COMP1036 – Revision

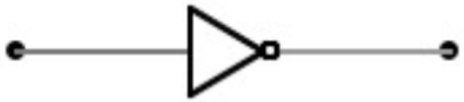
General Information

- Exam – 50% of the total mark
- 1 hour – 2 questions
- 17:00 29th Dec 2023 (TB-118 & TB-226)

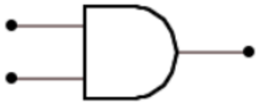
Part 1

Elementary Logic Gates

$A = \bar{A}$

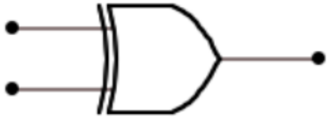


$A\ AND\ B = A \cdot B$

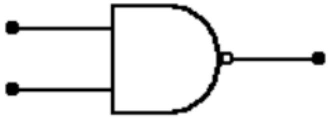


$A\ OR\ B = A + B$

$A\ XOR\ B = A \oplus B$



$A\ NAND\ B = \overline{A \cdot B}$

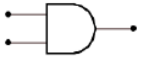


$A\ NOR\ B = \overline{A + B}$

Collection of Elementary Logic Gates

$A = \bar{A}$ 

A	Ā
0	1

$A\ AND\ B = A \cdot B$ 

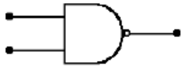
A	B	$A \bullet B$
0	0	0
0	1	0
1	0	0
1	1	1

$A\ OR\ B = A + B$ 

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

$A\ XOR\ B = A \oplus B$ 

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$A\ NAND\ B = \overline{A \cdot B}$ 

A	B	$\overline{A \bullet B}$
0	0	1
0	1	1
1	0	1
1	1	0

$A\ NOR\ B = \overline{A + B}$ 

A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Logic

All chips constructed from elementary logic gates

- Every chip can be built from a combination of:
  - AND
  - OR
  - NOT
  - No integration, division, differentiation…
  - “Canonical Representation”
- AND, OR and NOT can be built from NAND

- Therefore every possible chip can be built from just the NAND gates!!!!

George Boole, 1815-1864

(“*A Calculus of Logic*”)

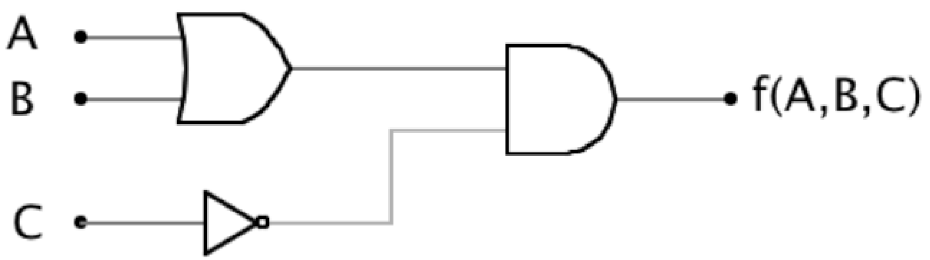
Boolean Function

- A Boolean function is a function that operates on binary inputs and return binary outputs
- Truth table is **every possible function evaluation** of the input variables
- [note 0 and 1 used to define false and true]
- Everything can be defined by a truth table

Composite Gates

$$f(A,B,C) = (A+B)\bullet\overline{C}$$

(A OR B) AND NOT C



A	B	C	*f*(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Precedence

- Precedence

    Parentheses evaluated first

    Then **Not**

    Then **And**

    Then **Or**

Not X Or Y And Z = (Not X) Or (Y And Z)

Not X And Y Or Z = ((Not X) And Y) Or Z

Brackets over-rule everything…use when in doubt

((Not (X)) And (Y)) Or (Z)

Laws of Boolean Algebra

1. Law of Identity

$A = A$ $\qquad$ $\overline{A} = \overline{A}$

2. Commutative Law

$A \cdot B = B \cdot A$ $\qquad$ $A + B = B + A$

3. Associative Law

$(A \cdot B) \cdot C = A \cdot (B \cdot C)$ $\qquad$ $(A + B) + C = A + B + C$

4. Idempotent Law

$A \cdot A = A$ $\qquad$ $A + A = A$

5. Double Negative Law

$\overline{\overline{A}} = A$

6. Complementary Law

$A \cdot \overline{A} = 0$ $\qquad$ $A + \overline{A} = 1$

7. Law of Intersection

$A \cdot 1 = A$ $\qquad$ $A \cdot 0 = 0$

8. Law of Union

$A + 1 = 1$ $\qquad$ $A + 0 = A$

9. Distributive Law

$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ $\qquad$ $A + (B \cdot C) = (A + B) \cdot (A + C)$

10. Law of Absorption

$A \cdot (A + B) = A$ $\qquad$ $A + A \cdot B = A$

11. Law of Common Identities

$A \cdot (\overline{A} + B) = AB$ $\qquad$ $A + (\overline{A} \cdot B) = A + B$

12. De Morgan's Law

$\overline{A \cdot B} = \overline{A} + \overline{B}$ $\qquad$ $\overline{A + B} = \overline{A} \cdot \overline{B}$

Simplify Boolean Expression

Not(Not(x) And Not(x Or y)) =

Not(Not(x) And (Not(x) And Not(y))) =

Not((Not(x) And Not(x)) And Not(y)) =

Not(Not(x) And Not(y)) =

Not(Not(x Or y))=

double negation

x Or y

Boolean Arithmetic

Binary to Decimal

- Each binary digit corresponds to a power of 2:

Place	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
Weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	$= 128$	$= 64$	$= 32$	$= 16$	$= 8$	$= 4$	$= 2$	$= 1$

- Where the digit is 1, we add the corresponding weight
- Example: convert $1100\ 1010_2$ into decimal

$$
\begin{aligned}
1100\,1010_2 = 1 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 \\
+ 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\
= 128 + 64 + 8 + 2 = 202_{10}
\end{aligned}
$$

Decimal to Binary

- Repeatedly divide by 2, until we reach 0
- The **right**/**left**-most binary digit is the **first**/**last** remainder
- E.g. $101_{10} = 1100101_2$

101	Remainder
50	1
25	0
12	1
6	0
3	0
1	1
0	1

- Example: convert 163_{10} into binary
- 10100011_2

Decimal to Binary (look-up table)

- 87 = 64 (64 = 2^6 , the biggest 2^n that 87 is divisible by) + 23 (reminder)
- 87 = 64 + 16 (16 = 2^4 , the biggest 2^n that 23 is divisible by) + 7 (reminder)
- 87 = 64 + 16 + 4 (4 = 2^2 , the biggest 2^n that 7 is divisible by) + 3 (reminder)
- 87 = 64 + 16 + 4 + 2 (2 = 2^1 , the biggest 2^n that 3 is divisible by) + 1 (reminder)
- 87 = 64 + 16 + 4 + 2 + 1(1 = 2^0 , the biggest 2^n that 1 is divisible by) + 0 (reminder)
- Stop when reminder = 0

Representing Negative Numbers

- So far, unsigned numbers
  - How are negative numbers represented on a computer?
- What we use in decimal notation
  - +/− and 0, 1, 2, · · ·
- Such a representation is called **sign and magnitude**
- For binary numbers – define **leftmost** bit to be the **sign**
  - $0 \Rightarrow +$, $1 \Rightarrow -$
  - Rest of bits can be numerical value of number
  - Hence, only seven bits are left in a byte (apart from the sign bit), the magnitude can range from 0000000 (0) to 1111111 (127)
- Problems?

One's Complement

- Alternatively, a system known as **one's complement** can be used to represent negative numbers
- A negative binary number is the bitwise **NOT** applied to it — the "**complement**" of its positive counterpart
- E.g. the ones' complement form of 00101011 (43_{10}) becomes 11010100 (-43_{10})
- Still has two representations of 0: 00000000 (+0) and 11111111 (−0)
- The range of signed numbers using one's complement is represented by $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$ and ± 0
  - A conventional eight-bit byte is -127_{10} to $+127_{10}$ with zero being either 00000000 (+0) or 11111111 (−0)

Excess-n

- **Excess-n**, also called offset binary or biased representation, uses a pre-specified number *n* as a biasing value
- A value is represented by the unsigned number which is *n* greater than the intended value
- Therefore 0 is represented by *n*, and −*n* is represented by the all-zeros bit pattern
- E.g. Excess-3
  - 0 is represented by 0011 (3)
  - +1 is represented by 0100 (4), +2 is represented by 0101(5)…
  - -1 is represented by 0010 (2), -2 is represented by 0001 (1)
  - -3 is represented by 0000 (0)

Two’s Complement

- The **two's complement** of an N-bit binary number is defined as the complement with respect to 2^N
  - It is the result of subtracting the number from 2^N
  - -x is represented as 2^N-x
- There’s a quicker way to calculate 2^N-x:
  - x + (1’s complement of x) = 2^N-1 (all 1 bits)
  - 2^N-x = (1’s complement of x) +1
  - Take the bitwise inverse (**NOT**) of x, then add 1 to result
- An N-bit two's-complement numeral system can represent every integer in the range $-(2^{N-1})$ to $+(2^{N-1} - 1)$
  - One’s complement: $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$
- The sum of a number and its two's complement will always equal 0 (the last digit is ignored)
  - The sum of a number and its one's complement will always equal -0 (all 1 bits)

Example of 4-Bit Signed Encodings

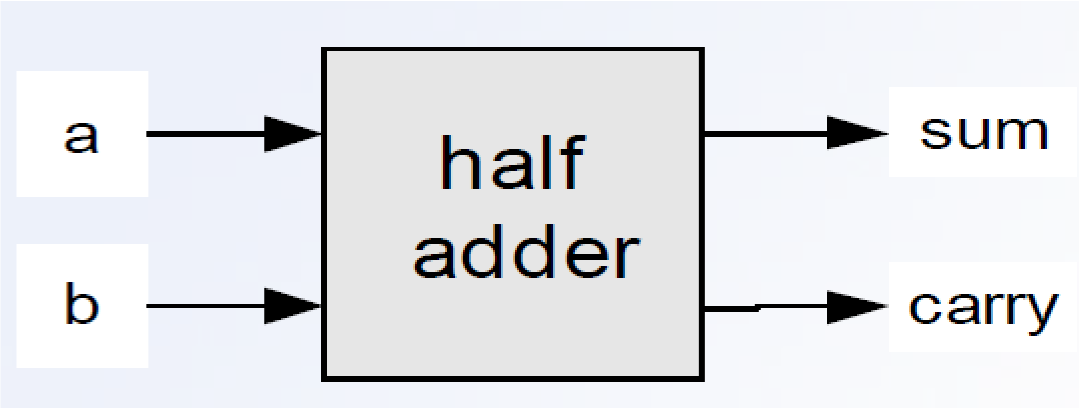
Sign and Mag.		Ones' Comp.		Excess-3		Two's Comp.	
1111	−7	1000	−7	0000	−3	1000	−8
1110	−6	1001	−6	0001	−2	1001	−7
1101	−5	1010	−5	0010	−1	1010	−6
1100	−4	1011	−4	0011	0	1011	−5
1011	−3	1100	−3	0100	+1	1100	−4
1010	−2	1101	−2	0101	+2	1101	−3
1001	−1	1110	−1	0110	+3	1110	−2
1000	−0	1111	−0	0111	+4	1111	−1
0000	+0	0000	+0	1000	+5	0000	0
0001	+1	0001	+1	1001	+6	0001	+1
0010	+2	0010	+2	1010	+7	0010	+2
0011	+3	0011	+3	1011	+8	0011	+3
0100	+4	0100	+4	1100	+9	0100	+4
0101	+5	0101	+5	1101	+10	0101	+5
0110	+6	0110	+6	1110	+11	0110	+6
0111	+7	0111	+7	1111	+12	0111	+7

Adder

- Build an Adder:
  - Half adder: adds two bits
  - Full adder: adds three bits
  - Adder: adds two integers

Half Adder

- Add **two** single binary digits and provide the **output** plus a **carry value**
- It has two inputs, called A(a) and B(b), and two outputs S (sum) and C (carry)



Half Adder

- Least significant bit in the addition is called sum (a+b)
- Most significant bit is called carry (carry of a+b)

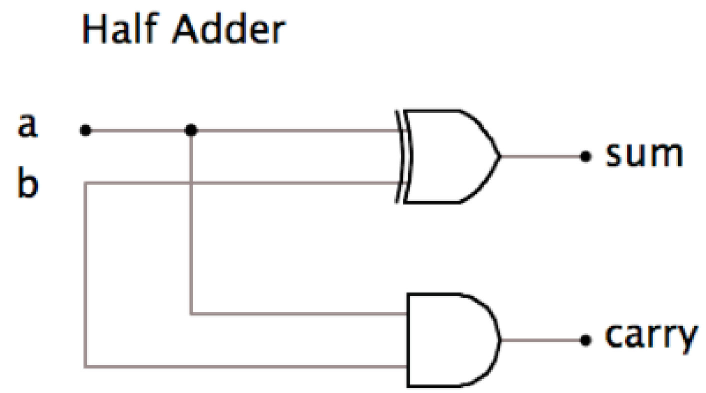
a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- **Never has a situation when sum and carry are both 1**

Half Adder

a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- The common representation uses a XOR and a AND gate



Full Adder

- Add **three** single binary digits and provide the **output** plus a **carry value**
- It has three inputs, called A, B and Carry(in), and two outputs S (sum) and Carry(out)

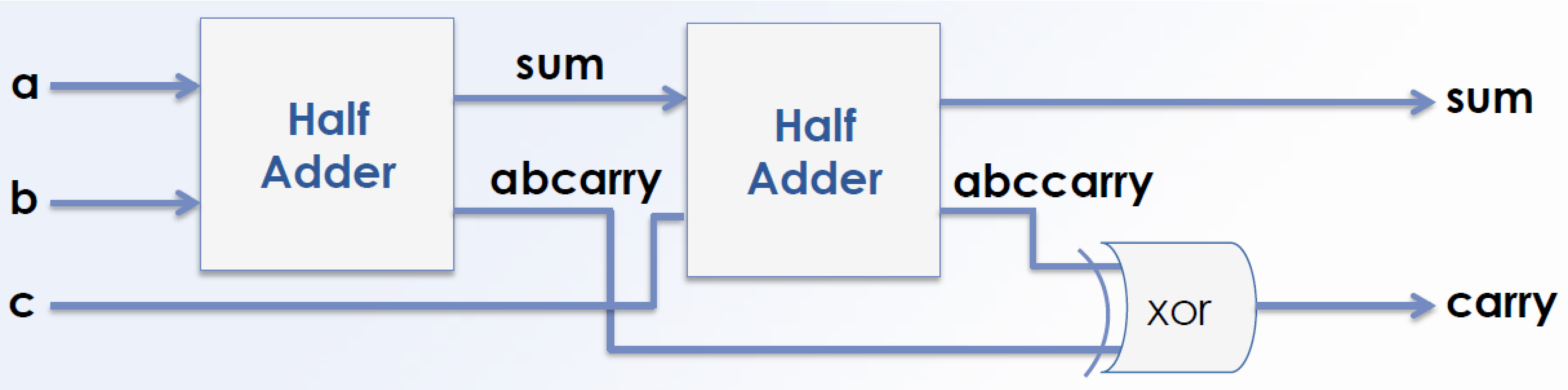
Full Adder

- Least significant bit in the addition is called sum (a+b+c_in)
- Most significant bit is called carry(out) (carry of a+b+c_in)

a	b	Carry(in)	Carry(out)	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder: Implementation

- Use two half adders to build a full adder



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Sequential Logic and ALU

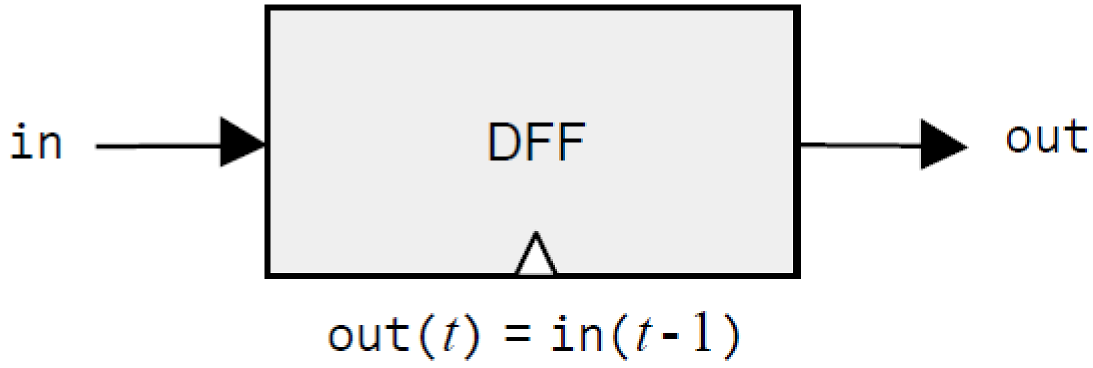
Sequential Logic Circuits

- Combinational chips compute functions that **depend solely on combinations of their input values**

- Sequential Logic Circuits
  - Output depends **not only on the present value** of its input signals but **on the sequence of past inputs**, the input history as well

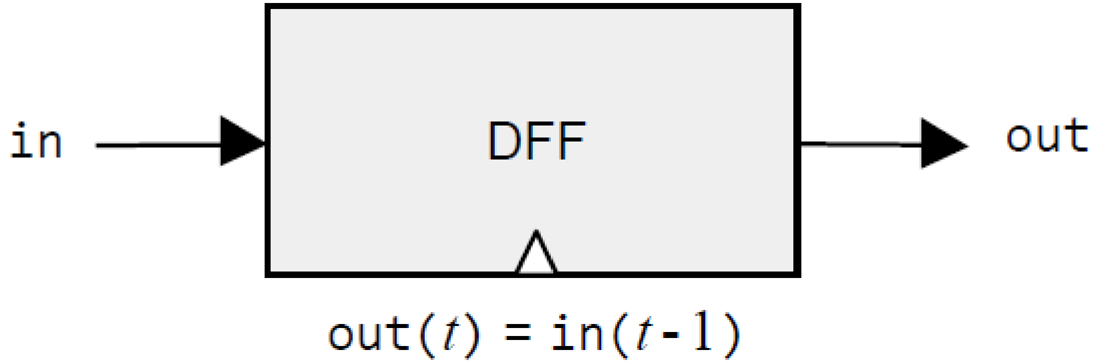
Flip Flops

- The flip flop is the most elementary sequential element in the computer
- Data Flip Flop (DFF): the simplest state keeping gate (built-in)



- Contains a **single** bit **input** and a **single** bit **output**

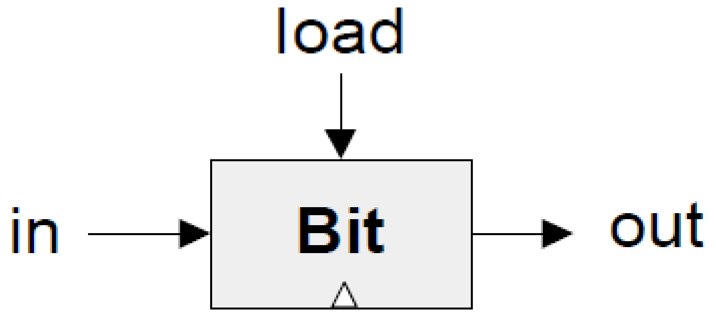
Flip Flops



- The gate outputs its previous input: out(t)= in(t-1)
- Implementation: a gate that can flip between two stable states:
  - Remembering 0/Remembering 1
  - Also can be made from looping NAND gates

Register

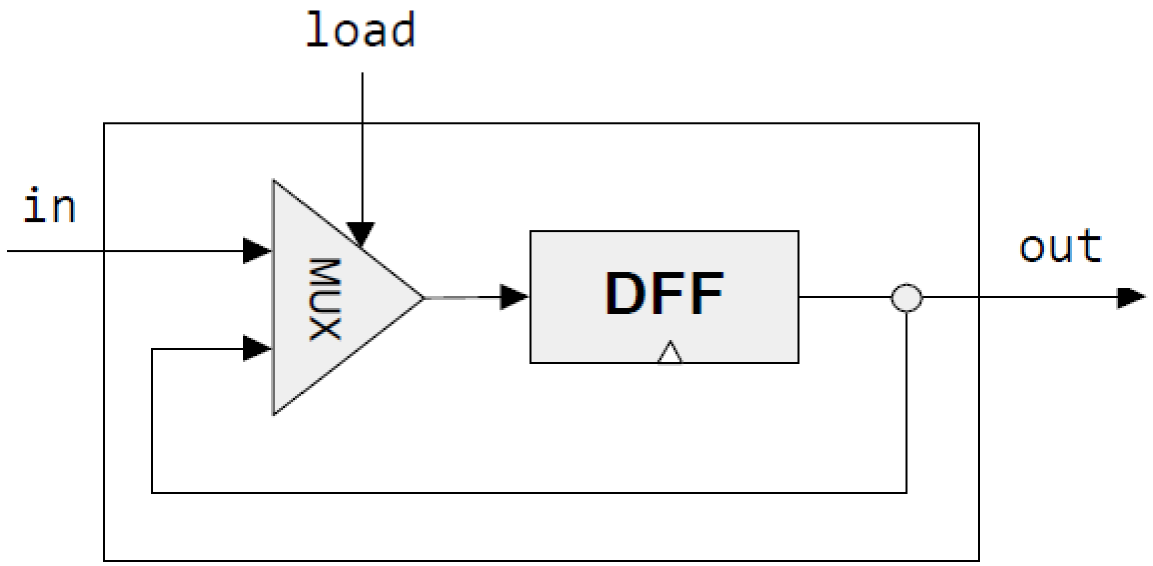
- A register is a storage device that can "**store**" or "**remember**" a value over time
- Typically is composed of flip flops
- 1-bit register:
  - Store (maintain) a bit
  - Until it is instructed to load(store) another bit



$$
\begin{array}{ll}
\text{if } \texttt{load}(t) \text{ then} & \texttt{out}(t+1) = \texttt{in}(t) \\
\text{else} & \texttt{out}(t+1) = \texttt{out}(t)
\end{array}
$$

1-bit Register: Implementation

- The select bit of the Mux can become the load bit!

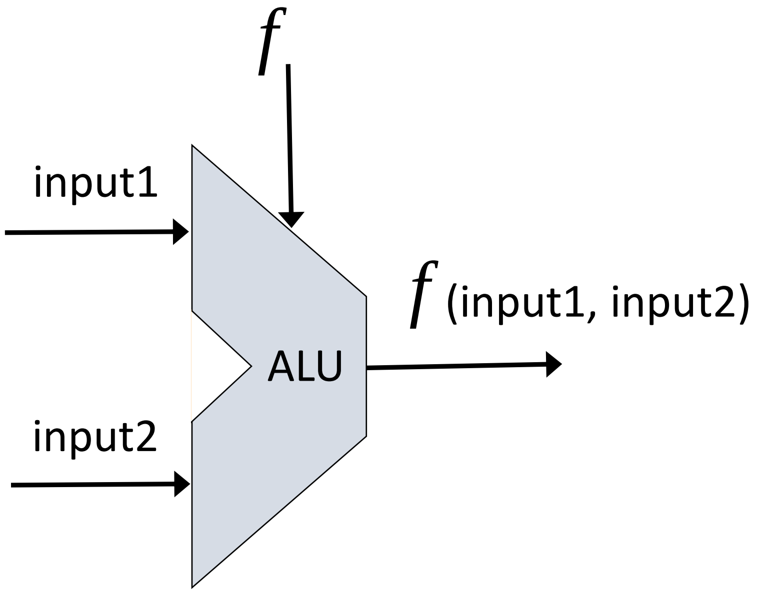


Arithmetic Logical Unit

- **A combinational circuit** that performs arithmetic and bitwise operations on integers represented as binary numbers.
- Input the **data** and some **code for the operation**
- Output will be some **data** and any **additional information**
- ALUs perform simple functions, because of this they can be executed at high speeds (i.e., very short propagation delays)

The Arithmetic Logical Unit

The ALU computes a function on the two inputs, and outputs the result



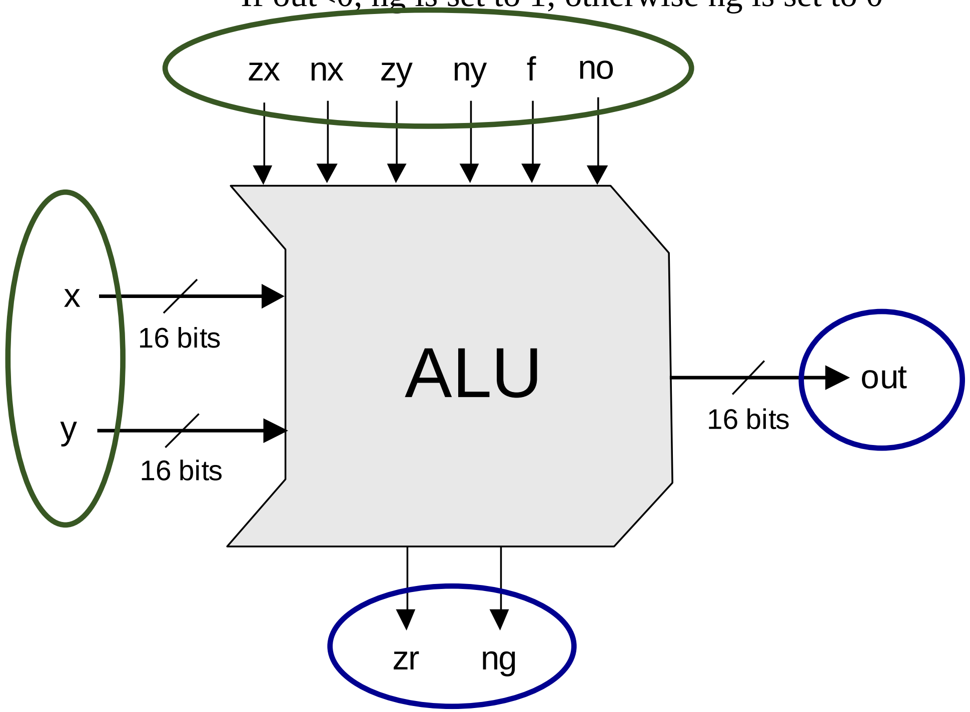
f : one out of a family of pre-defined arithmetic and logical functions

- Arithmetic functions: integer addition, multiplication, division, ...
- logical functions: And, Or, Xor, …

Which functions should the ALU perform?
A hardware / software tradeoff.

The Hack ALU

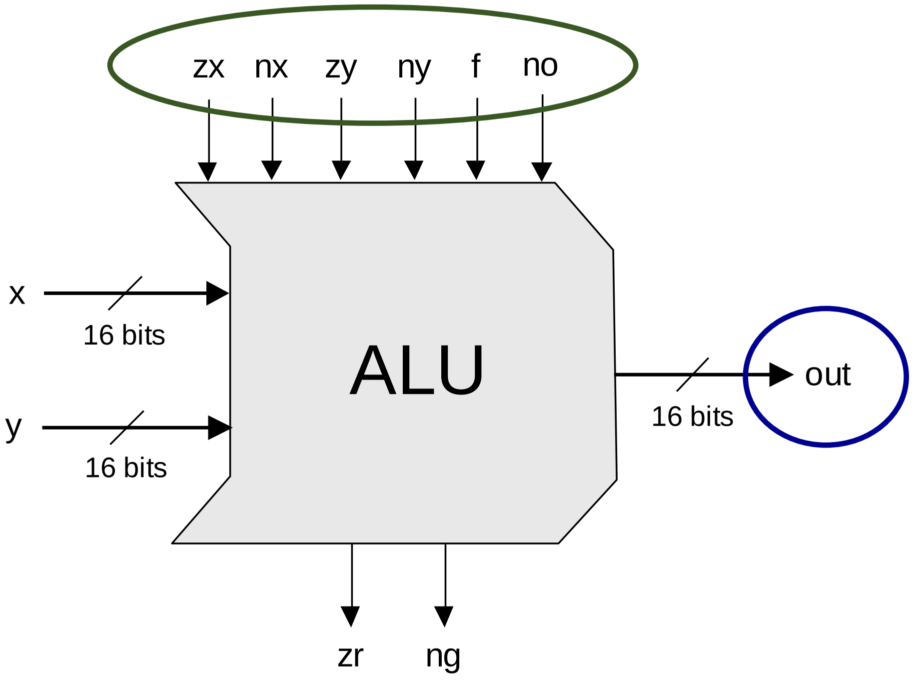
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Which function to compute is set by six 1-bit inputs
- Computes one out of a family of 18 functions
- Also outputs two 1-bit values
  - if the ALU output is 0, zr is set to 1; otherwise zr is set to 0
  - If out<0, ng is set to 1; otherwise ng is set to 0



out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

The Hack ALU

To cause the ALU to compute a function, set the control bits to one of the binary combinations listed in the table.



control bits

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Memory

Fetch-Decode-Execute Cycle

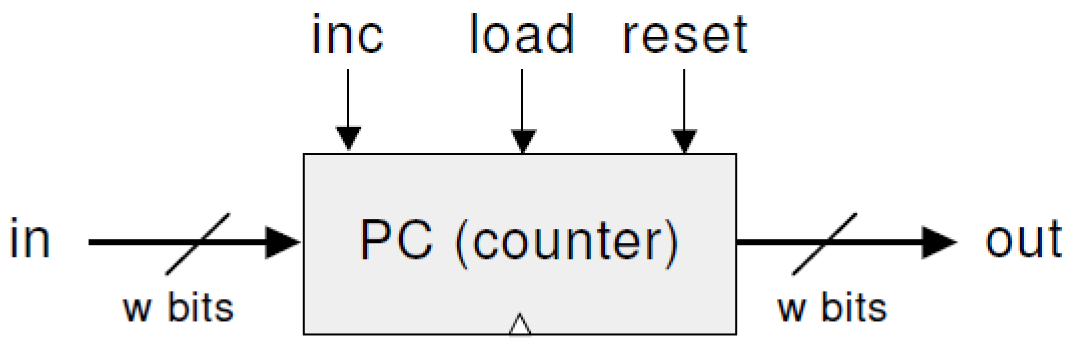
- At some level, every programmable processor implements a **fetch-execute cycle**
- Automatically implemented by processor hardware, allows processor to move through program steps
- Fetch — The opcode for the instruction is fetched from memory
- Decode — Opcode decoded to work what parts of the CPU are needed
- Execute — CPU processes the instruction
- And repeat for the next instruction

Fetch-Execute Algorithm

```
Repeat {
  Fetch (PC):
    • Fetch the instruction word (at PC)
    • Instruction decoded
    • Calculate next instruction address
  Execute (ALU, Registers and Control):
    • Read operands
    • Executes the operations
    • Write/store results
}
```

Program Counter

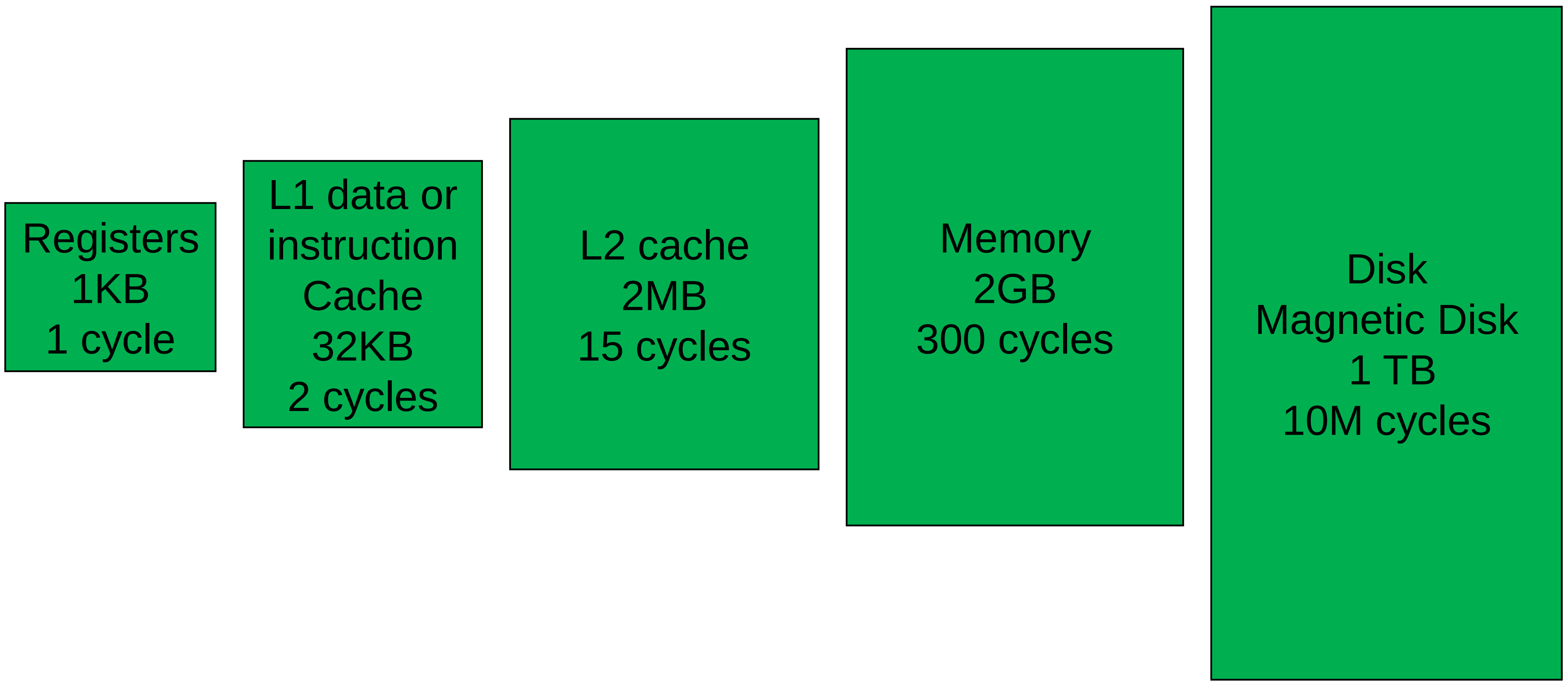
- Program counter emits the address of the next instruction.
  - To start/restart the program execution: PC=0
  - No jump: PC++
  - Unconditional jump: PC=A
  - Conditional jump: if (cond.) PC=A else PC++
- Implementation requirements:
  - Counting
  - Put counter in correct working mode based on the 3 different control bits




```
If reset(t-1) then out(t)=0
   else if load(t-1) then out(t)=in(t-1)
        else if inc(t-1) then out(t)=out(t-1)+1
             else out(t)=out(t-1)
```

Memory Hierarchy

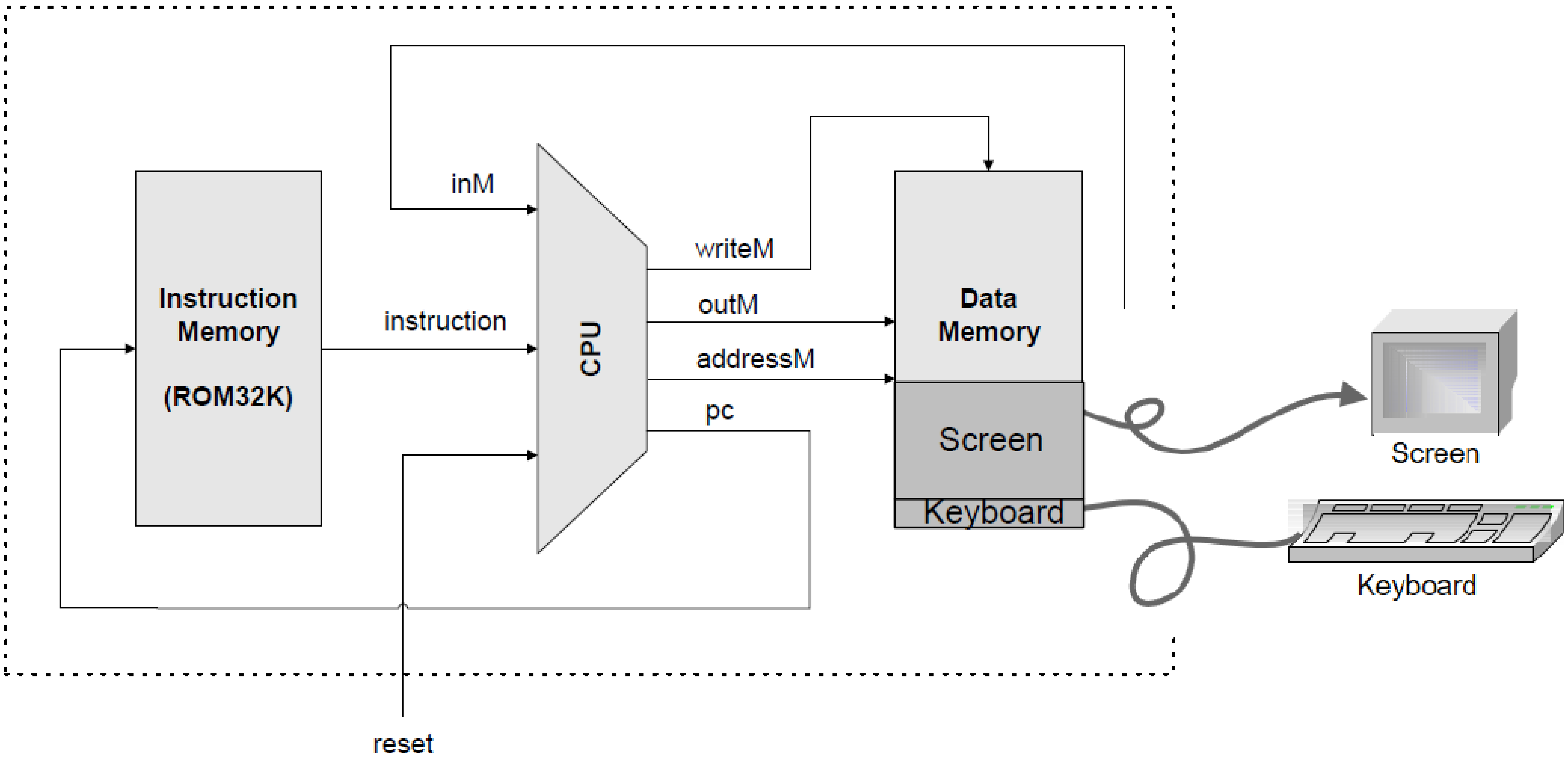
- As it goes further, capacity and latency increase



The Hack Computer: Main Parts

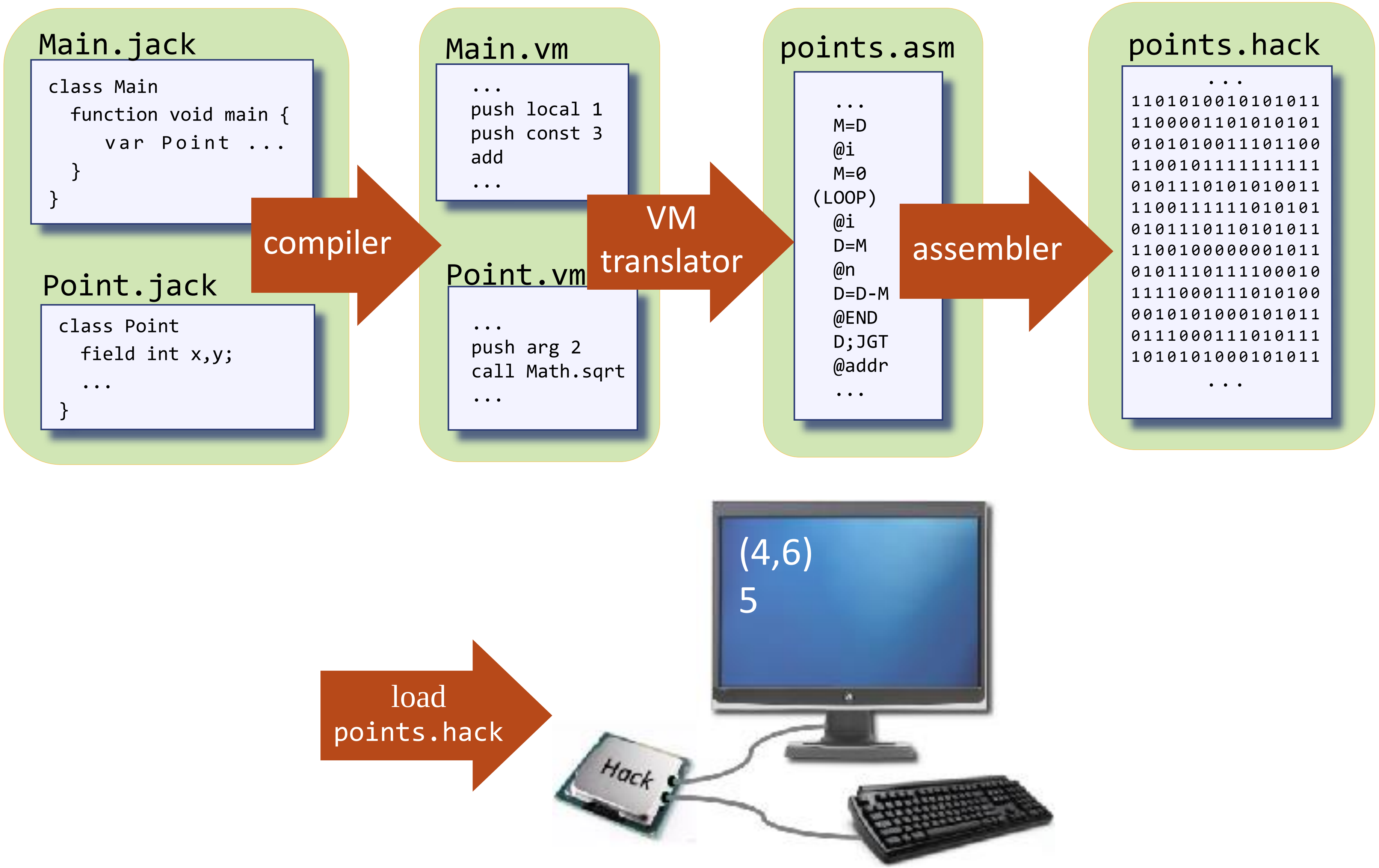
- Instruction memory (ROM)
- Memory (RAM)
  - Data memory
  - Screen (memory map)
  - Keyboard (memory map)
- CPU
- Computer (the logic that holds everything together)

The Hack Computer (Put Together)



Part 2

Big picture



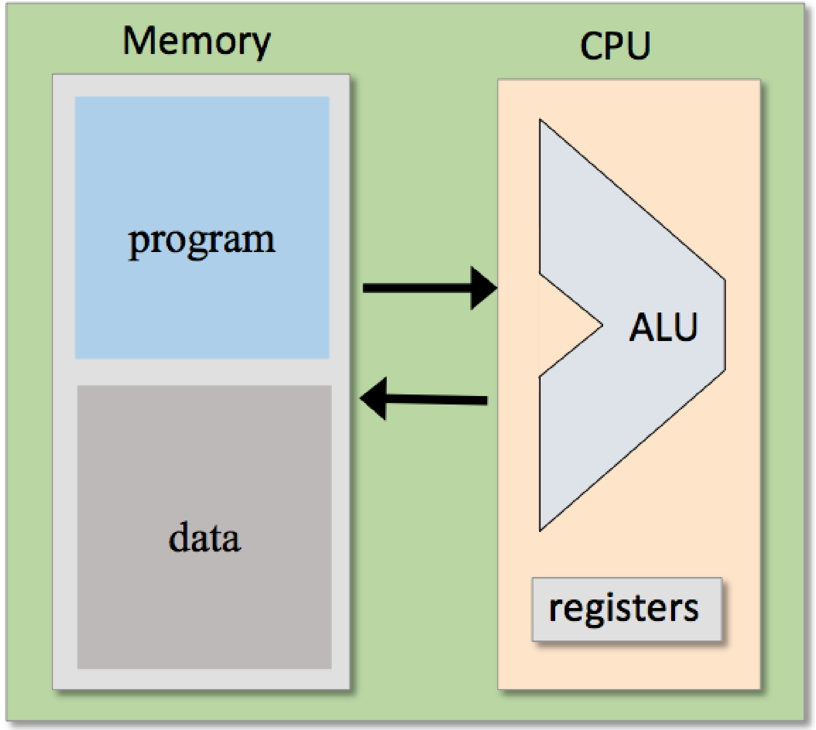
Main.jack
class Main
function void main {
var Point ...
}
}
Point.jack
class Point
field int x,y;
...
}
compiler
Main.vm
...
push local 1
push const 3
add
...
Point.vm
...
push arg 2
call Math.sqrt
...
VM translator
points.asm
...
M=D
@i
M=0
(LOOP)
@i
D=M
@n
D=D-M
@END
D;JGT
@addr
...
assembler
points.hack
...
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

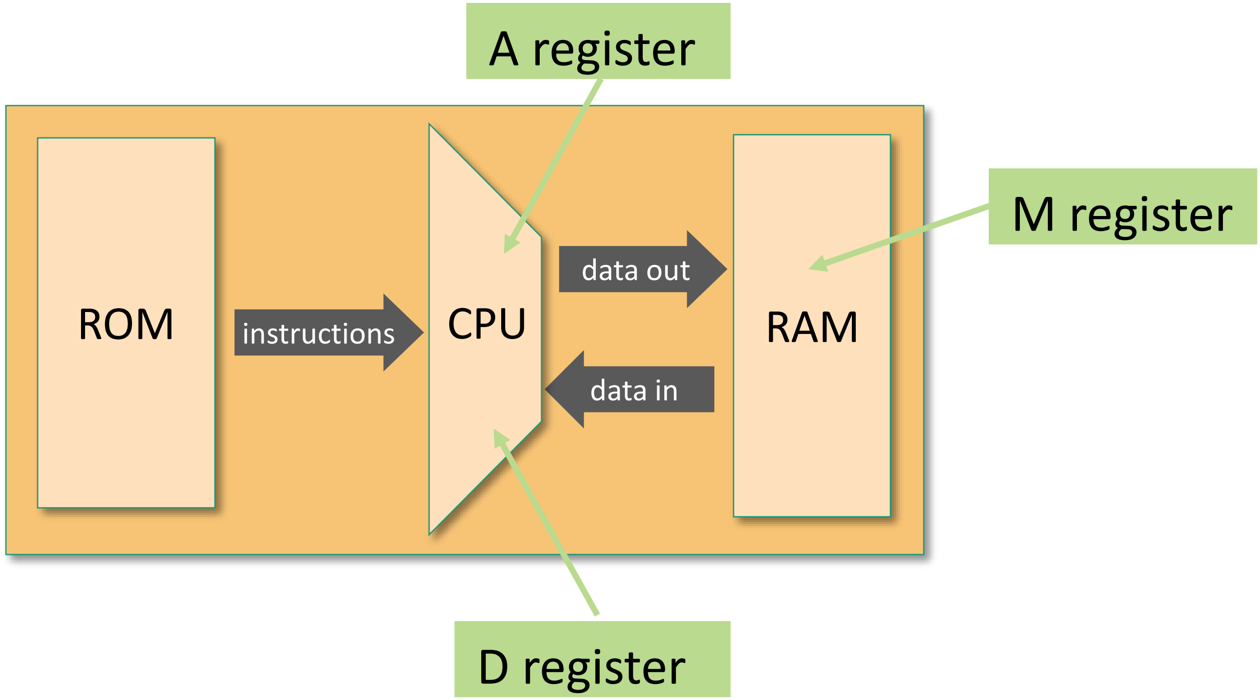
- A *machine language* can be viewed as an agreed-upon formalism, designed to manipulate a *memory* using a *processor* and a set of *registers*. (Nisan & Schocken)



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 ≤ 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`

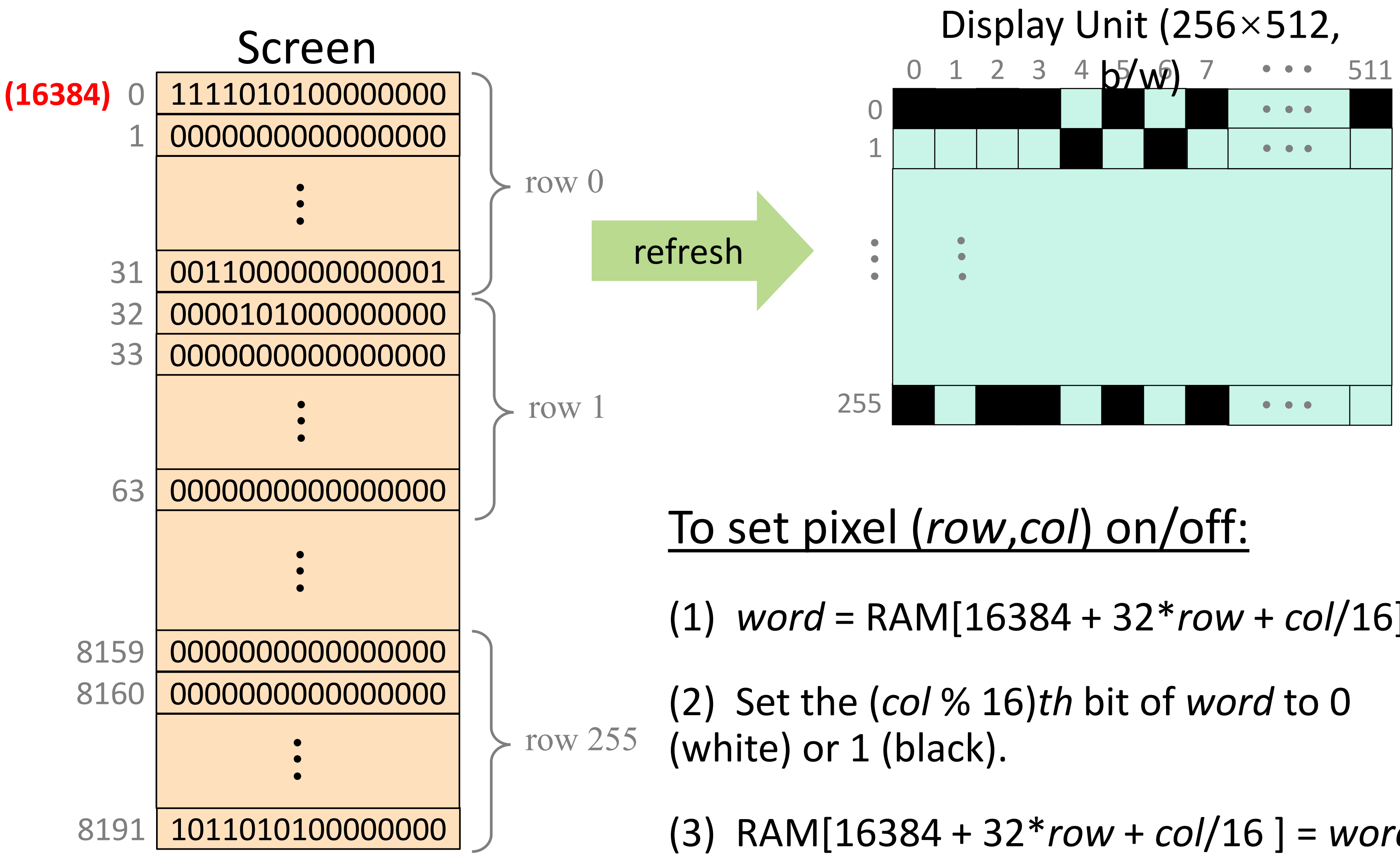
opcode | not used | *comp* bits | *dest* bits | *jump* bits

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

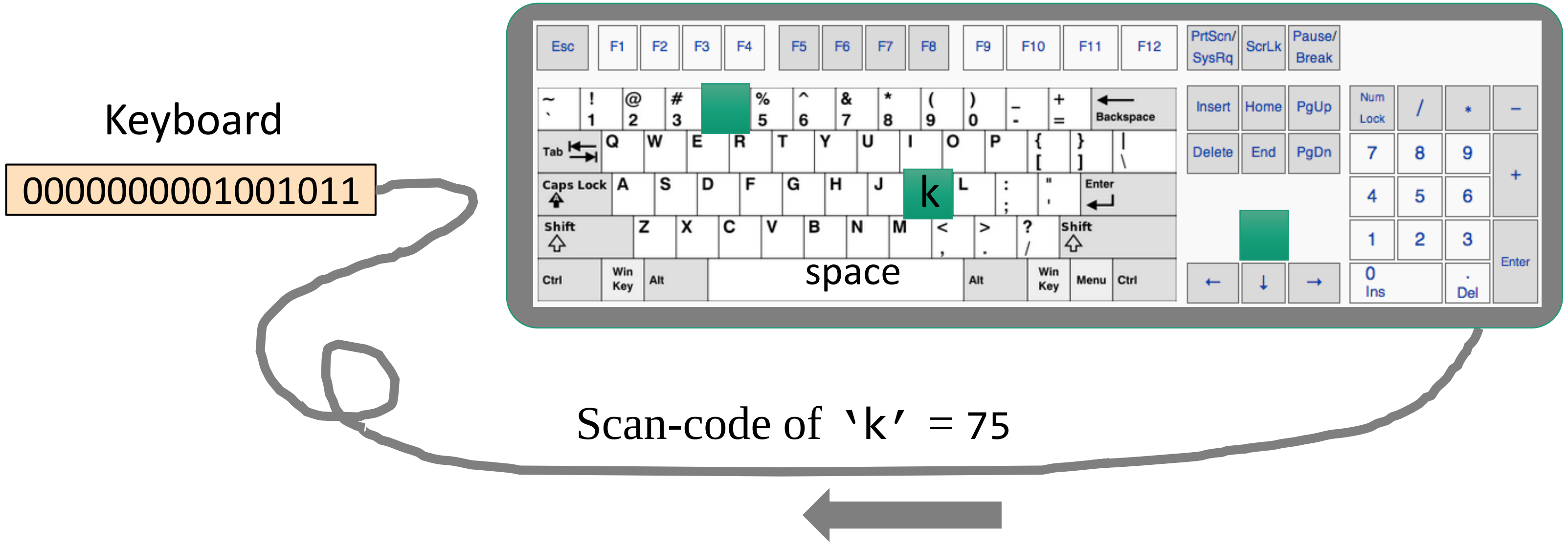
<u>To set pixel (*row*,*col*) on/off:</u>

(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].

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

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

  @R1
  M=0    // RAM[1]=0
  @END
  0;JMP  // goto end

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

(END)
  @END // end
  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	

<u>Implications:</u>

- 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]

// 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
```

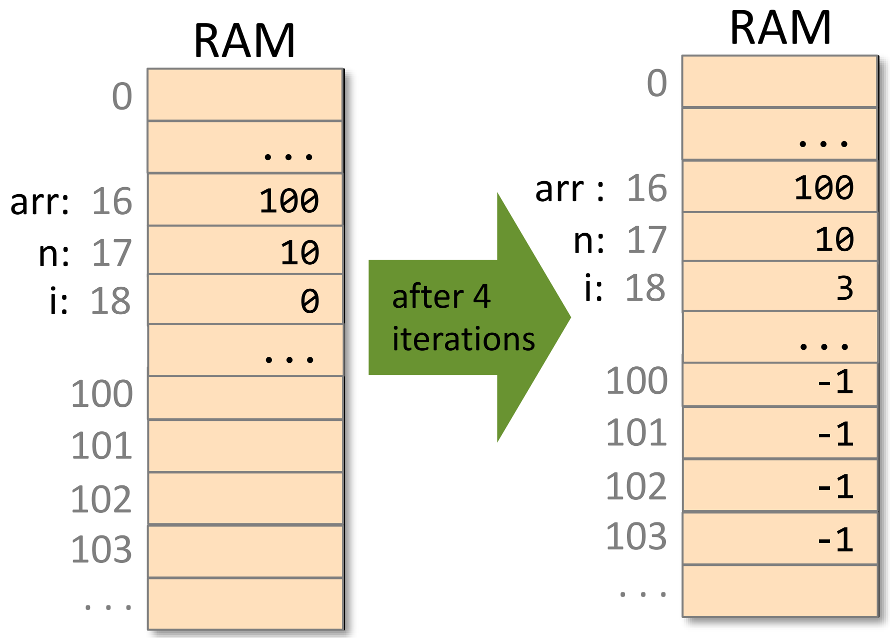
```
   @sum
   D=M // D = sum
   @i
   D=D+M // D = sum + i
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP // goto LOOP
(STOP)
   @sum
   D=M // D = sum
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP // end
```

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

<u>Symbolic syntax:</u>

@ *value*

Examples:

@21

@foo

Where *value* is either

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

<u>Binary syntax:</u>

0 *valueInBinary*

Example:

0000000000010101

<u>Translation to binary:</u>

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

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

Label declaration: (*label*)

Variable declaration: @*variableName*

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

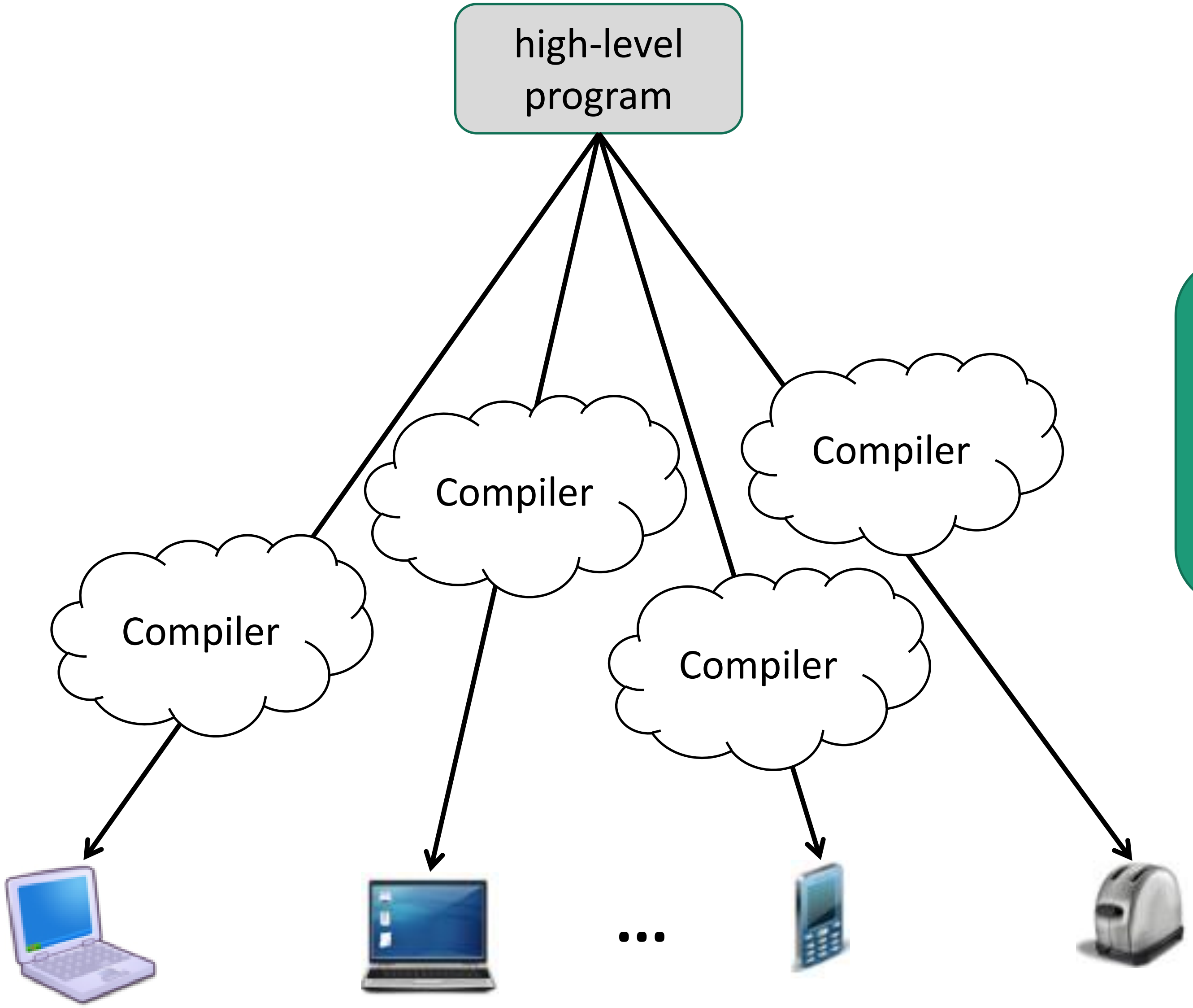
Outlines

- Hack assembly programming
- Assembler
- Virtual machine

Why we need virtual machine?

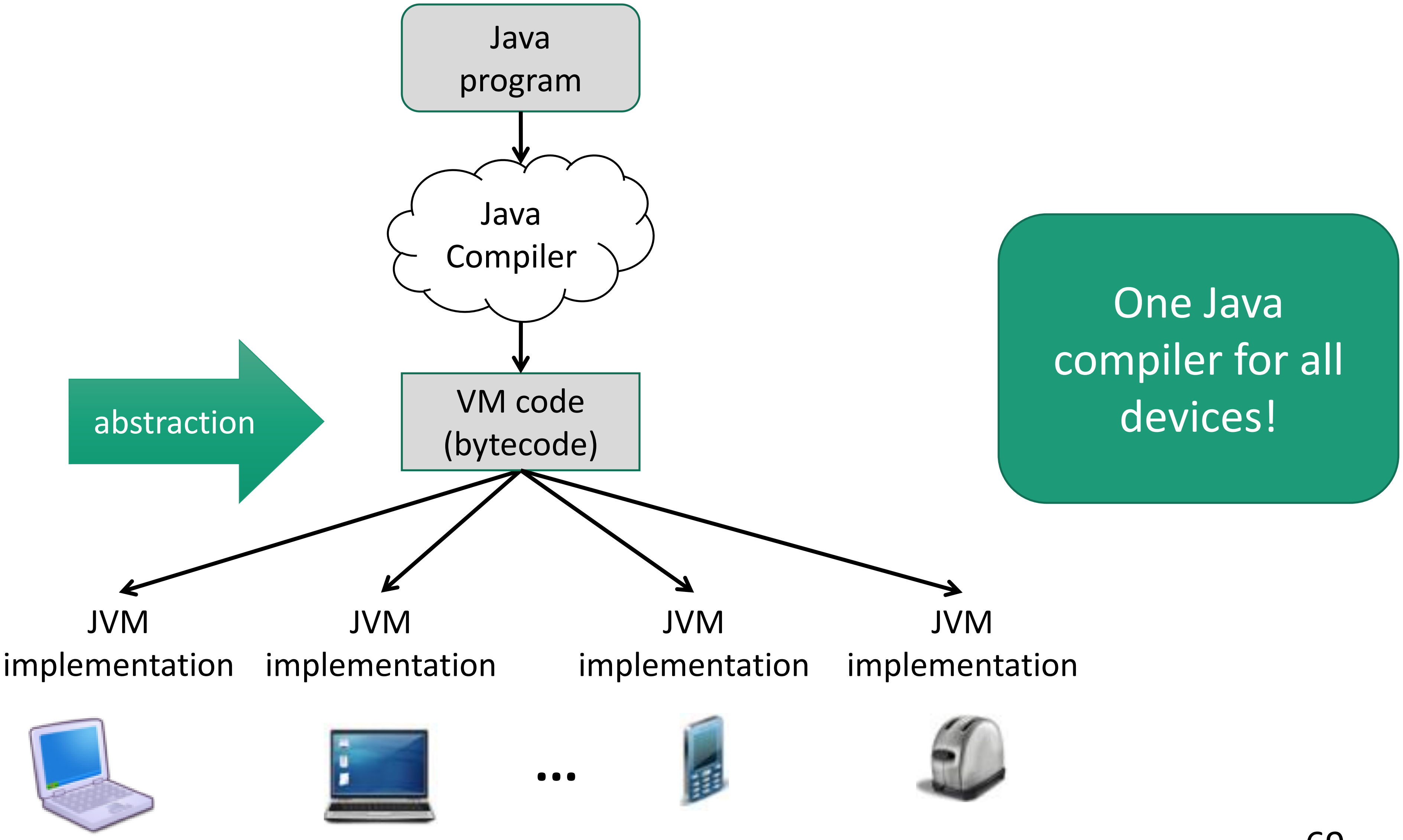
- **Code transportability**
  - ➢**Many** high-level languages can work on the same platform: virtual machine.
  - ➢VM may be implemented with relative ease on **multiple** target platforms.
  - ➢As a result, VM-based software can run on **many** processors and operating systems without modifying source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

Program compilation: 1-tier



One compiler for each device!

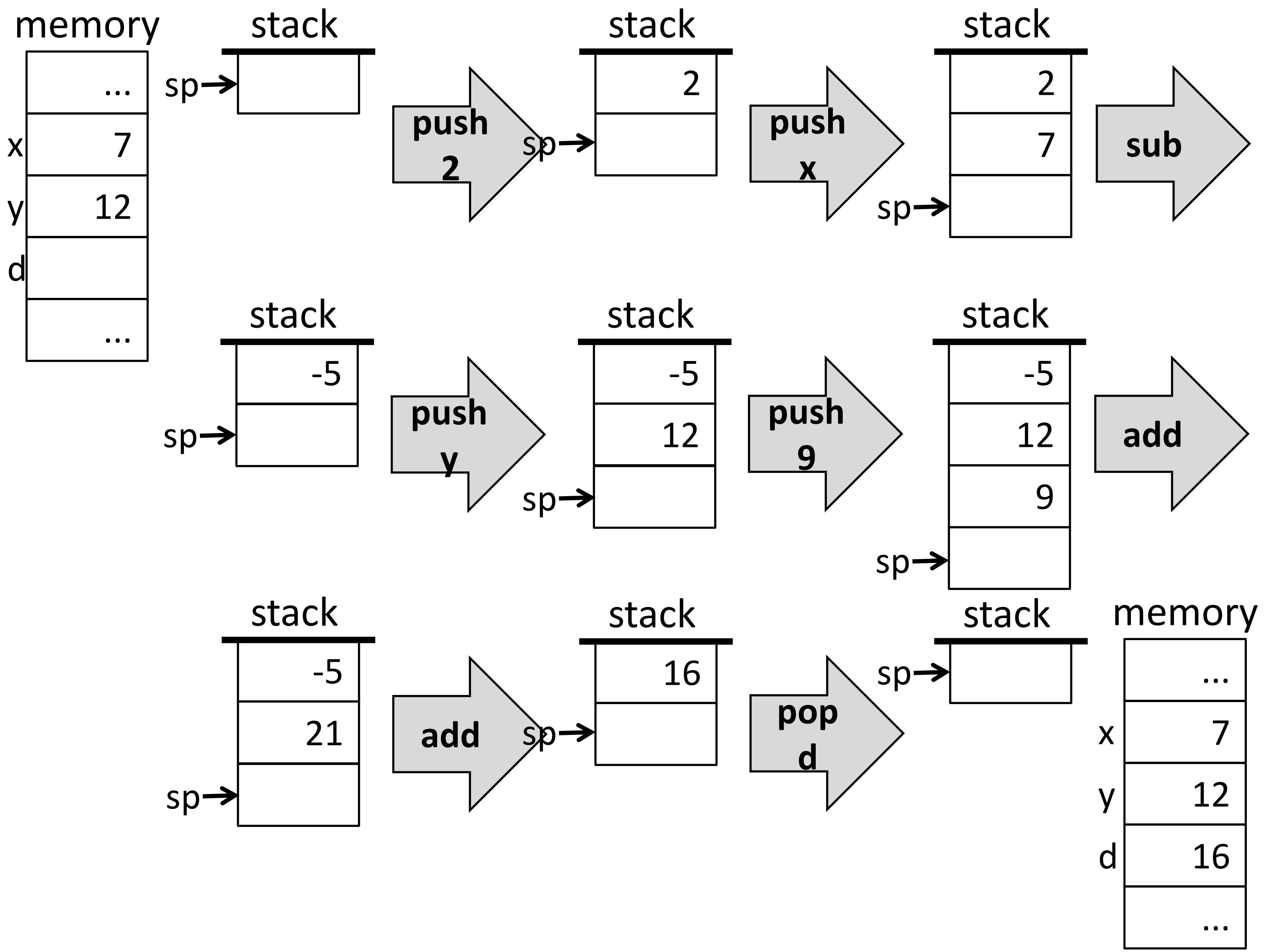
Program compilation: 2-tier



Arithmetic commands

VM code

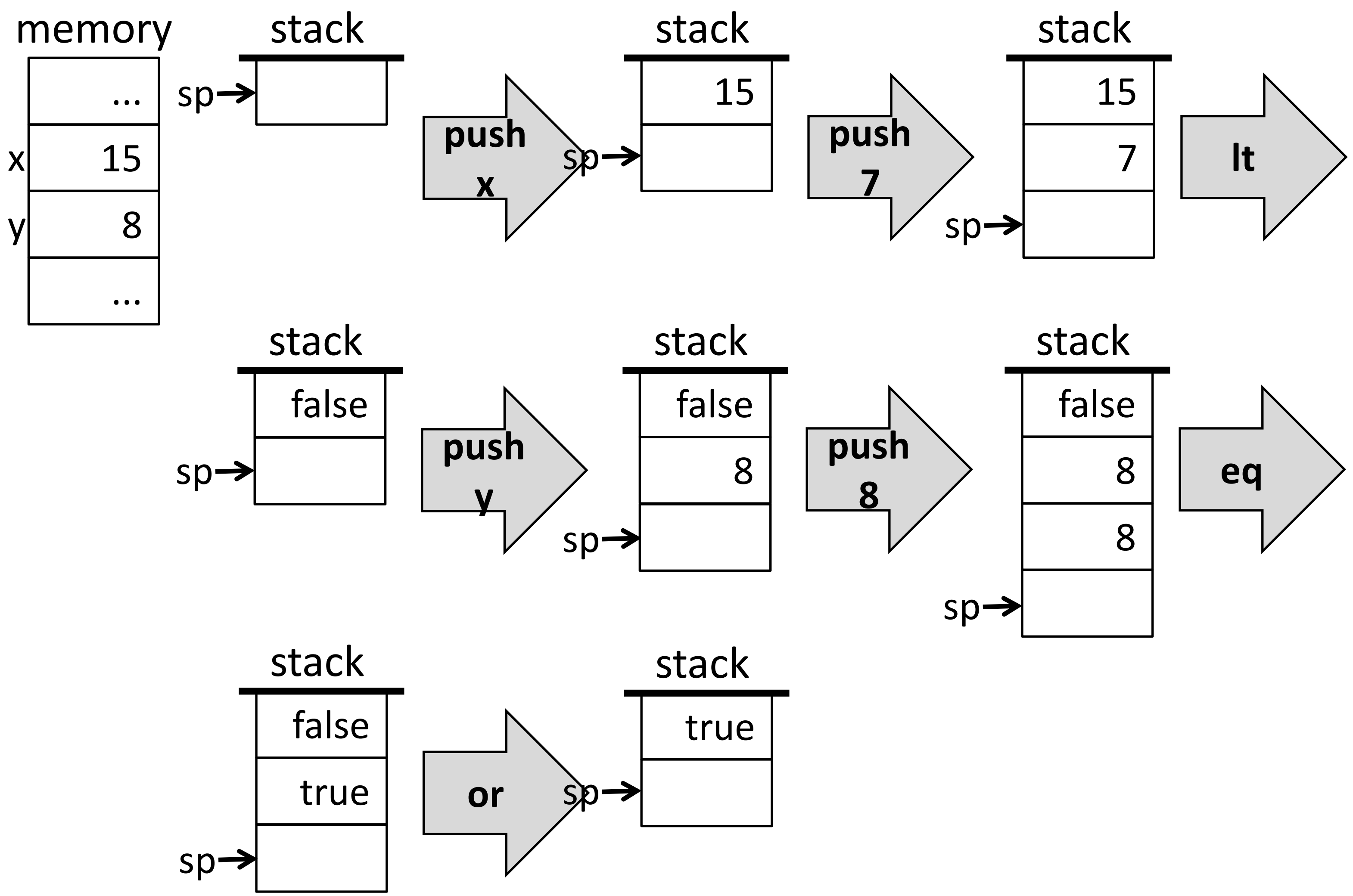
```
// d=(2-x) +
// (y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



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.

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
```

RAM

0	257	p
1	1024	q
2	1765	
...	...	
256	19	
257	23	
258	903	
...	...	
1024	5	
1025	12	
1026	-3	
...	...	

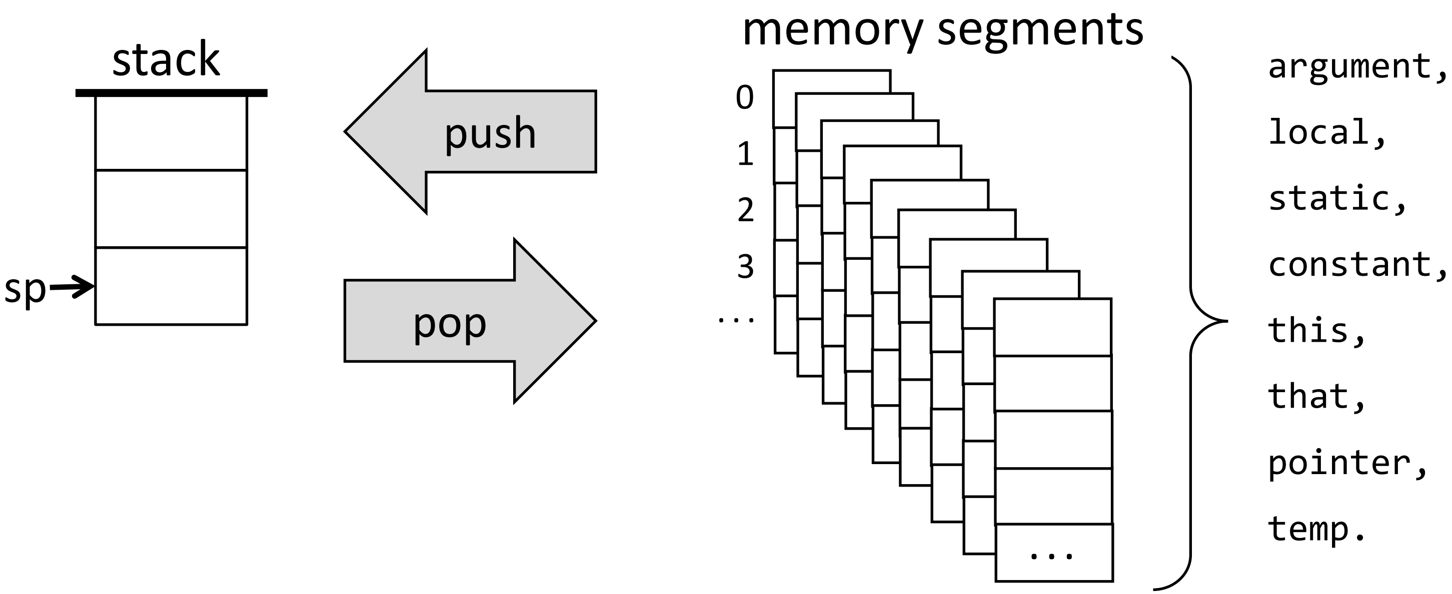
Notation:

*p // the memory location that p points at

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

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

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.

Implement **push constant** *i*

VM code:

push constant *i*

VM Translator

Assembly psuedo code:

*SP = *i*, SP++

(no pop constant operation)

Hack assembly:

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

<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
```

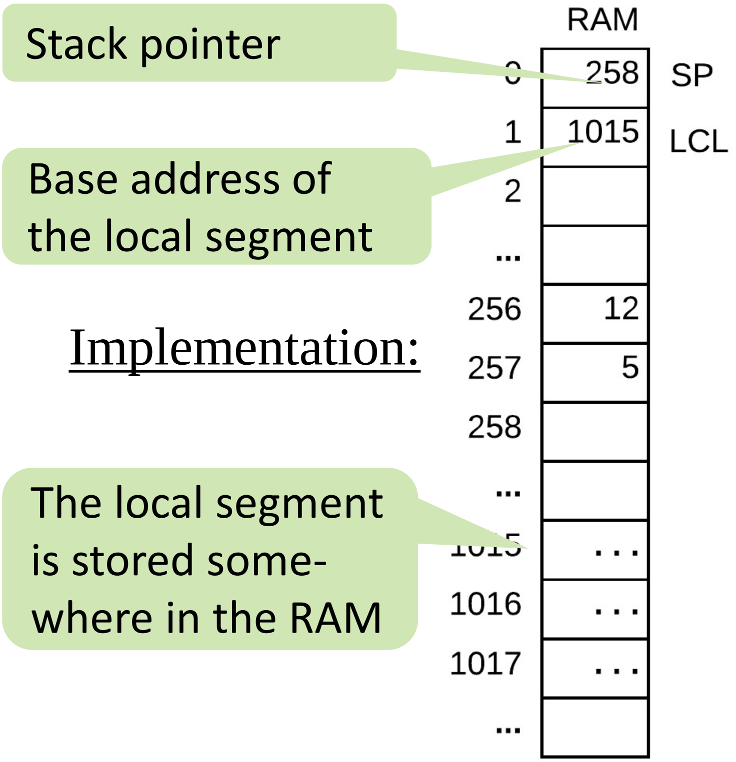
```
push local i
```

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

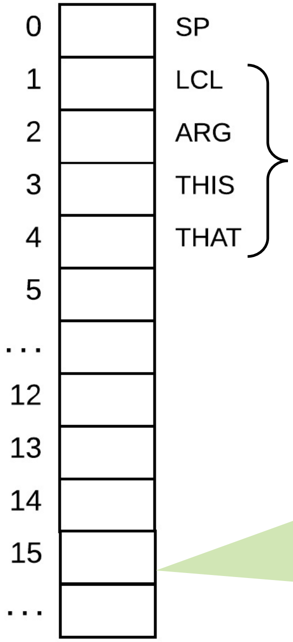
Assembly pseudo code:

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

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

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

Host RAM



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 temp** *i*

VM code:

```
push temp i
```

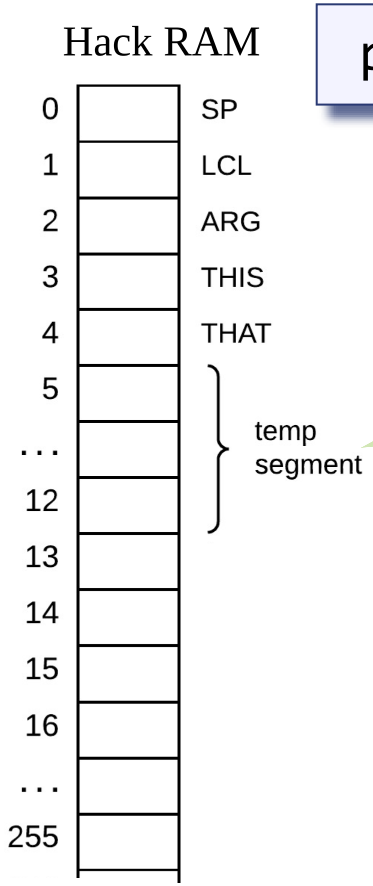
```
pop temp i
```

VM Translator

Assembly psuedo code:

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

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



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
```

```
pop pointer 0/1
```

VM Translator

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
- Implementation (VM translation):
  - The assembly language has similar branching commands.

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

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

<u>Handling call:</u>

- 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

<u>Handling function:</u>

- 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
```

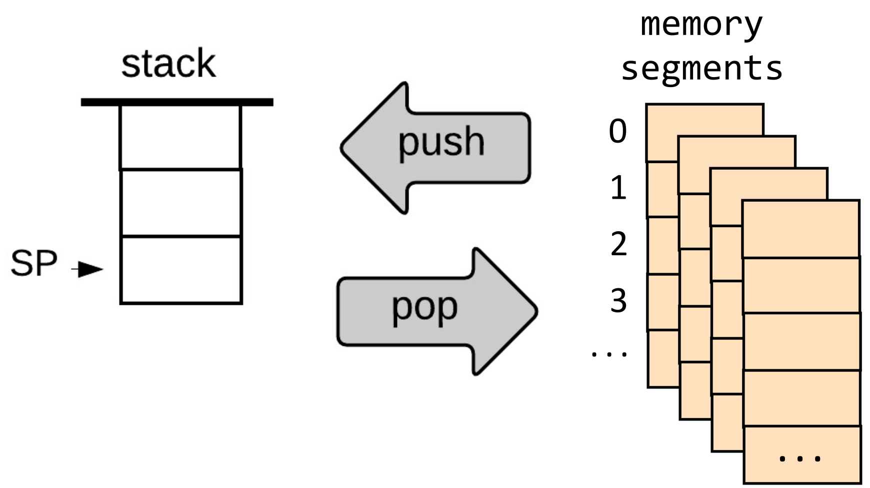
callee

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.

The function's state



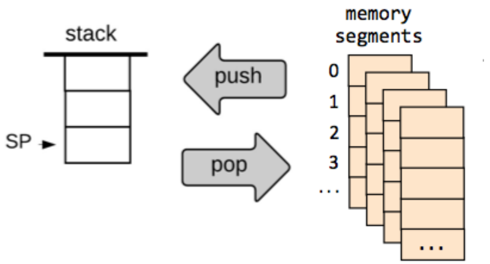
<u>During run-time:</u>

- 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

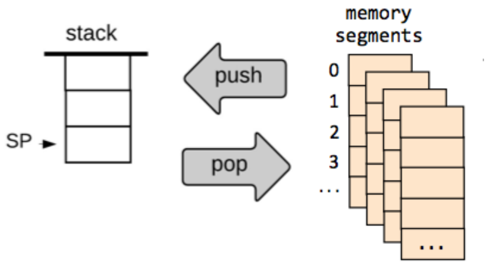
```
function main 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   return
```

caller




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

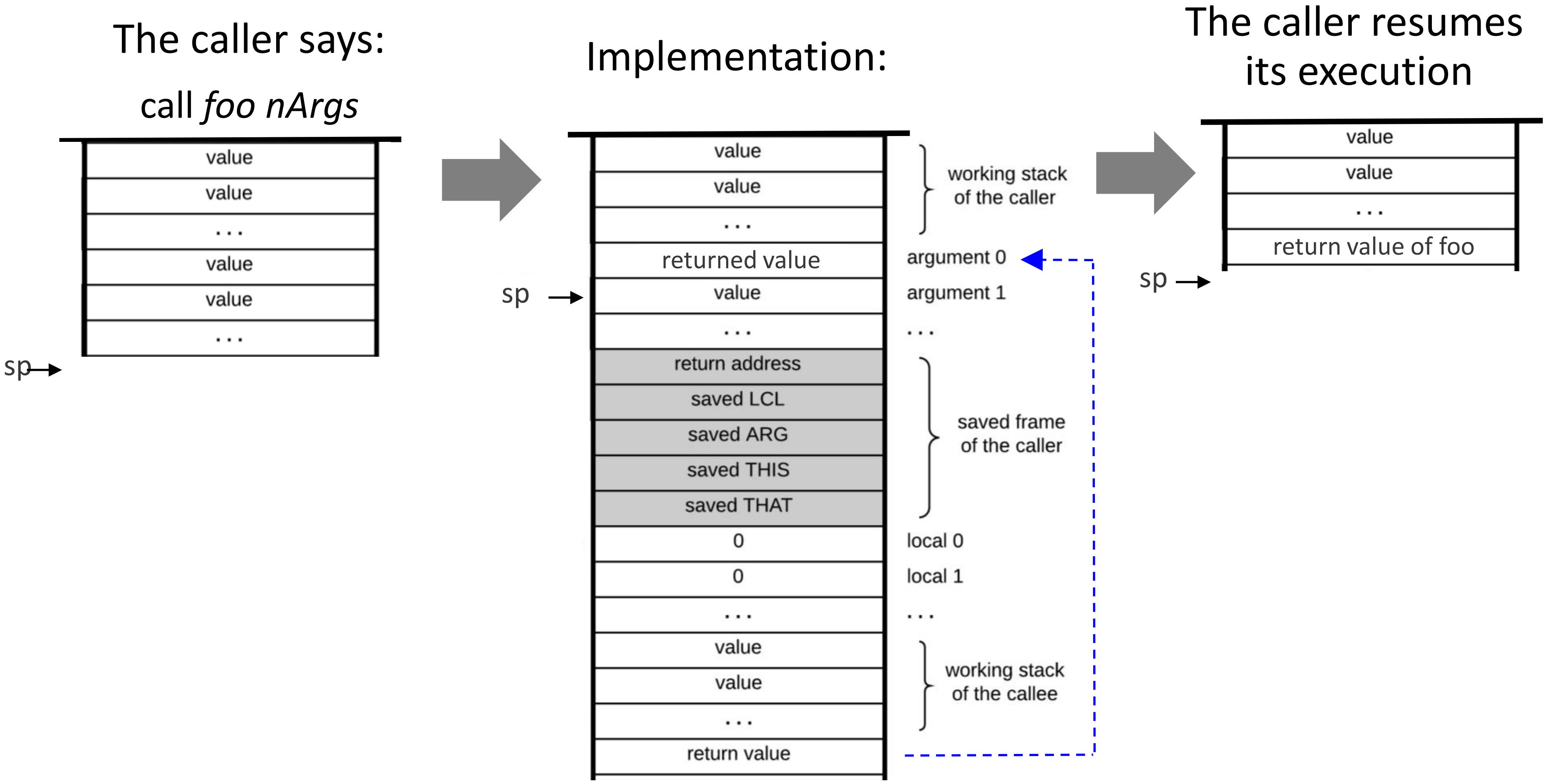
callee



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



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.