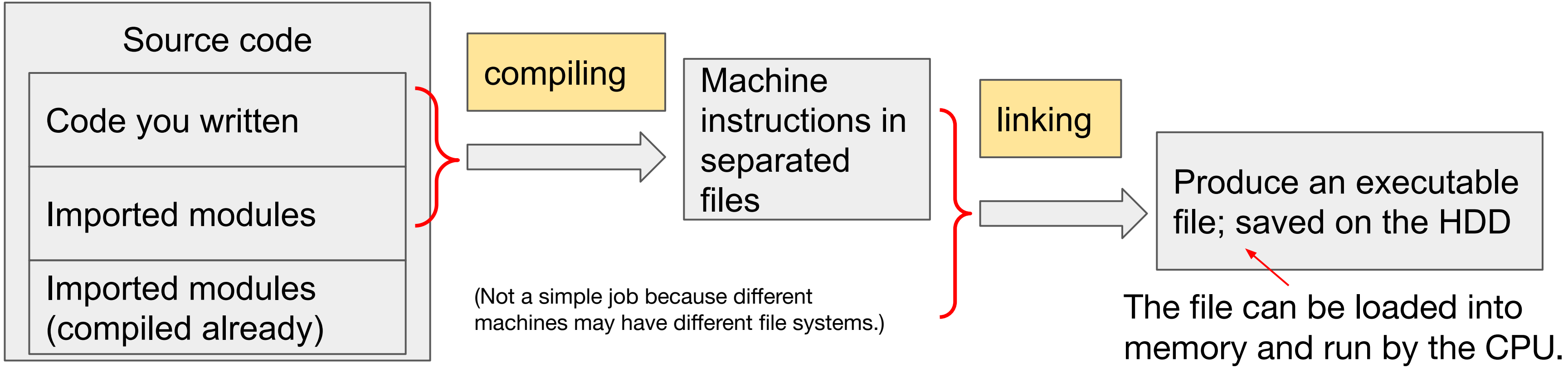
# Running codes in high-level languages

- CPUs only “understand” machine languages.
- Programs in high-level languages must be translated into machine languages
  ⇒ compiling and linking are needed

# Compiling and Linking

Translating the source code to a runnable file

- Compilation: produce the machine language instructions according to the source code
- Linking: merge all the files that are needed to execute your code. For example, the modules/packages you imported in your source code.



# Two types of the translation

- using a compiler: C, C++, Java, etc.
- using an interpreter: Python, Javascript, Matlab
- But both need compiling and linking

**Figure 1-18** Compiling a high-level program and executing it

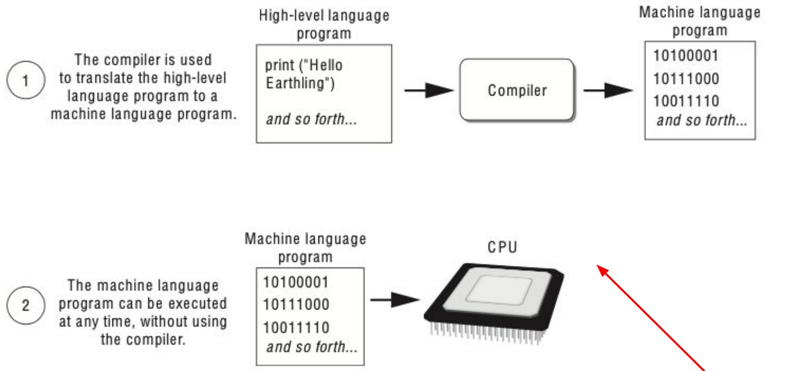
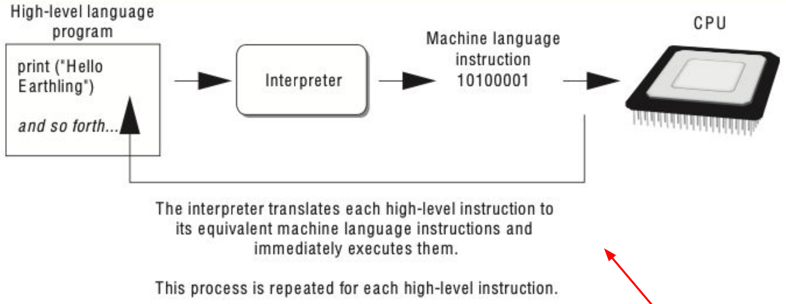



**Figure 1-19** Executing a high-level program with an interpreter



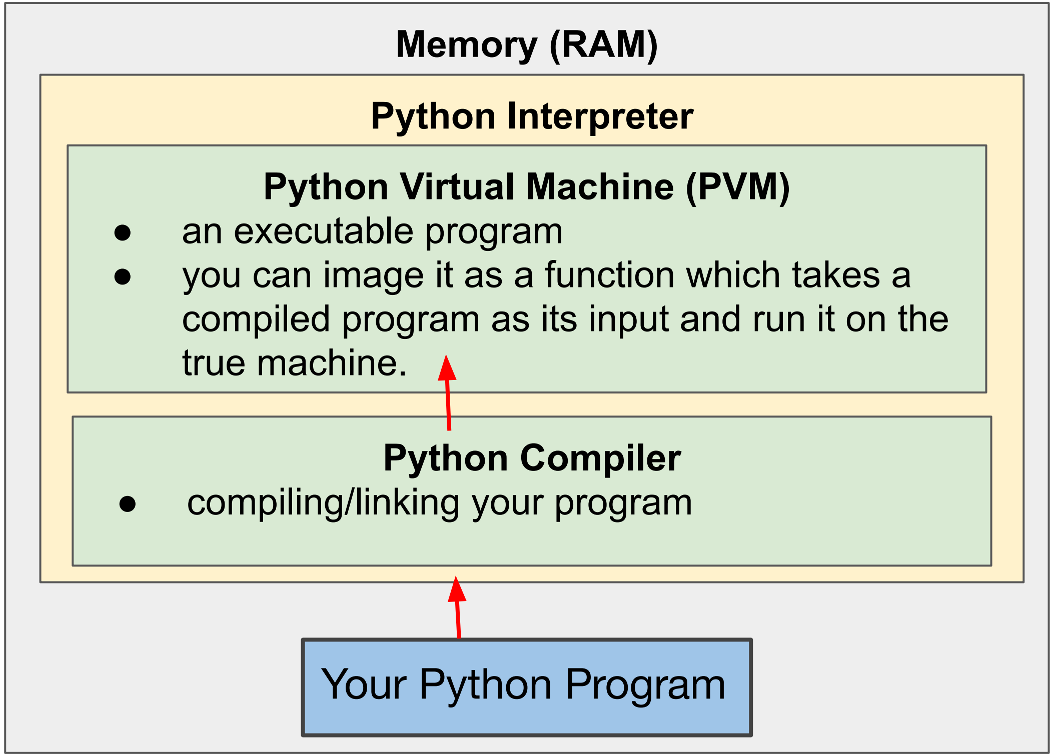

Translating the all instructions into machine language in one time.

Translating and executing instructions one by one.

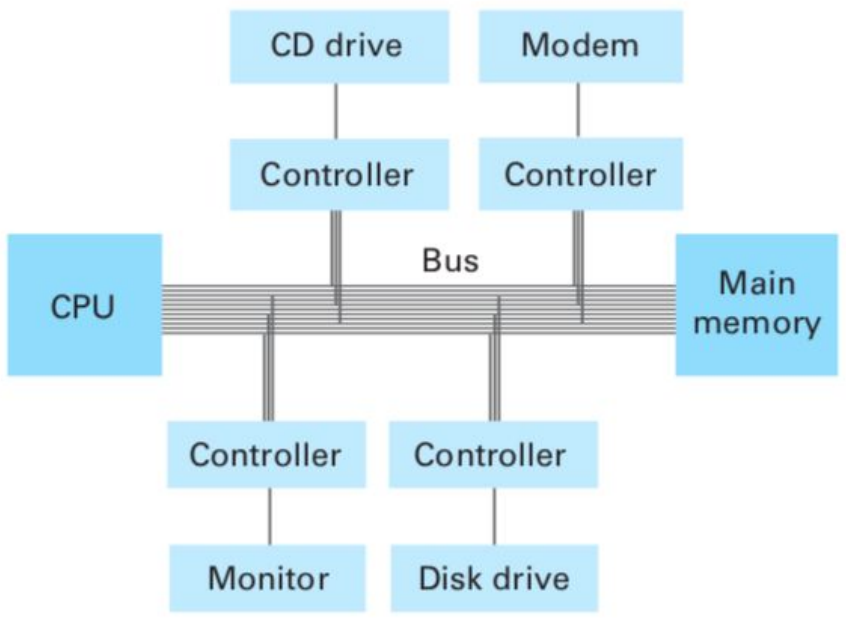
# Running Python code

Every statement in your program will be compiled and linked by the compiler, then be executed by the PVM, one by one.

- The compiler translates a statement into the instructions can only be understood by the PVM, called **bytecode**;
- PVM runs the bytecode;
- No need to make an executable file on HDD, then, loading and executing it. (because the PVM is already loaded in memory and can run the code instantly)



# Communicating with peripheral devices




CPU and memory communicate with devices such as disk drivers, keyboards through an intermediary apparatus, called controller.

A controller translates messages and data back and forth between the computer and the device.

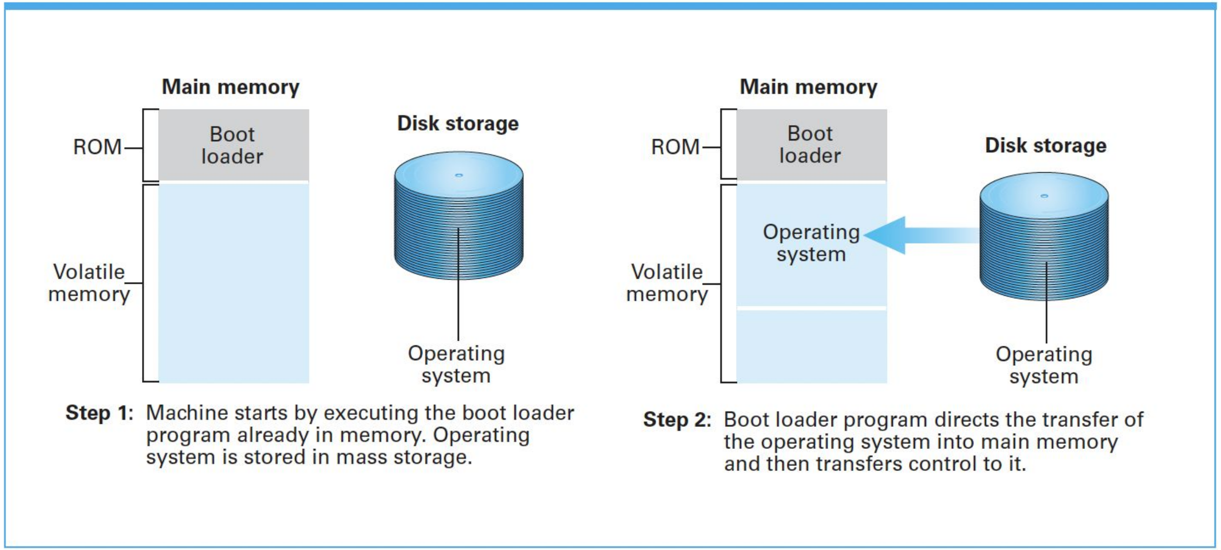
Some communication standards:

- Universal serial bus (usb)
- High definition multimedia interface (HDMI)
- DisplayPort (DP)

# To get it started $\Rightarrow$ two steps

1. When you turn on your computer, the program (i.e., boot loader) stored in the ROM is in the main memory and is executed immediately.
2. Boot loader will load the operating system into main memory and transfer control to it.

**Figure 3.5** The booting process



Appendix:

- Play with RodRego
- Binary number measurement

# Register machine: RodRego

RodRego is a simplified computer:

- Only has ten registers (each has an ID, from 0-9)
- Three instructions: INC, DEB, END

You may watch the introduction, then, try the exercises in the following slides

- Introduction to RodRego: https://stream.nyu.edu/media/RodRego/0_a4mklaon
  - Introduction slides: https://docs.google.com/presentation/d/1Gyg6Crp48d5sPz6_NkJqfbQ5yVEbNbwbad0evt4KmSE/edit?usp=sharing
- Go to RodRego: https://rodrego.it.tufts.edu/

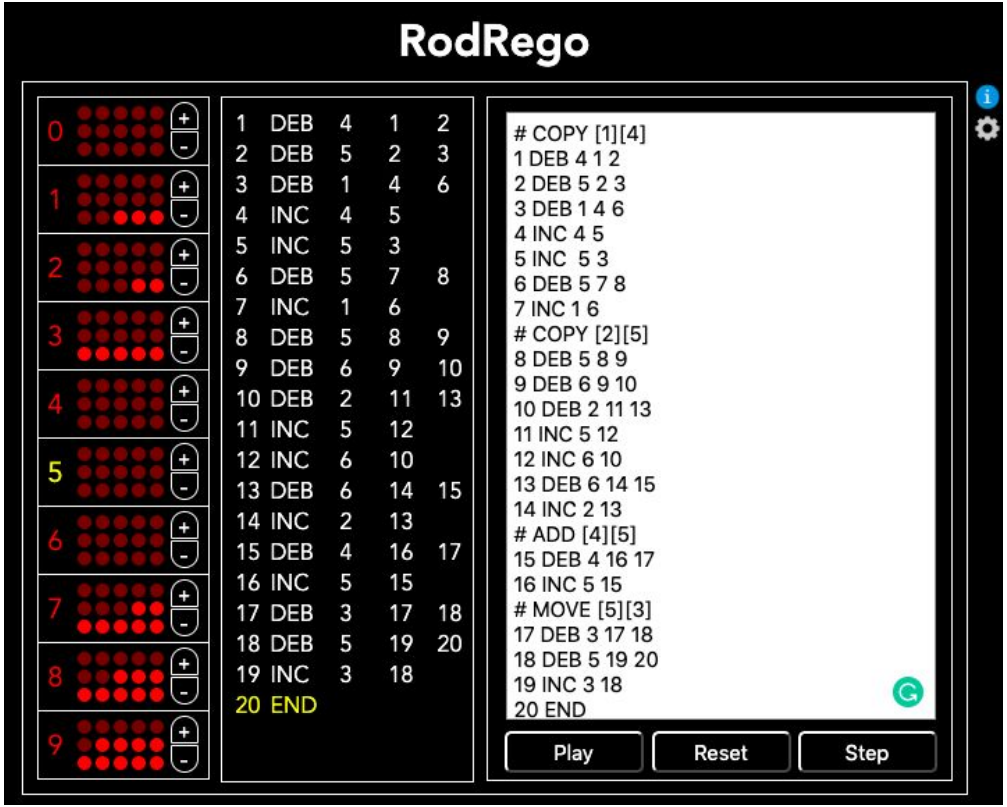
# Now it's your turn.

1. Non-destructive ADD [1] [2] [3]: add Reg 1 and Reg 2, and save the result in Reg [3] (register 1 and 2 stay)
2. MULTIPLY [1] [2] [3]: multiply Reg 1 and Reg 2, and save the result in Reg [3]

# Non-destructive add [1] [2] [3]

1. Copy [1] [4]
2. Copy [2] [5]
3. ADD [4] [5]
4. CLEAR [3]
5. MOVE [5] [3]

Question: Can you do better?



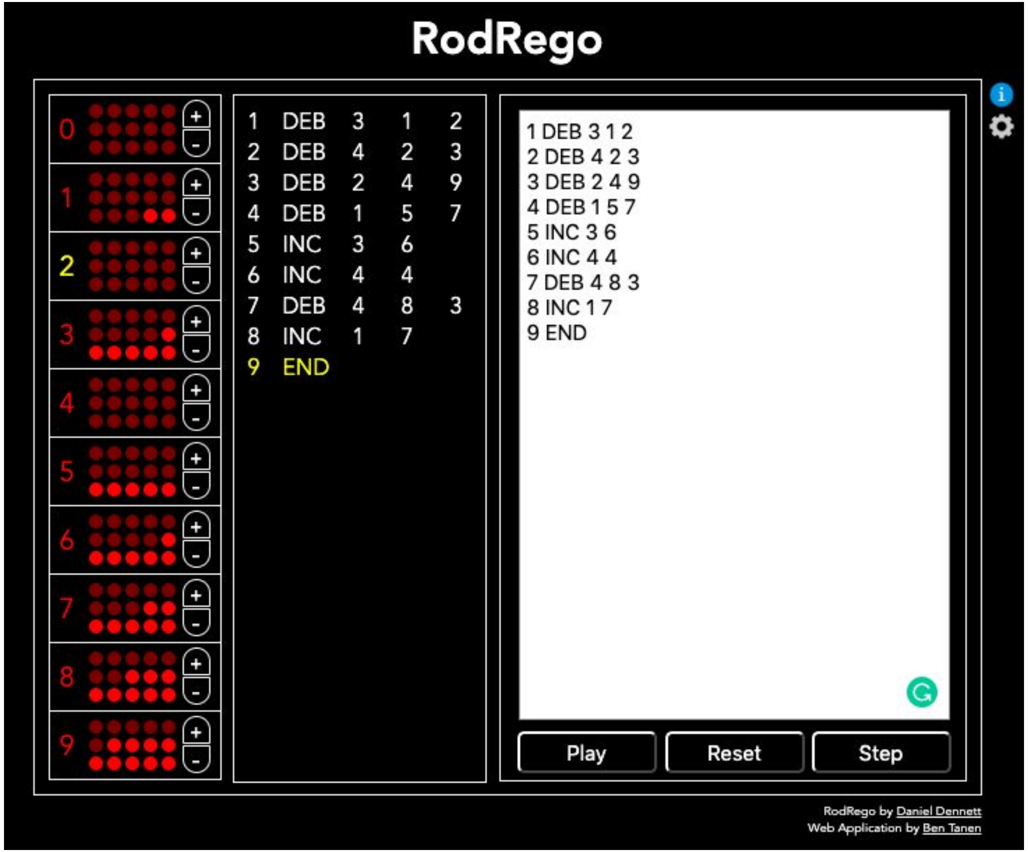
# Multiply [1][2][3]

Assume:

Reg 1 = N

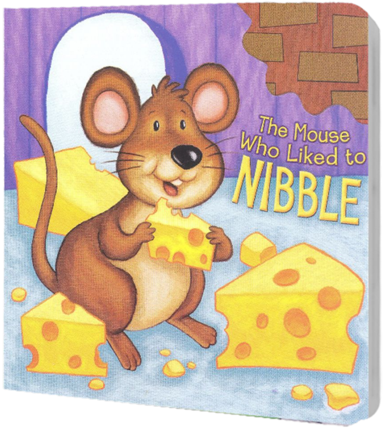
Reg 2 = M

1. CLEAR [3]
2. LOOP M times:
    - COPY [1][3]



# Binary number measurement

# Several definitions you should know




- a bit : a binary digit
  - It has two possible values: 0, and 1.
- a byte: a group of eight bits
  - It can represent $2^8 = 256$ possibilities.
- a nibble: a group of four bits (no longer a commonly use)
  - It can represent $2^4 = 16$ possibilities.
- a word: the size of the chunk of data that a microprocessor handling at a time.
  - It dependents on the architecture of the microprocessor.
  - a 64-bit microprocessor means it operates on 64-bit word

# File size

1 byte = 8 bit

| | | |
|---|---|---|
| 1024 byte = 1 KB | KB = Kilobyte | $2^{10} = 1024 \approxeq 10^3$ |
| 1024 KB = 1 MB | MB = Megabyte | $2^{20} \approxeq 10^6$ |
| 1024 MB = 1 GB | GB = Gigabyte | $2^{30} \approxeq 10^9$ |
| 1024 GB = 1 TB | TB = Terabyte | $2^{40} \approxeq 10^{12}$ |
| 1024 TB = 1 PB | PB = Petabyte | $2^{50} \approxeq 10^{15}$ |