University of Nottingham
UK | CHINA | MALAYSIA

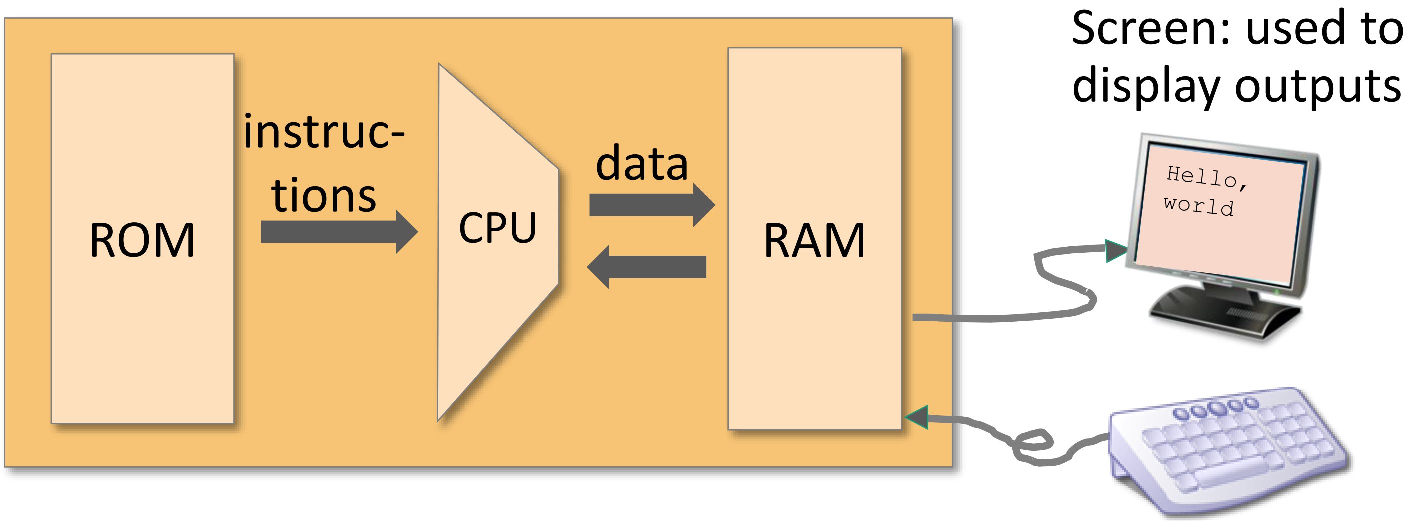
Input-Output
and
Assembler

Dr. Wooi Ping Cheah

Outlines

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

Input / output



Screen: used to display outputs

Keyboard: used to enter inputs

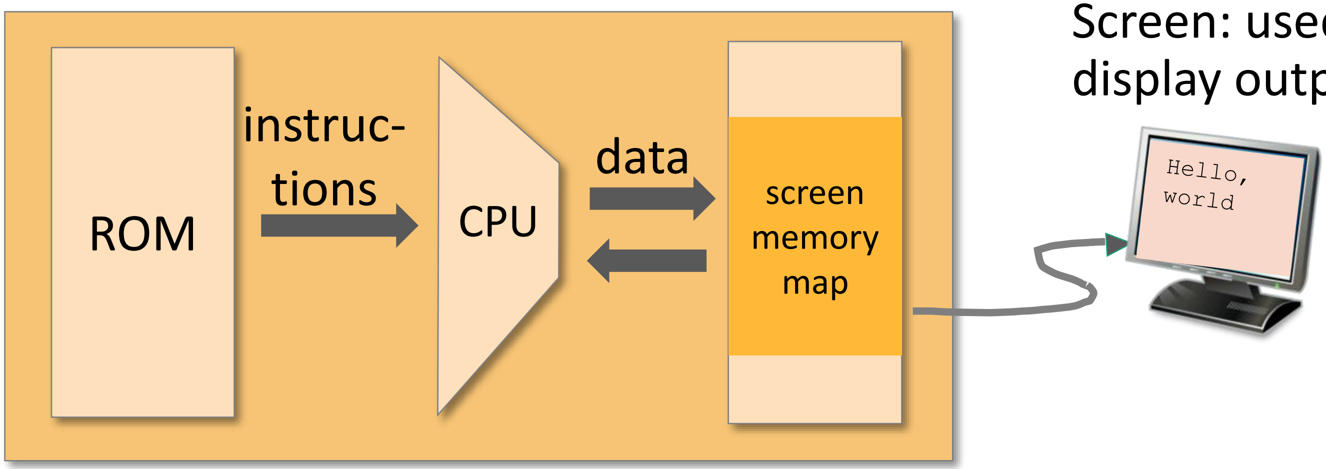
<u>I/O handling</u> (high-level):

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

<u>I/O handling</u> (low-level):

Bit manipulation.

Memory mapped output

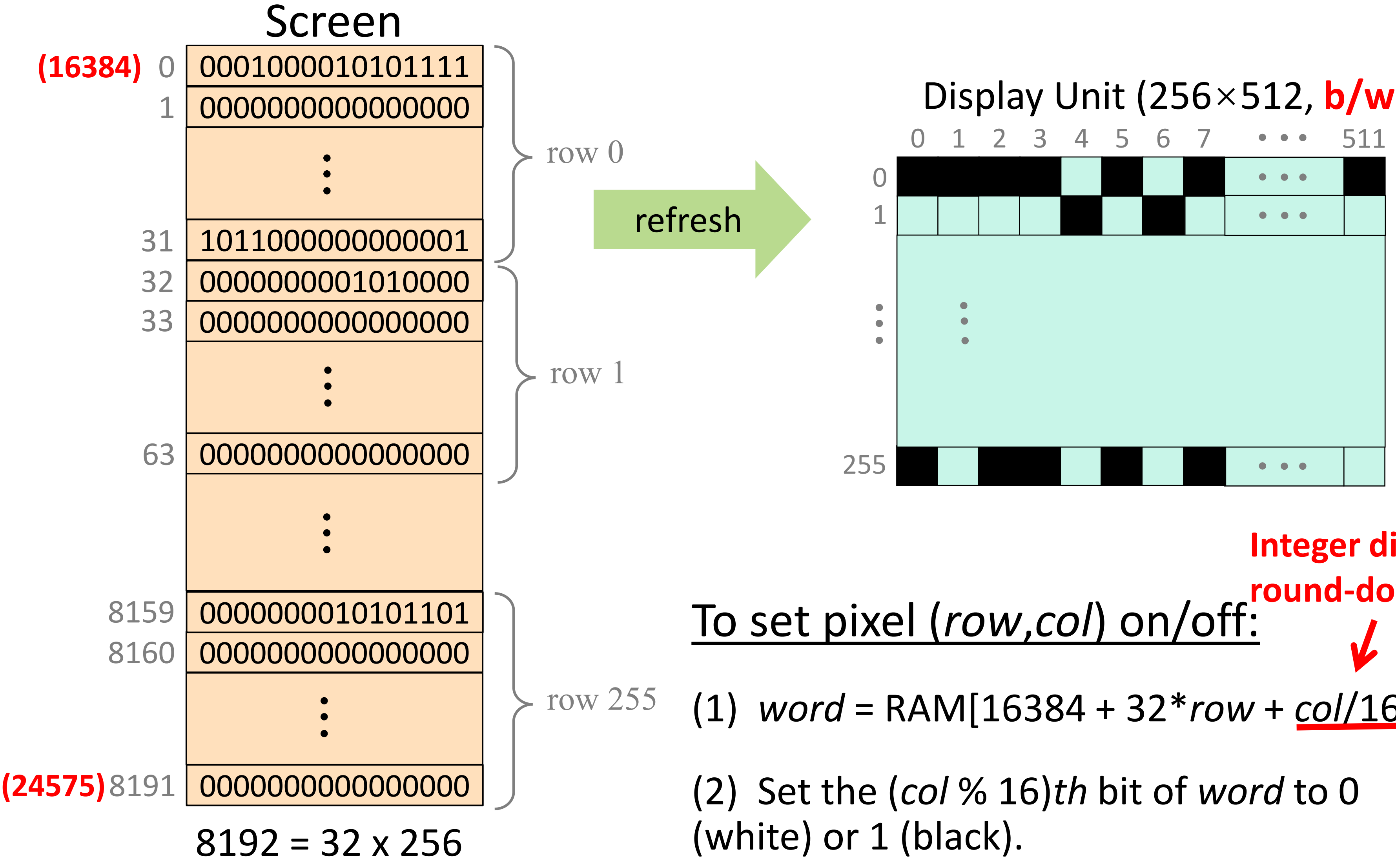


Screen: used to display outputs

- 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

16 X 32 = 512



Screen

(16384)	0	0001000010101111
	1	0000000000000000
		⋮
	31	1011000000000001
	32	0000000001010000
	33	0000000000000000
		⋮
	63	0000000000000000
		⋮
	8159	0000000010101101
	8160	0000000000000000
		⋮
(24575)	8191	0000000000000000

8192 = 32 x 256

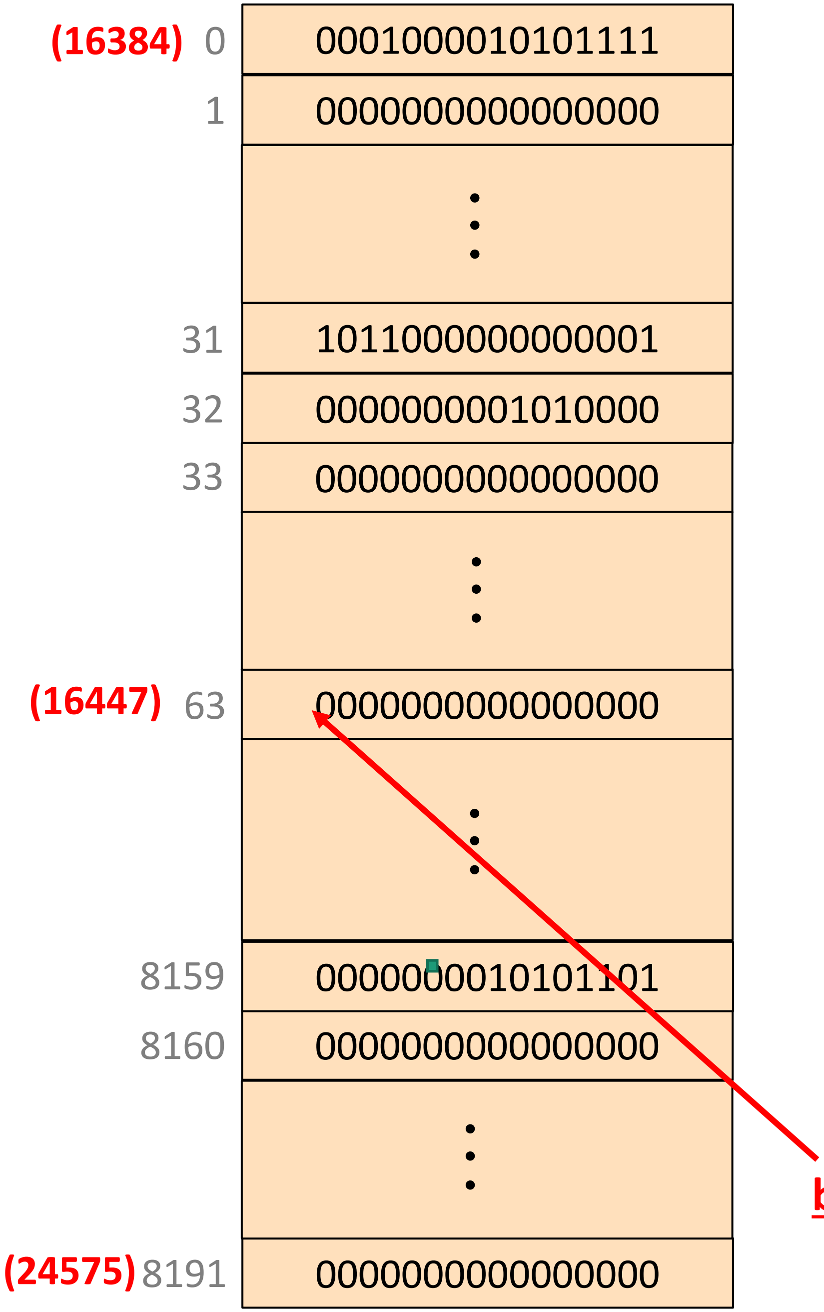
Integer division, round-down.

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

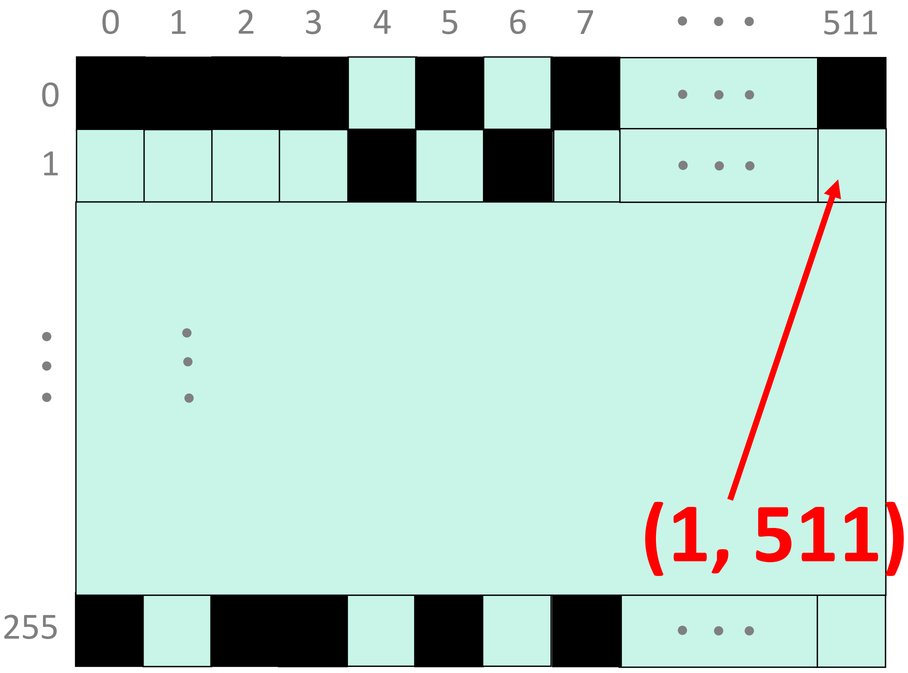
(1) *word* = RAM[16384 + 32**row* + *col*/16]

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

Screen



Display Unit (256×512, **b/w**)



(1) $word = \text{RAM}[16384 + \underset{32}{32*\overset{1}{row}} + \underset{31}{\overset{511}{col}/16}]$

word=63

(2) Set the ($\overset{511}{col}$ % 16)*th* bit of *word* to 0 (white) or 1 (black).

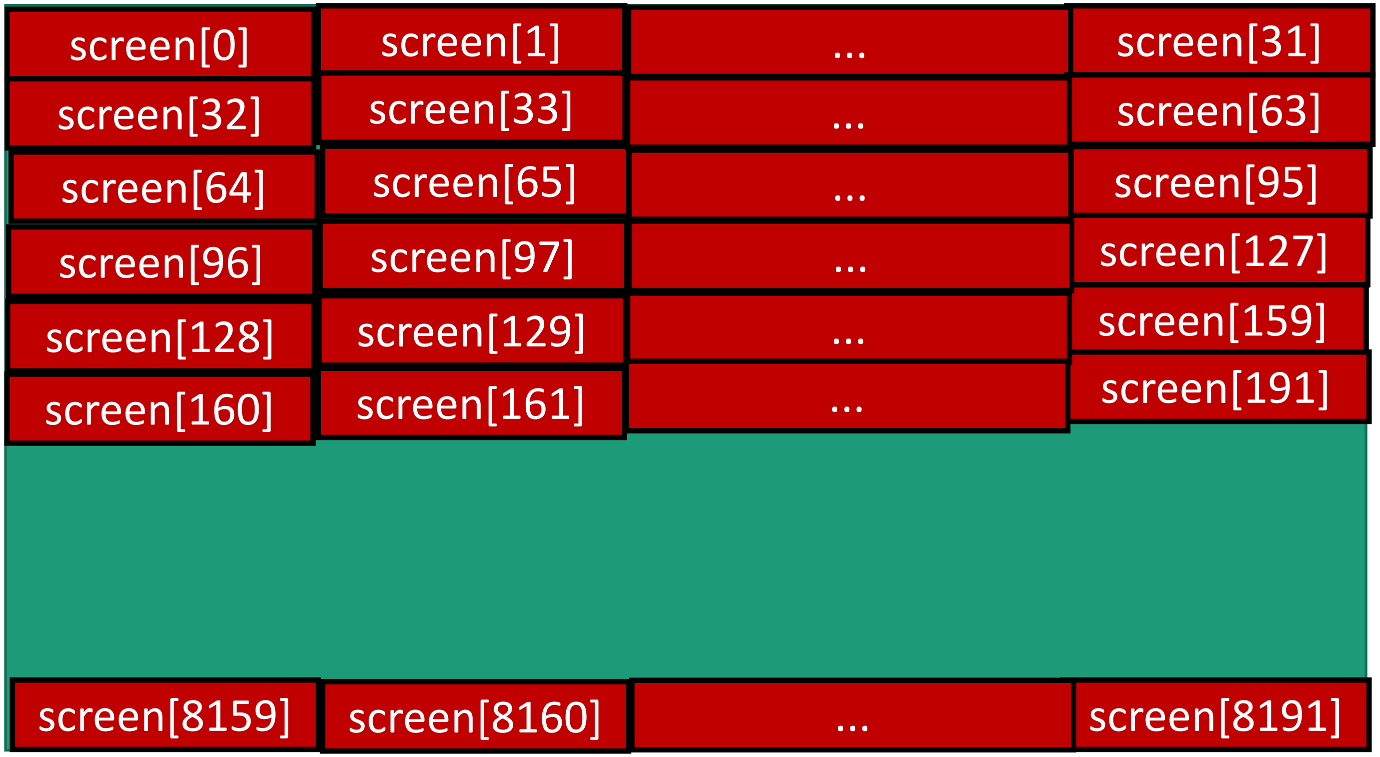
Hack Screen

512-bit wide

col→

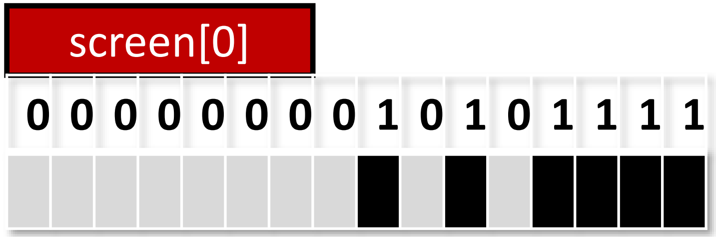
256-bit high

row→



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]

screen[0] = 1111010100000000



screen[0]

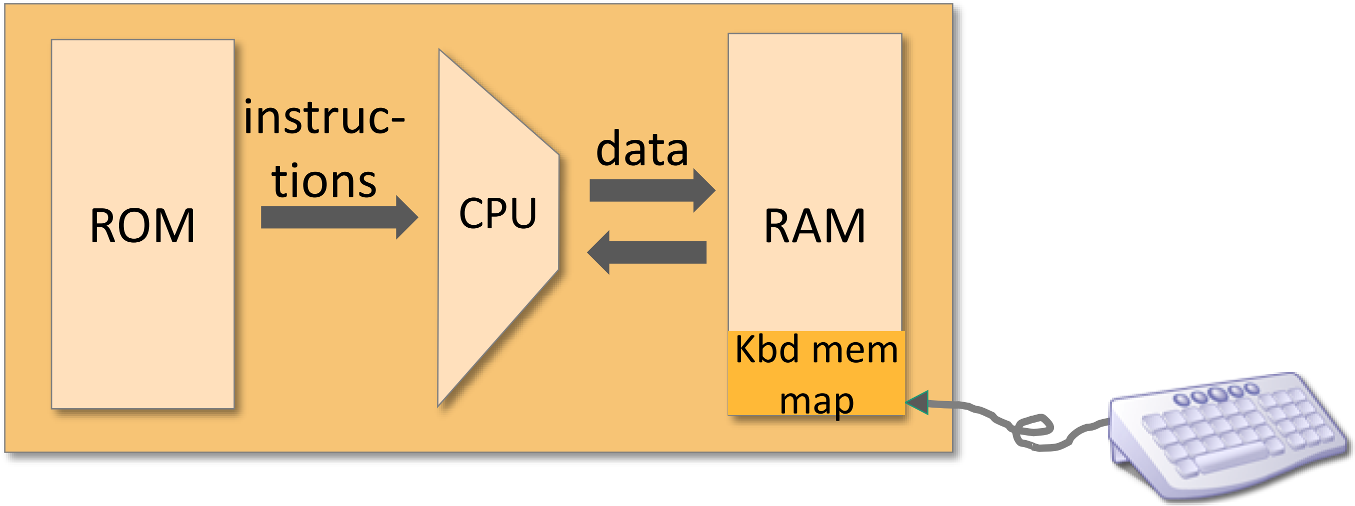
0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	1
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬛	⬜	⬛	⬜	⬛	⬛	⬛	⬛

1 for black

0 for white

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

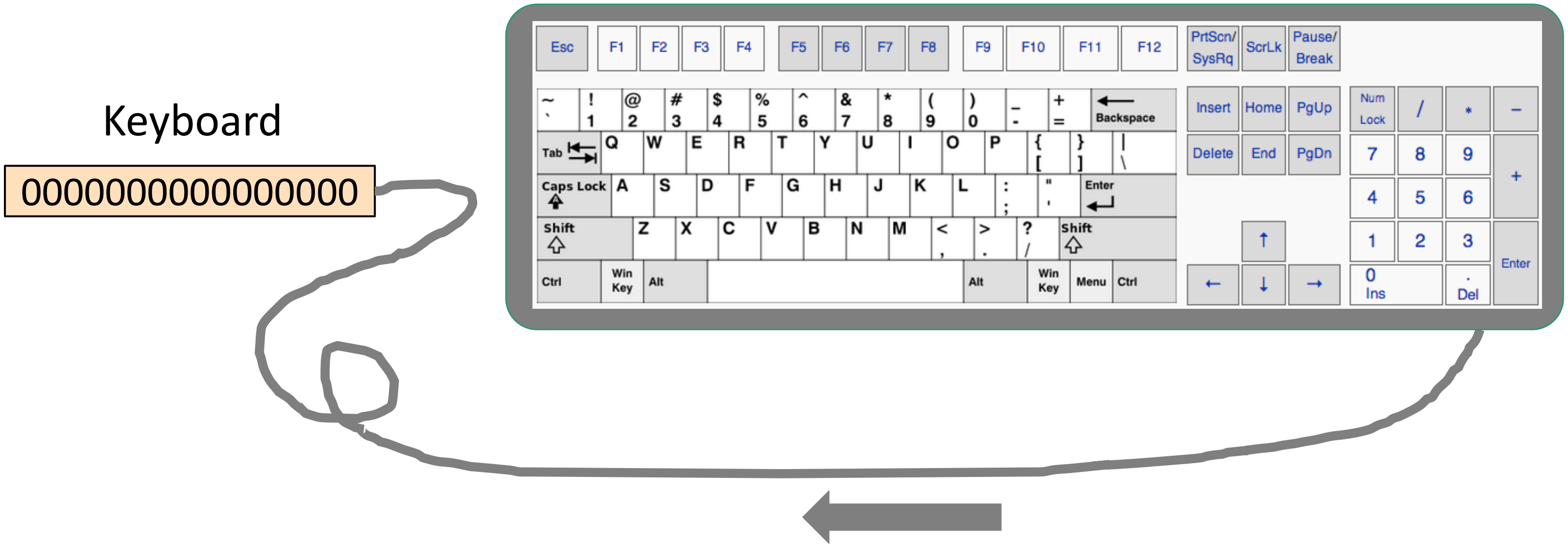
Input



Keyboard: used to enter inputs

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

Memory mapped input

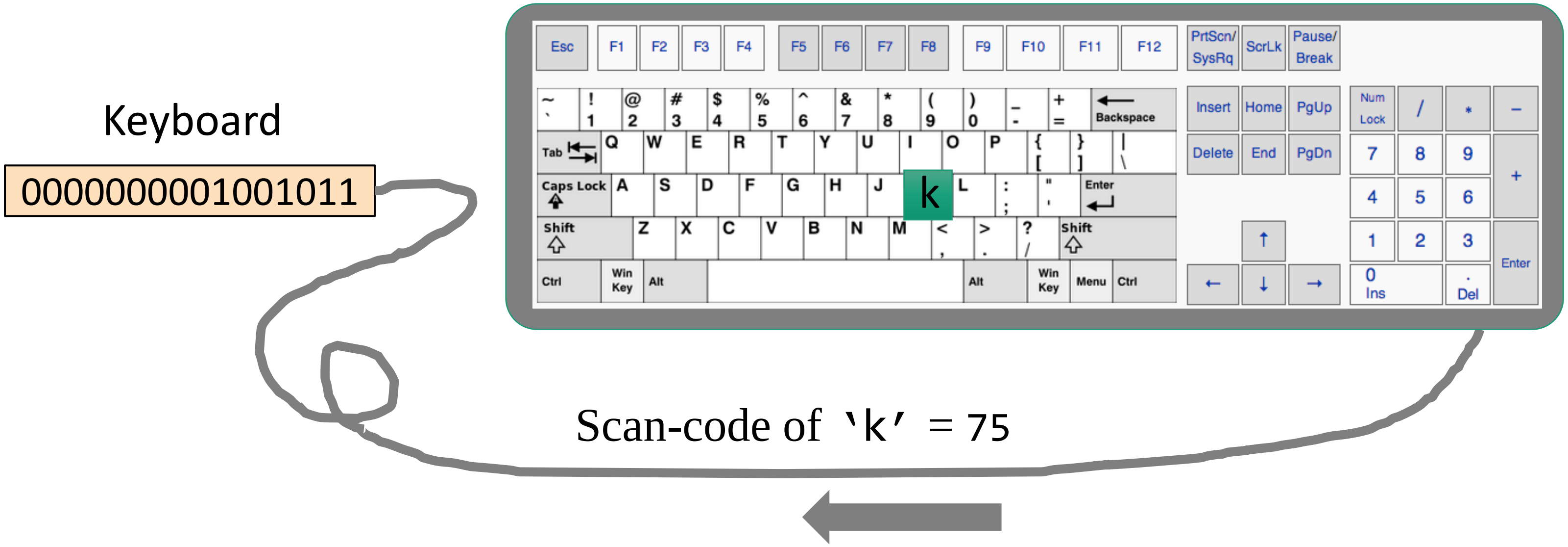


Keyboard

0000000000000000

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

Memory mapped input



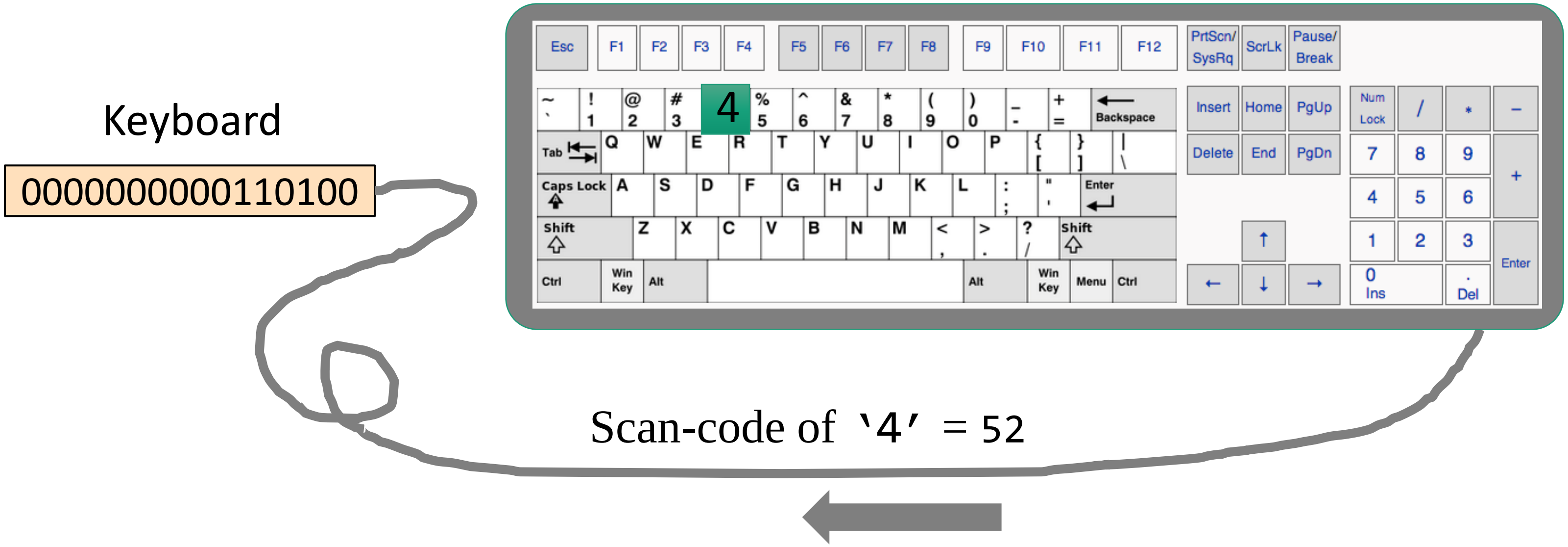
Keyboard

0000000001001011

Scan-code of `'k'` = 75

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

Memory mapped input



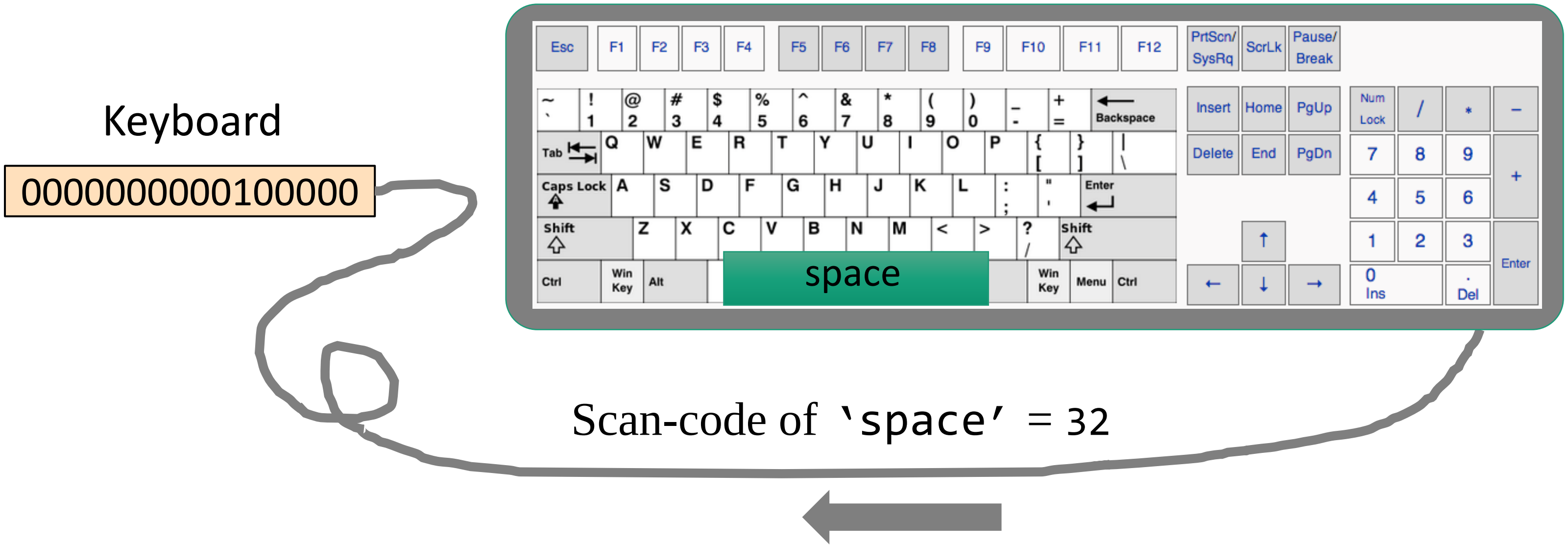
Keyboard

0000000000110100

Scan-code of ‘4’ = 52

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

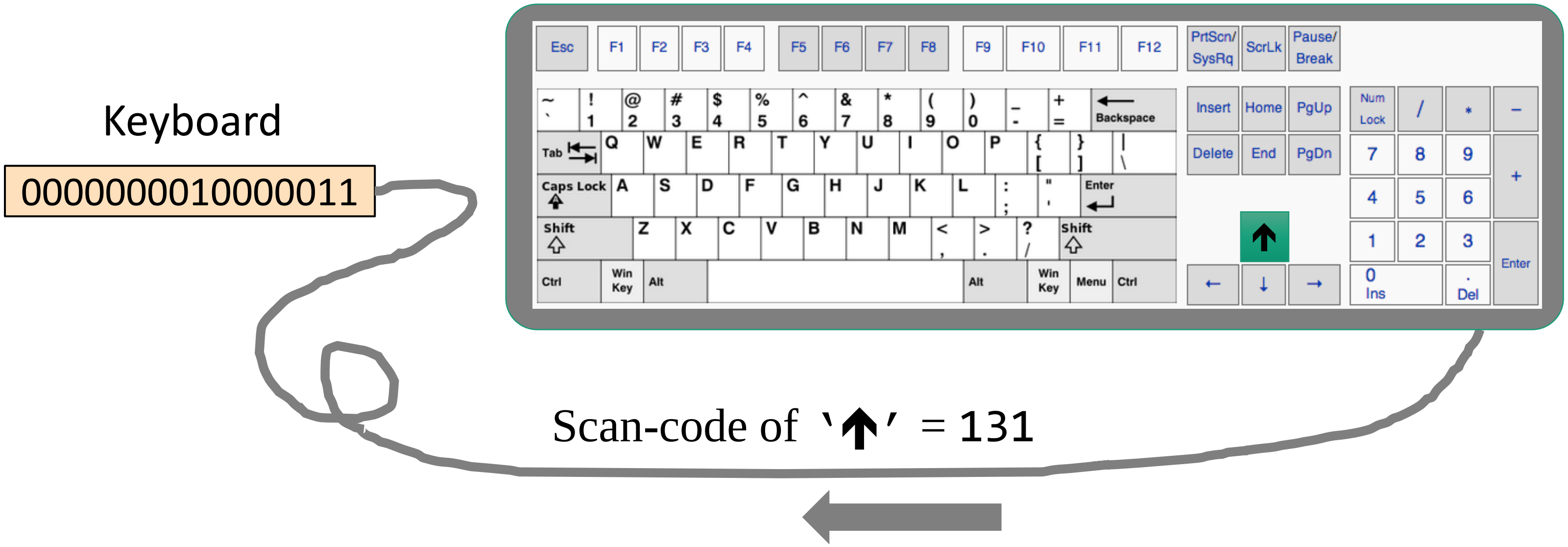
Memory mapped input



Scan-code of `'space'` = 32

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

Memory mapped input



Keyboard

0000000010000011

Scan-code of '↑' = 131

- 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

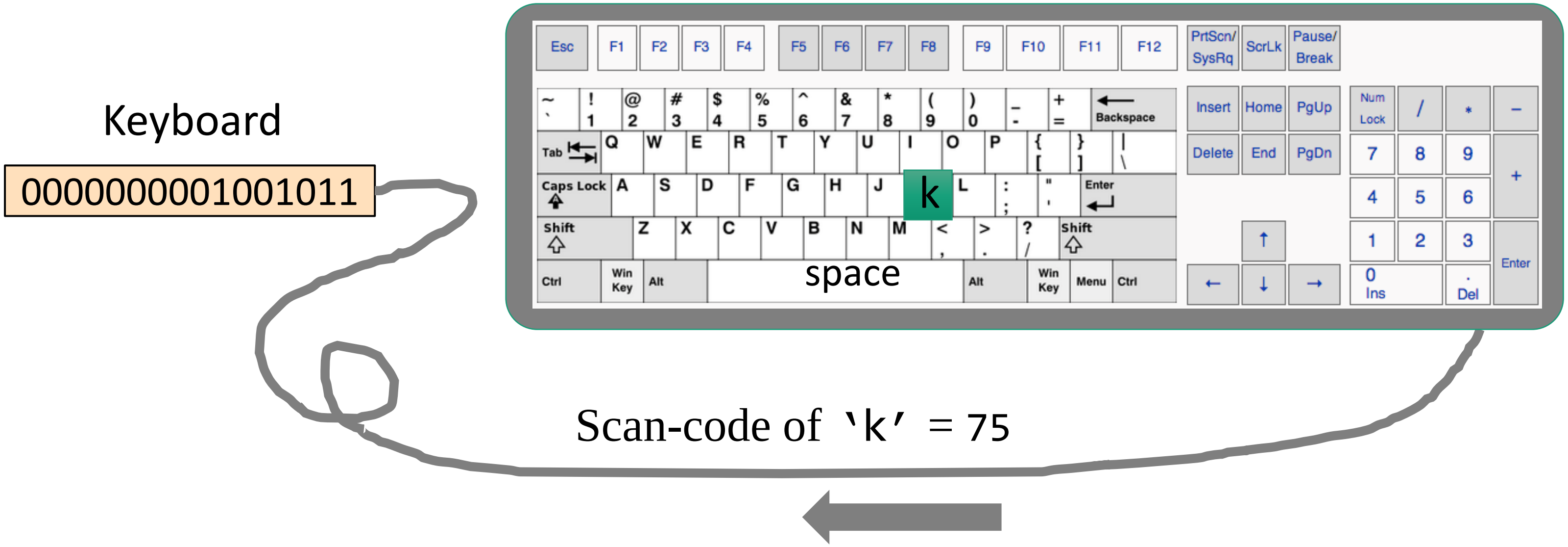
[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

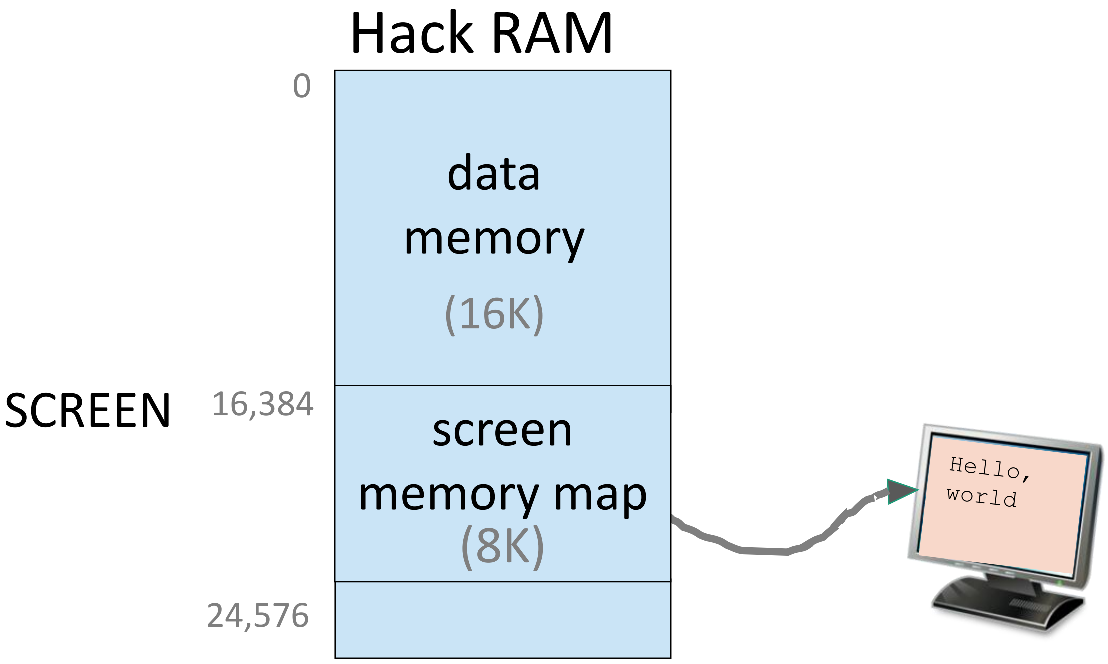
key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
…	…
f12	152

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]**.

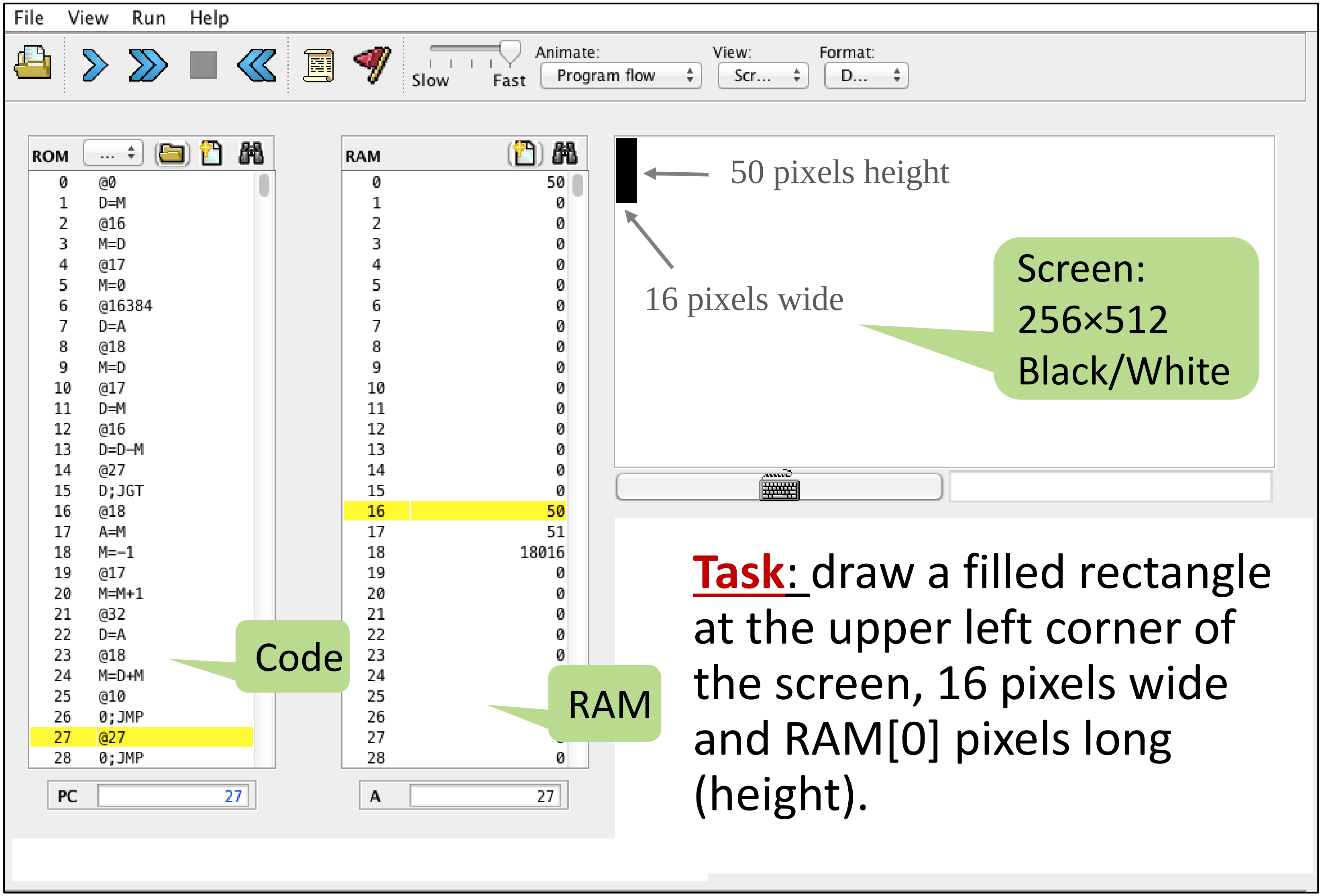
Output



Hack language convention:

- SCREEN: base address of the screen memory map

Handling the screen (example)



ROM:

```
0   @0
1   D=M
2   @16
3   M=D
4   @17
5   M=0
6   @16384
7   D=A
8   @18
9   M=D
10  @17
11  D=M
12  @16
13  D=D-M
14  @27
15  D;JGT
16  @18
17  A=M
18  M=-1
19  @17
20  M=M+1
21  @32
22  D=A
23  @18
24  M=D+M
25  @10
26  0;JMP
27  @27
28  0;JMP
```

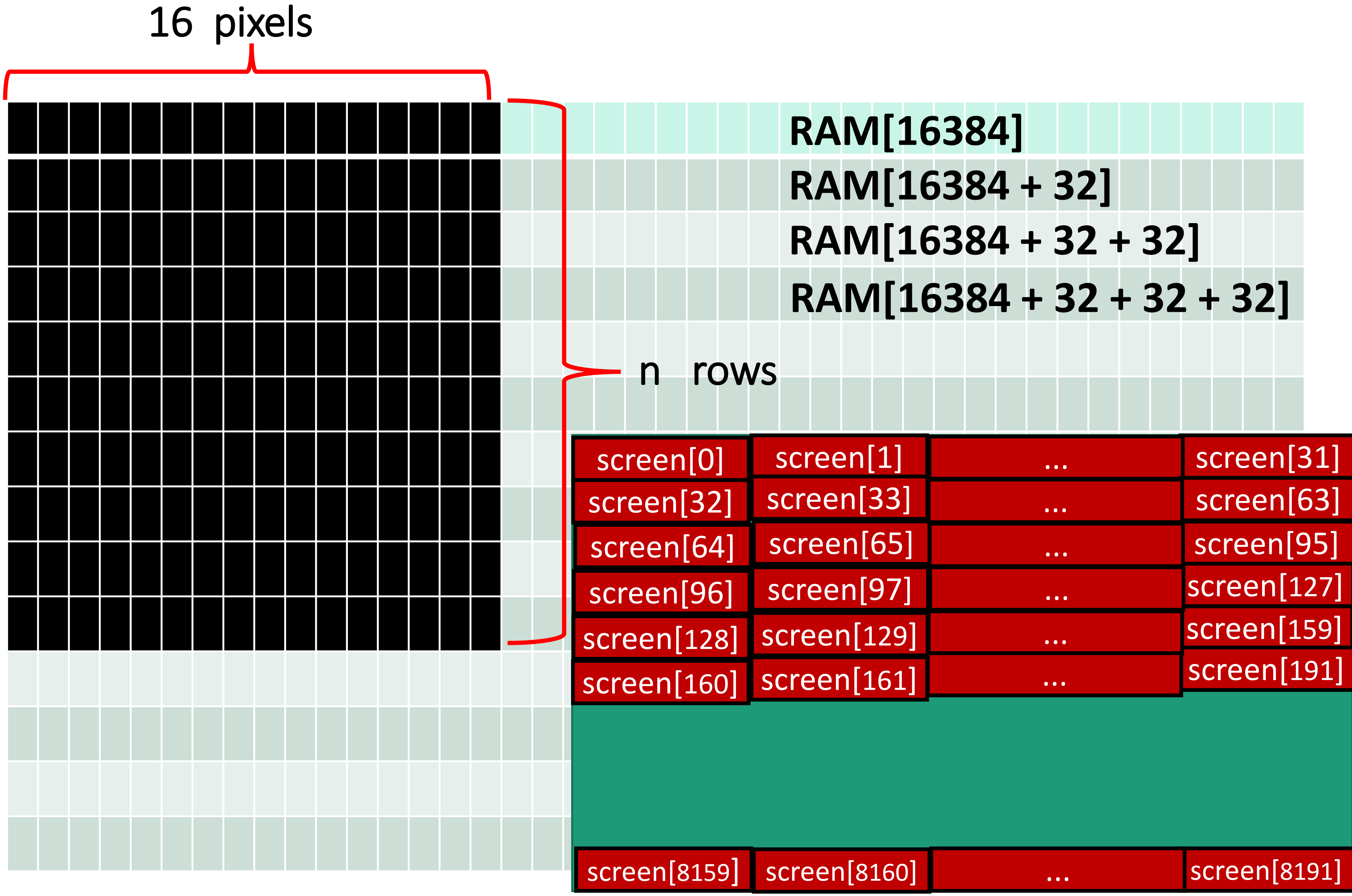
RAM:

0	50
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	50
17	51
18	18016
19	0
20	0
21	0
22	0
23	0
24	
25	
26	
27	0
28	0

PC 27

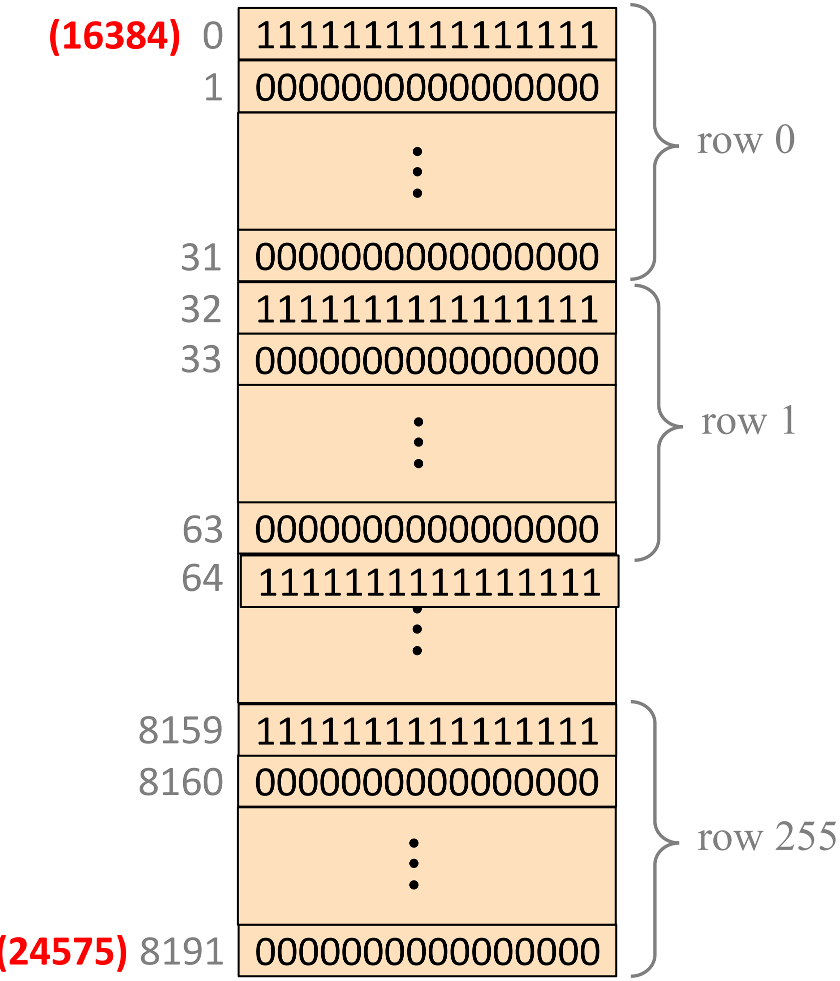
A 27

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



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]

Screen


(16384) 0 1111111111111111

1 0000000000000000

⋮

31 0000000000000000

row 0

32 1111111111111111

33 0000000000000000

⋮

63 0000000000000000

row 1

64 1111111111111111

⋮

8159 1111111111111111

8160 0000000000000000

⋮

(24575) 8191 0000000000000000

row 255

Two's Complement

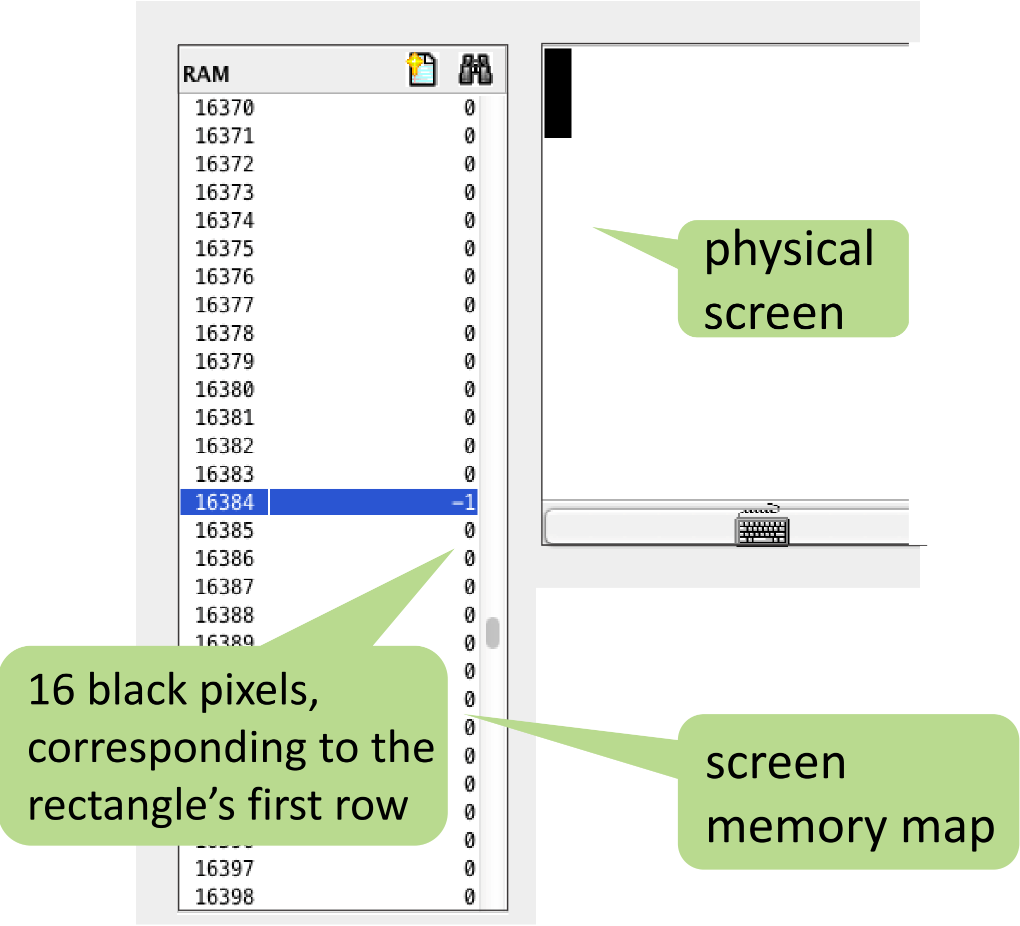
1 = 0000000000000001

-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
```



```
16370     0
16371     0
16372     0
16373     0
16374     0
16375     0
16376     0
16377     0
16378     0
16379     0
16380     0
16381     0
16382     0
16383     0
16384    -1
16385     0
16386     0
16387     0
16388     0
16389     0
          0
          0
          0
          0
          0
          0
          0
          0
16397     0
16398     0
```

physical screen

16 black pixels, corresponding to the rectangle's first row

screen memory map

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)

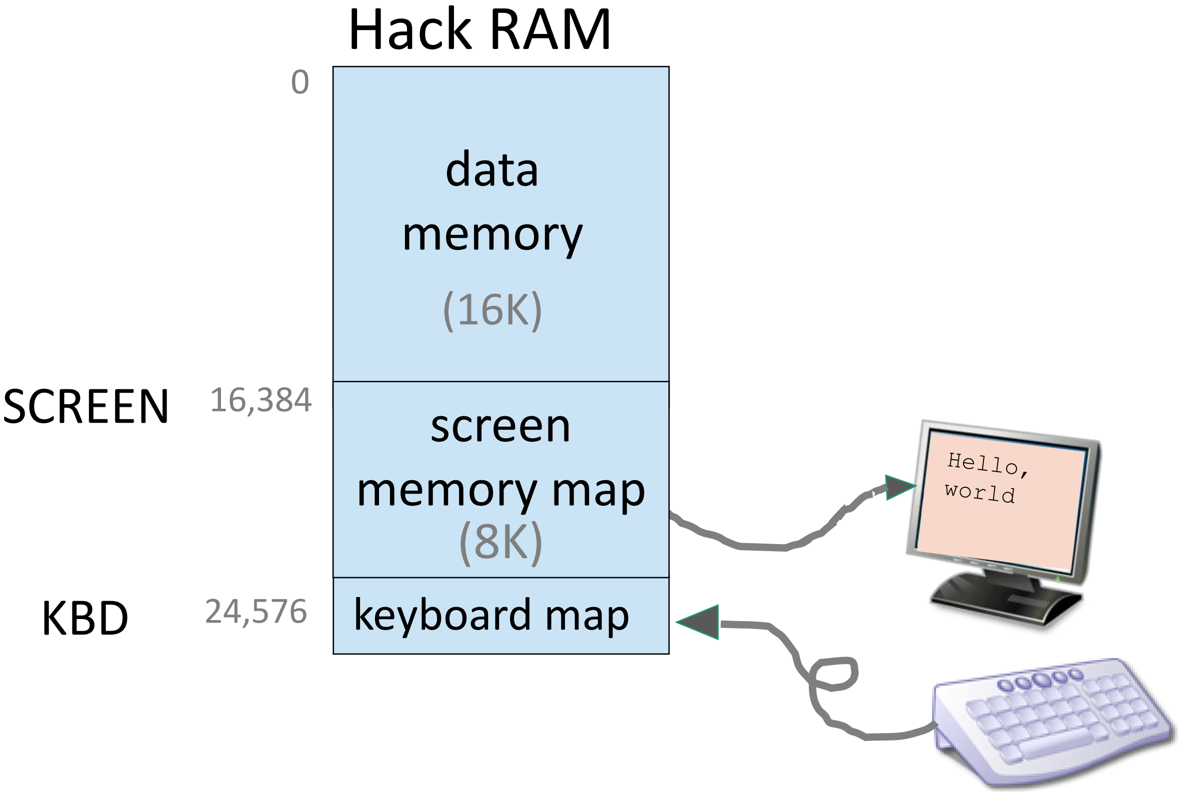
Pseudo code

```
// 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
```

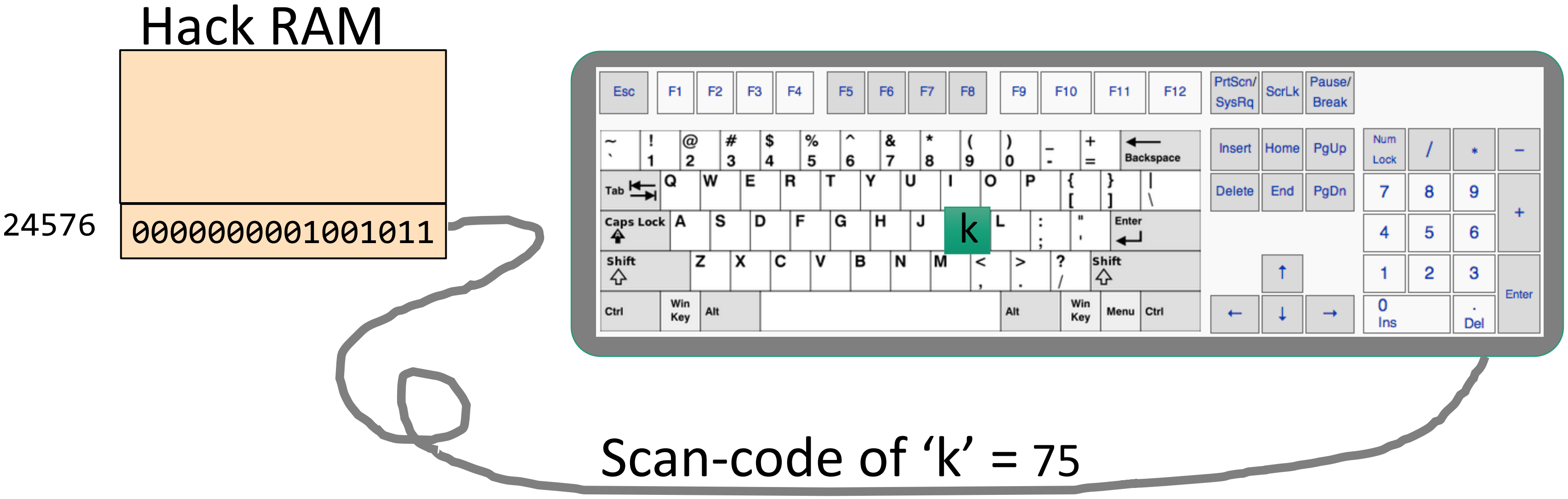
Input



Hack language convention:

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

Handle the keyboard



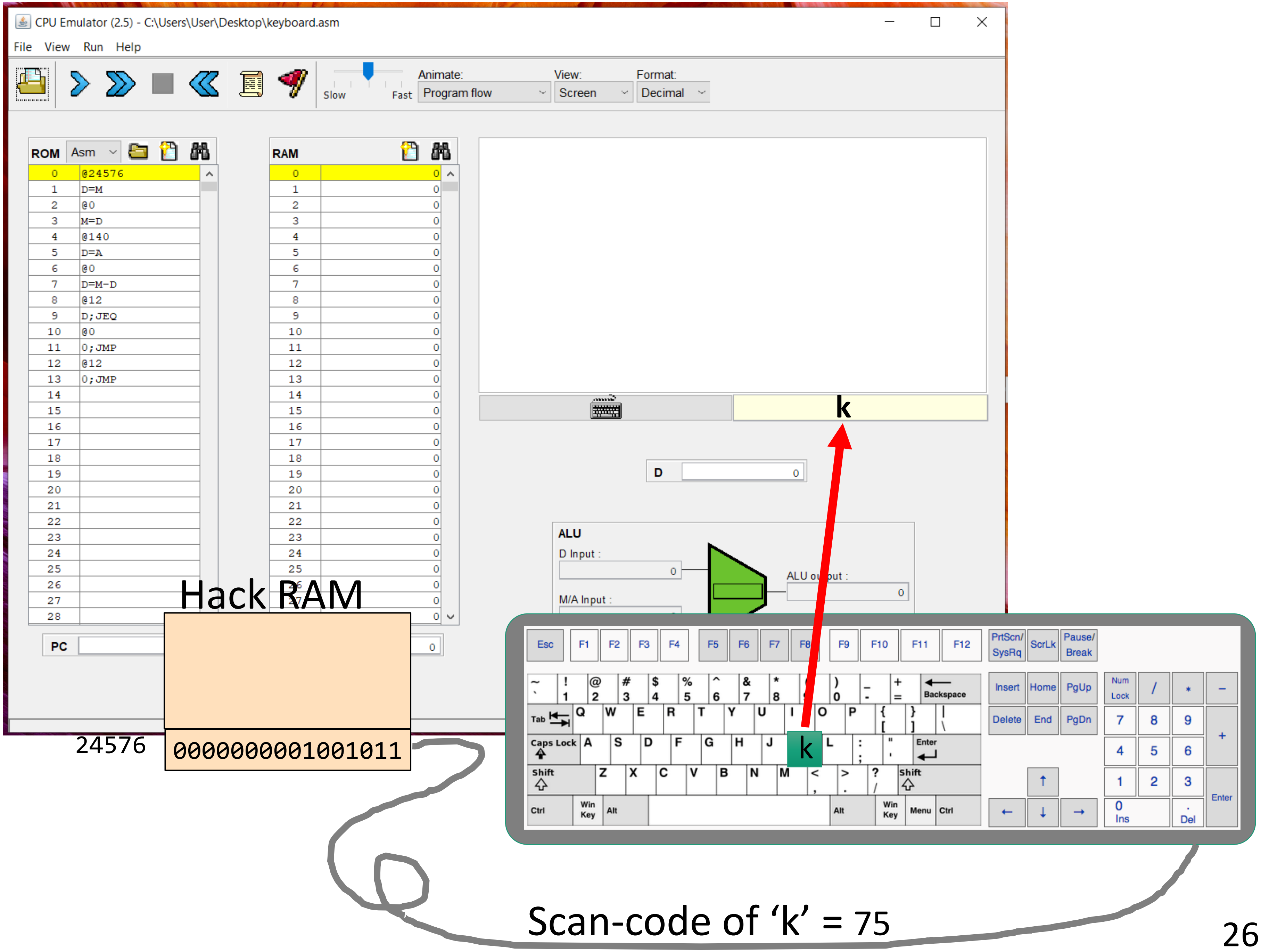
Hack RAM

24576 `0000000001001011`

Scan-code of ‘k’ = 75

<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.



Animate: Program flow
View: Screen
Format: Decimal

```
0   @24576
1   D=M
2   @0
3   M=D
4   @140
5   D=A
6   @0
7   D=M-D
8   @12
9   D;JEQ
10  @0
11  0;JMP
12  @12
13  0;JMP
```

RAM 0–28: 0

PC: 0

D: 0

D Input : 0

ALU output : 0

Hack RAM

24576 0000000001001011

Scan-code of ‘k’ = 75

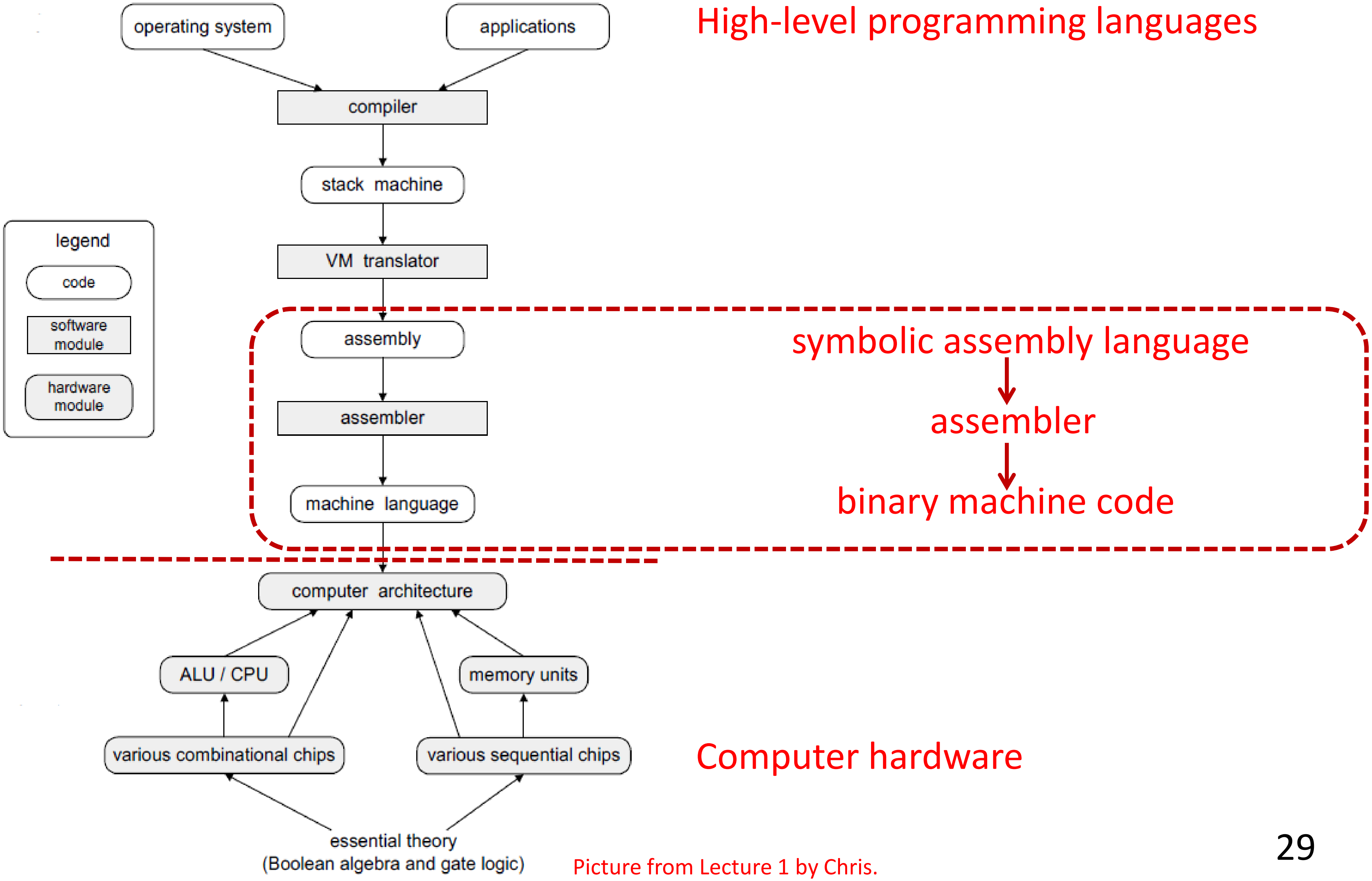
Keyboard input (example)

```
// 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
```

Outlines

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

Overview of computer system



High-level programming languages

symbolic assembly language

↓

assembler

↓

binary machine code

Computer hardware

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

Hack binary code

(target language)

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
...
```

What are the rules of the game?

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
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
...
```

Hack language specification: A-instruction

Symbolic syntax:

`@` *value*

Where *value* is either

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

Examples:

`@21`

`@foo`

Binary syntax:

0 *valueInBinary*

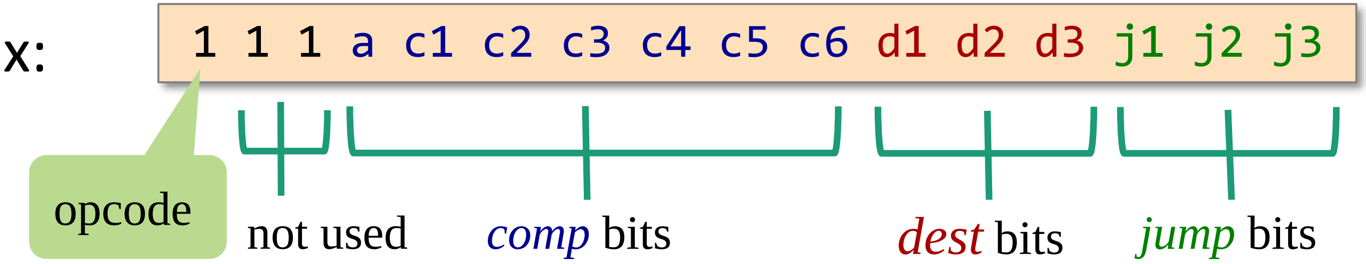
Example:

0000000000010101

Hack language specification: C-instruction

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 language specification: symbols

Pre-defined symbols:

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576

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

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

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

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

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
```

Challenges:

Handling...

- White space
- Instructions

Handling programs without symbols

Assembly program (without symbols)

```
@16
M=1
@17
M=0
@16
D=M
@0
D=D-M
@18
D;JGT
@16
D=M
@17
M=D+M
@16
M=M+1
@4
0;JMP
@17
D=M
@1
M=D
@22
0;JMP
```

Assembler
for symbol-less
Hack programs

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
```

<u>Challenges:</u>

Handling…

- White space
  - Ignore it
- Instructions

Translating A-instructions

Symbolic syntax:

@ *value*

Where *value* is either
- a non-negative decimal constant or
- a symbol referring to such a constant

Examples:

@21

@foo

Binary syntax:

0 *valueInBinary*

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

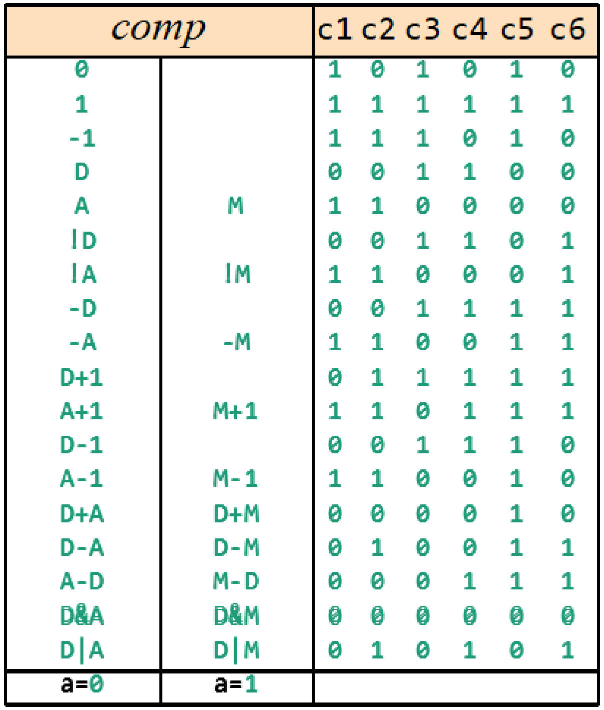
Exercise: translate assembly code

- Translate the following assembly code to binary code:

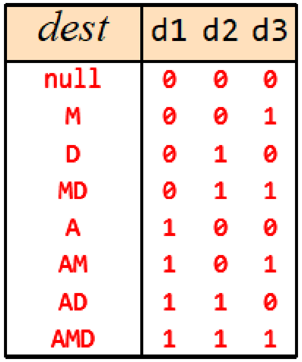
```
@10
D=A
@R10
M=D
(END)
@END
0;JMP
```

Binary syntax:

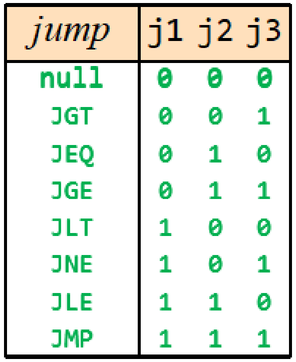
1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3



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.

The overall assember logic

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
```

For each instruction

- **Parse** the instruction: break it into its underlying fields
- **A-instruction**:
  Translate the decimal value into a binary string.
- **C-instruction**:
  For each field in the instruction, **generate** corresponding binary code.
- **Assemble** translated binary codes into a complete 16-bit machine instruction.
- **Write** 16-bit instruction to the output file.

Outlines

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

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
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576

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

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`

Symbol table

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
```

Symbol table

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
STOP	18
END	22
i	16
sum	17

Initialization:
Add the pre-defined symbols

First pass:
Add the label symbols

Second pass:
Add the var symbols

Usage:

To resolve a symbol, look up its value in the symbol table.

The assembly process

- Initialization: handle predefined symbols.
  - ➢Construct an empty symbol table.
  - ➢Add the pre-defined symbols to the symbol table.
- First pass: handle label symbols.
  - ➢Scan the entire program;
  - ➢For each “instruction” of the form **`(xxx)`**:

    Add the pair (xxx, *address*) to the symbol table, where *address* is the number of the instruction following `(xxx)`.
- Second pass: handle variable symbols and instructions

The assembly process - second pass

- Set *n* to 16
- Scan the entire program again; for each instruction:
  - If it is @*symbol*, look up *symbol* in symbol table;
    - If (*symbol, value*) is found, use *value* to complete instruction’s translation;
    - If not found:
      - ✓ Add (*symbol*, *n*) to the symbol table,
      - ✓ Use *n* to complete instruction’s translation,
      - ✓ *n*++
  - If it is a C-instruction, complete instruction’s translation
  - Write translated instruction to output file.

Outlines

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

Develop an assembler

- **Reading** and **parsing** commands.
- **Converting** mnemonics to codes.
- **Handling** symbols.

Assumption: The symbolic code is error-free.

Read and parse commands

- Start reading a file with a given name.
  - Constructor for a **Parser** object that accepts a string specifying a file name.
  - Handle reading text files.
- Move to the next command in the file
  - Are we finished? `boolean hasMoreCommands()`.
  - Get the next command: `void advance()`.
  - Read one line at a time.
  - Skip whitespace including comments.
- Get the fields of the current command.
  - ... more on next slide

Read and parse commands

- Get the fields of the current command.
    - ➢Type of current command (A-Command, C-Command, or Label)
    - ➢Handle fields as strings:

```
D=M+1;   JGT                        @sum
```

```
String dest(); String comp(); String jump();                    String variable();
```

Translate mnemonic to code

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
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

Recap: parsing + translating

Symbolic syntax:

dest = *comp* ; *jump*

Binary syntax:

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

```
// Assume that current command is
//      D = M+1; JGT

String c=parser.comp();   // “M+1”
String d=parser.dest();   // “D”
String j=parser.jump();   // “JGT”

String cc = Code.comp(c);   // “1110111”
String dd = Code.dest(d);   // “010”
String jj = Code.jump(j);   // “001”

String out = “111” + cc + dd + jj;
```

- Parser
  - ➢ Parse the command line in symbolic syntax.
- Coder
  - ➢ Generate binary code according to binary syntex.

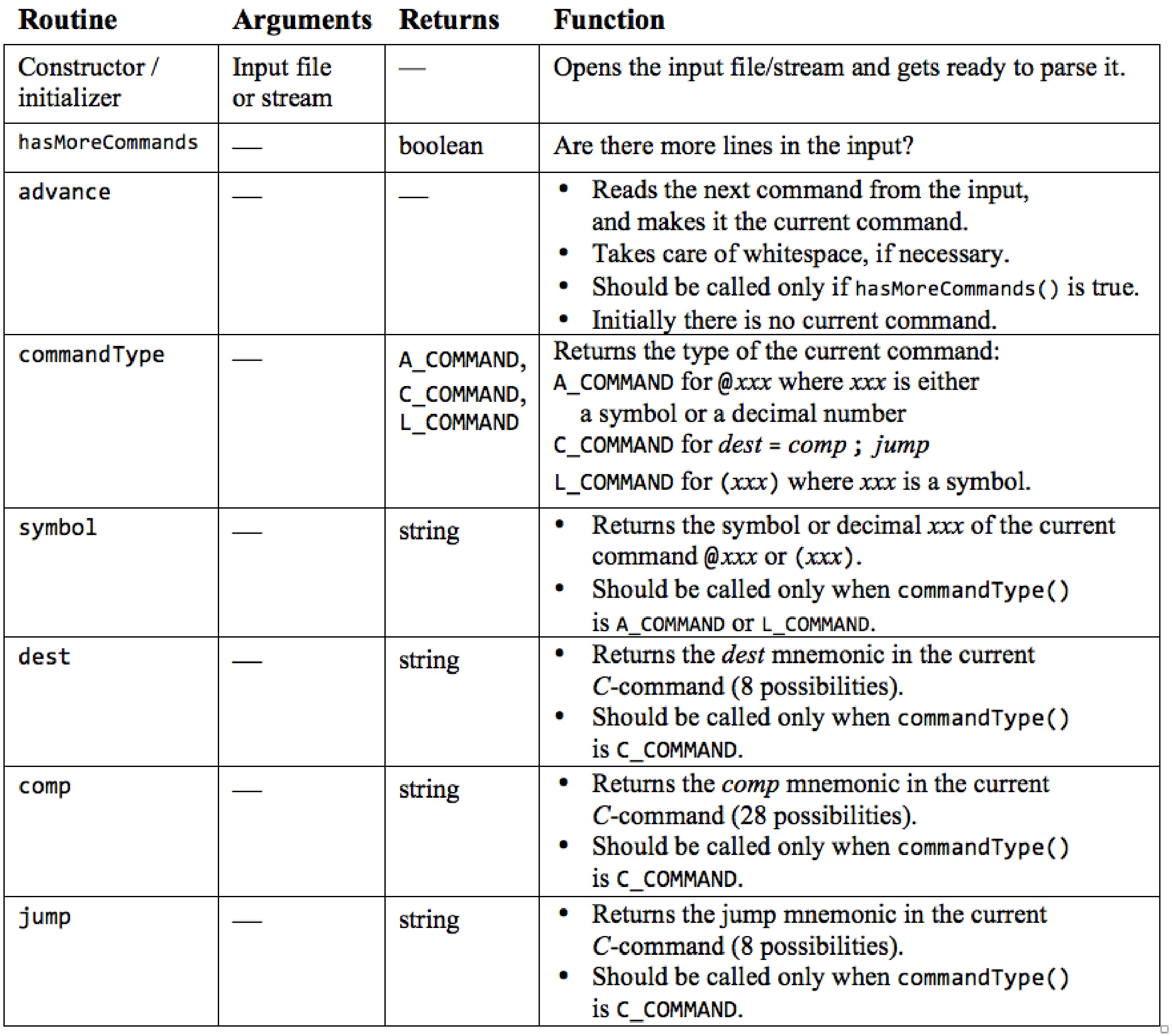
The symbol table

- Create a new empty table,
- Add all the **pre-defined symbols** to the table,
- While reading input, add labels and variables to table.
  - **Labels:** for “(*xxx*)” command, add the symbol *xxx* and the address of the next machine language command.
    - Maintain this running address,
    - Do it in the first pass.
  - **Variables:** for “@*xxx*” command, if *xxx* not a number, nor in the table, add symbol *xxx* and the next free address to table.
- For “@*xxx*” command, if *xxx* not a number, consult the table to replace symbol *xxx* with its address.

Overall logic

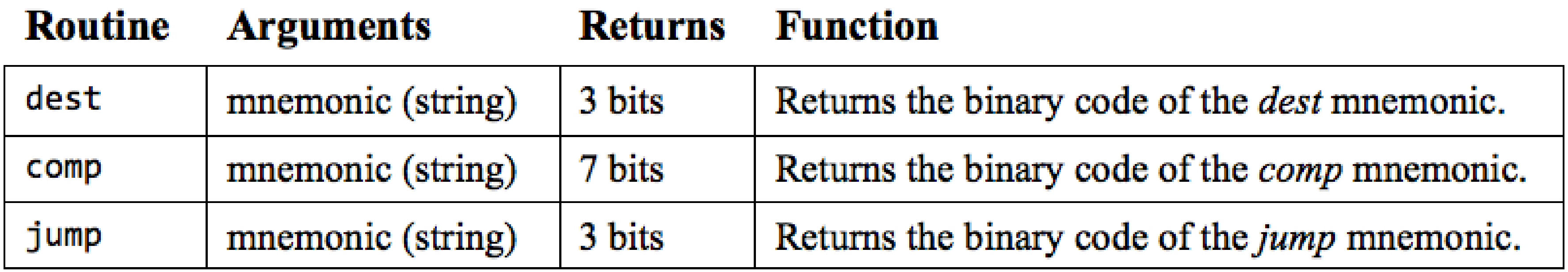
- Initialization
    - Of ***Parser***,
    - Of ***Symbol Table**, e.g.* add built-in symbols.
- First pass: read all commands, pay attention to ***labels*** and update the ***Symbol Table***,
- Second pass: restart reading and translating commands, in a main loop, pay attention to ***variables***.
    - Get the next assembly language command and parse it,
    - For ***A***-commands: Translate symbols to binary addresses,
    - For ***C***-commands: get code for each part and put them together,
    - Output the resulting machine language command.

`Parser` module



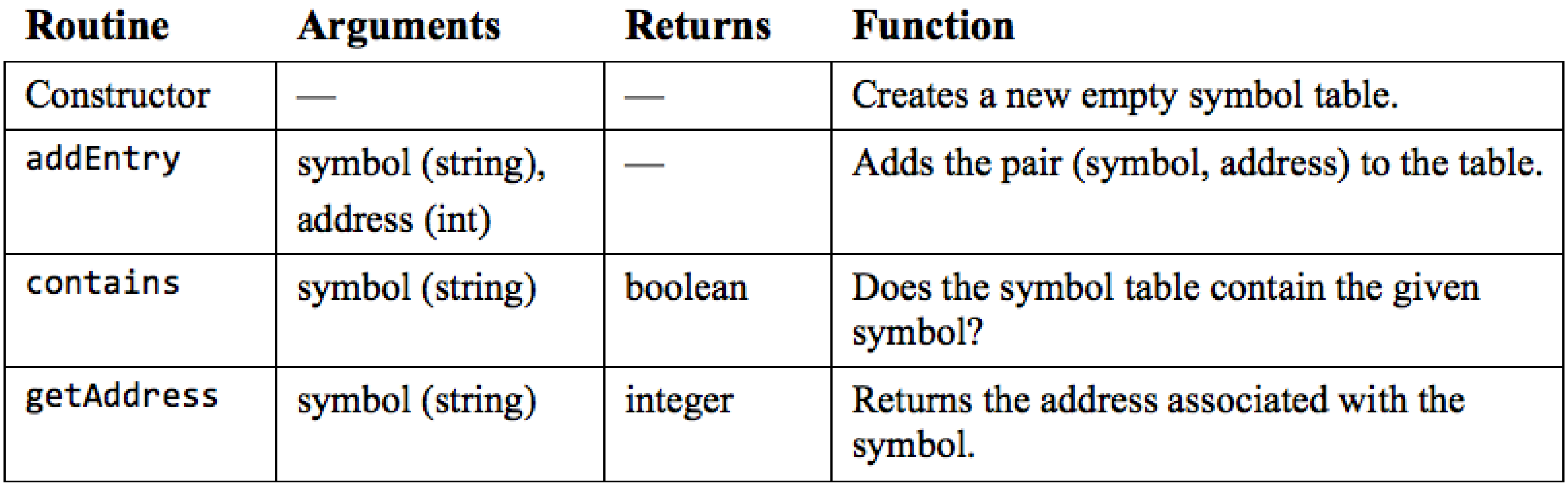
Routine	Arguments	Returns	Function
Constructor / initializer	Input file or stream	—	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	—	boolean	Are there more lines in the input?
`advance`	—	—	• Reads the next command from the input, and makes it the current command. • Takes care of whitespace, if necessary. • Should be called only if `hasMoreCommands()` is true. • Initially there is no current command.
`commandType`	—	`A_COMMAND`, `C_COMMAND`, `L_COMMAND`	Returns the type of the current command: `A_COMMAND` for @*xxx* where *xxx* is either a symbol or a decimal number `C_COMMAND` for *dest* = *comp* ; *jump* `L_COMMAND` for (*xxx*) where *xxx* is a symbol.
`symbol`	—	string	• Returns the symbol or decimal *xxx* of the current command @*xxx* or (*xxx*). • Should be called only when `commandType()` is `A_COMMAND` or `L_COMMAND`.
`dest`	—	string	• Returns the *dest* mnemonic in the current *C*-command (8 possibilities). • Should be called only when `commandType()` is `C_COMMAND`.
`comp`	—	string	• Returns the *comp* mnemonic in the current *C*-command (28 possibilities). • Should be called only when `commandType()` is `C_COMMAND`.
`jump`	—	string	• Returns the jump mnemonic in the current *C*-command (8 possibilities). • Should be called only when `commandType()` is `C_COMMAND`.

Code module



Routine	Arguments	Returns	Function
`dest`	mnemonic (string)	3 bits	Returns the binary code of the *dest* mnemonic.
`comp`	mnemonic (string)	7 bits	Returns the binary code of the *comp* mnemonic.
`jump`	mnemonic (string)	3 bits	Returns the binary code of the *jump* mnemonic.

`SymbolTable` module



Routine	Arguments	Returns	Function
Constructor	—	—	Creates a new empty symbol table.
`addEntry`	symbol (string), address (int)	—	Adds the pair (symbol, address) to the table.
`contains`	symbol (string)	boolean	Does the symbol table contain the given symbol?
`getAddress`	symbol (string)	integer	Returns the address associated with the symbol.

Summary

- Introduction to assembler
  - ➢Translate assembly language (symbolic code) to machine language (binary code).
- Assembly process
  - ➢White space, including comments. (Simply ignore it)
  - ➢Instructions. (Parser and Code)
  - ➢Symbols. (SymbolTable)
- Develop an assembler
  - ➢Initialization: Parser and SymbolTable (Predefined symbols)
  - ➢First pass: handle labels.
  - ➢Second pass: handle variables and instructions.

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.