Machine Language

Dr. Ren Jianfeng

A self-introduction

- Dr. Ren Jianfeng
  - Assistant Professor, **UNNC**.
  - Office: PMB445
  - Office hour: 1-3 pm Tuesday.
- Education
  - Ph.D in Engineering, **Nanyang Technological University**, Singapore, 2015.
  - Master of Science in Signal Processing, **Nanyang Technological University**, Singapore, 2009.
  - Bachelor of Engineering, Electrical & Electronics Engineering, **National University of Singapore**, 2001.

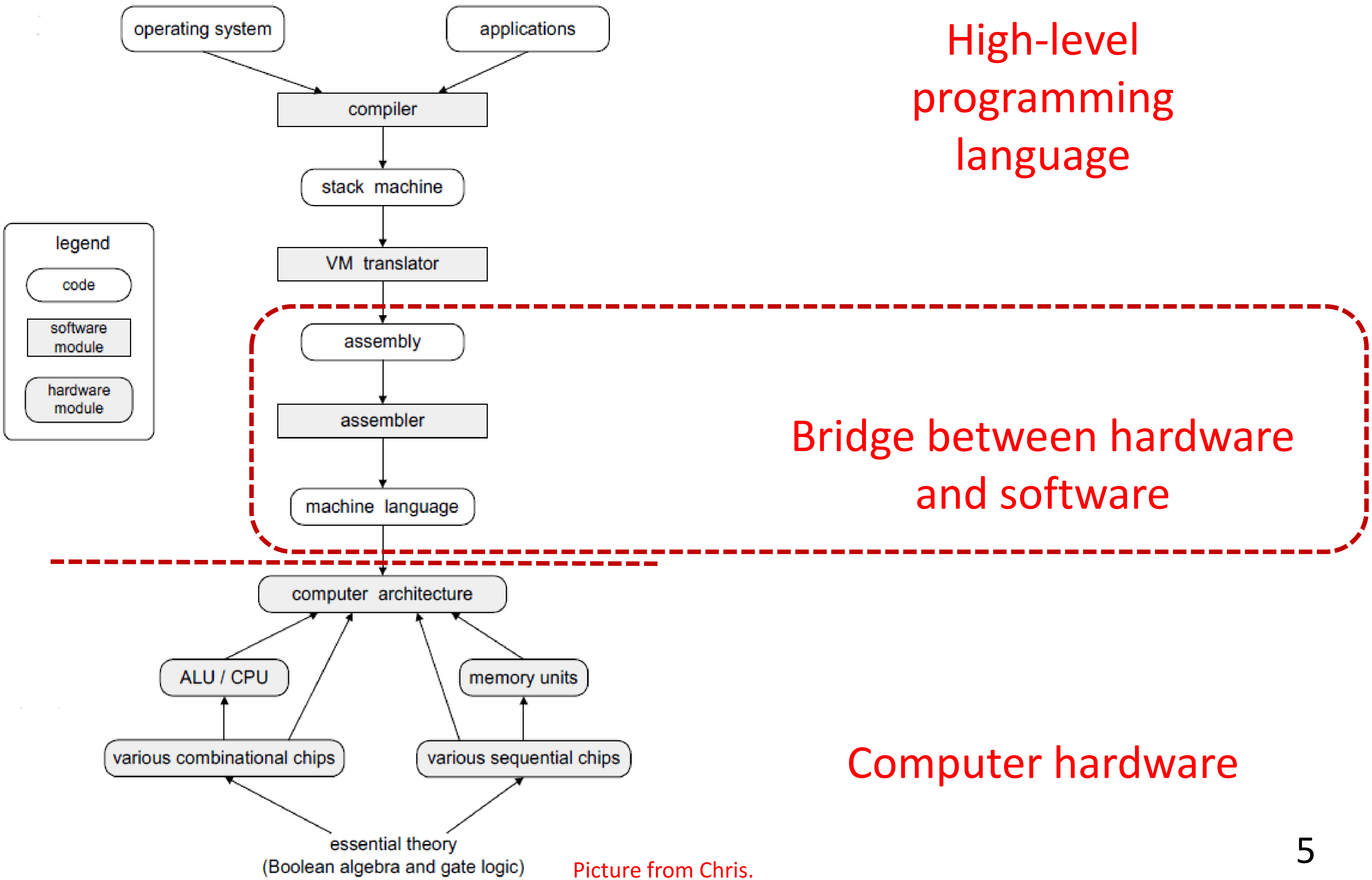
Research interests

- Image/video processing,
- Computer vision,
- Statistical pattern recognition,
- Deep learning,
- Human computer interaction,
- Radar target recognition.

Outlines

- Introduction to machine language
- Some basic operations
- Hack basics
- Hack assembly programming

Why learning machine language?



High-level programming language

Bridge between hardware and software

Computer hardware

Picture from Chris.

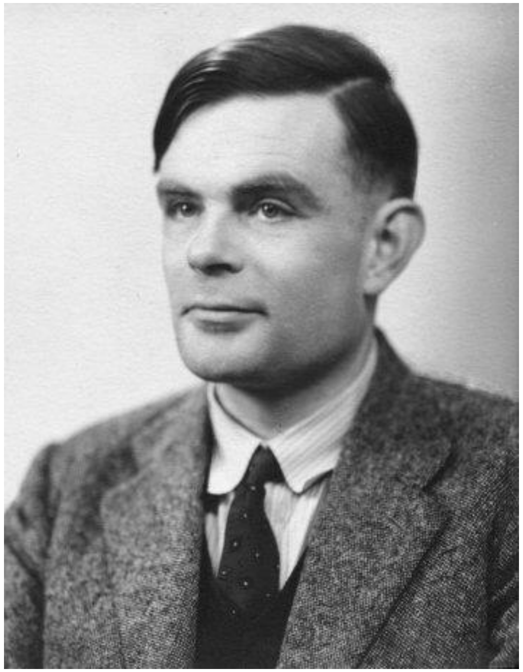
Computers are flexible

- Many **software** programs can run on the same **hardware**.

Universality

- Many **software** programs can run on the same **hardware**.

Theory



Alan Turing:

Universal Turing Machine

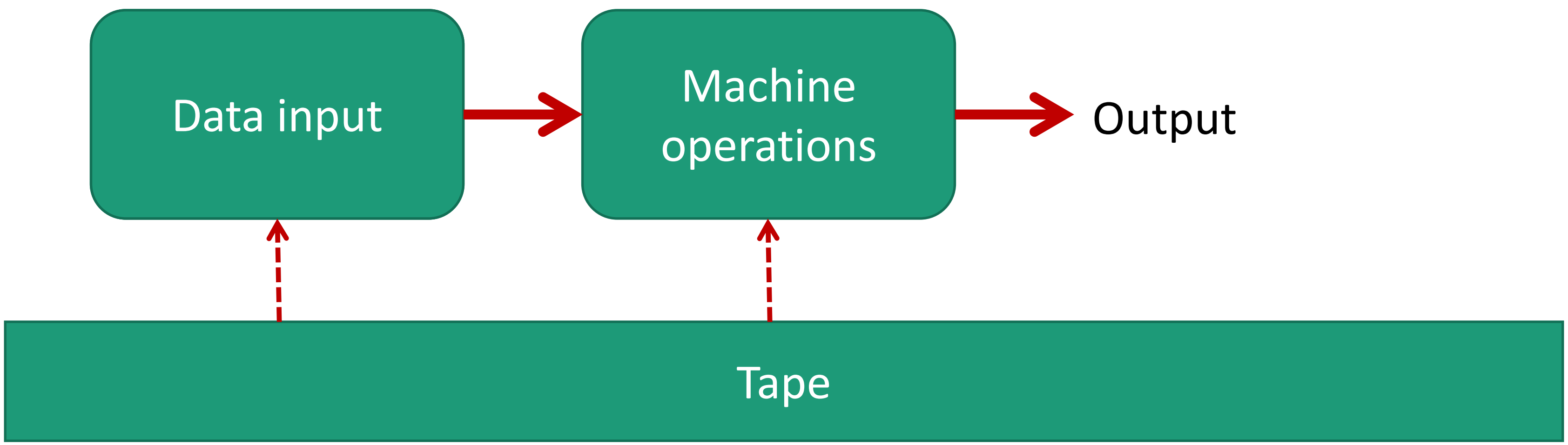
Practice

John Von Nuemann:

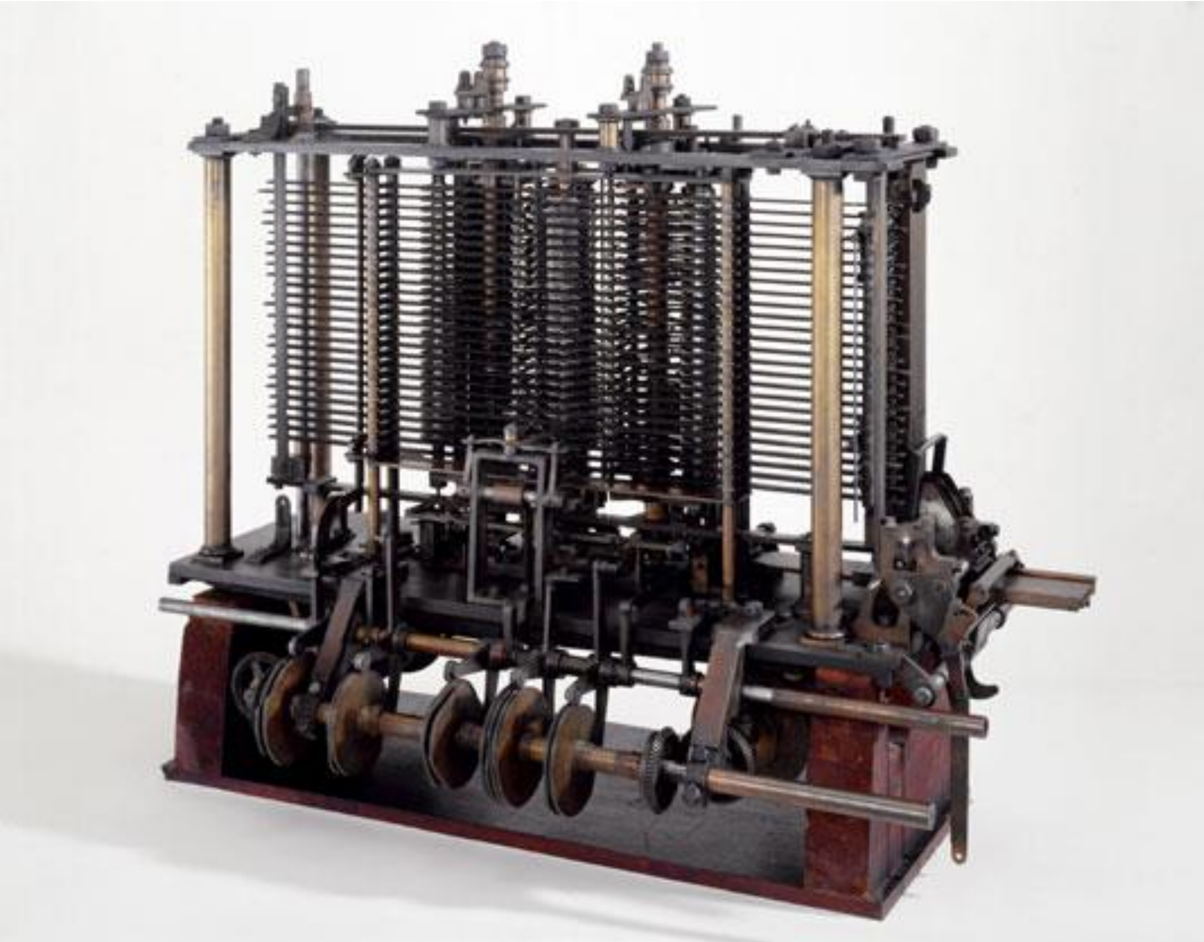
Stored Program Computer

Universal Turing machine

- A machine that can simulate an arbitrary machine operation on arbitrary input. (wikipedia)
  - Reading both the description of the machine to be simulated and the data input to the machine from its own tape.



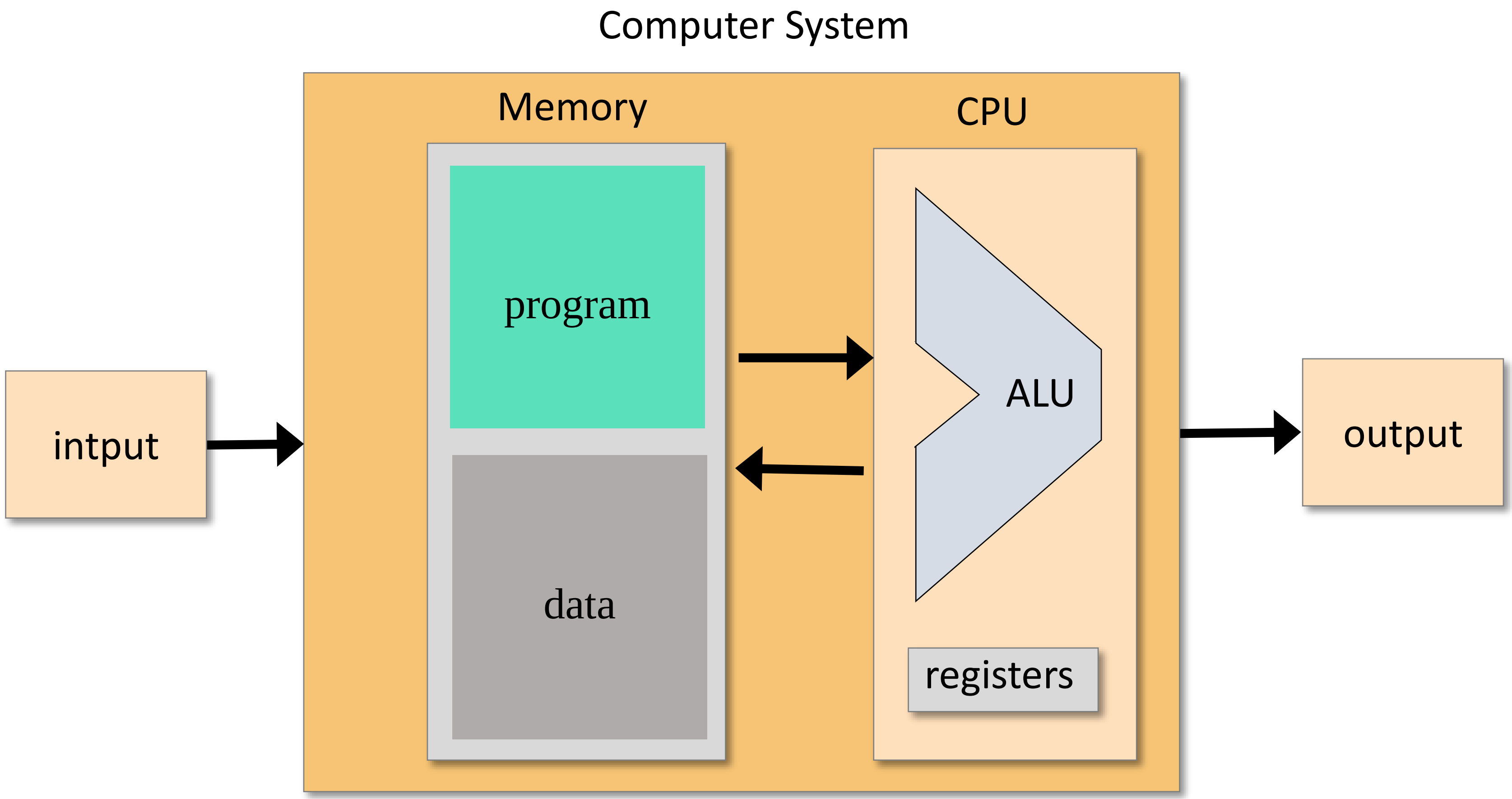
The first computer



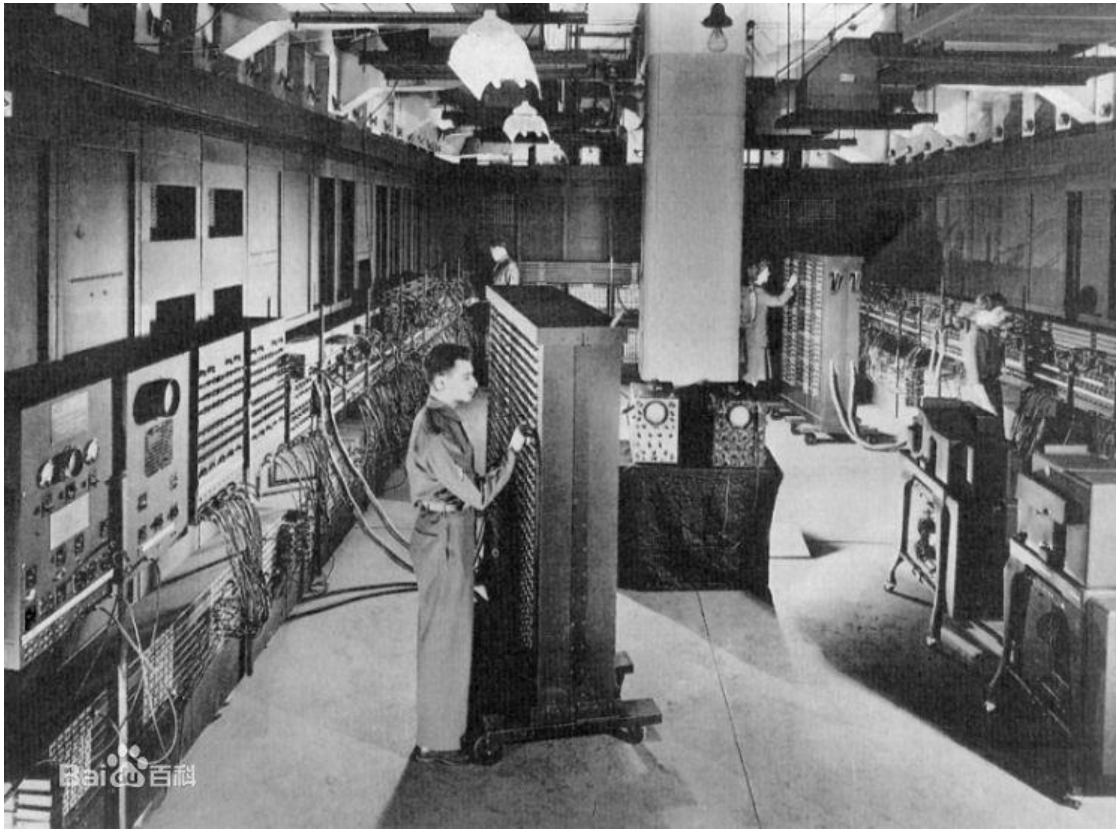
- Designed by Charles Babbage, in 1822.
- Powered by a steam engine.
- Use Punched Cards.

http://www.netprolive.com/computers.php

Stored program concept



ENIAC - first general-purpose computer

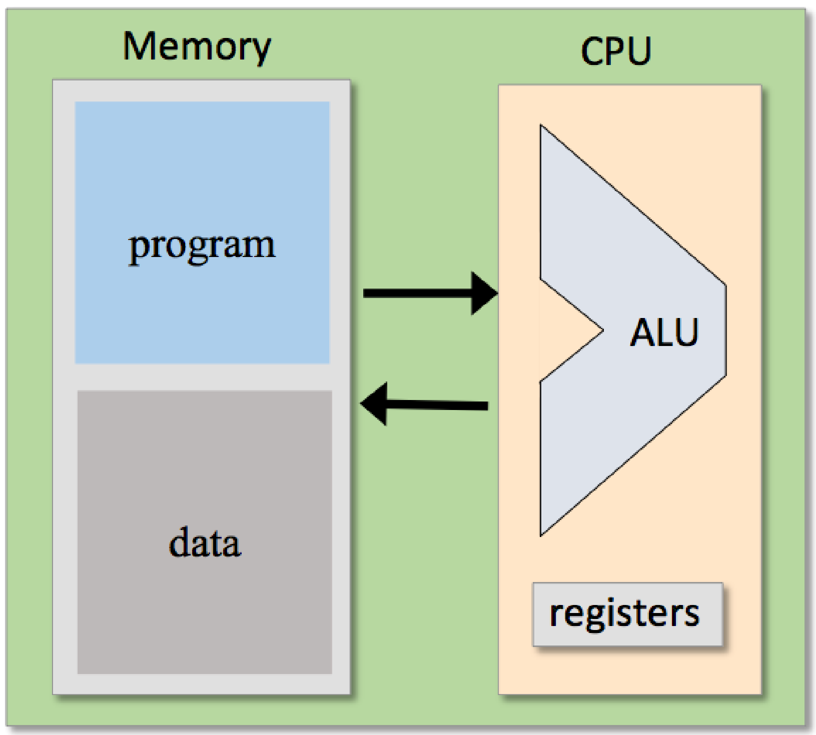


- Electronic Numerical Integrator And Computer (ENIAC).
- First Turing-complete computer.
- Announced in 1946.
- By Moore School of Electrical Engineering, University of Pennsylvania, US.

https://web.math.pmf.unizg.hr/~nela/prteme/ENIAC.htm

An informal definition: machine language

- A *machine language* can be viewed as an agreed-upon formalism, designed to manipulate a *memory* using a *processor* and a set of *registers*. (Nisan & Schocken)



List of machine languages

- ARM: 16-bit, 32-bit, 64-bit
- DEC: 12-bit, 16-bit, 18-bit, 32-bit, 36-bit, 64-bit
- Intel: 8008, 8080, 8085, Zilog Z80.
- X86: 16-bit x86, IA-32, x86-64
- IBM: 305, 650, 701, …
- **MIPS**
- Motorola 6800, 68000 family
- **Hack assembly**
- …

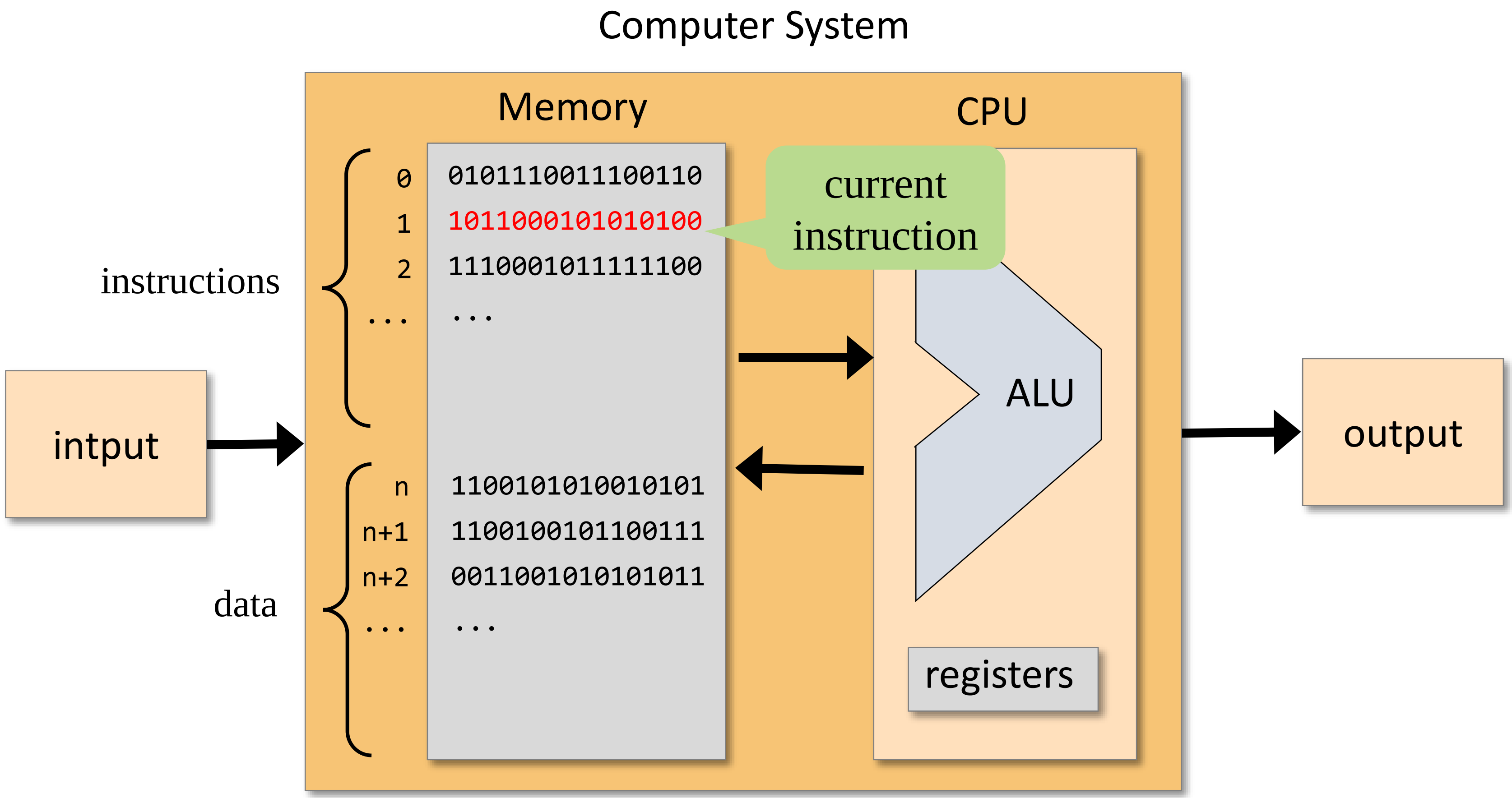
Machine language is hardware-dependent.

https://en.wikipedia.org/wiki/List_of_programming_languages_by_type#Machine_languages

Machine language at a glance

- Processor (CPU)
  - ALU, memory access, control (branching).
  - E.g. add R1, R2, R3 // R1 ← R2 + R3.
- Memory
  - Collection of hardware devices that store data and instructions in a computer.
  - E.g. load R1, 67 // R1←Memory[67].
  - Slow access.
- Register
  - High-speed local memory.

Machine language



<u>Handling instructions:</u>

- `1011` means “addition” — operation
- `000101010100` means “operate on memory address 340” — addressing
- Next we have to execute the instruction at address 2 — control

Compilation

high-level program

```
while (n < 100) {
    sum += arr[i];
    n++
}
```

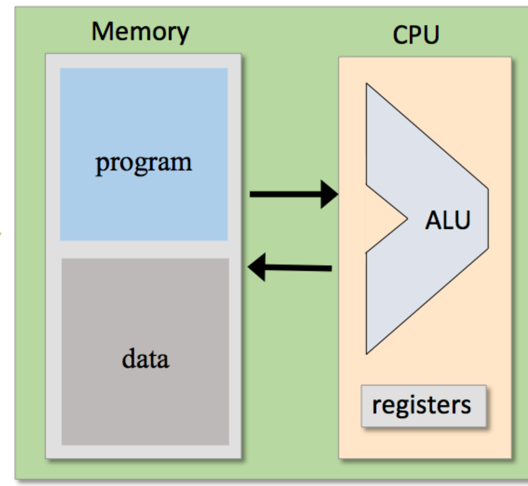
compile

machine language

```
0101111100111100
1010101010101010
1101011010101010
1001101010010101
1101010010101010
1110010100100100
0011001010010101
1100100111000100
1100011001100101
0010111001010101
...
```

load and execute

hardware



Virtual machine and **assembly language** in between. We will come back to them later.

Machine language

Binary instruction:

`1011 0001 10000001`

add 1 Mem[129]

- Difficult to understand.

Assembly language:

```
add 1, Mem[129]
```

Mem[129] ←Mem[129] + 1

Friendlier version: `index` stands for `Mem[129]`

```
add 1, index
```

- Symbolic machine language instructions.
- Much easier to understand,
- Use assembler to translate assembly language to binary instructions.

Assembler

Assembly Language

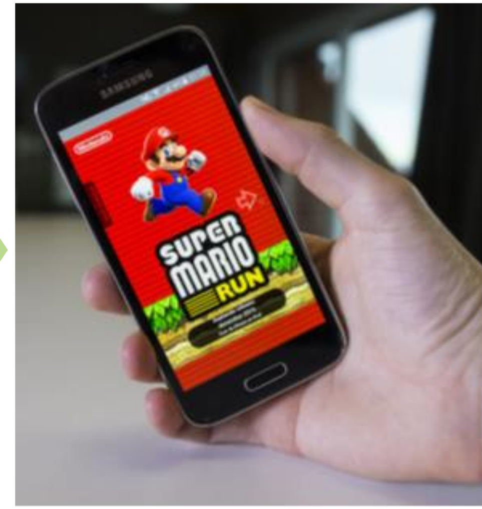
```
   @i
   M=1  // i = 1
   @sum
   M=0  // sum = 0
(LOOP)
   // if i>RAM[0],
   // GOTO WRITE
   @i
   D=M
   @R0
   D=D-M
   @WRITE
   D;JGT
   ...  // Etc.
```

assembler

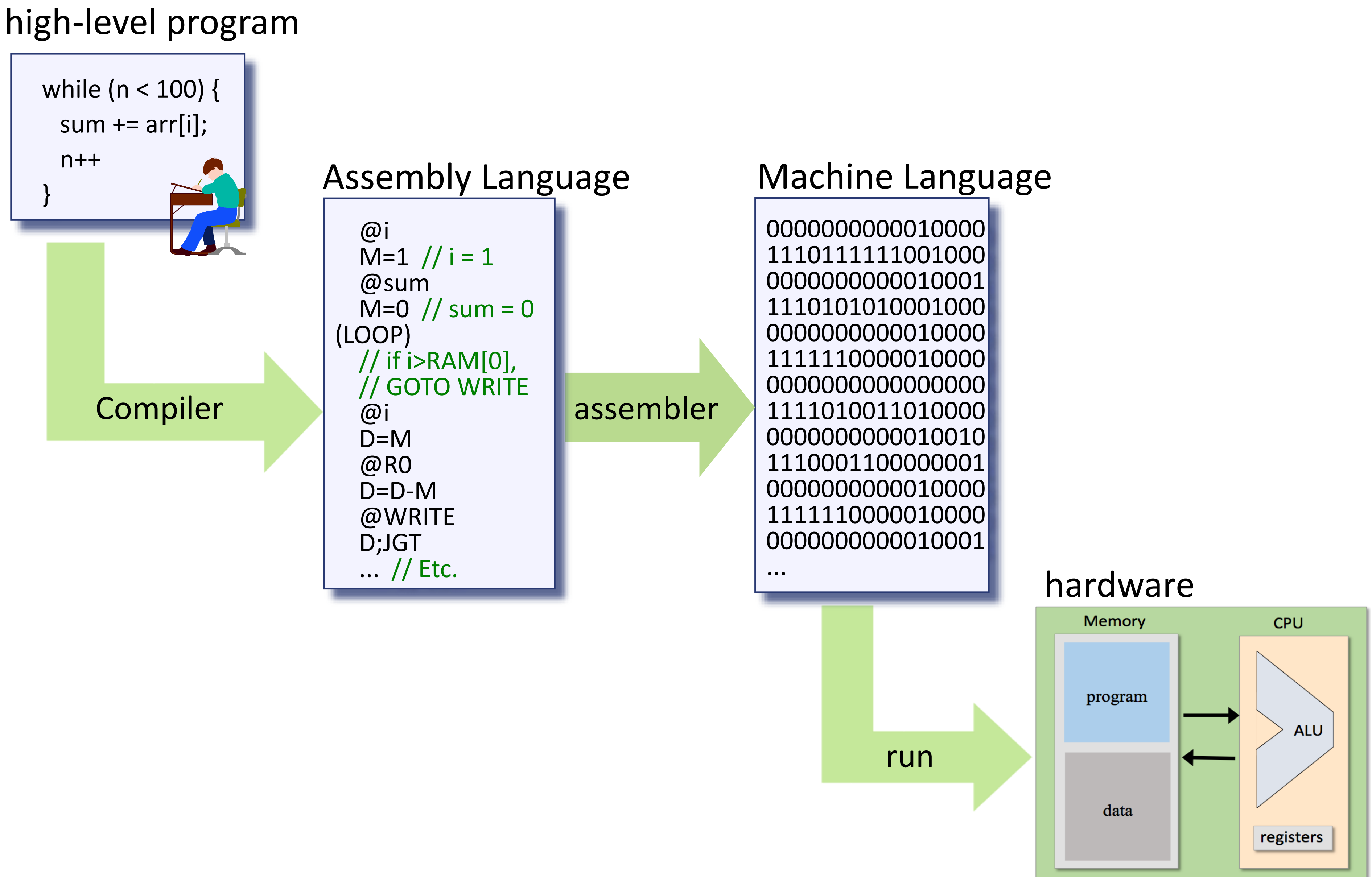
Machine Language

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
```

run



Recap



high-level program
while (n < 100) {
sum += arr[i];
n++
}
Compiler
Assembly Language
@i
M=1 // i = 1
@sum
M=0 // sum = 0
(LOOP)
// if i>RAM[0],
// GOTO WRITE
@i
D=M
@R0
D=D-M
@WRITE
D;JGT
... // Etc.
assembler
Machine Language
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
...
run
hardware
Memory
program
data
CPU
ALU
registers

Outlines

- Introduction to machine language
- Some basic operations
  - Arithmetic and logic operations
  - Memory access
  - Flow control
- Hack basics
- Hack assembly programming

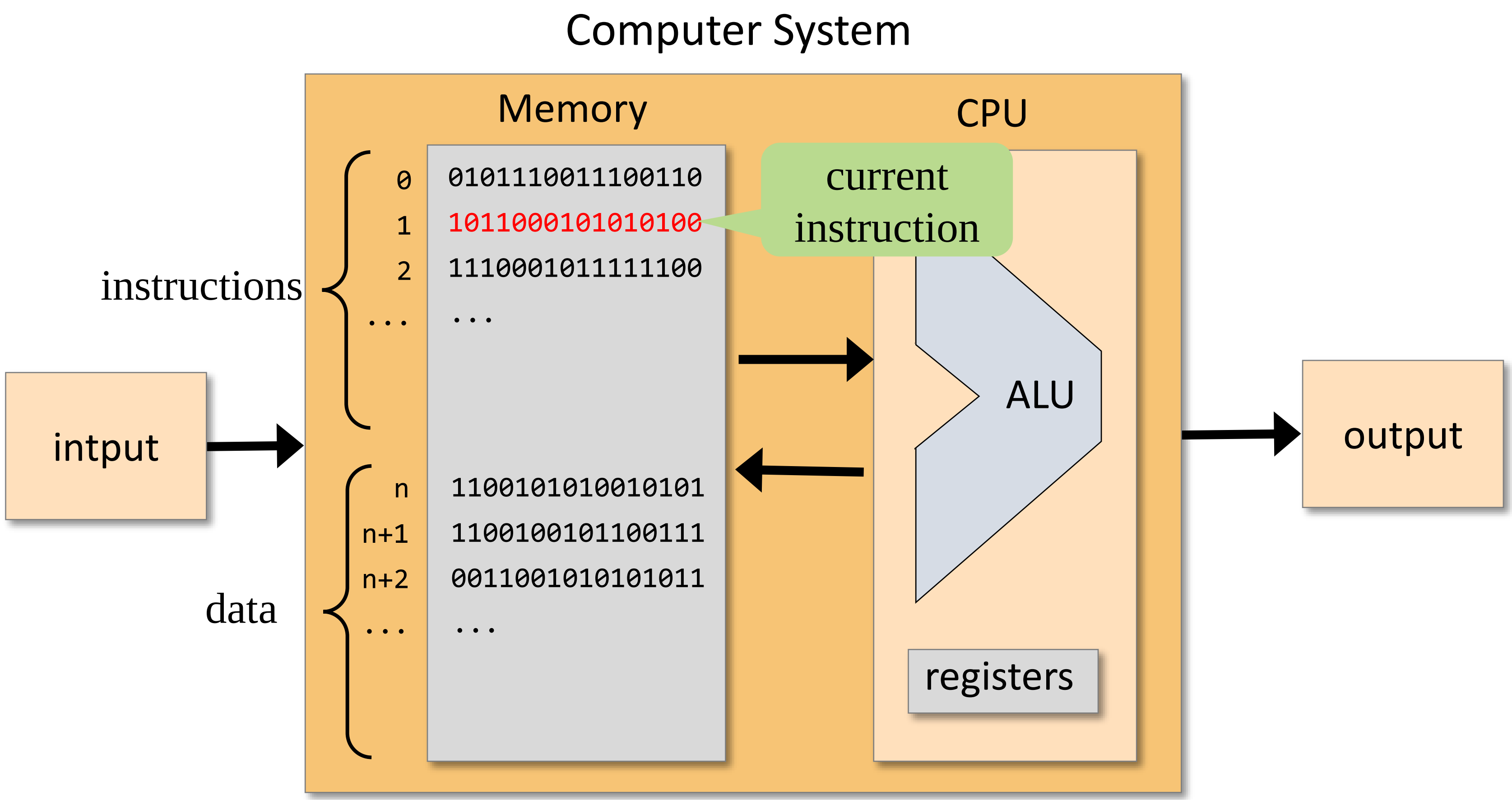
Arithmetic operations

- Addition/substraction
  - ADD R1, R2, R3 // R1 $\leftarrow$ R2 + R3, where R1, R2, // R3 are registers.
  - ADD R1, R2, index // R1 $\leftarrow$ R2 + index, where index // stands for the value of the // memory pointed at by the // user-defined label index. // e.g. index is RAM[129].

Logic operations

- Basic boolean operations:
  - ➢ Bitwise negation //0 $\longrightarrow$ 1, or 1 $\longrightarrow$ 0.
  - ➢ Bit shifting //00010111 left-shift by 2: 01011100
  - ➢ Bitwise And, Or, etc.
- Example:
  - ➢ AND R1, R1, R2 //R1 $\longleftarrow$ bitwise And of R1 and R2.

Memory access

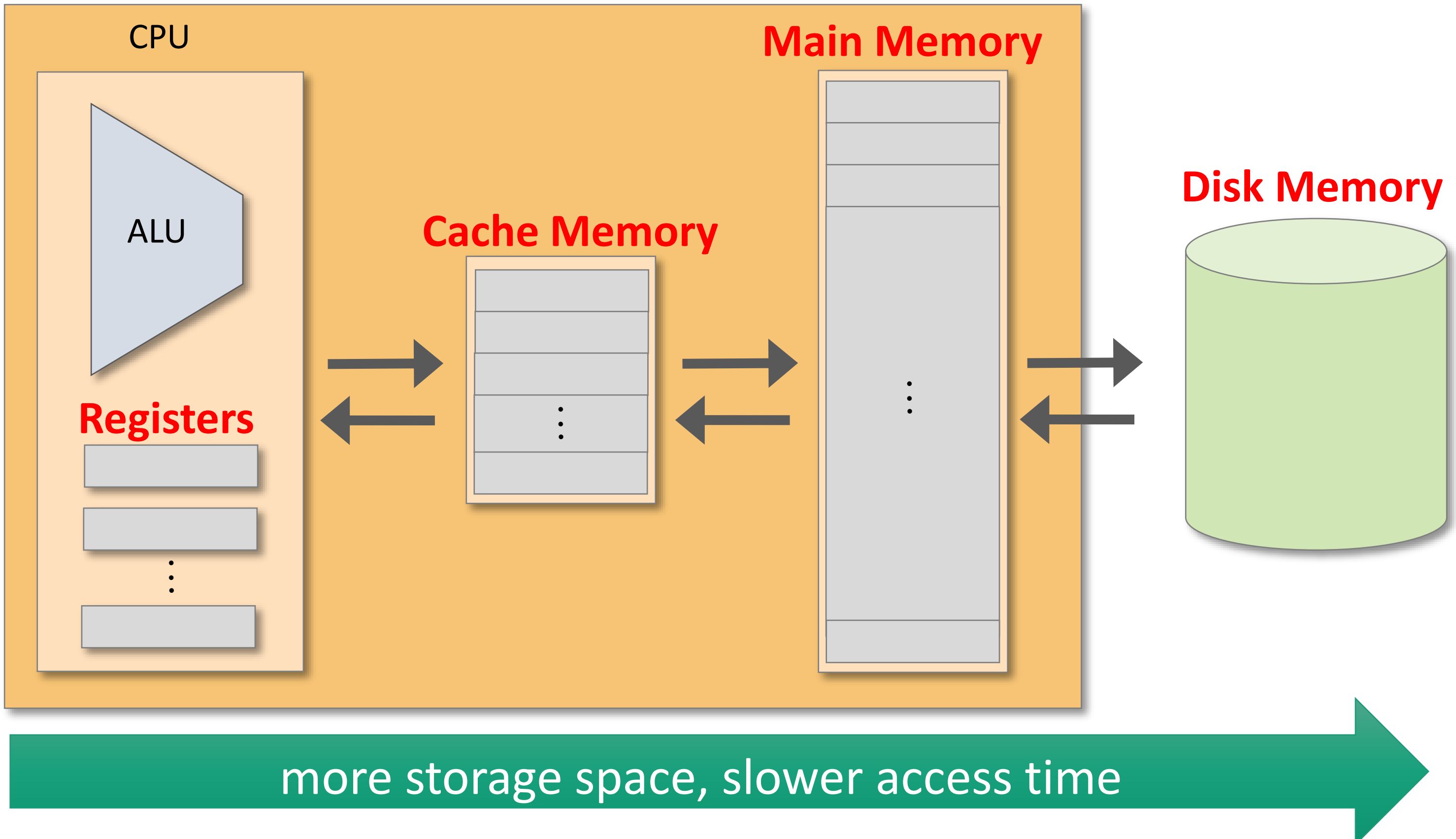


Which data the instruction should operate?

Check memory access address.

Memory hierarchy

- Accessing a memory location is expensive:
  - Need to supply a **long address**
  - Memory to CPU: **take time**
- Solution: memory hierarchy:



Memory hierarchy

Type	Description	Typical storage	Typical speed
CPU Register	Quickly accessible memory location available to a CPU.	**48** 128-Byte registers, 6 kB	≤1 CPU cycle
CPU Cache	A hardware cache used by CPU to reduce the cost to access data from main memory.	Intel i7 (2008), 8 MB L3 cache	3~14 CPU cycles
Main memory	Random-access memory (RAM).	4~8 GB	240 CPU cycles
Disk memory	Harddisk.	500 GB, 1 TB	10~30 ms

E.g. for a 2.5GHz CPU, 1 CPU cycle ≈ 0.4 ns.

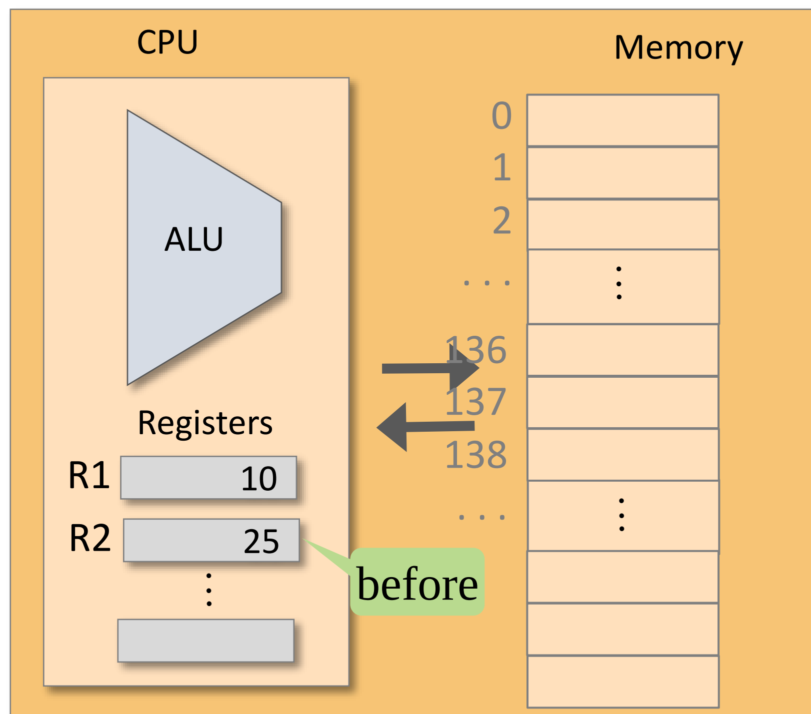
Registers

- The CPU typically contains a few, easily accessed, *registers.*
- They are the **central part** of the machine language.

Data registers:

add R1, R2 // R2 ← R1 + R2

	R1	R2
Before add	10	25
After add		



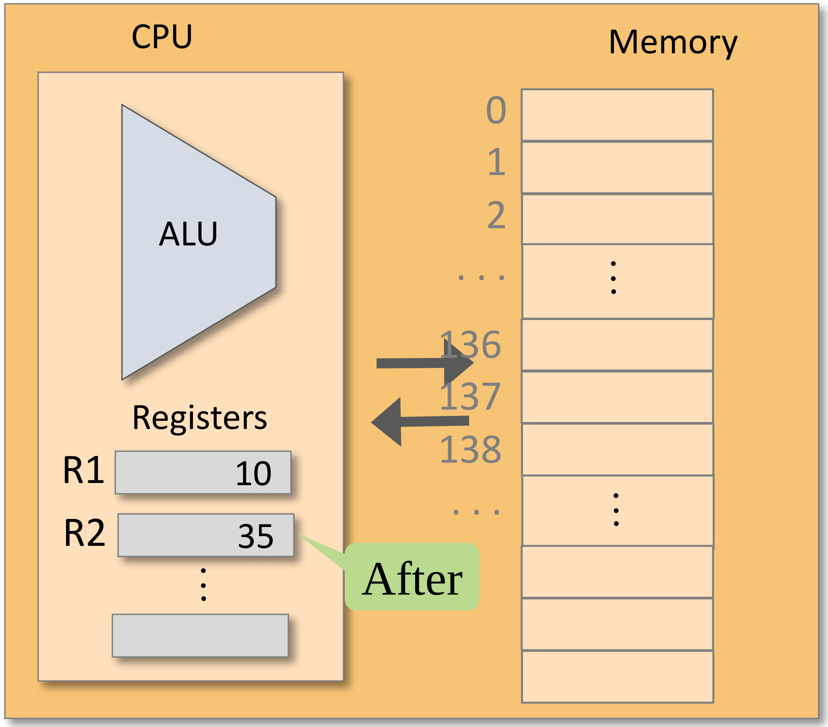
Registers

- The CPU typically contains a few, easily accessed, *registers.*
- They are the **central part** of the machine language.

Data registers:

add R1, R2 // R2 ← R1 + R2

	R1	**R2**
Before add	10	25
After add	10	35



Registers

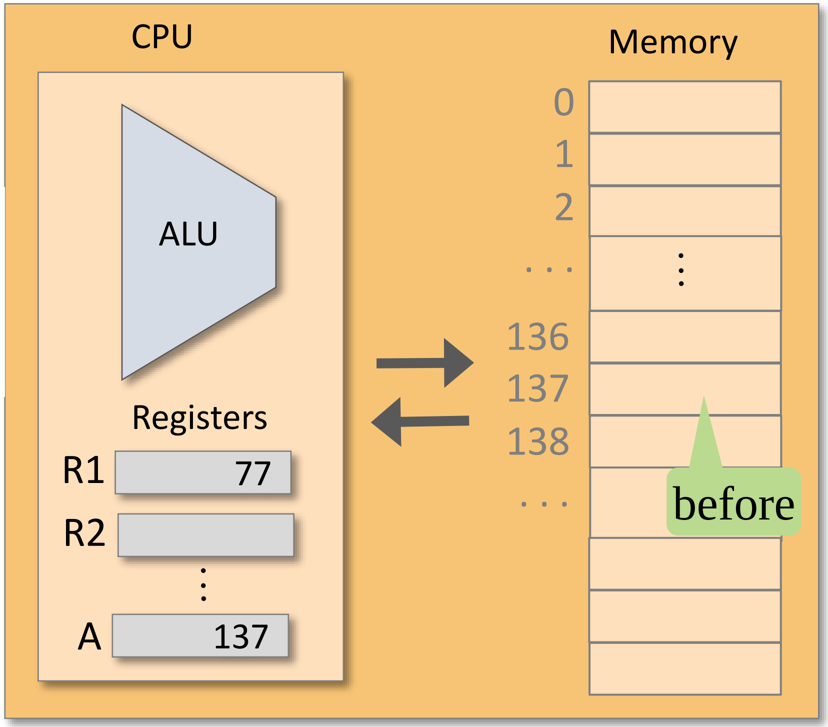
- The CPU typically contains a few, easily accessed, *registers.*
- They are the **central part** of the machine language.

Data registers:

add R1, R2 // R2 ← R1 + R2

Address registers:

store R1, @A // @A ← R1



Registers

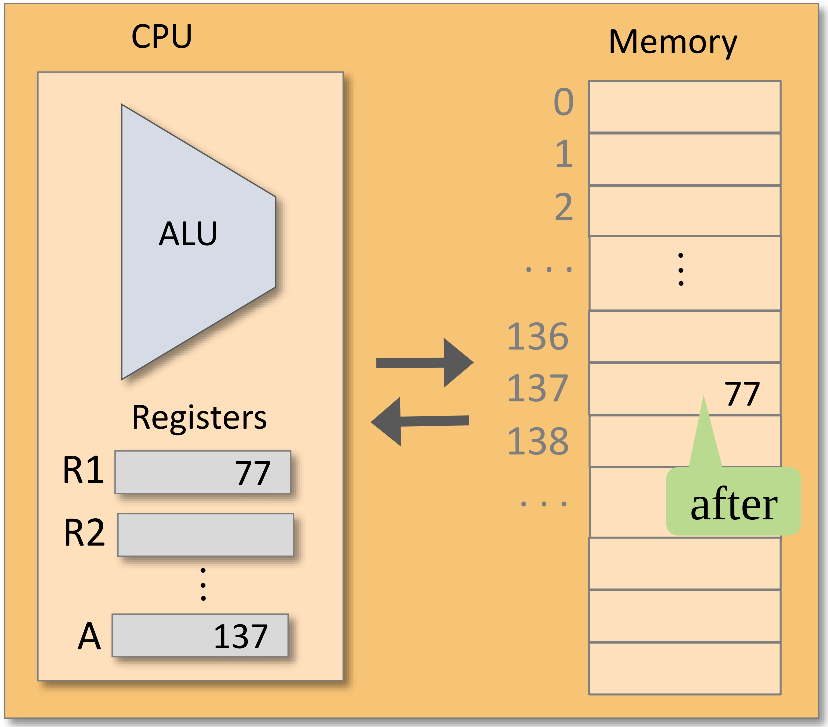
- The CPU typically contains a few, easily accessed, *registers.*
- They are the **central part** of the machine language.

Data registers:

add R1, R2 // R2 ← R1 + R2

Address registers:

store R1, @A // @A ← R1



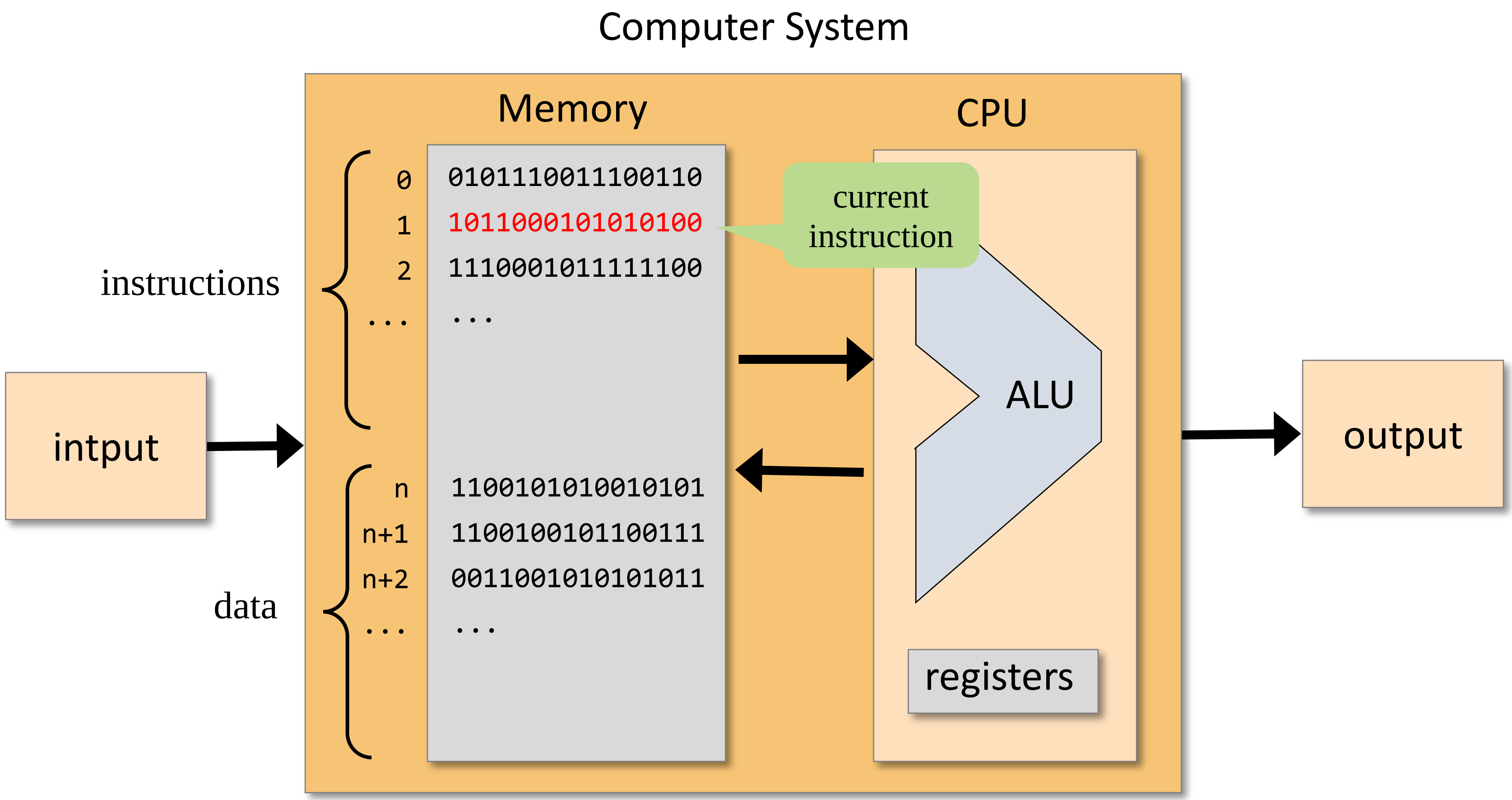
Addressing modes

- Register
  - ADD R1, R2 // R2 ← R2 + R1
  - Access data from a **register** R2.
- Direct
  - ADD R1, M[67] // Mem[67] ← Mem[67] + R1
  - LOAD R1, 67 // R1 ← Mem[67]
  - Access data from **fixed memory address** 67.
- Indirect
  - ADD R1, @A // Mem[A] ←Mem[A] + R1
  - Access data from **memory address specified by variable A**.
- Immediate
  - ADD 67, R1 // R1 ← R1 + 67
  - LOADI R1, 67 // R1 ← 67
  - Access the data of **value** 67 immediately.

Input / Output

- Many types of input and output devices:
  - Keyboard, mouse, camera, sensors, printers, screen, sound…
- The CPU needs some agreed-upon protocol to talk to each of them
  - Software **drivers** realize these protocols.
- One general method of interaction uses ***memory mapping***:
  - Memory location A_1 holds the direction of the last movement of the mouse.
  - Memory location A_2 tells the printer to print single-side or double side.

Flow control



Which instruction to process next?

Flow control

- Usually CPU executes instructions **in sequence**.
- Sometimes "**jump**" **unconditionally** to another location, e.g. implement a loop.

Example:

```
101:  load R1,0
102:  add 1, R1
103:  ...
...   //  do something with R1 value
...   ...
156:  jmp 102   // goto 102
```

Symbolic version:

```
    load R1,0
LOOP:
    add 1, R1
    ...
    //  do something with R1 value
    ...
    jmp LOOP   // goto loop
```

Flow control

- Usually CPU executes instructions **in sequence**.
- Sometimes "**jump**" **unconditionally** to another location, e.g. implement a loop
- Sometimes **jump** only if some **condition** is **met**:

Example:

```
    jgt R1, 0, CONT   // if R1>0 jump to CONT
    sub R1, 0, R1     // R1 ← (0 - R1)
CONT:
     ...
    // Do something with positive R1
```

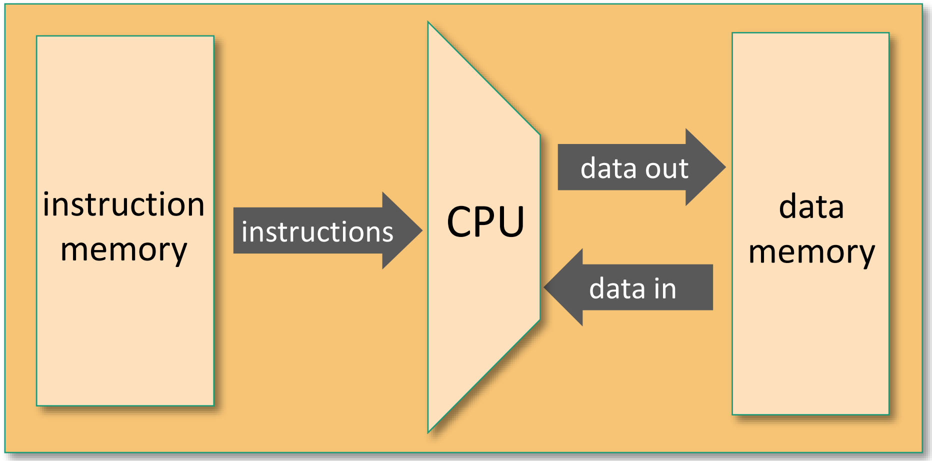
Recap

- Arithmetic and logic operations
  - Addition/substraction,
  - Bitwise operations.
- Memory access
  - Memory hierachy,
  - Data register/address register,
  - Four addressing modes,
  - Input/output memory mapping.
- Flow control
  - Run in sequence,
  - Jump conditionally/unconditionally.

Outlines

- Machine language
- Some basic operations
- Hack basics
  - ➢ Hack computer
  - ➢ Hack machine language
  - ➢ Hack input / output
- Hack assembly programming

Hack computer: hardware



A **16-bit** machine consisting of:

- Data memory (RAM): a sequence of **16-bit** registers:

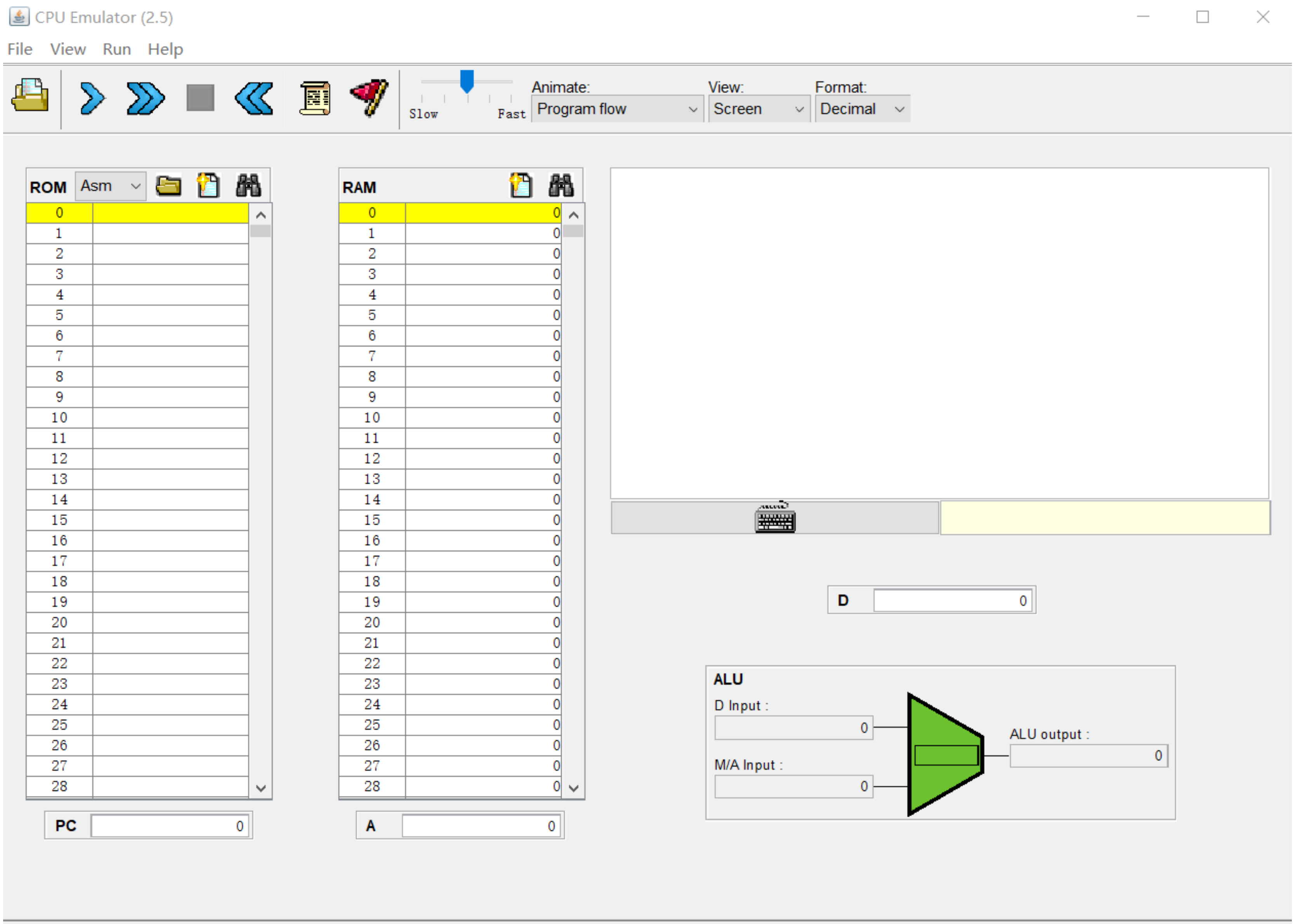
  RAM[0], RAM[1], RAM[2],...

- Instruction memory (ROM): a sequence of **16-bit** registers:

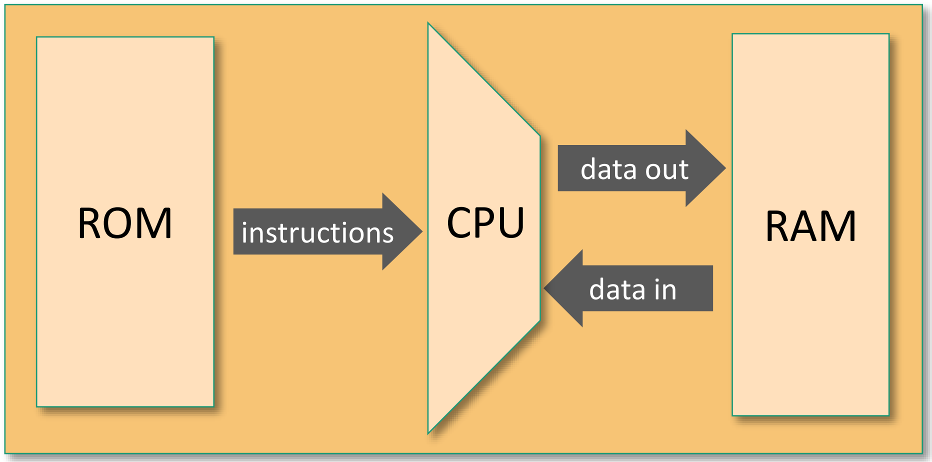
  ROM[0], ROM[1], ROM[2],...

- Central Processing Unit (CPU): performs **16-bit** instructions
- Instruction bus / data bus / address buses.

CPU Emulator



Hack computer: software

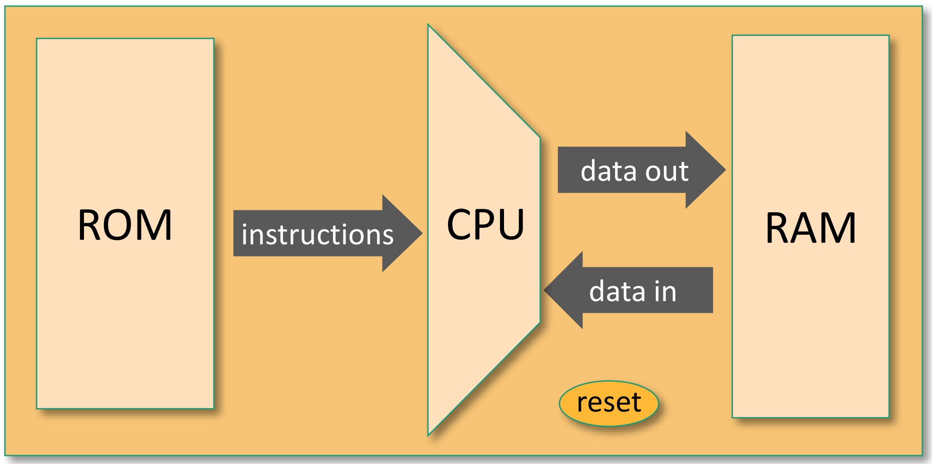


- Hack machine language:
  - 16-bit A-instructions
  - 16-bit C-instructions

Hack program = sequence of instructions written in the

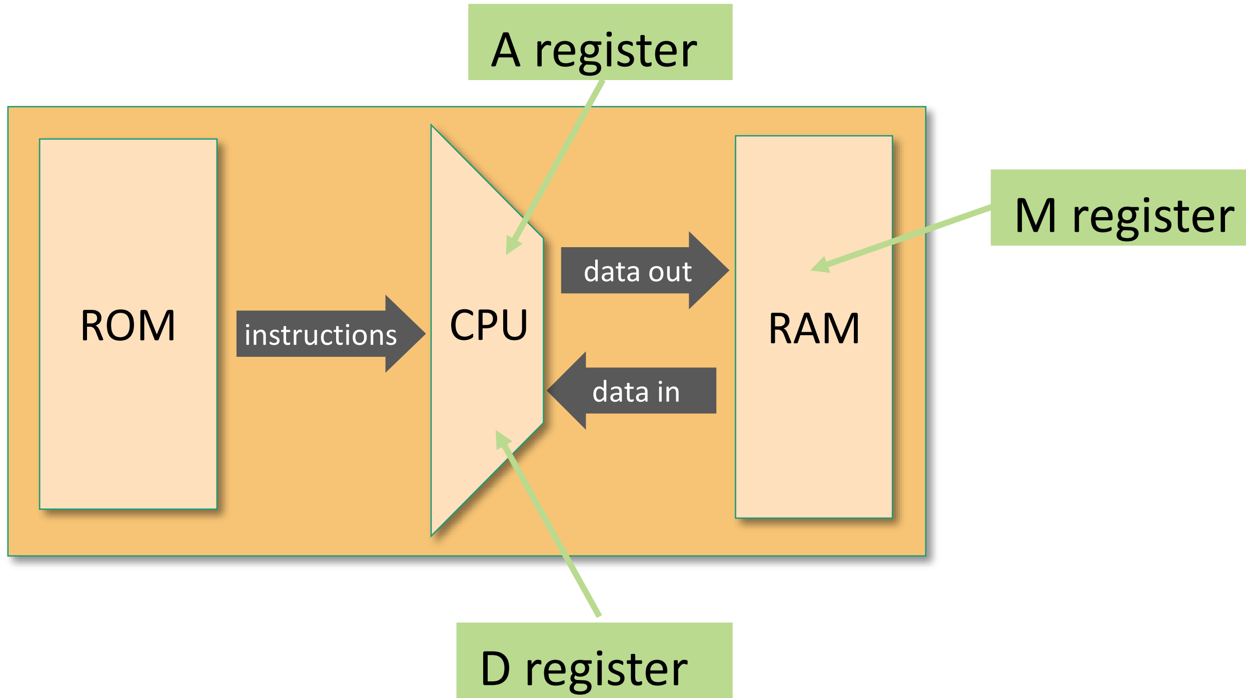
Hack machine language.

Hack computer: start



- The ROM is loaded with a Hack program.
- When the ***reset*** button is pushed, the program starts running.

Hack computer: registers



- Three 16-bit registers:
  - D: Store data
  - A: Store data / address the memory
  - M: Represent currently addressed memory register: **M = RAM[A]**

Instructions

- Every operation involving a ***memory*** location requires **two** Hack commands:
  - *A-instruction: address instruction*
    - Set the address to operate on.

      E.g., @17 // *A* $\leftarrow$ 17.
  - *C-instruction: command instruction*
    - Specify desired operation.

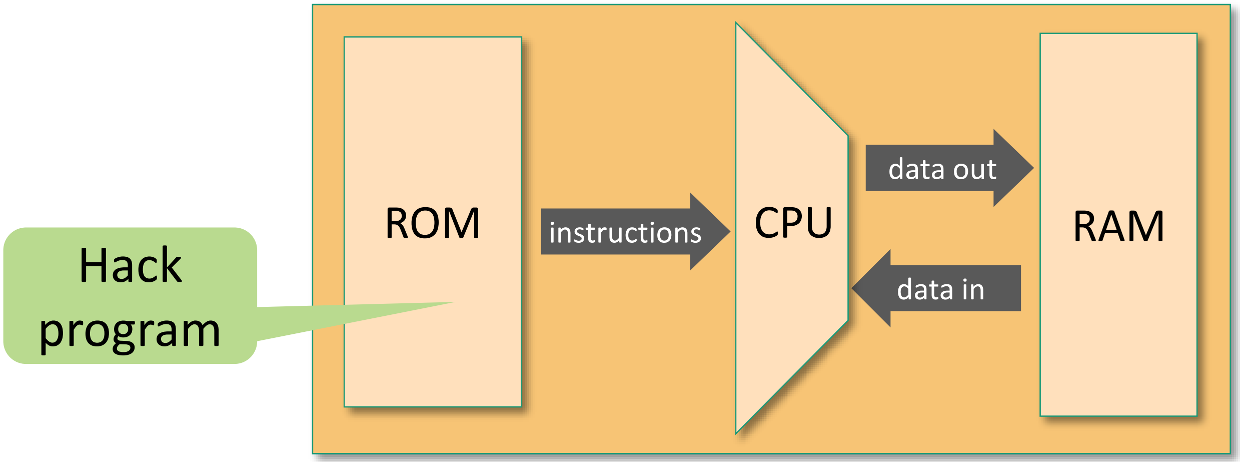
      E.g., @17 // 17 refers to memory location 17, *A* $\leftarrow$ 17.

      M=1 // RAM[17] = 1. *C-instruction*.

Outlines

- Introduction to machine language
- Some basic operations
- Hack basics
  - Hack computer
  - Hack machine language
  - Hack input / output
- Hack assembly programming

Hack machine language



Two ways to express the same semantics:

Symbolic:

```
@17
D+1;JLE
```

translate

assembler

Binary:

```
0000000000010001
1110011111000110
```

load & execute

Assembly language

Machine language

A-instruction specification

Semantics: Set the A register to ***value*** (memory address)

Symbolic syntax:

@ *value*

Example:

@21

set A to 21

Where *value* is either:

- a non-negative decimal constant ≤ 65535 (=2^{15}-1) or
- a symbol referring to a constant (*come back to this later*)

Binary syntax:

0 *value*

Where *value* is a 15-bit binary constant

Example:

0000000000010101

set A to 21

opcode signifying an A-instruction

C-instruction specification

<u>Syntax:</u> *dest = comp ; jump* (both *dest* and *jump* are optional)

where:

comp = **0**, **1**, **-1**, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A
M, !M, -M, M+1, M-1, D+M, D-M, M-D, D&M, D|M

dest = null, M, D, MD, A, AM, AD, AMD (M refer to **RAM[A]**)

jump = null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (***comp jump* 0**) is true, jumps to execute the instruction at **ROM[A].**

C-instruction specification

Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

opcode

not used

comp bits

dest bits

jump bits

comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a==0	a==1						

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

C-instruction: symbolic examples

```
// Set the D register to -1
D = -1   // only constants like 0 ,1, -1 can be directly assigned to D.
```

```
// Set the D register to 300
@300  // A = 300
D = A   // D = 300
```

```
// Sets RAM[300] to the value of the D register plus 1
@300   // A = 300, M refer to RAM[300]
M=D+1 // RAM[300] = D + 1
```

C-instruction: symbolic examples

```
// Set RAM[5] = 7;
@7      // A = 7;
D = A   // D = 7; we cannot directly assign 7 to D,
        // e.g. statement "D = 7" is not correct.
@5      // A = 5, M refer to RAM[5];
M = D   // RAM[5] = 7
```

```
// If (D-1 == 0), jump to execute the instruction stored in ROM[56]
@56        // A = 56
D-1;JEQ    // if (D-1 == 0) goto to instruction ROM[A]
```

Exercise: C-instruction

```
// Set RAM[0] = 16.
```

Exercise: C-instruction - answer

```
// Set RAM[0] = 16.
@16     // Set A = 16.
D=A     // Set D = 16.
@0      // Set A = 0.
M=D     // Set RAM[0] = 16.
```

Quiz: C-instruction

```
// Set RAM[0] = 16, RAM[1] = 32, then swap RAM[0] and RAM[1],
// using RAM[2] as temporary variable.
```

Quiz: C-instruction - answer

```
// Set RAM[0] = 16, RAM[1] = 32, then swap RAM[0] and RAM[1],
// using RAM[2] as temporary variable.
```

```
//RAM[0] = 16;
@16
D=A
@0
M=D

//RAM[1] = 32;
@32
D=A
@1
M=D
```

```
//swap, RAM[2]=RAM[0]
@0
D=M
@2
M=D

//RAM[0] = RAM[1]
@1
D=M
@0
M=D
```

```
//RAM[1]=RAM[2]
@2
D=M
@1
M=D
```

C-instruction: symbolic to binary

Symbolic:	Binary:
MD=D+1	1110011111011000
M=1	1110111111001000
D+1;JLE	1110011111000110

Hack program at a glance

Symbolic code

```
// Computes RAM[1] = 1+...+RAM[0]
// Usage: put a number in RAM[0]
@16   // RAM[16] represents i
M=1   // i = 1
@17   // RAM[17] represents sum
M=0   // sum = 0

@16
D=M
@0
D=D-M
@18    // if i>RAM[0] goto 18
D;JGT

@16
D=M
@17
M=D+M  // sum += i
@16
M=M+1  // i++
@4     // goto 4 (loop)
0;JMP

@17
D=M
@1
M=D    // RAM[1] = sum
@22    // program's end
0;JMP  // infinite loop
```

Observations:

- Hack program:
  a sequence of Hack instructions
- White space is permitted
- Comments are welcome
- There are better ways to write symbolic Hack programs.

No need to understand for now ...
We will come back to this shortly.

Hack programs: symbolic and binary

Symbolic code

```
    // Computes RAM[1] = 1+...+RAM[0]
    // Usage: put a number in RAM[0]
0   @16   // RAM[16] represents i
1   M=1   // i = 1
2   @17   // RAM[17] represents sum
3   M=0   // sum = 0

4   @16
5   D=M
6   @0
7   D=D-M
8   @18    // if i>RAM[0] goto 18
9   D;JGT

10  @16
11  D=M
12  @17
13  M=D+M  // sum += i
14  @16
15  M=M+1  // i++
16  @4     // goto 4 (loop)
17  0;JMP

18  @17
19  D=M
20  @1
21  M=D    // RAM[1] = sum
22  @22    // program's end
23  0;JMP  // infinite loop
```

translate

assembler

Binary code

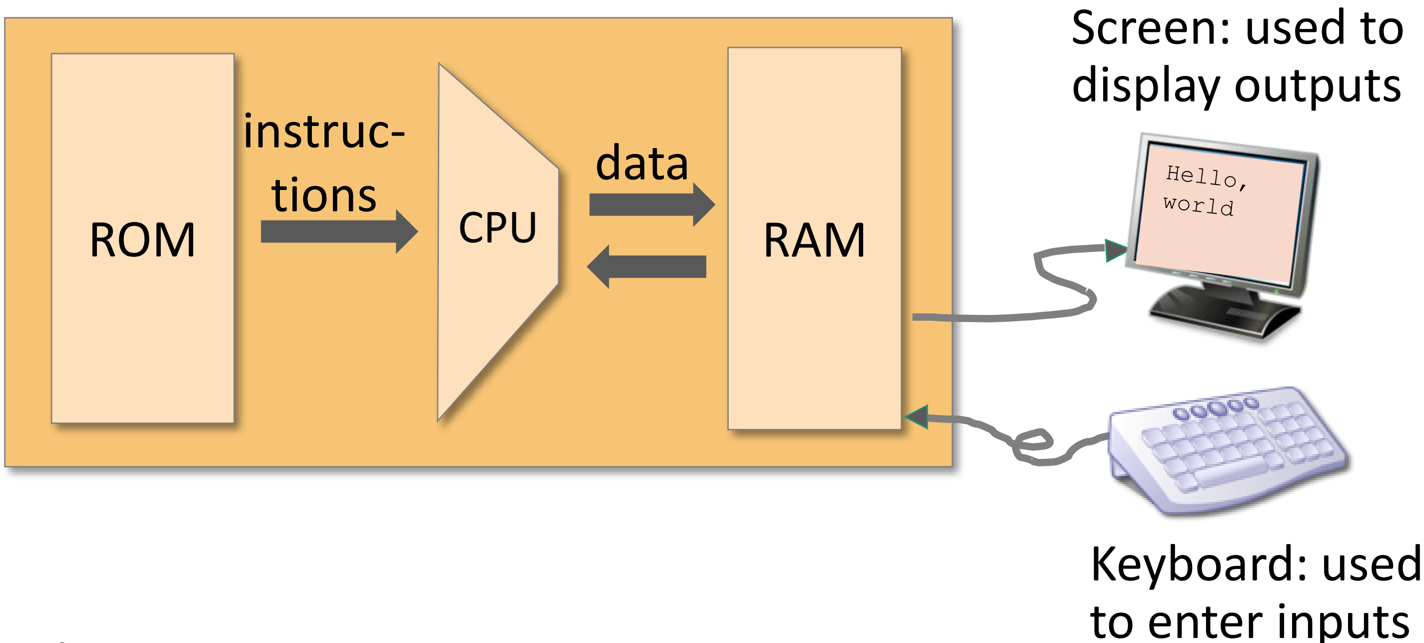
```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010001
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010101
1110101010000111
```

execute

Outlines

- Introduction to machine language
- Some basic operations
- Hack basics
  - Hack computer
  - Hack machine language
  - Hack input / output
- Hack assembly programming

Input / output



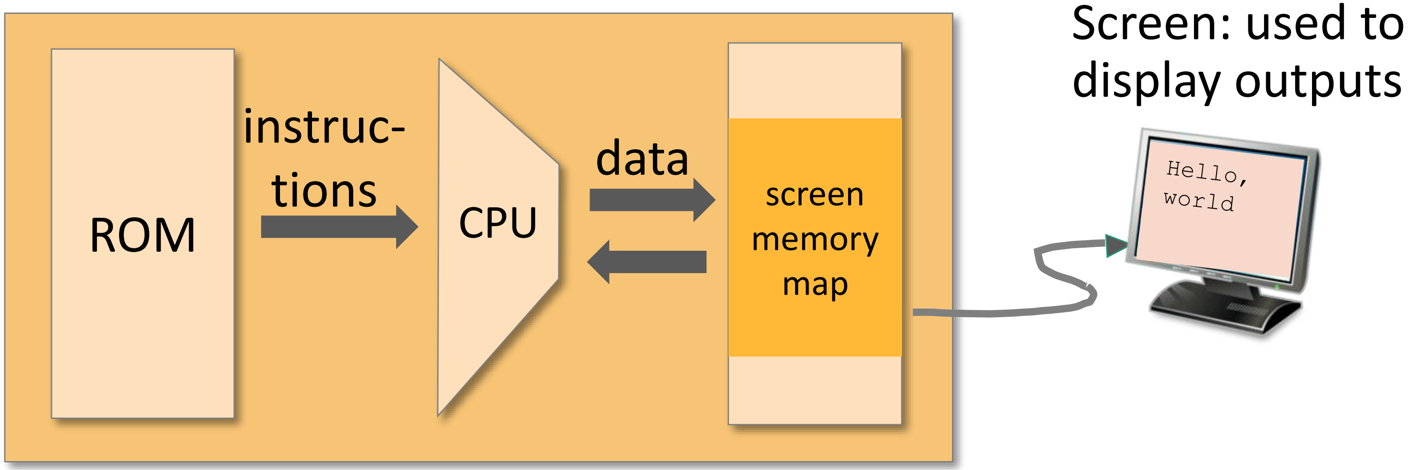
I/O handling (high-level):

Software libraries enabling text, graphics, audio, video, etc.

I/O handling (low-level):

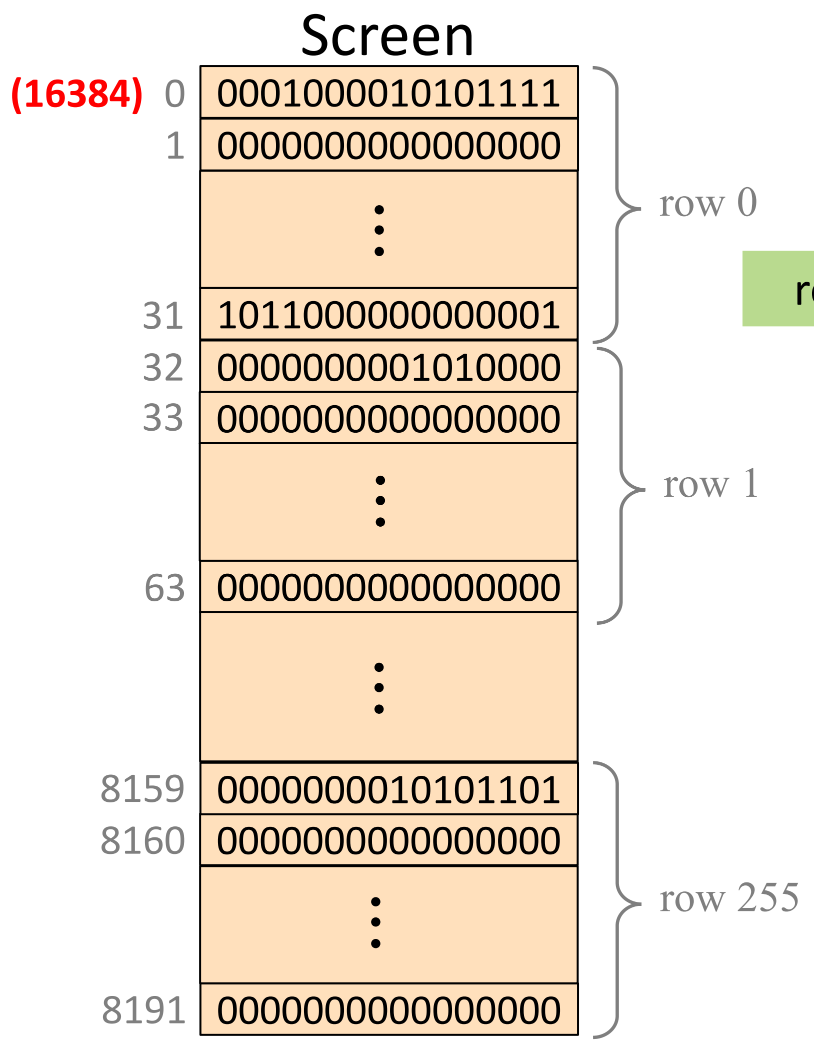
Bits manipulation.

Memory mapped output

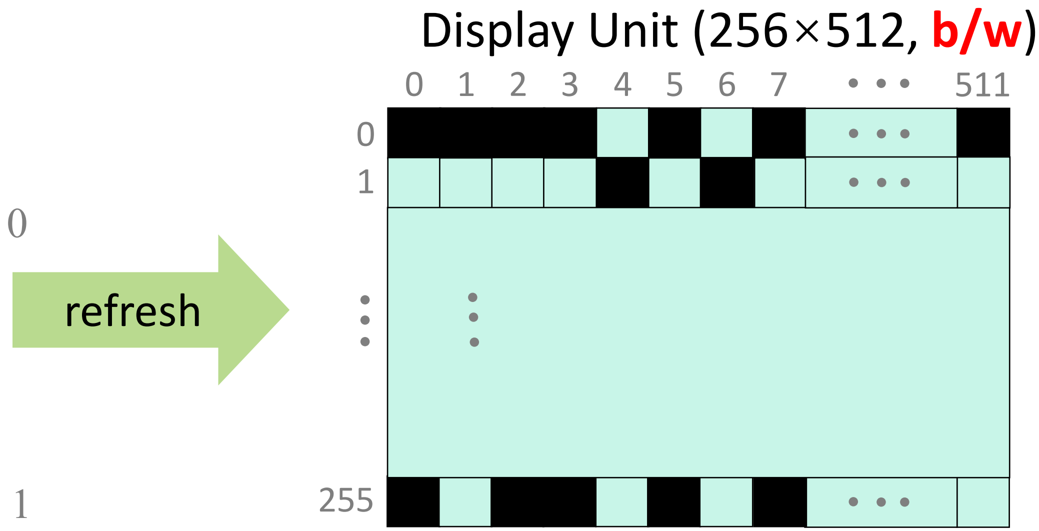


- Memory mapped output
  - A **designated memory** area to manage a display unit.
  - The physical display is continuously *refreshed* from the memory map, many times per second. (It is slow in Hack computer.)
  - Output is effected by writing code that manipulates the screen memory map.

Memory mapped output



8192 = 256×512/16



Integer division, round-down.

<u>To set pixel (*row*,*col*) on/off:</u>

(1) *word* = RAM[16384 + 32**row* + <u>*col*/16</u>]

(2) Set the (*col* % 16)*th* bit of *word* to 0 (white) or 1 (black).

(3) RAM[16384 + 32**row* + *col*/16] = *word*

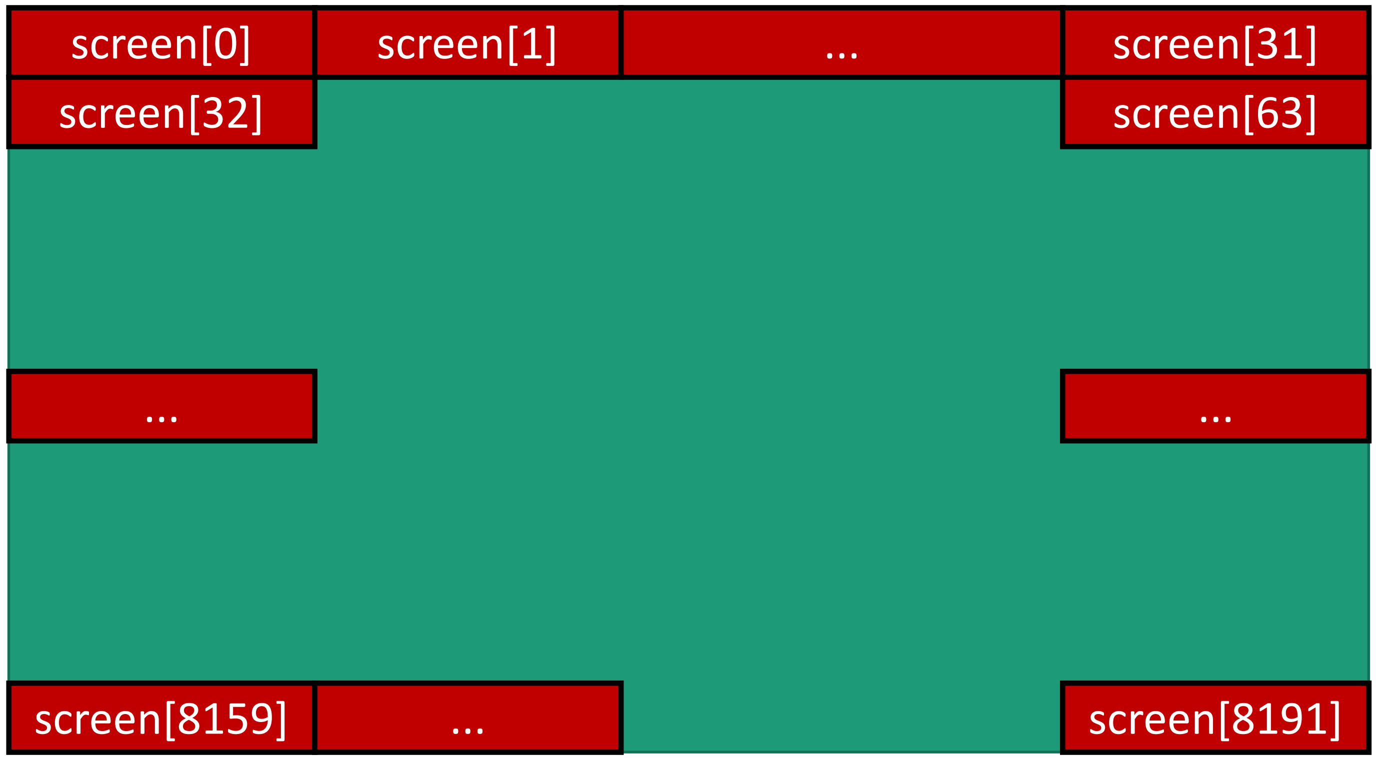
Hack Screen

512-bit wide

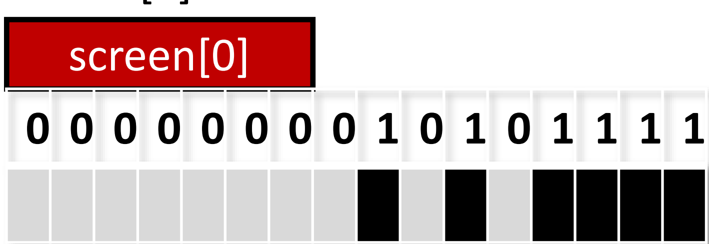
col⟶

256-bit high

row⟶



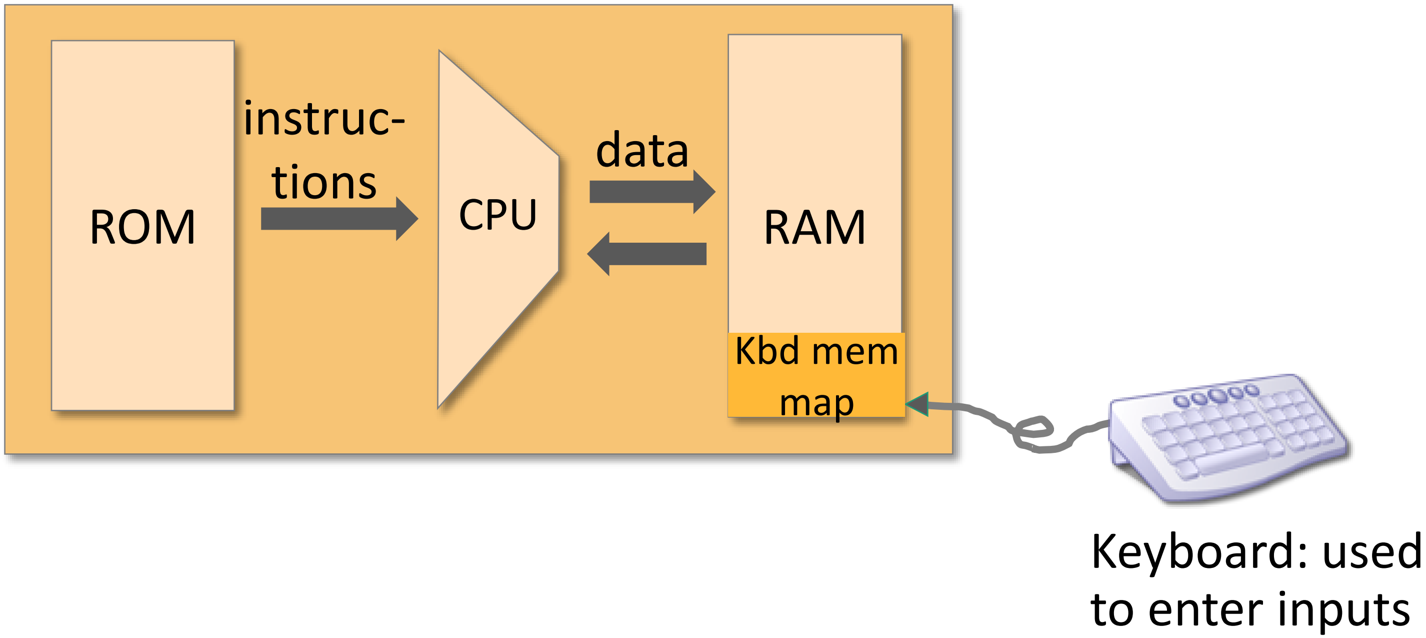
screen[0] = 1111010100000000



1 for black
0 for white

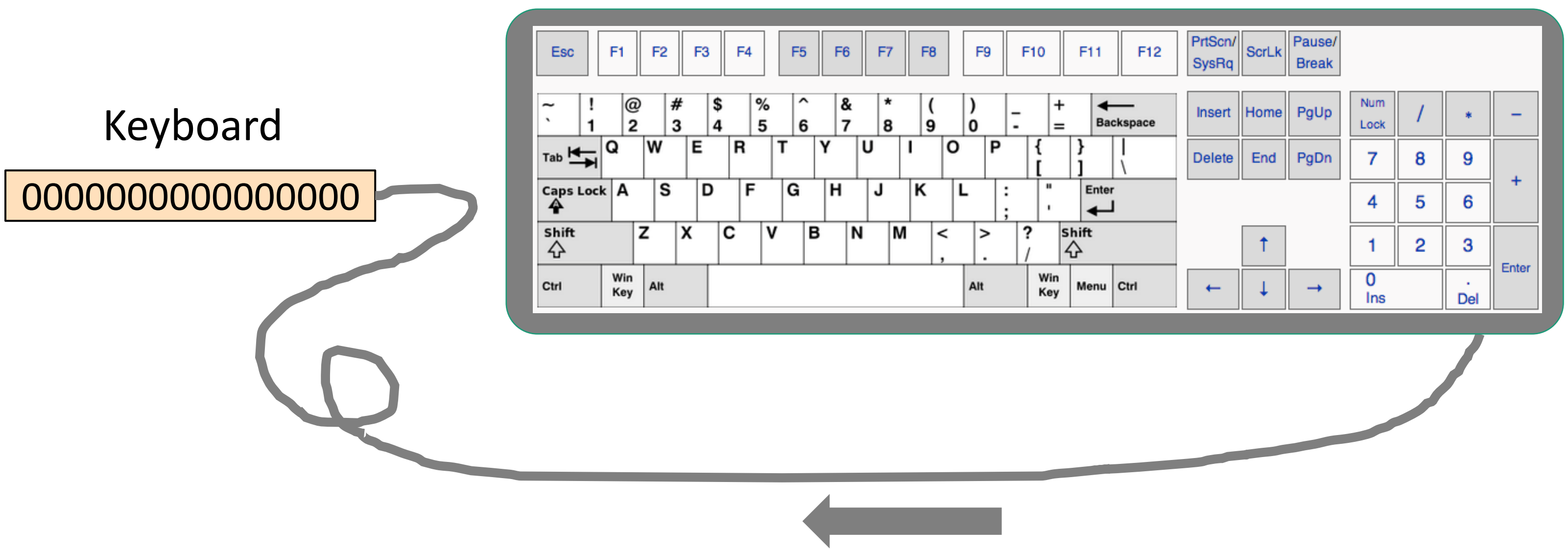
Bit[0], Bit[1], ..., Bit[15]

Input



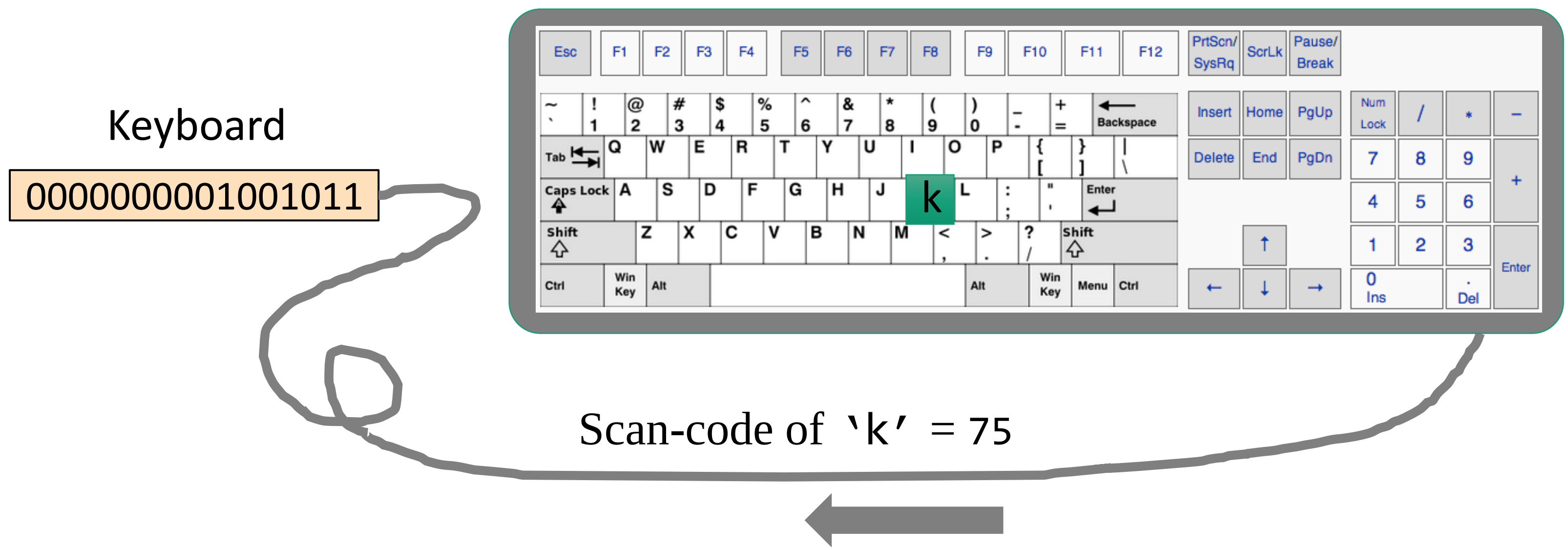
The physical keyboard is associated with a *keyboard memory map*.

Memory mapped input



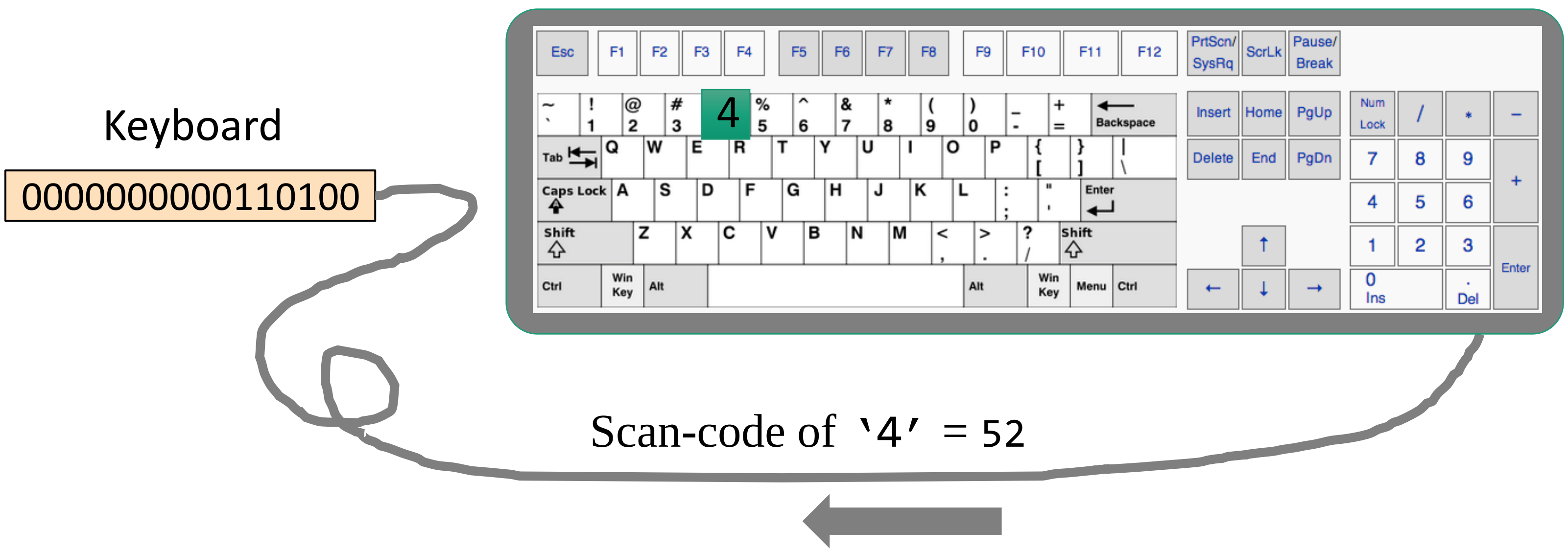
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



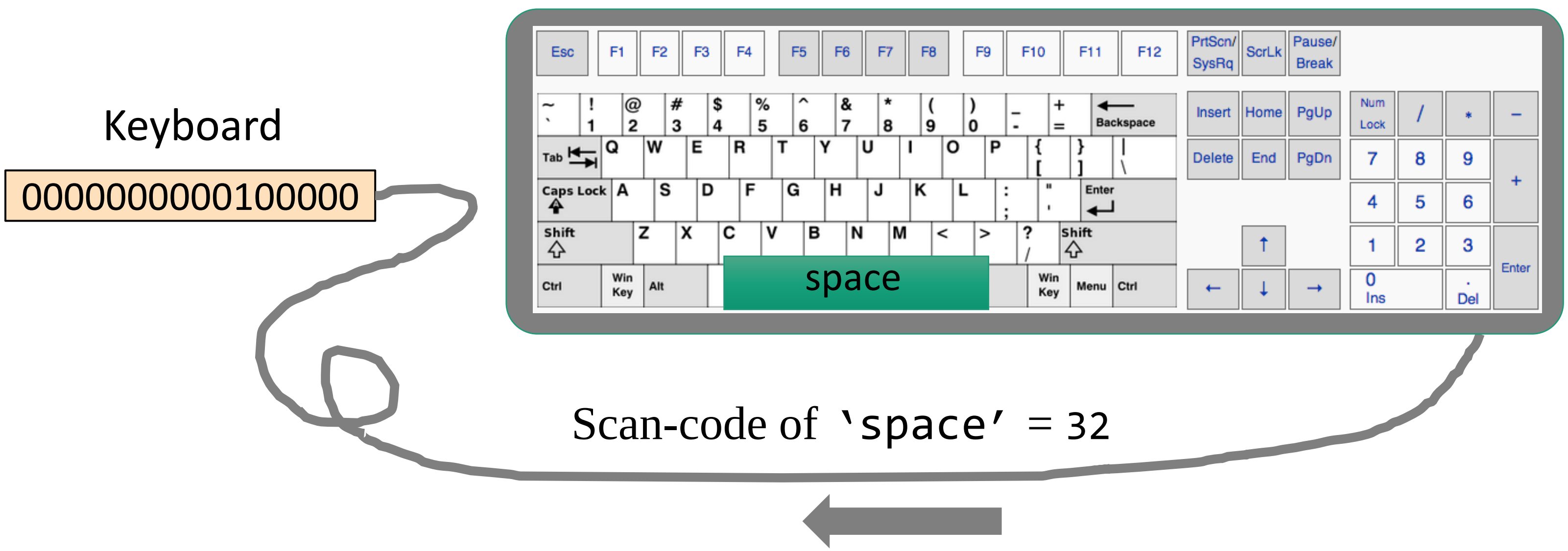
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



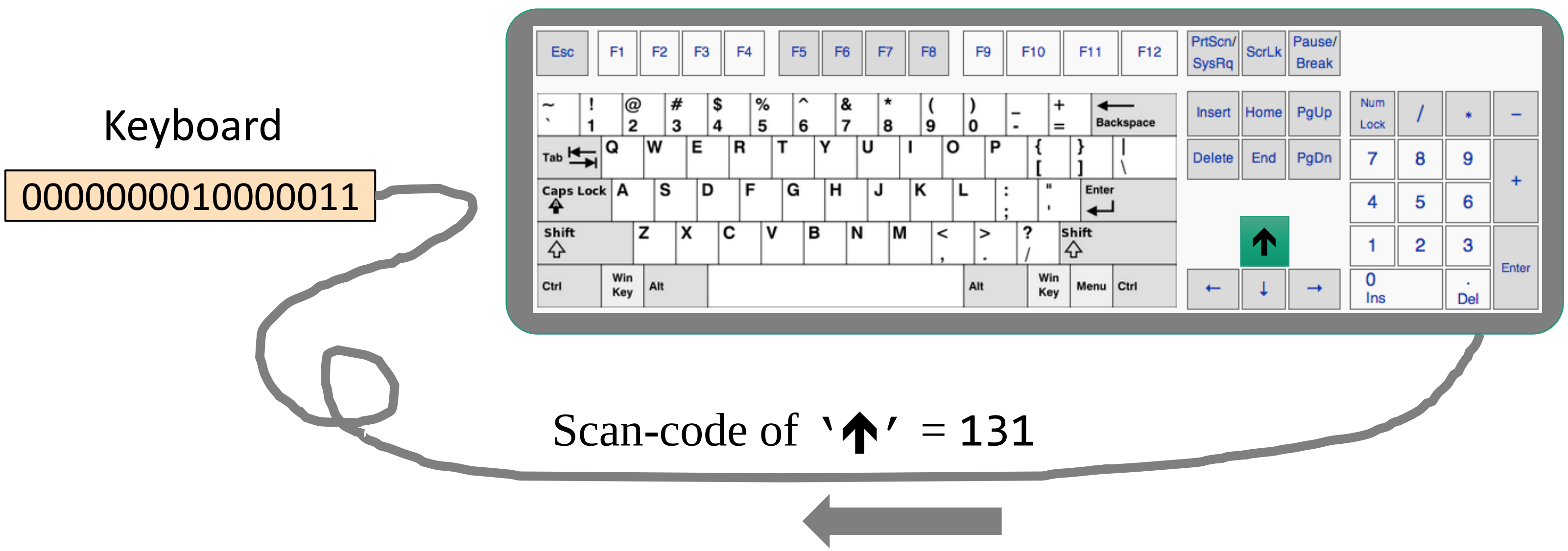
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*
- When no key is pressed, the resulting code is 0.

The Hack character set

key	code
(space)	32
!	33
“	34
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

key	code
0	48
1	49
…	…
9	57

:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	…
…	…
Z	90

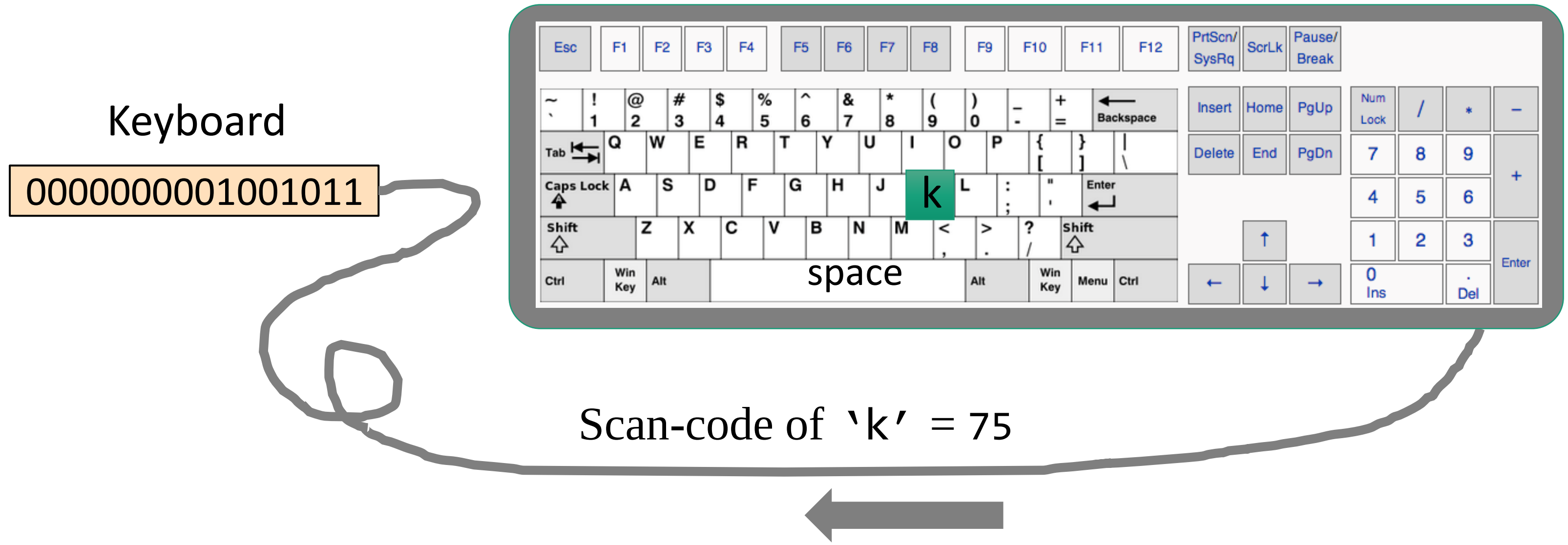
[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
…	…
f12	152

Handle the keyboard



- To check which key is currently pressed:
  - Probe the contents of the Keyboard chip
  - In the Hack computer: probe the contents of **RAM[24576]**.

Recap

- Hack computer
  - ➢16-bit machine,
  - ➢D/A/M registers.
- Hack machine language
  - ➢**A-instruction**,
  - ➢**C-instruction**.
- Hack input / output
  - ➢Screen: set screen memory for display,
  - ➢keyboard: probe memory for keyboard input.

Outlines

- Introduction to machine language
- Some basic operations
- Hack basics
- Hack assembly programming
  - Registers and memory
  - Branching, variables, iteration
  - Pointers, input/output

Hack assembly language (overview)

A-instruction:

```
@value    // A  = value
```

where *value* is either a constant or a symbol referring to such a constant

C-instruction:

dest = comp ; jump

(both *dest* and *jump* are optional)

where:

comp = 0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A
M, !M, -M, M+1, M-1, D+M, D-M, M-D, D&M, D|M

dest = null, M, D, MD, A, AM, AD, AMD (M refers to RAM[A])

jump = null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP

Semantics:

- Compute the value of *comp*
- Store the result in *dest*
- If the Boolean expression (*comp jump 0*) is true, jump to execute the instruction at ROM[A]

Hack assembler

Assembly program

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]
   @R1
   D=M
   @temp
   M=D    // temp = R1

   @R0
   D=M
   @R1
   M=D    // R1 = R0

   @temp
   D=M
   @R0
   M=D    // R0 = temp

(END)
   @END
   0;JMP
```

Hack assembler

Binary code

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
0000000000000000
1111110000010000
0000000000000001
1110001100001000
0000000000010000
1111110000010000
0000000000000000
1110001100001000
0000000000001100
1110101010000111
```

load & execute

We'll develop a Hack assembler later in this module.

CPU emulator

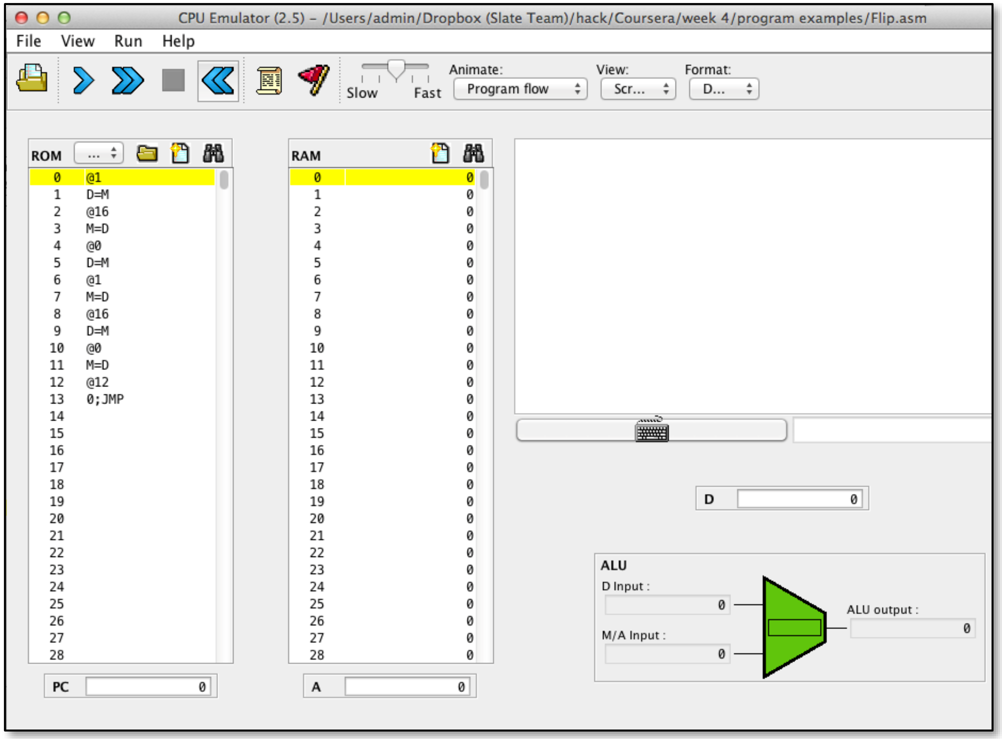
Assembly program

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]
   @R1
   D=M
   @temp
   M=D    // temp = R1

   @R0
   D=M
   @R1
   M=D    // R1 = R0

   @temp
   D=M
   @R0
   M=D    // R0 = temp

(END)
   @END
   0;JMP
```

load

(The simulator translates from symbolic to binary as it loads)

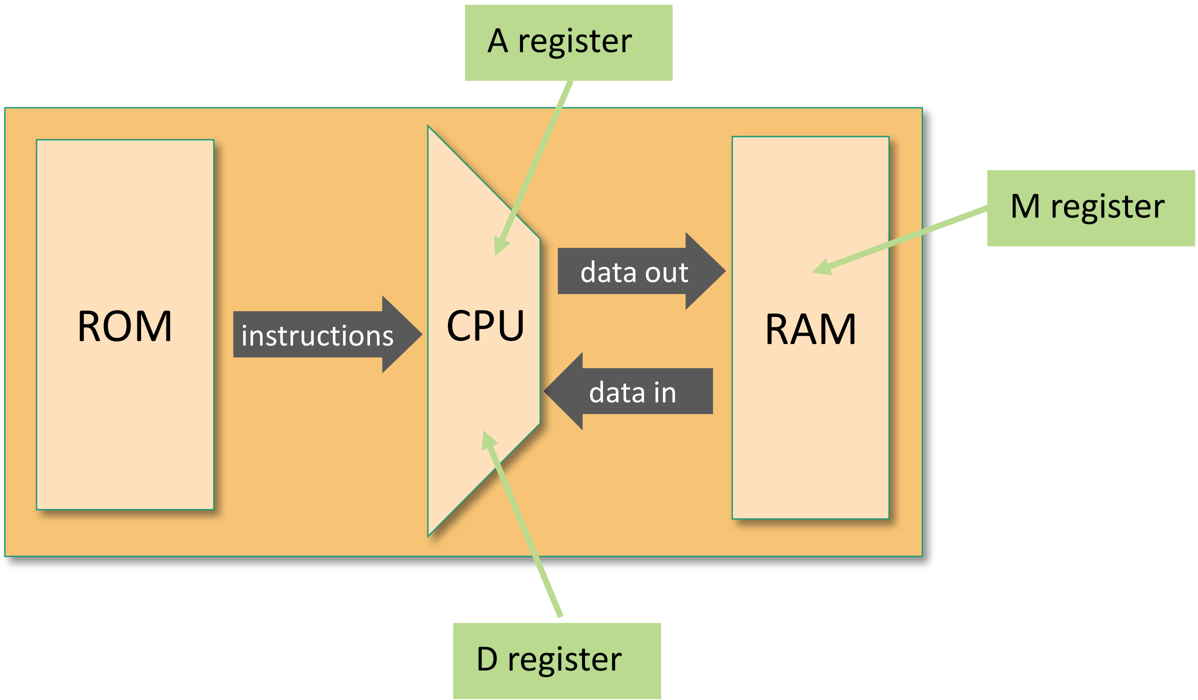
CPU Emulator



- A software tool
- Convenient for debugging and executing symbolic Hack programs.

Registers and memory

- D: Store data.
- A: Store data / address the memory.
- M: Currently addressed memory register: M = RAM[A].



Registers and memory

- D: Store data.
- A: Store data / address the memory.
- M: Currently addressed memory register: M = RAM[A].

Typical operations:

```
// D++
D=D+1

// D=10
@10
D=A
```

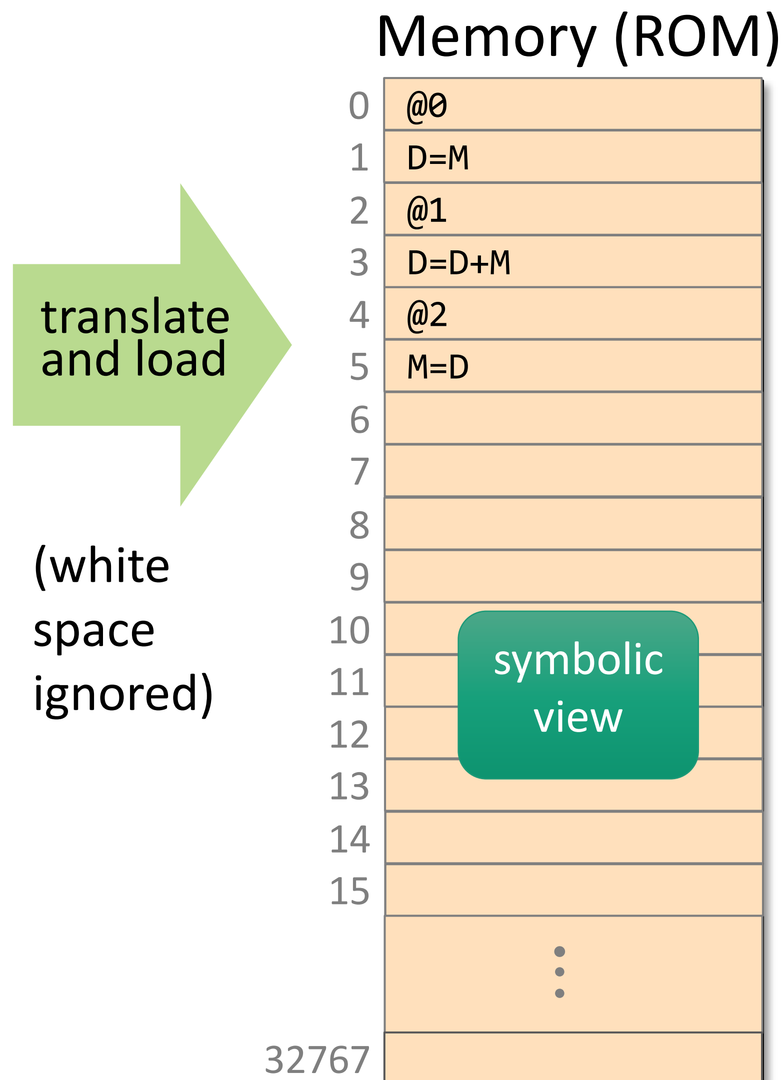
```
// D=RAM[17]
@17
D=M

// RAM[17]=D
@17
M=D
```

```
// RAM[17]=10
@10
D=A
@17
M=D

// RAM[5] = RAM[3]
@3
D=M
@5
M=D
```

Program example: add two numbers

Hack assembly code

```
   // Program: Add2.asm
   // Computes: RAM[2] = RAM[0] +
   // RAM[1]
   // Usage: put values in RAM[0],
   // RAM[1]
0  @0
1  D=M   // D = RAM[0]

2  @1
3  D=D+M // D = D + RAM[1]

4  @2
5  M=D   // RAM[2] = D
```



Program example: add two numbers

Hack assembly code

```
  // Program: Add2.asm
  // Computes: RAM[2] = RAM[0] +
  // RAM[1]
  // Usage: put values in RAM[0],
  // RAM[1]
0 @0
1 D=M   // D = RAM[0]

2 @1
3 D=D+M // D = D + RAM[1]

4 @2
5 M=D   // RAM[2] = D
```

translate and load

(white space ignored)

Memory (ROM)

0	0000000000000000
1	1111110000010000
2	0000000000000001
3	1111000010010000
4	0000000000000010
5	1110001100001000
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
⋮	
32767	

binary view

Terminate a program

Hack assembly code

```
  // Program: Add2.asm
  // Computes: RAM[2] = RAM[0] +
  // RAM[1]
  // Usage: put values in RAM[0],
  // RAM[1]
0 @0
1 D=M   // D = RAM[0]

2 @1
3 D=D+M // D = D + RAM[1]

4 @2
5 M=D   // RAM[2] = D
```

translate and load

Memory (ROM)

0	@0
1	D=M
2	@1
3	D=D+M
4	@2
5	M=D
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
⋮	
32767	

malicious code starts here ...

Attack on the computer

Terminate a program

Hack assembly code

```
    // Program: Add2.asm
    // Computes: RAM[2] = RAM[0] + RAM[1]
    // Usage: put values in RAM[0], RAM[1]
0   @0
1   D=M   // D = RAM[0]

2   @1
3   D=D+M // D = D + RAM[1]

4   @2
5   M=D   // RAM[2] = D

6   @6
7   0;JMP
```

- Jump to instruction number A (which happens to be 6),
- 0: syntax convention for JMP instruction.

translate and load

Memory (ROM)

0	@0
1	D=M
2	@1
3	D=D+M
4	@2
5	M=D
6	@6
7	0;JMP
8	
9	
10	
11	
12	
13	
14	
15	
⋮	
32767	

<u>Best practice:</u>

To terminate a program safely, end it with an infinite loop.

Built-in symbols

The Hack assembly language features *built-in symbols*:

symbol	value
R0	0
R1	1
...	...
R15	15

Attention: Hack is case-sensitive!
R5 and r5 are different symbols.

These symbols can be used to denote “virtual registers”

Example: suppose we use RAM[5] to represent some variable, and we wish to let RAM[5]=7

implementation:

```
// let RAM[5] = 7
@7
D=A

@5
M=D
```

better style:

```
// let RAM[5] = 7
@7
D=A

@R5
M=D
```

Built-in symbols

The Hack assembly language features *built-in symbols*:

symbol	value
R0	0
R1	1
...	...
R15	15
SCREEN	16384
KBD	24576

symbol	value
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

- R0, R1 ,…, R15 : “virtual registers”, can be used as variables.
- SCREEN and KBD : base addresses of I/O memory maps
- Remaining symbols: used in the implementation of the Hack virtual machine, discussed in Chapter 7-8.

Outlines

- Introduction to machine language
- Some basic operations
- Hack basics
- Hack assembly programming
  - Registers and memory
  - Branching, variables, iteration
  - Pointers, input/output

Branching

```
    // Program: Signum.asm
    // Computes: if R0>0
    //              R1=1
    //           else
    //              R1=0
    // Usage: put a value in RAM[0],
    //        run and inspect RAM[0].

0      @R0
1      D=M    // D = RAM[0]

2      @8
3      D;JGT  // If R0>0 goto 8

4      @R1
5      M=0    // RAM[1]=0
6      @10
7      0;JMP  // goto end

8      @R1
9      M=1    // R1=1

10     @10    // end
11     0;JMP
```

Labels

```
    // Program: Signum.asm
    // Computes: if R0>0
    //              R1=1
    //           else
    //              R1=0
    // Usage: put a value in RAM[0],
    //        run and inspect RAM[1].

0      @R0
1      D=M    // D = RAM[0]

2      @POSITIVE
3      D;JGT  // If R0>0 goto POSITIVE

4      @R1
5      M=0    // RAM[1]=0
6      @END
7      0;JMP  // goto end

    (POSITIVE)
8      @R1
9      M=1    // R1=1

    (END)
10     @END // end
11     0;JMP
```

referring to a label

declaring a label

resolving labels

Memory

0	@0
1	D=M
2	@8 // @POSITIVE
3	D;JGT
4	@1
5	M=0
6	@10 // @END
7	0;JMP
8	@1
9	M=1
10	@10 // @END
11	0;JMP
12	
13	
14	
15	
⋮	
32767	

Label resolution rules:

- **Label declarations generate no code!!!**
- Each reference to a label is replaced with a reference to the instruction number **following** that label's declaration.

Labels

```
// Program: Signum.asm
// Computes: if R0>0
//             R1=1
//           else
//             R1=0
// Usage: put a value in RAM[0],
//        run and inspect RAM[1].

0      @R0
1      D=M    // D = RAM[0]

2      @POSITIVE
3      D;JGT  // If R0>0 goto 8

4      @R1
5      M=0    // RAM[1]=0
6      @END
7      0;JMP  // goto end

     (POSITIVE)
8      @R1
9      M=1    // R1=1

     (END)
10     @END // end
11     0;JMP
```

referring to a label

declaring a label

resolving labels

Memory

0	`@0`
1	`D=M`
2	`@8 // @POSITIVE`
3	`D;JGT`
4	`@1`
5	`M=0`
6	`@10 // @END`
7	`0;JMP`
8	`@1`
9	`M=1`
10	`@10 // @END`
11	`0;JMP`
12	
13	
14	
15	
⋮	
32767	

<u>Implications:</u>

- Instruction numbers no longer needed in symbolic programming
- The symbolic code becomes ***relocatable***.

Variables

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

   @R1
   D=M
   @temp
   M=D      // temp = R1

   @R0
   D=M
   @R1
   M=D      // R1 = R0

   @temp
   D=M
   @R0
   M=D      // R0 = temp

(END)
   @END
   0;JMP
```

symbol used for the first time

symbol used again

resolving symbols

Symbol resolution rules:

- A reference to a symbol without label declaration is treated as a reference to a variable.
- If the reference @*symbol* occurs in the program for first time, *symbol* is allocated to address **16** onward (say *n*), and the generated code is @*n*.
- All subsequencet @*symbol* commands are translated into @*n*.

Note: variables are allocated to **RAM[16]** onward.

Memory

0	@1
1	D=M
2	@16 // @temp
3	M=D
4	@0
5	D=M
6	@1
7	M=D
8	@16 // @temp
9	D=M
10	@0
11	M=D
12	@12
13	0;JMP
14	
15	
	⋮
32767	

Variables

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

   @R1
   D=M
   @temp
   M=D     // temp = R1

   @R0
   D=M
   @R1
   M=D     // R1 = R0

   @temp
   D=M
   @R0
   M=D     // R0 = temp

(END)
   @END
   0;JMP
```

resolving symbols

Implications:

symbolic code is easy to read and debug

Memory

0	@1
1	D=M
2	@16 // @temp
3	M=D
4	@0
5	D=M
6	@1
7	M=D
8	@16 // @temp
9	D=M
10	@0
11	M=D
12	@12
13	0;JMP
14	
15	
	⋮
32767	

Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]
   n = R0
   i = 1
   sum = 0
LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

assembly code

```
// Program: Sum1toN.asm
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number n in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0

   @i
   M=1   // i = 1

   @sum
   M=0   // sum = 0
   ...
```

Memory

0	@0
1	D=M
2	@16 // @n
3	M=D
4	@17 // @i
5	M=1
6	@18 // @sum
7	M=0
8	...
9	
⋮	⋮
32767	

Variables are allocated to consecutive RAM locations from address 16 onward

Iterative processing

pseudo code

```
// Compute RAM[1] =
      1+2+ ... +RAM[0]
  n = R0
  i = 1
  sum = 0
LOOP:
  if i > n goto STOP
  sum = sum + i
  i = i + 1
  goto LOOP
STOP:
  R1 = sum
```

assembly code

```
// Compute RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in
      RAM[0]
  @R0
  D=M
  @n
  M=D   // n = R0
  @i
  M=1   // i = 1
  @sum
  M=0   // sum = 0
(LOOP)
  @i
  D=M // D = i
  @n
  D=D-M // D = i - n
  @STOP
  D;JGT  // if i > n goto STOP
```

```
  @i
  D=M // D = i
  @sum
  M=D+M // sum = sum + i
  @i
  M=M+1  // i = i + 1
  @LOOP
  0;JMP // goto LOOP
(STOP)
  @sum
  D=M // D = sum
  @R1
  M=D    // RAM[1] = sum
(END)
  @END
  0;JMP // end
```

Program execution

assembly program

```
// Compute RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M // D = i
   @n
   D=D-M // D = i - n
   @STOP
   D;JGT  // if i > n goto STOP
   @i
   D=M // D = i
   @sum
   M=D+M // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP // goto to LOOP
(STOP)
   @sum
   D=M // D = sum
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP // end
```

iterations

	0	1	2	3	...
RAM[0]:	3				
n:	3				
i:	1	2	3	4	...
sum:	0	1	3	6	...

Sample exam question

```
   @5
   D=A
   @R0
   M=D

   @R0
   D=M
   @n
   M=D
   @pre
   M=0
   @cur
   M=1

(LOOP)
   @cur
   D=M
   @n
   D=D-M
   @STOP
   D;JGT
```

```
   @pre
   D=M
   @cur
   D=D+M
   @nex
   M=D

   @cur
   D=M
   @pre
   M=D

   @nex
   D=M
   @cur
   M=D
   @LOOP
   0;JMP
```

```
(STOP)
   @nex
   D=M
   @R1
   M=D

(END)
   @END
   0;JMP
```

2.(a) I. Please derive the value of RAM[1] after the execution of this piece of code.
[4 marks]

From 2019-2020 CSF exam.

Hint: this piece of code implements Fibonacci number.

Writing assembly programs

assembly program

```
// Compute RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
  @R0
  D=M
  @n
  M=D   // n = R0
  @i
  M=1   // i = 1
  @sum
  M=0   // sum = 0
(LOOP)
  @i
  D=M // D = i
  @n
  D=D-M // D = i - n
  @STOP
  D;JGT  // if i > n goto STOP
  @i
  D=M // D = i
  @sum
  M=D+M // sum = sum + i
  @i
  M=M+1  // i = i + 1
  @LOOP
  0;JMP // goto to LOOP
(STOP)
  @sum
  D=M // D = sum
  @R1
  M=D    // RAM[1] = sum
(END)
  @END
  0;JMP // end
```

Best practice:

- **Design** the program using pseudo code,
- **Write** the program in assembly language,
- **Test** the program (on paper) using a variable-value trace table.

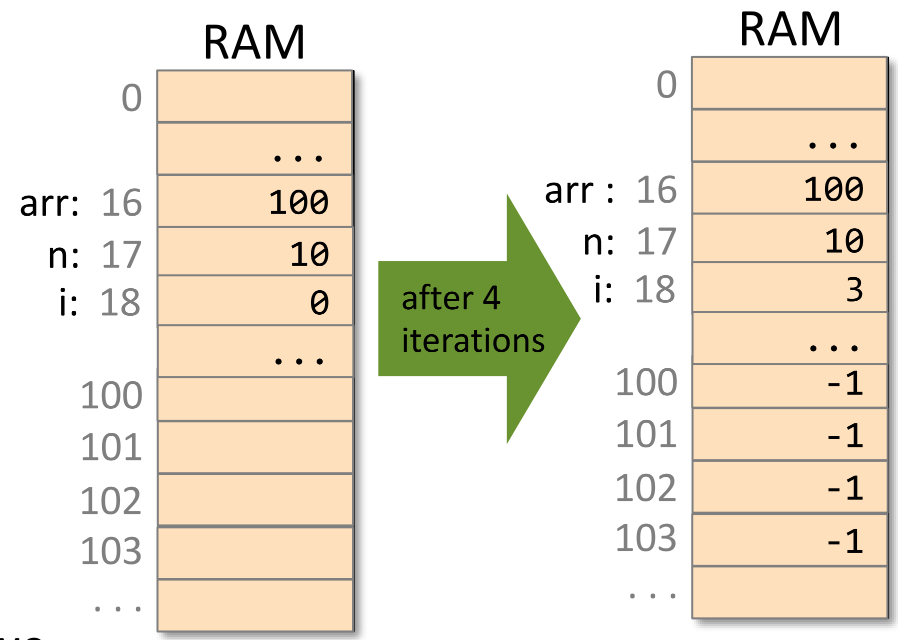
Outlines

- Introduction to machine language
- Some basic operations
- Hack basics
- Hack assembly programming
  - Registers and memory
  - Branching, variables, iteration
  - Pointers, input/output

Pointers

Example:

```
// for (i=0; i<n; i++) {
//     arr[i] = -1
// }
```



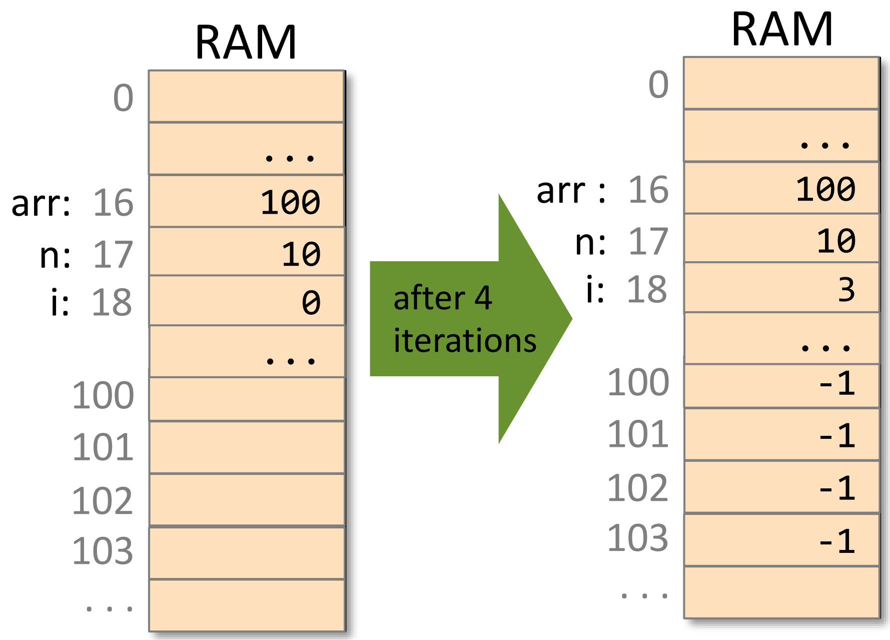
<u>Observations:</u>

- The array is implemented as a **block of memories**.
- To access these memories one by one, we need a variable to hold the current address.
- *Variables that represent addresses* are called <u>**pointers**</u>.
- There is nothing special about pointer variables, except that their values are interpreted as addresses.

Pointers

Example:

```
// for (i=0; i<n; i++) {
//     arr[i] = -1
// }
  // Suppose that arr=100 and n=10
  // Let arr = 100
  @100
  D=A //D = 100
  @arr
  M=D // arr = 100
  // Let n = 10
  @10
  D=A // D = 10
  @n
  M=D // n = 10
  // Let i = 0
  @i
  M=0 // i = 0
  // Loop code continues
  // in next slide...
```

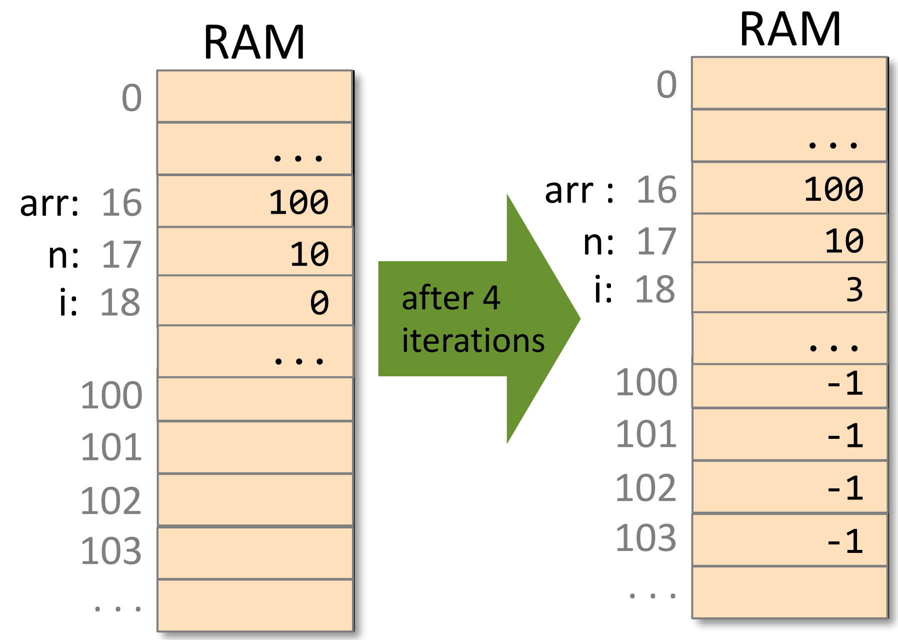


Pointers

Example:

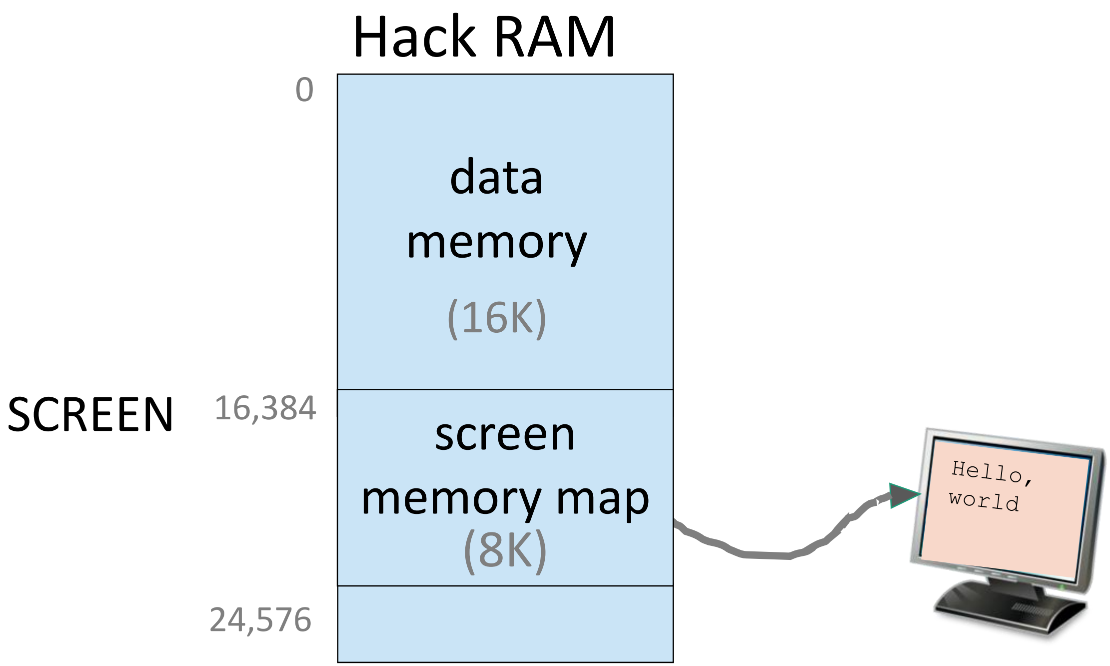
```
(LOOP)
  // if (i==n) goto END
  @i
  D=M // D = i
  @n
  D=D-M // D = i-n
  @END
  D;JEQ // if (i==n) goto END
  // RAM[arr+i] = -1
  @arr
  D=M // D = arr
  @i
  A=D+M // A = arr + i
  M=-1 // M[arr+i] = -1
   // i++
  @i
  M=M+1 // i = i + 1
  // goto LOOP
  @LOOP
  0;JMP
(END)
  @END
  0;JMP // END
```

typical pointer manipulation



- Pointers: Variables that store memory addresses (like `arr`).
- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like **A =** ***expression***.
- Semantics:
  "set the address register to some value".

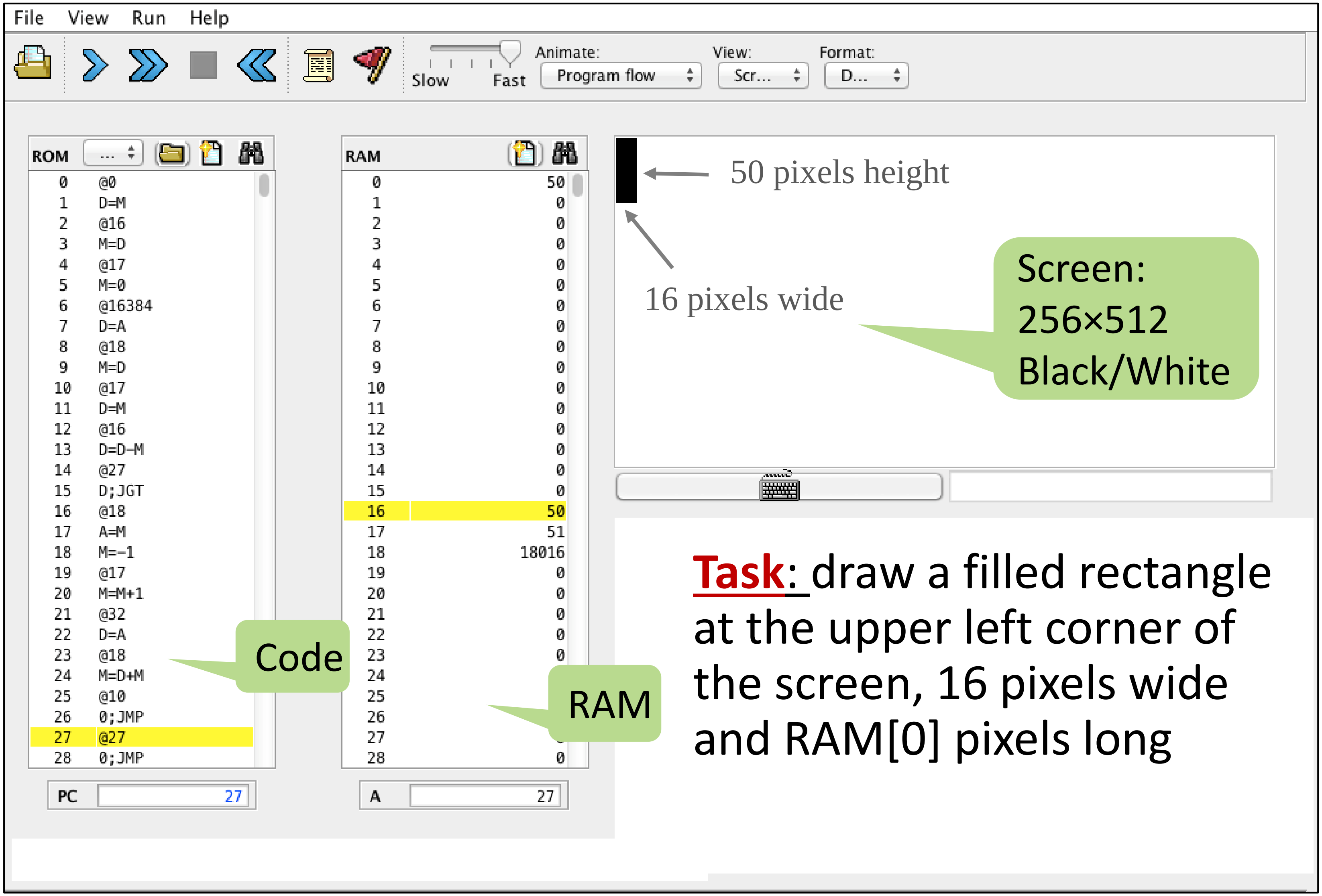
Output



Hack language convention:

- SCREEN: base address of the screen memory map

Handling the screen (example)

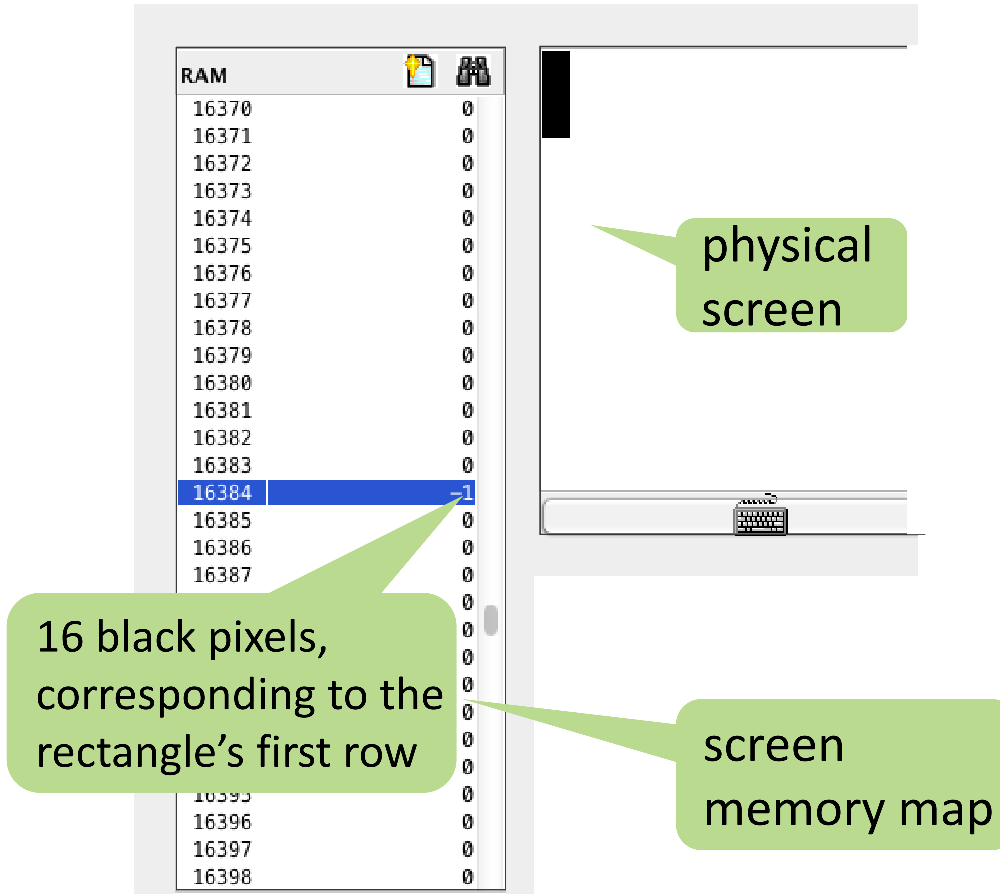


Task: draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and RAM[0] pixels long

Handling the screen (example)

Pseudo code

```
// for (i=0; i<n; i++) {
//     draw 16 black pixels at the
//     beginning of row i
// }

addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if i == n goto END
  RAM[addr] = -1  //
     1111111111111111
  // advances to the next row
  // 512 = 16 × 32
  addr = addr + 32
  i = i + 1
  goto LOOP

END:
  goto END
```



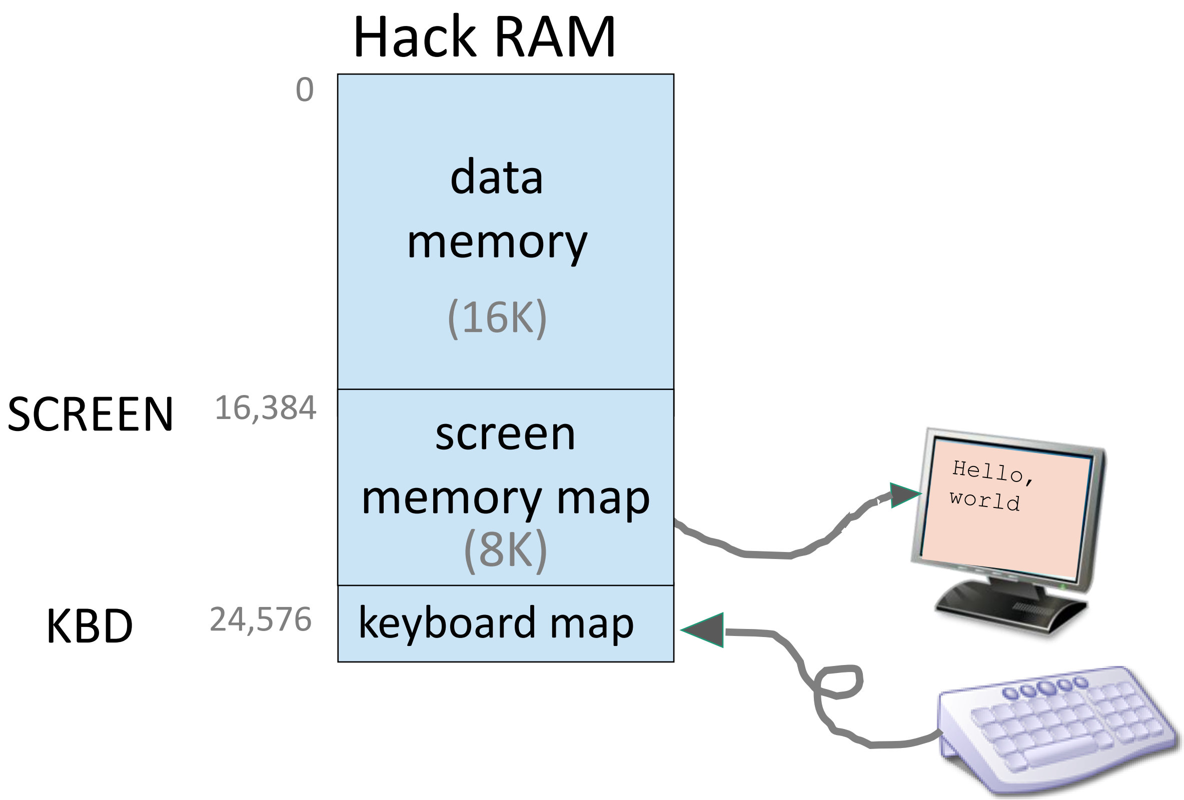
Handling the screen (example)

Assembly code

```
// Program: Rectangle.asm
// Draws a filled rectangle at the
// screen's top left corner, with
// width of 16 pixels and height of
// RAM[0] pixels.
// Usage: put a non-negative number
// (rectangle's height) in RAM[0].

   @SCREEN
   D=A
   @addr
   M=D  // addr = 16384
      // (screen's base address)
   @R0
   D=M
   @n
   M=D  // n = RAM[0]

   @i
   M=0  // i = 0
```

```
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ  // if i==n goto END

   @addr
   A=M
   M=-1   // RAM[addr]=1111111111111111

   @i
   M=M+1  // i = i + 1
   @32
   D=A   // D = 32
   @addr
   M=D+M  // addr = addr + 32
   @LOOP
   0;JMP  // goto LOOP

(END)
   @END   // program's end
   0;JMP  // infinite loop
```

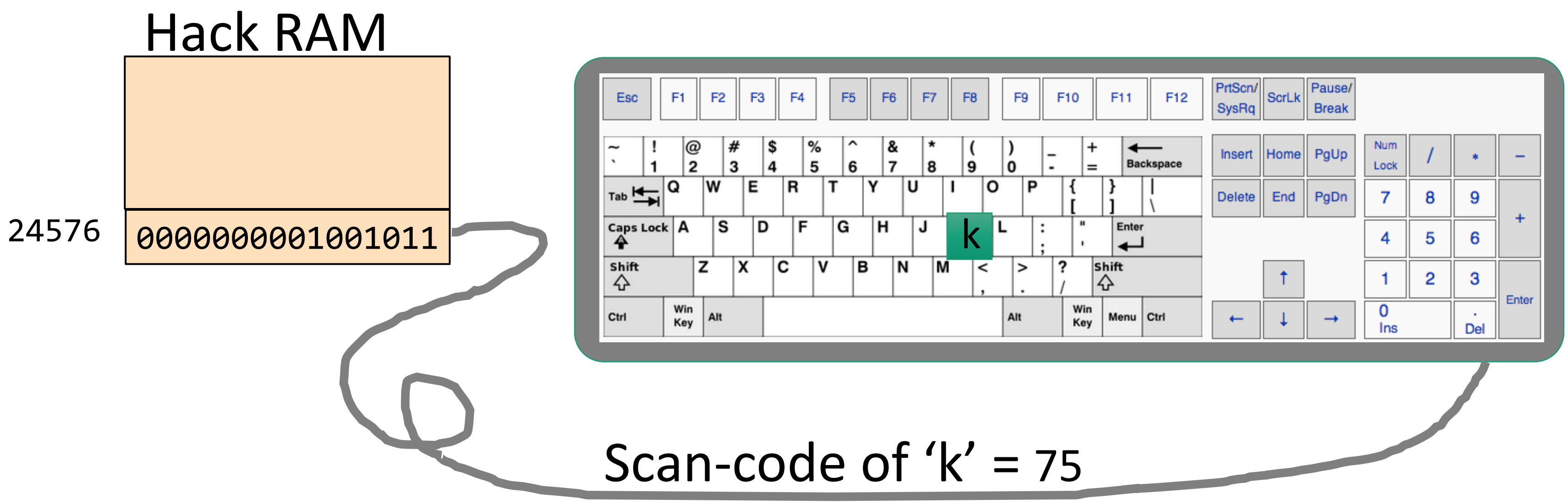
Input



<u>Hack language convention:</u>

- SCREEN: base address of the screen memory map
- KBD: address of the keyboard memory map

Handle the keyboard



<u>To check which key is currently pressed:</u>

- Read the contents of `RAM[24576]` (address `KBD`).
- If the register contains `0`, no key is pressed.
- Otherwise, the register contains the scan code of the currently pressed key.

Keyboard input (example)

```
// Example: Run an infinite loop to listen to the
     keyboard input
(LOOP)
  // check keyboard input
  @KBD
  D = M //get keyboard input

  @R0
  M=D  //set R0 to keyboard input

  //if R0 = 'esc', goto END
  @140 // 'esc' = 140
  D=A
  @R0
  D=M-D
  @END
  D;JEQ

  @LOOP
  0;JMP // an infinite loop.
(END)
  @END
  0;JMP //end
```

Comments on assembly programming

High level code

```
for (i=0; i<n; i++) {
    arr[i] = -1
}
```

Compiler

Machine language

```
    ...
    @i
    M=0
(LOOP)
    @i
    D=M
    @n
    D=D-M
    @END
    D;JEQ
    @arr
    D=M
    @i
    A=D+M
    M=-1
    @i
    M=M+1
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
```

<u>Assembly programming is:</u>

- Low-level
- Efficient (or not)
- Intellectually challenging.

Recap

- Registers and memory
  - A-instruction/C-instruction,
  - Terminate a program,
  - Built-in symbols.
- Branching, variables, iteration
  - Labels,
  - Variables, RAM[16] onwards.
  - Example: iterative processing,
  - Best practice: Pseudo-code, assembly code, trace table.
- Pointers, input/output
  - Pointers for an array,
  - Hack screen display,
  - Keyboard input.

Summary

- Why we need machine language.
  - ➢The function of machine language in the hierarchy.
- General knowledge on machine language
  - ➢Arithmetic/logic operation
  - ➢memory addressing
  - ➢program control
- Hack machine language
  - ➢A-instruction, C-instruction
  - ➢Symbolic (assembly) / binary machine language
- Programming in Hack assembly
  - ➢Registers and memory
  - ➢Branching, variables, iteration
  - ➢Pointers, input/output

Q & A

Acknowlegement

- This set of lecture notes are based on the lecture notes provided by Noam Nisam / Shimon Schocken.
- You may find more information on: www.nand2tetris.org.