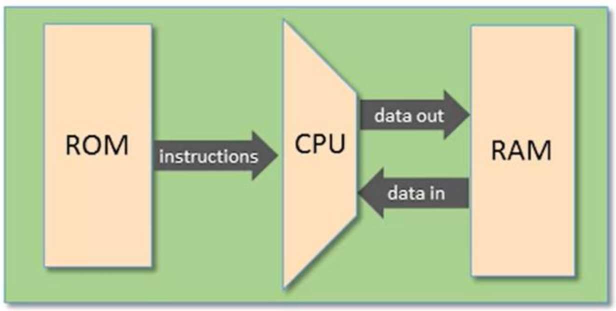
Week 4 – Hack Assembly Language

Hack computer: software



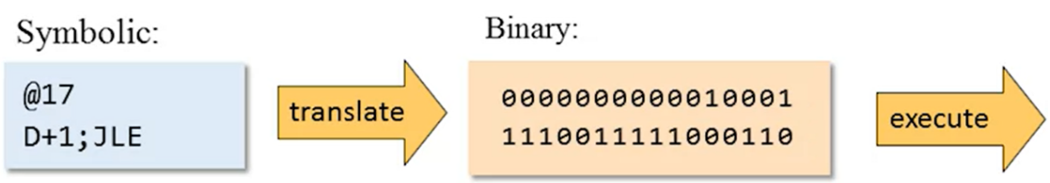
Hack machine language:

- A-instructions
- C-instructions

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

Two ways to express the same semantics:

- Binary code
- Symbolic language



The A-instruction: symbolic and binary syntax

<u>Semantics:</u> Set the A register to *value*

<u>Symbolic syntax:</u>

```
@value
```

Where *value* is either:

- a non-negative decimal constant ≤ 32767 ($=2^{15}-1$) or
- a symbol referring to such a constant (later)

Example:

```
@21
```

Effect: sets the A register to 21

<u>Binary syntax:</u>

```
0value
```

Where *value* is a 15-bit binary number

Example:

```
0000000000010101
```

Effect: sets the A register to 21

The C-instruction: symbolic and binary syntax

Symbolic syntax:

```
dest = comp ; jump
```

Binary syntax:

$$\underbrace{1}_{\text{op code}}\ \underbrace{1\ 1}_{\text{not used}}\ \underbrace{a\ c1\ c2\ c3\ c4\ c5\ c6}_{\textit{comp}\text{ bits}}\ \underbrace{d1\ d2\ d3}_{\textit{dest}\text{ bits}}\ \underbrace{j1\ j2\ j3}_{\textit{jump}\text{ bits}}$$

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

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

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						

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

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

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

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

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

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

The C-instruction: symbolic and binary syntax

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

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

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

Hack programs: symbolic and binary

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
@17    // if i>RAM[0] goto 17
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
@21    // program's end
0;JMP  // infinite loop
```

translate

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
```

Hack assembly instructions

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
```

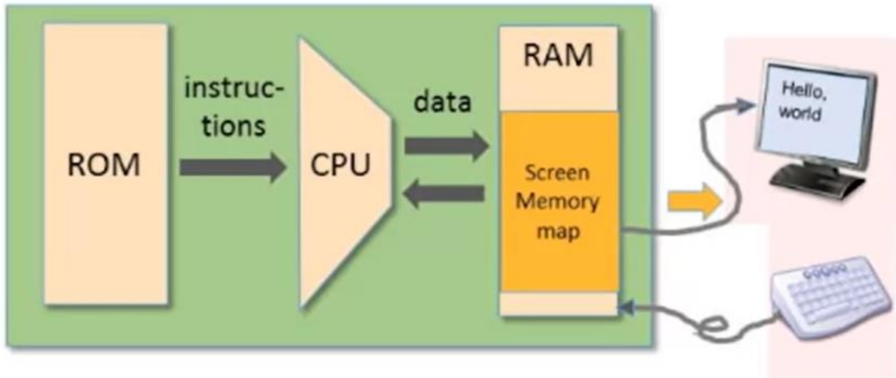
(`M` refers to `RAM[A]`)

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

Semantics:

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

Hack computer platform: Output



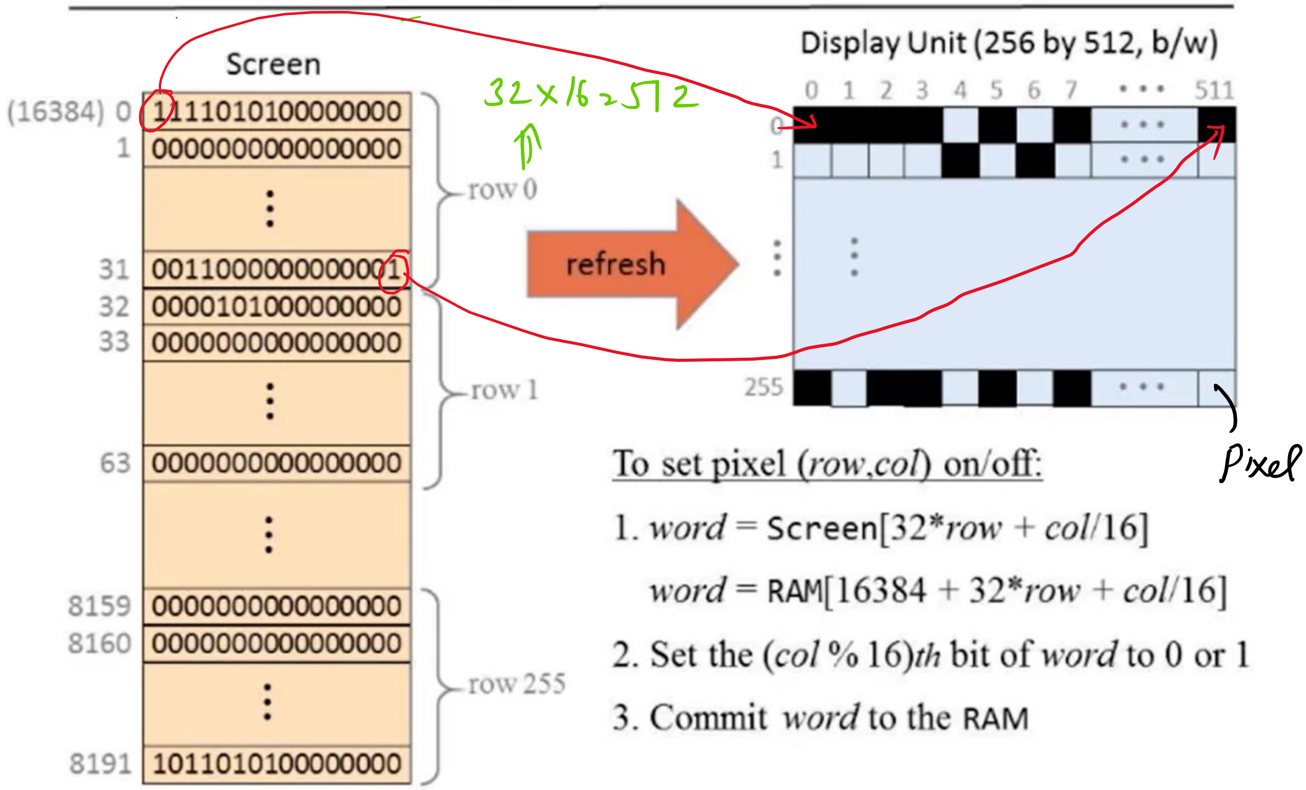
<u>Screen memory map</u>

A designated memory area, dedicated to manage a display unit

The physical display is continuously *refreshed* from the memory map, many times per second

Output is effected by writing code that manipulates the screen memory map.

Screen memory map



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

1. *word* = Screen[32**row* + *col*/16]

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

2. Set the (*col* % 16)*th* bit of *word* to 0 or 1
3. Commit *word* to the RAM

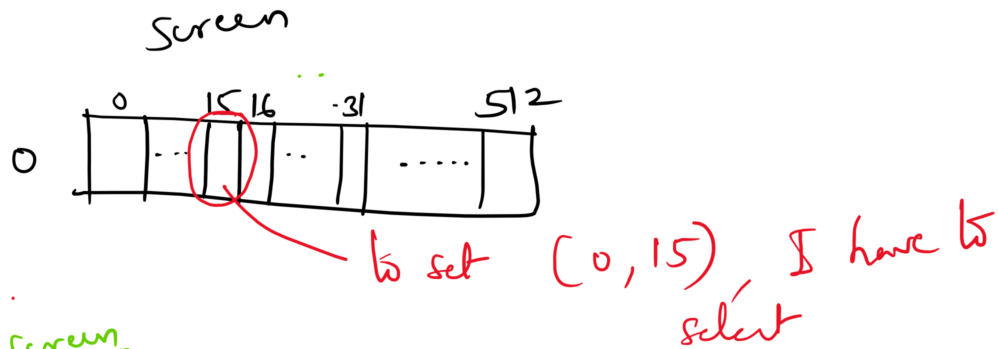
① Row 1 → Represented by 1st 32 registers.
Row 2 → next 32 Registers and so on.

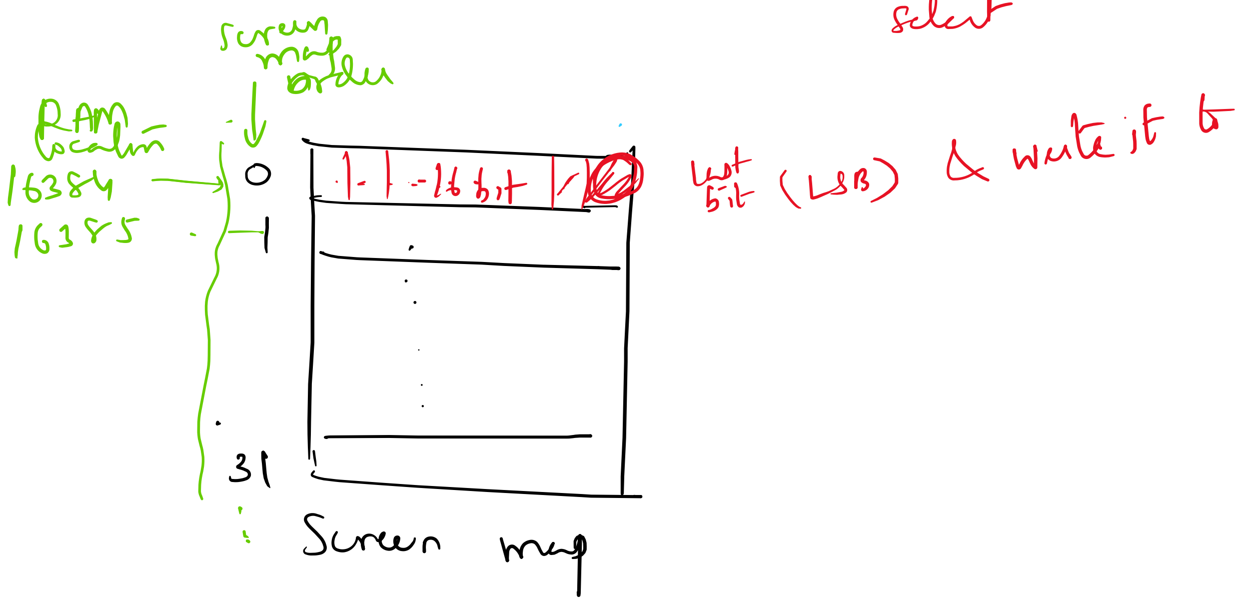
② $\underbrace{8191}_{\text{registers}} \times 16 = 256 \times 512$

③ To select the pixel (row, col)

→ Selecting a row (has 16 values) means to select the corresponding row in screen.

eg.





⇒ But we know that we can't change a single bit, instead we have to access that RAM register and change the word stored in the register.

⇒ So to change (0, 15) we will select the RAM location 16384.

⇒ Then we set the last bit of the current word & then load it back to the register.

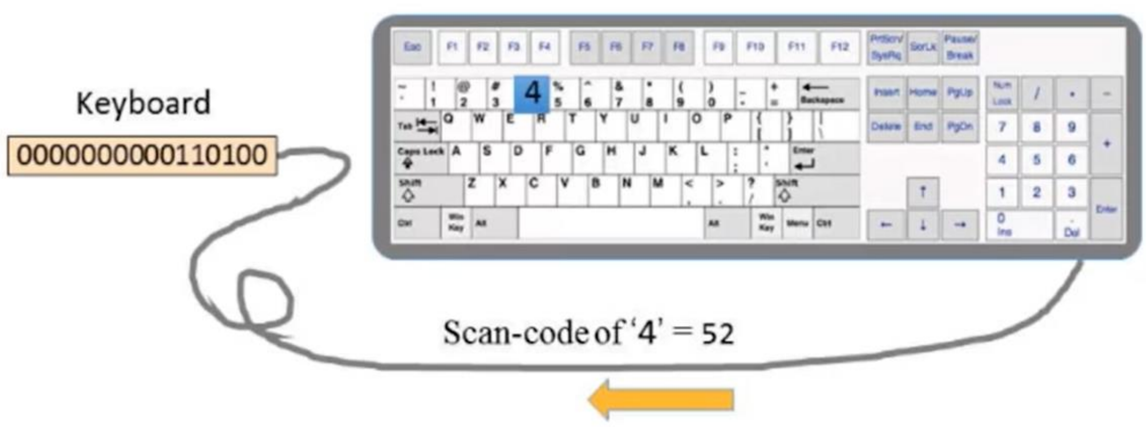
(1) So in general this can be written as

* Word = RAM[16384 + 32*row + col/16]
* Set the (col % 16) th bit of word to 0 or 1.
* Commit word to RAM.

0 ⇒ off

1 ⇒ ON

Keyboard memory map



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

The Hack character set

Key	Code
0	48
1	49
...	...
9	57

Key	Code
A	65
B	66
...	...
Z	90

When no key is pressed, the resulting code is 0

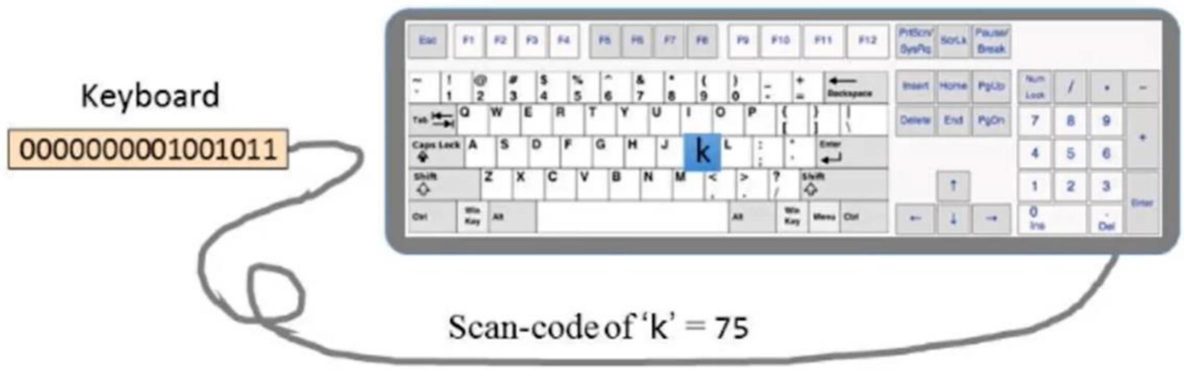
Key	Code
(space)	32
!	33
“	34
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

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

Key	Code
[	91
/	92
]	93
^	94
_	95

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

Keyboard memory map



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

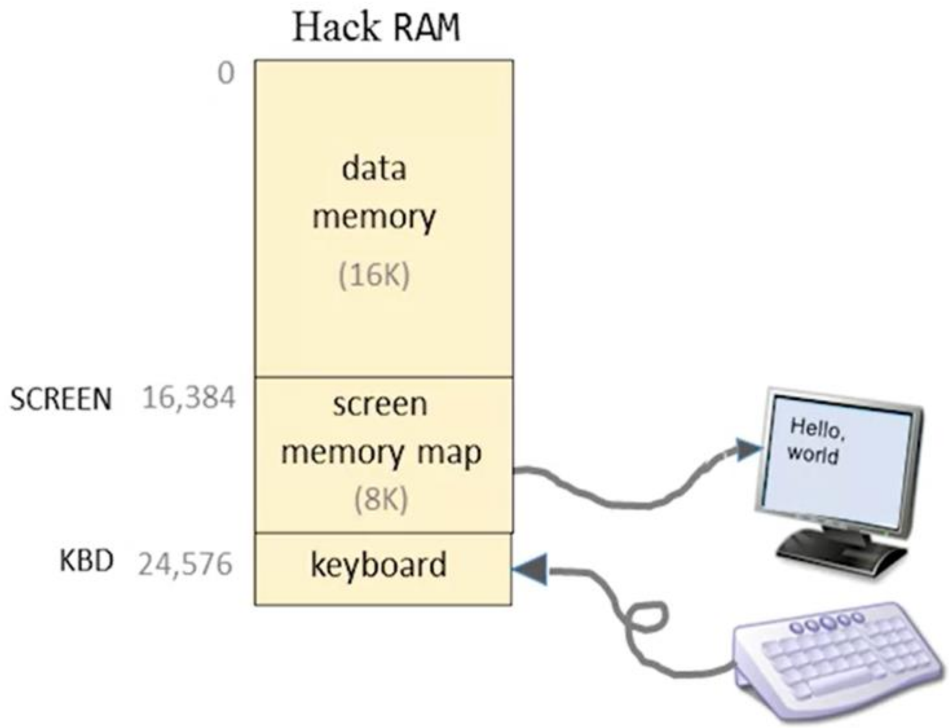
- Probe the contents of the `Keyboard` chip
- In the Hack computer: probe the contents of `RAM[24576]`

If the register contains 0, no key is pressed.

1. Only single register is required.
   16 bits ⇒ 2^{16} combinations. [That is more than required to represent all keys in a keyboard].

2. Location : RAM [24576]

Input / output



Hack language convention:

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