



University of
Nottingham

UK | CHINA | MALAYSIA

Machine Language Part 1

Dr. Wooi Ping Cheah

A self-introduction

- Dr. Wooi Ping Cheah
 - Visiting Assistant Professor, UNNC.
 - Office: PMB323
 - Office Hour: 3pm-5pm Wednesday.

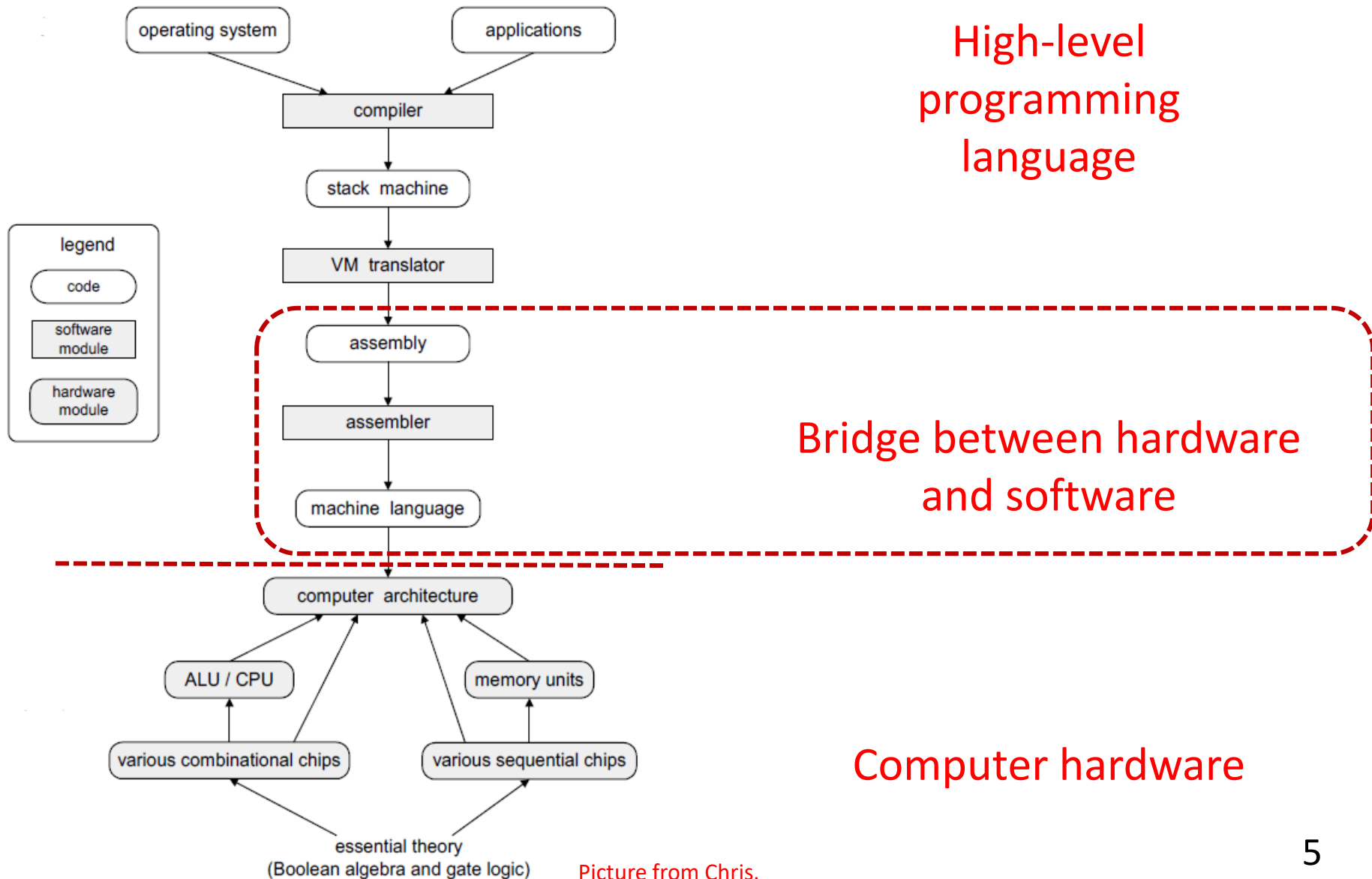
Research interests

- Soft Computing
- Knowledge Engineering
- Software Engineering
- Artificial Intelligence
- Programming Paradigms

Outlines

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

Why learning machine language?



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

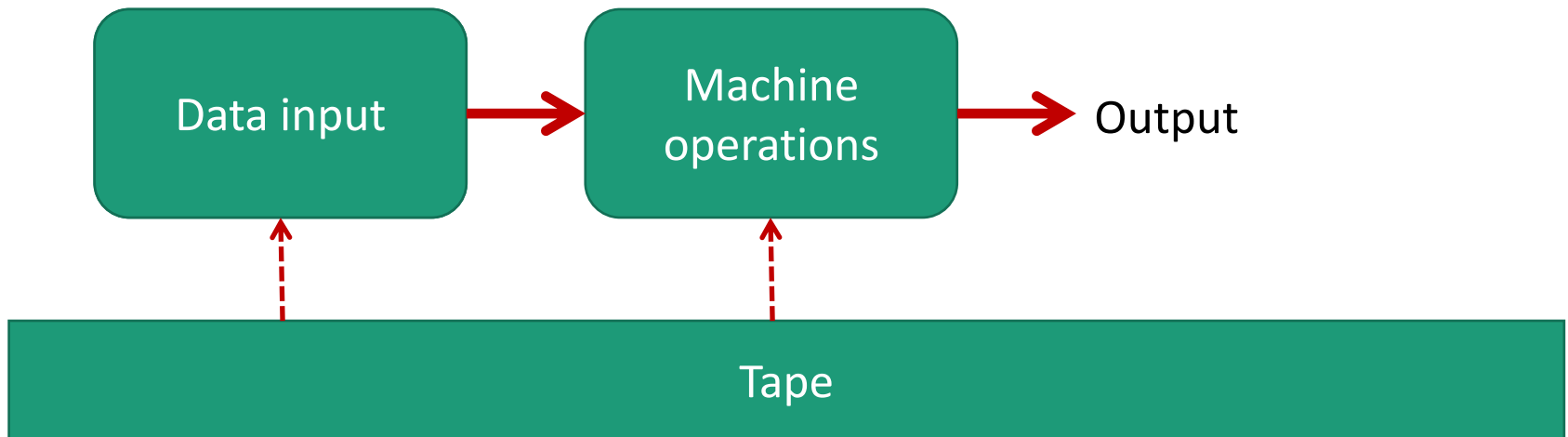


John Von Nuemann:

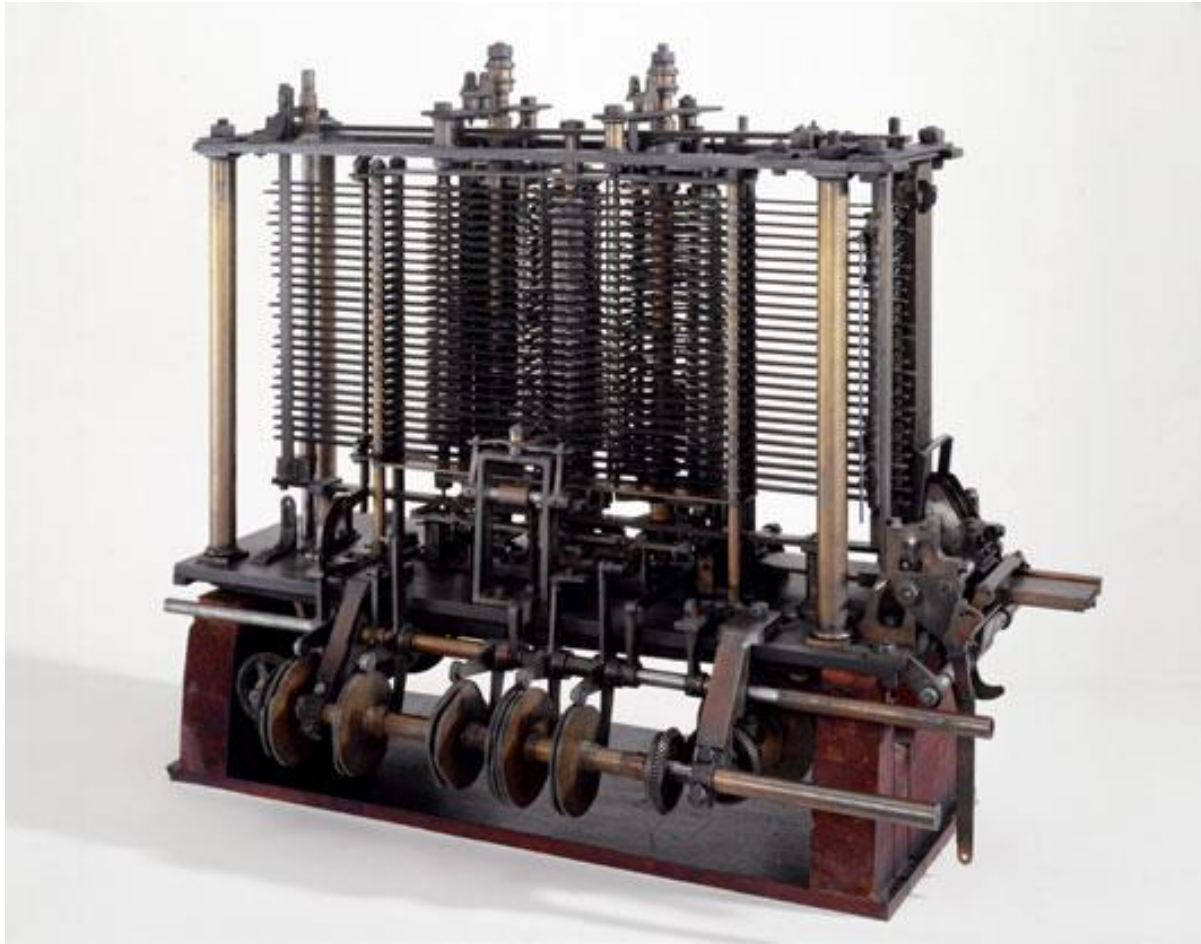
Stored Program Computer

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.

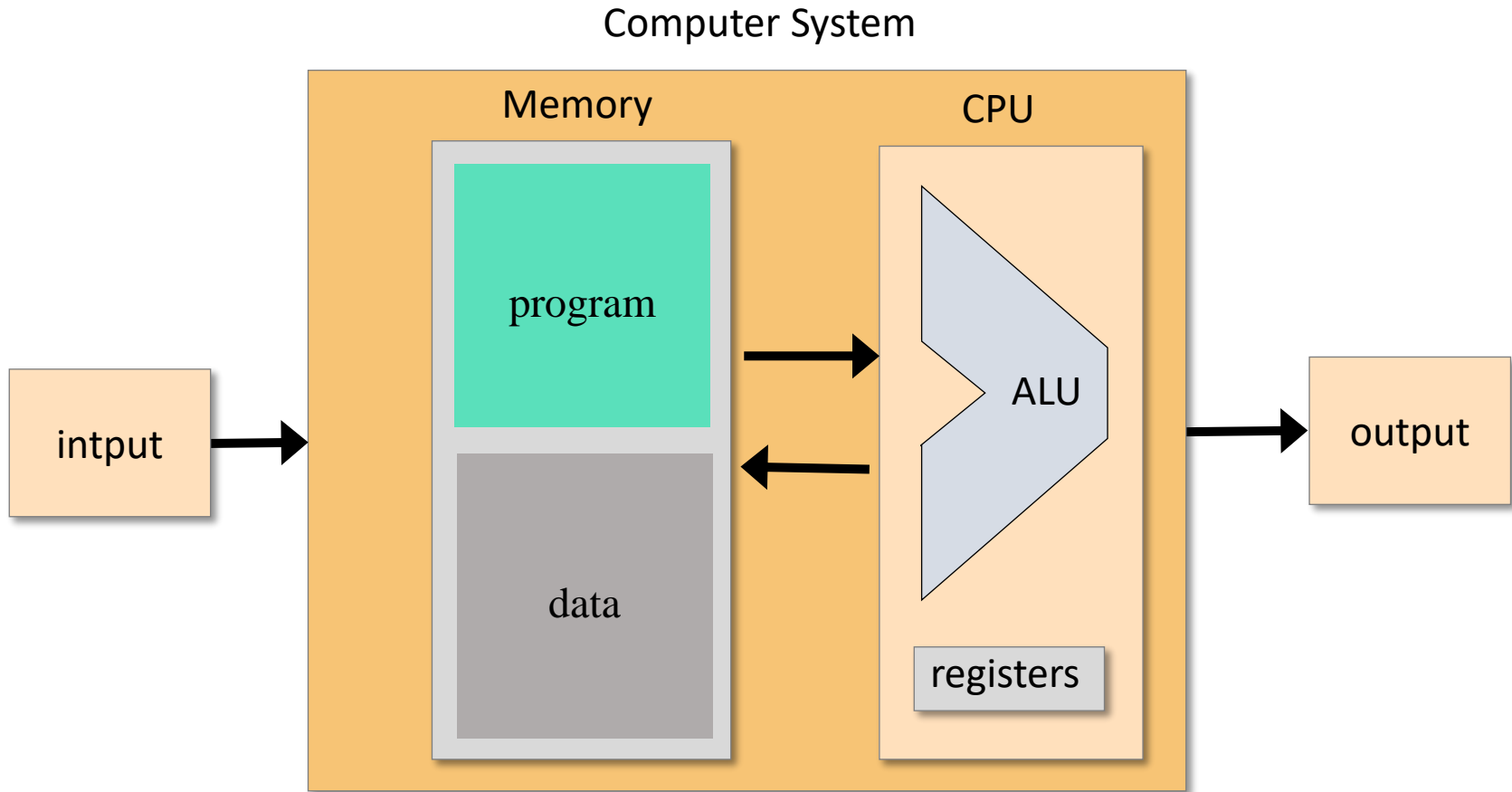


The first computer

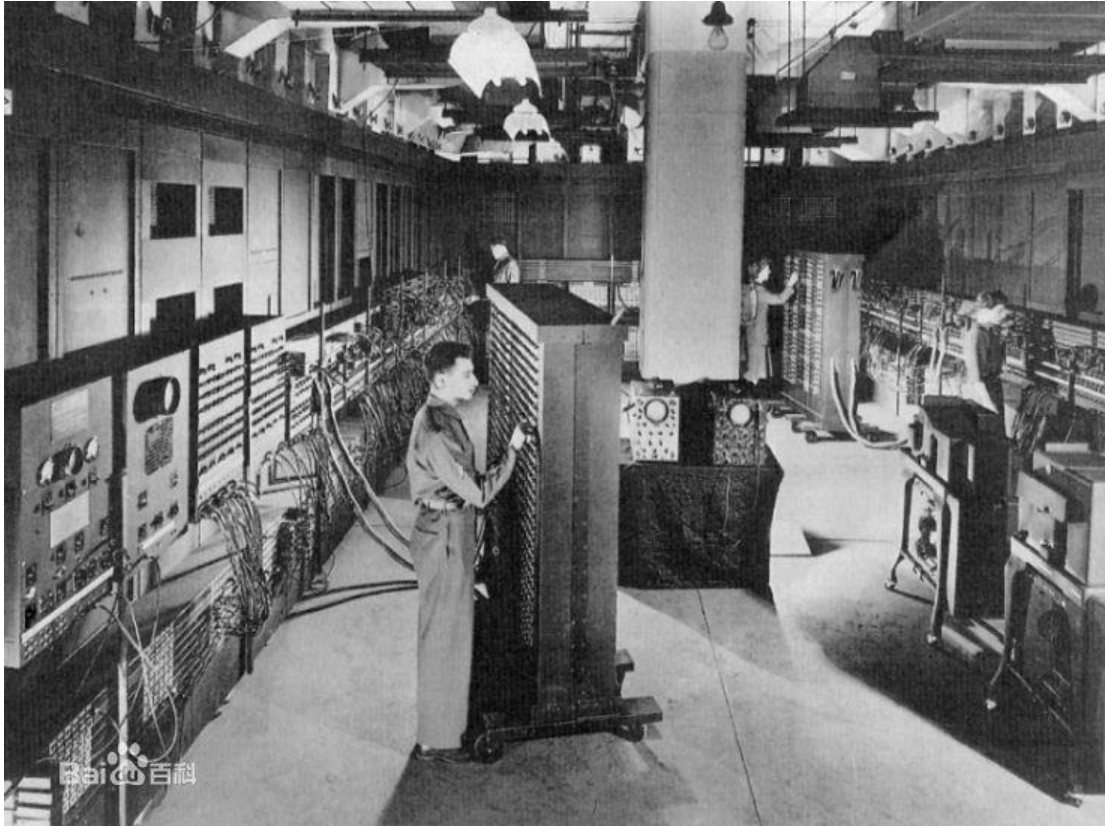


- Designed by Charles Babbage, in 1822.
- Powered by a steam engine.
- Use Punched Cards.

Stored program concept



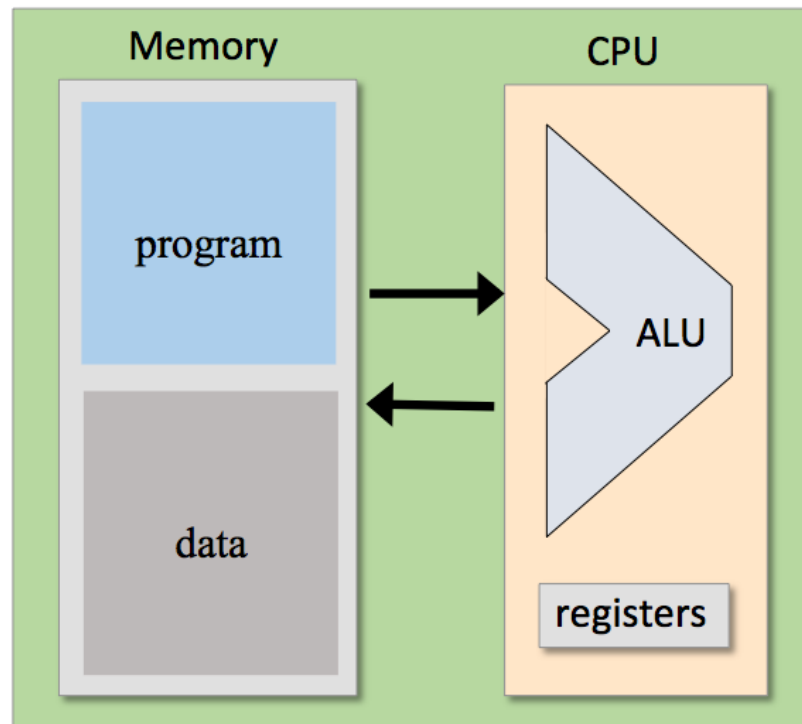
ENIAC - first general-purpose computer



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

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.**

Machine language at a glance

- Processor (CPU)

- ALU, memory access, control (branching).

- E.g. add R1, R2, R3 // $R1 \leftarrow R2 + R3$.

- Memory

- Collection of hardware devices that store data and instructions in a computer.

- E.g. load R1, 67 // $R1 \leftarrow \text{Memory}[67]$.

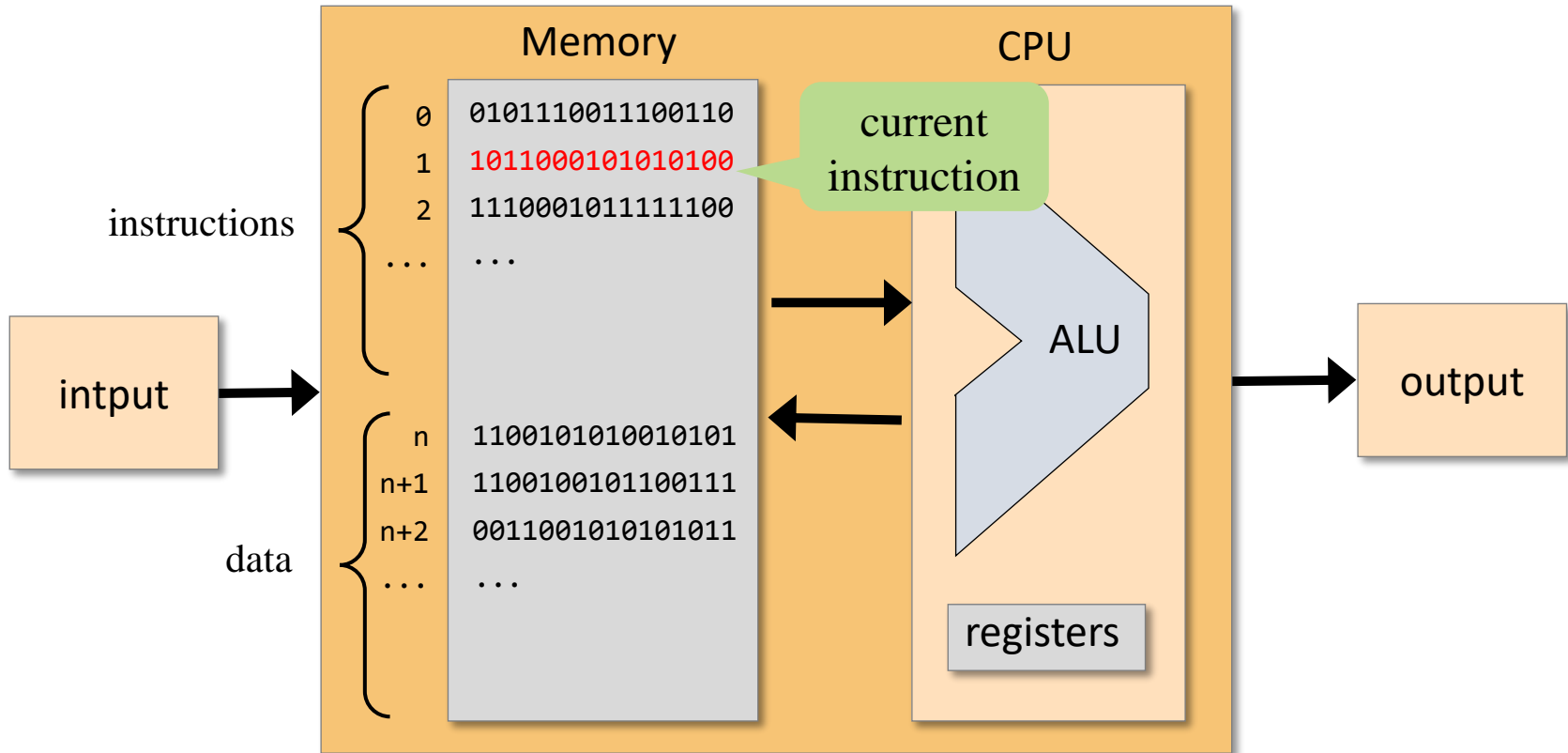
- Slow access.

- Register

- High-speed local memory.

Machine language

Computer System



Handling instructions:

- 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++;  
}
```



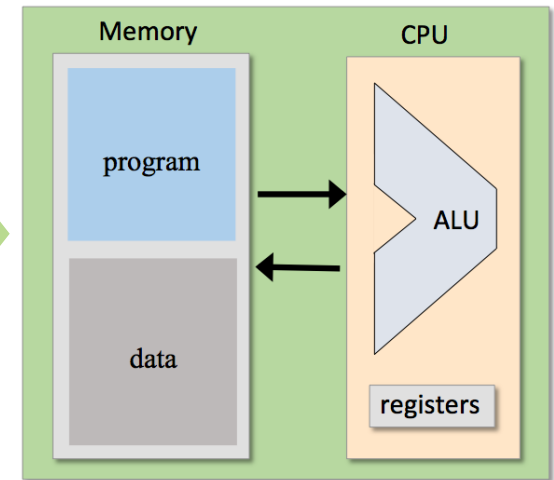
compile

machine language

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

load and
execute

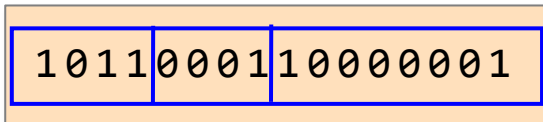
hardware



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

Machine language

Binary instruction:



add 1 Mem[129]

- Difficult to understand.

Assembly language:

add 1, Mem[129]

$\text{Mem}[129] \leftarrow \text{Mem}[129] + 1$

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

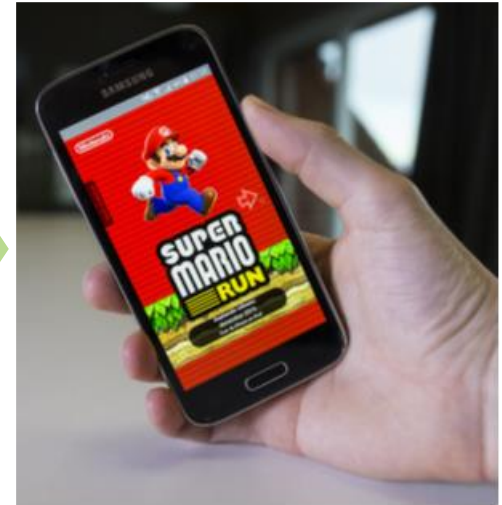
```
@i
M=1 // i = 1
@sum
M=0 // sum = 0
(LOOP)
// if i>RAM[0],
// GOTO WRITE
@i
D=M
@R0
D=D-M
@WRITE
D;JGT
... // Etc.
```

assembler

Machine Language

```
00000000000010000
1110111111001000
00000000000010001
1110101010001000
00000000000010000
1111110000010000
00000000000000000
1111010011010000
00000000000010010
1110001100000001
00000000000010000
1111110000010000
00000000000010001
...
```

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.
```

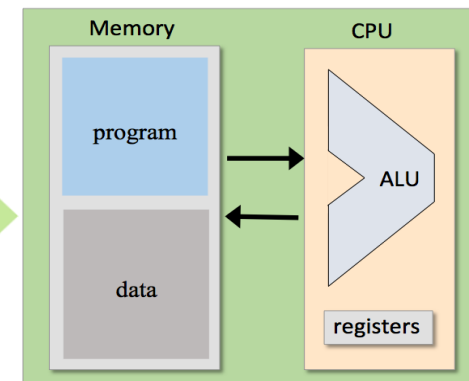
assembler

Machine Language

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

run

hardware



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 // $R1 \leftarrow R2 + \text{index}$, 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 $//0 \rightarrow 1, \text{ or } 1 \rightarrow 0.$

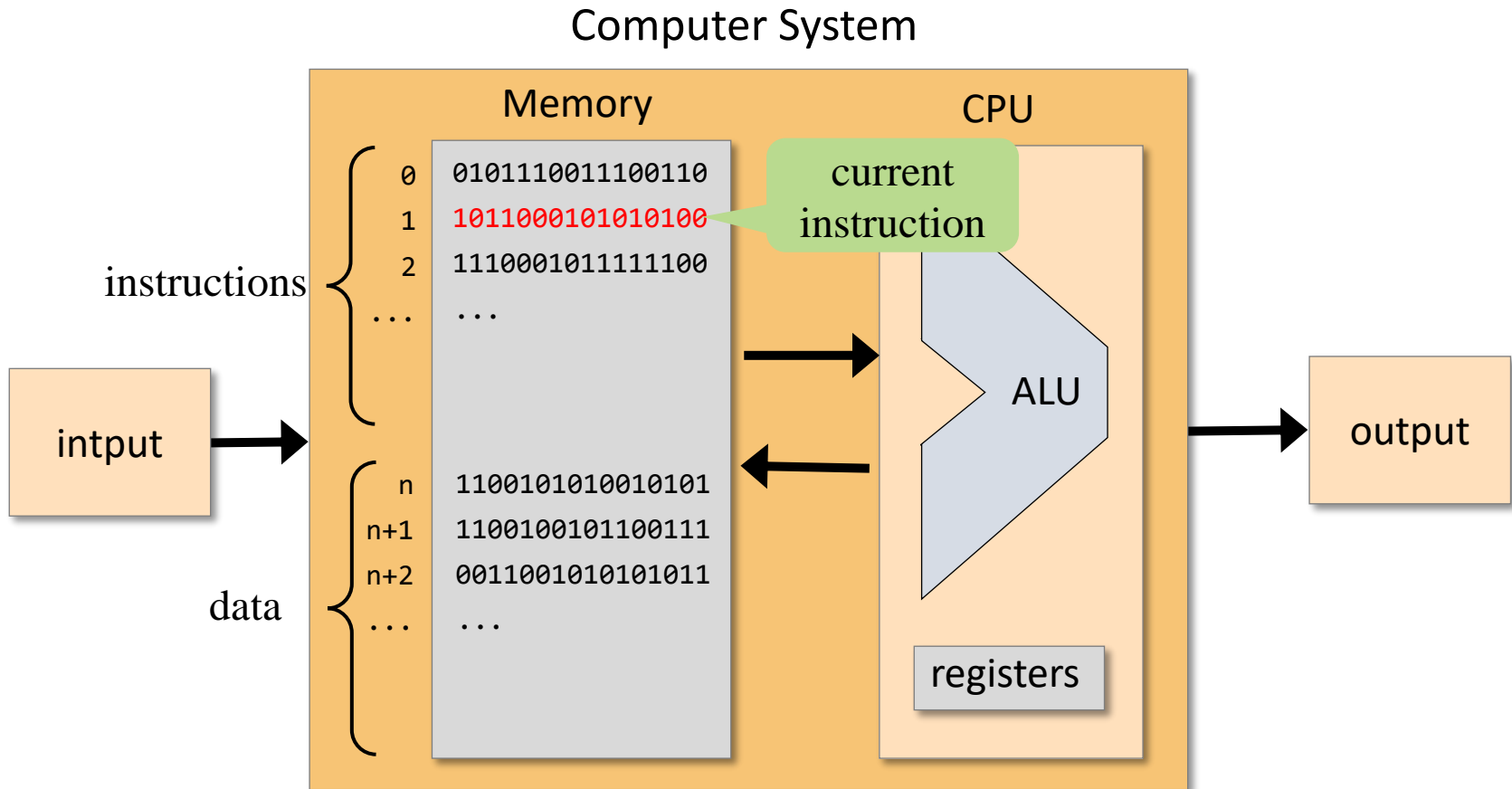
- Bit shifting $//00010111 \text{ left-shift by 2: } 01011100$

- Bitwise And, Or, etc.

- Example:

- AND R1, R1, R2 $//R1 \leftarrow \text{bitwise And of R1 and R2.}$

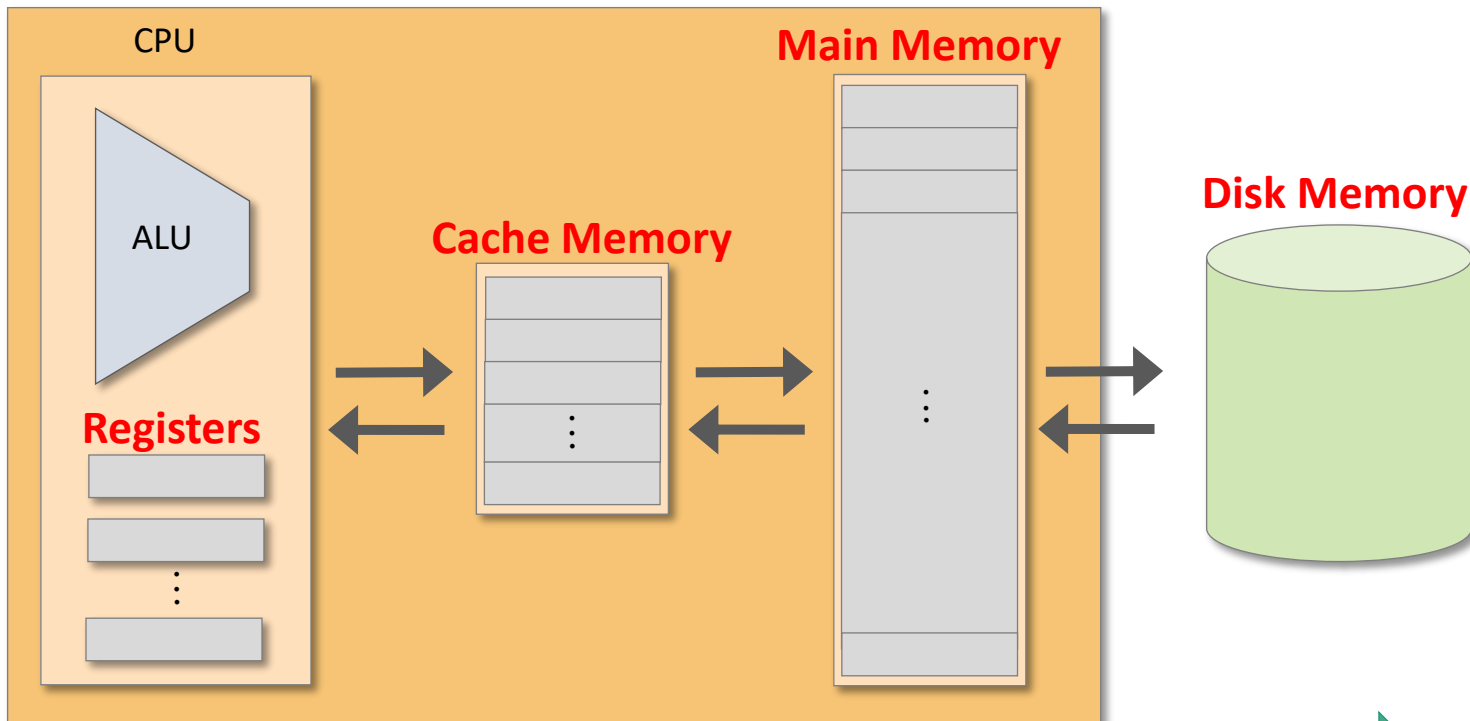
Memory access



*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:



more storage space, slower access time

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 \approx 0.4 ns.

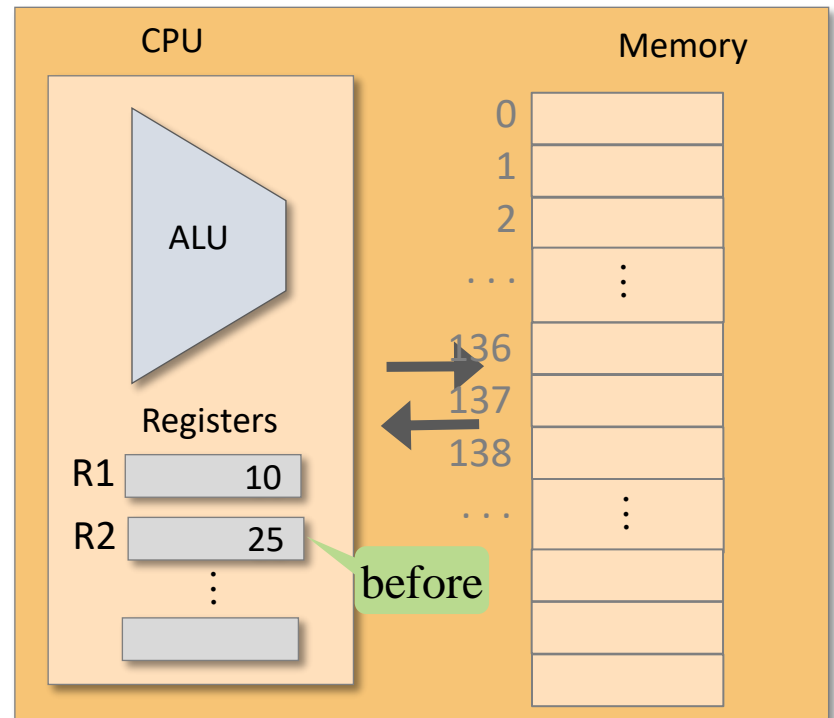
Registers

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

Data registers:

add R1, R2 // $R2 \leftarrow R1 + R2$

	R1	R2
Before add	10	25
After add		



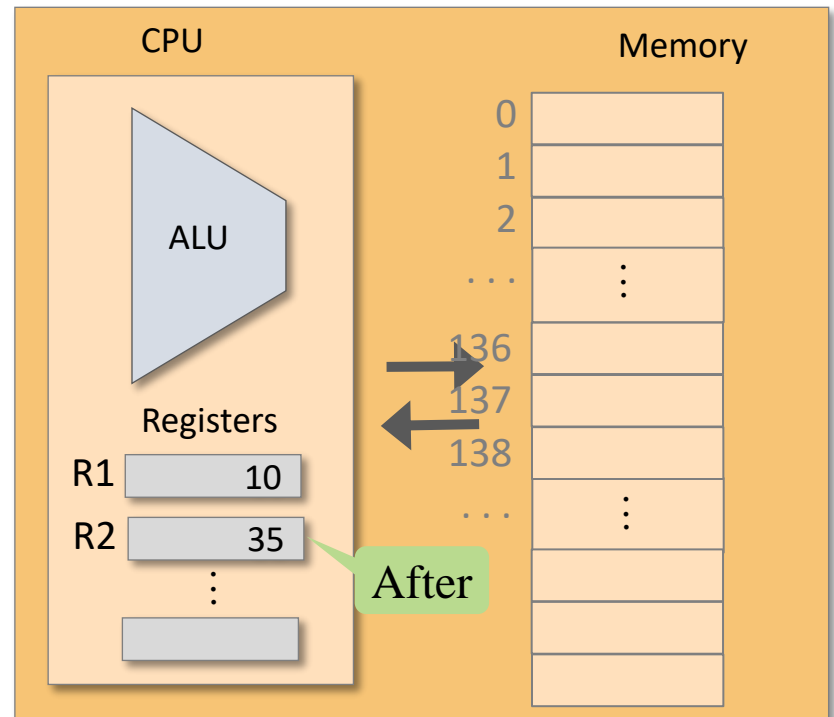
Registers

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

Data registers:

add R1, R2 // $R2 \leftarrow R1 + R2$

	R1	R2
Before add	10	25
After add	10	35



Registers

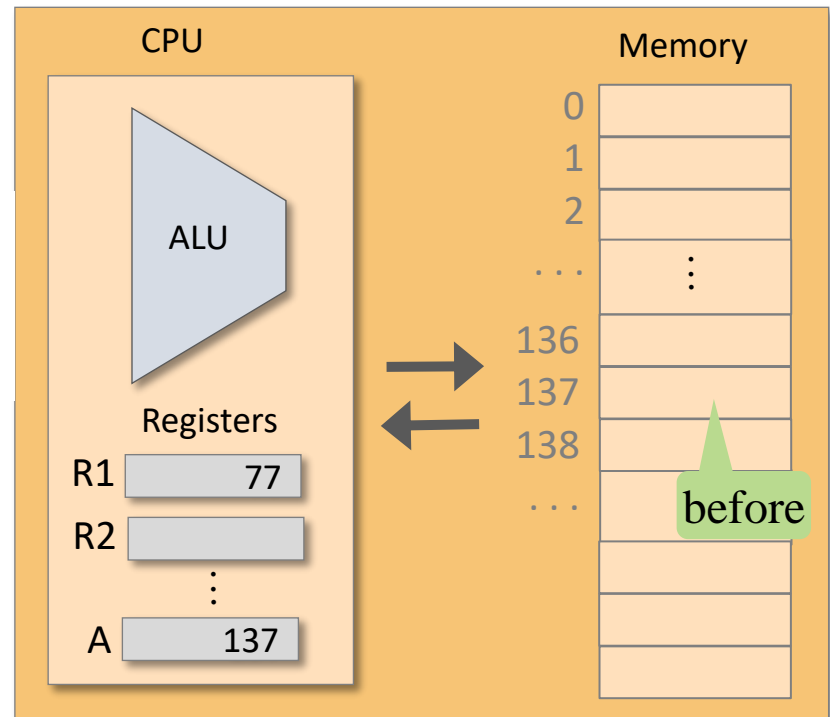
- The CPU typically contains a few, easily accessed, *registers*.
- They are the **central part** of the machine language.

Data registers:

add R1, R2 // $R2 \leftarrow R1 + R2$

Address registers:

store R1, @A // $@A \leftarrow R1$



Registers

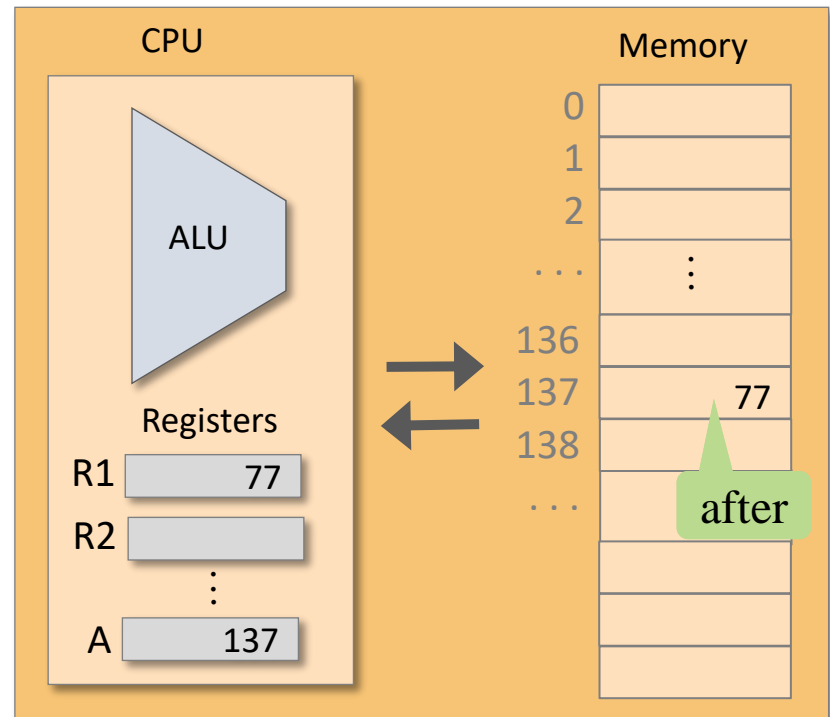
- The CPU typically contains a few, easily accessed, *registers*.
- They are the **central part** of the machine language.

Data registers:

add R1, R2 // $R2 \leftarrow R1 + R2$

Address registers:

store R1, @A // $@A \leftarrow R1$



Addressing modes

- Register

- ADD R1, R2 // $R2 \leftarrow R2 + R1$

- Access data from a **register** R2.

- Direct

- ADD R1, M[67] // $\text{Mem}[67] \leftarrow \text{Mem}[67] + R1$

- LOAD R1, 67 // $R1 \leftarrow \text{Mem}[67]$

- Access data from **fixed memory address** 67.

- Indirect

- ADD R1, @A // $\text{Mem}[A] \leftarrow \text{Mem}[A] + R1$

- Access data from **memory address specified by variable A**.

- Immediate

- ADD 67, R1 // $R1 \leftarrow R1 + 67$

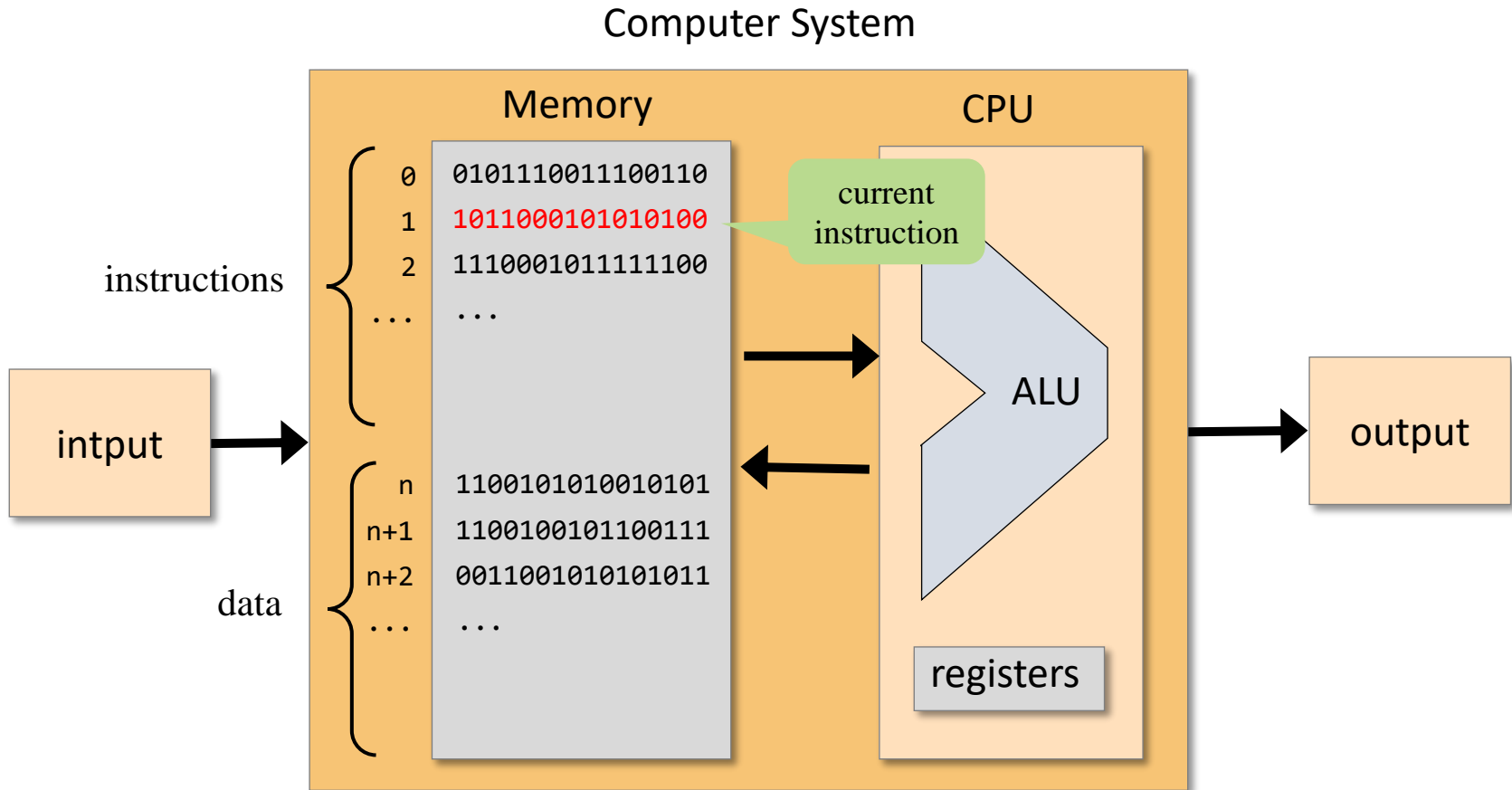
- LOADI R1, 67 // $R1 \leftarrow 67$

- 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
```

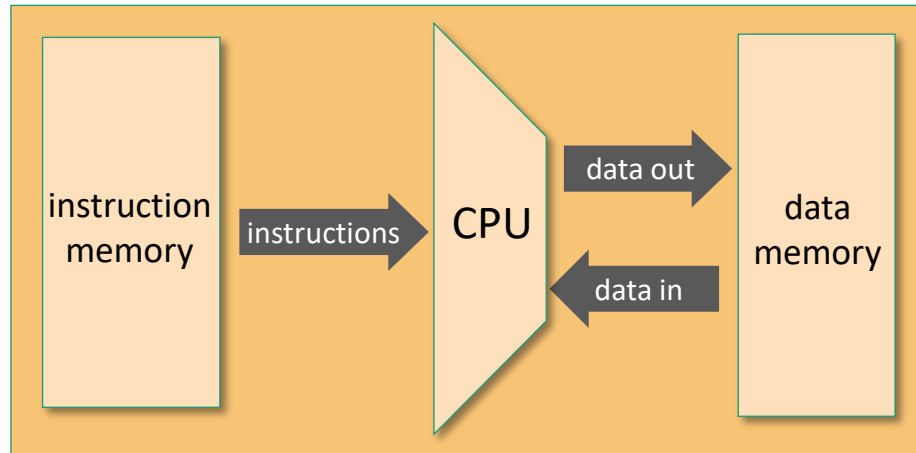
Recap

- Arithmetic and logic operations
 - Addition/subtraction,
 - Bitwise operations.
- Memory access
 - Memory hierarchy,
 - 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

CPU Emulator (2.5)

File View Run Help

The CPU Emulator (2.5) interface features a menu bar (File, View, Run, Help) and a toolbar with icons for file operations, execution (single step, break point, step over, step into), and speed control (Slow, Fast). It also includes dropdowns for 'Animate' (Program flow), 'View' (Screen), and 'Format' (Decimal).

On the left, there are two memory tables:

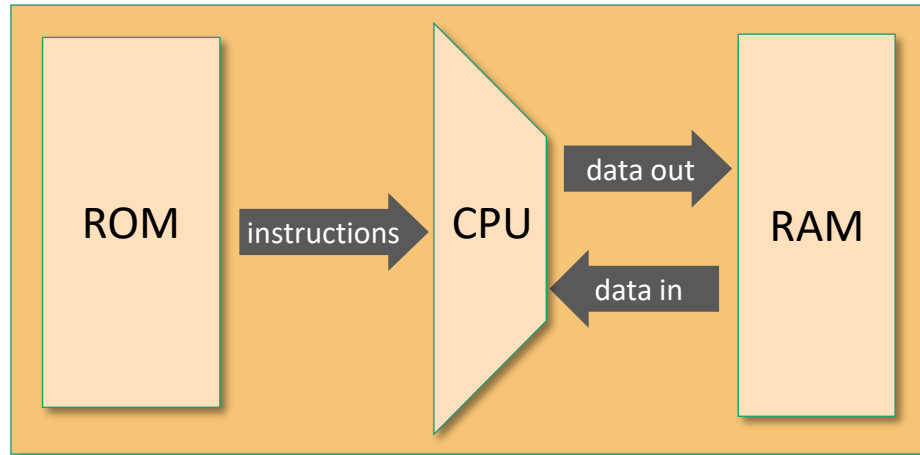
ROM	
Asm	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

RAM	
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

Below the memory tables are registers: PC (0) and A (0).

The right side of the interface shows a large screen area, a D register (0), and an ALU component. The ALU has two inputs: D Input (0) and M/A Input (0), and an output: ALU output (0).

Hack computer: software

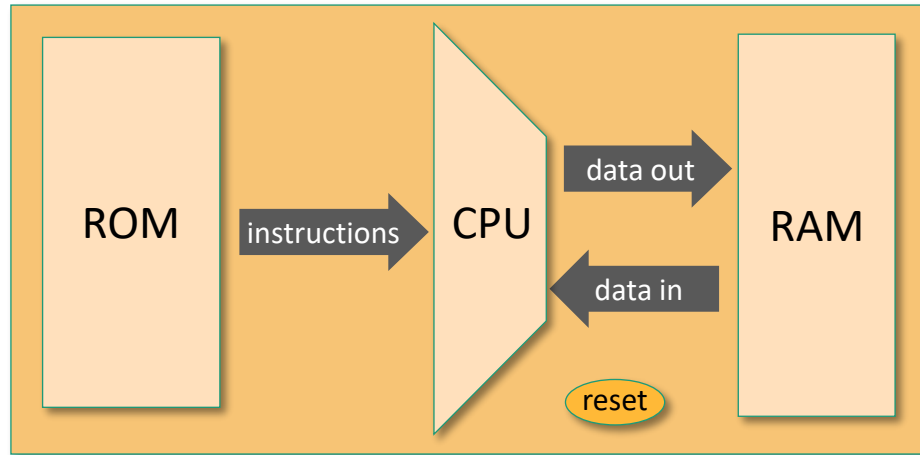


- Hack machine language:

- 16-bit A-instructions
- 16-bit C-instructions

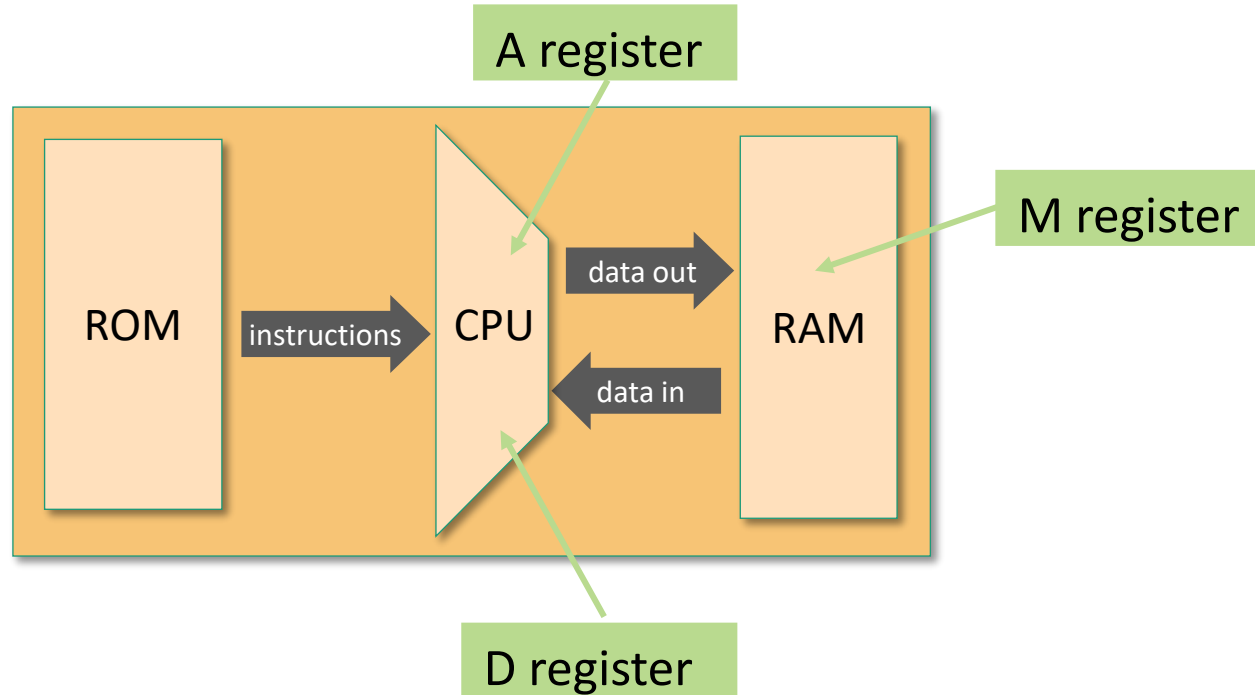
Hack program = sequence of instructions written in the
Hack machine language.

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 of 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 ← 17.*

➤ *C-instruction: command instruction*

❑ Specify desired operation.

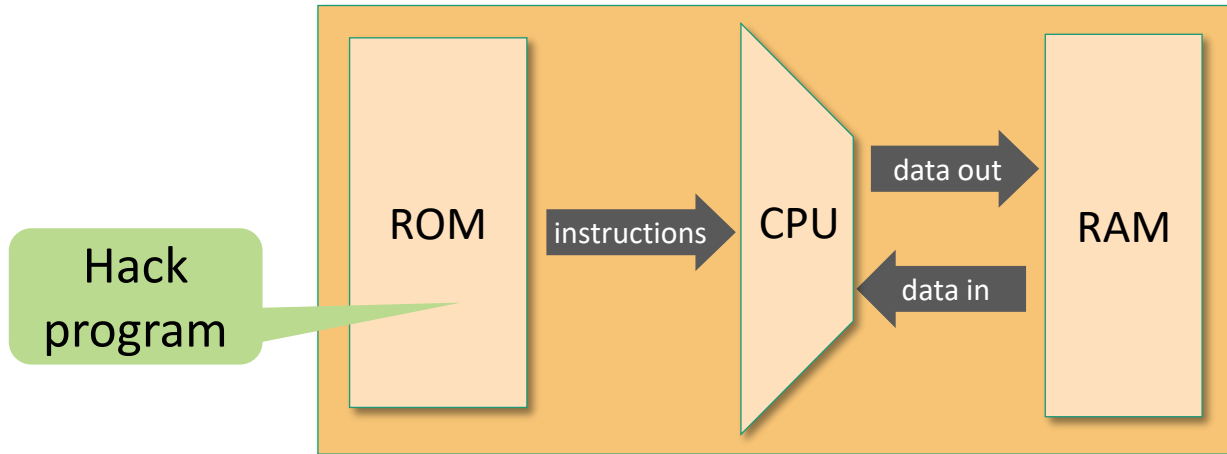
E.g., @17 *// 17 refers to memory location 17, A ← 17.*

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

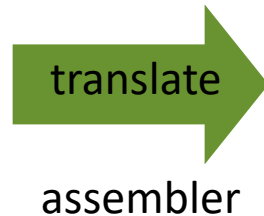


Two ways to express the same semantics:

Symbolic:

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

Assembly
language



Binary:

```
0000000000010001  
1110011111000110
```

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 $\leq 32767 (=2^{15}-1)$ 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:

0 00000000000010101

opcode signifying
an A-instruction

set A to 21

C-instruction specification

Syntax: *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

<i>comp</i>		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						

<i>dest</i>	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

<i>jump</i>	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

dest = *comp* ; *jump*

Symbolic:

MD=D+1

M=1

D+1;JLE

Binary:

1110011111011000

1110111111001000

1110011111000110

<i>comp</i>		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						

<i>dest</i>	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

<i>jump</i>	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

Hack program at a glance

Symbolic code

```
0 // Computes RAM[1] = 1+...+RAM[0]
1 // Usage: put a number in RAM[0]
2 @16 // RAM[16] represents i
3 M=1 // i = 1
4 @17 // RAM[17] represents sum
5 M=0 // sum = 0
6
7 @16
8 D=M
9 @0
10 D=D-M
11 @18 // if i>RAM[0] goto 18
12 D;JGT
13
14 @16
15 D=M
16 @17
17 M=D+M // sum += i
18 @16
19 M=M+1 // i++
20 @4 // goto 4 (loop)
21 0;JMP
22
23 @17
24 D=M
25 @1
26 M=D // RAM[1] = sum
27 @22 // program's end
28 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

```
0 // Computes RAM[1] = 1+...+RAM[0]
1 // Usage: put a number in RAM[0]
2 @16 // RAM[16] represents i
3 M=1 // i = 1
4 @17 // RAM[17] represents sum
5 M=0 // sum = 0
6
7 @16
8 D=M
9 @0
10 D=D-M
11
12 @18 // if i>RAM[0] goto 18
13 D;JGT
14
15 @16
16 D=M
17 @17
18 M=D+M // sum += i
19 @16
20 M=M+1 // i++
21 @4 // goto 4 (loop)
22 0;JMP
23
24 @17
25 D=M
26 @1
27 M=D // RAM[1] = sum
28 @22 // program's end
29 0;JMP // infinite loop
```

Binary code

```
00000000000010000
1110111111001000
00000000000010001
1110101010001000
00000000000010000
11111100000010000
00000000000000000
1111010011010000
00000000000010001
11100011000000001
00000000000010000
11111100000010000
00000000000010001
1111000010001000
00000000000010000
1111110111001000
00000000000000100
1110101010000111
00000000000010001
11111100000010000
00000000000000001
1110001100001000
00000000000010101
1110101010000111
```

translate

assembler

execute

Acknowledgement

- 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.