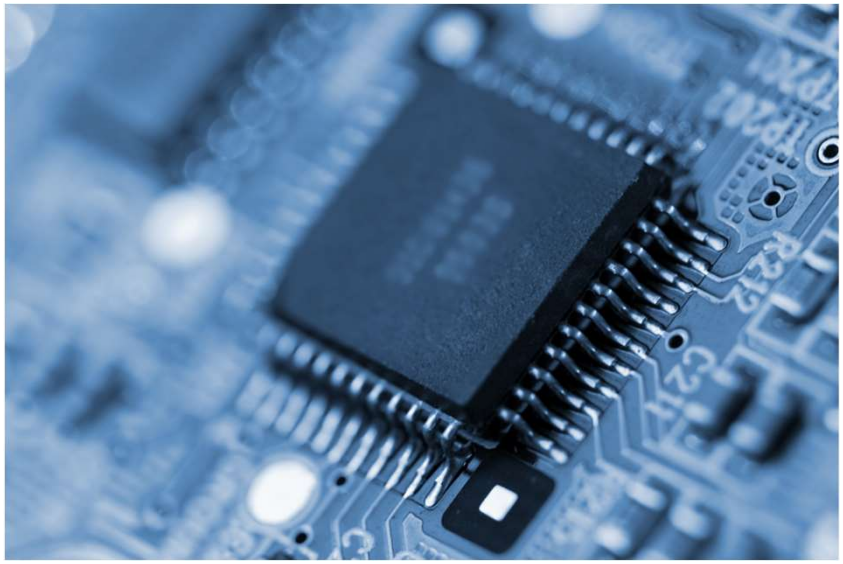
Computer Processors

Assembly Programming



Chapter 4: Machine Language

Overview

- Machine languages
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Low Level Programming

- Basic
- Iteration
- Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Program example 1: Add

Task: R2 ← R0 + R1 + 17

Add.asm

```
// Sets R2 to R0 + R1 + 17
```

Program example 1: Add

Task: R2 ← R0 + R1 + 17

Add.asm

```
// Sets R2 to R0 + R1 + 17
// D = R0
@R0
D=M
// D = D + R1
@R1
D=D+M
// D = D + 17
@17
D=D+A
// R2 = D
@R2
M=D
```

Program example 2: `Signum`

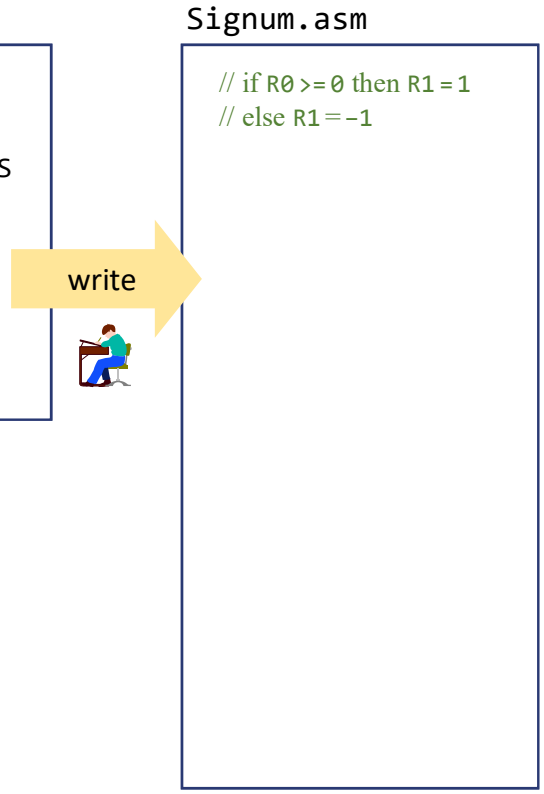
Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
```

Program example 2: Signum

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```



Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
```

Program example 2: Signum

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

write

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```

Best practice

When writing a (non-trivial) assembly program, always start with writing pseudocode.

Then translate the pseudo instructions into assembly, line by line.

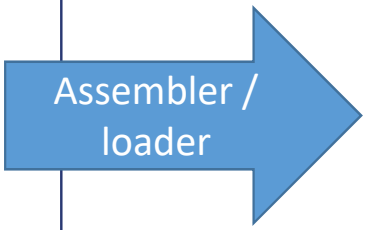
Program example 2: Signum

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```



(Note how the assembler mapped all the symbols on physical addresses)

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	
11	
12	
13	
14	
...	

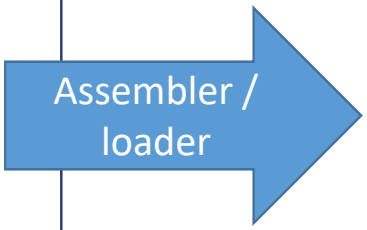
Watch out: Security breach

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```



Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	
11	
12	
13	
14	
...	

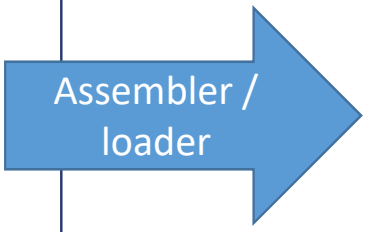
Watch out: Security breach

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```



Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	0111111000111110
11	1010101001011110
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

The memory is never empty

Watch out: Security breach

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```

Program execution:

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	0111111000111110
11	1010101001011110
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Watch out: Security breach

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```

Program execution:

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	0111111000111110
11	1010101001011110
12	Malicious
13	Code
14	0101011100110111
...	

Watch out: Security breach

Pseudocode

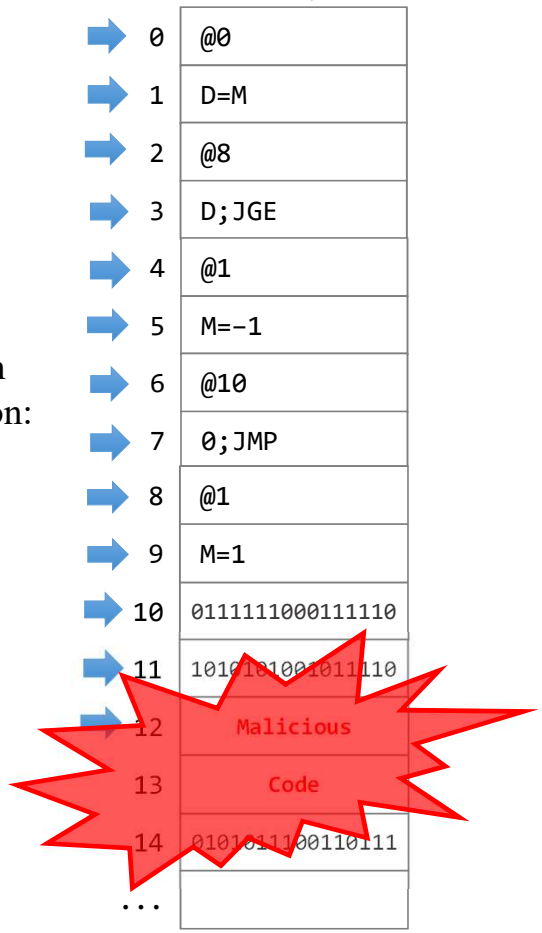
```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```

Program execution:

Memory



Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
```

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Infinite loop

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
    // if R0 >= 0 goto POS
    @R0
    D=M
    @POS
    D;JGE
    // R1 = -1
    @R1
    M=-1
    // goto END
    @END
    0;JMP
(POS)
    // R1 = 1
    @R1
    M=1
(END)
    @END
    0;JMP
```

Assembler / loader

Infinite loop

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Program execution:

Memory

➡	0	@0
➡	1	@D=M
➡	2	@8
➡	3	@D;JGE
➡	4	@1
➡	5	@M=-1
➡	6	@10
➡	7	@0;JMP
	8	@1
	9	@M=1
➡	10	@10
➡	11	0;JMP
	12	0100100110011011
	13	1110010011111111
	14	0101011100110111
	...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Program execution:

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Program execution:

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Program execution:

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Program execution:

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Program execution:

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Program execution:

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
   if (R0 ≥ 0) goto POS
   R1 = -1
   goto END
POS:
   R1 = 1
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1
// else R1 = -1
      // if R0 >= 0 goto POS
      @R0
      D=M
      @POS
      D;JGE
      // R1 = -1
      @R1
      M=-1
      // goto END
      @END
      0;JMP
(POS)
      // R1 = 1
      @R1
      M=1
(END)
      @END
      0;JMP
```

Memory

0	@0
1	@D=M
2	@8
3	@D;JGE
4	@1
5	@M=-1
6	@10
7	@0;JMP
8	@1
9	@M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Best practice:

Terminate every assembly program with an infinite loop.

Program example 3: Max

Task: Set R0 to *max*(R1, R2)

Examples: *max*(5,3) = 5, *max*(2,7) = 7, *max*(4,4) = 4

Pseudocode

```
// if (R1 > R2) then R0 = R1
// else            R0 = R2
...
```

Program example 3: Max

Task: Set R0 to *max*(R1, R2)

Examples: *max*(5,3) = 5, *max*(2,7) = 7, *max*(4,4) = 4

Pseudocode

```
// if (R1 > R2) then R0 = R1
// else             R0 = R2
...
```

Max2.asm

```
// You do it
```

- Write the pseudocode
- Translate and write the assembly code in a text file named Max2.asm
- Load Max2.asm into the CPU emulator
- Put some values in R1 and R2
- Run the program, one instruction at a time
- Make sure that the program puts the correct value in R0.

Chapter 4: Machine Language

Overview

- Machine languages
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Low Level Programming


Basic


Iteration

- Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Iterative processing

Example: Compute $1 + 2 + 3 + ... + N$

Pseudocode

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
```

Iterative processing

Example: Compute $1 + 2 + 3 + ... + N$

Pseudocode

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   i = 1
   sum = 0
LOOP:
   if (i > R0) goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

Iterative processing

Example: Compute $1 + 2 + 3 + ... + N$

Pseudocode

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   i = 1
   sum = 0
LOOP:
   if (i > R0) goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

Hack assembly

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
```

Iterative processing

Example: Compute $1 + 2 + 3 + ... + N$

Pseudocode

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   i = 1
   sum = 0
LOOP:
   if (i > R0) goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

Hack assembly

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   // i = 1
   @i
   M=1
   // sum = 0
   @sum
   M=0
(LOOP)
   // if (i > R0) goto STOP
   @i
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   // sum = sum + i
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D
   // i = i + 1
   @i
   M=M+1
   // goto LOOP
   @LOOP
   0;JMP
```

(code continues here)

Iterative processing

Example: Compute $1 + 2 + 3 + ... + N$

Pseudocode

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   i = 1
   sum = 0
LOOP:
   if (i > R0) goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

Hack assembly

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   // i = 1
   @i
   M=1
   // sum = 0
   @sum
   M=0
(LOOP)
   // if (i > R0) goto STOP
   @i
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   // sum = sum + i
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D
   // i = i + 1
   @i
   M=M+1
   // goto LOOP
   @LOOP
   0;JMP
```

(code continues here)

```
(STOP)
   // R1 = sum
   @sum
   D=M
   @R1
   M=D
   // infinite loop
(END)
   @END
   0;JMP
```

Chapter 4: Machine Language

Overview

- Machine languages
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Low Level Programming

- ✓ Basic
- ✓ Iteration
- ➡ Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

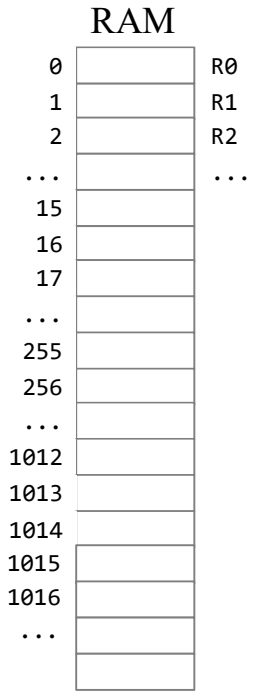
- Usage
- Specification
- Output
- Input
- Project 4

Pointer-based processing

Example 1: Set the register at address *addr* to -1

Input: `R0`: Holds *addr*

```
// Sets RAM[R0] to -1
// Usage: Put some non-negative value in R0
```

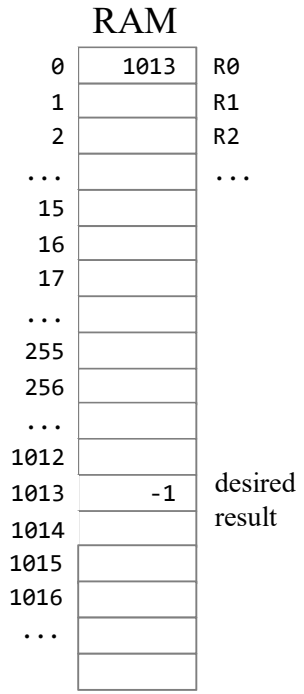


Pointer-based processing

Example 1: Set the register at address *addr* to -1

Input: `R0`: Holds *addr*

```
// Sets RAM[R0] to -1
// Usage: Put some non-negative value in R0
```



Pointer-based processing

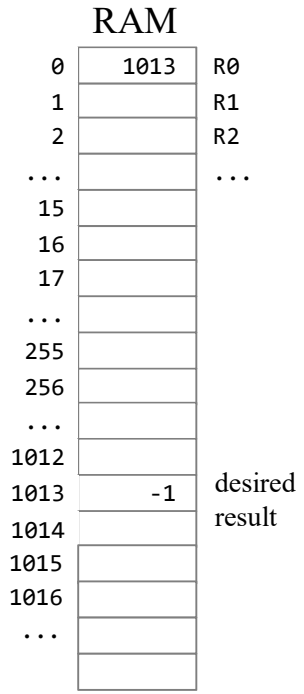
Example 1: Set the register at address *addr* to -1

Input: `R0`: Holds *addr*

```
// Sets RAM[R0] to -1
// Usage: Put some non-negative value in R0
@R0
A=M
M=-1
```

The key instruction:

In the Hack machine language, pointer-based processing is realized by setting the address register to the address that we want to access, using the instruction `A=...`



Pointer-based processing

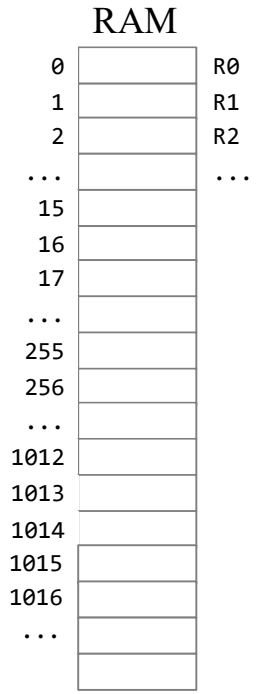
Example 1: Set the register at address *addr* to –1

Input: `R0`: Holds *addr*

```
// Sets RAM[R0] to -1
// Usage: Put some non-negative value in R0
@R0
A=M
M=-1
```

Example 2:

```
// Sets RAM[R0] to R1
// Usage: Put some non-negative value in R0,
//        and some value in R1.
```



Pointer-based processing

Example 1: Set the register at address *addr* to –1

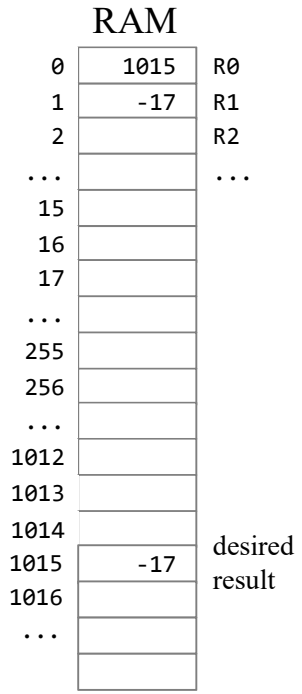
Input: `R0`: Holds *addr*

```
// Sets RAM[R0] to -1
// Usage: Put some non-negative value in R0
@R0
A=M
M=-1
```

Example 2:

```
// Sets RAM[R0] to R1
// Usage: Put some non-negative value in R0,
//        and some value in R1.
```

?



Pointer-based processing

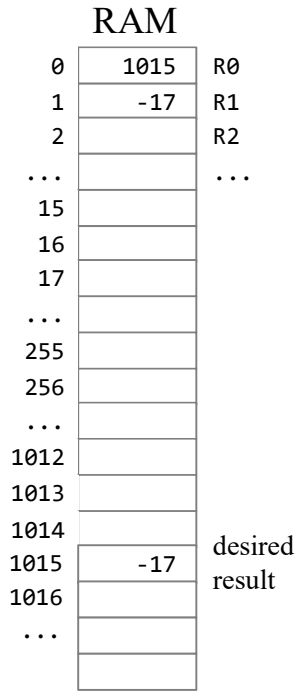
Example 1: Set the register at address *addr* to –1

Input: `R0`: Holds *addr*

```
// Sets RAM[R0] to -1
// Usage: Put some non-negative value in R0
@R0
A=M
M=-1
```

Example 2:

```
// Sets RAM[R0] to R1
// Usage: Put some non-negative value in R0,
//        and some value in R1.
@R1
D=M
@R0
A=M
M=D
```

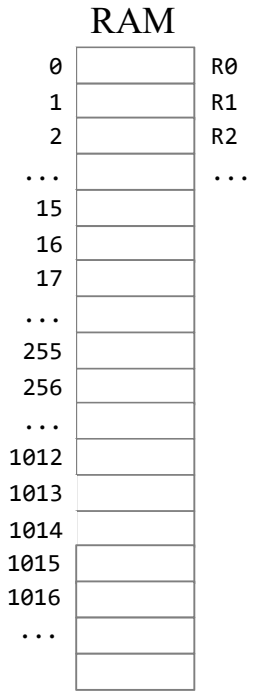


Pointer-based processing

Example 3: Get the value of the register at *addr*

Input: `R0`: Holds *addr*

```
// Gets R1 = RAM[R0]
// Usage: Put some non-negative value in R0
```



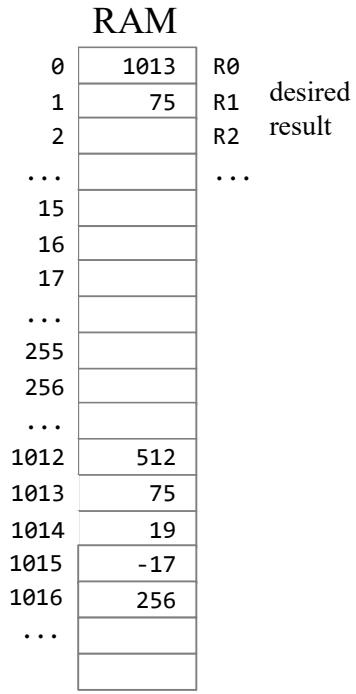
Pointer-based processing

Example 3: Get the value of the register at *addr*

Input: `R0`: Holds *addr*

```
// Gets R1 = RAM[R0]
// Usage: Put some non-negative value in R0

?
```

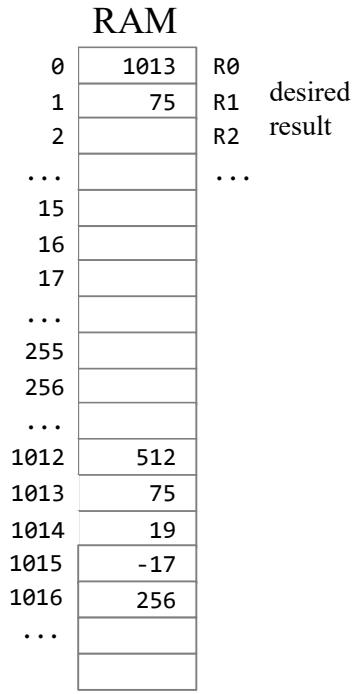


Pointer-based processing

Example 3: Get the value of the register at *addr*

Input: `R0`: Holds *addr*

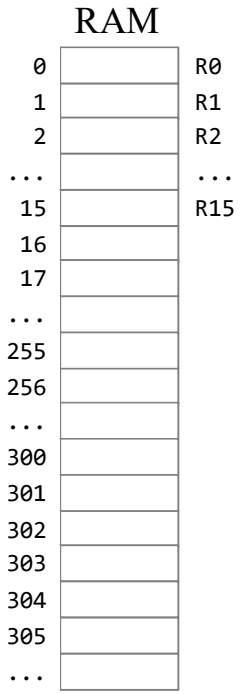
```
// Gets R1 = RAM[R0]
// Usage: Put some non-negative value in R0
@R0
A=M
D=M
@R1
M=D
```



Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to -1

Inputs: R0: *base*
R1: *n*



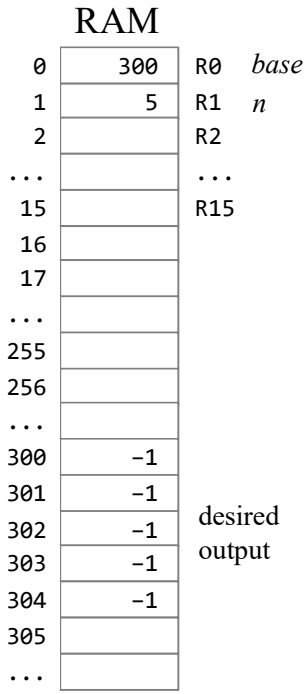
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



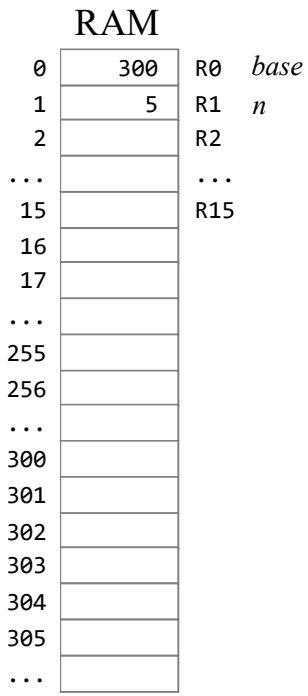
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



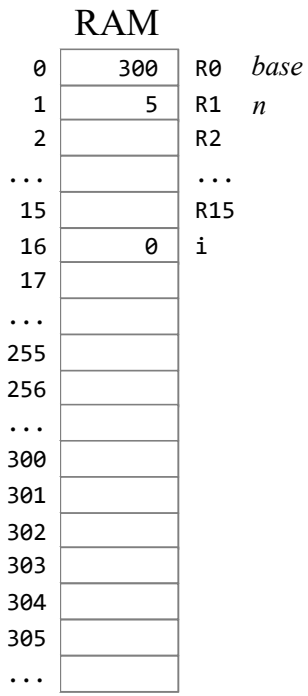
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



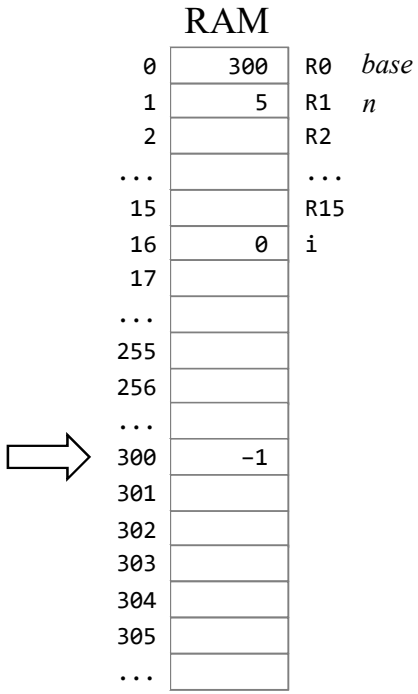
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



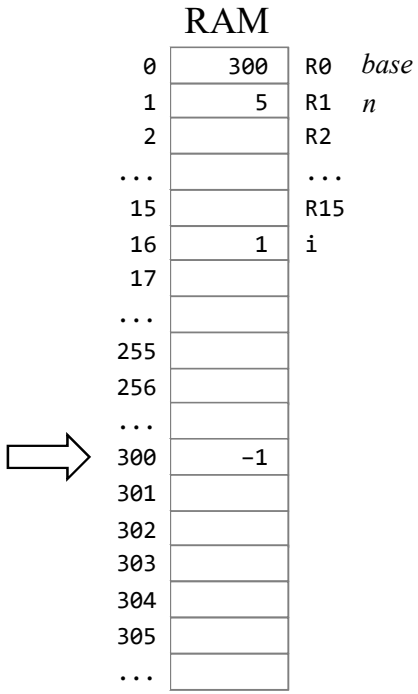
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



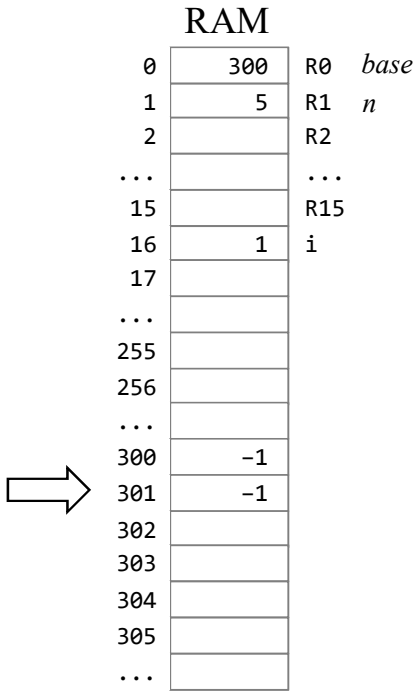
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



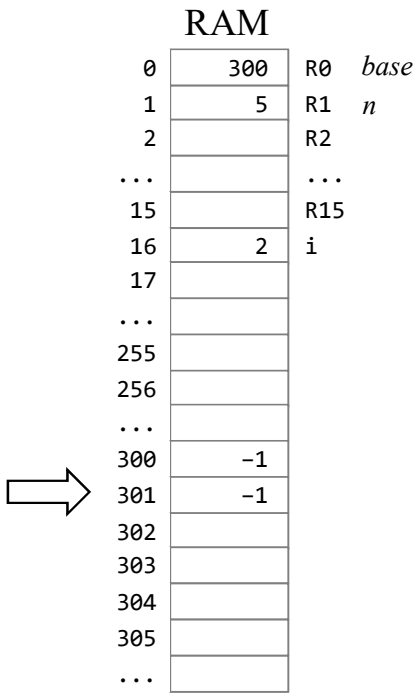
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



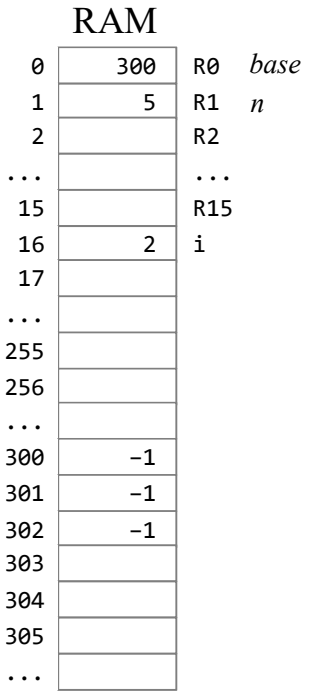
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



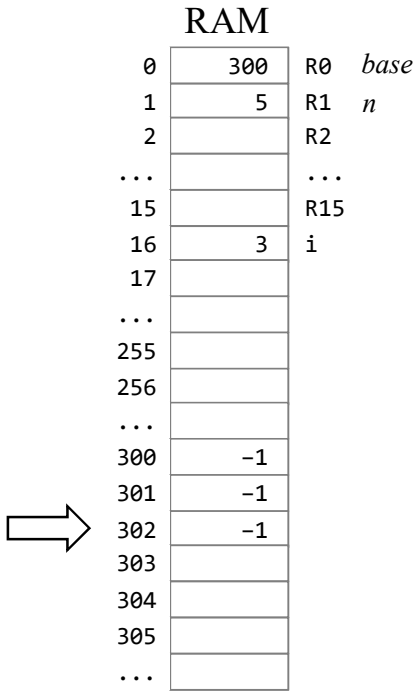
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



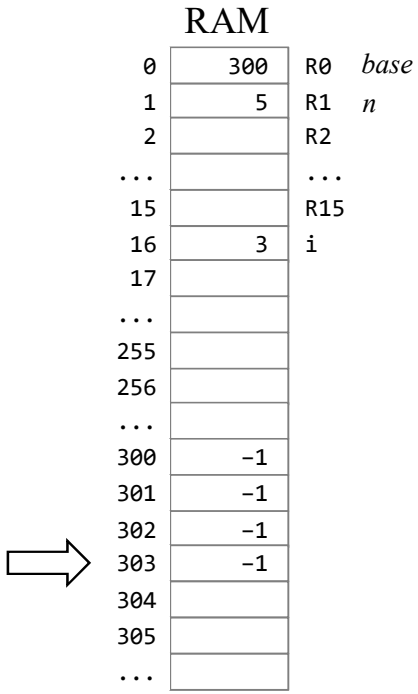
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



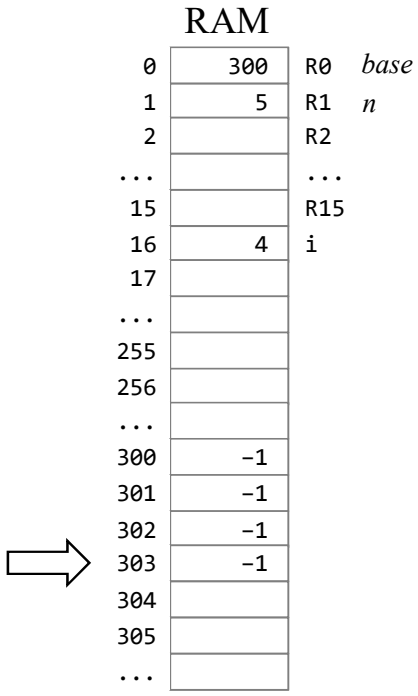
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



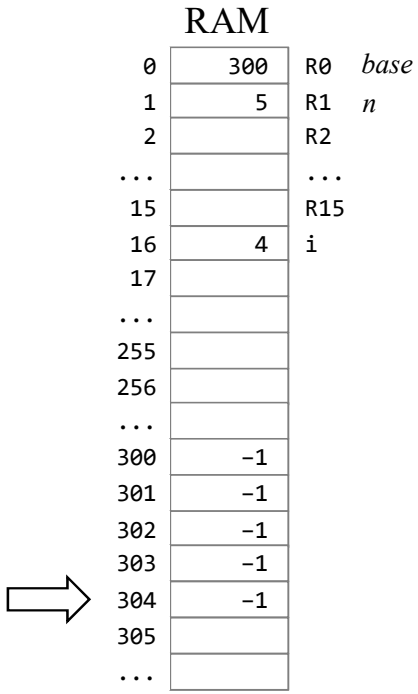
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



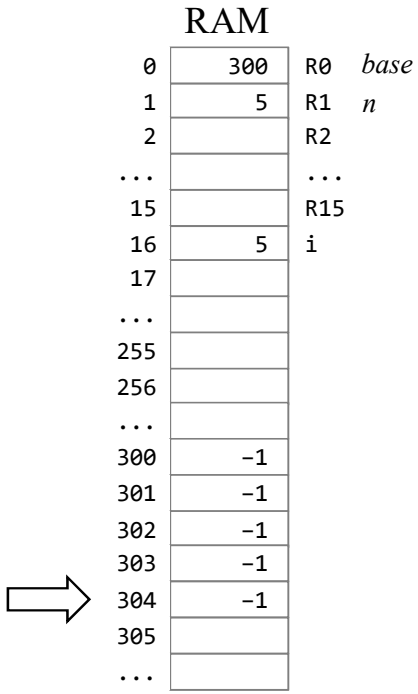
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



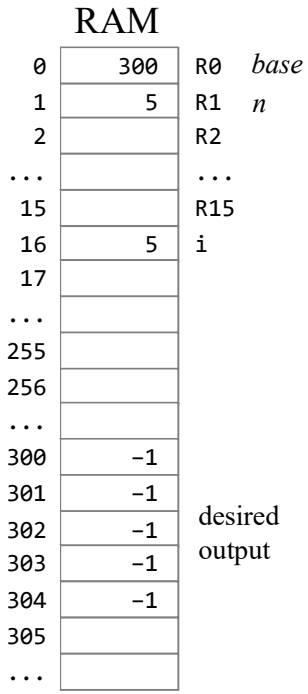
Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

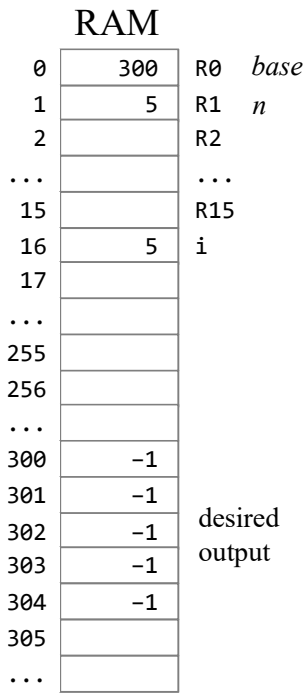
Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
...
// RAM[R0 + i] = -1
```

The key operation

?



Pointer-based processing

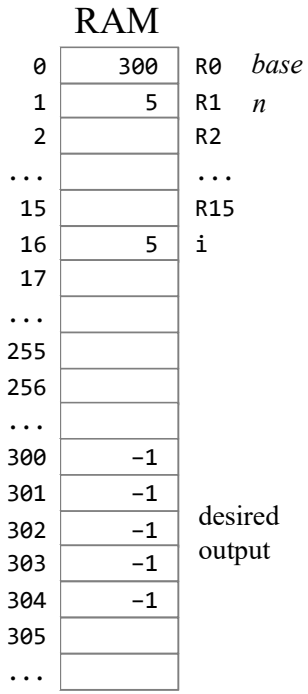
Example 4: Set the first *n* entries of the memory block beginning in address *base* to -1

Inputs: R0: *base*
R1: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
...
// RAM[R0 + i] = -1
@R0
D=M
@i
A=D+M
M=-1
...
```

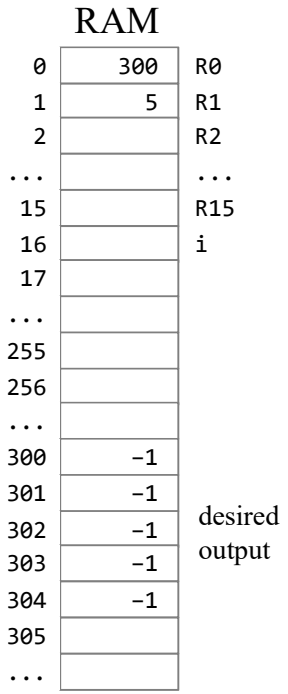
The key operation



Pointer-based processing

Pseudocode

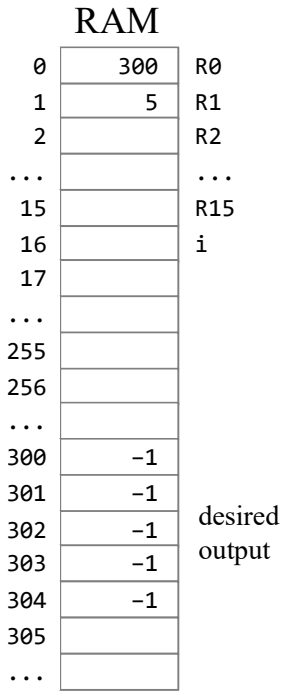
```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
```



Pointer-based processing

Pseudocode

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
   i=0
LOOP:
   if (i==R1) goto END
   RAM[R0+i] = -1
   i = i+1
   goto LOOP
END:
```



Pointer-based processing

Pseudocode

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
   i=0
LOOP:
   if (i==R1) goto END
   RAM[R0+i] = -1
   i = i+1
   goto LOOP
END:
```

Assembly code

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1

?
```

RAM

0	300	R0
1	5	R1
2		R2
...		...
15		R15
16		i
17		
...		
255		
256		
...		
300	-1	
301	-1	
302	-1	desired output
303	-1	
304	-1	
305		
...		

Pointer-based processing

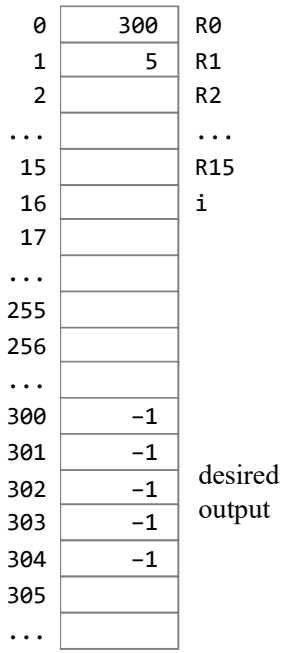
Pseudocode

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
    i=0
LOOP:
    if (i==R1) goto END
    RAM[R0+i] = -1
    i = i+1
    goto LOOP
END:
```

Assembly code

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
      // i = 0
      @i
      M=0
(LOOP)
      // if (i == R1) goto END
      @i
      D=M
      @R1
      D=D-M
      @END
      D;JEQ
      // RAM[R0 + i] = -1
      @R0
      D=M
      @i
      A=D+M
      M=-1
      // i = i + 1
      @i
      M=M+1
      // goto LOOP
      @LOOP
      0;JMP
(END)
      @END
      0;JMP
```

RAM



Array processing

Pseudocode

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
    i=0
LOOP:
    if (i==R1) goto END
    RAM[R0+i] = -1
    i = i+1
    goto LOOP
END:
```

Assembly code

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
      // i = 0
      @i
      M=0
(LOOP)
      // if (i == R1) goto END
      @i
      D=M
      @R1
      D=D-M
      @END
      D;JEQ
      // RAM[R0 + i] = -1
      @R0
      D=M
      @i
      A=D+M
      M=-1
      // i = i + 1
      @i
      M=M+1
      // goto LOOP
      @LOOP
      0;JMP
(END)
      @END
      0;JMP
```

RAM

0	300	R0
1	5	R1
2		R2
...		...
15		R15
16		i
17		
...		
255		
256		
...		
300	-1	
301	-1	
302	-1	desired output
303	-1	
304	-1	
305		
...		

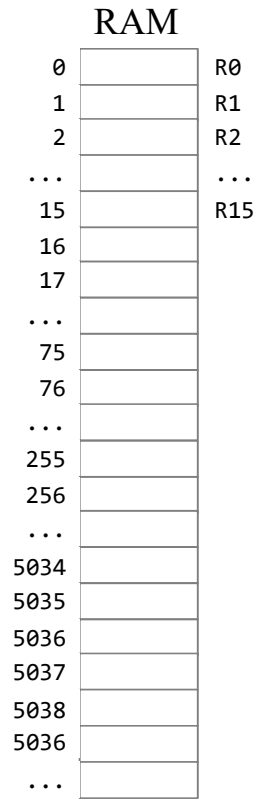
Array processing

Is done similarly, using pointer-based access to the memory block that represents the array.

Array processing

High-level code (e.g. Java)

```
...
// Declares variables
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
// (code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
   sum = sum + arr[j];
}
...
```

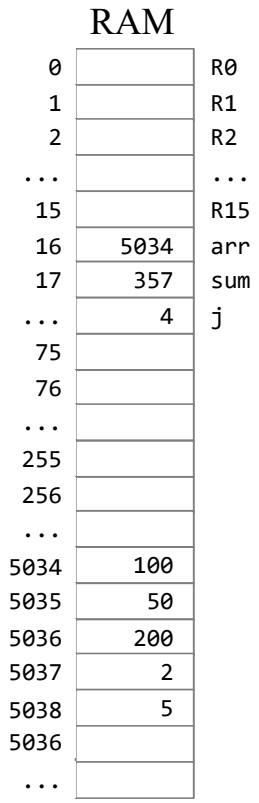


Array processing

High-level code (e.g. Java)

```
...
// Declares variables
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
// (code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
   sum = sum + arr[j];
}
...
```

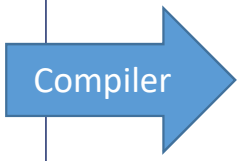
Memory state after executing this code:



Array processing

High-level code (e.g. Java)

```
...
// Declares variables
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
// (code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
   sum = sum + arr[j];
}
...
```



Hack assembly

```
...
```

RAM

0		R0
1		R1
2		R2
...		...
15		R15
16	5034	arr
17	357	sum
...	4	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	5	
5036		
...		

Array processing

High-level code (e.g. Java)

```
...
// Declares variables
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
// (code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
   sum = sum + arr[j];
}
...
```

Compiler

Hack assembly

```
...
// sum = sum + arr[j]
@arr
D=M
@j
A=D+M
D=M
@sum
M=M+D
...
```

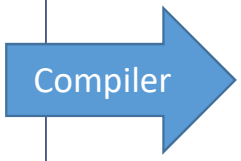
RAM

0		R0
1		R1
2		R2
...		...
15		R15
16	5034	arr
17	357	sum
...	4	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	5	
5036		
...		

Array processing

High-level code (e.g. Java)

```
...
// Declares variables
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
// (code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
   sum = sum + arr[j];
}
...
// Increments each array element
for (int j=0; j<5; j++) {
   arr[j] = arr[j]+1
}
...
```



Hack assembly

```
...
// sum = sum + arr[j]
@arr
D=M
@j
A=D+M
D=M
@sum
M=M+D
...
```

RAM

0		R0
1		R1
2		R2
...		...
15		R15
16	5034	arr
17	357	sum
...	4	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	5	
5036		
...		

Array processing

High-level code (e.g. Java)

```
...
// Declares variables
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
// (code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
   sum = sum + arr[j];
}
...
// Increments each array element
for (int j=0; j<5; j++) {
   arr[j] = arr[j]+1
}
...
```

Compiler

Hack assembly

```
...
// sum = sum + arr[j]
@arr
D=M
@j
A=D+M
D=M
@sum
M=M+D
...
// arr[j] = arr[j] + 1
```

?

RAM

0		R0
1		R1
2		R2
...		...
15		R15
16	5034	arr
17	357	sum
...	4	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	5	
5036		
...		

Array processing

High-level code (e.g. Java)

```
...
// Declares variables
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
// (code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
   sum = sum + arr[j];
}
...
// Increments each array element
for (int j=0; j<5; j++) {
   arr[j] = arr[j]+1
}
...
```

Compiler

Hack assembly

```
...
// sum = sum + arr[j]
@arr
D=M
@j
A=D+M
D=M
@sum
M=M+D
...
// arr[j] = arr[j] + 1
@arr
D=M
@j
A=D+M
M=M+1
...
```

RAM

0		R0
1		R1
2		R2
...		...
15		R15
16	5034	arr
17	357	sum
...	4	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	5	
5036		
...		

Every high-level array access operation involving *arr*[*expression*] can be compiled into Hack code that realizes the operation using the low-level memory access instruction `A =` *arr* + *expression*

Chapter 4: Machine Language

Overview

- Machine languages
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

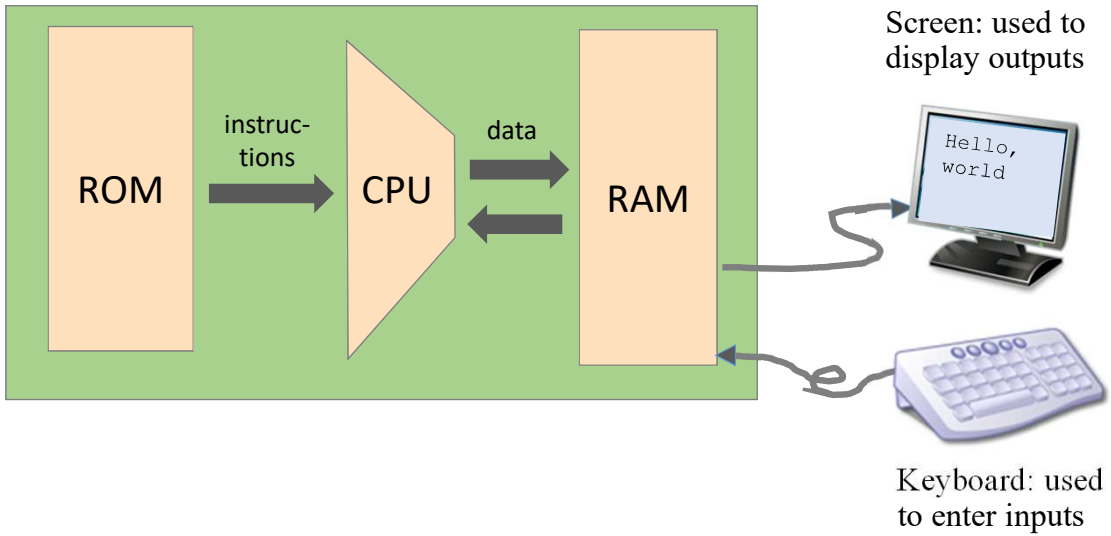
Usage

Specification

Output

- Input
- Project 4

Input / output



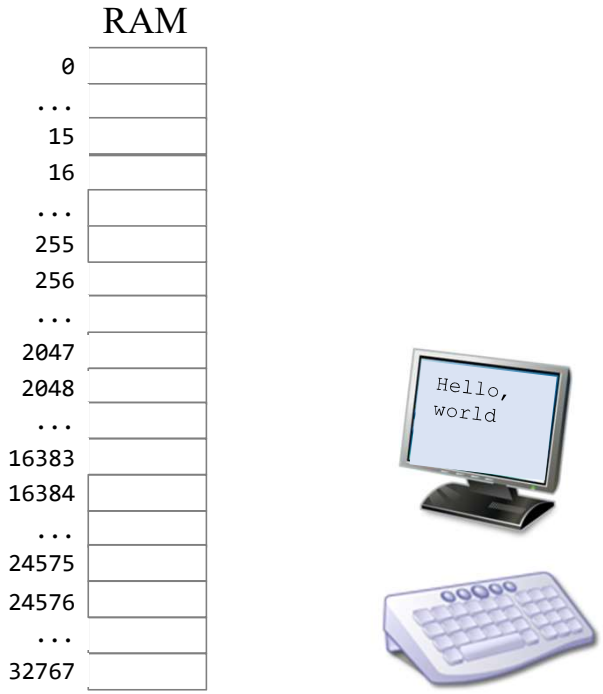
High-level I/O handling:

Software libraries for inputting / outputting text, graphics, audio, video, …

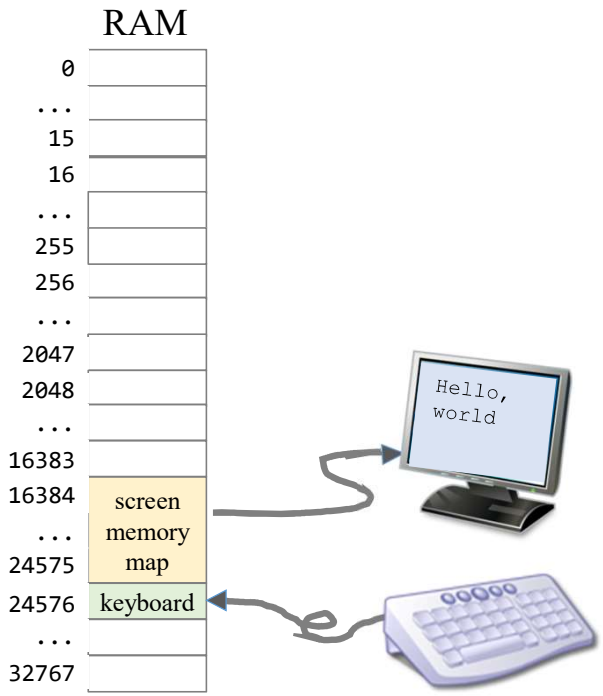
Low-level I/O handling:

Manipulating bits in memory resident *bitmaps*.

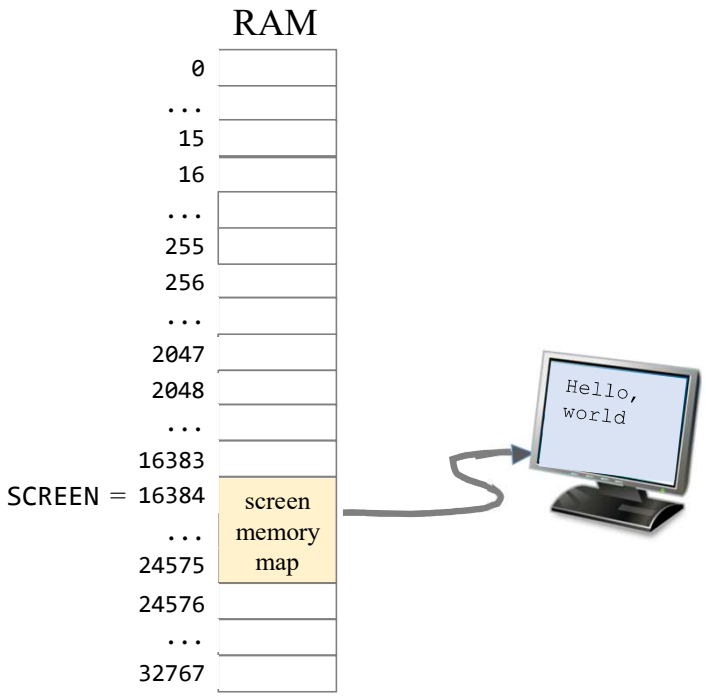
Bitmaps


RAM
0
...
15
16
...
255
256
...
2047
2048
...
16383
16384
...
24575
24576
...
32767
Hello,
world

Bitmaps


RAM
0
...
15
16
...
255
256
...
2047
2048
...
16383
16384
...
24575
24576
...
32767
screen memory map
keyboard
Hello, world

Bitmaps



Screen memory map:

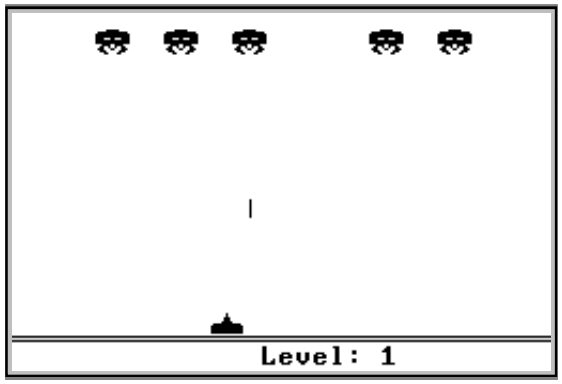
An 8K memory block, dedicated to representing a black-and-white display unit

Base address: `SCREEN` = 16384 (predefined symbol)

Output is effected by writing bits in the screen memory map.

Bitmaps

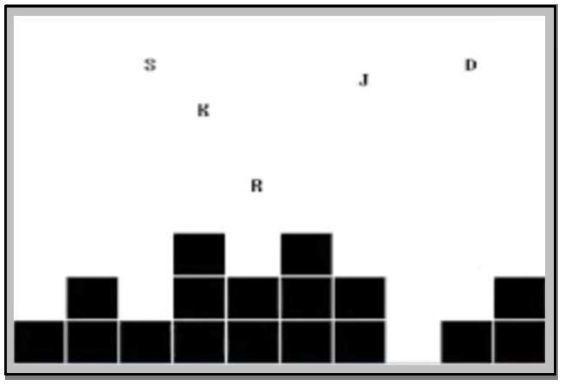
Physical screen



Screen shots of computer games
developed on the Hack computer

Bitmaps

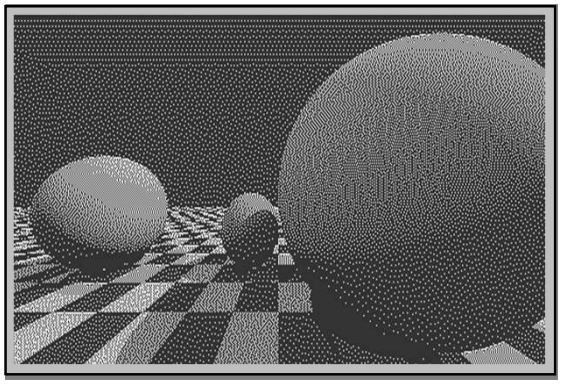
Physical screen



Screen shots of computer games
developed on the Hack computer

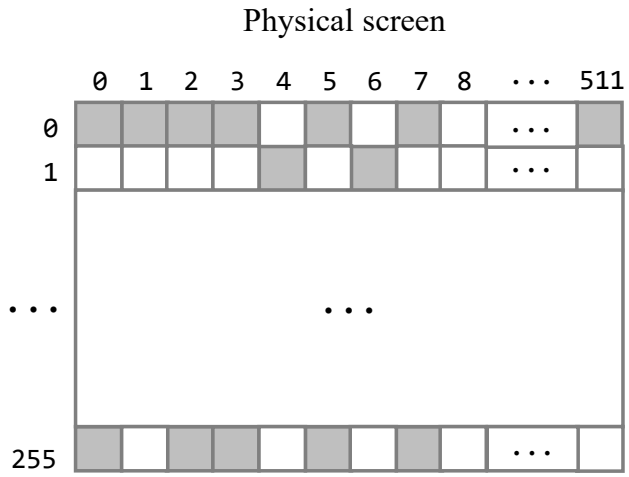
Bitmaps

Physical screen

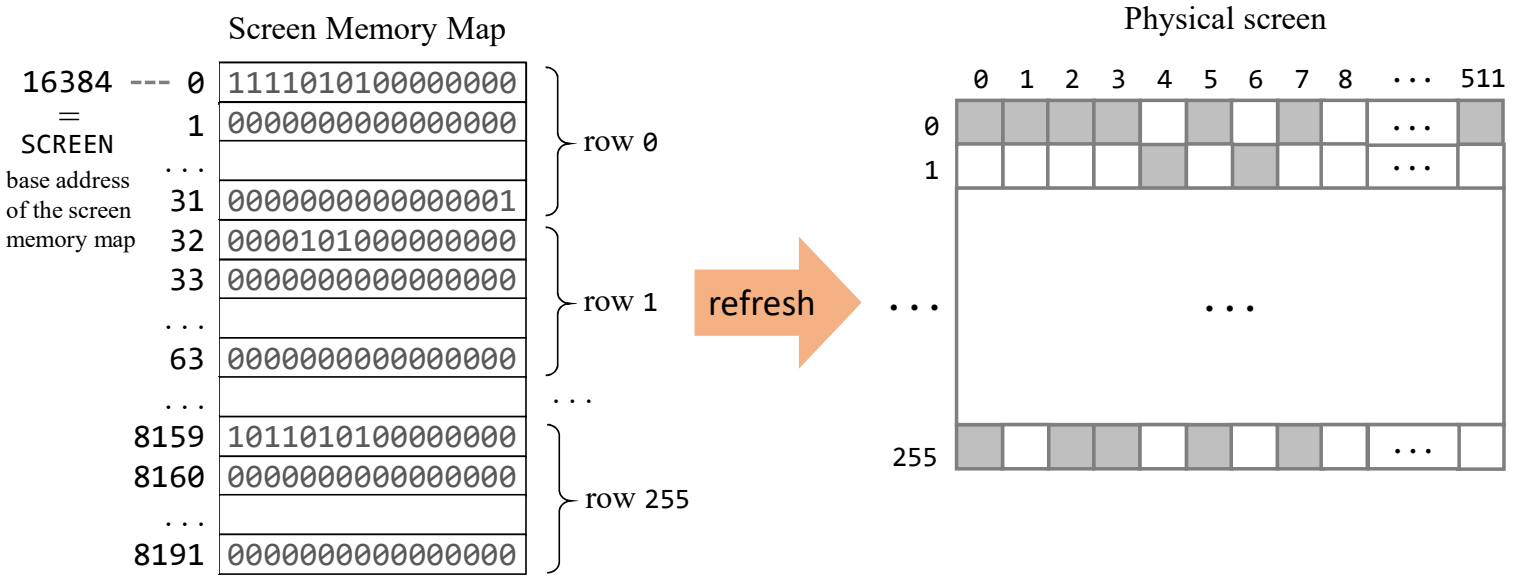


Screen shots of computer games
developed on the Hack computer

Bitmaps



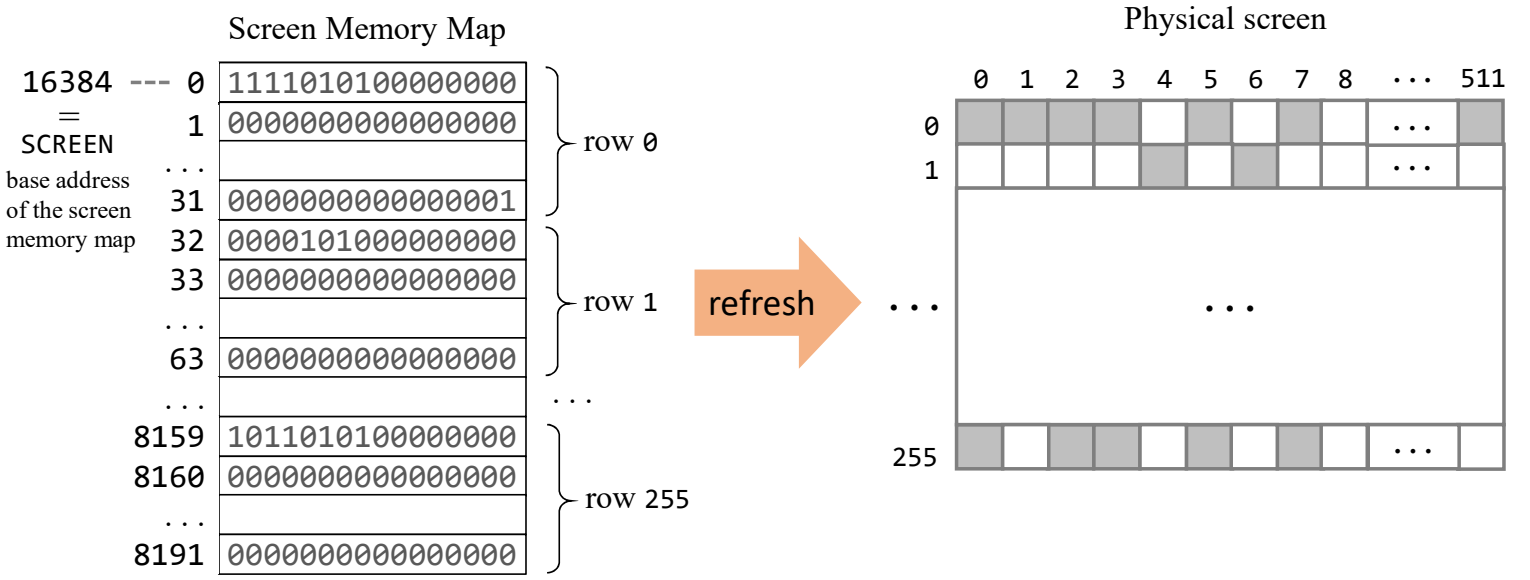
Bitmaps



Mapping:

The pixel in location (*row*, *col*) in the physical screen is represented by the $(col \% 16)th$ bit in RAM address $\text{SCREEN} + 32 * row + col/16$

Bitmaps



To set pixel (*row* , *col*) to black or white:

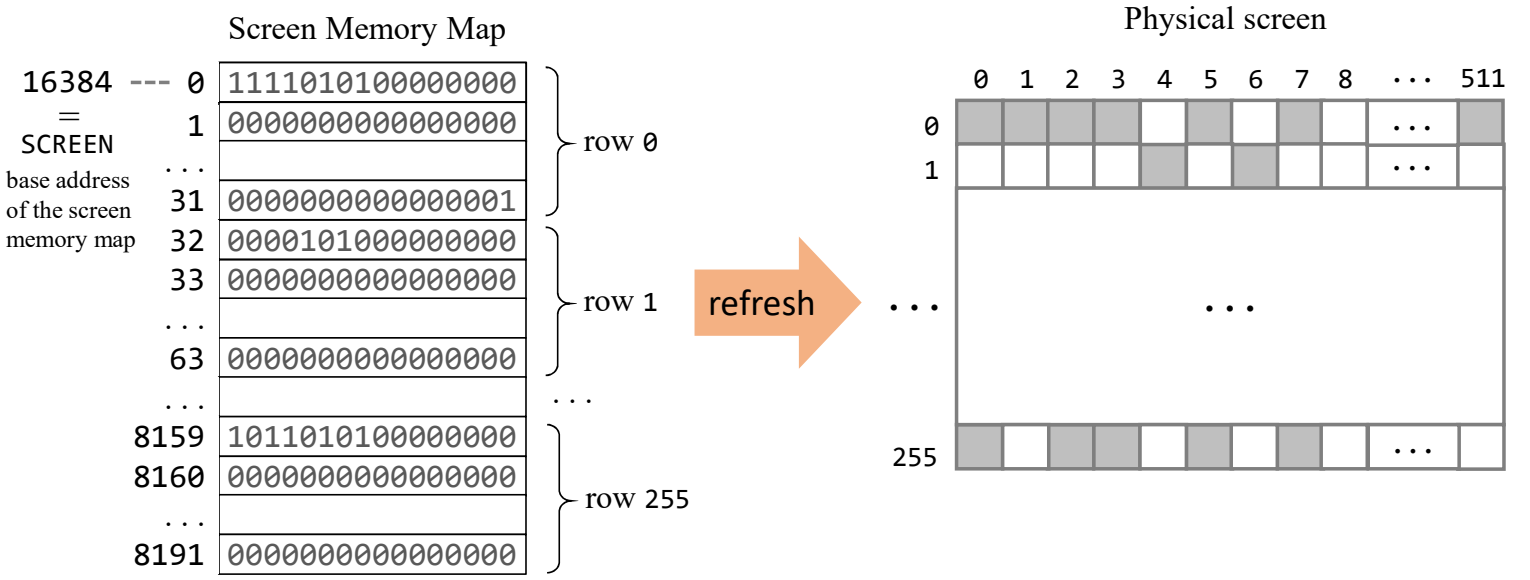
(1) *addr* ← SCREEN + 32**row* + *col*/16

(2) *word* ← RAM[*addr*]

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

(3) RAM[*addr*] ← *word*

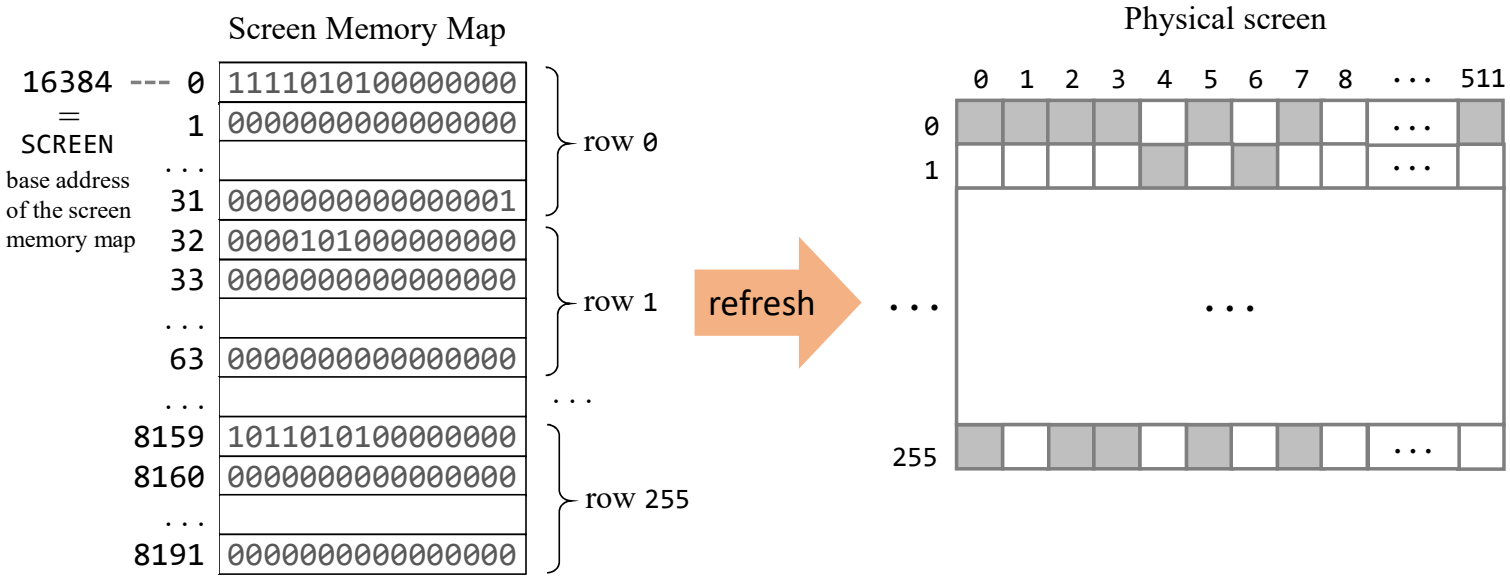
Bitmaps



Examples of simple patterns that can be drawn relatively easily:

```
// Sets the first (left) 16 pixels
// of the top row to black
```

Bitmaps

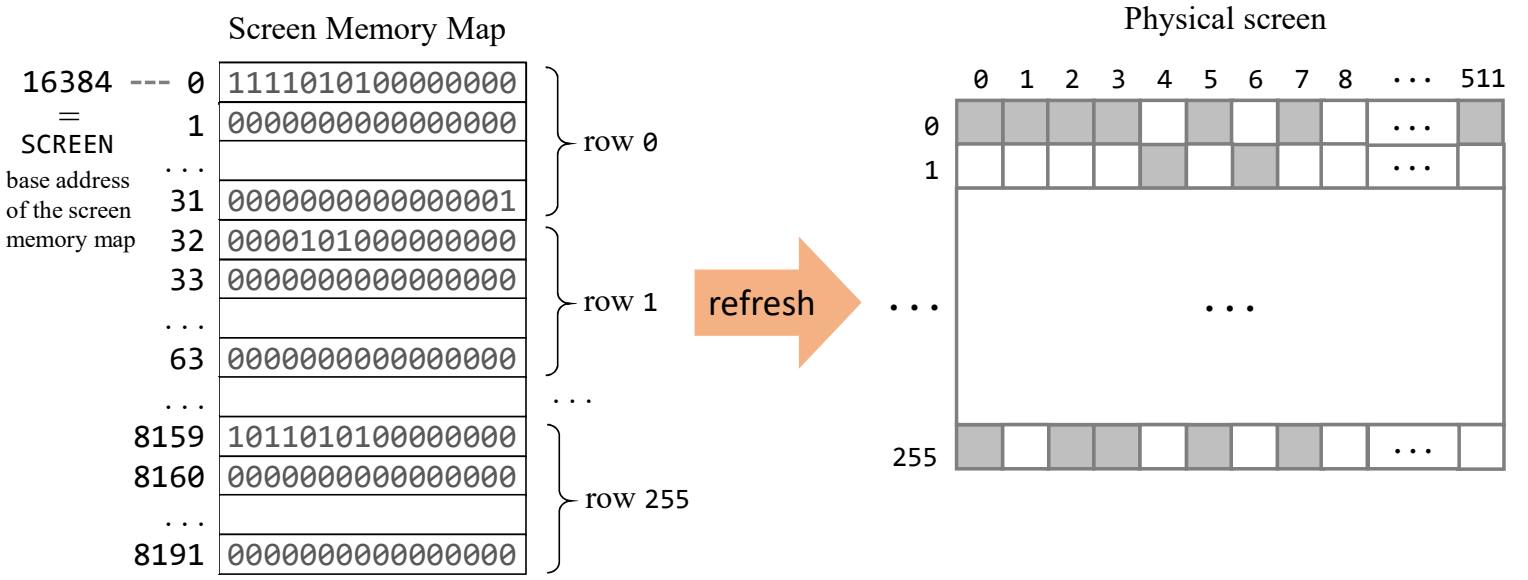


Examples of simple patterns that can be drawn relatively easily:

```
// Sets the first (left) 16 pixels
// of the top row to black
@SCREEN
M=-1        // -1 = 1111111111111111
```

```
// Sets the first 16 pixels
// of row 2 to black
?
```

Bitmaps



Examples of simple patterns that can be drawn relatively easily:

```
// Sets the first (left) 16 pixels
// of the top row to black
@SCREEN
M=-1        // -1 = 1111111111111111
```

```
// Sets the first 16 pixels
// of row 2 to black
@64
D=A
@SCREEN
A=A+D
M=-1
```

Chapter 4: Machine Language

Overview

- Machine languages
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Low Level Programming

- Basic
- Iteration
- Pointers

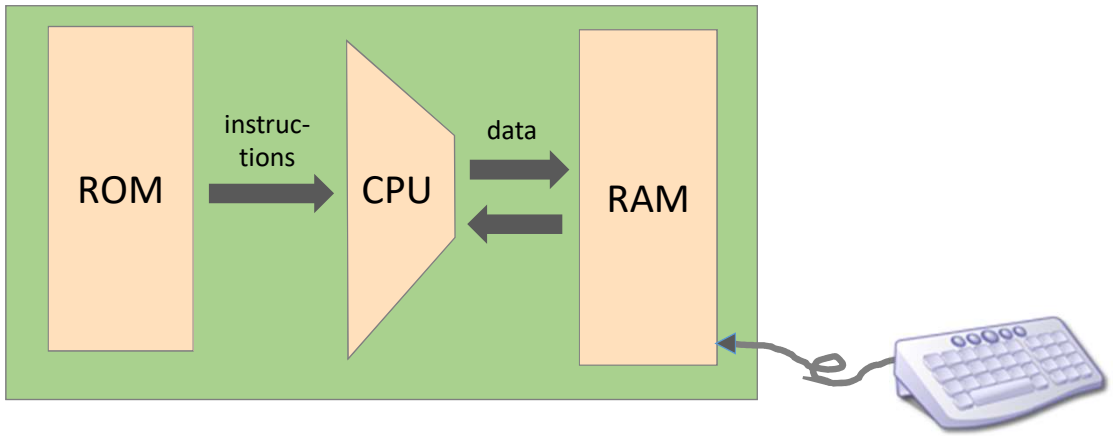
Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- ✓ Usage
- ✓ Specification
- ✓ Output
- ➡ Input
- Project 4

Input



Keyboard: used to enter inputs

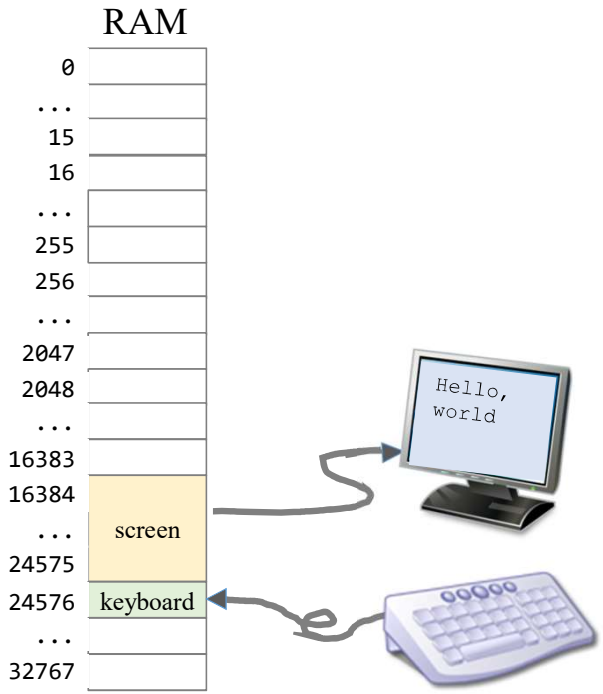
High-level input handling

`readInt`, `readString`, ...

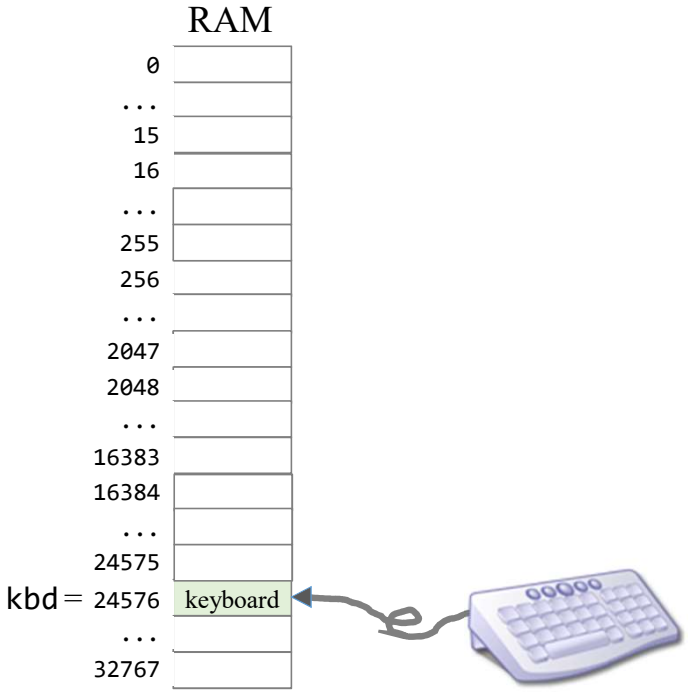
Low-level input handling

Read bits.

Hack RAM


RAM
0
...
15
16
...
255
256
...
2047
2048
...
16383
16384
...
24575
24576
...
32767
screen
keyboard
Hello,
world

Hack RAM



Keyboard memory map

A single 16-bit register, dedicated to representing the keyboard

Base address: `KBD` = 24576 (predefined symbol)

Reading inputs is affected by probing this register.

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

key	code
:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	…
…	…
Z	90

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

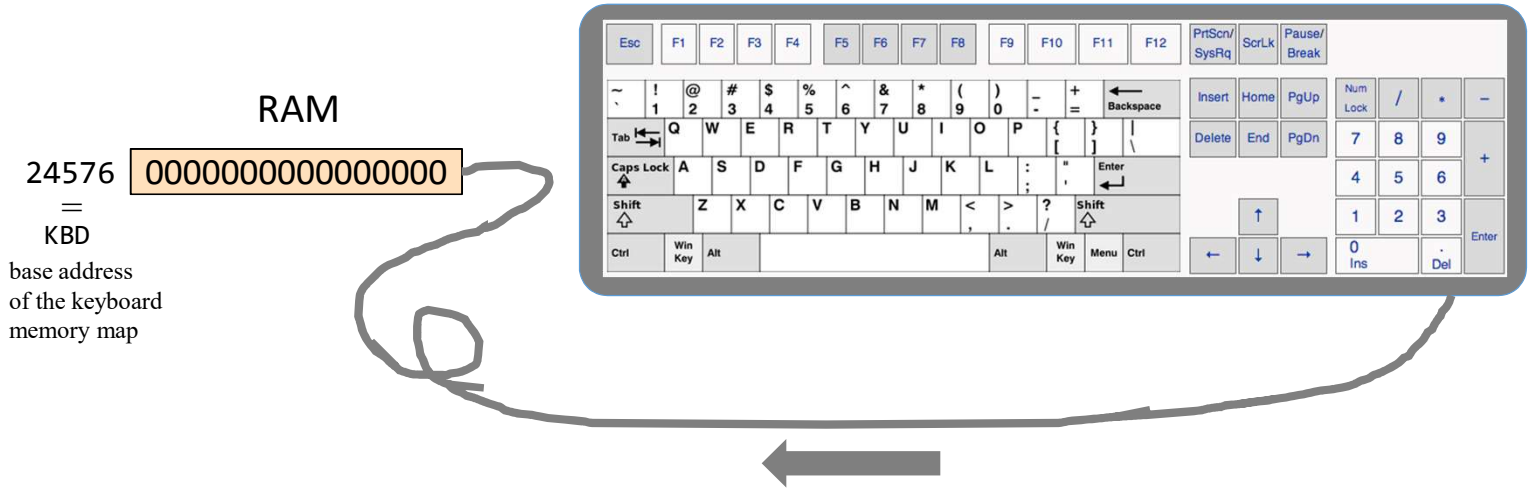
key	code
a	97
b	98
c	99
…	…
z	122

key	code
{	123
|	124
}	125
~	126

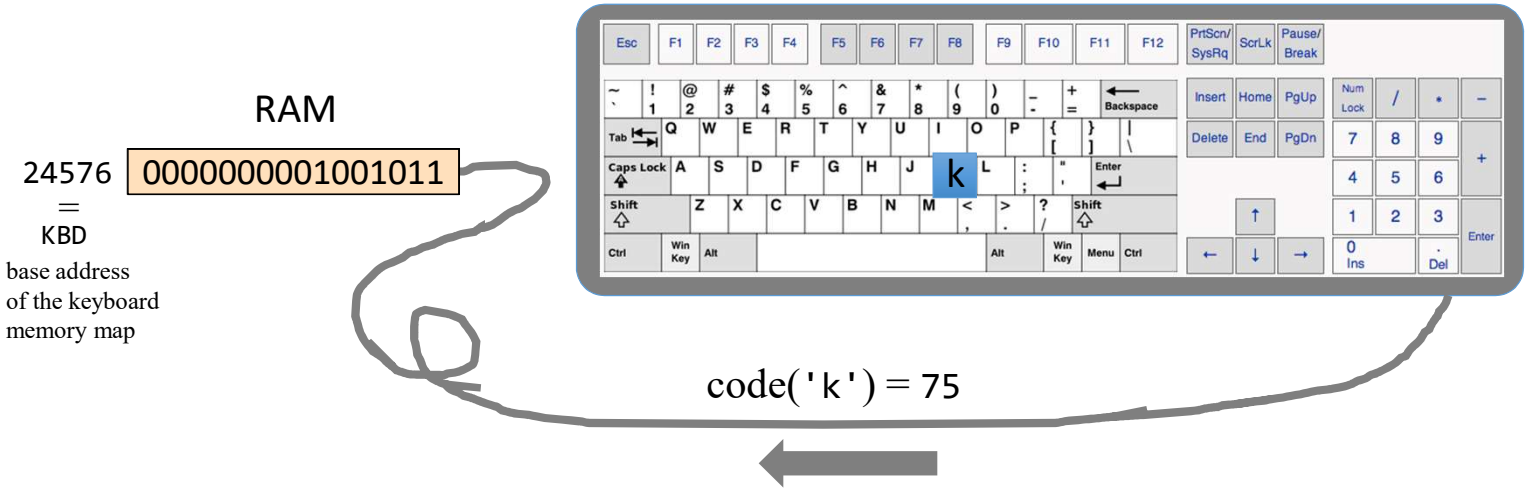
key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
. . .	. . .
f12	152

(Subset of Unicode)

Memory mapped input

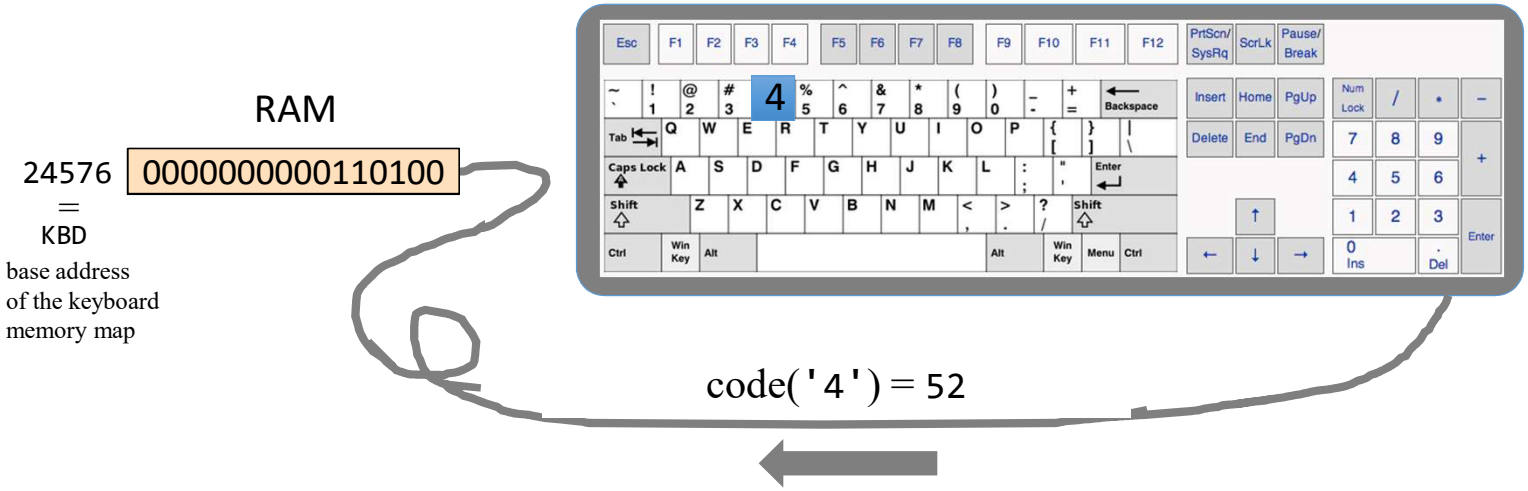


Memory mapped input



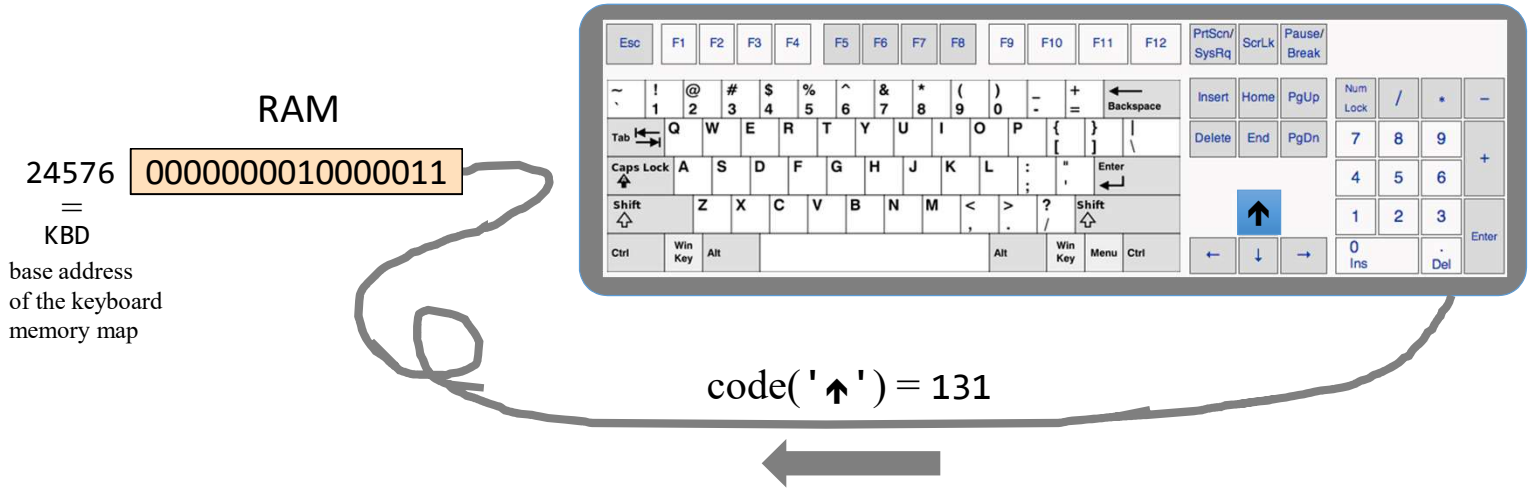
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input



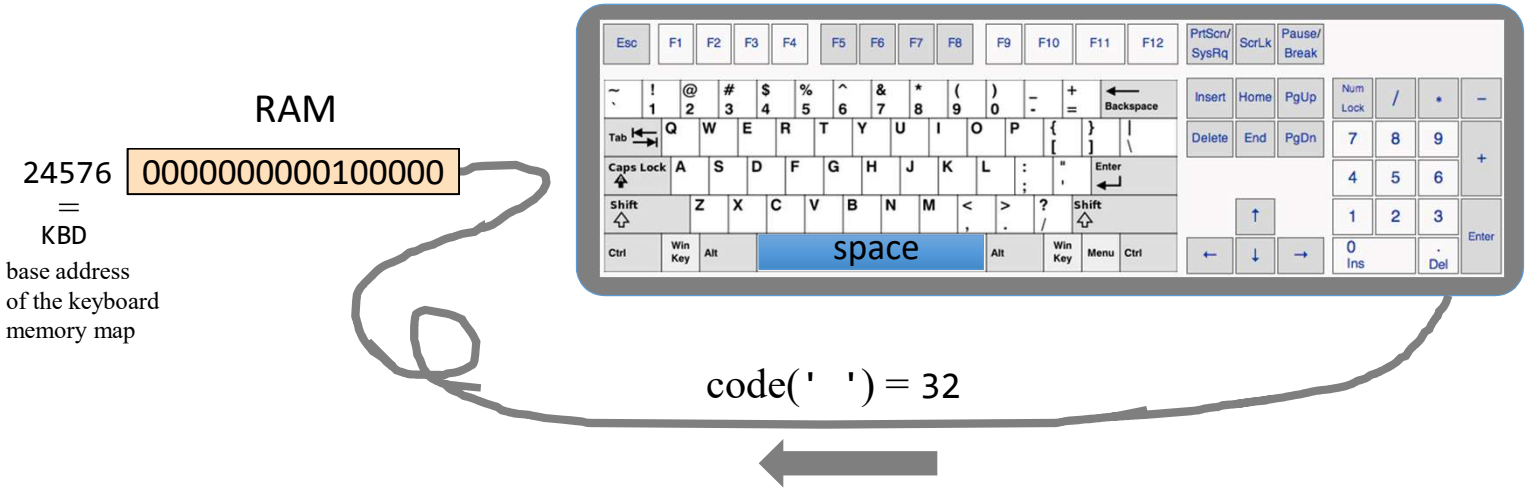
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input



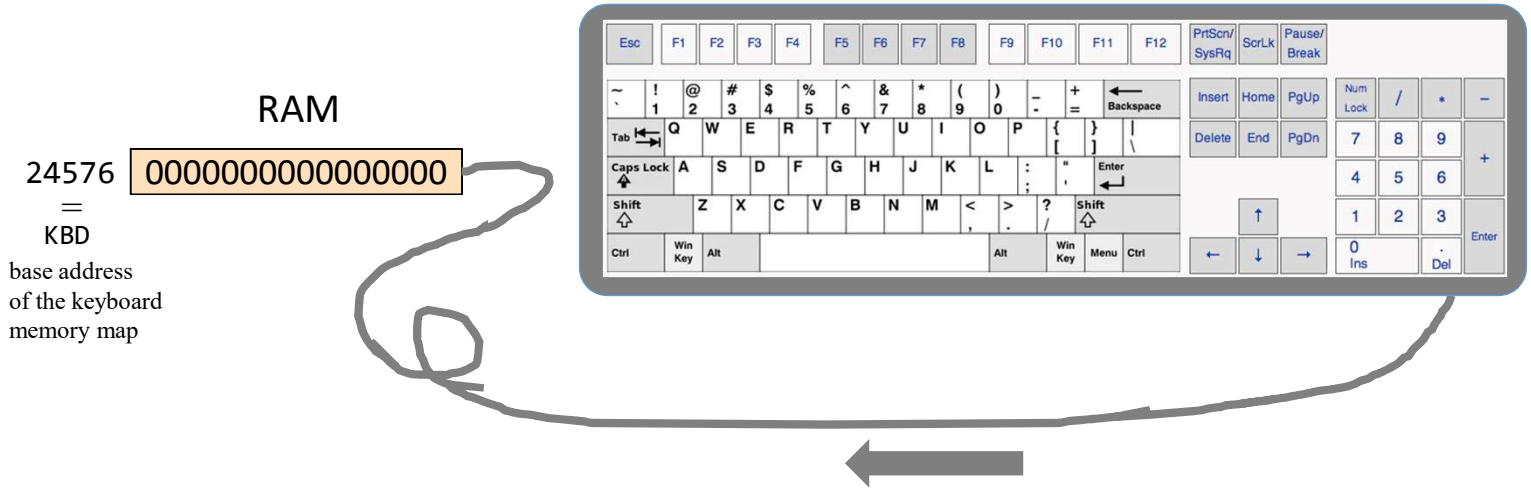
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input



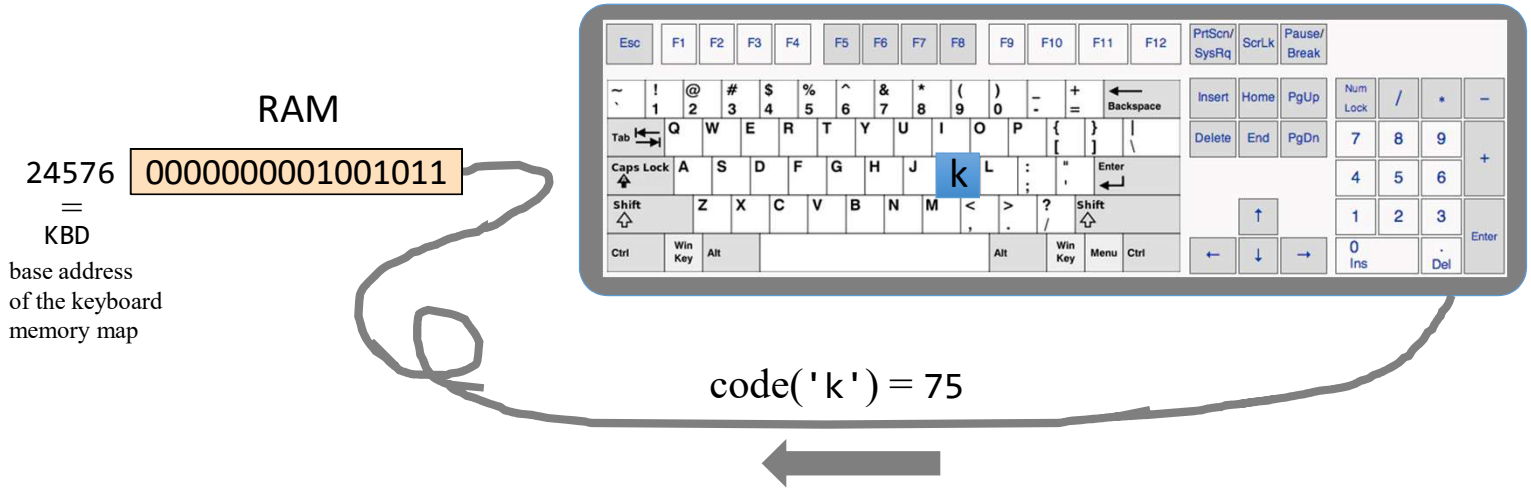
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input



When no key is pressed, the resulting code is `0`.

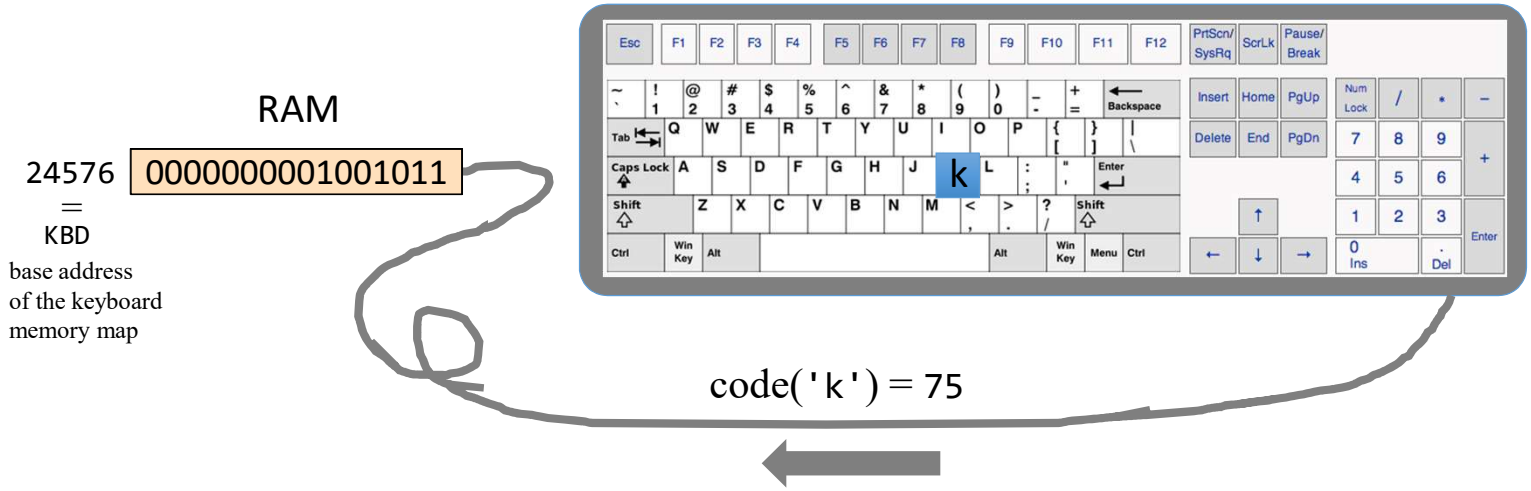
Reading inputs



Examples:

```
// Set D to the character code of
// the currently pressed key
```

Reading inputs

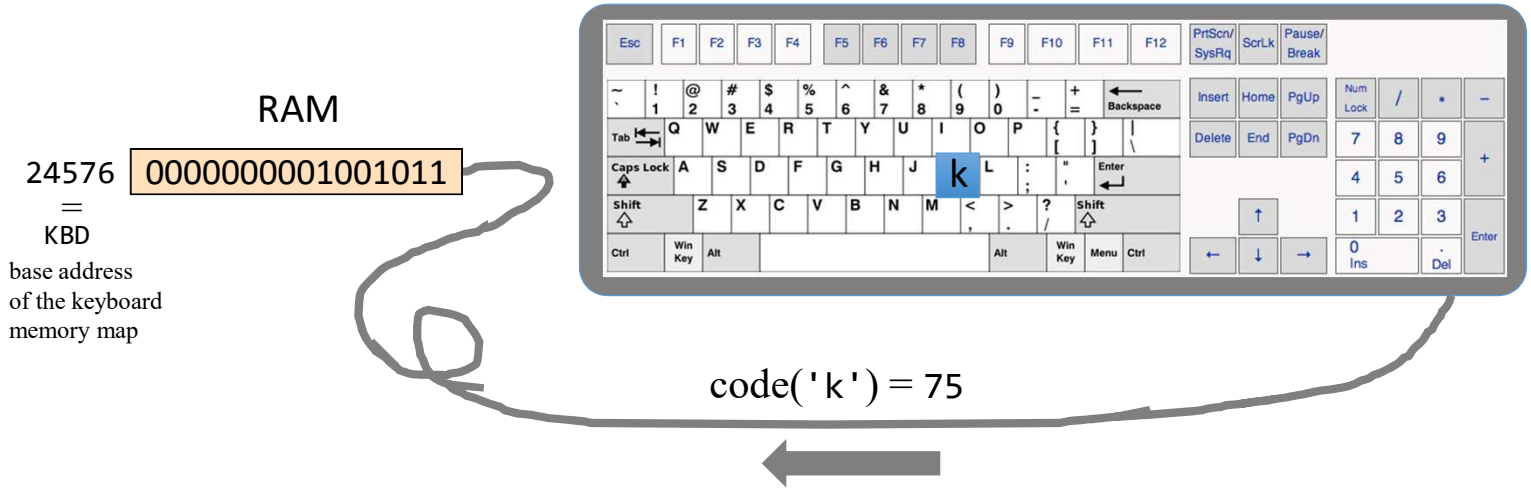


Examples:

```
// Set D to the character code of
// the currently pressed key
@KBD
D=M
```

```
// If the currently pressed key is 'q', goto END
```

Reading inputs

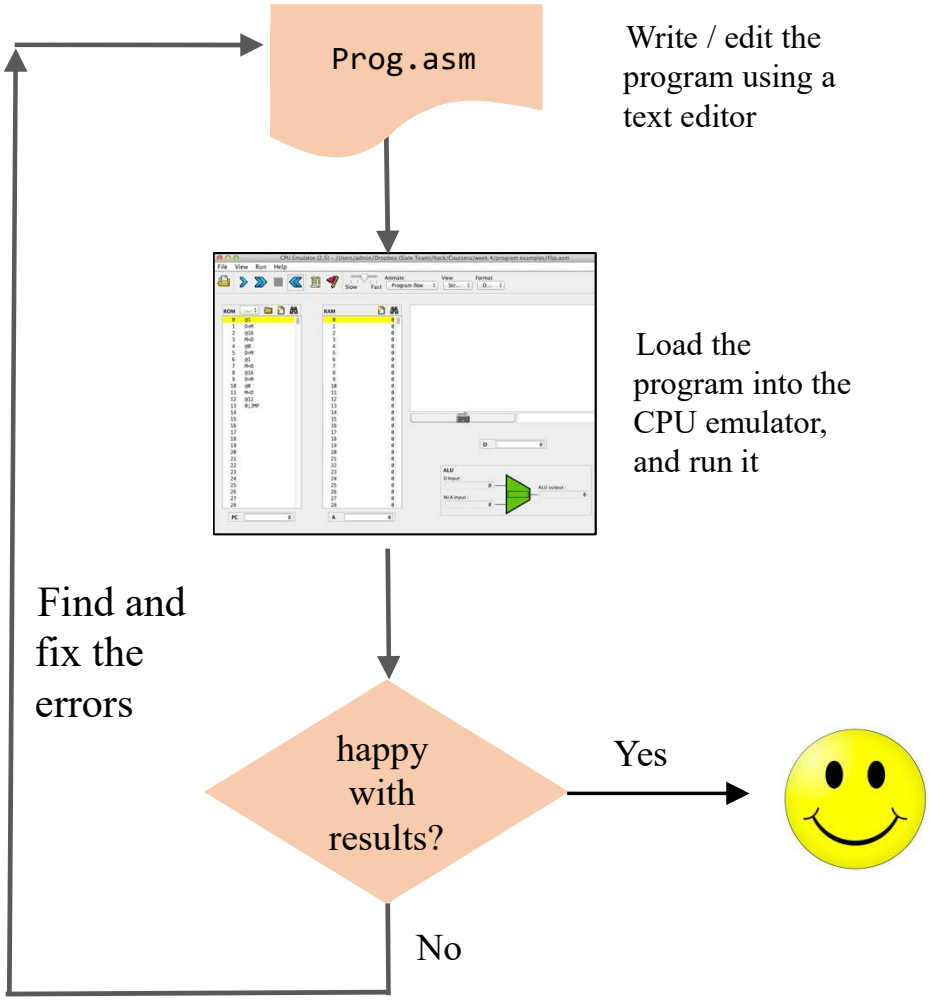


Examples:

```
// Set D to the character code of
// the currently pressed key
@KBD
D=M
```

```
// If the currently pressed key is 'q', goto END
@KBD
D=M
@113   // 'q'
D=D-A
@END
D;JEQ
```

Program development process



Translation options

1. Let the CPU emulator translate into binary code (as seen on the left)

2. Use the supplied assembler:

- Find it on your PC in `nand2tetris/tools`
- See the *Assembler Tutorial* in Project 6 (`www.nand2tetris.org`)

Implementation notes

Well-written low-level code is

- Compact
- Efficient
- Elegant
- Self-describing

Tips

- Use symbolic variables and labels
- Use sensible variable and label names
- Variables: lower-case
- Labels: upper-case
- Use indentation
- Start by writing pseudocode.

Nand to Tetris Roadmap (Part I: Hardware)