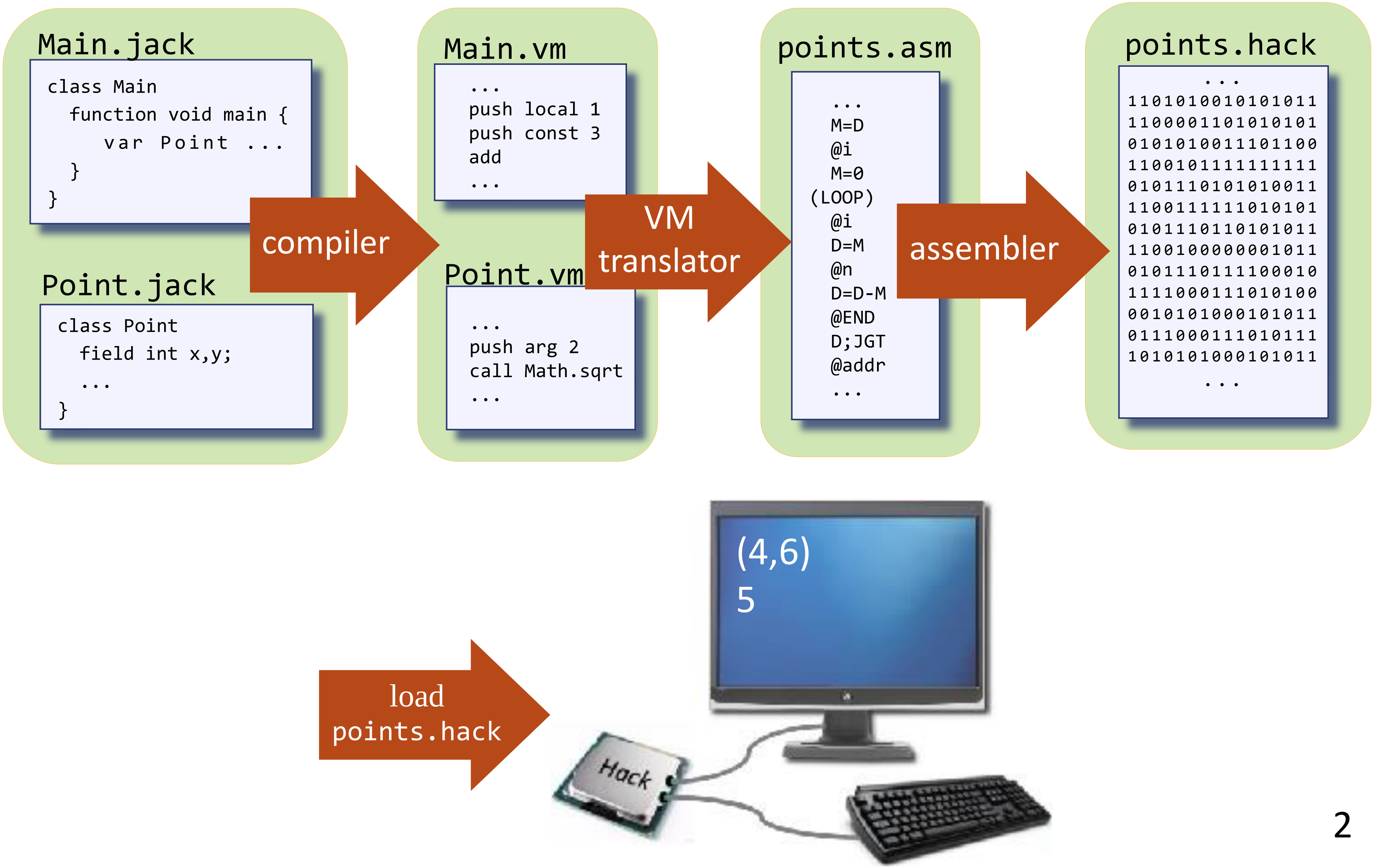
Revision Slides

Dr. Ren Jianfeng

Big picture

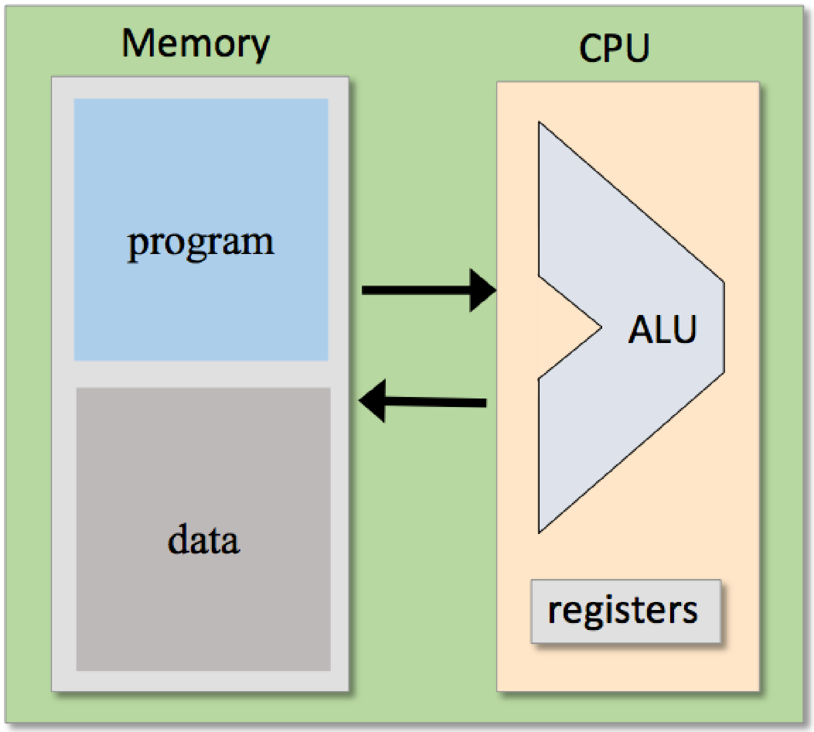

Main.jack
class Main
function void main {
var Point ...
}
}
Point.jack
class Point
field int x,y;
...
}
compiler
Main.vm
...
push local 1
push const 3
add
...
Point.vm
...
push arg 2
call Math.sqrt
...
VM translator
points.asm
...
M=D
@i
M=0
(LOOP)
@i
D=M
@n
D=D-M
@END
D;JGT
@addr
...
assembler
points.hack
...
1101010010101011
1100001101010101
0101010011101100
1100101111111111
0101110101010011
1100111111010101
0101110110101011
1100100000001011
0101110111100010
1111000111010100
0010101000101011
0111000111010111
1010101000101011
...
load points.hack
(4,6)
5
Hack

Outlines

- Hack assembly programming
- Assembler
- Virtual machine

An informal definition

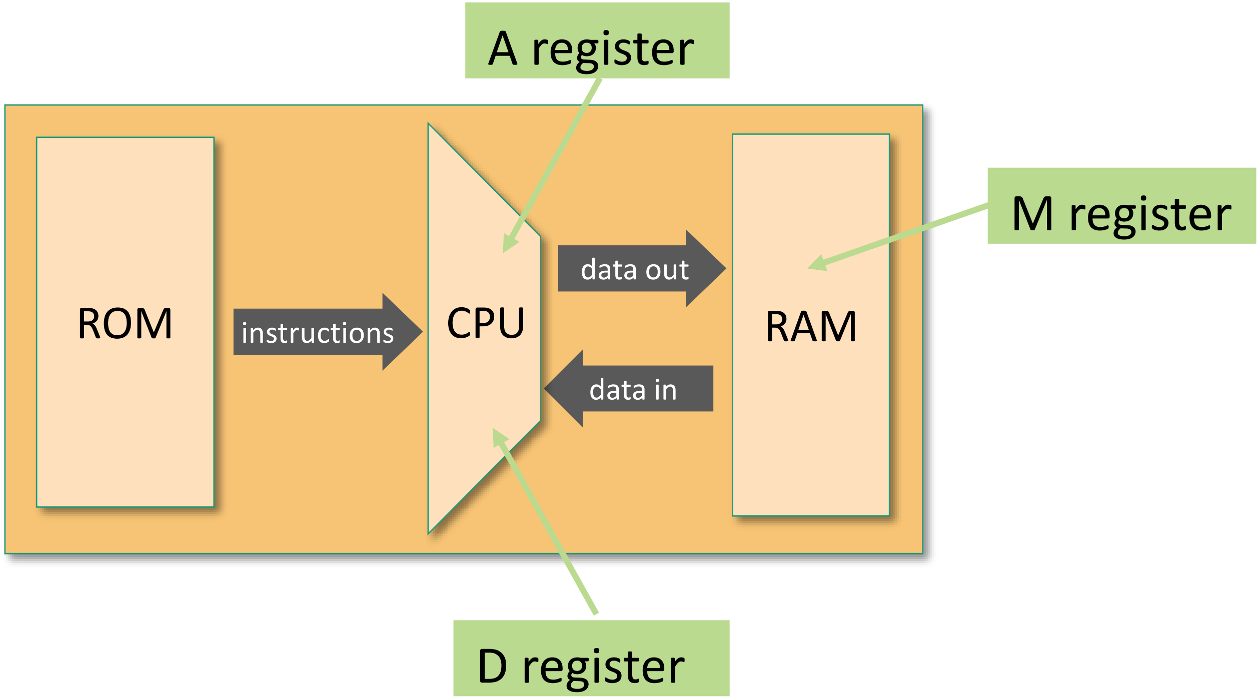
- A *machine language* can be viewed as an agreed-upon formalism, designed to manipulate a *memory* using a *processor* and a set of *registers*. (Nisan & Schocken)



Addressing modes

- Register
  - ADD R1, R2 // $R2 \leftarrow R2 + R1$
  - Access data from a **register** R2.
- Direct
  - ADD R1, M[67] // $Mem[67] \leftarrow Mem[67] + R1$
  - LOAD R1, 67 // $R1 \leftarrow Mem[67]$
  - Access data from **fixed memory address** 67.
- Indirect
  - ADD R1, @A // $Mem[A] \leftarrow Mem[A] + R1$
  - Access data from **memory address specified by variable A**.
- Immediate
  - ADD 67, R1 // $R1 \leftarrow R1 + 67$
  - LOADI R1, 67 // $R1 \leftarrow 67$
  - Access the data of **value** 67 immediately.

Hack computer: registers



- Three 16-bit registers:
  - D: Store data
  - A: Store data / address the memory
  - M: Represent currently addressed memory register: **M = RAM[A]**

A-instruction specification

Semantics: Set the A register to *value*

Symbolic syntax:

@ *value*

Example:

@21

set A to 21

Where *value* is either:

- a non-negative decimal constant $\le$ 65535 (=2^{15}-1) or
- a symbol referring to a constant (*come back to this later*)

Binary syntax:

0 *value*

Where *value* is a 15-bit binary constant

Example:

0000000000010101

opcode signifying an A-instruction

set A to 21

C-instruction

<u>Syntax:</u>

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 refer to **RAM[A]**)

jump =

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

<u>Semantics:</u>

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

C-instruction specification

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

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

opcode

not used

comp bits

dest bits

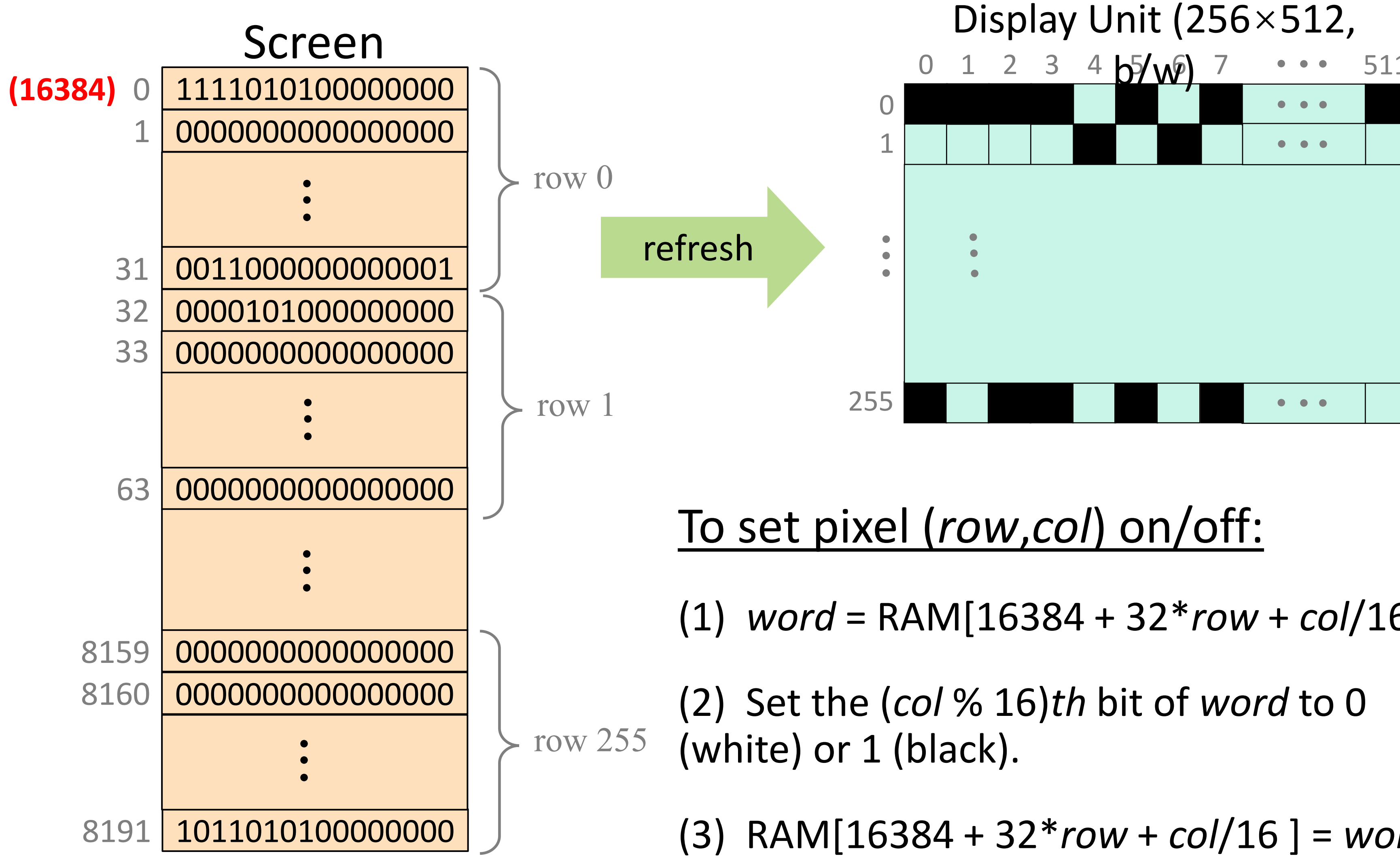
jump bits

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

Memory mapped output



8192 = 256×512/16

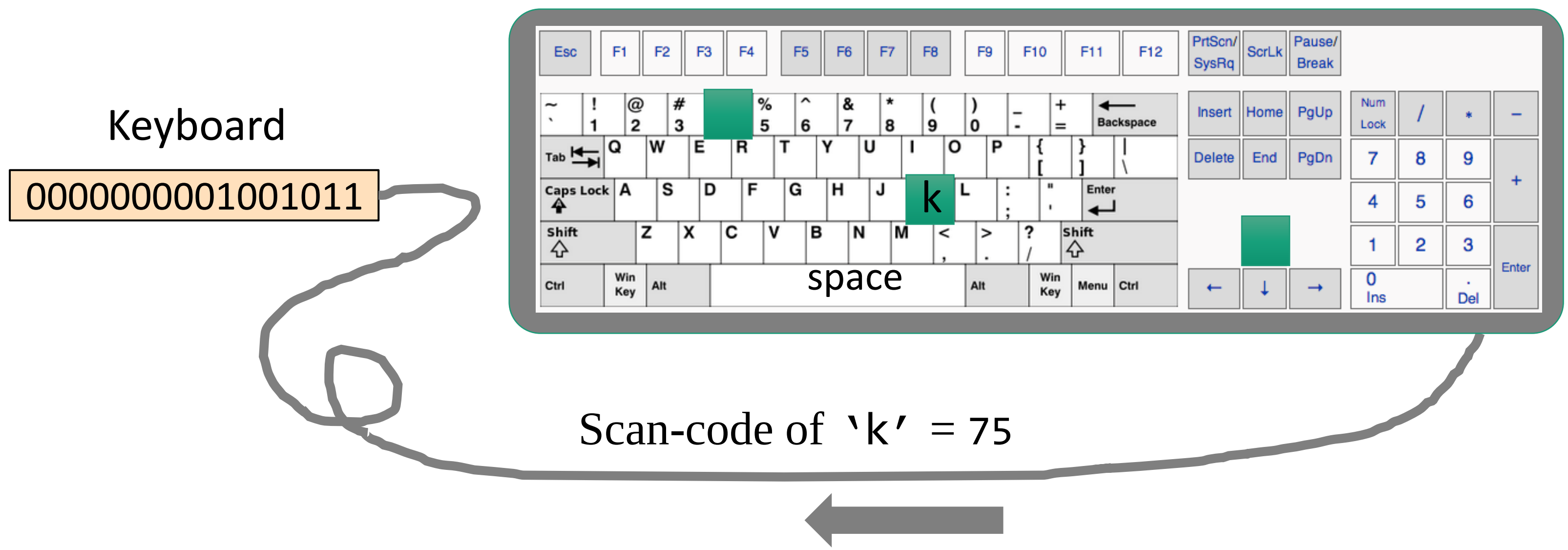
<u>To set pixel (*row*,*col*) on/off:</u>

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

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

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

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]**.

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
```

- Jump to instruction number A (which happens to be 6)
- 0: syntax convention for jmp instructions

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	

<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
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 chapters 7-8.

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

Implications:

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

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	

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

Symbol resolution rules:

- 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	

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
```

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
```

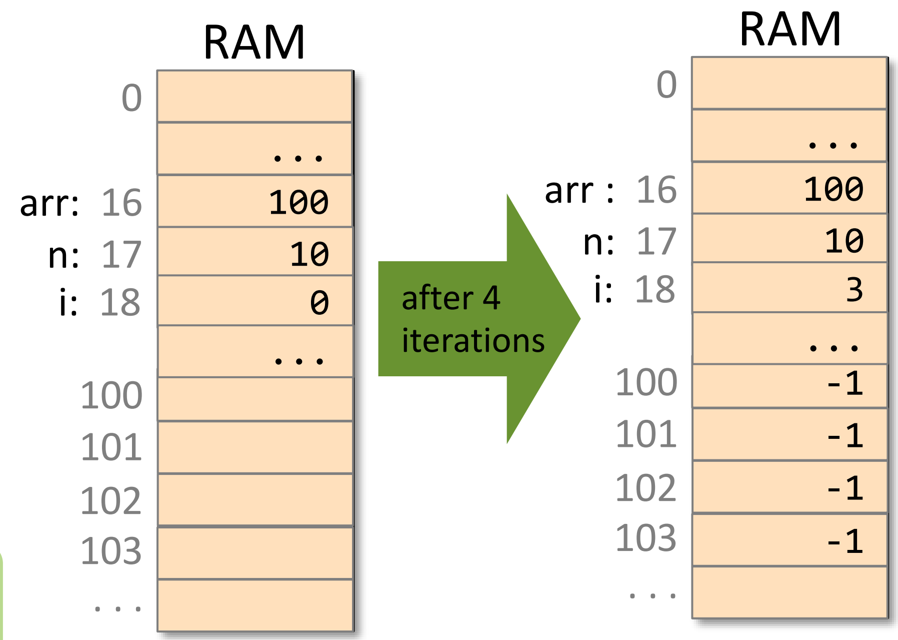
```
   @sum
   D=M // D = sum
   @i
   D=D+M // D = sum + i
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP // goto LOOP
(STOP)
   @sum
   D=M // D = sum
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP // end
```

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
  @LOOP
  0;JMP // goto LOOP
(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
- Assembler
- Virtual machine

Translating A-instructions

Symbolic syntax:

@ *value*

Examples:

@21

@foo

Where *value* is either

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

Binary syntax:

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`

Hack language specification: symbols

Pre-defined symbols:

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

Label declaration: (*label*)

Variable declaration: *@variableName*

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

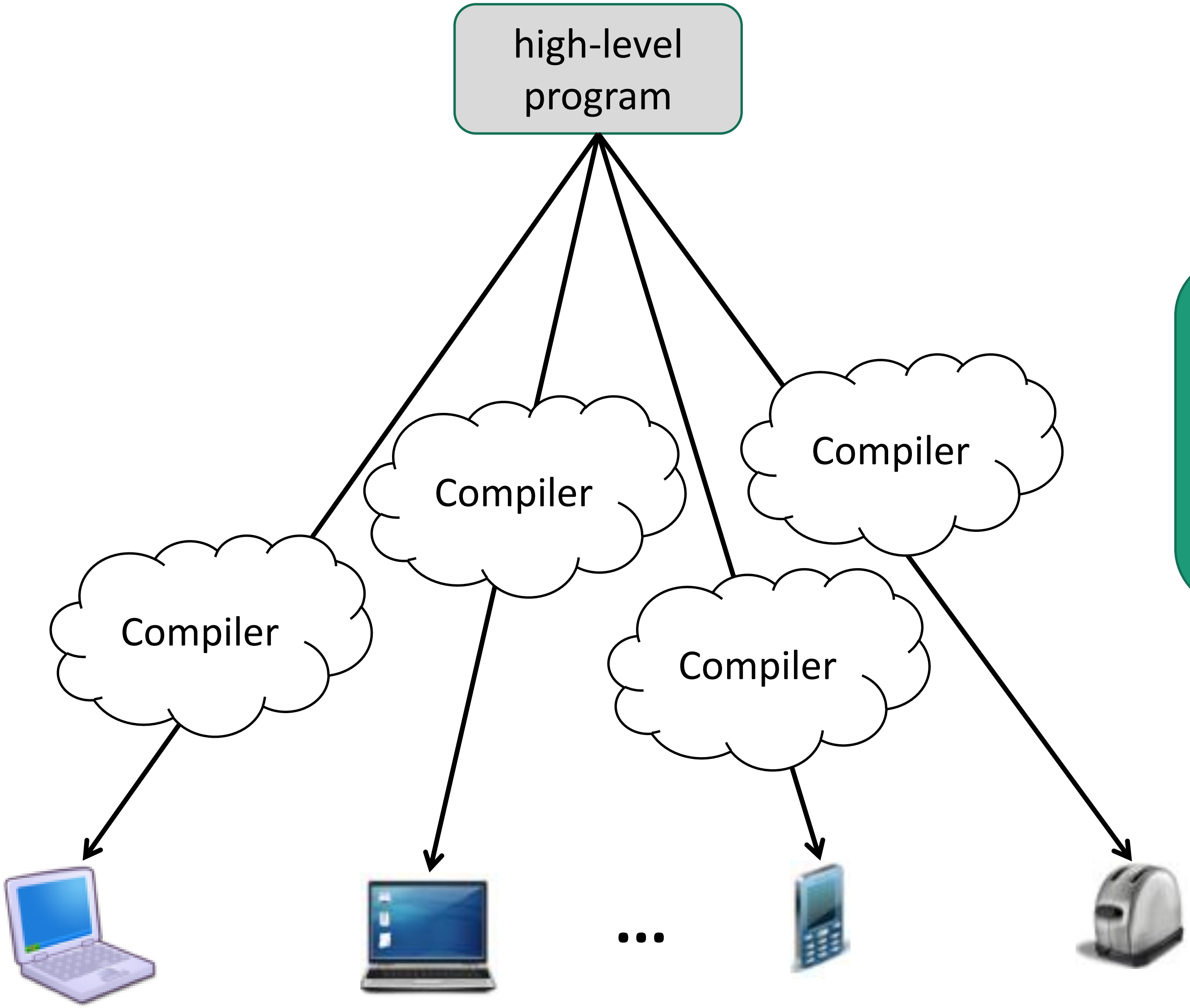
Outlines

- Hack assembly programming
- Assembler
- Virtual machine

Why we need virtual machine?

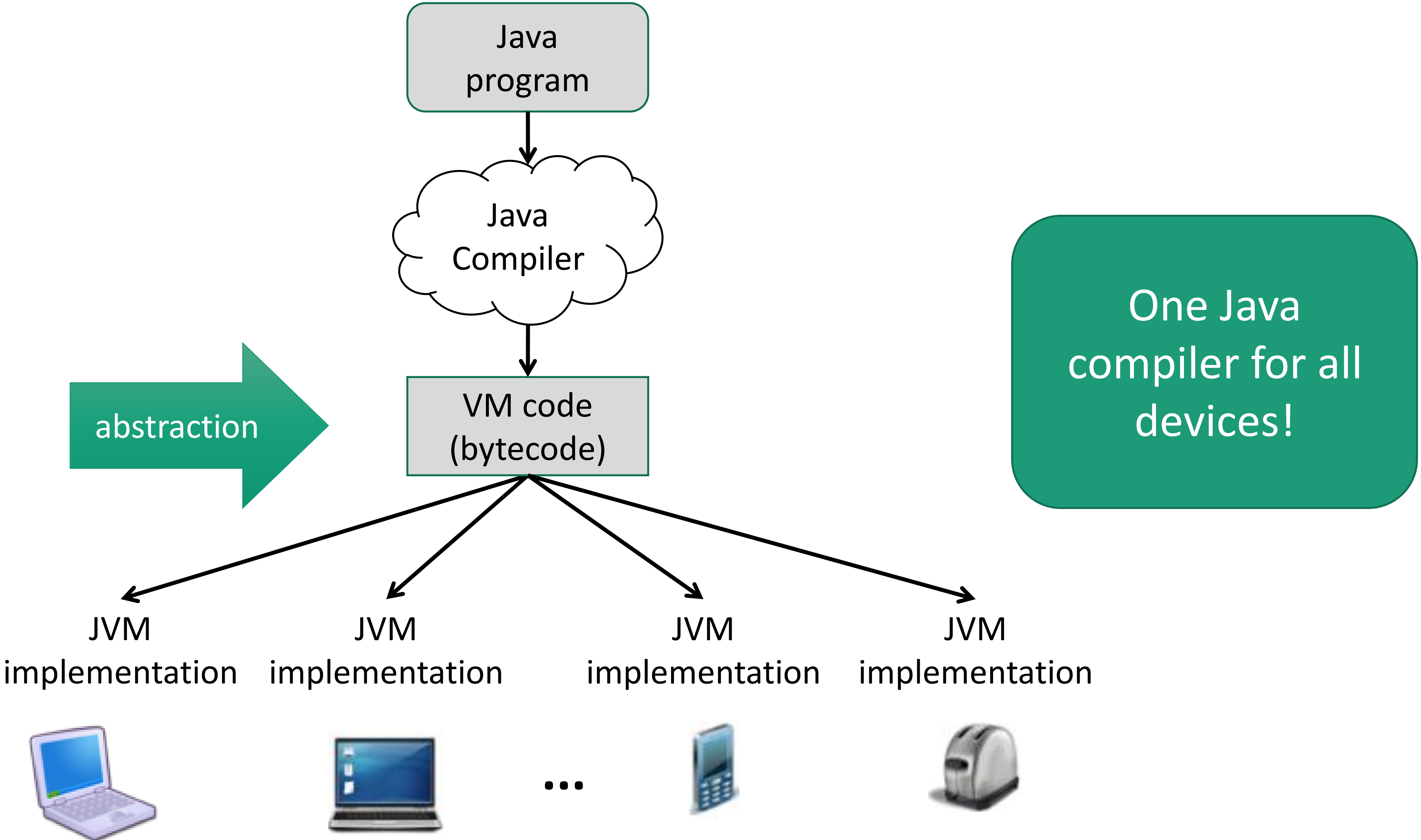
- **Code transportability**
  - **Many** high-level languages can work on the same platform: virtual machine.
  - VM may be implemented with relative ease on **multiple** target platforms.
  - As a result, VM-based software can run on **many** processors and operating systems without modifying source code.

Program compilation: 1-tier



One compiler for each device!

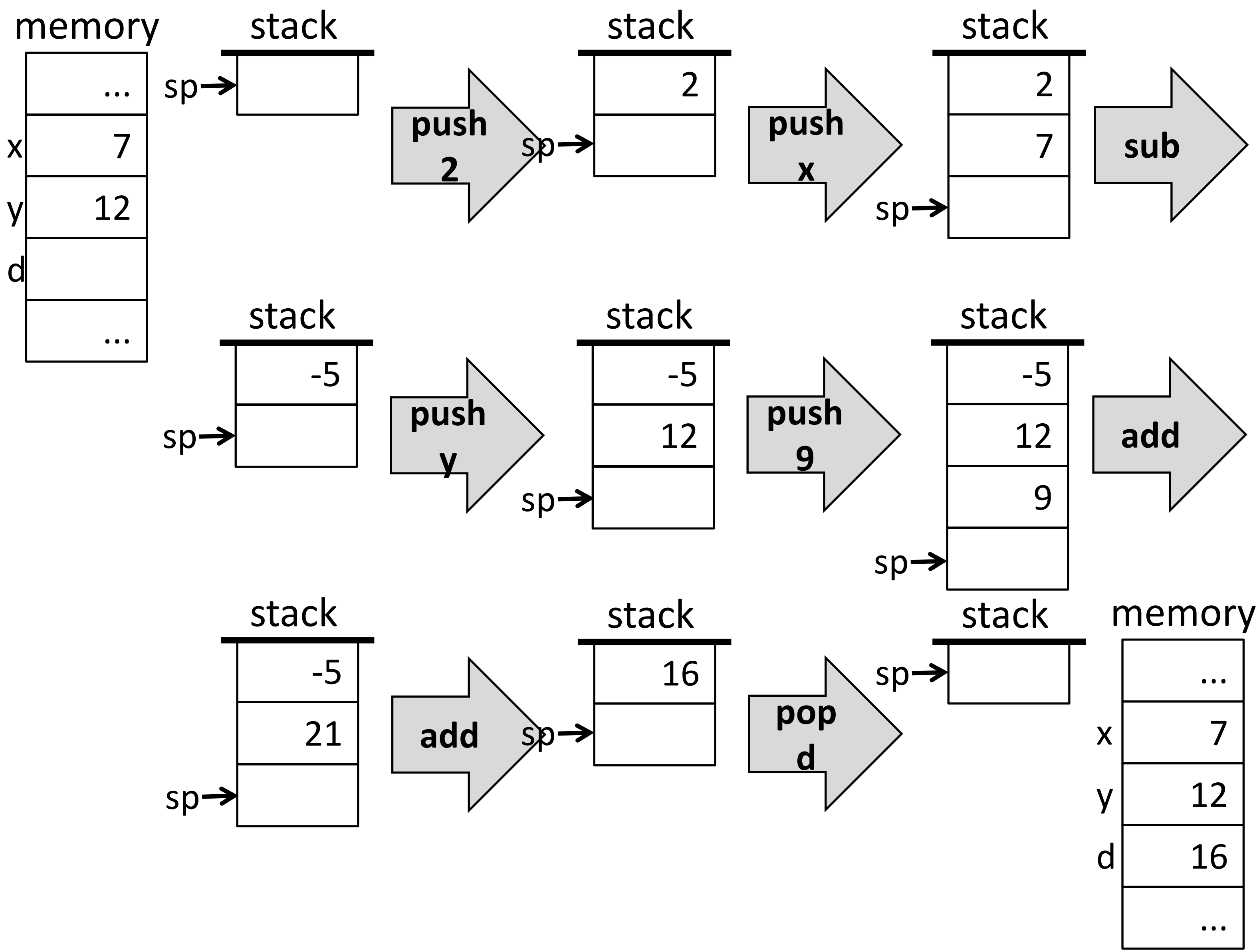
Program compilation: 2-tier



Arithmetic commands

VM code

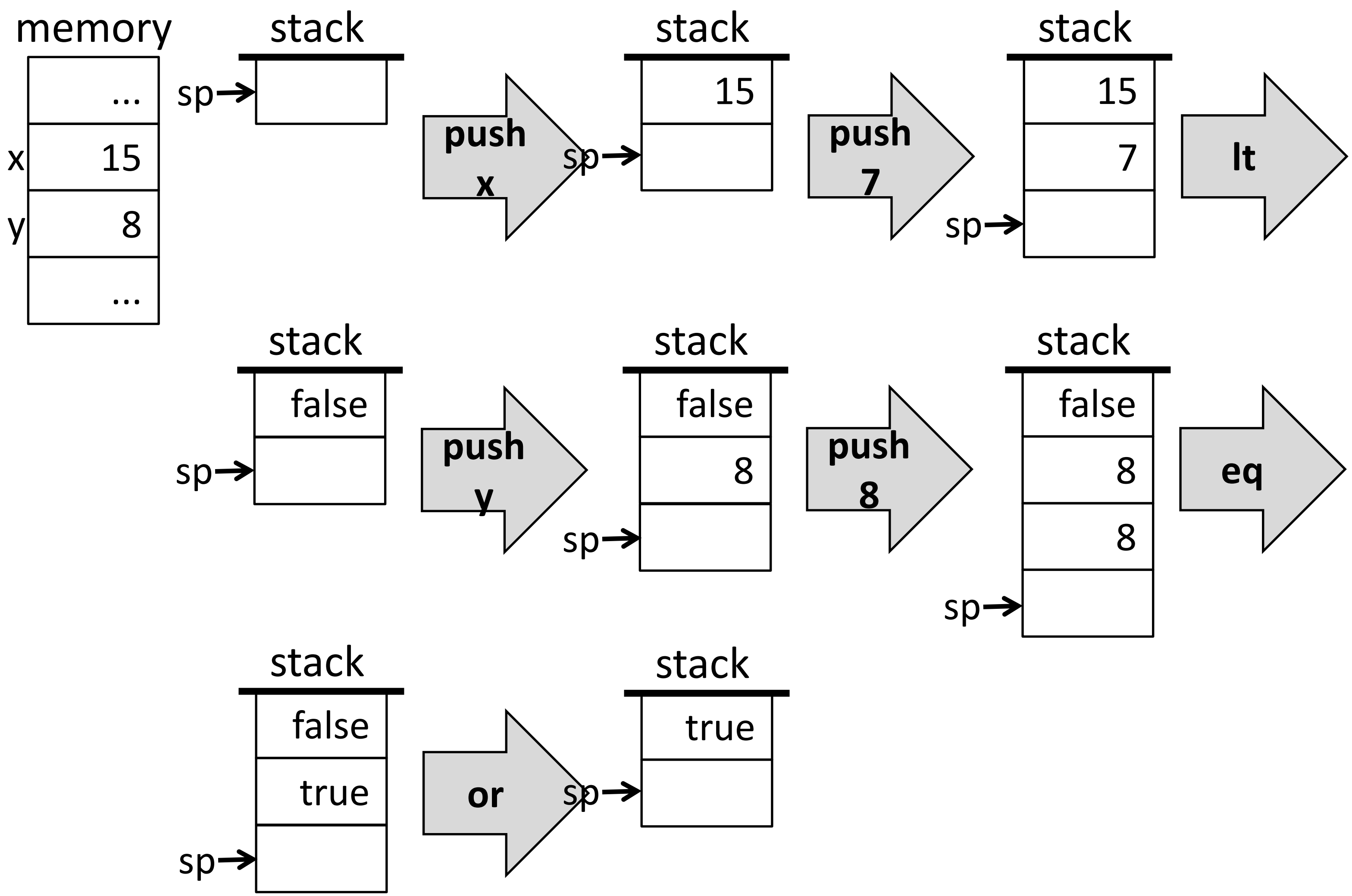
```
// d=(2-x) +
// (y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Logical commands

VM code

```
// (x<7)
// or
// (y==8)
push x
push 7
lt
push y
push 8
eq
or
```

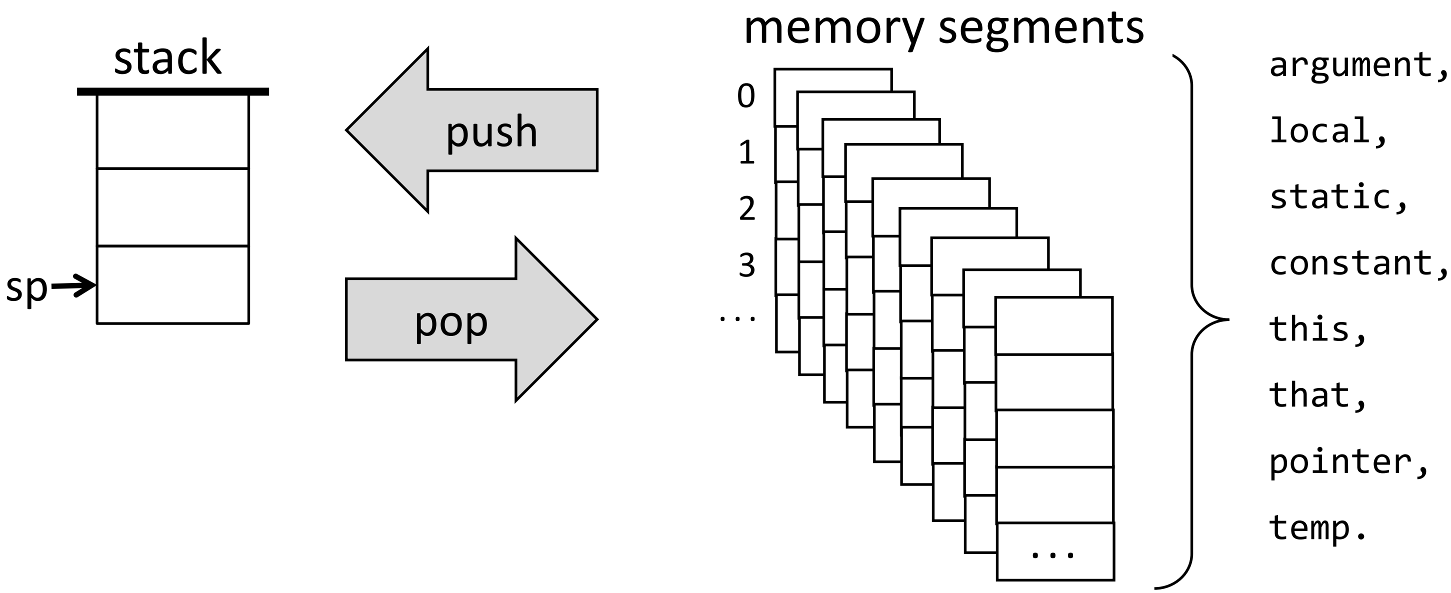


Arithmetic / Logical commands

Command	Return value	Return value
add	$x + y$	integer
sub	$x - y$	integer
neg	$-y$	integer
eq	$x == y$	boolean
gt	$x > y$	boolean
lt	$x < y$	boolean
and	x and y	boolean
or	x or y	boolean
not	not x	boolean

<u>Observation:</u> Any arithmetic or logical expression can be expressed and evaluated by applying some sequence of the above operations on a stack.

Memory segments



Syntax: push *segment i*

where *segment* is: argument, local, static, constant,
this, that, pointer, or temp

and *i* is a non-negative integer.

Syntax: pop *segment i*

Where *segment* is: argument, local, static,
this, that, pointer, or temp

and *i* is a non-negative integer.

Program control

- goto *label*
  - ➢ jump to execute the command just after *label*
- if-goto *label*
  - ➢ *cond* = pop
  - ➢ if *cond* jump to execute the command just after *label*
- label *label*
  - ➢ label declaration command
- Implementation (VM translation):
  - ➢ Translate each branching command into assembly instructions that effect the specified operation on the host machine.

Simple implementation:

the assembly language has similar branching commands.

Functions in VM language

```
// Computes 3 +5 * 8
0 function main 0
1   push constant 3
2   push constant 8
3   push constant 5
4   call mult 2
5   add
6   return
```

caller

```
// Computes the product of two given arguments
0 function mult 2
1   push constant 0
2   pop local 0
3   push constant 1
4   pop local 1
5 label LOOP
6   push local 1
7   push argument 1
    //... computes the product into local 0
19 label END
20   push local 0
21   return
```

callee

Implementation

We can write low-level code to

- Handle the VM command call
- Handle the VM command function
- Handle the VM command return.

Functions in VM language

```
// Computes 3 +5 * 8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
    //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling call:

- Determine the **return address** within the caller's code;
- **Save** the caller's return address, stack and memory segments;
- **Pass parameters** from the caller to the callee;
- **Jump** to execute the callee.

Functions in VM language

```
// Computes 3 +5 * 8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
    //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling function:

- Initialize the local variables of the callee;
- Handle some other simple initializations (later);
- Execute the callee function.

Functions in VM language

```
// Computes 3 +5 * 8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling return:

(a function always ends by pushing a return value on the stack)

- **Return** the *return value* to the caller;
- **Recycle** the memory resources used by the callee;
- **Reinstate** the caller's stack and memory segments;
- **Jump** to the return address in the caller's code.

Pointer manipulation

Pseudo assembly code

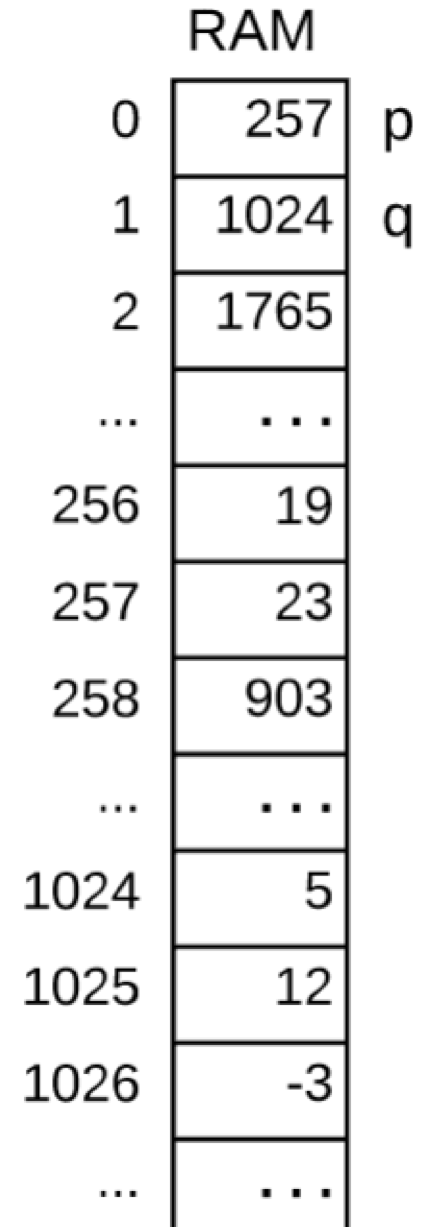
```
D = *p // D becomes 23

p--     // RAM[0] becomes 256
D = *p // D becomes 19

*q = 9 // RAM[1024] becomes 9
q++     // RAM[1] becomes 1025
```

In Hack:

```
@p
A=M
D=M
```



Notation:

*p // the memory location that p points at

x-- // decrement: x = x - 1

x++ // increment: x = x + 1

Implement **push constant** *i*

VM code:

push constant *i*

VM Translator

Assembly psuedo code:

*SP = *i*, SP++

Hack assembly:

```
// D = i
@i
D=A
// *SP=D
@SP
A=M
M=D
// SP++
@SP
M=M+1
```

(no pop constant operation)

<u>Implementation</u>:

Supplies the specified constant.

Implement `pop local` *i*

Abstraction

pop local ***i***

Implementation:

addr=LCL+ ***i***, SP--, *addr=*SP

***i* is a constant here!!!**
but LCL is a variable.

Hack assembly:

```
@i      // addr=LCL+i
D=A
@LCL
D=D+M
@addr
M=D
@SP     // SP--
M=M-1
@SP     // D=*SP
A=M
D=M
@addr   // *addr=D
A=M
M=D
```

Implement **push/pop** `local` *i*

VM code:

```
pop local i
```

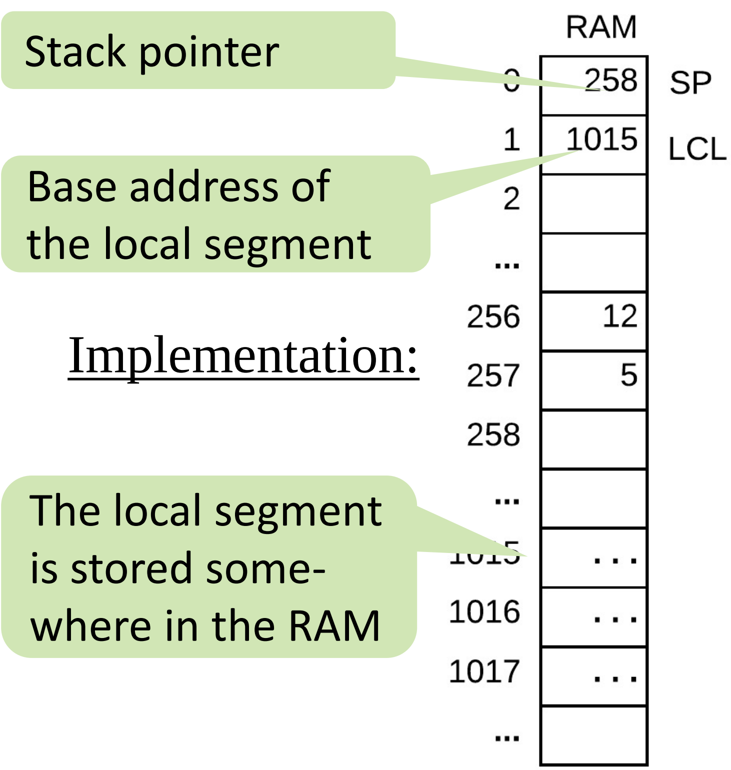
```
push local i
```

VM Translator

Assembly pseudo code:

addr = LCL+ *i*, SP--, *addr = *SP

addr = LCL+ *i*, *SP = *addr, SP++



Implementation:

Hack assembly:

```
// implement
// push local i
// addr=LCL+i
@i
D=A
@LCL
D=D+M
@addr
M=D
```

```
// *SP = *addr
@addr// D=*addr
A=M
D=M
@SP  // *SP=D
A=M
M=D
// SP++
@SP
M=M+1
```

Implement `push / pop local / argument / this / that` *i*

VM code:

`push` *segment i*

`pop` *segment i*

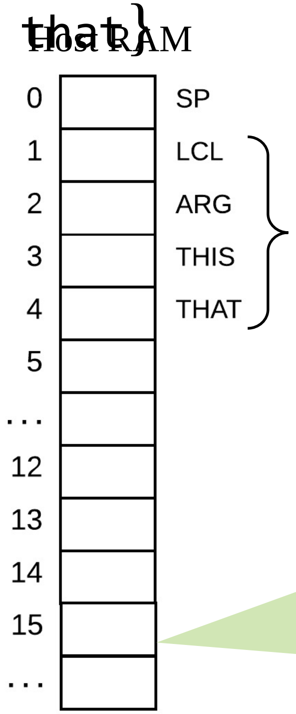
VM translator

Assembly pseudo code:

addr = *segmentPointer + i*, *SP = *addr, SP++

addr = *segmentPointer + i*, SP--, *addr = *SP

segment = {`local`, `argument`, `this`, `that`}



base addresses of the four segments are stored in these pointers

the four segments are stored somewhere in the RAM

- `push/pop local` *i*
- `push/pop argument` *i*
- `push/pop this` *i*
- `push/pop that` *i*

implemented precisely the same way.

Implement **push/pop static** *i*

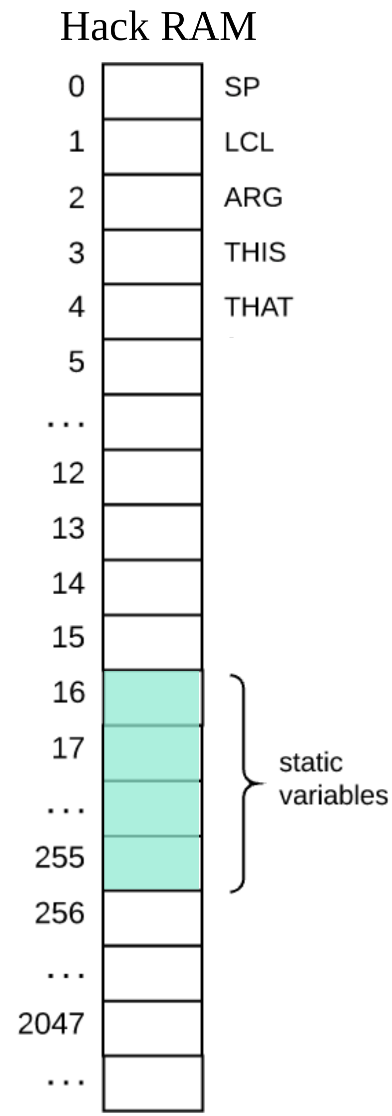
VM code:

```
// File Foo.vm
...
pop static 5
...
pop static 2
...
```

VM translator

Generated assembly code:

```
...
// D = stack.pop (code omitted)
@Foo.5
M=D
...
// D = stack.pop (code omitted)
@Foo.2
M=D
...
```



<u>The challenge:</u>

Static variables should be seen by all the methods in a program.

<u>Solution:</u>

Store them in some "**global space**":

- Have the VM translator translate each VM reference static *i* (in file Foo.vm) into an assembly reference Foo.*i*
- Following assembly, the Hack assembler will map these references onto RAM[16], RAM[17], ..., RAM[255]
- Therefore, the entries of the static segment will end up being mapped onto RAM[16], RAM[17], ..., RAM[255], in the order in which they appear in the program.

Implement **push/pop temp *i***

VM code:

```
push temp i
```

```
pop temp i
```

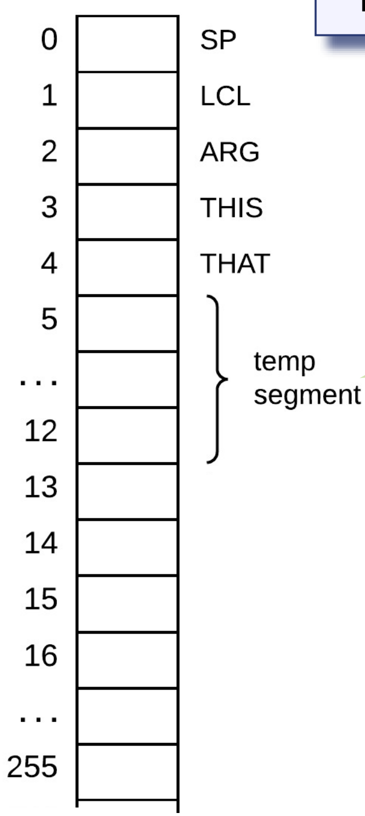
VM Translator

Assembly psuedo code:

addr = 5 + *i*, *SP = *addr, SP++

addr = 5 + *i*, SP--, *addr = *SP

Hack RAM



A fixed, **8-place** memory segment, stored in RAM locations 5 to 12

Implement **push/pop pointer 0/1**

VM code:

```
push pointer 0/1
```

VM Translator

```
pop pointer 0/1
```

Assembly psuedo code:

```
*SP = THIS/THAT, SP++
```

```
SP--, THIS/THAT = *SP
```

A fixed, 2-place segment:

- accessing pointer 0 should result in accessing THIS
- accessing pointer 1 should result in accessing THAT

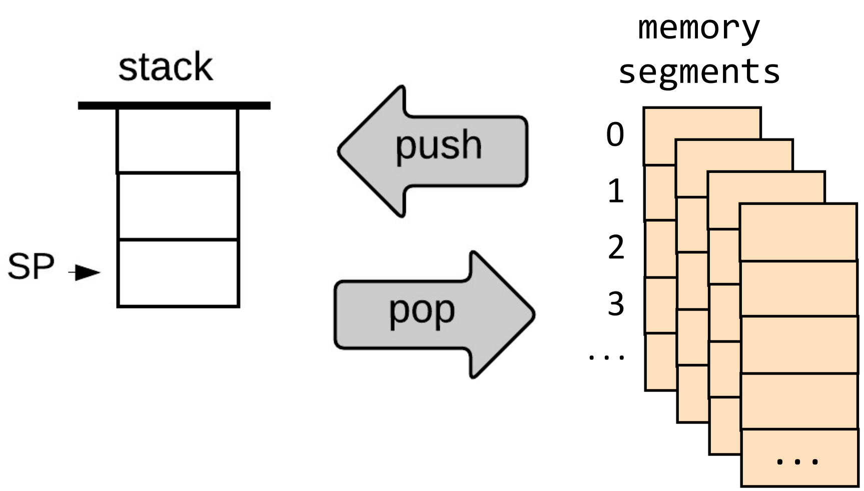
Implementation:

Supplies THIS or THAT // (the base addresses of this and that).

Branching

- goto *label*
  - jump to execute the command just after *label*
- if-goto *label*
  - *cond* = pop
  - if *cond* jump to execute the command just after *label*
- label *label*
  - label declaration command
- <u>Implementation</u> (VM translation):
  - The assembly language has similar branching commands.

The function's state



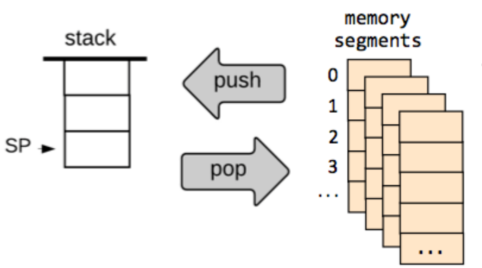
<u>During run-time:</u>

- Each function uses a **working stack** + **memory segments**
- The working stack and some of the segments should be:
  - Created when the function starts running,
  - Maintained as long as the function is executing,
  - Recycled when the function returns.

The function's state

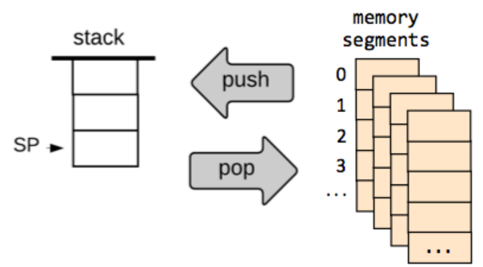
```
function main 0
    push constant 3
    push constant 8
    push constant 5
    call mult 2
    add
    return
```

caller




```
function mult 2
    push constant 0
    pop local 0
    ...
label LOOP
    push local 1
    //... computes the product
label END
    push local 0
    return
```

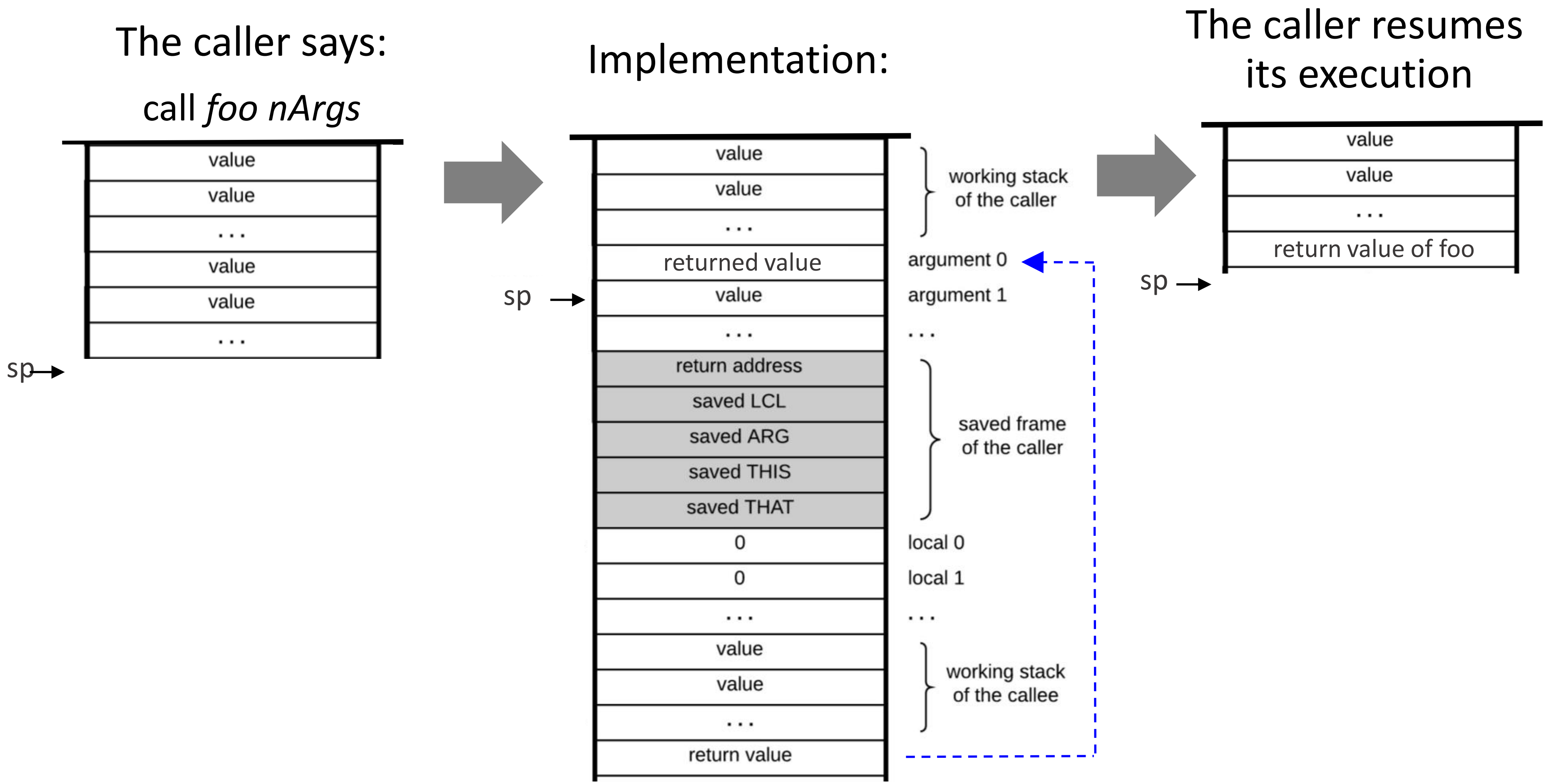
callee



Challenge:

- Maintain the states of all the functions up the calling chain.
- Can be done by using a single ***global*** *stack*.

Recap: function call and return



Booting

VM program convention

- one file in any VM program is expected to be named Main.vm;
- one VM function in this file is expected to be named main

VM implementation conventions

- the stack starts in address 256 in the host RAM
- when the VM implementation starts running, or is reset, it starts executing an argument-less OS function named Sys.init
- Sys.init is designed to call Main.main, and then enter an infinite loop

These conventions are realized by the following code:

```
// Bootstrap code (should be
   written in assembly)
SP = 256
call Sys.init
```

In the Hack platform, this code should be put in the ROM, starting at address 0

Final remark

- Be sure that you know how to program in hack assembly language.
- Be sure that you know how to translate hack assembly codes into binary machine codes.
- Be sure that you know how to perform stack operations in VM. (VM abstraction)
- Be sure that you know how to program in VM code. (VM abstraction)
- Be sure that you know how to translate VM codes to hack assembly codes. (VM implementation)
- Make sure that you understand all the **examples**, **exercises** and **quizes** given in the lecture slides.