Computer Architecture

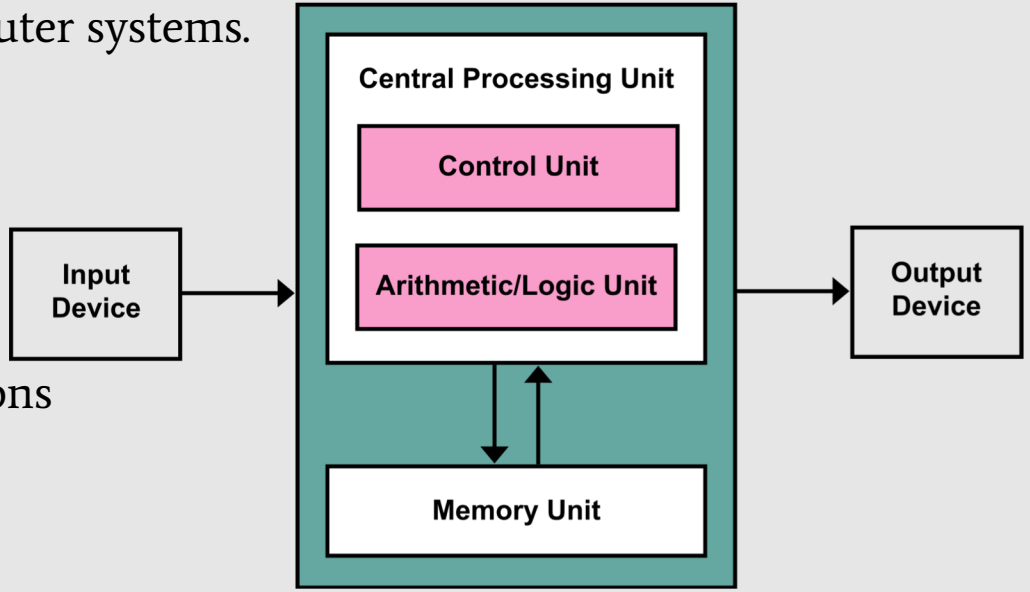
Agenda

- Computer architecture
- Machine language
  - Two philosophies
  - Vole: a simple computer
  - Cycle of program execution
- Programming language

Computer architecture

von Neumann architecture

- The mainstream of modern computer systems.
- Has a central processor unit
  - Control unit
  - ALU
  - Registers (memory in CPU)
- Memory stores data and instructions
- Input and output mechanisms



The schematic is from from Wikipedia.

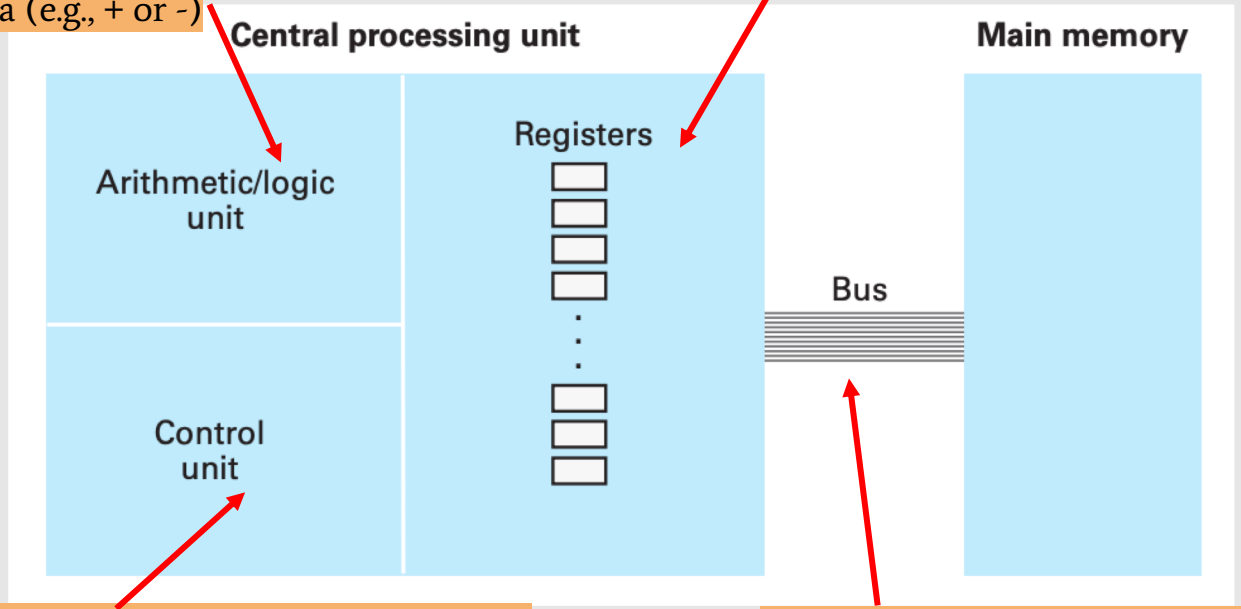
The core of a computer

General purpose register

temporarily storing the information

Special purpose register
- Instruction register
- Program counter

performing operations on data (e.g., + or -)



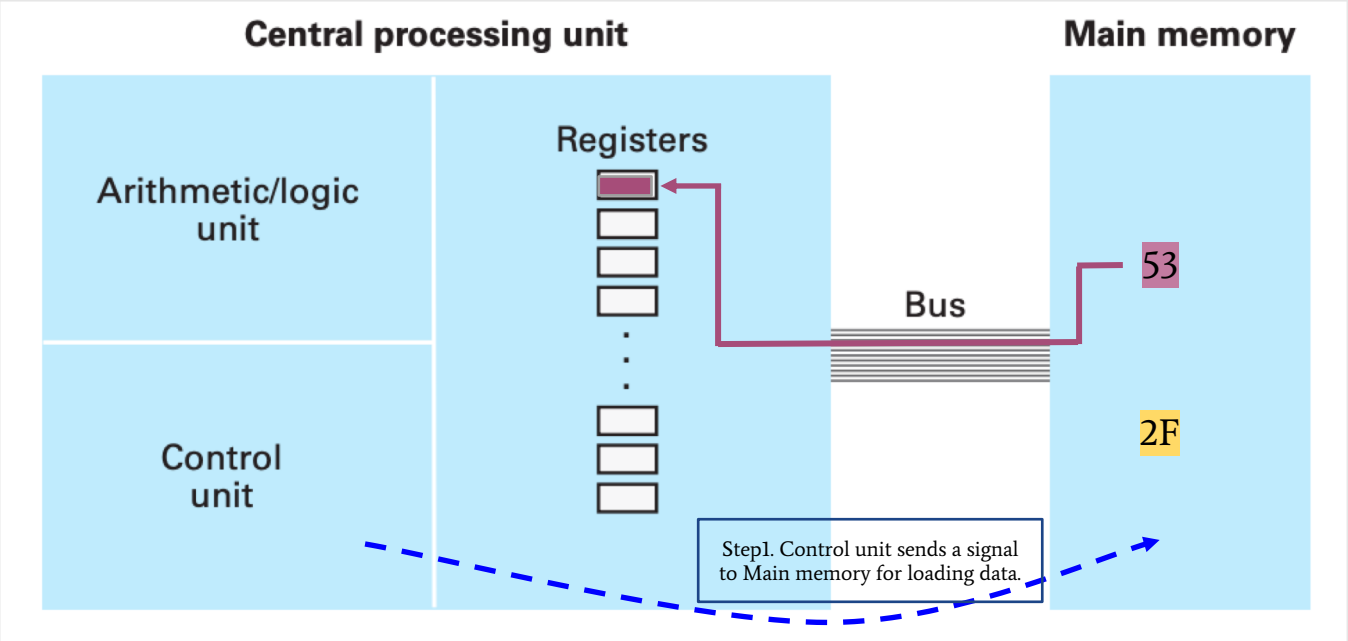
coordinating the machine's activities (e.g., when to load/write/+/- data)

transferring data between CPU and Main memory

How a CPU adds two numbers

To add two values (e.g., 53_{hex}, and $2F_{hex}$) stored in Main memory.

Step 1. Load 53 from Main memory into a register

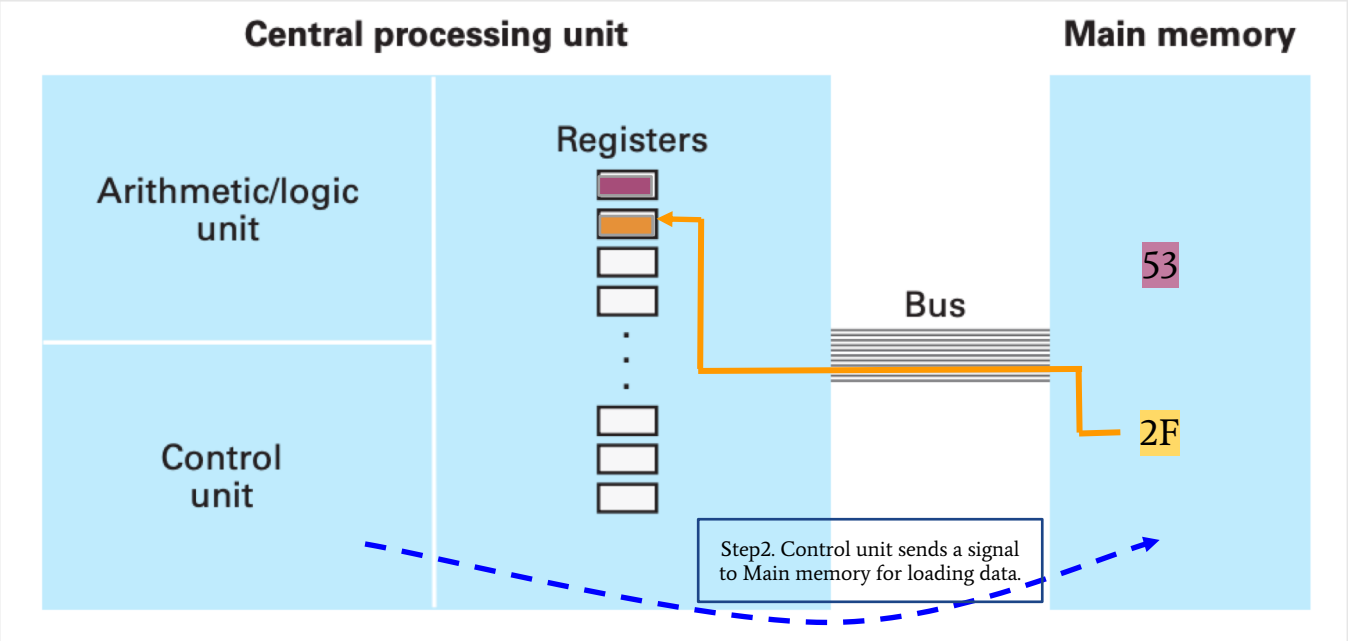


How CPU add two numbers

To add two values (e.g., 53_{hex}, and $2F_{hex}$) stored in Main memory.

Step 1. Load 53 from Main memory into a register

Step 2. Load 2F from Main memory into a register

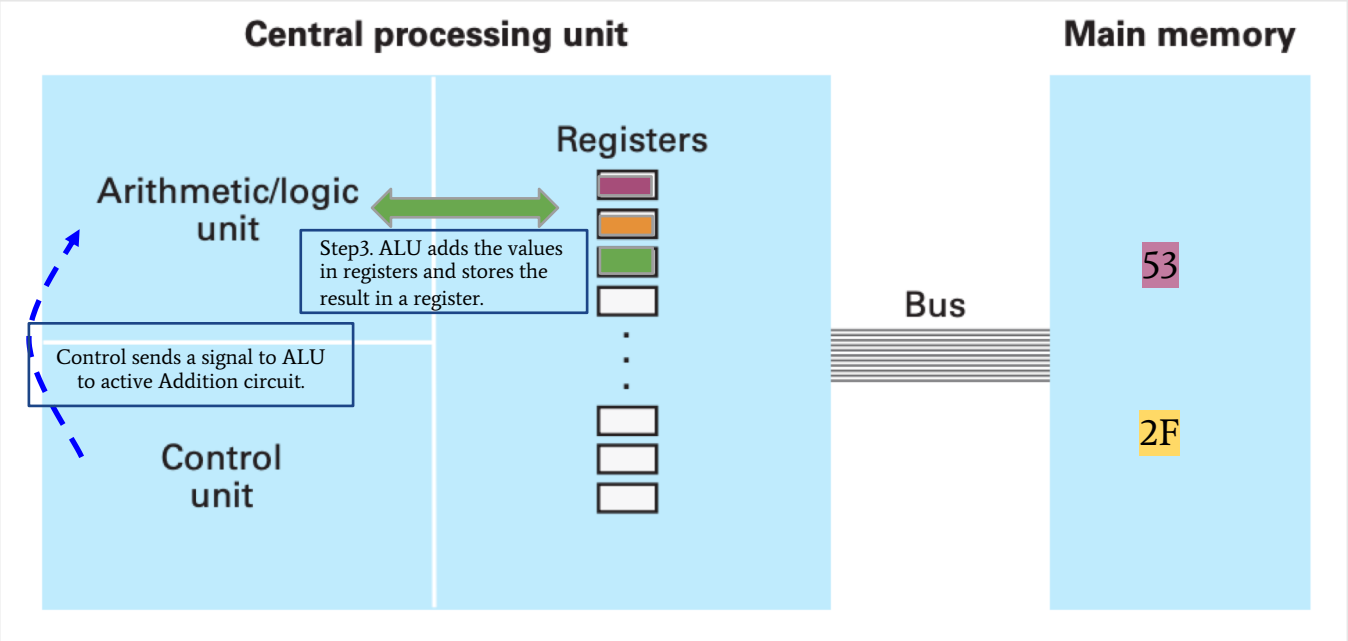


How CPU add two numbers

To add two values (e.g., 53_{hex}, and $2F_{hex}$) stored in Main memory.

Step 1，2. Load two values from Main memory into two registers

Step 3. ALU operates on the registers; the result is temporary stored in a register.



How CPU add two numbers

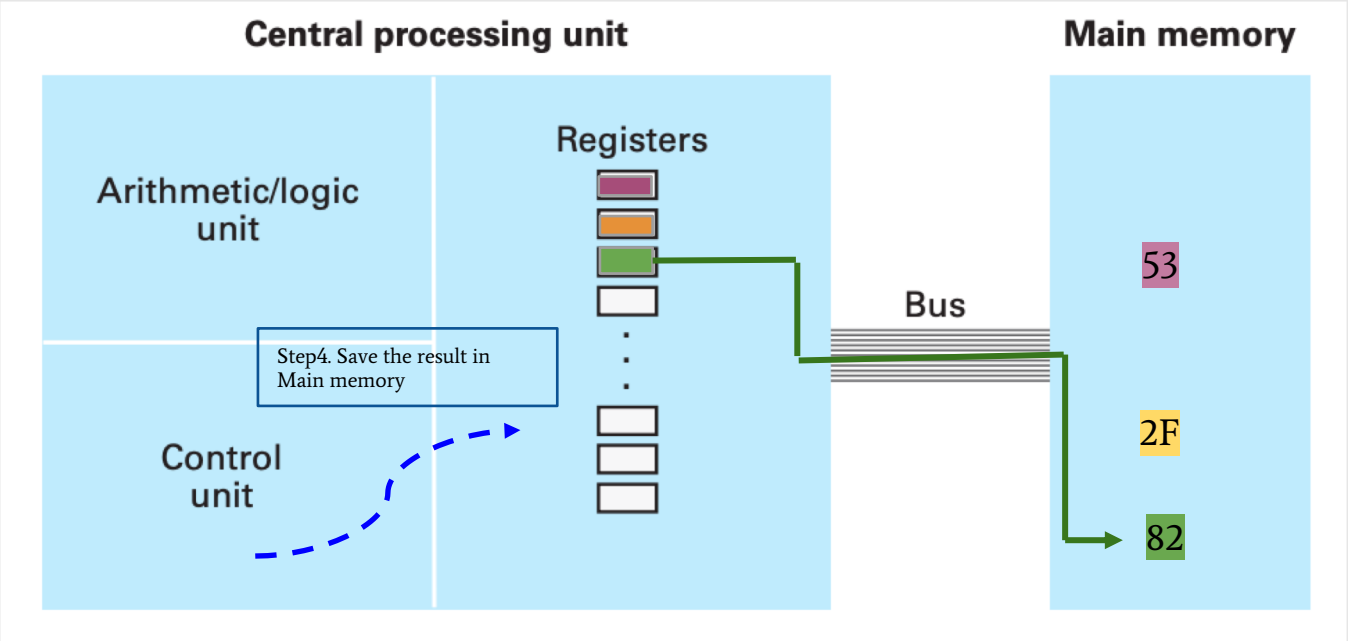
To add two binary values (e.g., 1001, 0010) stored in Main memory.

Step 1， 2. Load two values from Main memory into two registers

Step 3. ALU operates on the registers; the result is temporary stored in a register

Step 4. Store the result in Main memory

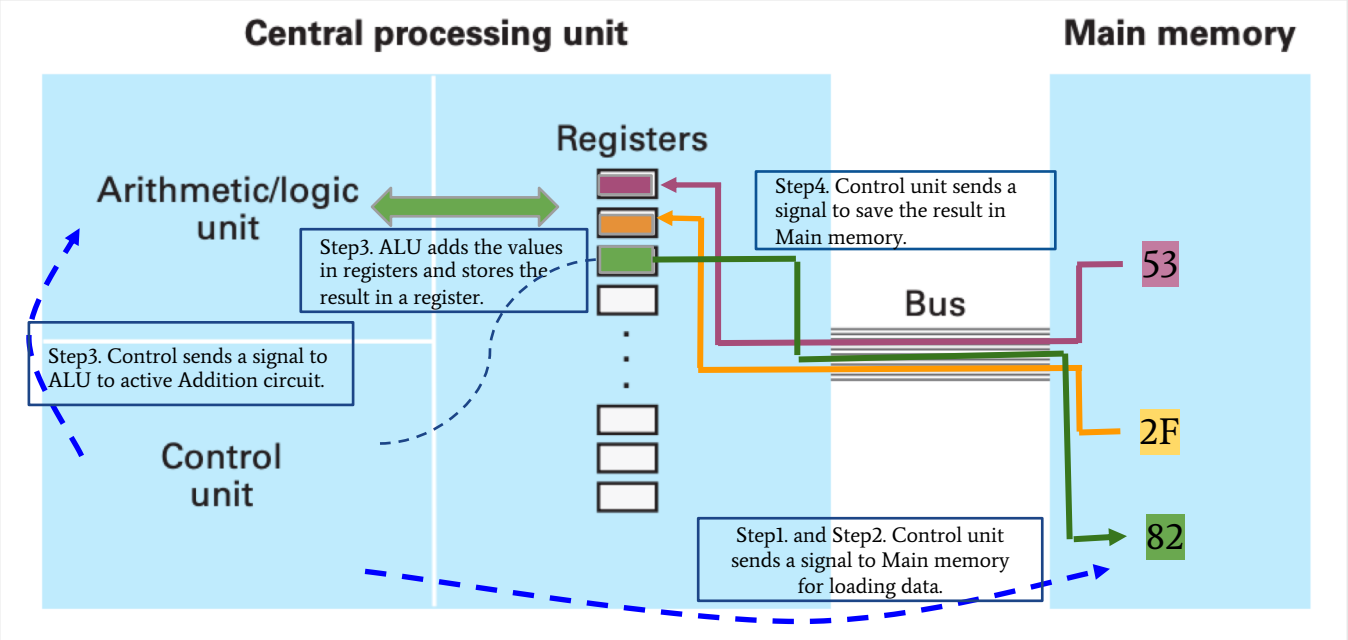
Step 5. Stop



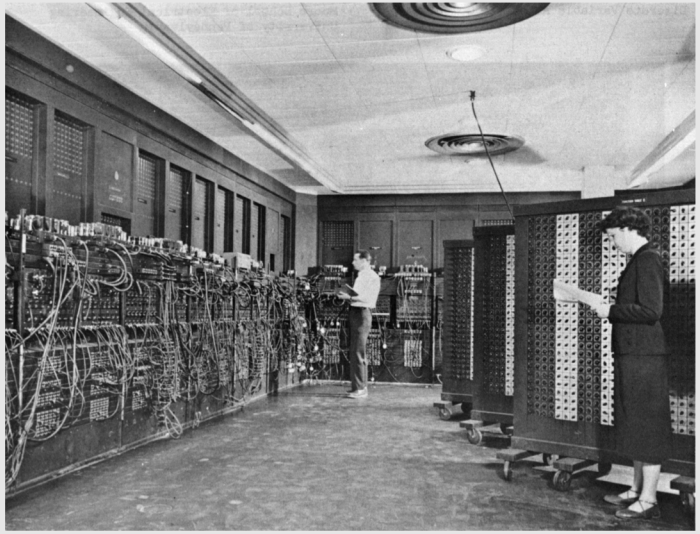
How CPU add two numbers (overview)

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

1. Load 53 from Main memory into a register
2. Load 2F 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



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

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

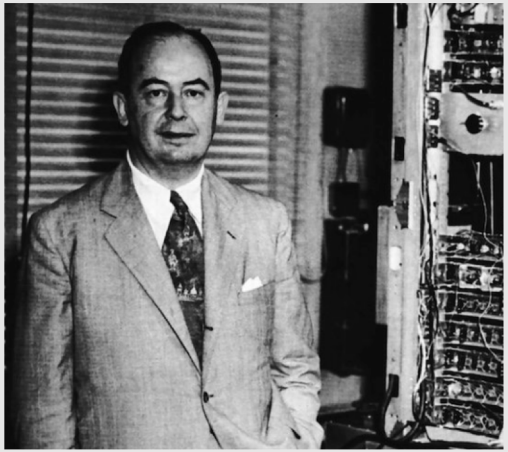
- To program was to reconnect the circuit blocks for different tasks.
- Reprogramming took days

Nowadays, to program is to instruct the control unit. No human intervention is needed to make hardware changes.

Stored-program computers

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



John Von Neumann, 1903-1957

Why instructions can also be stored in main memory?

- They are represented by 0s and 1s.
- Instructions and data can have the same format.

Machine language

Machine language

In stored-program computers, instructions are represented in bit patterns.

- Machine Language is the collection of instructions encoded as bit patterns and the encoding system (instruction set).
- The number of instructions that a particular CPU can have is limited and the collection of all those instructions is called the Instruction Set.

Two philosophies of CPU architecture

- RISC (reduced instructions set computer)
  - CPUs are designed to execute a minimal set of machine instructions, e.g., it has A+B but no A x B. (A x B can be done by run A+A for B times)
  - Light weight and low power consumption.
  - ARM(Advanced RISC Machine). Found in mobiles and tablets etc.
- 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, please have a look at this. (Don't be surprised by its name).

The instruction repertoire

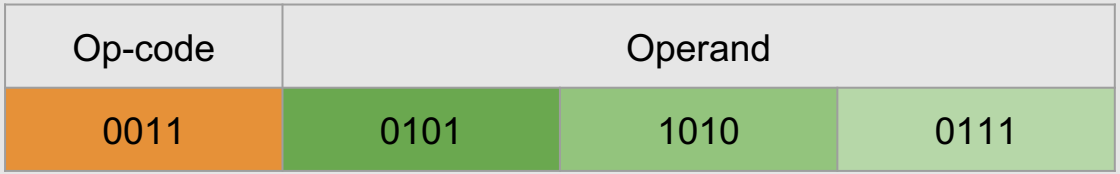
Whether RISC or CISC, machine's instructions can be categorized into 3 groups:

- 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

Encoding instructions

Machine instructions should follow some **predefined** **format** so that CPUs can "understand" them.

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



An example of a bit pattern of a 16-bit instruction.

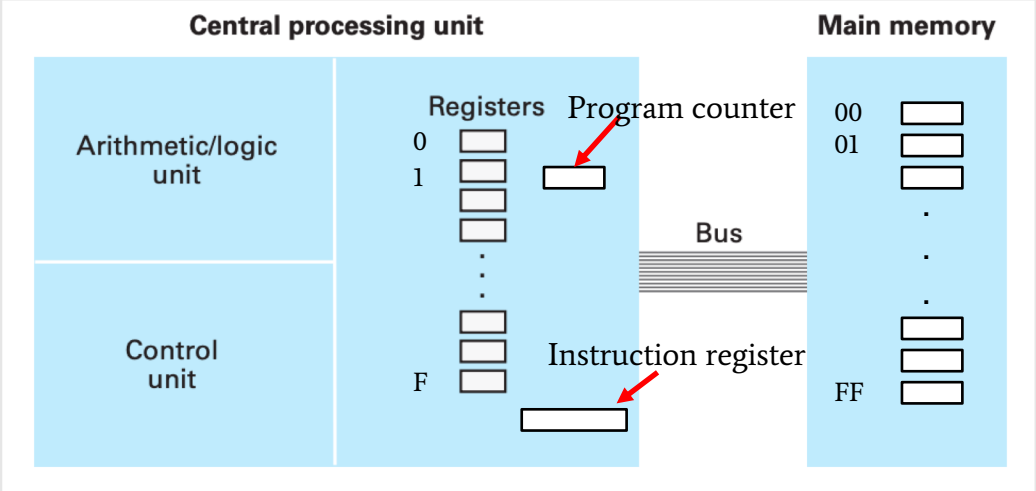
- Different designs of processors may have different instruction formats.
  - Binary code runs on a PC cannot run on an Apple Macintosh. (PC uses x86; Macintosh doesn't.)

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

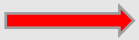
Machine language of Vole

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 convenient 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)

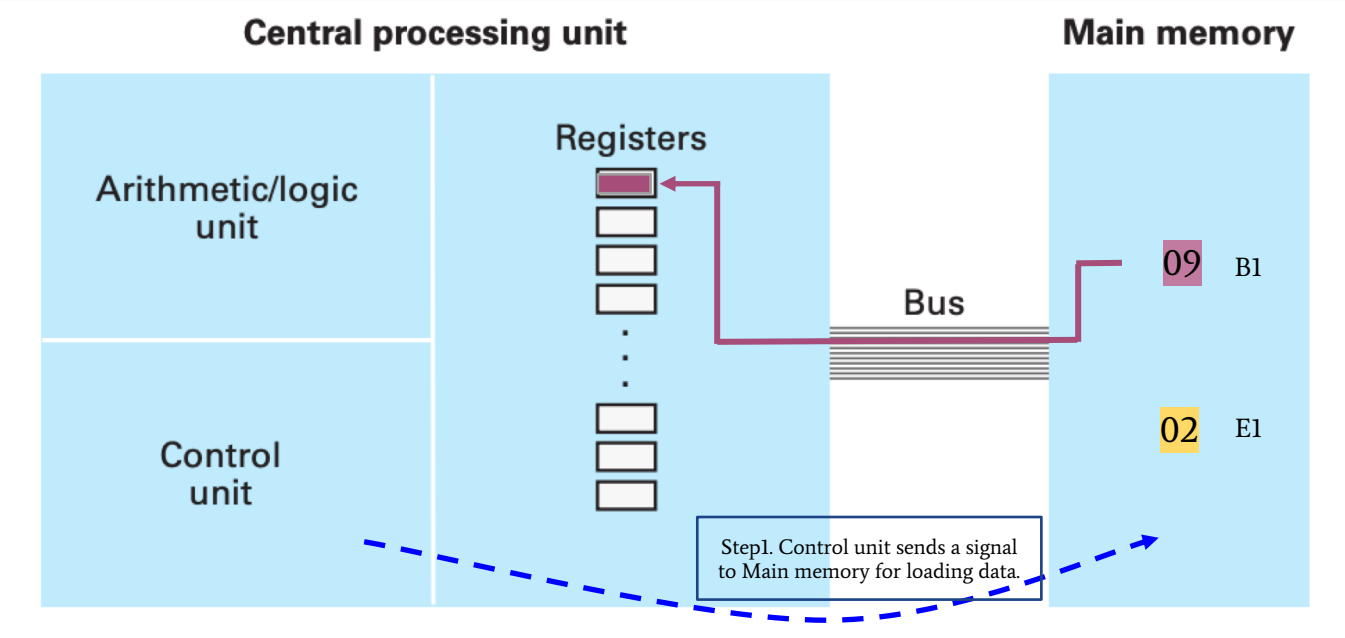
Op-code	Operand	Description	Example
7	RST	OR the bit patterns in register S and T and place the result in register R.	7CB4 would cause the result of ORing the contents of registers B and 4 to be placed in register C.
8	RST	AND the bit patterns in register S and T and place the result in register R.	8045 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.

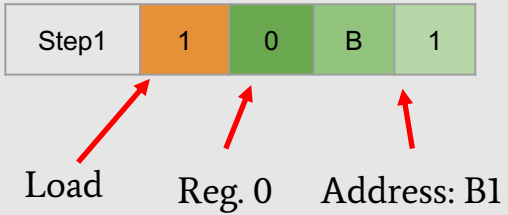
R, S, and T:
3 different registers

X, and Y:
Hexadecimal digits

Translate it into Vole's machine language

Adding two numbers (assume 09 at B1 and 02 at E1; numbers are hexadecimal)

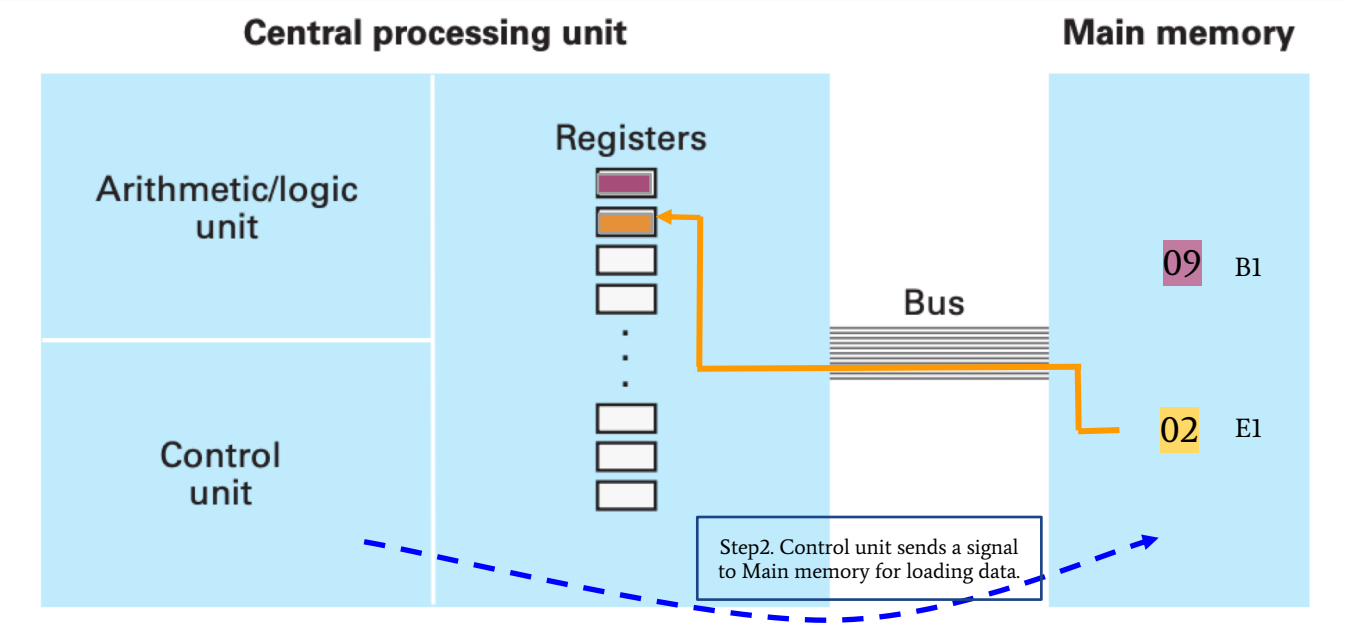




Loading the value in B1 into Reg. 0

Translate it into Vole

Adding two numbers (assume 09 at B1, 02 at E1; numbers are hexadecimal)



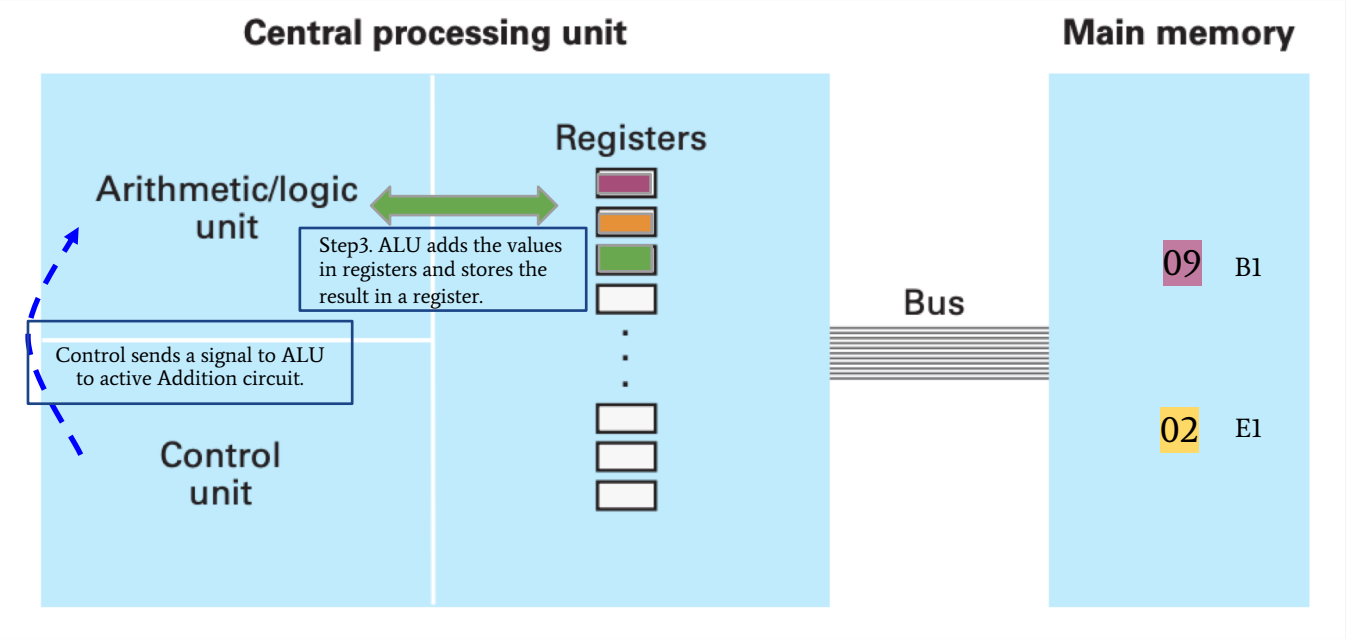
Step1	1	0	B	1
Step2	1	1	E	1

Load Reg. 1 Address: E1

Loading the value in E1 into Reg. 1

Translate it into Vole

Adding two numbers (assume 09 at B1, 02 at E1; numbers are hexadecimal)



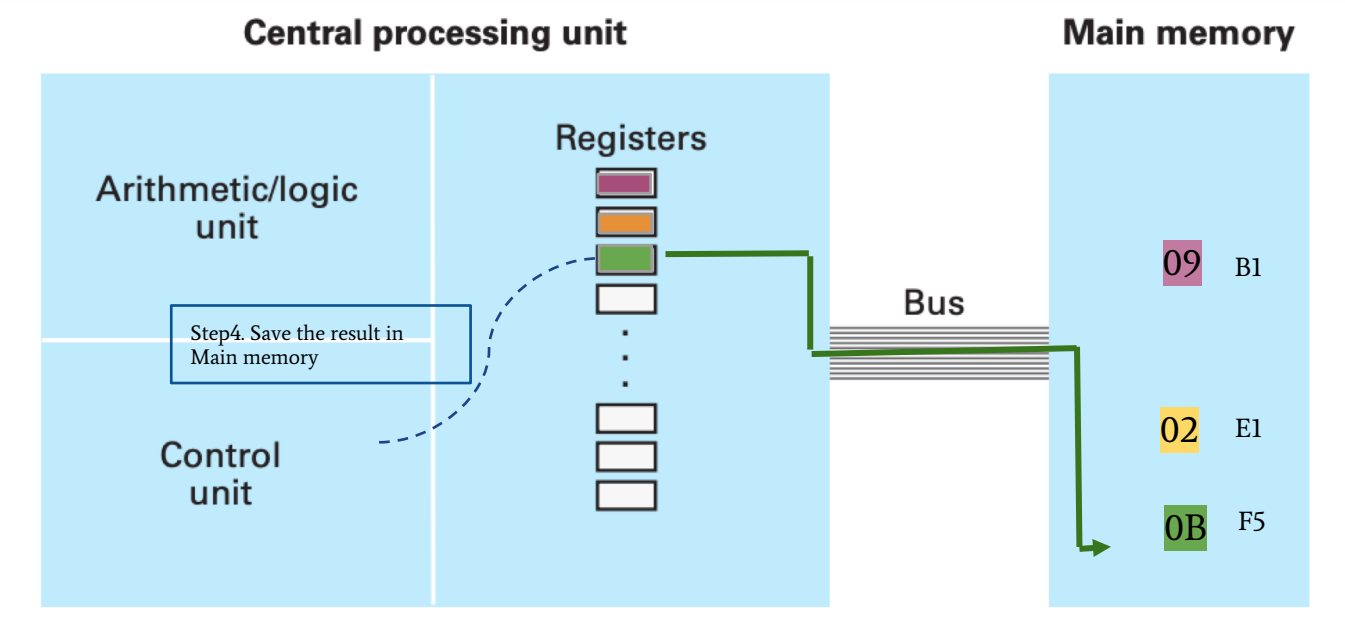
Step1	1	0	B	1
Step2	1	1	E	1
Step3	5	2	0	1

Add Reg. 2 Reg. 0 Reg. 1

Adding values in Reg.0 and Reg.1 and put the result into Reg.2

Translate it into Vole

Adding two numbers (assume 09 at B1, 02 at E1; numbers are hexadecimal)



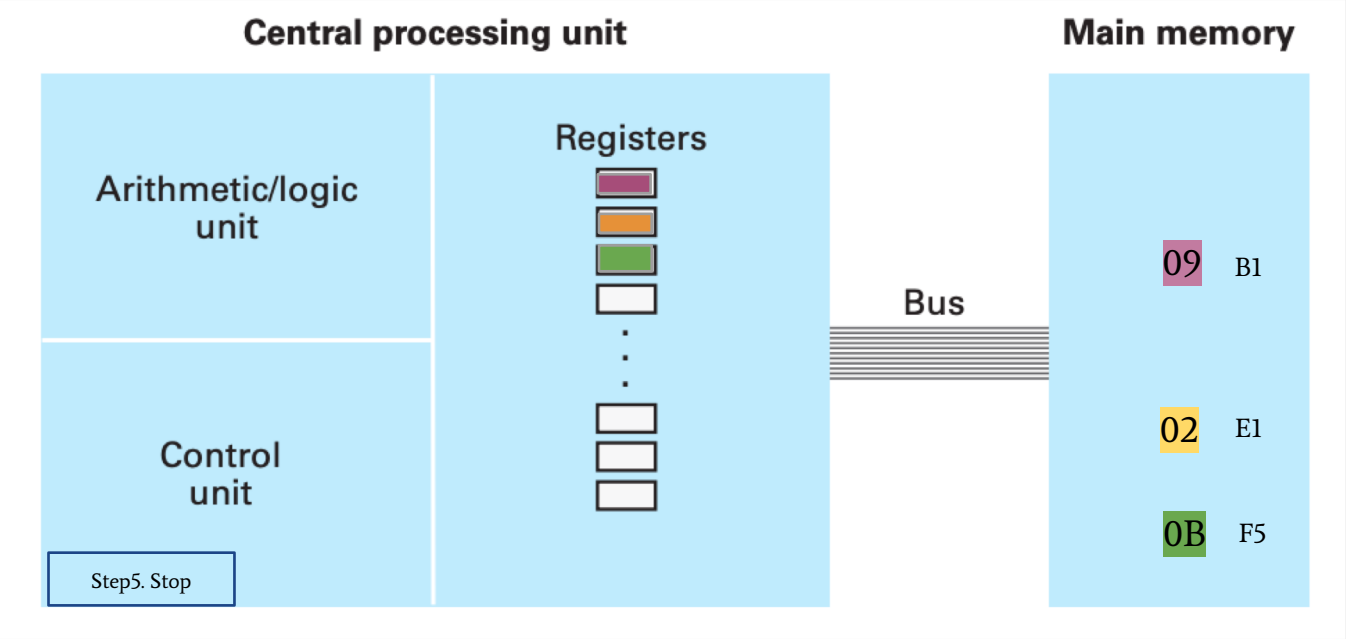
Step1	1	0	B	1
Step2	1	1	E	1
Step3	5	2	0	1
Step4	3	2	F	5

Store Reg. 2 Address: F5

Storing the value in Reg.2 to main memory F5

Translate it into Vole

Adding two numbers (assume 09 at B1, 02 at E1; numbers are hexadecimal)

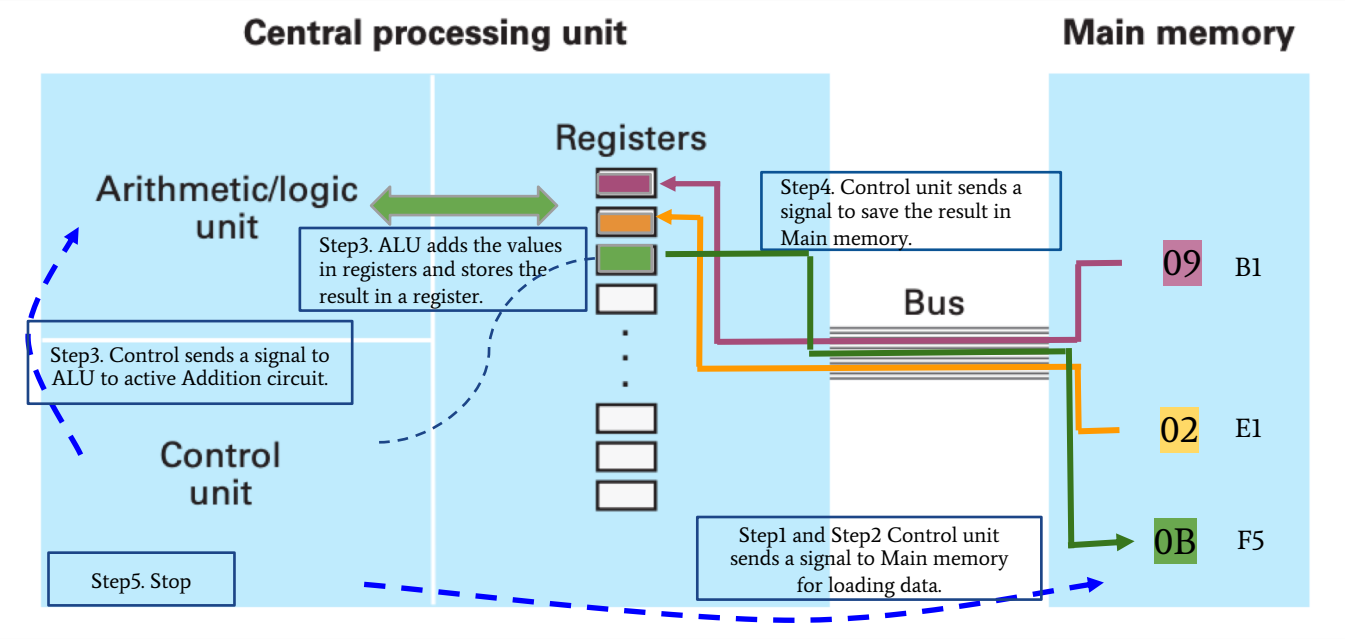


Step1	1	0	B	1
Step2	1	1	E	1
Step3	5	2	0	1
Step4	3	2	F	5
Step5	C	0	0	0

Halt

Translate it into Vole (overview)

Adding two numbers (numbers are hexadecimal)



Step1	1	0	B	1
Step2	1	1	E	1
Step3	5	2	0	1
Step4	3	2	F	5
Step5	C	0	0	0

In each step, the machine loads an instruction from the memory and executes, and so on until it loads the halt. This is the cycle of execution.

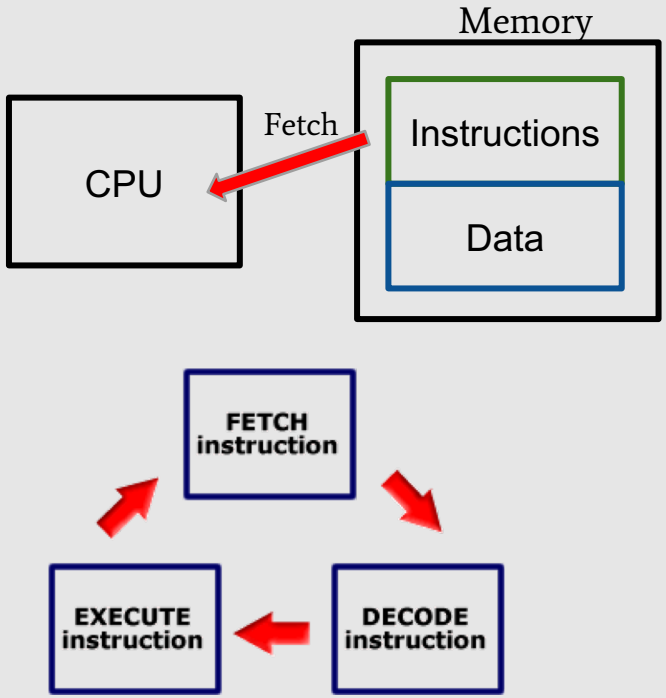
The cycle of execution

In a stored-program computer, instructions are stored in memory.

- A computer executes a program stored in its memory by fetching the instructions from memory into the CPU, decoding it and executing it.
  - Fetch → Decode → Execute → Fetch .....

e.g., In the program "adding two numbers", we put Step 1 to Step 5 one by one in memory. It takes 5 cycles to finish the program.

To execute the program properly, the computer should know where to fetch the instruction for the next cycle when it finishes the current one.



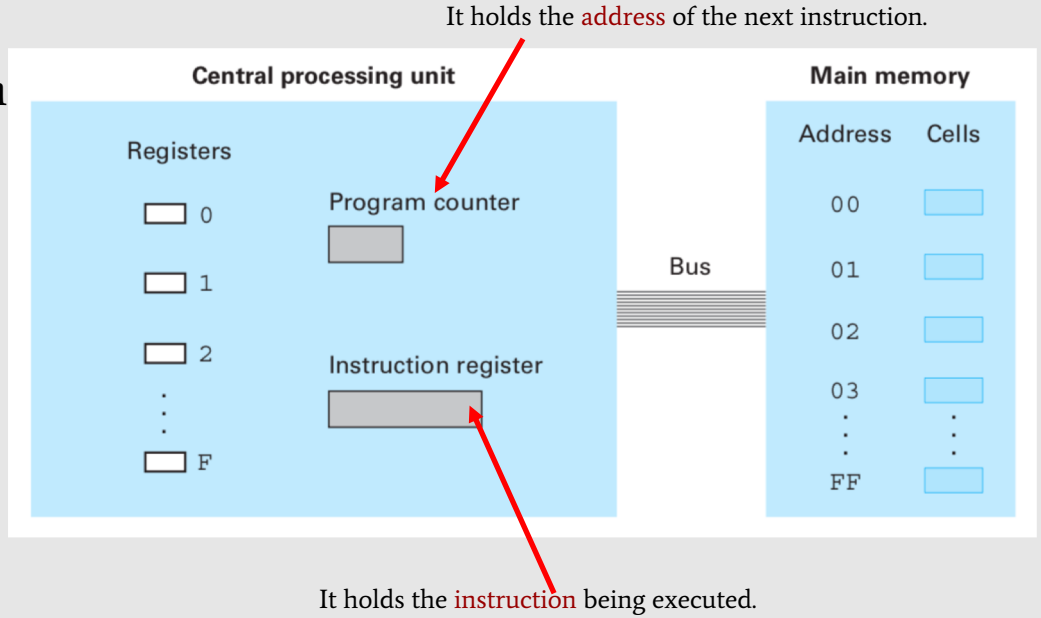
The circle of execution

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, the CPU requests main memory to provide it with the instruction that is stored at the address indicated by program counter.

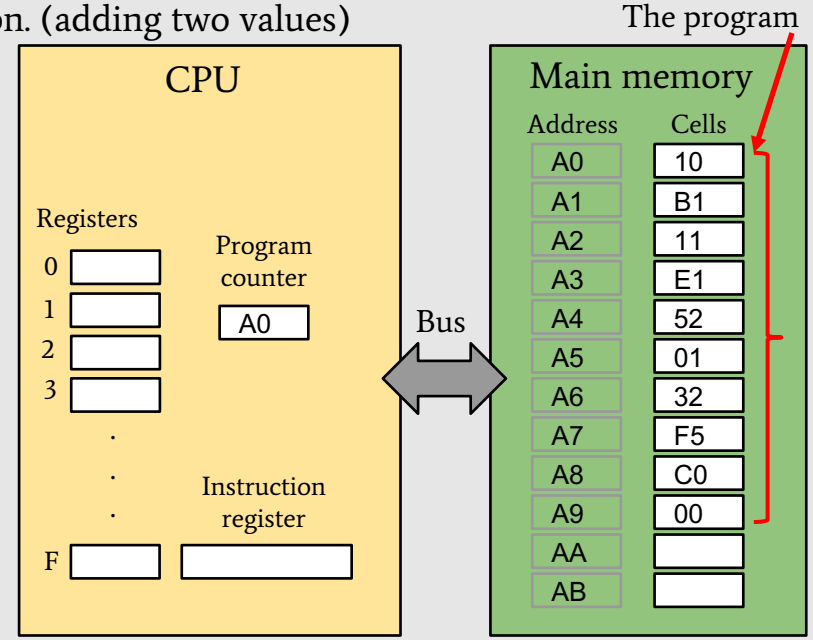
The instruction is loaded into the instruction register where the CPU decodes the instruction (e.g., breaking the operand field according to the op-code).



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

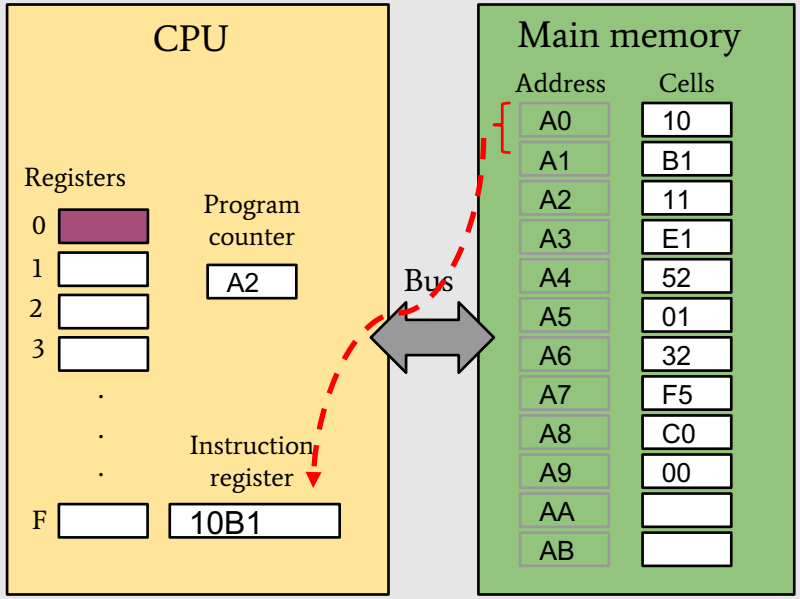


The first instruction is loaded from A0, because Program counter is set to be A0 at the beginning.

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

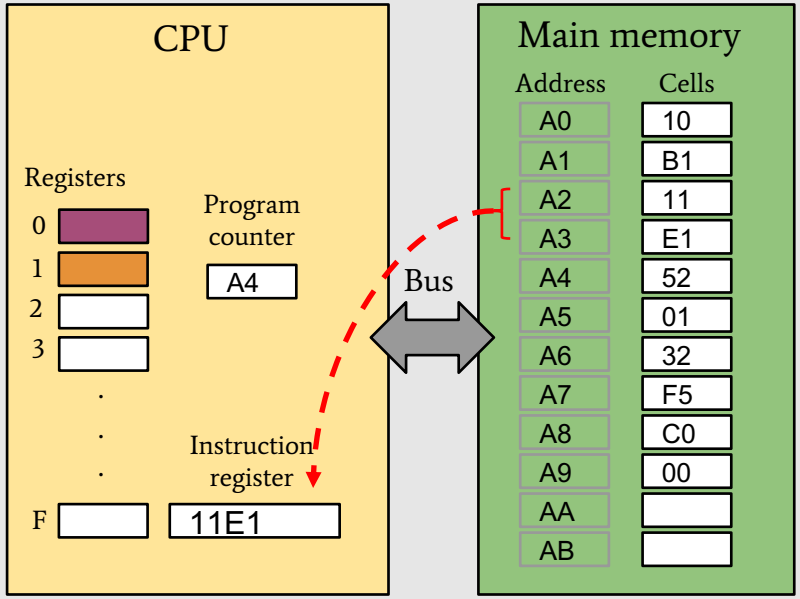


Loading from A0: Since each instruction in Vole has 2 bytes long, the CPU fetches two memory cells from the main memory and **increments** the **PC** by **2** so that the next fetch will start from A2.

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

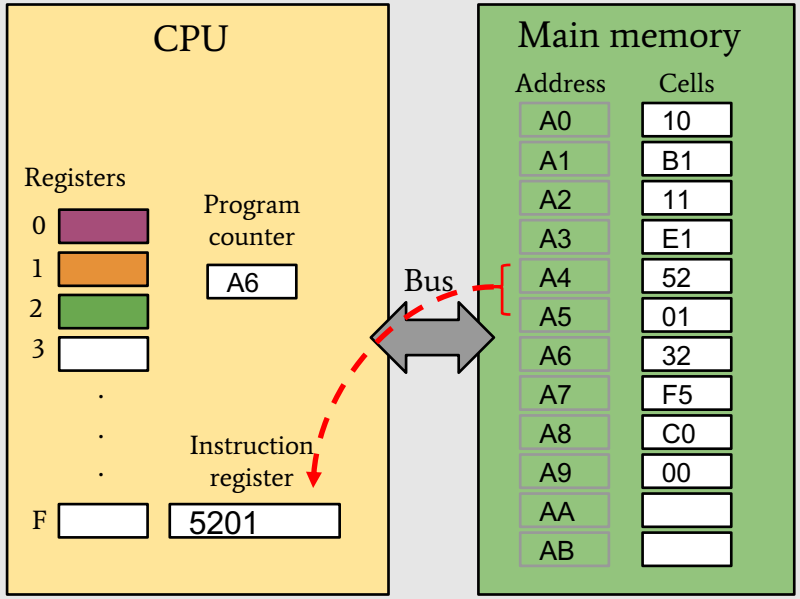


Loading from A2, incrementing PC by two to A4

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

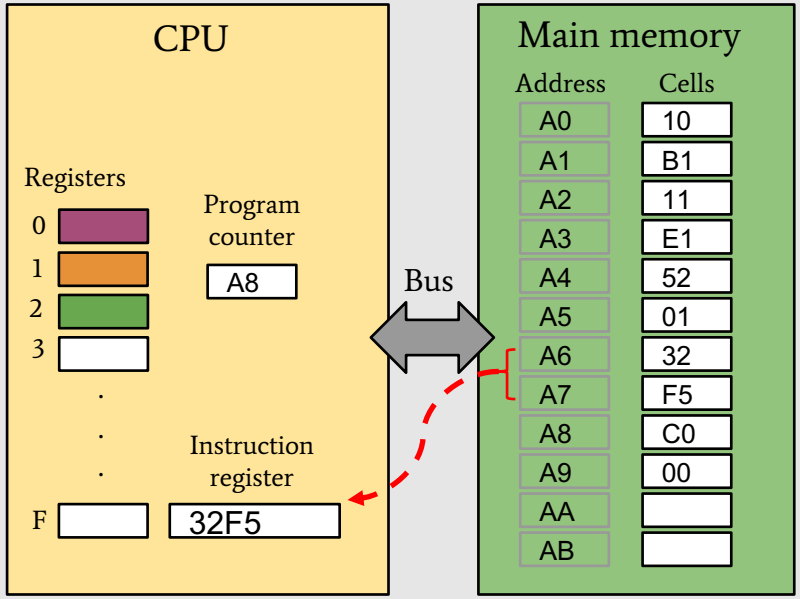


Loading from A4, incrementing PC by two to A6.

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

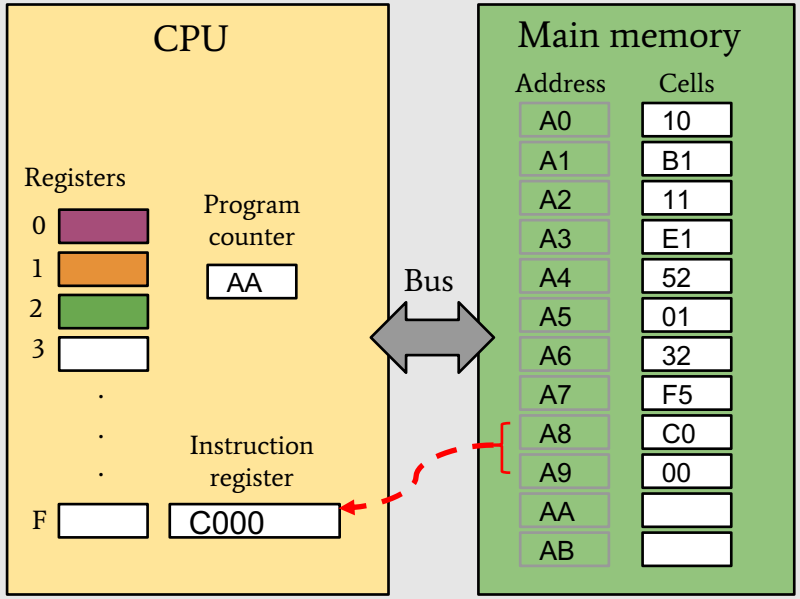


Loading from A6, incrementing PC by two to A8

Example of execution

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

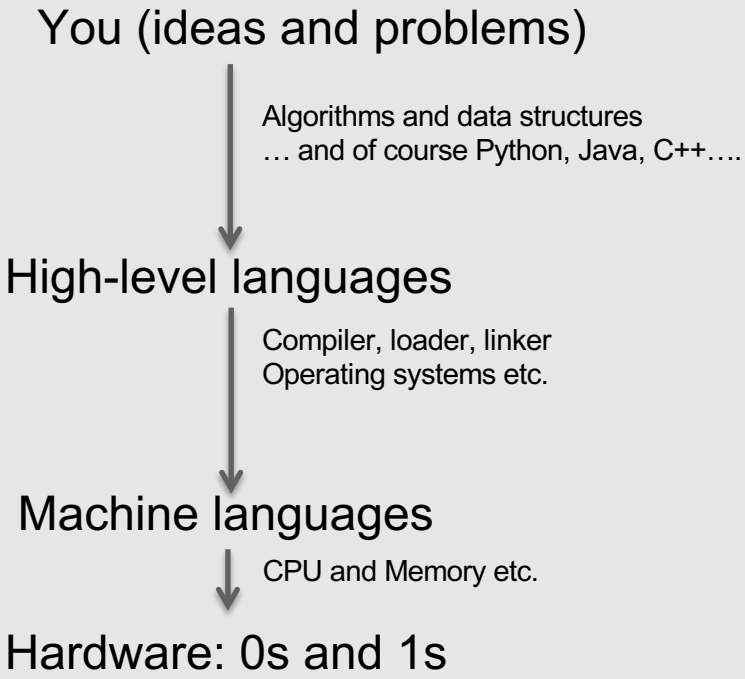
Encoded instructions	Translation
10B1	Load register 0 from address B1
11E1	Load register 1 from address E1
5201	Add contents of register 0 and 1; leave the result in register 2
32F5	Store the contents of register 2 at address F5
C000	Halt



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

Programming

Layers and abstractions



Barbara Liskov (MIT)
The power of abstraction
Turning award speech
http://amturing.acm.org/vp/liskov_1108679.cfm

Finding the max in a list

mymax.c mymax.py

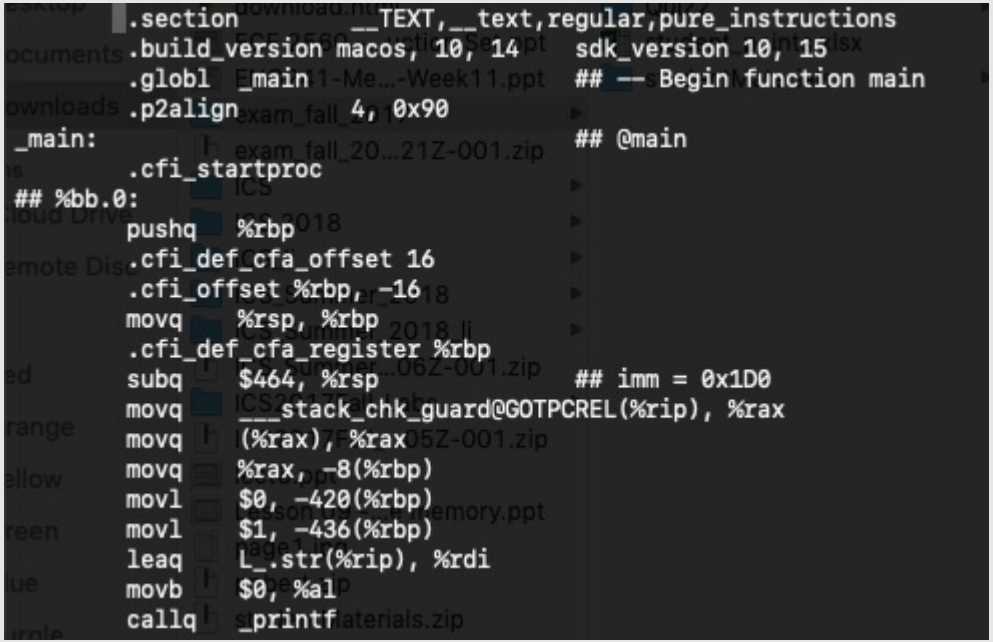
```
1
2  array_size = int(input('how many? '))
3  array = []
4  for i in range(array_size):
5      array.append(int(input('> ')))
6
7
8  my_max = array[0]
9  location = 0
10 for i in range(1, array_size):
11     if array[i] > my_max:
12         my_max = array[i]
13         location = i
14
15 print('max is: ', my_max, 'at ', location + 1)
```

Do

mymax.c mymax.py

```c
#include <stdio.h>

int main()
{
  int array[100], my_max, array_size, i, location = 1;

  printf("How many? \n");
  scanf("%d", &array_size);

  for (i = 0; i < array_size; i++)
    scanf("%d", &array[c]);

  my_max = array[0];
  location = 0
  for (i = 1; i < array_size; i++)
  {
    if (array[c] > my_max)
    {
       my_max  = array[i];
       location = i;
    }
  }

  printf("max is %d at location %d\n", my_max, location + 1);
}
```

Do it in assembly code: the job of the compiler



```
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 10, 14    sdk_version 10, 15
        .globl  _main                   ## -- Begin function main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $464, %rsp              ## imm = 0x1D0
        movq    ___stack_chk_guard@GOTPCREL(%rip), %rax
        movq    (%rax), %rax
        movq    %rax, -8(%rbp)
        movl    $0, -420(%rbp)
        movl    $1, -436(%rbp)
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
```

- Pushq, subq, movq etc. are *instructions*

Programming languages

Two major types of programming languages

- Low Level Languages:
  - *Machine language*
    The only language that is directly understood by the computer
  - *Assembly language*
    Need a translator program(assembler) to be translated into machine language

- High Level Languages

  C C++ Java Python

  need to be converted to machine language for the computer to understand

Programming languages

- Compiler is a program translator that translates the source code written in a higher level language to a lower level language.

  It scans the entire program first and then translates it into machine code.

- An interpreter is another type of program translator used for translating higher level language into machine language.

  It takes one statement at a time, translate it into machine language and immediately execute it. Translation and execution are carried out for each statement.

Programming languages

- **Compiled language**

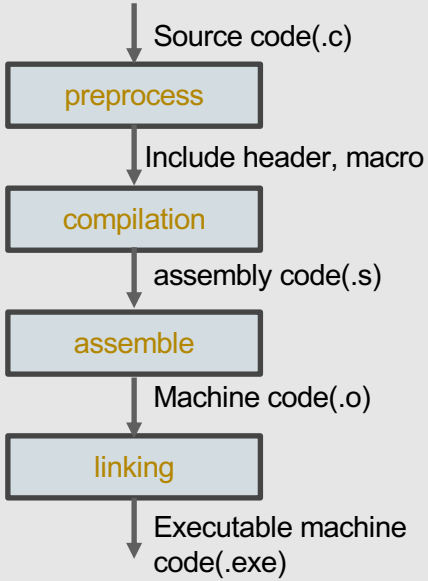
  Compiled languages need a “build” step - they need to be manually compiled first. You need to “rebuild” the program every time you need to make a change.

  C, C++, Java

- **Interpreted Languages**

  Interpreters will run through a program line by line and execute each command

  Python, JavaScript



Interpretation VS Compilation

	Interpretation	Compilation
How it treats input “x+2”	Compute x+2	Generate a program that computes x+2
When it happens	During execution	Before execution
What it complicates/slows	Program execution	Program development
Decisions made at	Run time	Compile time

Major choice we’ll see repeatedly: do it at compile time or at run time?
Which is faster ?

Moving on....

BBC History of Computers Clip #6
BBC History of Computers Clip #7

Evolution of computer interface

What are some of the examples today?

Evolution of computer interface

What are some of the examples today?

- Speech: even on your phone
- Vision: Google glass
- Gesture: Xbox
- Gets harder, but also more exciting/interesting (e.g. automatic driving car)

The KISS principle

- There are two versions
  - **K**eep **I**t **S**imple and **S**tupid (Apple)
  - **K**eep **I**t **S**imple, but not **S**impler (Einstein)

- Both are important
- More arts than science

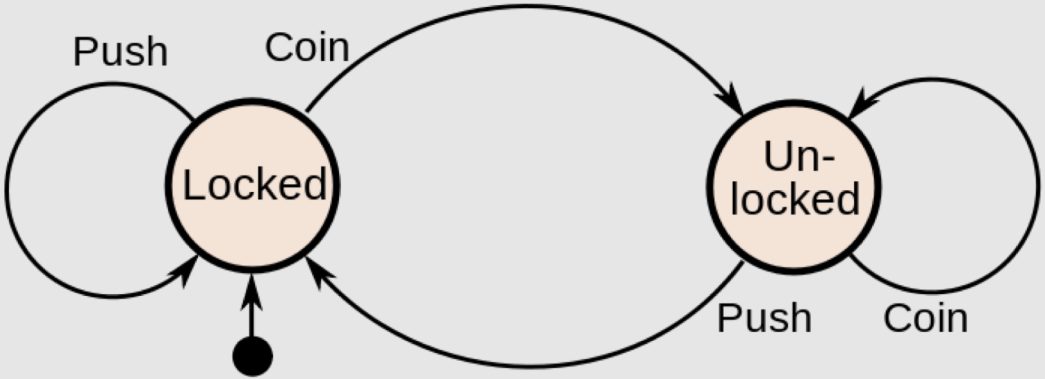
State machine (Automaton)

State space and (finite) state machine

The FSM can change from one state to another in response to some external input; the change from one state to another is called a *transition*.
An FSM is defined by a list of its states, its initial state, and the conditions for each transition

S is a finite, non-empty set of states
s_0 is an initial state, an element of S.
Σ is the input alphabet (a finite, non-empty set of symbols).
$\delta(current\ state, input)$ is the state-transition function

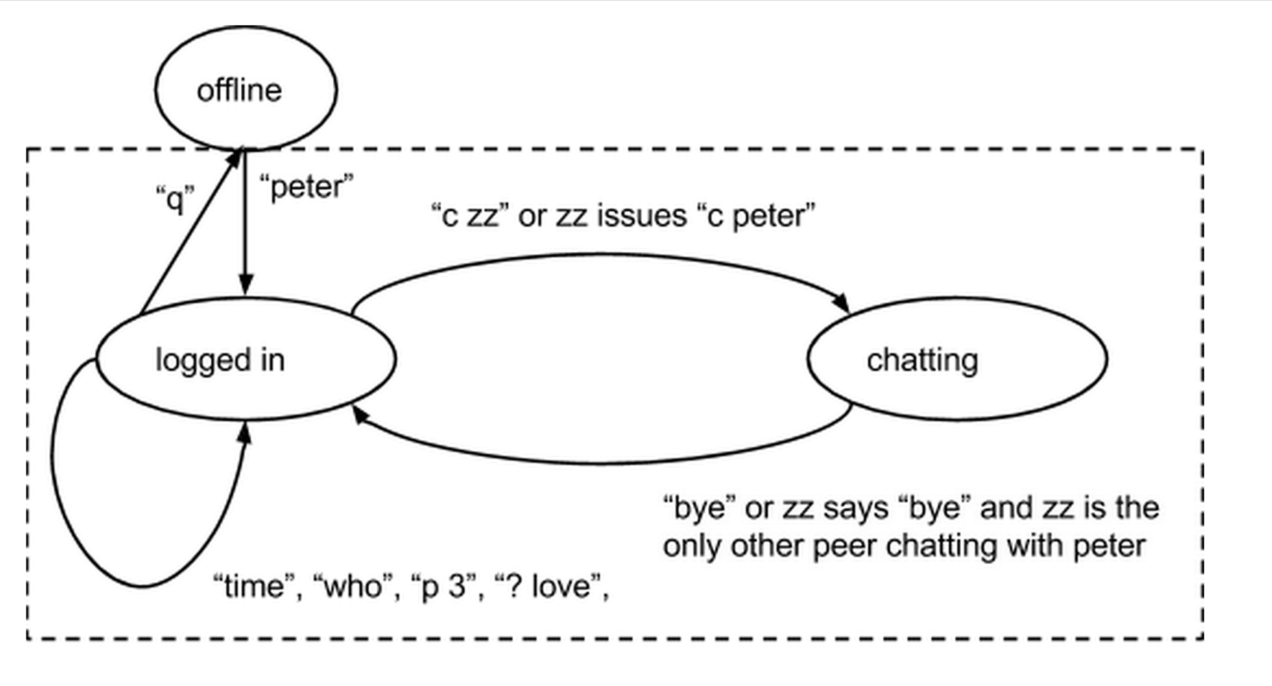
Finite State Machine: turnstile



State: $S = \{locked, unlocked\}$
Inputs: $\Sigma = \{'insert\ coin',\ 'push'\}$
Transition function: the table on the rig
Initial state: Locked

Current state	Input	New state
locked	Push	Locked
locked	coin	unlocked
Unlocked	coin	Unlocked
Unlocked	coin	Locked

Behind scene: state machine of the chat client



In-class programming exercise

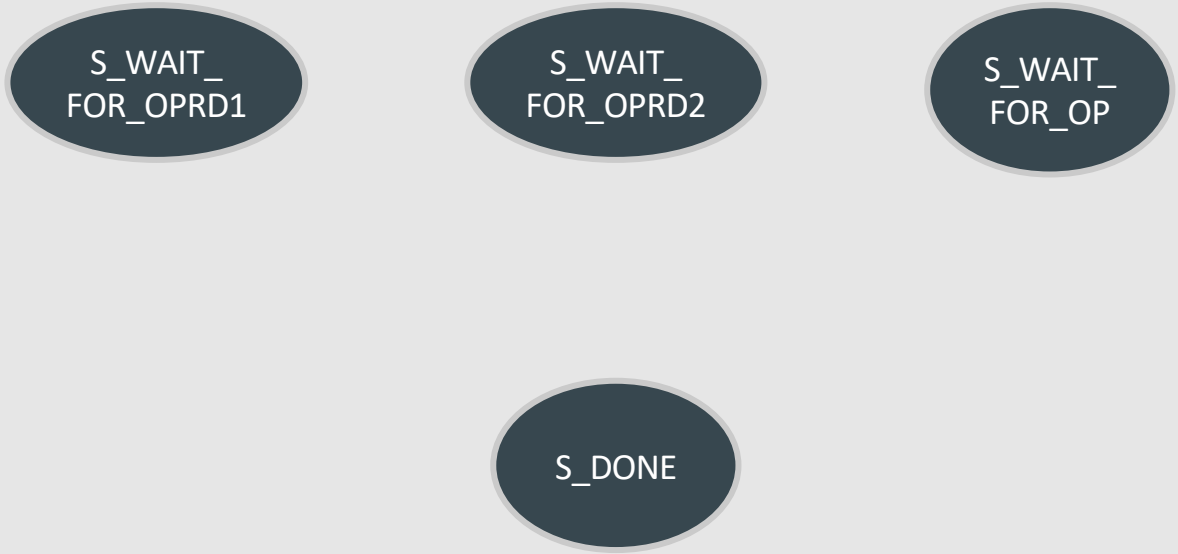
State machine

Now build a calculator

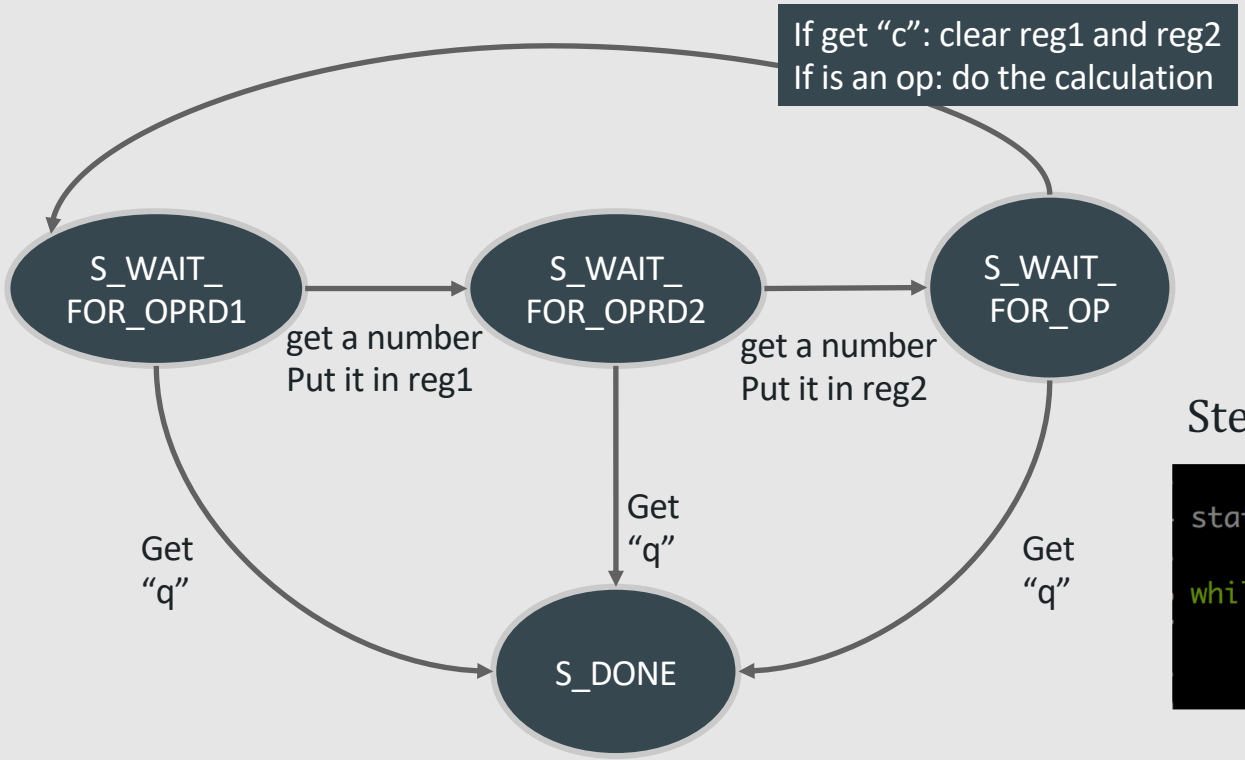
```
 9  S_WAIT_FOR_OPRD1 = 1
10  S_WAIT_FOR_OPRD2 = 2
11  S_WAIT_FOR_OP = 3
12  S_DONE = 4
13
14  OPERATOR_LIST = ('+', '-', '*', '/')
15  PROMPT = """The calculator expects inputs in the order:
16  1st operand, 2nd operand, operator
17  - Operators are integers!
18  - Input 'q' will quit, at any point
19  - Input 'c' when expecting an operator will reset the calculator
20  Have fun!
21  ---------
22  """
```

Calculator state machine

Step 1: fill the logic (what should happen) on the transition arcs



Calculator state machine



Step 2: code it up

```
state = S_WAIT_FOR_OPRD1

while state != S_DONE:
    # your code here
    ...
```