Machine Language
Part 2

Dr. Wooi Ping Cheah

Outlines

- Hack assembly programming
  - Registers and memory
  - Branching, variables, iteration
  - Pointers
- Hack input / output

Hack assembly language (overview)

A-instruction:

@*value* // A = value

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

C-instruction:

dest = *comp* ; *jump*

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

where:

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

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

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

Semantics:

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

Hack assembler

Assembly program

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

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

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

(END)
   @END
   0;JMP
```

Hack assembler

Binary code

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

load & execute

Use Mnemonics

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

CPU emulator

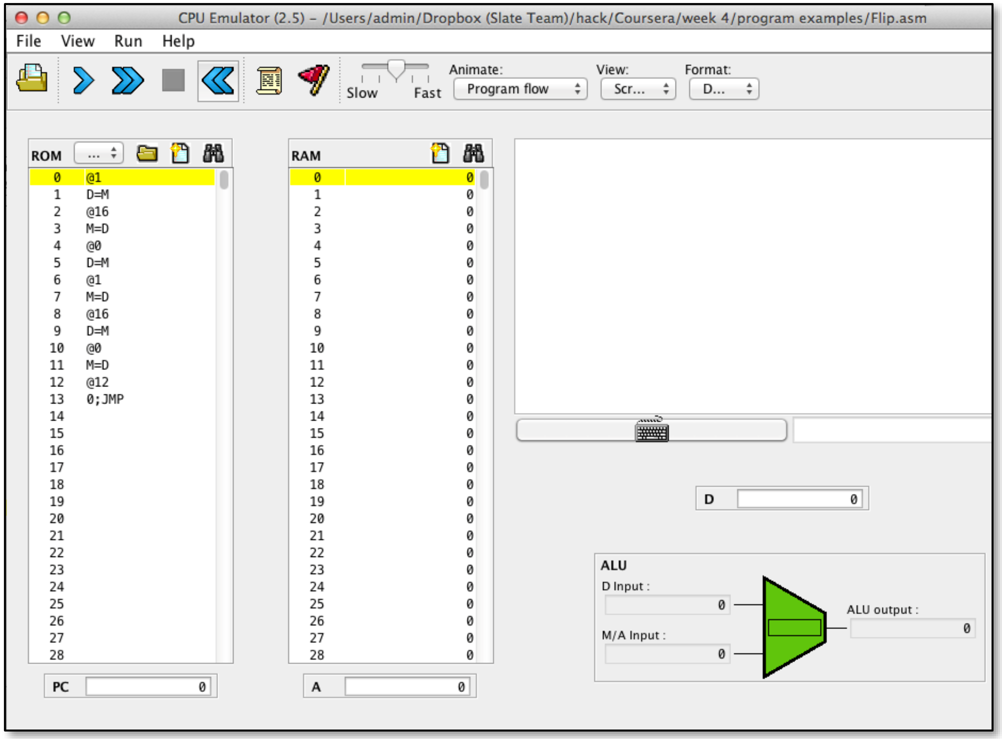
Assembly program

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

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

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

(END)
   @END
   0;JMP
```

load

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

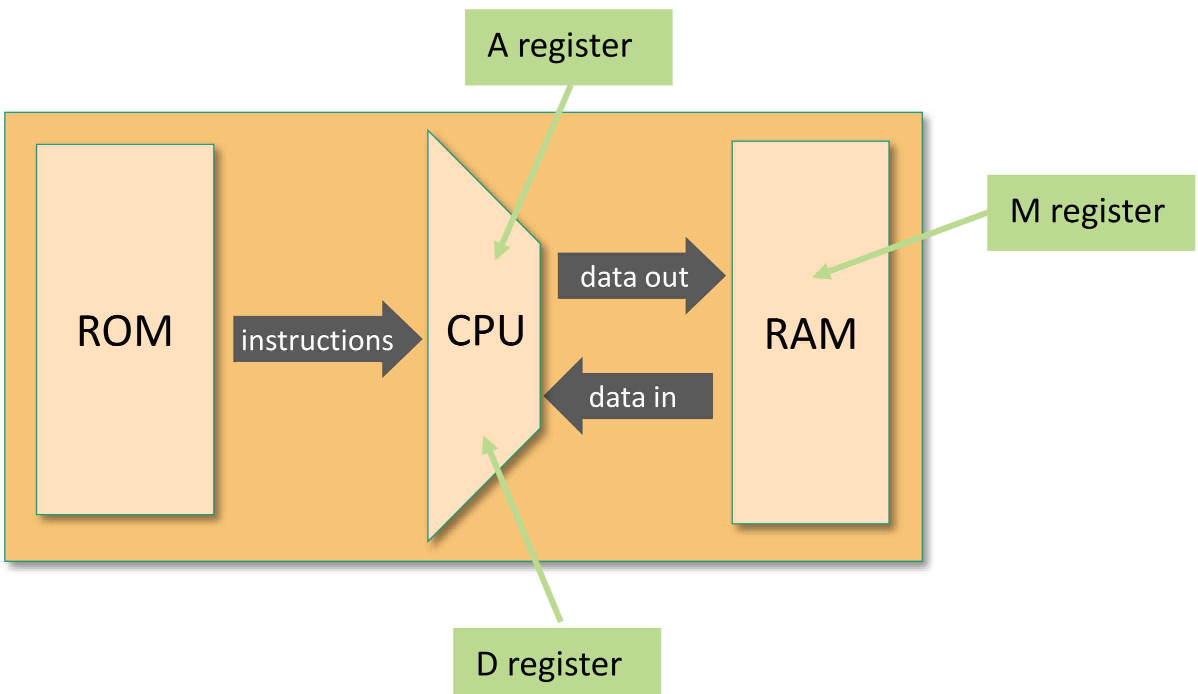
CPU Emulator



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

Registers and memory

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



Registers and memory

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

Typical operations:

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

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

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

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

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

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

Program example: add two numbers

Hack assembly code

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

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

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

translate and load

(white space ignored)

Memory (ROM)

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

symbolic view

Program example: add two numbers

Hack assembly code

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

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

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

translate and load

(white space ignored)

Memory (ROM)

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

binary view

Terminate a program

Hack assembly code

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

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

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

translate and load

Memory (ROM)

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

malicious code starts here ...

Attack on the computer

Terminate a program

Hack assembly code

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

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

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

6  @6
7  0;JMP
```

translate and load

Memory (ROM)

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

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

<u>Best practice:</u>

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

Built-in symbols

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

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

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

These symbols can be used to denote “virtual registers”

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

implementation:

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

@5
M=D
```

better style:

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

@R5
M=D
```

Built-in symbols

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

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

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

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

Outlines

- Hack assembly programming
  - Registers and memory
  - Branching, variables, iteration
  - Pointers

- Hack input / output

Branching

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

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

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

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

8      @R1
9      M=1    // RAM[1]=1

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

RAM

	M
0	-4
1	0
2	
3	
4	
	. . .
100	
101	
102	
103	
. . .	

	M
0	4
1	1
2	
3	
4	
	. . .
100	
101	
102	
103	
. . .	

For condition jump, D register will be used in checking the condition.

Labels

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

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

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

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

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

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

referring to a label

declaring a label

resolving labels

ROM

Memory

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

Label resolution rules:

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

Labels

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

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

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

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

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

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

referring to a label

declaring a label

resolving labels

Memory

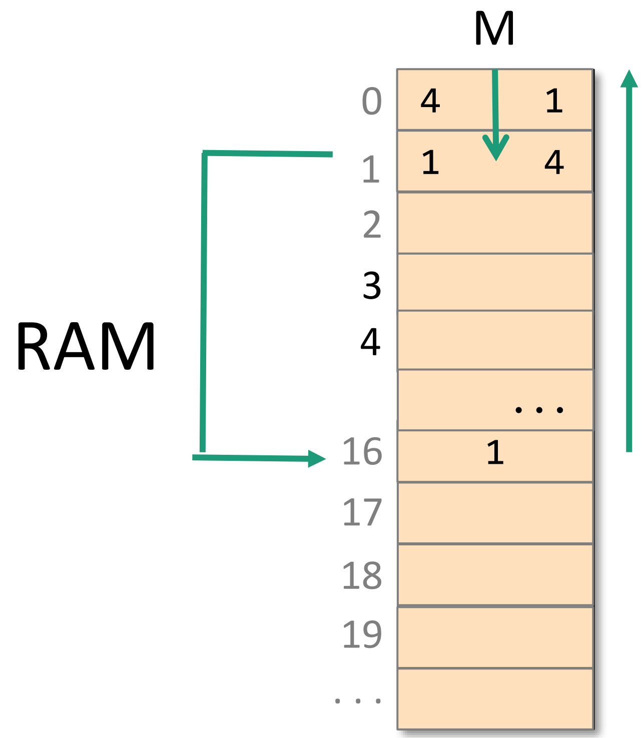
0	`@0`
1	`D=M`
2	`@8      // @POSITIVE`
3	`D;JGT`
4	`@1`
5	`M=0`
6	`@10     // @END`
7	`0;JMP`
8	`@1`
9	`M=1`
10	`@10     // @END`
11	`0;JMP`
12	
13	
14	
15	
	⋮
32767	

Implications:

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

Variables

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

// RAM[16] = R1
// R1 = R0
// R0 = RAM[16]

   @R1
   D=M
   @16
   M=D     // RAM[16] = R1

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

   @16
   D=M
   @R0
   M=D     // R0 = RAM[16]

(END)
   @END
   0;JMP
```



Variables

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

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

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

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

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

(END)
   @END
   0;JMP
```

symbol used for the first time

symbol used again

resolving symbols

<u>Symbol resolution rules:</u>

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

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

Memory

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

Variables

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

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

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

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

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

(END)
   @END
   0;JMPa
```

resolving symbols

Implications:

symbolic code is easy to read and debug

Memory

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

Iterative processing

pseudo code

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

assembly code

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

   @i
   M=1   // i = 1

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

M

		M
	0	10
	1	55
	2	
	3	
	4	
		...
n	16	10
i	17	1
sum	18	0
	19	

Memory

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

Variables are allocated to consecutive RAM locations from address 16 onward

Iterative processing

pseudo code

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

i > n ⇔ i – n > 0

assembly code

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

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

Program execution

assembly program

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

Variable-Value Trace Table

iterations

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

Sample exam question

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

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

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

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

   @cur
   D=M
   @pre
   M=D

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

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

(END)
   @END
   0;JMP
```

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

From 2019-2020 CSF exam.

Hint: this piece of code implements Fibonacci number.

Writing assembly programs

assembly program

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

Best practice:

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

Outlines

- Hack assembly programming
  - Registers and memory
  - Branching, variables, iteration
  - Pointers

- Hack input / output

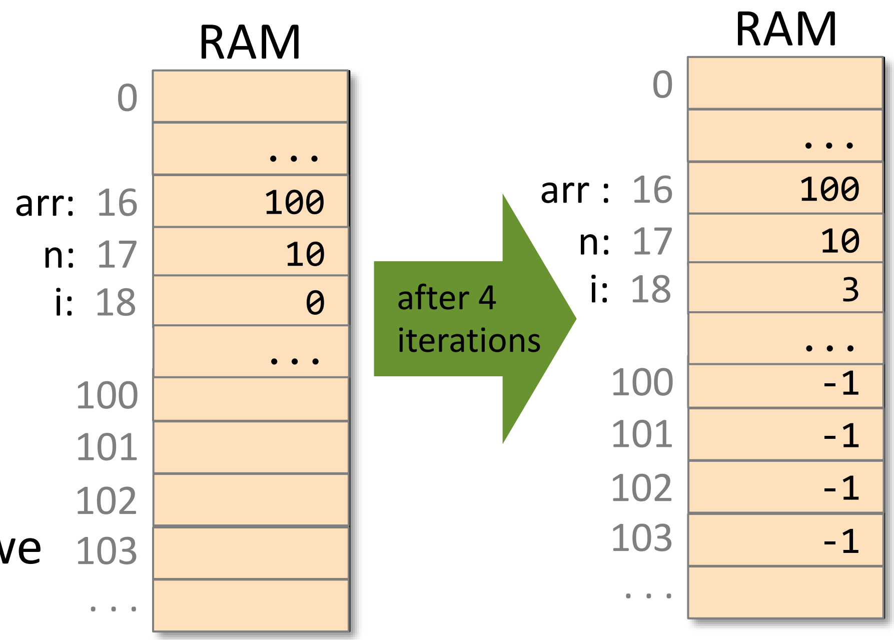
Pointers

Example:

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

<u>Observations:</u>

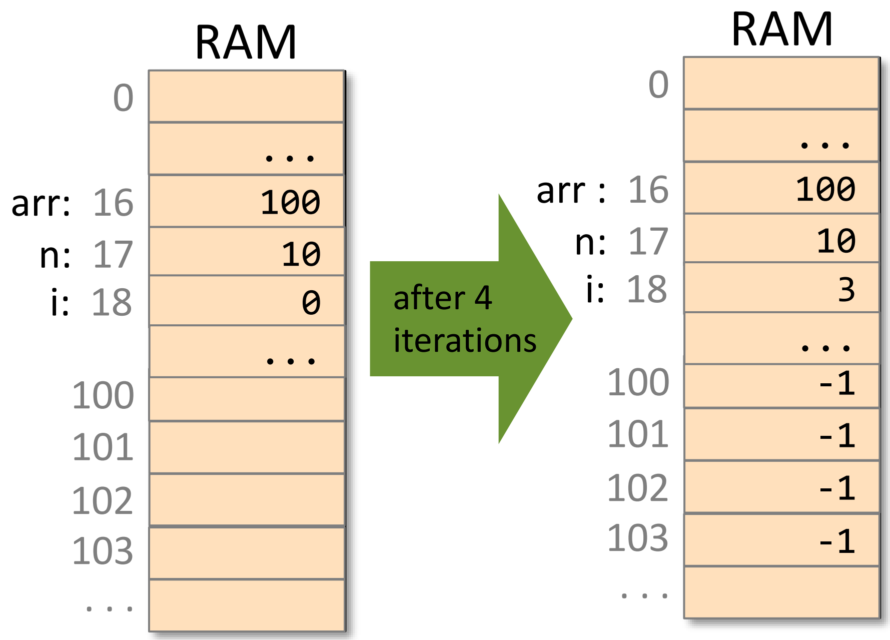
- The array is implemented as a **block of memories**.
- To access these memories one by one, we need a variable to hold the current address.
- *Variables that represent addresses* are called <u>**pointers**</u>.
- There is nothing special about pointer variables, except that their values are interpreted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



Pointers

Example:

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

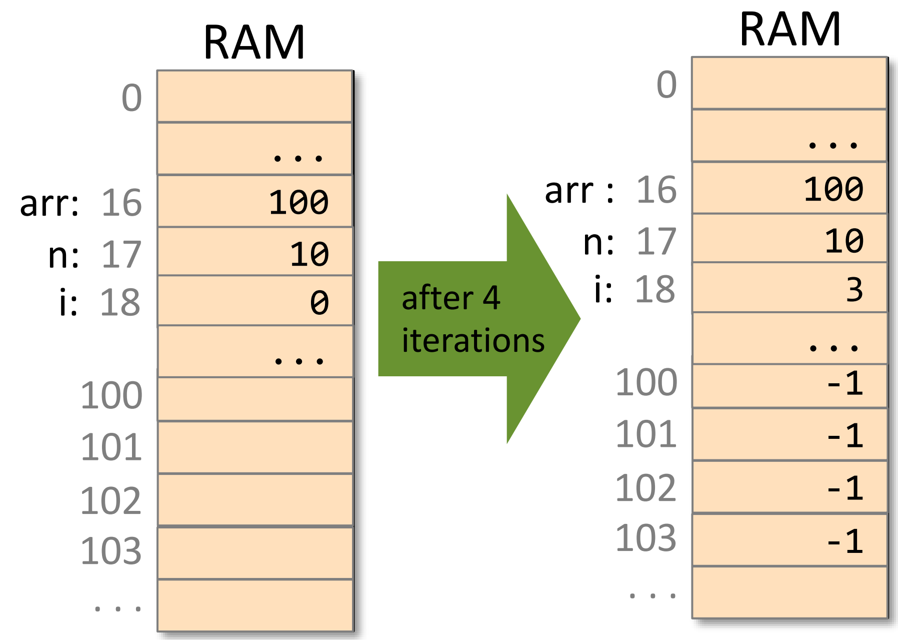


Pointers

Example:

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

typical pointer manipulation



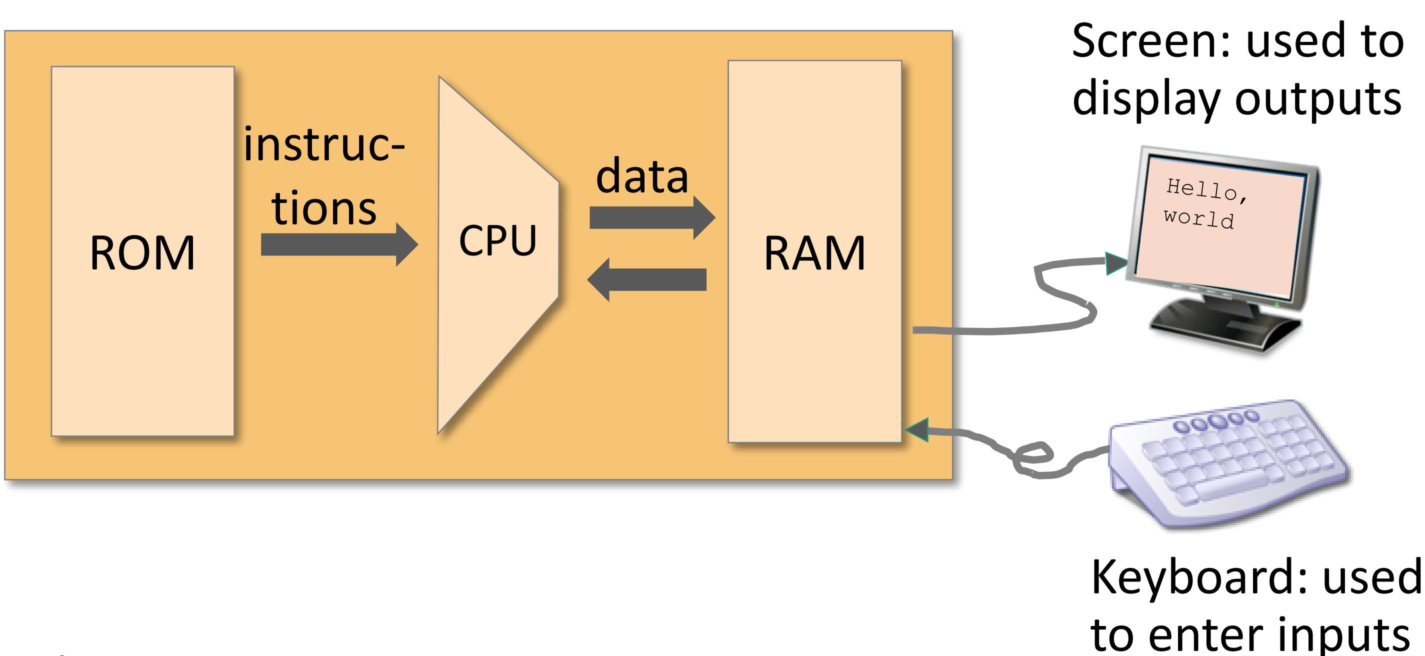
- Pointers: Variables that store memory addresses (like `arr`).
- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like **A =** ***expression***.
- Semantics:
  “set the address register to some value”.

Outlines

- Hack assembly programming
    - Registers and memory
    - Branching, variables, iteration
    - Pointers

- Hack input / output

Input / output



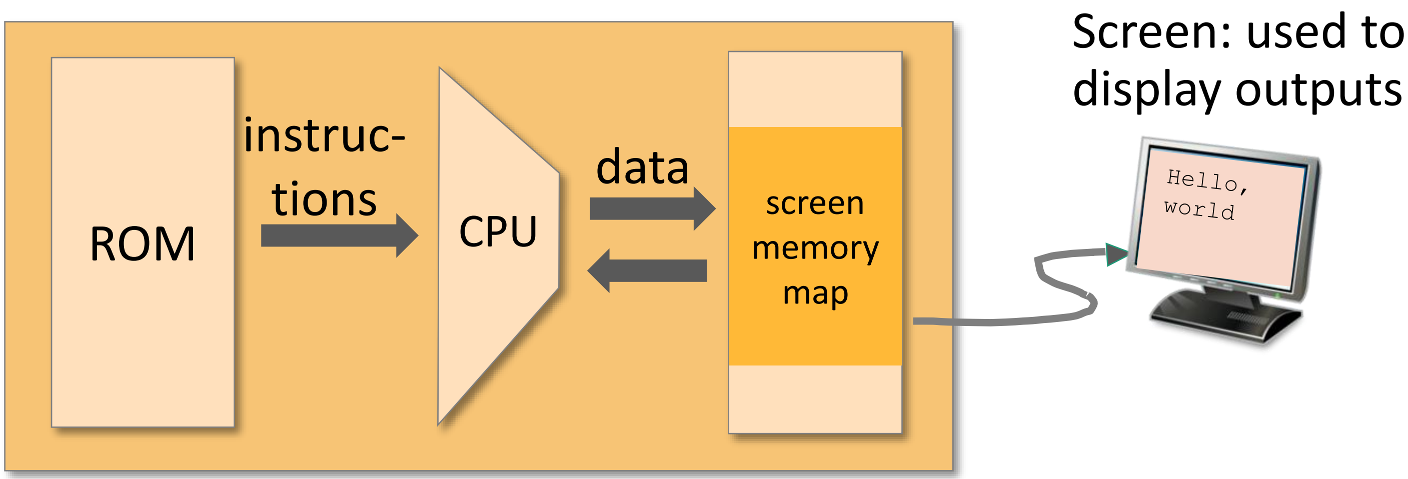
<u>I/O handling</u> (high-level):

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

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

Bit manipulation.

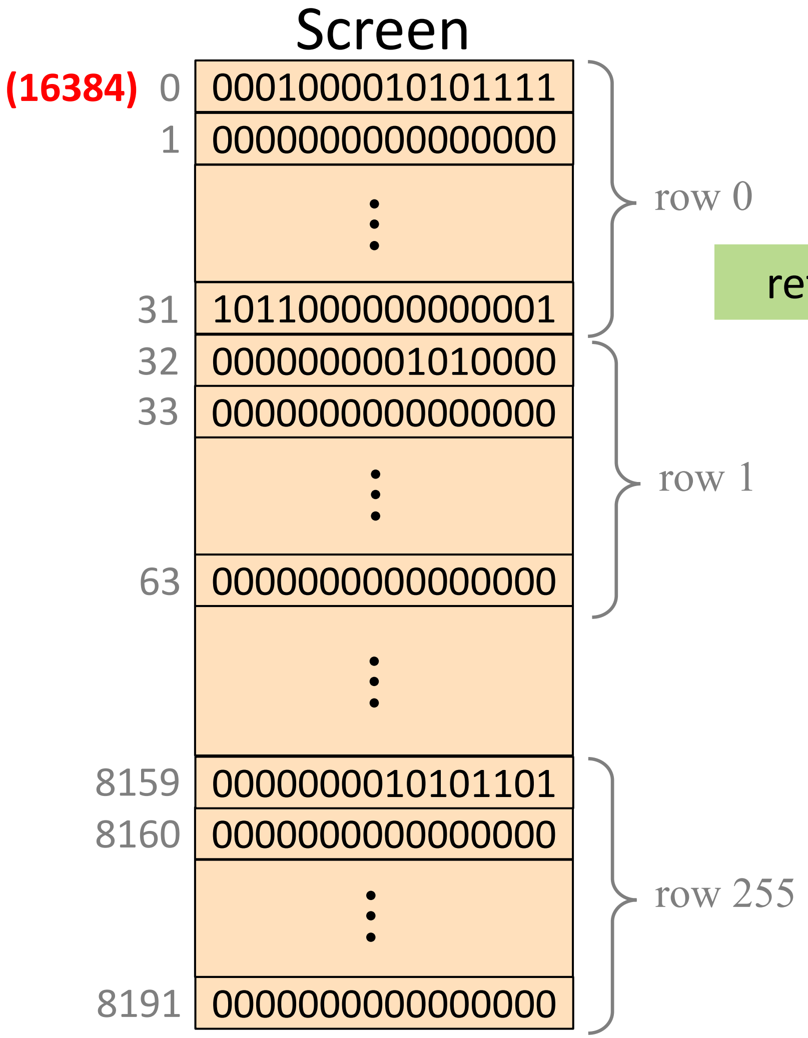
Memory mapped output



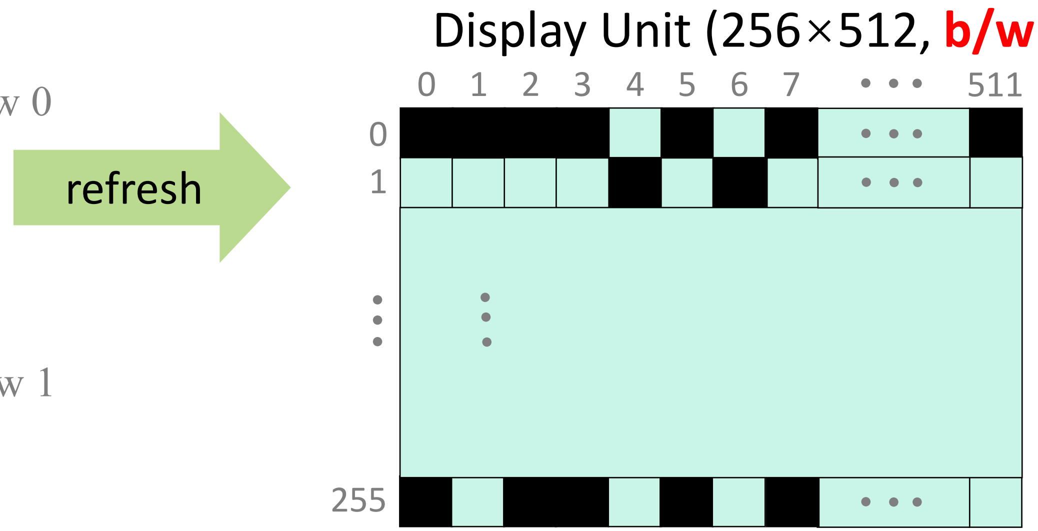
- Memory mapped output
  - ➢ A **designated memory** area to manage a display unit.
  - ➢ The physical display is continuously *refreshed* from the memory map, many times per second. (It is slow in Hack computer.)
  - ➢ Output is effected by writing code that manipulates the screen memory map.

Memory mapped output

16 X 32 = 512



8192 = 256×512/16

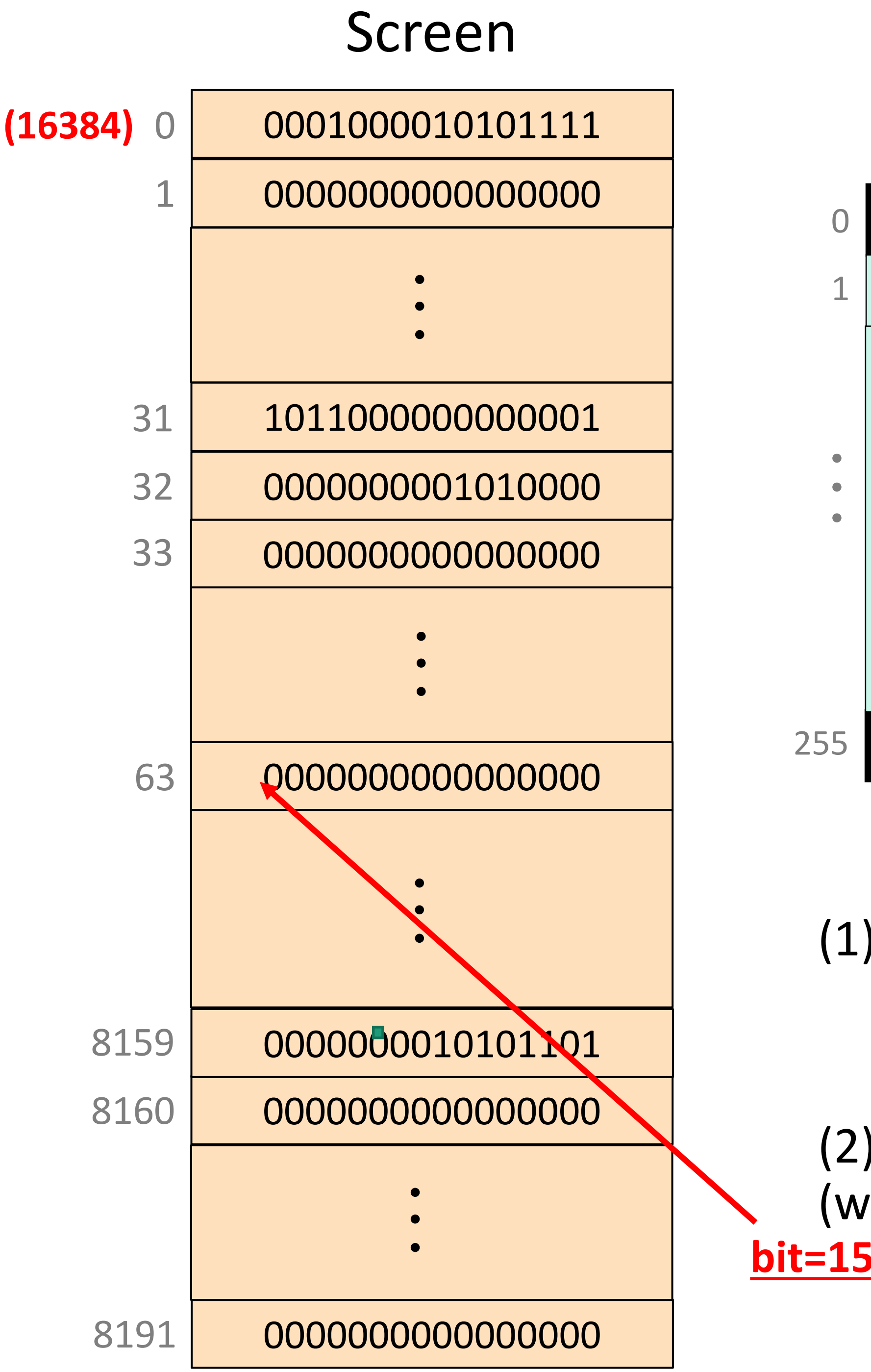


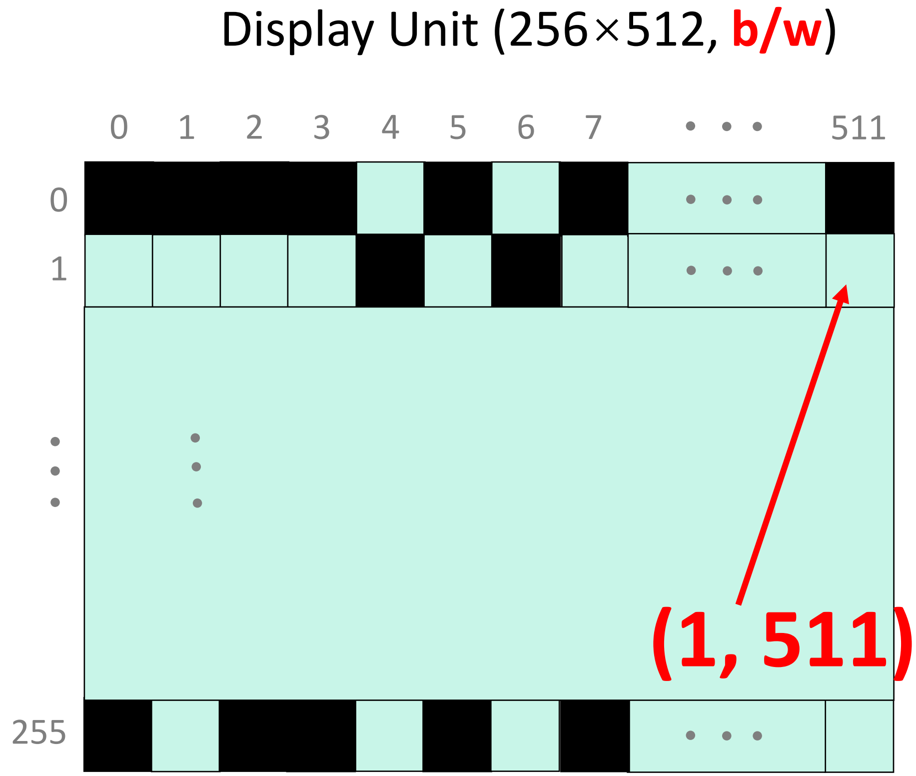
Integer division, round-down.

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

(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

32 31

word=63

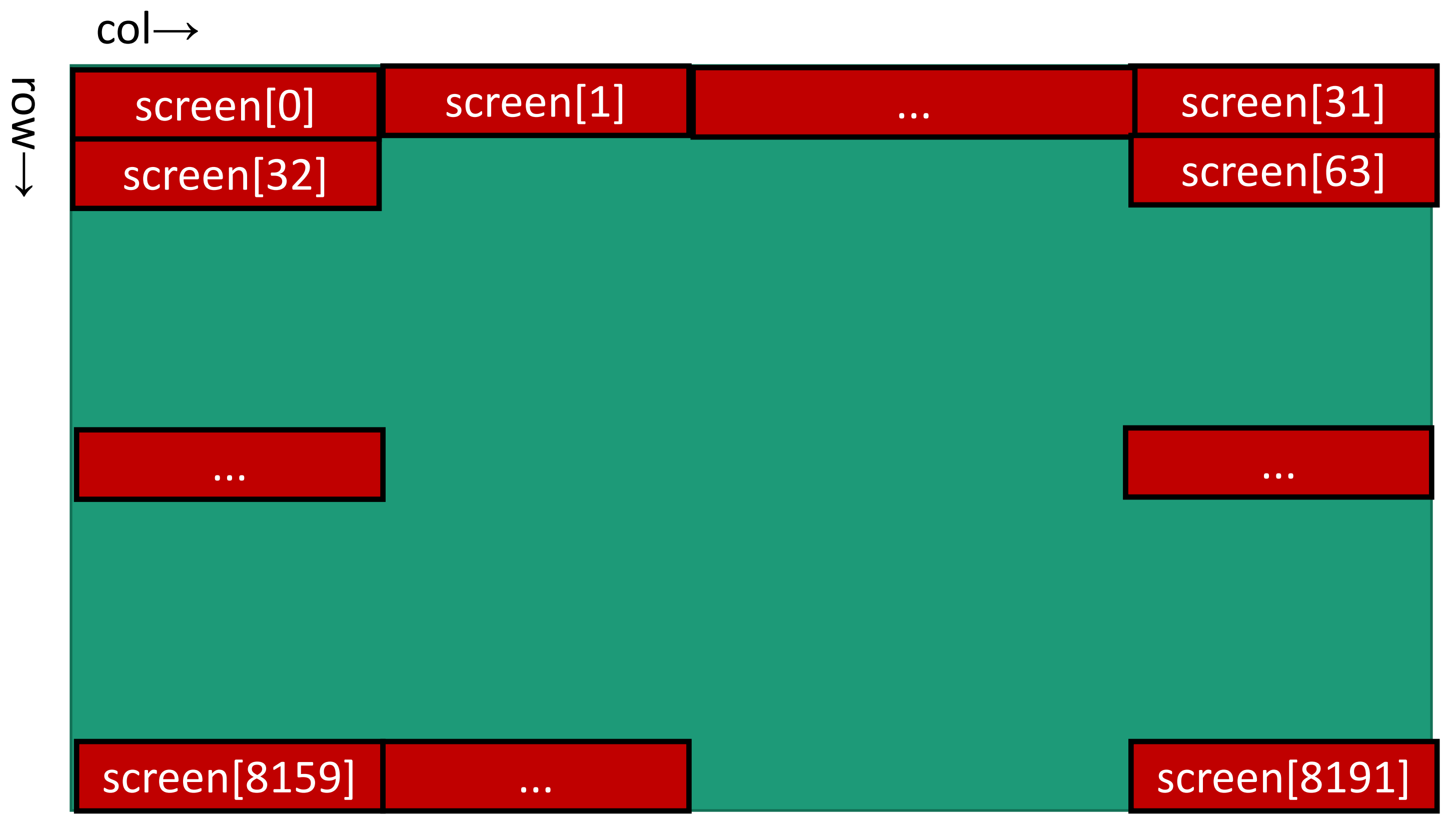
511

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

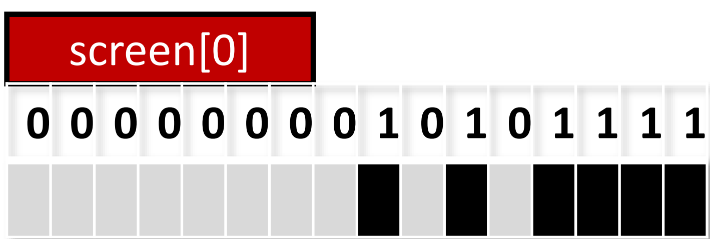
Hack Screen

512-bit wide

256-bit high



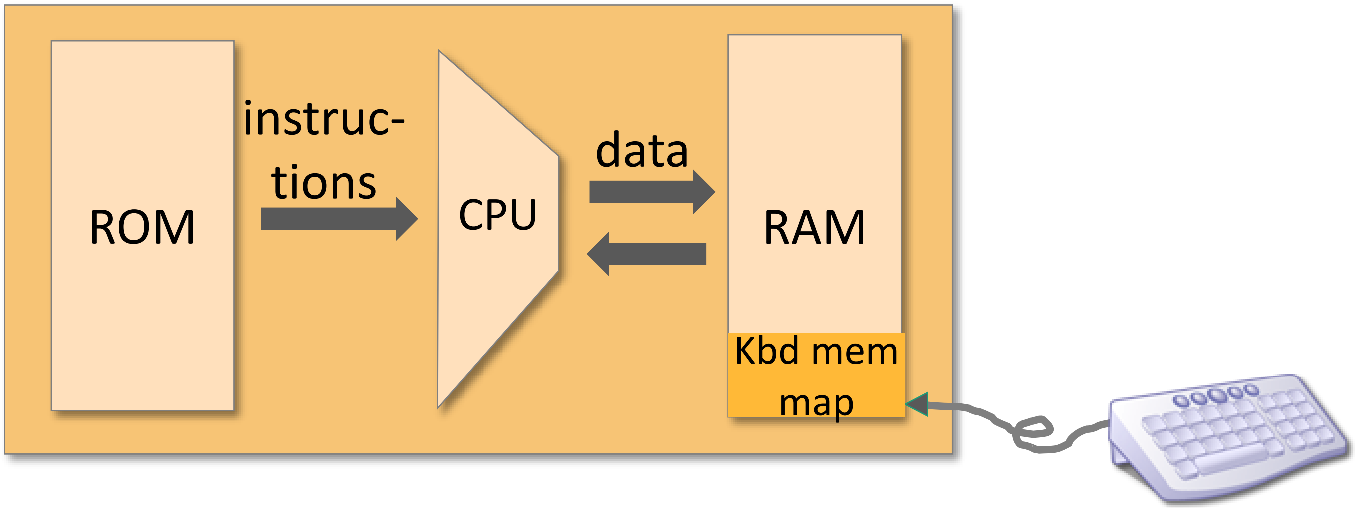
screen[0] = 1111010100000000



1 for black
0 for white

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

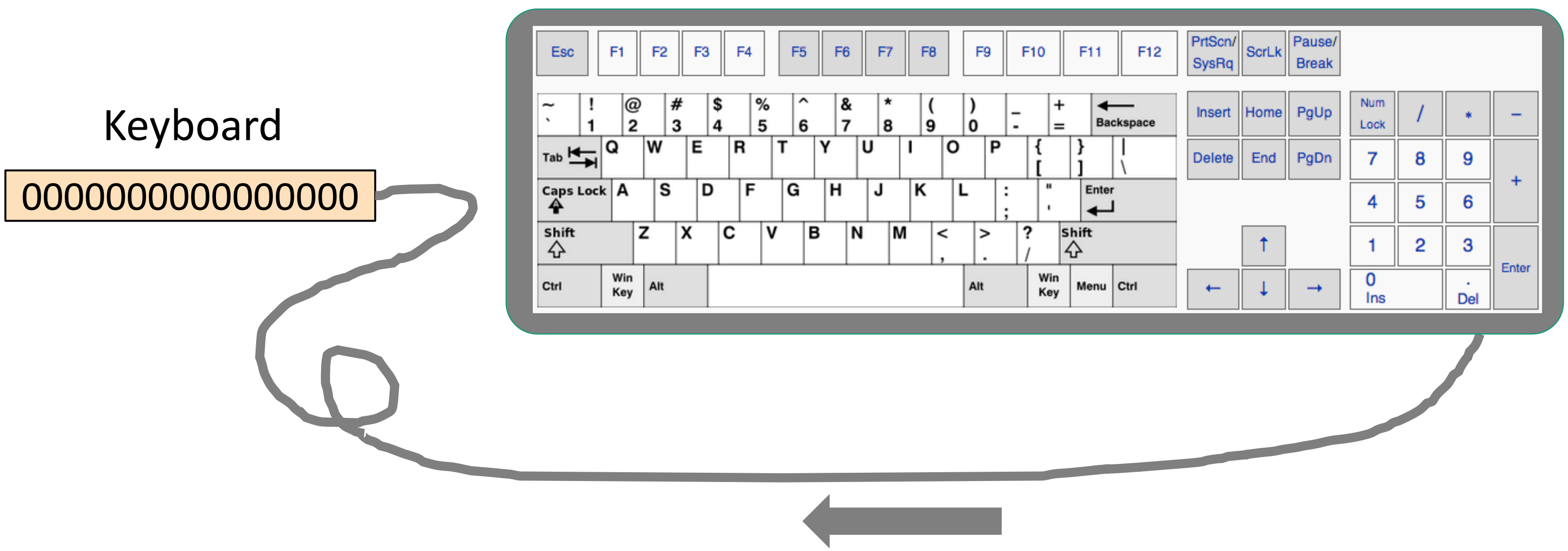
Input



Keyboard: used to enter inputs

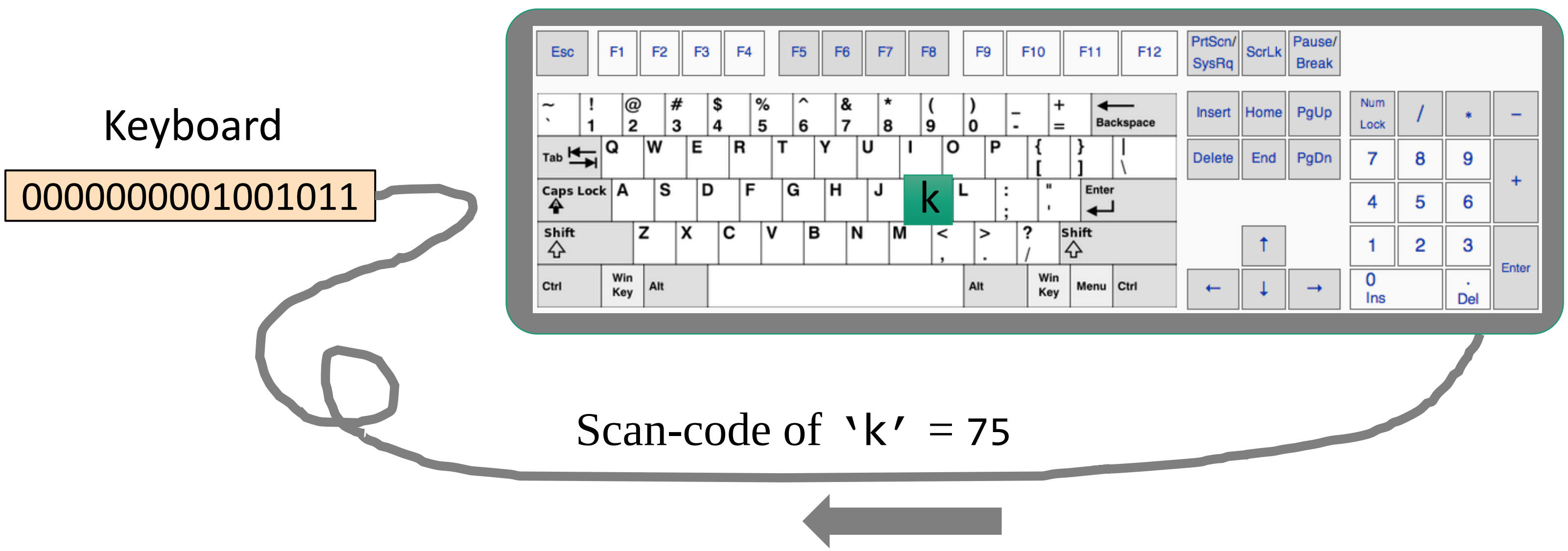
The physical keyboard is associated with a *keyboard memory map*.

Memory mapped input



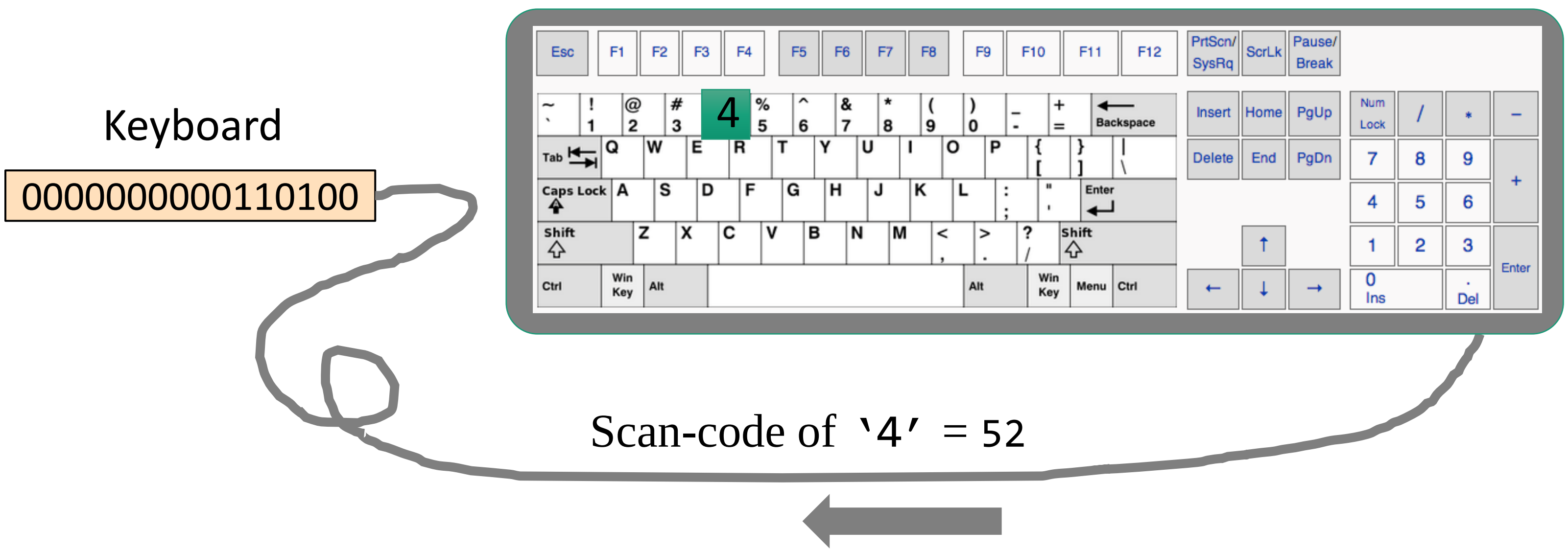
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



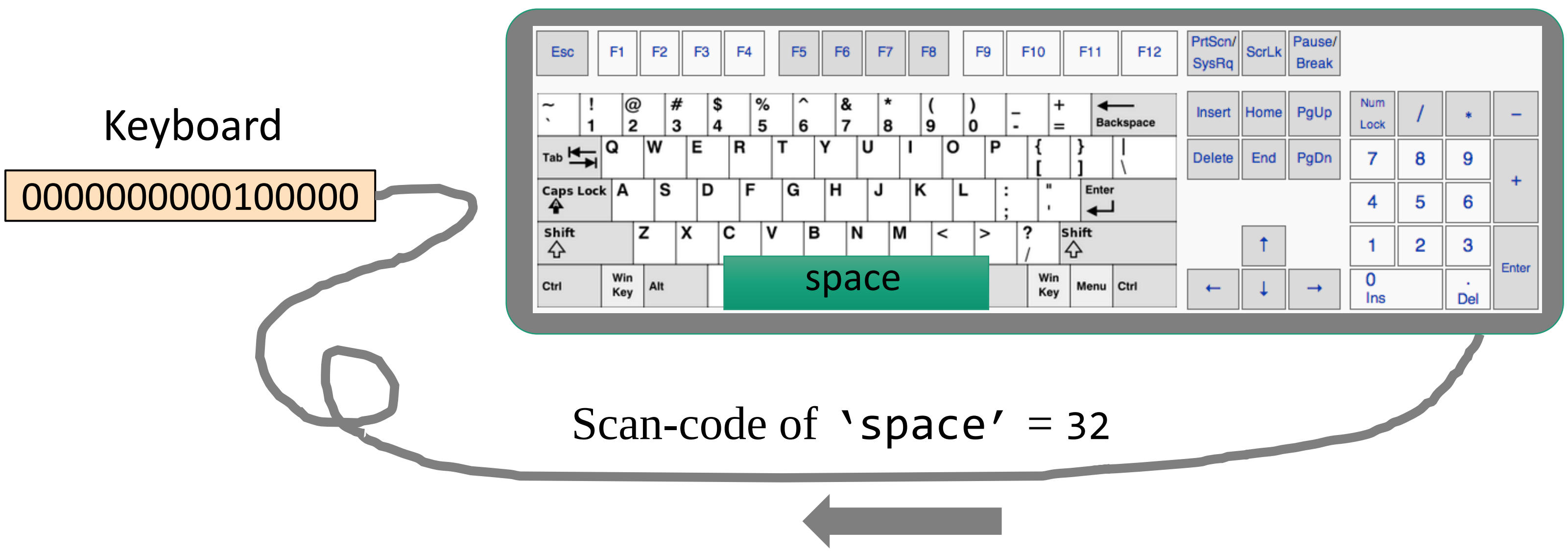
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

Memory mapped input



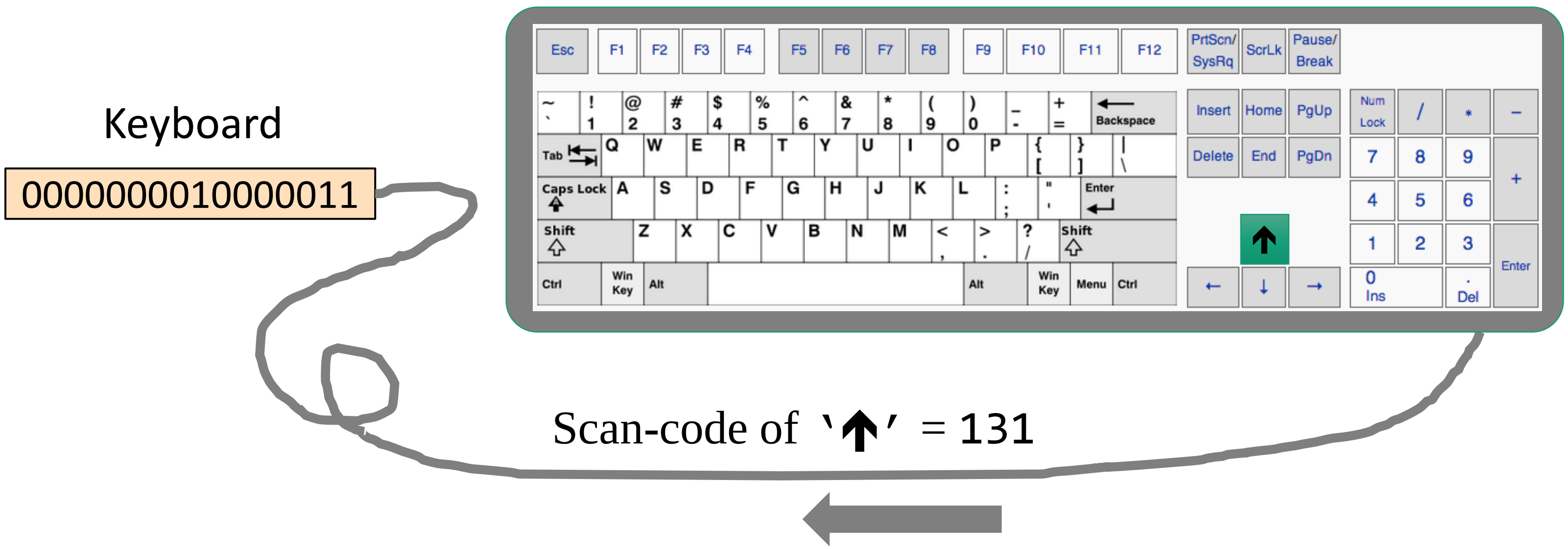
- When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map.*

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

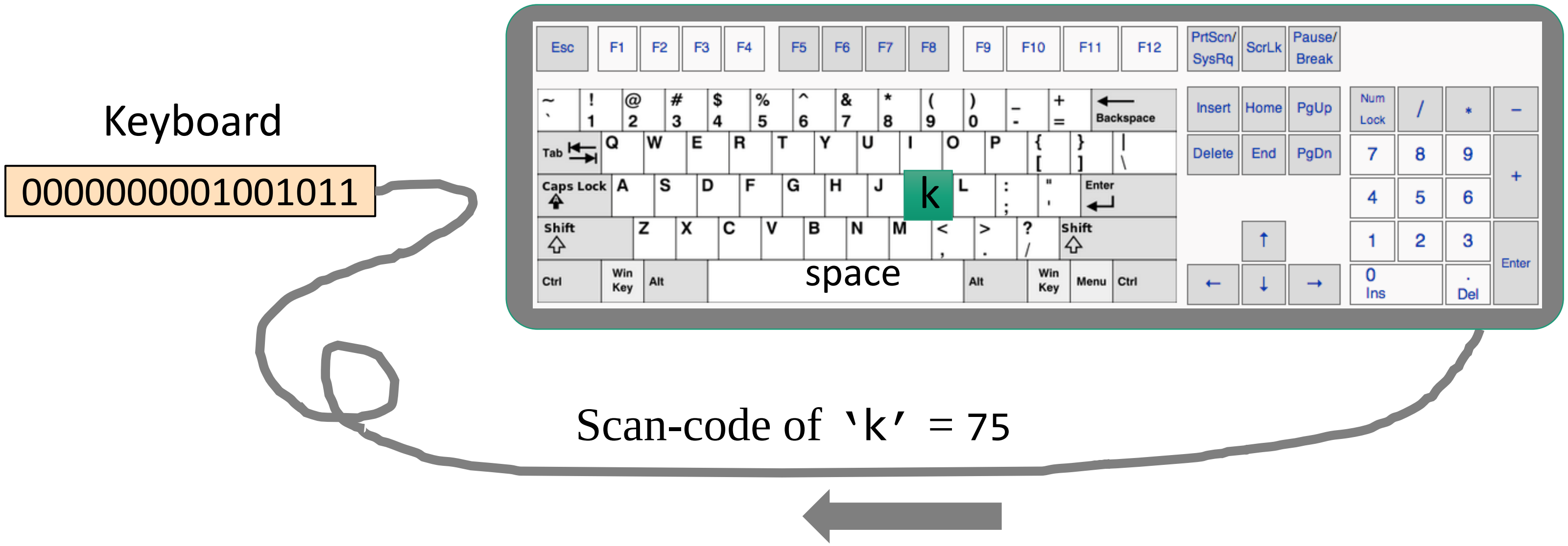
[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

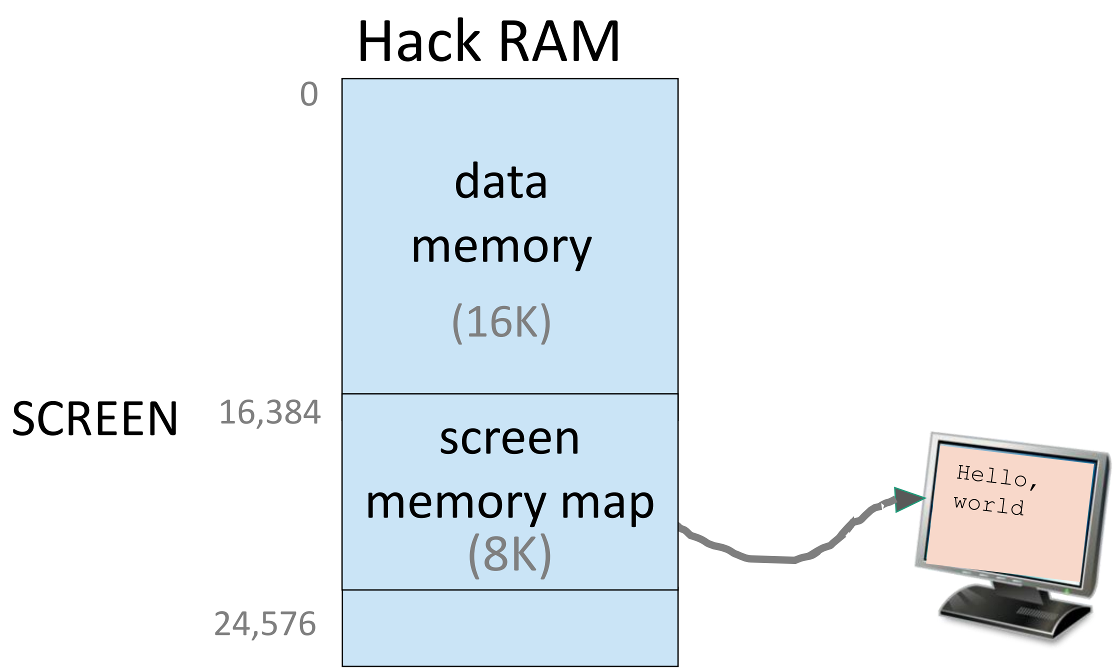
key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
…	…
f12	152

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]**.

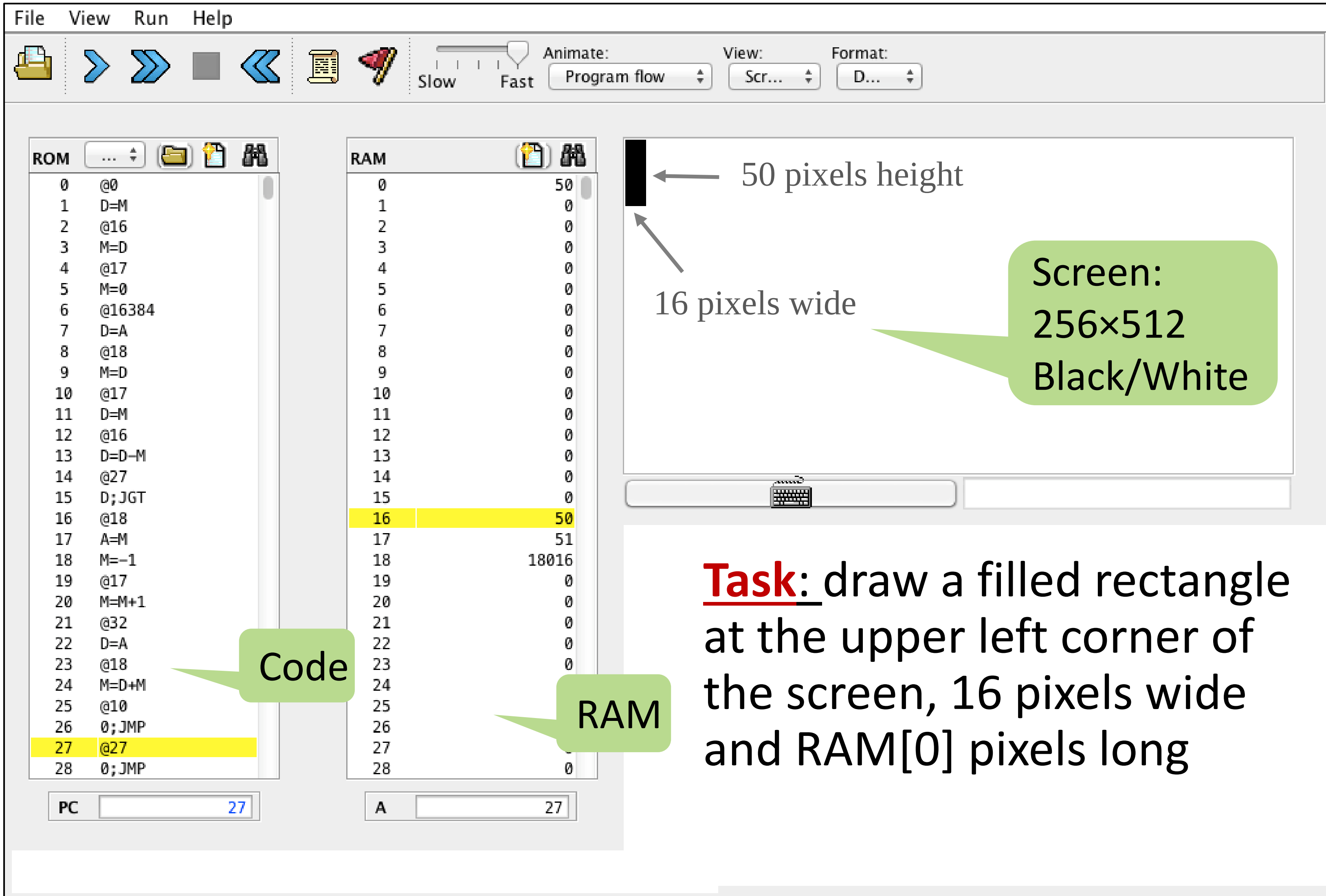
Output



<u>Hack language convention:</u>

- SCREEN: base address of the screen memory map

Handling the screen (example)

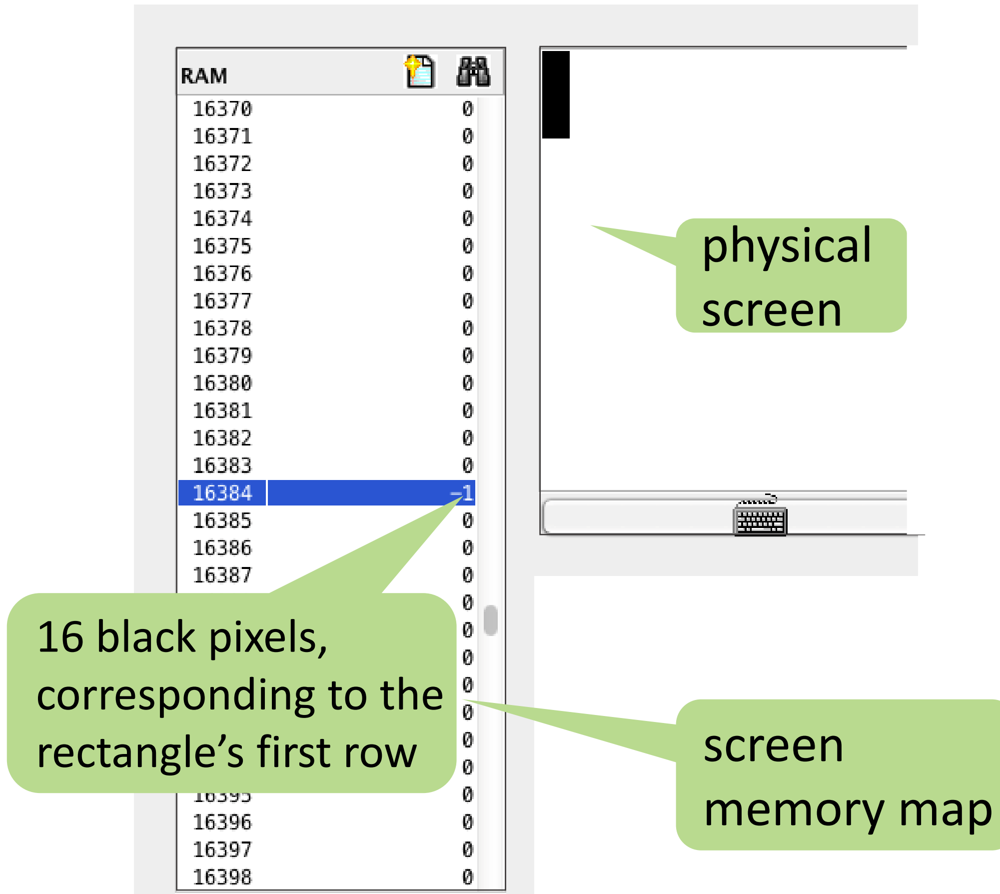


Task: draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and RAM[0] pixels long

Handling the screen (example)

Pseudo code

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

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

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

END:
  goto END
```



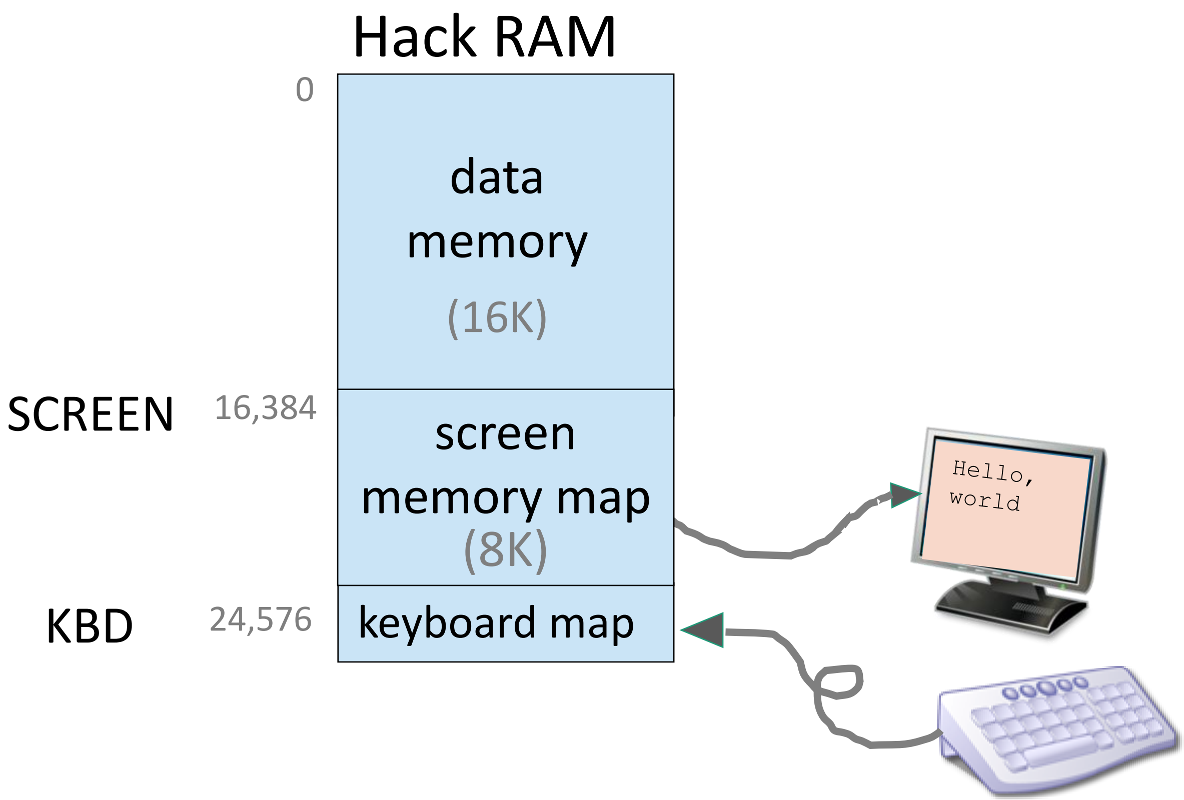
Handling the screen (example)

Assembly code

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

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

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

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

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

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

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

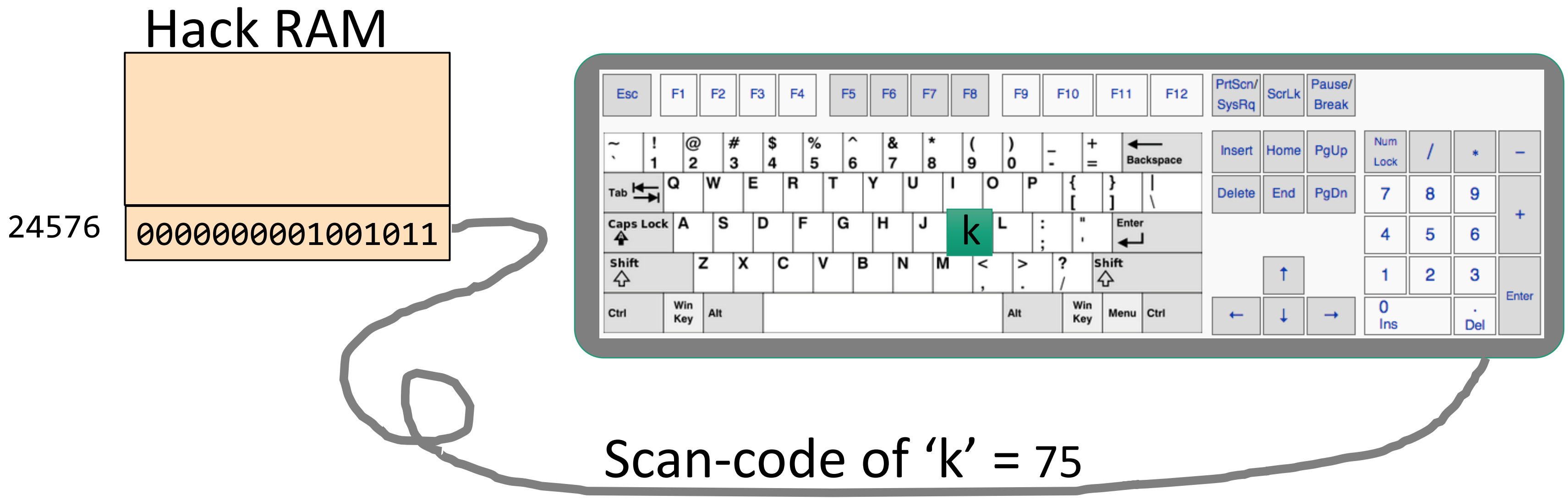
Input



<u>Hack language convention:</u>

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

Handle the keyboard



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

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

Keyboard input (example)

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

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

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

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

Comments on assembly programming

High level code

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

Compiler

Machine language

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

<u>Assembly programming is:</u>

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

Summary

- Hack assembly programming
  - Registers and memory
  - Branching, variables, iteration
  - Pointers

- Hack input / output

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.