University of Nottingham
UK | CHINA | MALAYSIA

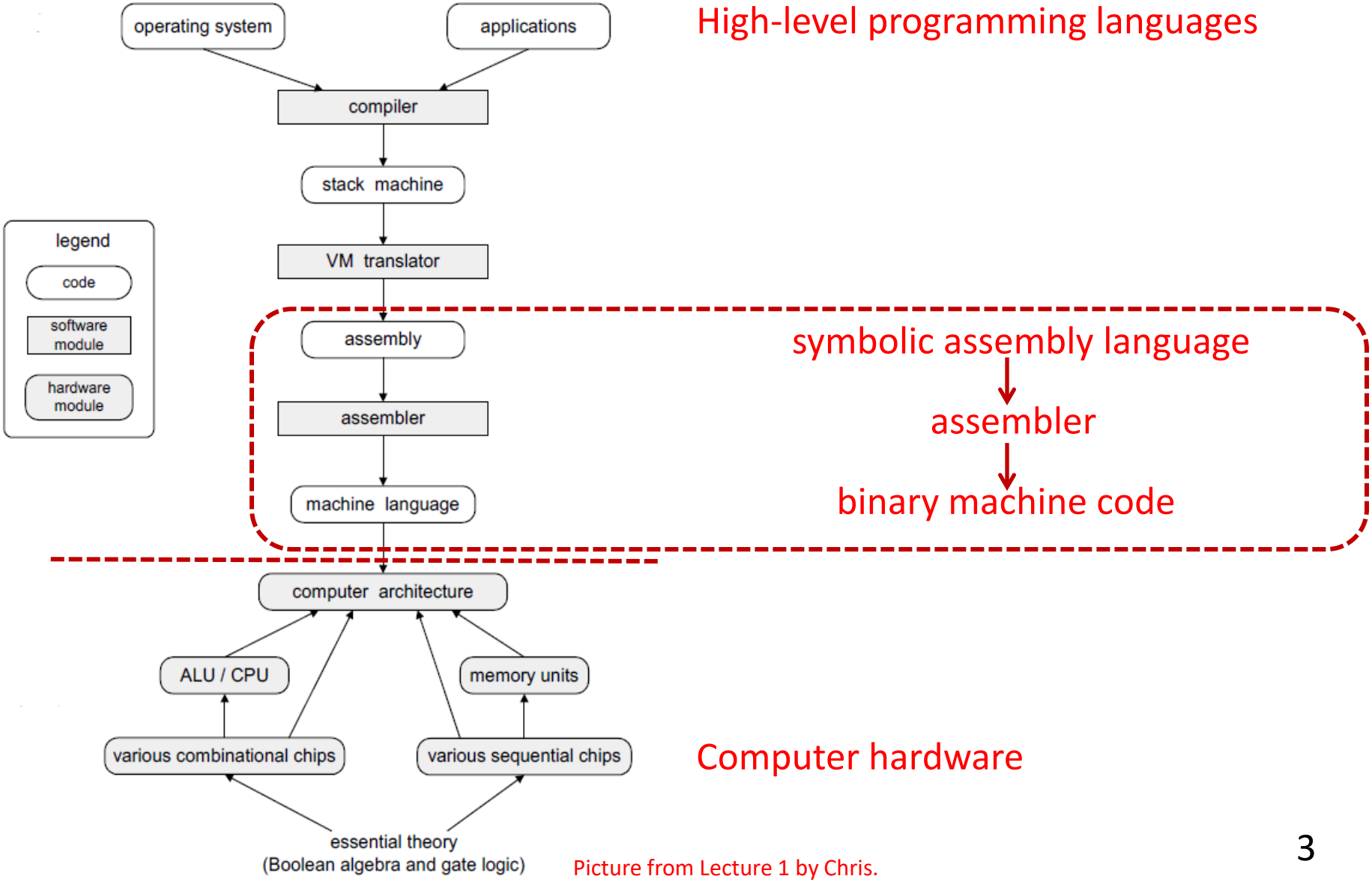
Assembler and Input-Output

Dr. Wooi Ping Cheah

Outlines

- Introduction to assembler
- Translate Hack assembly program
  - Translate program without symbols
  - Translate program with symbols
- Hack input / output

Overview of computer system



Picture from Lecture 1 by Chris.

The translator's challenge (overview)

Hack assembly code

(source language)

```
// Computes RAM[1]=1+...+RAM[0]
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0

(LOOP)
   @i    // if i>RAM[0] goto STOP
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   @i    // sum += i
   D=M
   @sum
   M=D+M
   @i    // i++
   M=M+1
   @LOOP // goto LOOP
   0;JMP
   ...
```

Assembler

What are the rules of the game?

Hack binary code

(target language)

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1110000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
...
```

The translator's challenge (overview)

Hack assembly code

(source language)

```
// Computes RAM[1]=1+...+RAM[0]
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0

(LOOP)
   @i    // if i>RAM[0] goto STOP
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   @i    // sum += i
   D=M
   @sum
   M=D+M
   @i    // i++
   M=M+1
   @LOOP // goto LOOP
   0;JMP
   ...
```

Assembler

Based on the syntax rules of:

- The source language
- The target language

Hack binary code

(target language)

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1110000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
...
```

Hack language specification: A-instruction

Symbolic syntax:

`@ value`

Examples:

`@21`

`@foo`

Where *value* is either

- a non-negative decimal constant or
- a symbol referring to such a constant

Binary syntax:

`0 valueInBinary`

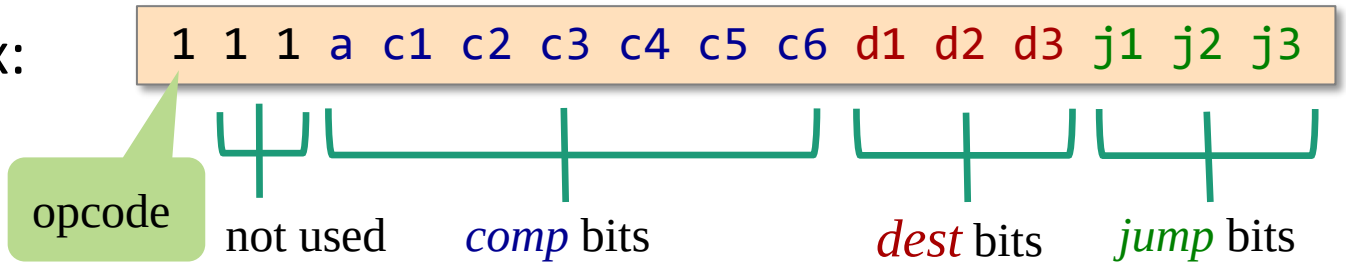
Example:

`0000000000010101`

Hack language specification: C-instruction

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

Binary syntax:



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 language specification: symbols

Pre-defined symbols:

symbol	value	symbol	value
R0	0	SP	0
R1	1	LCL	1
R2	2	ARG	2
...	...	THIS	3
R15	15	THAT	4
SCREEN	16384		
KBD	24576		

Label declaration: (*label*)

Variable declaration: *@variableName*

```
// Computes RAM[1]=1+...+RAM[0]
   @i    //variable
   M=1   // i = 1
   @sum
   M=0   // sum = 0

(LOOP) //label
   @i    // if i>RAM[0] goto STOP
   D=M
   @R0 //built-in symbols
   D=D-M
   @STOP
   D;JGT
   @i    // sum += i
   D=M
   @sum
   M=D+M
   @i    // i++
   M=M+1
   @LOOP // goto LOOP
   0;JMP
   ...
```

The Hack language: a translator's perspective

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
```

Assembly program elements:

- White space
  - Empty lines / indentation
  - Line comments
  - In-line comments
- Instructions
  - A-instructions
  - C-instructions
- Symbols
  - Predefined symbols
  - Variables
  - Labels

Outlines

- Introduction to assembler
- Translate Hack assembly program
  - Translate program without symbols
  - Translate program with symbols
- Hack input / output

Handling programs without symbols

Assembly program (without symbols)

```
    // Computes RAM[1] = 1 + ... +
          RAM[0]
0       @16
1       M=1   // i = 1
2       @17
3       M=0   // sum = 0

4       @16   // if i>RAM[0] goto STOP
5       D=M
6       @0
7       D=D-M
8       @18
9       D;JGT
10      @16    // sum += i
11      D=M
12      @17
13      M=D+M
14      @16    // i++
15      M=M+1
16      @4     // goto LOOP
17      0;JMP
18      @17
19      D=M
20      @1
21      M=D    // RAM[1] = the sum
22      @22
23      0;JMP
```

Assembler for symbol-less Hack programs

Challenges:

Handling…

- White space
- Instructions

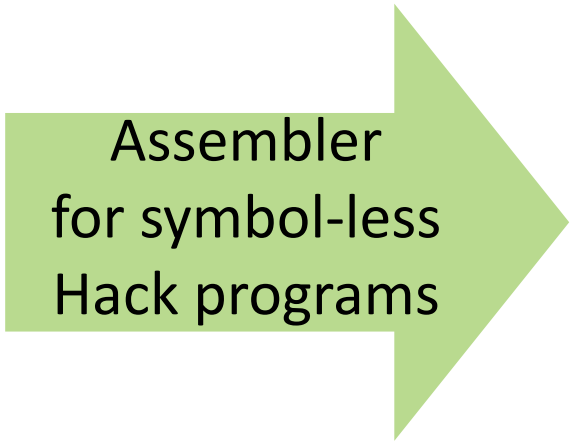
Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

Handling programs without symbols

Assembly program (without symbols)

```
0     @16
1     M=1
2     @17
3     M=0
4     @16
5     D=M
6     @0
7     D=D-M
8     @18
9     D;JGT
10    @16
11    D=M
12    @17
13    M=D+M
14    @16
15    M=M+1
16    @4
17    0;JMP
18    @17
19    D=M
20    @1
21    M=D
22    @22
23    0;JMP
```



Challenges:

Handling…

- White space
  - Ignore it
- Instructions

Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

Translating A-instructions

Symbolic syntax:

`@value`

Examples:

`@21`

`@foo`

Where *value* is either

- a non-negative decimal constant or
- a symbol referring to such a constant

Binary syntax:

`0valueInBinary`

Example:

`0000000000010101`

Translation to binary:

- If *value* is a **decimal constant**, generate the equivalent **binary constant**.
- If *value* is a symbol, later.

Translating C-instructions

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

Example:

Symbolic: `MD=D+1`

Binary: `1 1 1 0 0 1 1 1 1 1 0 1 1 0 0 0`

Outlines

- Introduction to assembler
- Translate Hack assembly program
  - Translate program without symbols
  - Translate program with symbols
- Hack input / output

Handling symbols

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0

(LOOP)
   @i    // if i>RAM[0] goto STOP
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   @i    // sum += i
   D=M
   @sum
   M=D+M
   @i    // i++
   M=M+1
   @LOOP // goto LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D  // RAM[1] = the sum
(END)
   @END
   0;JMP
```

Pre-defined symbols:

Represent special memory locations.

Label symbols:

Represent destinations of goto instructions.

Variable symbols:

Represent memory locations where the programmer wants to maintain values.

Handling pre-defined symbols

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0

(LOOP)
   @i    // if i>RAM[0] goto STOP
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   @i    // sum += i
   D=M
   @sum
   M=D+M
   @i    // i++
   M=M+1
   @LOOP // goto LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D  // RAM[1] = the sum
(END)
   @END
   0;JMP
```

The Hack language specification describes 23 *pre-defined symbols*:

symbol	value	symbol	value
R0	0	SP	0
R1	1	LCL	1
R2	2	ARG	2
...	...	THIS	3
R15	15	THAT	4
SCREEN	16384		
KBD	24576		

Translating *@preDefinedSymbol* :

Replace *preDefinedSymbol* with its value.

Examples

Symbolic: `@R0`

Binary: `0000000000000000`

Handling label symbols

Assembly program

```
   // Computes RAM[1] = 1 + ... + RAM[0]
0     @i
1     M=1   // i = 1
2     @sum
3     M=0   // sum = 0

      (LOOP)
4     @i    // if i>RAM[0] goto STOP
5     D=M
6     @R0
7     D=D-M
8     @STOP
9     D;JGT
10    @i    // sum += i
11    D=M
12    @sum
13    M=D+M
14    @i    // i++
15    M=M+1
16    @LOOP // goto LOOP
17    0;JMP
      (STOP)
18    @sum
19    D=M
20    @R1
21    M=D  // RAM[1] = the sum
      (END)
22    @END
23    0;JMP
```

Label symbols

- Used to label destinations of goto commands,
- Declared by the pseudo-command **(xxx),**
- This directive defines the symbol **xxx,** to refer to the **memory location** holding the next instruction in the program.

symbol	value
LOOP	4
STOP	18
END	22

Translating *@labelSymbol* :

Replace *labelSymbol* with its value.

Examples

Symbolic: `@LOOP`

Binary: `0000000000000100`

Handling variable symbols

Assembly program

```
   // Computes RAM[1] = 1 + ... + RAM[0]
0     @i
1     M=1   // i = 1
2     @sum
3     M=0   // sum = 0

   (LOOP)
4     @i    // if i>RAM[0] goto STOP
5     D=M
6     @R0
7     D=D-M
8     @STOP
9     D;JGT
10    @i    // sum += i
11    D=M
12    @sum
13    M=D+M
14    @i    // i++
15    M=M+1
16    @LOOP // goto LOOP
17    0;JMP
   (STOP)
18    @sum
19    D=M
20    @R1
21    M=D  // RAM[1] = the sum
   (END)
22    @END
23    0;JMP
```

Variable symbols

- A symbol, not pre-defined, nor defined elsewhere as a label, then it is a *variable.*
- Each variable is assigned a unique memory address, starting at **16**.

symbol	value
i	16
sum	17

Translating *@variableSymbol* :

- First time see it, assign a unique memory address.
- Replace *variableSymbol* with this address.

Examples

Symbolic: @ i

Binary: 0000000000010000

Exercise: translate assembly code

- Translate the following assembly code to binary code:

```
@10
D=A
@R10
M=D
(END)
@END
0;JMP
```

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
null	0	0	0
M	0	0	1
D	0	1	0
MD	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
AMD	1	1	1

jump	j1	j2	j3
null	0	0	0
JGT	0	0	1
JEQ	0	1	0
JGE	0	1	1
JLT	1	0	0
JNE	1	0	1
JLE	1	1	0
JMP	1	1	1

Exercise: translate assembly code

- Translate the following assembly code to binary code:

@10	0000000000001010
D=A	1110110000010000
@R10	0000000000001010
M=D	1110001100001000
(END)	
@END	0000000000000100
0;JMP	1110101010000111

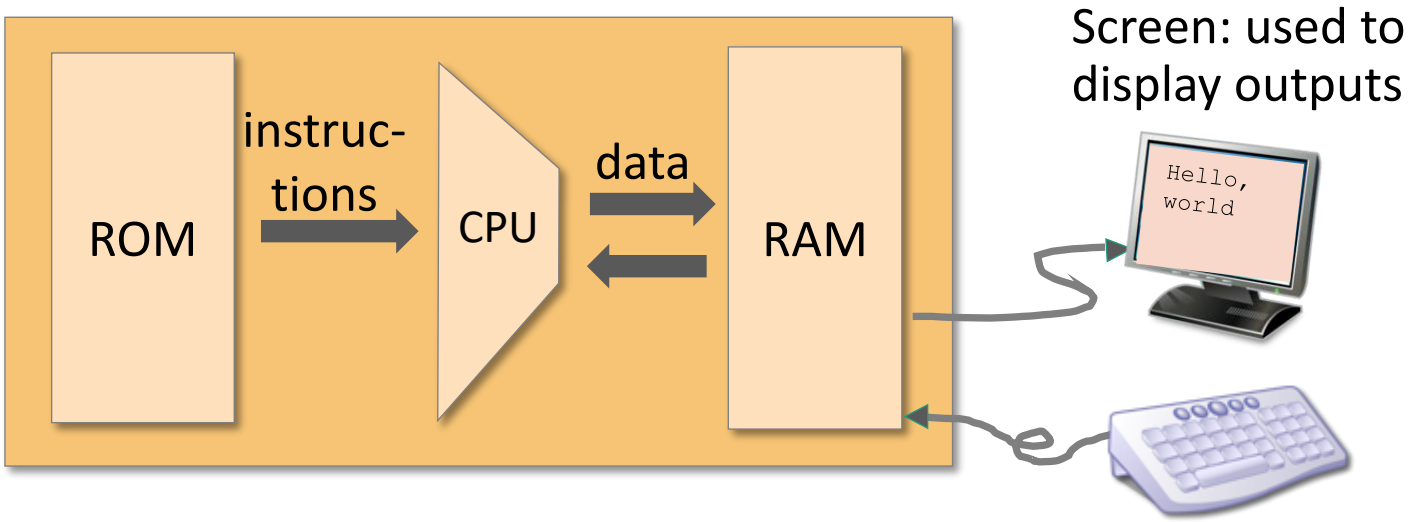
Thinking & self-study

- D register appears in 'dest' or 'comp', the binary codes are different.
- **Label declaration such as '(END)' does not account as one instruction!**
- Why 7 bits for 'comp', whereas only 3 bits for 'dest' and 'jump'?
- @value operation often requires **conversion from decimal number to binary number**. Practice it by yourself.
- Manual translation from assembly code to binary code is tedious. Practice it by yourself.

Outlines

- Introduction to assembler
- Translate Hack assembly program
  - Translate program without symbols
  - Translate program with symbols
- Hack input / output

Input / output



Screen: used to display outputs

Keyboard: used to enter inputs

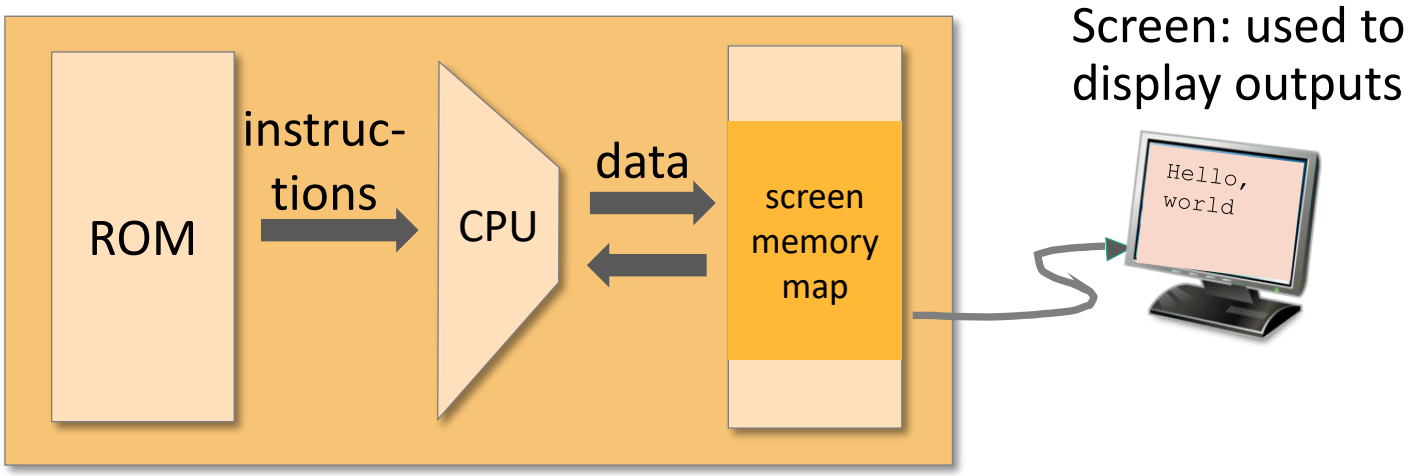
I/O handling (high-level):

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

I/O handling (low-level):

Bit manipulation.

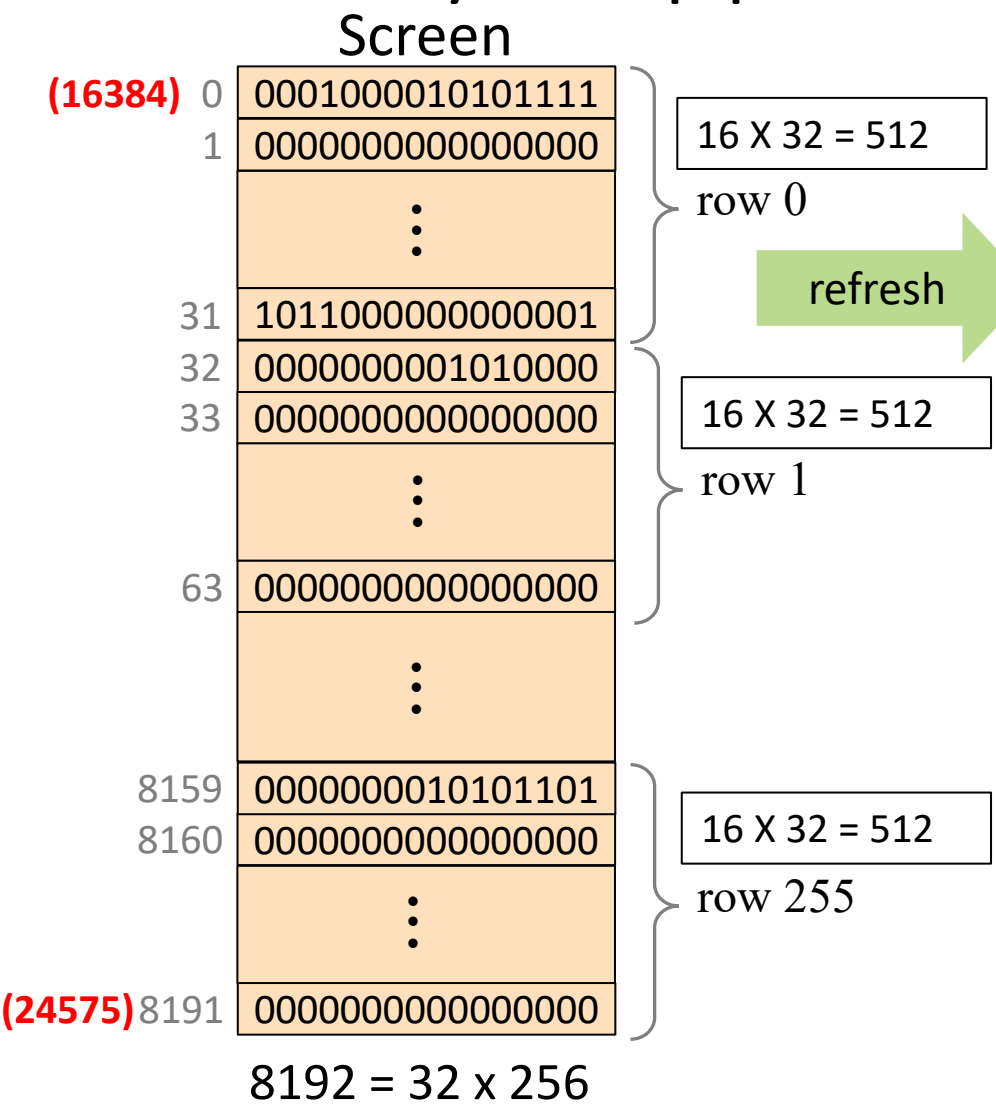
Memory mapped output



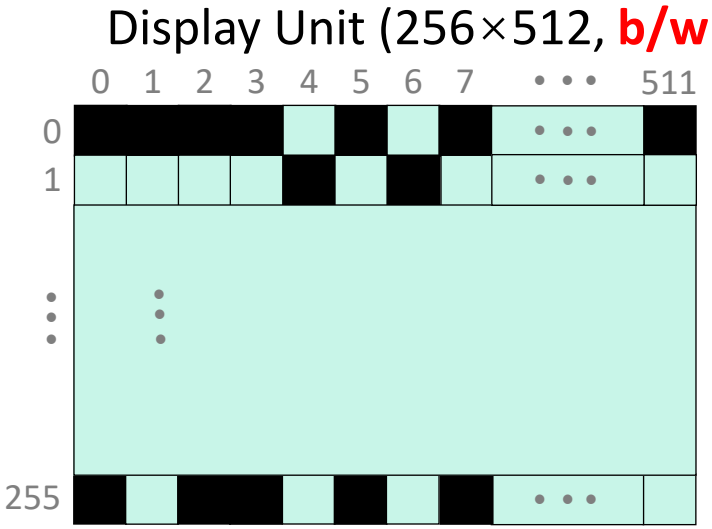
Screen: used to display outputs

- 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



refresh

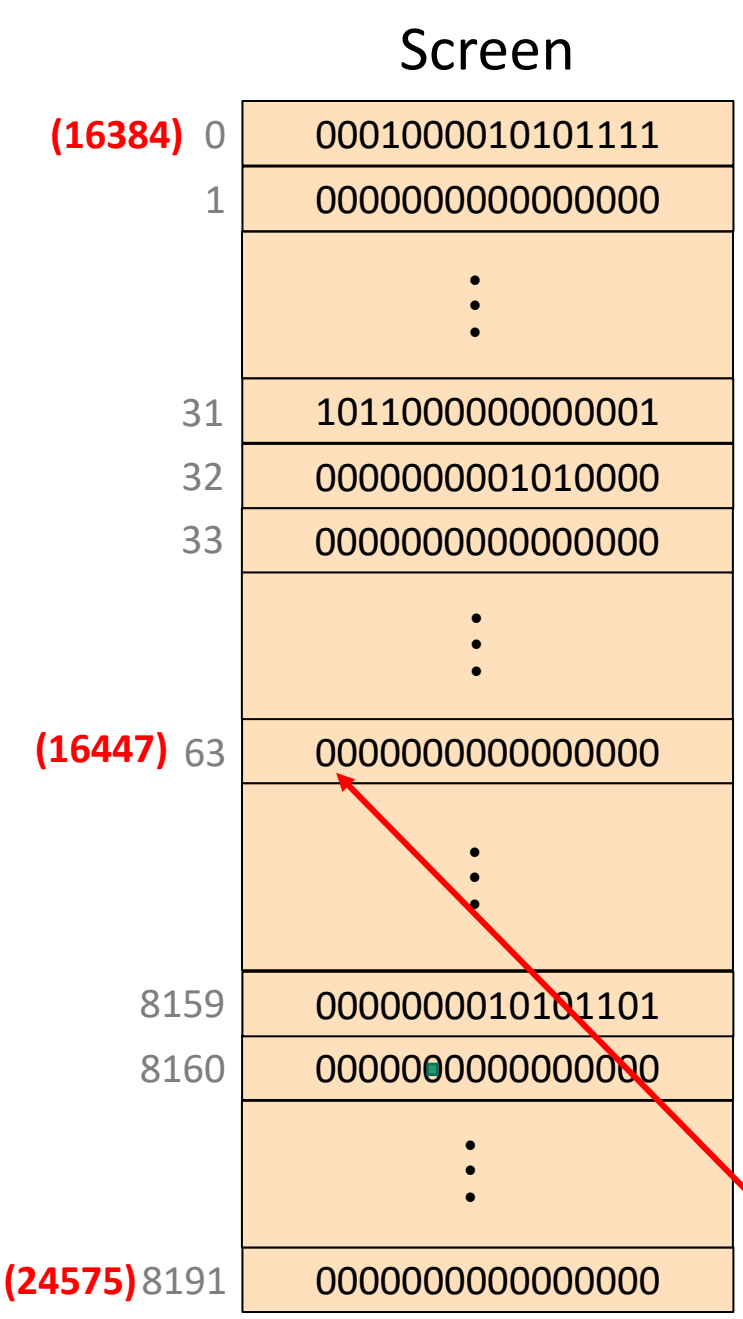


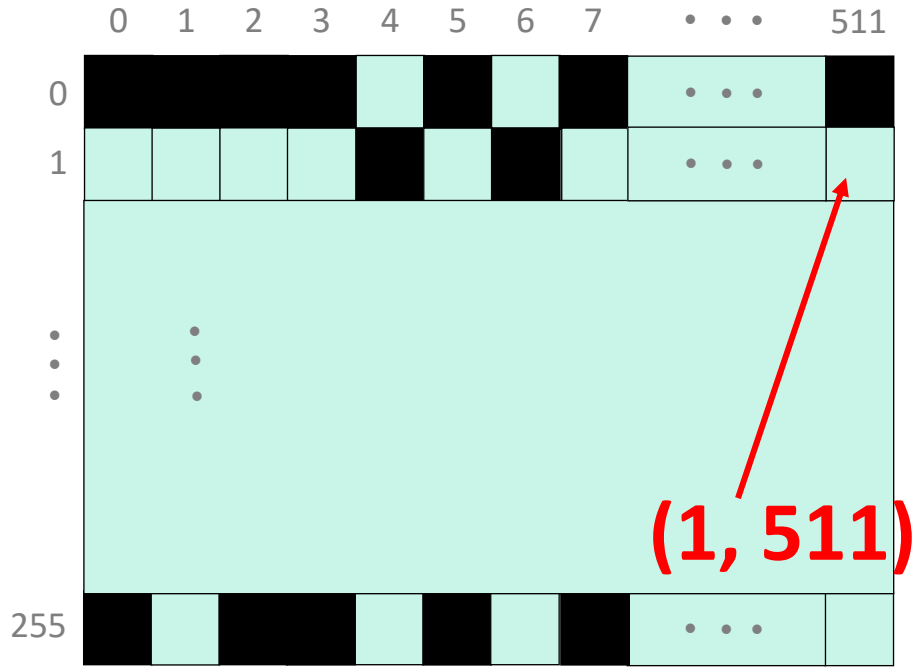
Integer division, round-down.

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

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

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





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

1 511

word=63 32 31

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

511

bit=15

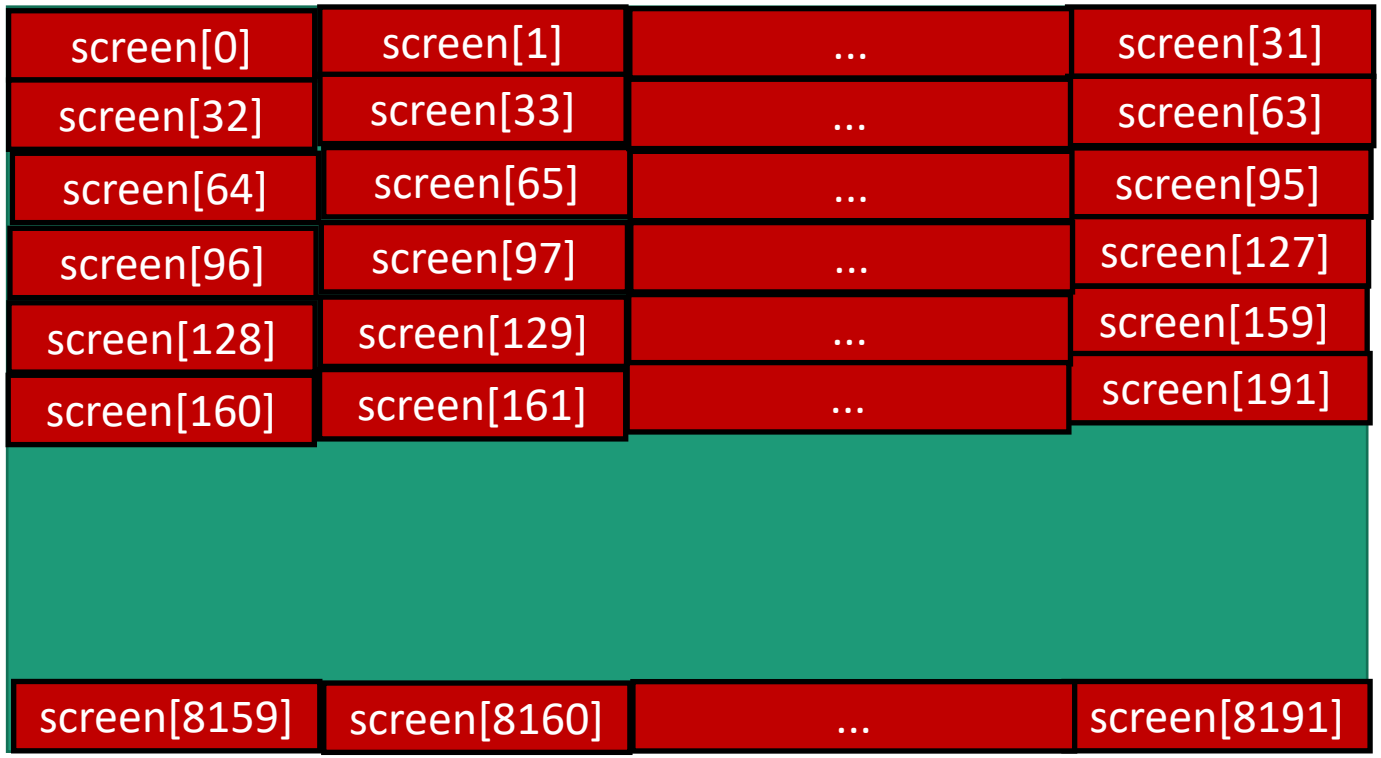
Hack Screen

512-bit wide

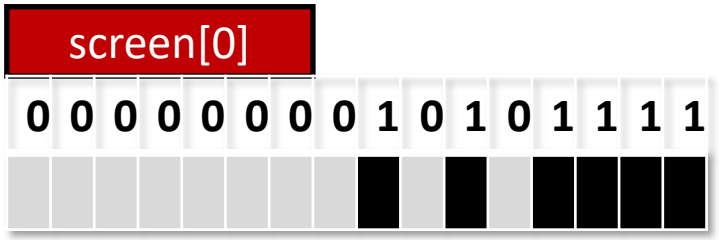
col→

256-bit high

row↓



screen[0] = 1111010100000000

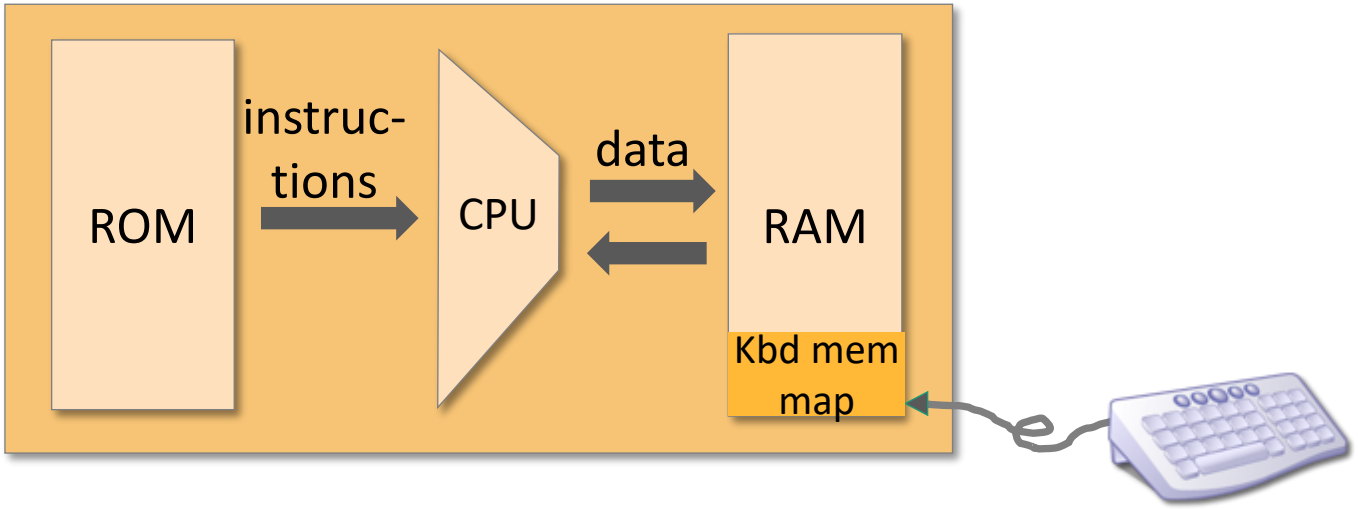


1 for black

0 for white

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

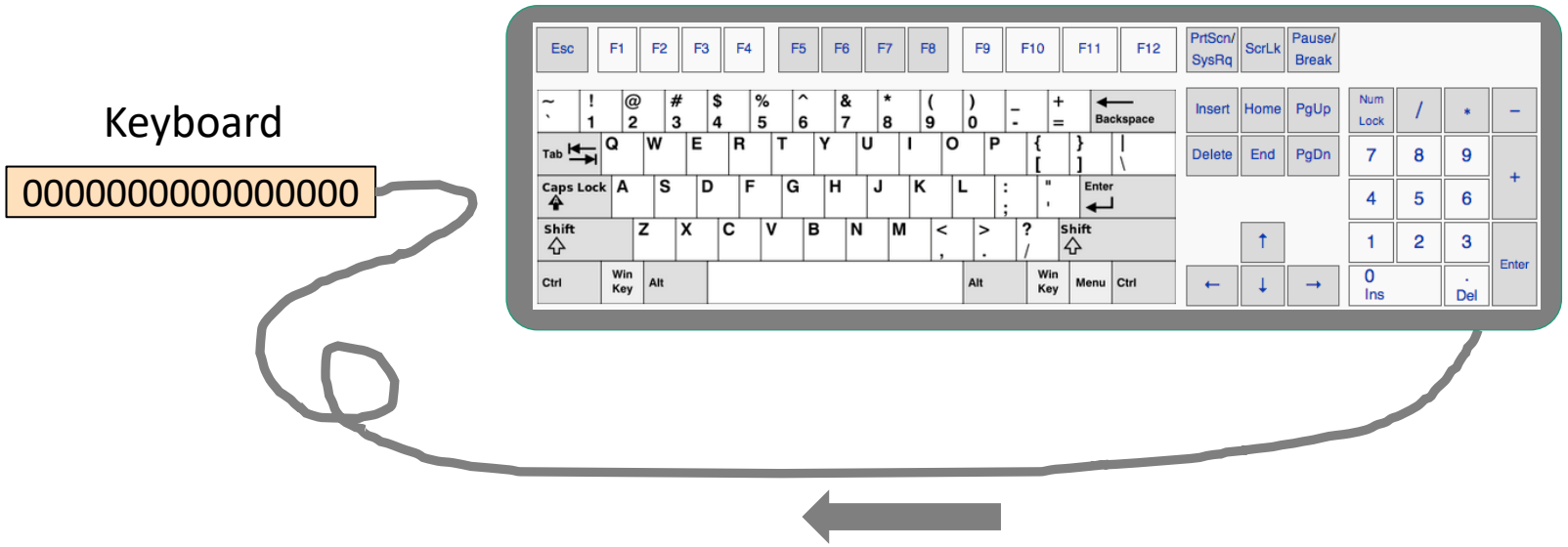
Input



Keyboard: used
to enter inputs

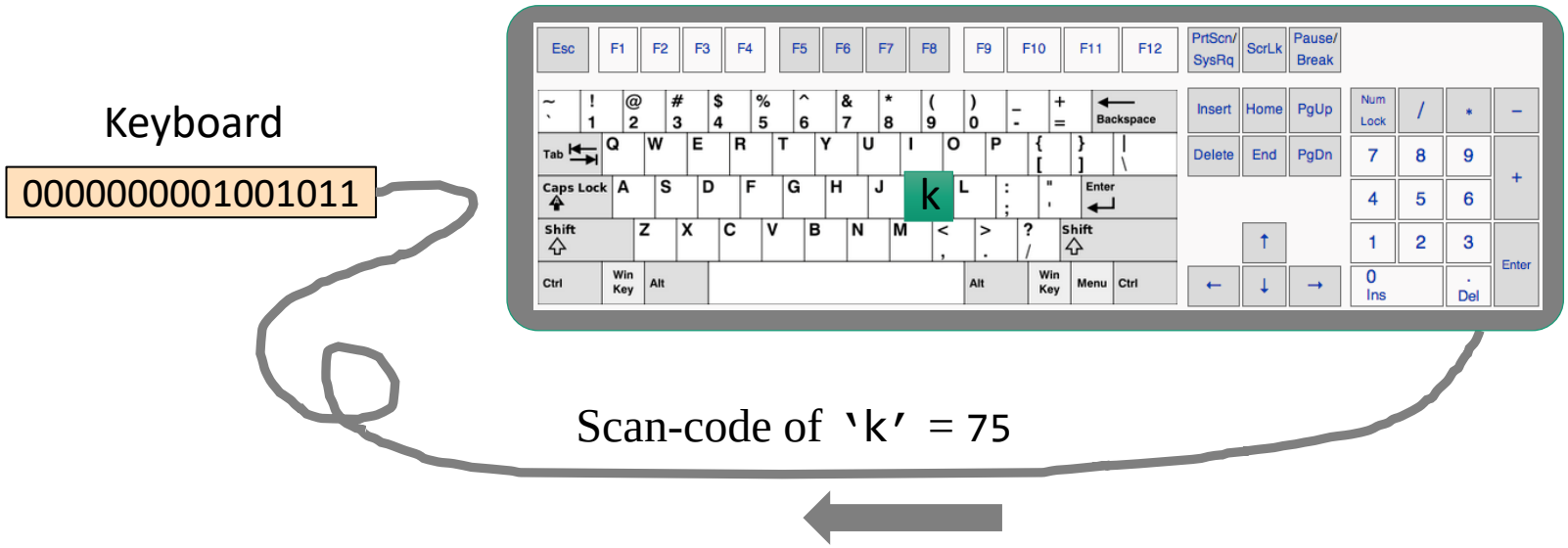
The physical keyboard is associated with a *keyboard memory map*.

Memory mapped input



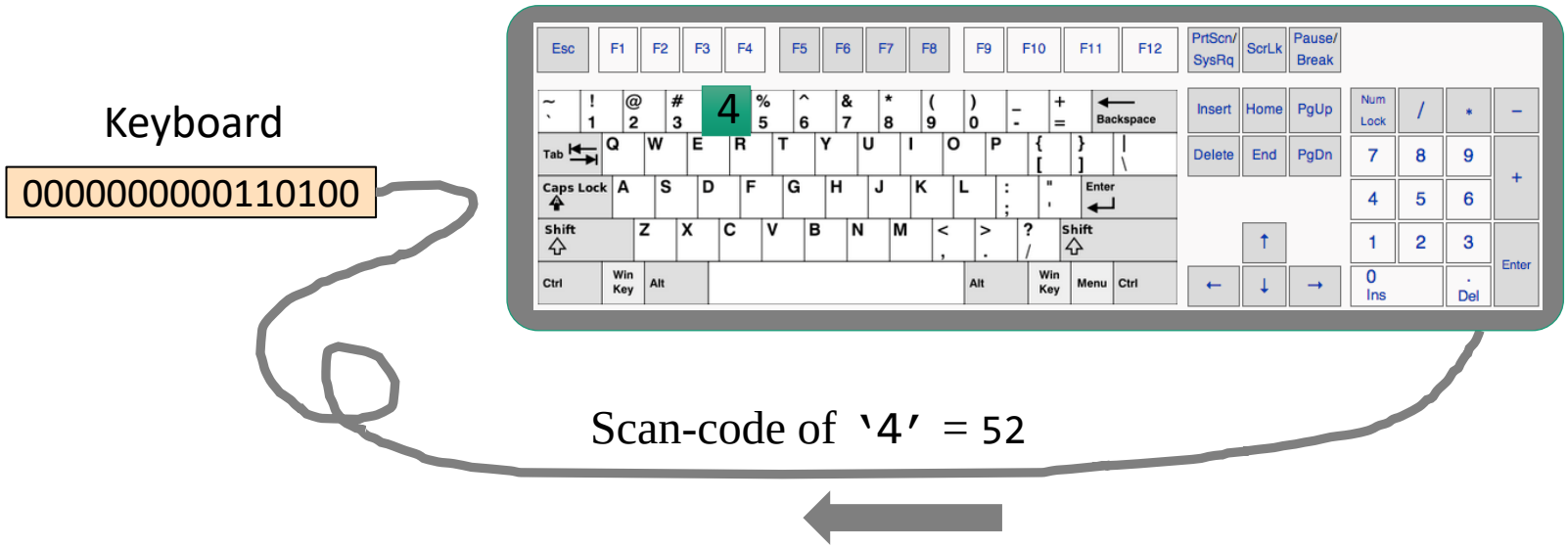
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



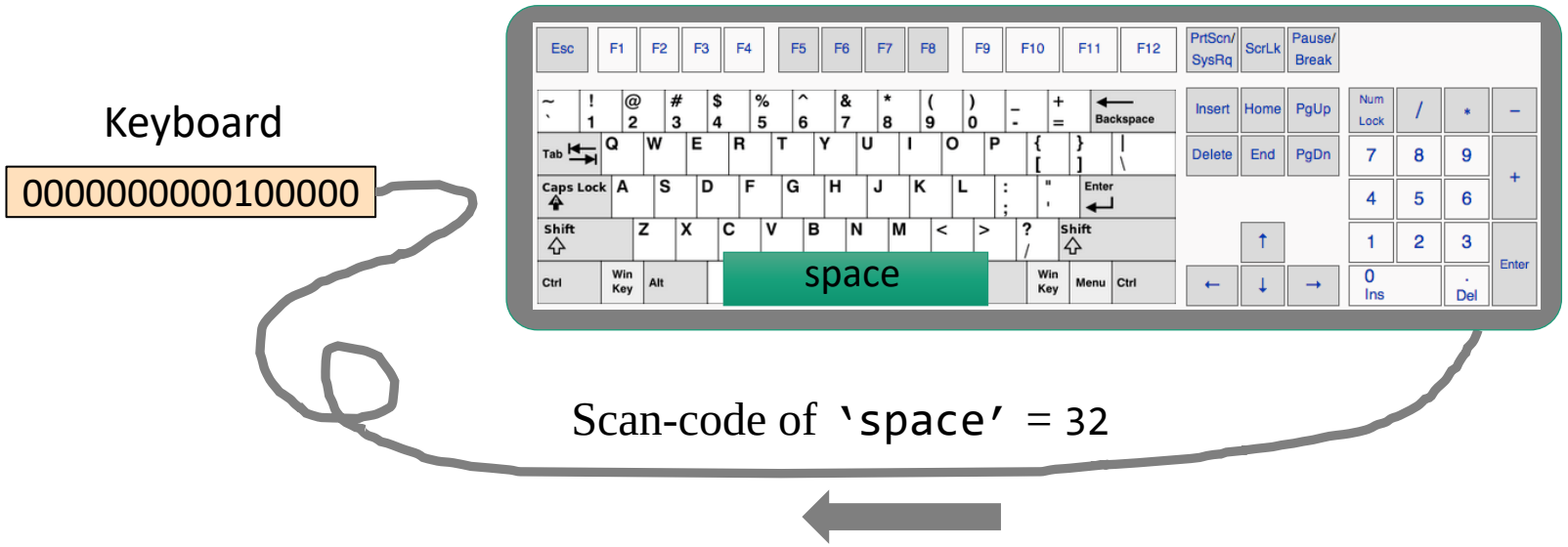
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



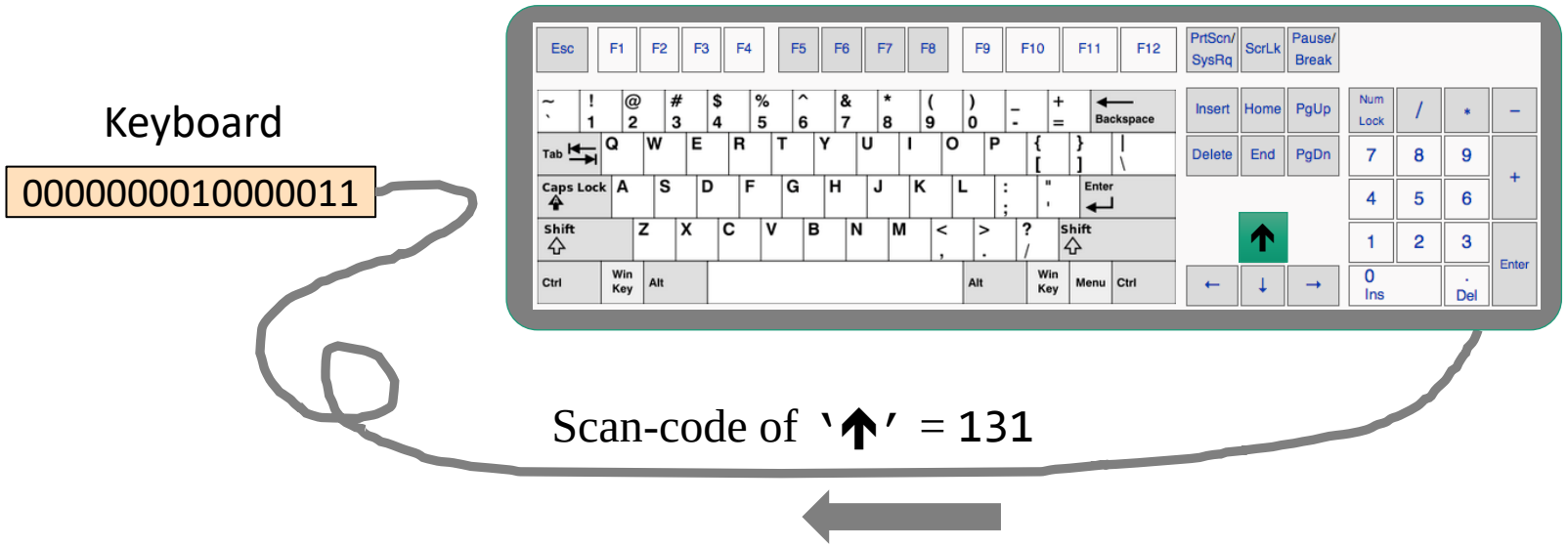
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

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

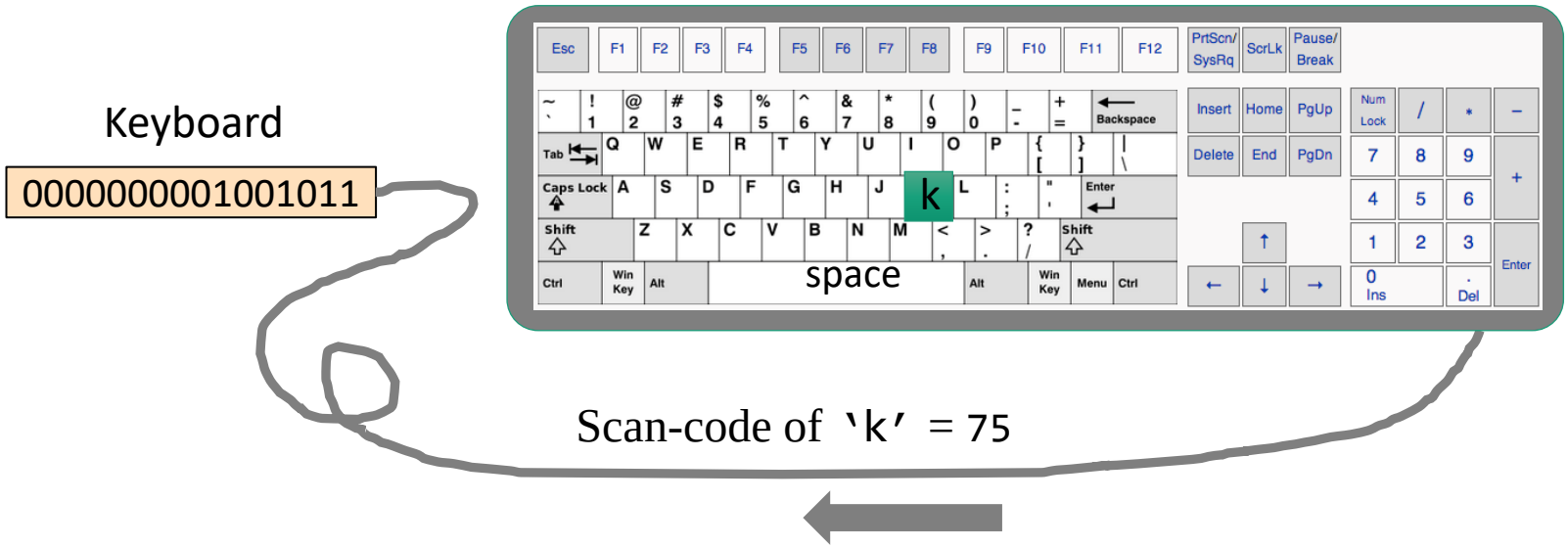
[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

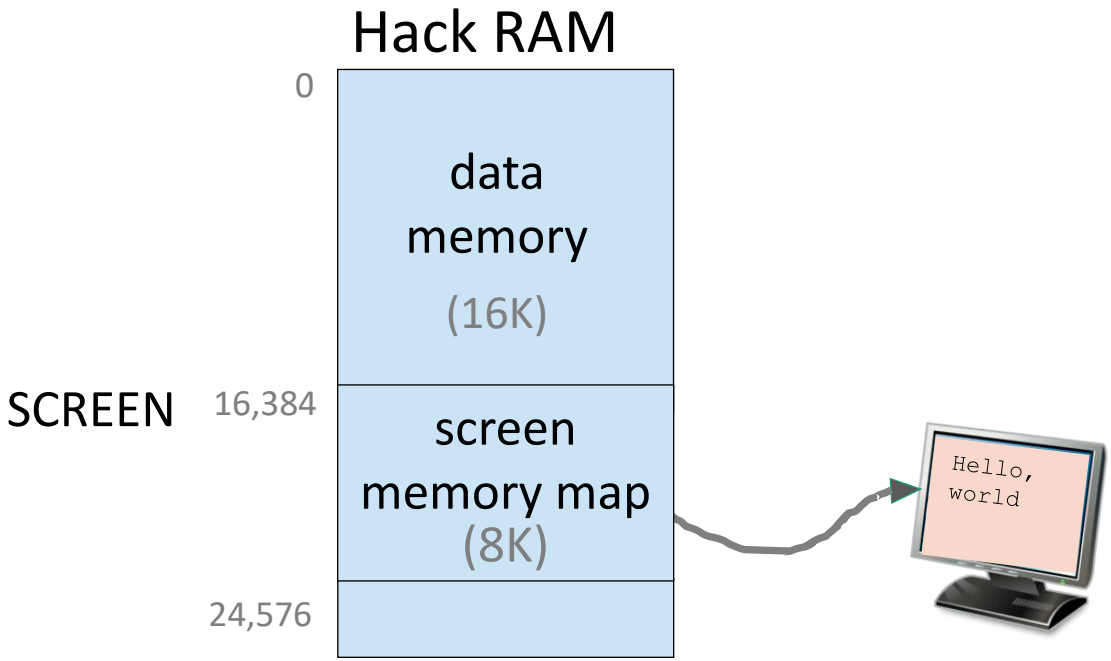
key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
…	…
f12	152

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]**.

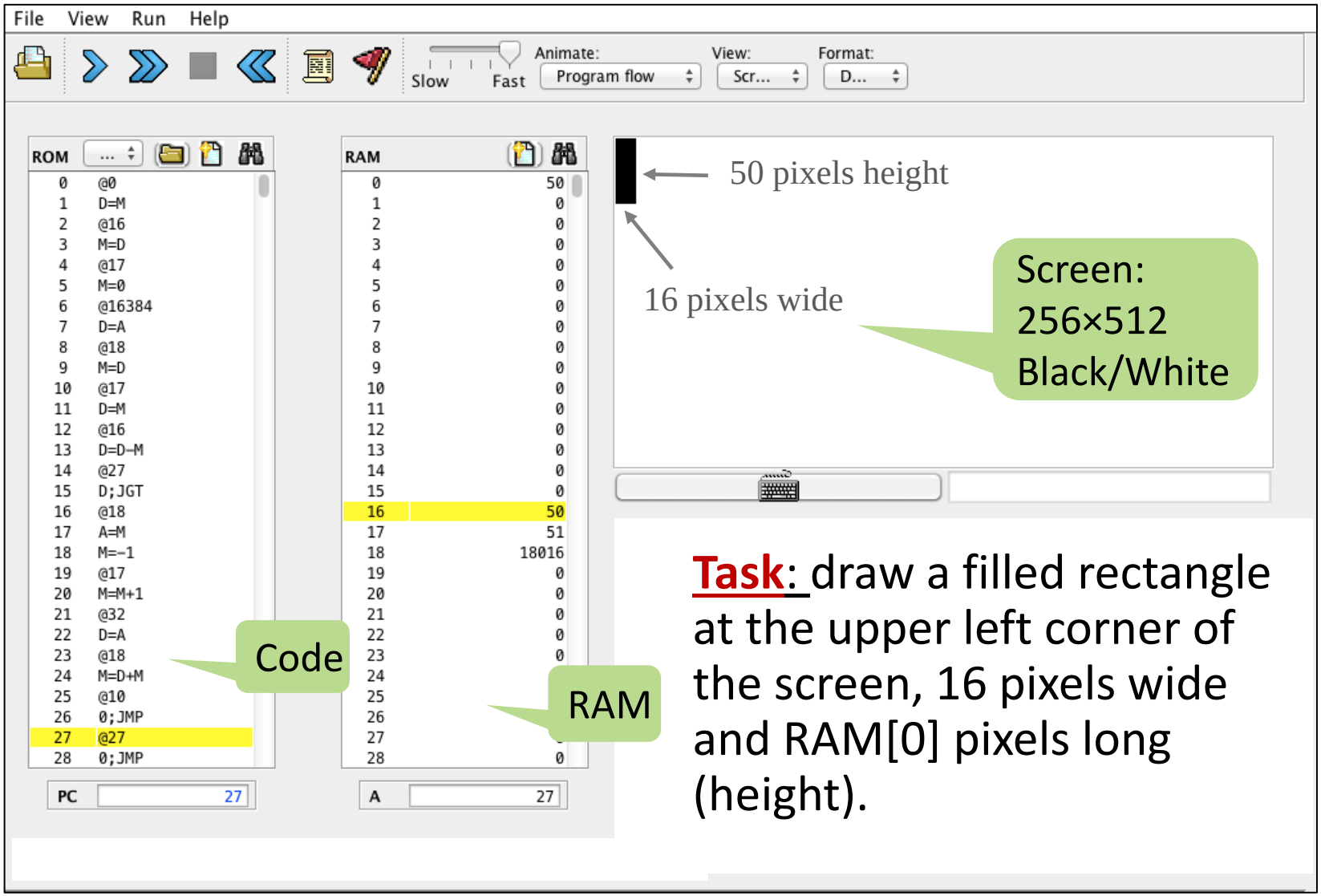
Output



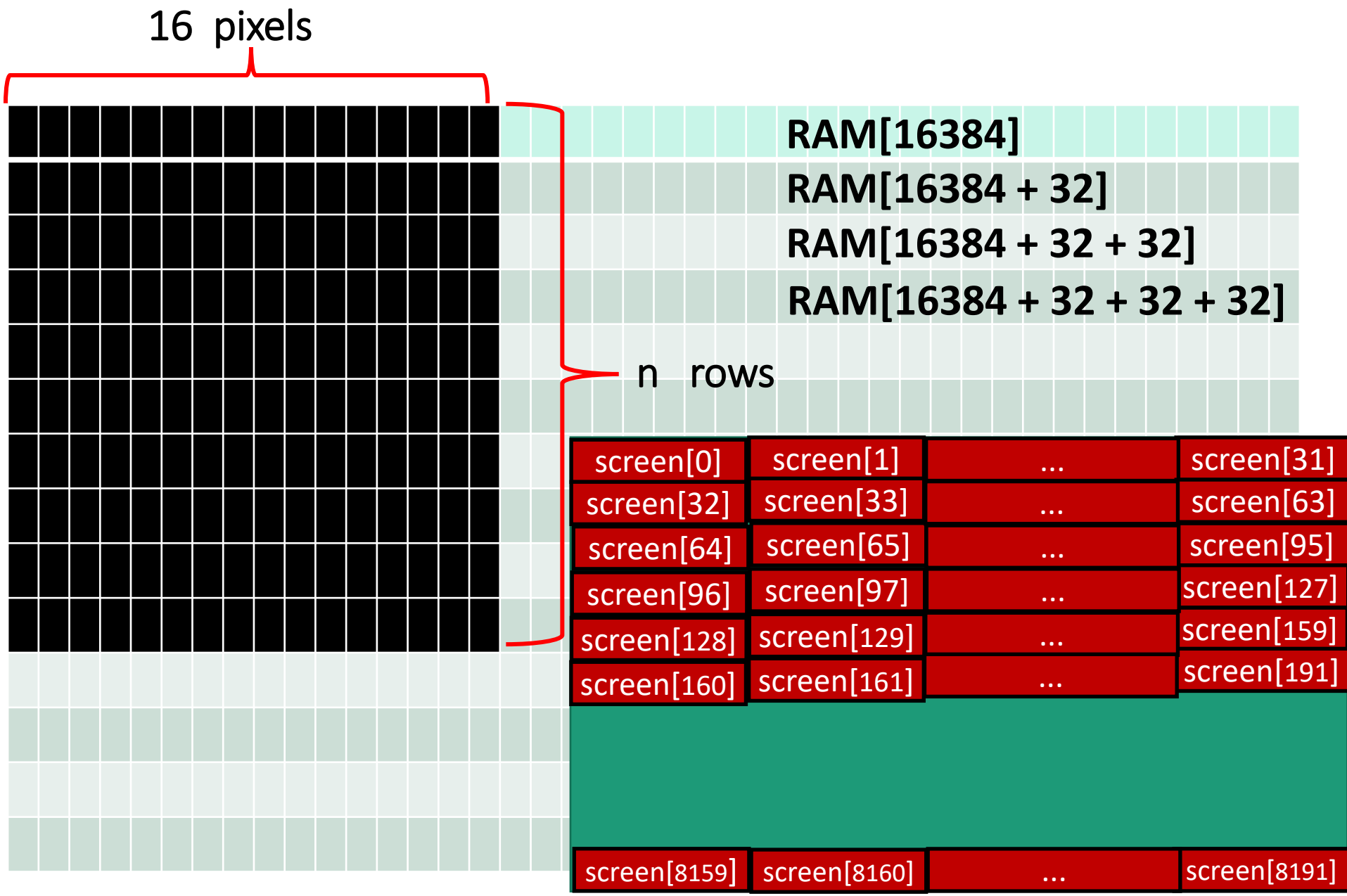
Hack language convention:

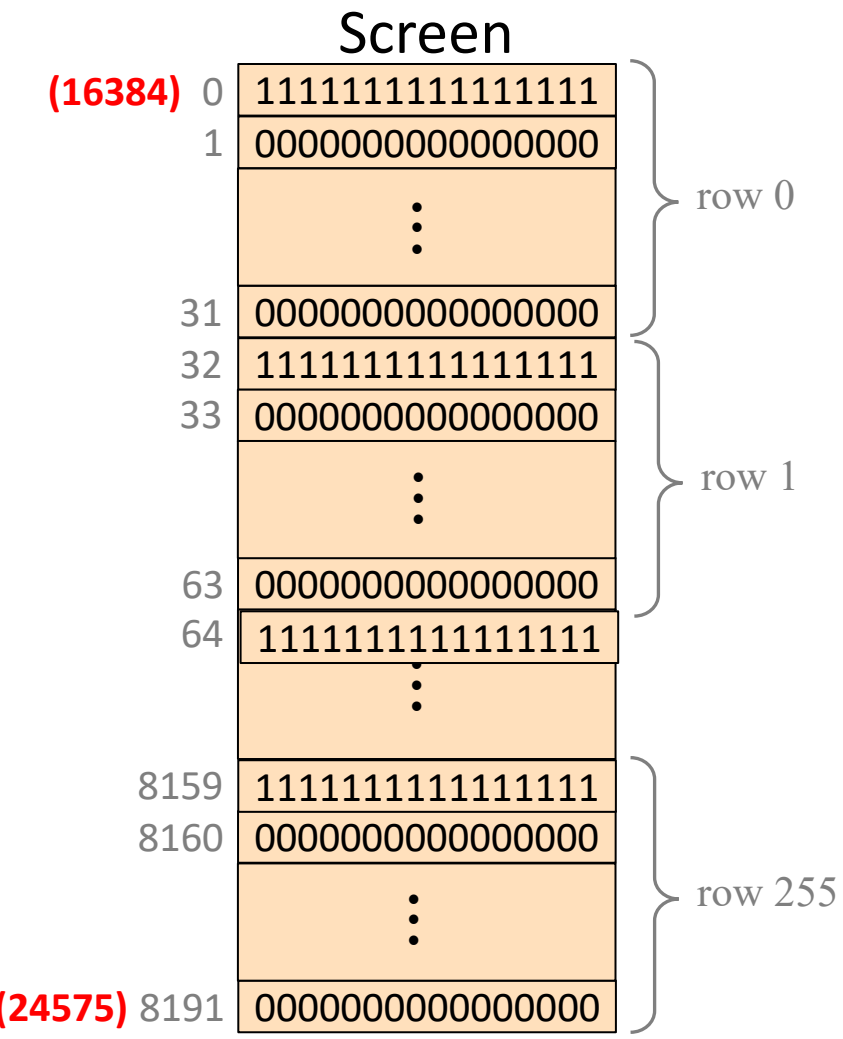
- SCREEN: base address of the screen memory map

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 (height).


16 pixels
n rows
RAM[16384]
RAM[16384 + 32]
RAM[16384 + 32 + 32]
RAM[16384 + 32 + 32 + 32]
screen[0]
screen[1]
...
screen[31]
screen[32]
screen[33]
...
screen[63]
screen[64]
screen[65]
...
screen[95]
screen[96]
screen[97]
...
screen[127]
screen[128]
screen[129]
...
screen[159]
screen[160]
screen[161]
...
screen[191]
screen[8159]
screen[8160]
...
screen[8191]



Two's Complement

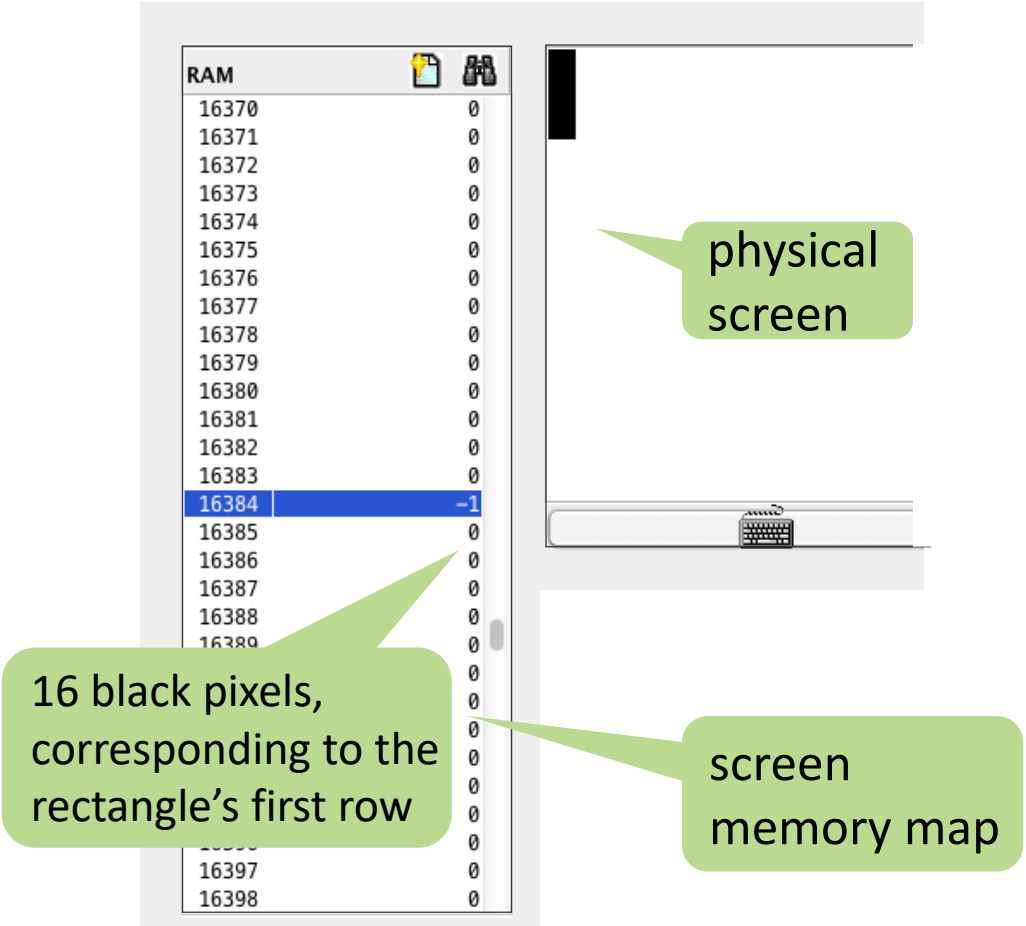
1 = 0000000000000001

-1 = 1111111111111111

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
```

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
```

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
```

Handling the screen (example)

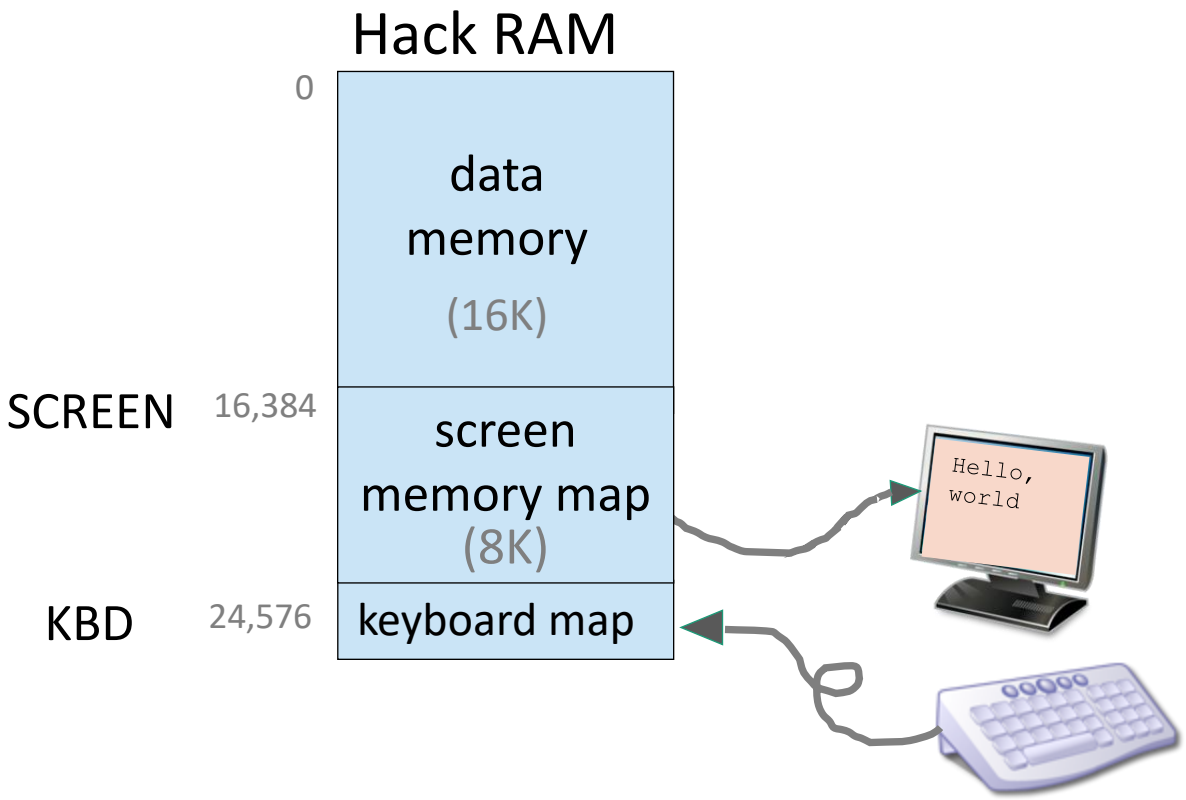
Pseudo code

```
// 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
  i = i + 1
  addr = addr + 32
goto LOOP

END:
  goto END
```

Assembly code

```
(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
```

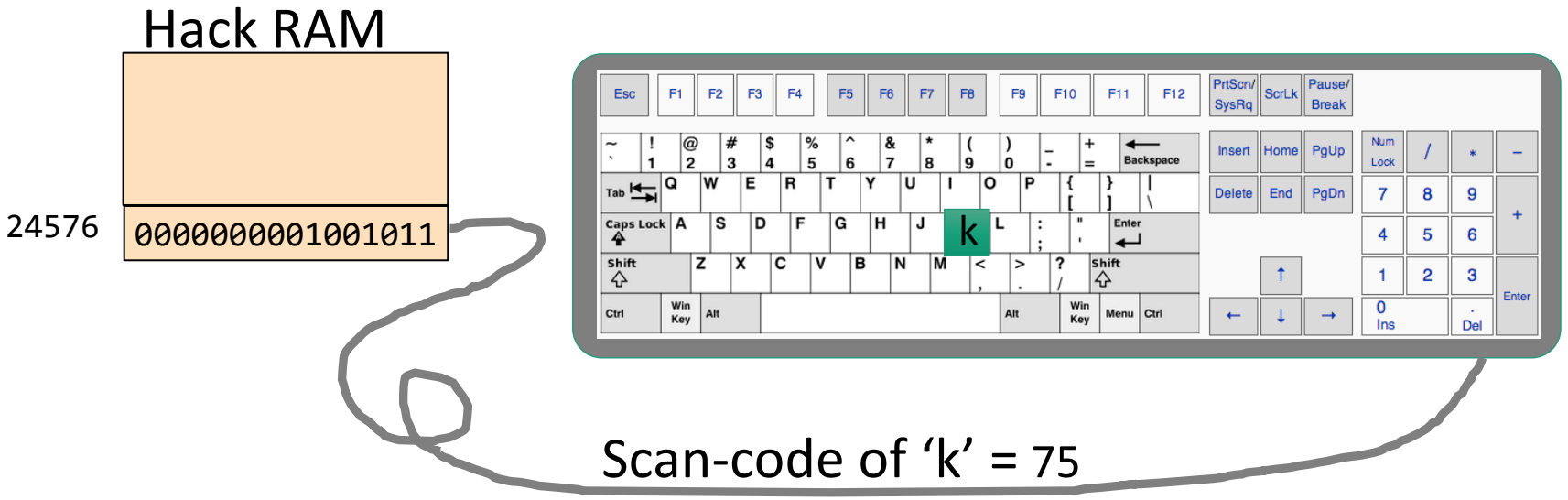
Input



Hack language convention:

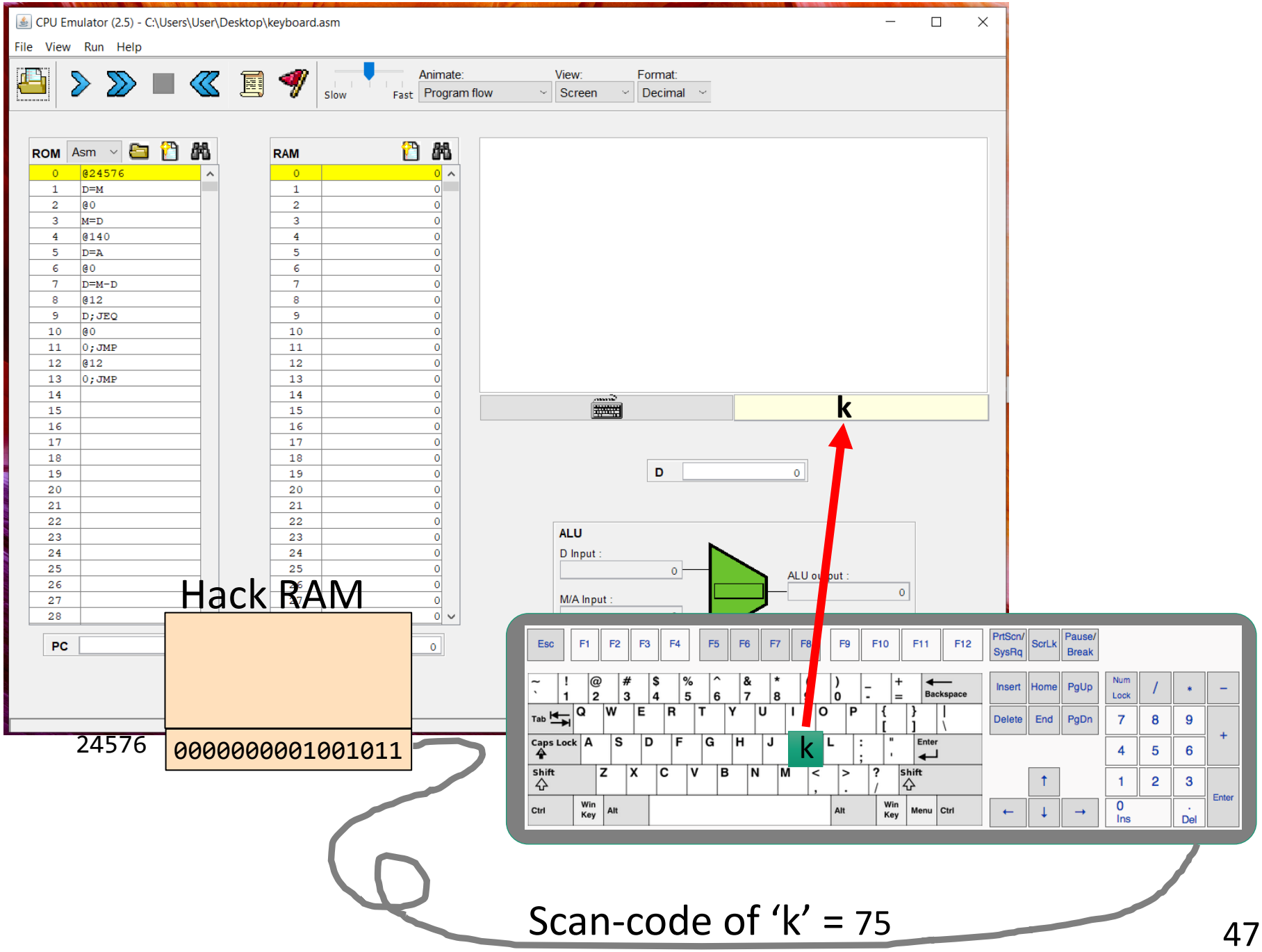
- SCREEN: base address of the screen memory map
- KBD: address of the keyboard memory map

Handle the keyboard



To check which key is currently pressed:

- 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.


CPU Emulator (2.5) - C:\Users\User\Desktop\keyboard.asm
File View Run Help
Animate: Program flow
View: Screen
Format: Decimal
Slow Fast
ROM Asm
RAM
k
D
ALU
D Input :
M/A Input :
ALU output :
PC
Hack RAM
24576
0000000001001011
Scan-code of ‘k’ = 75

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
```

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.