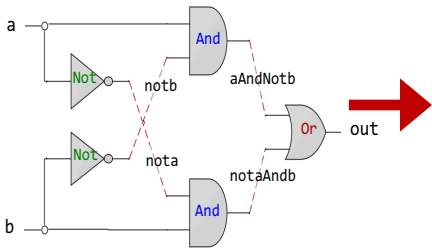
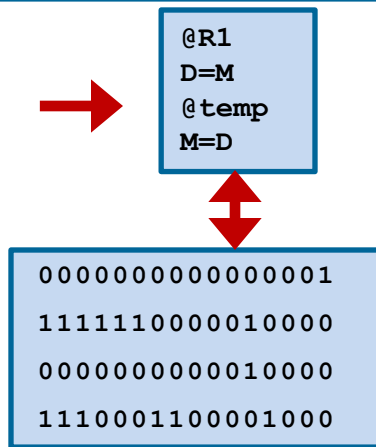
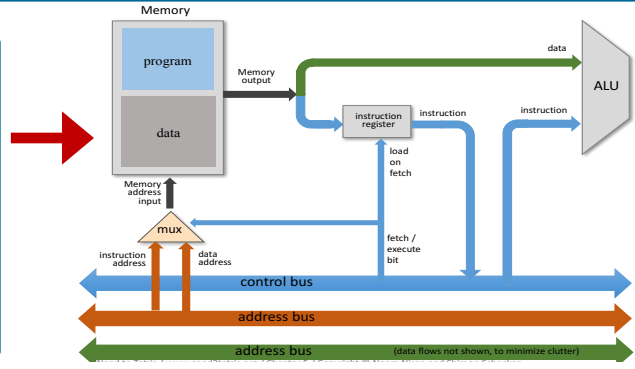




Digital Logic Design



```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```



Lecture # 23-24

Hack Assembly Programming

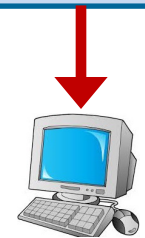
```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
msg: db "Learning is fun with Arif", 0Ah, 0h
len_msg: equ $ - msg
SECTION .text
main:
  mov rax,1
  mov rdi,1
  mov rsi,msg
  mov rdx,len_msg
  syscall
  mov rax,60
  mov rdi,0
  syscall
```

```
0: b8 01 00 00 00
5: bf 01 00 00 00
a: 48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

Slides of first half of the course are adapted from:
<https://www.nand2tetris.org>
Download s/w tools required for first half of the course from the following link:
<https://drive.google.com/file/d/0B9c0BdDz6XpZUh3X2dPR1o0MUE/view>

Instructor: Muhammad Arif Butt, Ph.D.





Today's Agenda

- Hack Assembly Programs
- A Hello World
- CPU Emulator
- Demo
- Program Termination
- Symbols in Hack Assembly Language
 - Built-in Symbols
 - Label Symbols
 - Variable Symbols
- Branching
- Iteration
- Pointers and Arrays

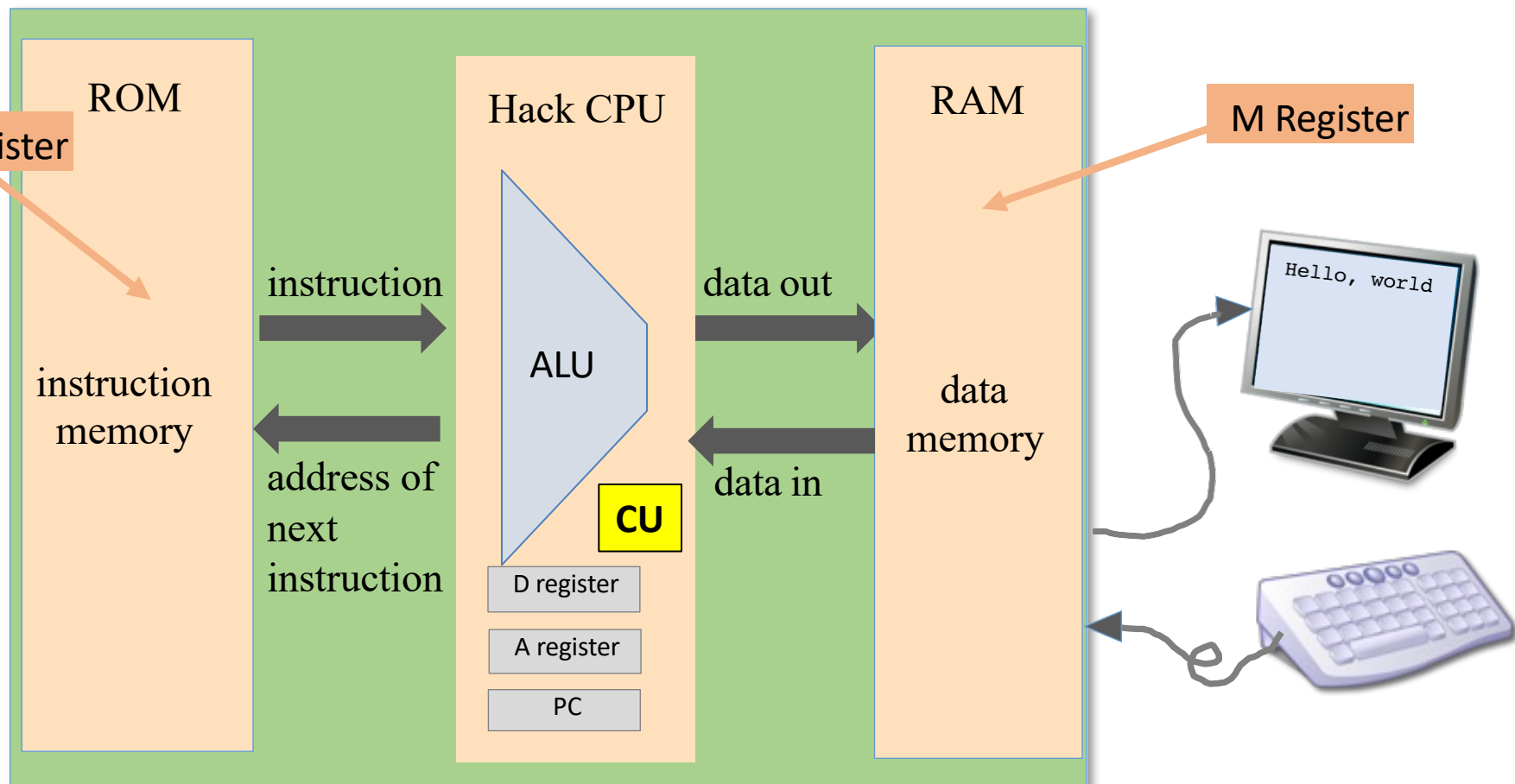




Review of Hack Computer Assembly Instructions



Hack Computer Architecture



Programmer Visible Registers:

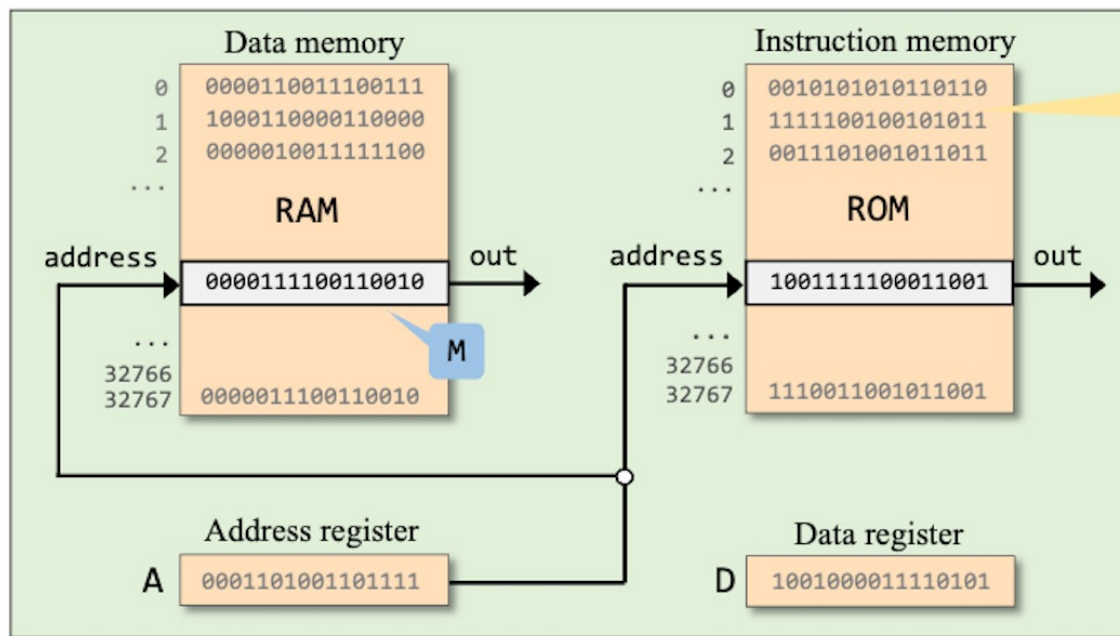
- D: Used to hold data value
- A: Used to hold data value / address of the memory
- M: Represents the currently selected memory register, i.e., $M=RAM[A]$

Data memory (RAM) & Instruction memory (ROM):

- Both are a sequence of 16-bit registers having 15 bit address, i.e., 32K 16 bit words



Recap: Registers and Memory



Loaded with a sequence of 16-bit Hack instructions

(Conceptual, partial view of the Hack computer architecture)

RAM

- Read-write data memory
- Addressed by the A register
- The selected register, RAM[A], is represented by the symbol M

ROM

- Read-only instruction memory
- Addressed by the (same) A register
- The selected register, ROM[A], contains the “current instruction”

- Should we focus on RAM[A], or on ROM[A]?
- Depends on the current instruction (later)



Recap: The Hack Assembly Instructions

The A-instruction:

Syntax: `@value`

Note: Hack Assembly is case sensitive

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

// D++
D=D+1

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

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

// RAM[23]=65
@65
D=A
@23
M=D
```

The C-instruction:

Syntax: `dest= comp ; jump`

`dest= comp`

`comp ; jump`

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

dest: null, M, D, A, MD, AM, AD, AMD

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

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

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

```
// goto instr at addr 72
@72
0; JMP

// if D+1 <= 0 then jump
to instr at addr 1024
@1024
D+1 ;JLE
```



Hack Assembly Programs



Example: addv0.asm

Hack assembly code

```

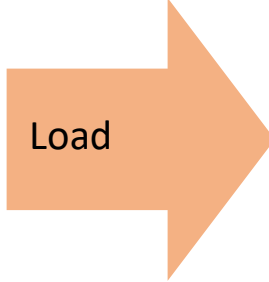
// Program: addv0.asm
// Computes: RAM[0] = 27 + 13

0  @27    // A = 27
1  D=A    // D = 27

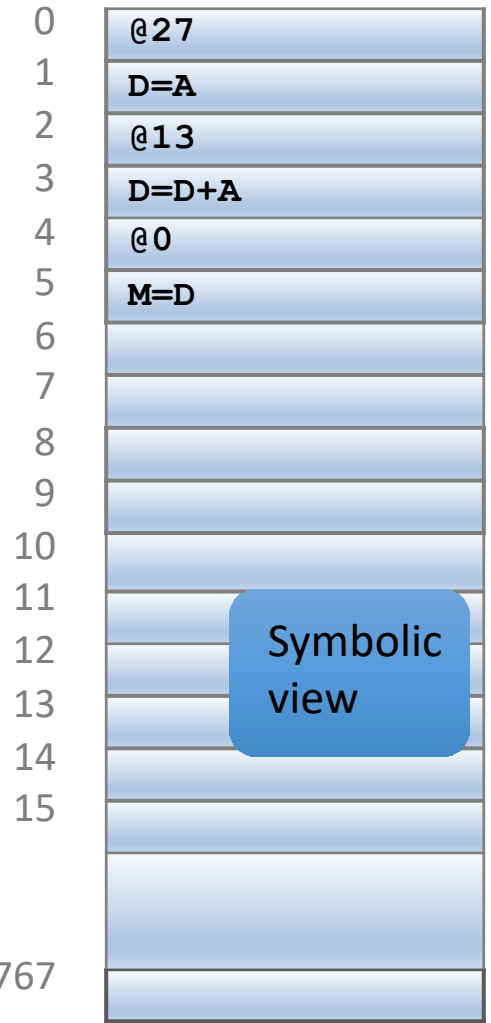
2  @13    // A = 13
3  D=D+A  // D = 27 + 13

4  @0.    // A = 0
5  M=D    // RAM[0] = D

```



Memory (ROM)



Symbolic view



Example: addv0.asm (cont...)

Hack assembly code

```

// Program: addv0.asm
// Computes: RAM[0] = 27 + 13

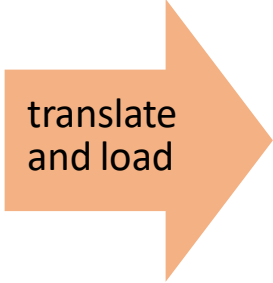
0  @27    // A = 27
1  D=A    // D = 27

2  @13    // A = 13

3  D=D+A  // D = 27 + 13

4  @0.    // A = 0
5  M=D    // RAM[0] = D

```



Memory (ROM)

0	0000000000011011
1	1110110000010000
2	0000000000001101
3	1110000010010000
4	0000000000000000
5	1110001100001000
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
32767	

Machine Code



Example: addv1.asm

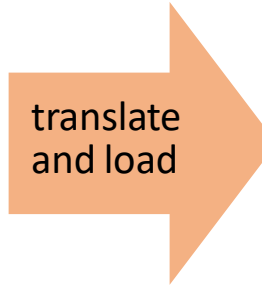
Hack assembly code

```
// Program: addv1.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
```



Memory (ROM)

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

32767

Binary view



Executing a Hack Assembly Program

```
// Program: adv1.asm
// Computes: RAM[2] = RAM[0] + RAM[1]
// Usage: put values in RAM[0], RAM[1]

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

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

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

**Hack
Assembler**

Translate

```
00000000000000000000
11111100000010000
00000000000000000001
1111000010010000
00000000000000000010
1110001100001000
```

**Load in
Hack Computer**

Execute

- We will develop Hack Assembler later in the course
- Now, we can use the CPU emulator for the purpose



Emulator

- In contrast to a simulator, an **emulator** attempt to mimic the hardware features of a production environment, as well as software features
- Emulation is the process of artificially executing code intended for a “foreign” architecture by converting it to the assembly/machine language of that CPU
- The **CPU Emulator** that we will be using is designed and developed by students of Interdisciplinary Center Herzliya Efi Arazi School of Computer Science, headed by Yaron Ukrainitz
- **HACK CPU Emulator** is a software tool build in Java. We can load Hack assembly program into CPU emulator’s instruction memory, the CPU emulator translate it into machine language and execute it
- Convenient for debugging and executing symbolic Hack programs in simulation



How to Download the CPU Emulator?

- Type the following URL in your browser:
<https://bitbucket.org/arifpucit/>
- In the public repositories pane, click the **coal-repo** repository, containing all the source codes as well as the software tools used in this course
- In the left pane, click **Downloads** to download the entire repository on your system. Now on your system just check the contents of **tools** directory that you have just downloaded

```
Arif-MacBook:arifpucit-coal-repo/tools$ ls
HardwareSimulator.sh      HardwareSimulator.bat
CPUEmulator.sh           CPUEmulator.bat
Assembler.sh             Assembler.bat
VMEulator.sh            VMEulator.bat
JackCompiler.sh         JackCompiler.bat
TextComparer.sh        TextComparer.bat
builtInChips      builtInVMCode  bin      OS
```



Starting the CPU Emulator

- Follow the following steps to start the CPU emulator on UNIX/Mac OS:
 - Open the terminal
 - Go to tools directory
 - Set execute permissions of the file `CPUEmulator.sh`
 - Execute it

```
tools — -bash — 80x24
(base) Arifs-MacBook-Pro:tools arif$ ls
Assembler.bat          JackCompiler.bat      VMEulator.sh
Assembler.sh           JackCompiler.sh       bin
CPUEmulator.bat       OS                     builtInChips
CPUEmulator.sh        TextComparer.bat      builtInVMCode
HardwareSimulator.bat TextComparer.sh
HardwareSimulator.sh  VMEulator.bat
(base) Arifs-MacBook-Pro:tools arif$ chmod +x CPUEmulator.sh
(base) Arifs-MacBook-Pro:tools arif$ ./CPUEmulator.sh
```



Loading an Assembly Program in CPU Emulator

Hack assembly code

```
// Program: adv1.asm
// Computes: RAM[2] = RAM[0] + RAM[1]
// Usage: put values in RAM[0], RAM[1]
```

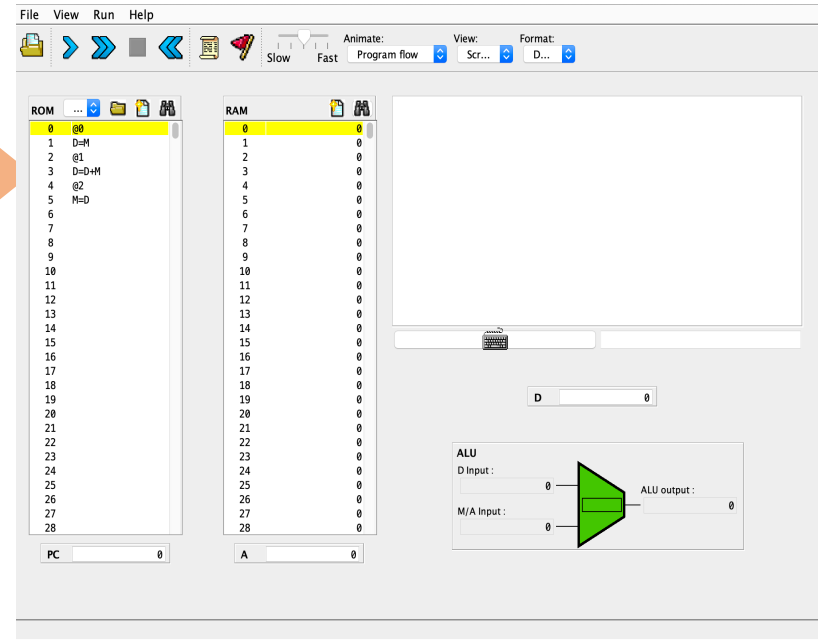
```
0 @0
1 D=M // D = RAM[0]
```

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

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

Load

(the simulator software translates from symbolic to binary as it loads)



CPU Emulator

- A software tool build in Java
- We can load Hack assembly program into CPU emulator's instruction memory, the CPU emulator translate it into machine language and execute it
- Convenient for debugging and executing symbolic Hack programs in simulation



GUI of Hack CPU Emulator

The screenshot shows the Hack CPU Emulator interface with the following components and callouts:

- ROM Asm:** A table of instructions. Row 15 is highlighted in yellow.
- RAM:** A table of memory addresses and values. Address 17 is highlighted in yellow.
- Registers:** A callout box labeled "registers" with lines pointing to the PC (Program Counter) and A (Accumulator) registers.
- Instruction Memory:** A callout box labeled "instruction memory" pointing to the ROM Asm table.
- Data Memory:** A callout box labeled "data memory" pointing to the RAM table.
- Screen:** A callout box labeled "screen" pointing to the large empty area on the right.
- Keyboard Enabler:** A callout box labeled "keyboard enabler" pointing to the keyboard icon at the bottom of the screen area.
- ALU:** A callout box labeled "ALU" pointing to the ALU component at the bottom right.

The ALU component shows the following values:

- D Input: 25
- M/A Input: 17184
- ALU output: 17184

The D register shows the value 17184.



Hack CPU Emulator: Instruction Memory

The screenshot shows the CPU Emulator interface with the following components and annotations:

- ROM Window:** Contains assembly code. Row 15 is highlighted in yellow. An orange callout points to it: "Next instruction is highlighted".
- RAM Window:** Shows memory addresses and values. Row 17 is highlighted in yellow. An orange callout points to it: "The loaded code can be viewed either in binary, or in symbolic notation (present view)".
- Program Counter (PC):** Located at the bottom left, showing the value 15. A yellow callout points to it: "Program counter (PC) (16-bit): Selects the next instruction." An orange arrow points upwards from the PC field.
- ALU:** Located at the bottom right, showing D Input: 25, M/A Input: M, and ALU output: 17184. A yellow callout points to the M/A Input: "Instruction memory (32K): Holds a machine language program".

ROM	Asm	RAM
0		0
1	D=M	1
2	@23	2
3	D, JLE	3
4	@16	4
5	M=D	5
6	@16384	6
7	D=A	7
8	@17	8
9	M=D	9
10	@17	10
11	A=M	11
12	M=-1	12
13	@17	13
14	D=M	14
15	@32	15
16	D=D+A	16
17	@17	17 17184
18	M=D	18
19	@16	
20	MD=M-1	
21	@10	
22	D, JGT	
23	@23	
24	0, JMP	
25		
26		
27		
28		



Hack CPU Emulator: Data Memory

CPU Emulator (1.4b3) - G:\examples\Rect.asm

File View Run Help

Animate: Program flow View: Screen Format: Decimal

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

RAM	
0	50
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	25
17	17184
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

PC 15 A 17

Data memory (32K RAM), used for:

- General-purpose data storage (variables, arrays, objects, etc.)
- Screen memory map
- Keyboard memory map

Address (A) register, used to:

- Select the current RAM location

OR

- Set the Program Counter (PC) for jumps (relevant only if the current instruction includes a jump directive).



Hack CPU Emulator: Registers

Registers (all 16-bit):

- **D:** Data register
- **A:** Address register
- **M:** Stands for the memory register whose address is the current value of the Address register

M (=RAM[A])

D

A

ALU

D Input : 25
M/A Input : 17184
ALU output : 17184

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

RAM	
0	50
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	25
17	17184
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

PC: 15 A: 17



Hack CPU Emulator: ALU

The screenshot shows the CPU Emulator interface with the following components:

- ROM Asm:** A list of instructions. Instruction 14, `D=M`, is highlighted in red and labeled "Current instruction". Instruction 15, `@32`, is highlighted in yellow.
- RAM:** A memory table where address 17 contains the value 17184. This address is highlighted in yellow and labeled "M (=RAM[A])".
- Registers:** The D register contains 17184 and is labeled "D". The A register contains 17 and is labeled "A".
- ALU:** A diagram showing the ALU operation. The D Input is 25, the M/A Input is 17184, and the ALU output is 17184. The operation is labeled "M".

Arithmetic logic unit (ALU)

- The ALU can compute various arithmetic and logical functions (let's call them f) on subsets of the three registers $\{M,A,D\}$
- All ALU instructions are of the form $\{M,A,D\} = f(\{M,A,D\})$ (e.g. $M=M-1$, $MD=D+A$, $A=0$, etc.)
- The ALU operation (LHS destination, function, RHS operands) is specified by the current instruction.



Hack CPU Emulator: Screen and Keyboard

Simulated screen: 256 columns by 512 rows, black & white memory-mapped device. The pixels are continuously refreshed from respective bits in an 8K memory-map, located at RAM[16384] - RAM[24575].

Simulated keyboard: One click on this button causes the CPU emulator to intercept all the keys subsequently pressed on the real computer's keyboard; another click disengages the real keyboard from the emulator.

ROM	Asm
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

RAM	
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

D 0

ALU
D Input : 0
M/A Input : 0
ALU output : 0

Script restarted



Hack CPU Emulator: Loading a Program

ROM Asm

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

RAM

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

PC 0

Load ROM

Look in: 05

- Add.hack
- Max.hack
- Rect.hack

File name: Rect.hack

Files of type: HACK / ASM Files

Load ROM

Cancel

Navigate to a directory and select a .hack OR .asm file.



Hack CPU Emulator: Loading a Program

The screenshot shows the CPU Emulator interface with the following components:

- File Menu:** File, View, Run, Help
- Control Buttons:** Folder icon, play/pause, stop, left arrow, right arrow, document icon, speaker icon, Slow/Fast toggle, Animate: Program flow, View: None, Format: Decimal
- ROM Table:**

Address	Value
0	0000000000000000
1	1111110000010000
2	0000000000101111
3	1110000000001110
4	0000000000100000
5	1110001100000000
6	0100000000000000
7	1110110000010000
8	0000000000010001
9	1110001100001000
10	0000000000010001
11	1111110000100000
12	1110111010001000
13	0000000000010001
14	1111110000010000
15	0000000000100000
16	1110000010010000
17	0000000000010001
18	1110001100001000
19	0000000000010000
20	1111110010011000
21	000000000001010
22	1110001100000001
23	0000000000010111
24	1110101010000111
25	
26	
27	
28	
- RAM Table:**

Address	Value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0
- PC:** 0
- A:** 0
- D:** 0
- ALU:** D Input: 0, M/A Input: 0, ALU output: 0

A yellow callout bubble points to the ROM table with the text: "Can switch from binary to symbolic representation".



Hack CPU Emulator: Running a Program

The screenshot shows the CPU Emulator interface with the following components and callouts:

- Callout 1:** "1. Enter a number, say 50." points to the RAM input field.
- Callout 2:** "2. Click the 'run' button." points to the right-pointing arrow button in the toolbar.
- Callout 3:** "3. To speed up execution, use the speed control slider" points to the slider between 'Slow' and 'Fast'.
- Callout 4:** "4. Watch here" points to the top-left corner of the main display area.

The ROM and RAM tables are as follows:

ROM	Asm
0	@0
1	D=
2	G
3	T
4	
10	@17
11	A=M
12	M=-1
13	@17
14	D=M
15	@32
16	D=D+A
17	@17
18	M=D
19	@16
20	MD=M-1
21	@10
22	D, JGT
23	@23
24	0, JMP
25	
26	
27	
28	

RAM	
0	50
1	0
2	0
3	0
11	
12	
13	
14	0
15	0
16	14
17	17536
18	0
19	
20	0
21	0
22	
23	
24	
25	0
26	0
27	0
28	0

At the bottom, the PC register is 12 and the A register is 17536.

Program's description: Draws a rectangle at the top left corner of the screen. The rectangle's width is 16 pixels, and its length is determined by the current contents of RAM[0].

Note: There is no need to understand the program's code in order to understand what's going on.



Running an Assembly Program in CPU Emulator





Program Termination



Terminating a Program

Hack assembly code

```
// Program: advv1.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
```

Memory (ROM)



NOP slide attack

Resulting from some attack on the computer

Malicious code start here

If you load above program inside the CPU emulator and run it using fast forward button. The computer continues to execute the program from instruction at address 0-5 and then continues executing onwards and does not halt



Terminating a Program

Hack assembly code

```
// Program: addv2.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

Best practice:

Remember computers never stand still. They always need to do some thing, i.e., execute some instruction.

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

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	



Running an Assembly Program in CPU Emulator





Symbols in Hack Assembly Language



Symbols in Hack Assembly Language

- Assembly Instructions can refer to memory locations (addresses) using either constants or symbols. Symbols are introduced into Hack assembly programs in the following three ways:
- **Predefined/Built-in Symbols:** These are a special subset of RAM addresses that can be referred to by any assembly program using virtual registers and I/O pointers
- **Label Symbols:** These are user defined symbols, which serve to label destinations of `goto` commands inside ROM (Program memory)
- **Variable Symbols:** These are also user defined symbols which are assigned unique memory addresses starting at RAM addresses 16 onwards



Pre-Defined / Built-in Symbols



Built-in Symbols: Virtual Registers

To simplify assembly programming, the symbols R0 to R15 are predefined to refer to RAM addresses 0 to 15 respectively

<u>symbol</u>	<u>value</u>
R0	0
R1	1
R2	2
...	...
R15	15

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

These symbols can be used to denote “virtual registers”

Example: Suppose a programmer wants to write a constant value 7 at RAM[5]

Implementation:

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

Better Style:

```
// let RAM[5] = 7
@7
D=A
@R5
M=D
```



Built-in Symbols: I/O Pointers

- The following two symbols SCREEN and KBD are predefined to refer to RAM addresses 16384 (0x4000) and 24576 (0x6000) respectively
- These are the base addresses of the screen and keyboard memory maps (discussed in detail in Lecture # 18)
- These symbols will come into play, when we will write assembly programs that deals with the screen and keyboard in the next lecture

<u>symbol</u>	<u>value</u>
SCREEN	16384
KBD	24576



Branching



Branching

- Branching is the fundamental ability to tell the computer to evaluate certain Boolean expression and based on the result, decide whether or not the flow of execution should continue the next instruction in sequence or jump to some other location in the code
- All programming languages support various branching mechanisms like `if...else`, `while...`, `for...`, and so on
- In machine language we have only one branching mechanism called `goto`



Unconditional Branching

Unconditional branching example (pseudocode)

```

0  instruction
1  instruction
2  instruction
3  instruction
4  goto 7
5  instruction
6  instruction
7  instruction
8  instruction
9  goto 2
10 instruction
11 ...

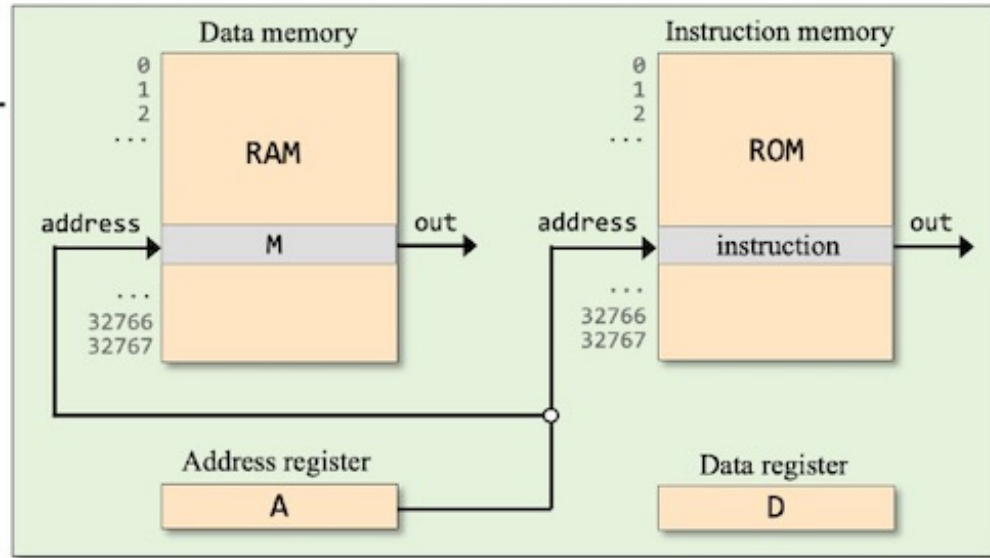
```

Flow of control:

```

0,1,2,3,4,
7,8,9,
2,3,4,
7,8,9,
2,3,4,
...

```

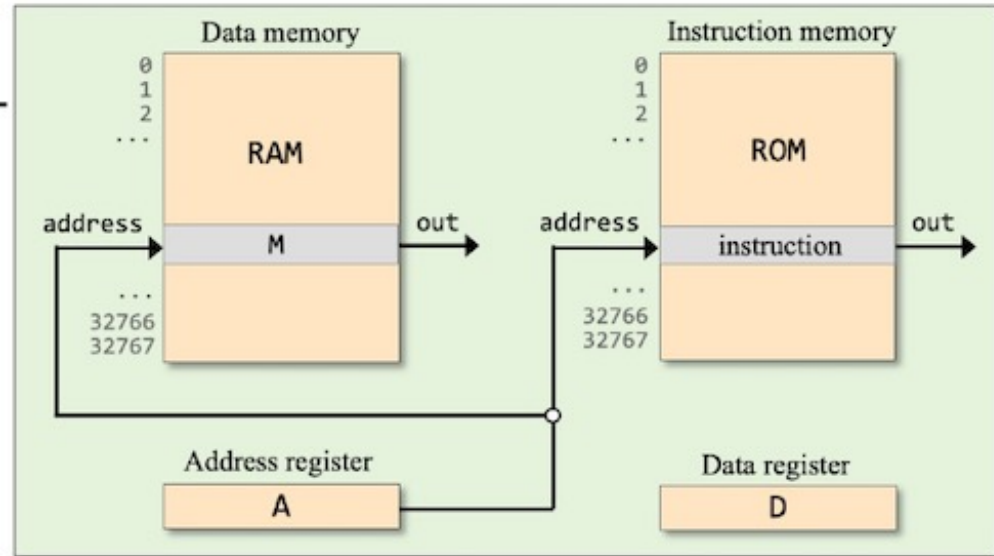




Conditional Branching

Conditional branching example (pseudocode)

```
0 instruction
1 instruction
2 instruction
3 instruction
4 if (condition) goto 7
5 instruction
6 instruction
7 instruction
8 instruction
9 instruction
... ..
```



Flow of control:

```
0,1,2,3,4,
if condition is true
    7,8,9,...
else
    5,6,7,8,9,...
```



Branching Example

```
// Program: ifelsev1.asm
// Computes: if R0 > 0
    R1 = 1
    else
    R1 = 0
// Usage: put a value in RAM[0], run and inspect RAM[1]
0 @R0 //Use of Built-in symbols
1 D=M //D = RAM[0]

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

4 @R1 //Use of Built-in symbols
5 M=0
6 @10
7 0;JMP

8 @R1 //Use of Built-in symbols
9 M=1

10 @10
11 0;JMP
```



Branching Example (cont...)

```
@R0
D=M
@8
D ; JGT
@R1
M=0
@10
0 ; JMP
@R1
M=1
@10
0 ; JMP
```

cryptic code

- If we remove all the comments as well as the line numbers, the code become quite unreadable or cryptic
- It is of course really difficult to understand what this code actually do
- Yet the code will work perfectly fine as expected by the programmer



Branching Example (cont...)

```
@R0  
D=M  
@8  
D ; JGT  
@R1  
M=0  
@10  
0 ; JMP  
@R1  
M=1  
@10  
0 ; JMP
```

cryptic code

“Instead of imagining that our main task as programmers is to instruct a computer what to do, let us concentrate rather on explaining to human beings (fellow programmers) what we intend a computer to do.”

– Donald Knuth



- The Art of Computer Programming - Volume 1 (Fundamental Algorithms)
- The Art of Computer Programming - Volume 2 (Semi-numerical Algorithms)
- The Art of Computer Programming - Volume 3 (Sorting and Searching)
- The Art of Computer Programming - Volume 4 (Combinatorial Algorithms)

Important

If our programs are not self documented, we will not be able to fix and extend them



Use of Predefined Symbols

Use of Pre-defined symbols:

(R0, R1)

```
0 @0
1 D=M

2 @8
3 D;JGT

4 @1
5 M=0
6 @10
7 0;JMP

8 @1
9 M=1

10 @10
11 0;JMP
```

`ifelsev0.asm`



```
0 @R0
1 D=M

2 @8
3 D;JGT

4 @R1
5 M=0
6 @10
7 0;JMP

8 @R1
9 M=1

10 @10
11 0;JMP
```

`ifelsev1.asm`



Running an Assembly Program in CPU Emulator



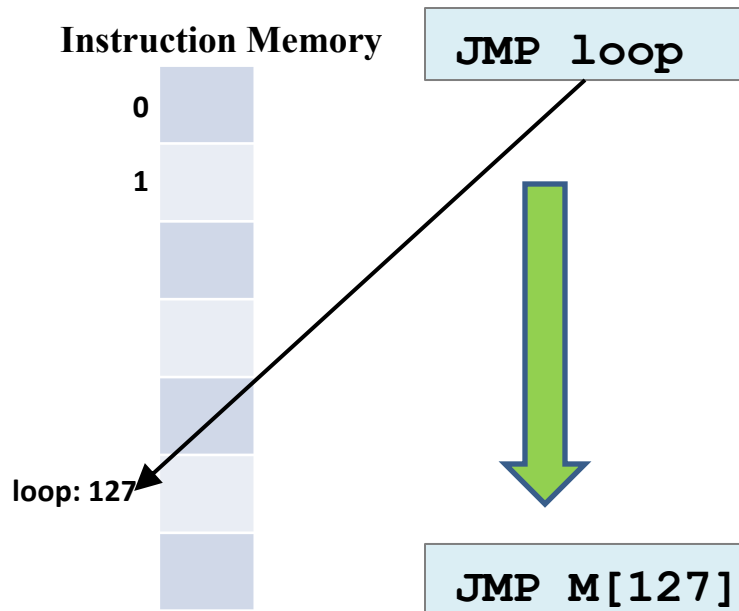


User Defined Symbols in Assembly Language

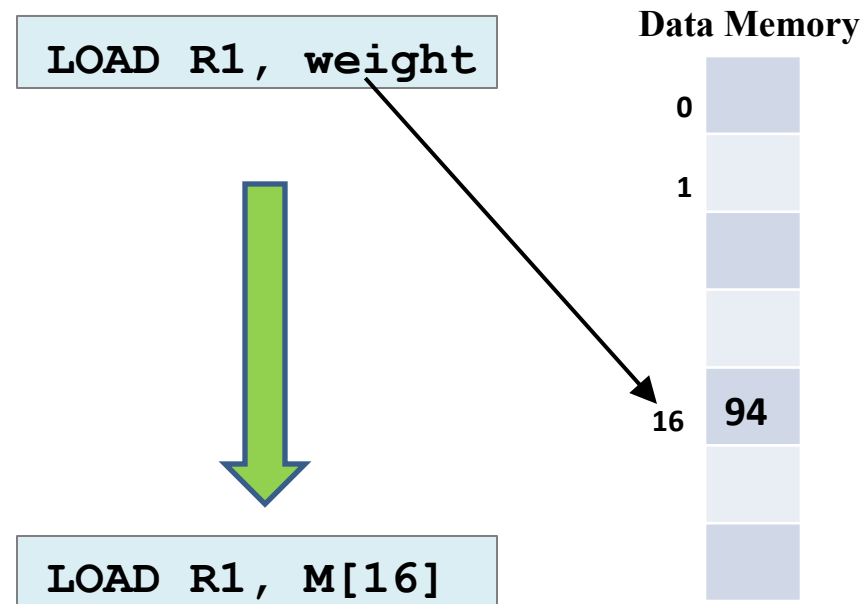
Assembly Instructions can refer to memory locations (addresses) using either constants or symbols. Other than the predefined/build-in symbols, an assembly programmer can use user-defined symbols:

- **Label Symbols** are addresses inside ROM and are used as destinations of *JMP* instructions
- **Variable Symbols** are addresses inside RAM and starts from addresses 16 onwards in Hack

Label Symbols:



Variable Symbols:





Use of Labels



Branching Example: Understanding Labels

```
// Program: ifelsev1.asm
// Computes: if R0 > 0
    R1 = 1
    else
    R1 = 0
// Usage: put a value in RAM[0], run and inspect RAM[1]
0 @R0 //Use of Built-in symbols
1 D=M //D = RAM[0]

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

4 @R1 //Use of Built-in symbols
5 M=0
6 @10
7 0;JMP

8 @R1 //Use of Built-in symbols
9 M=1

10 @10
11 0;JMP
```



Branching Example: Understanding Labels

```
// Program: ifelsev2.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 // @8
3 D;JGT // If R0>0 goto 8
4 @R1
5 M=0
6 @10
7 0;JMP
  (POSITIVE)
8 @R1
9 M=1
10 @10
11 0;JMP
```

Referring
to a label

declaring
a label

- These are user-defined symbols, which serve to label destinations of goto commands
- Declared by (xxx) directive
- So @xxx refer to the instruction number following the declaration
- A label can be declared only once and can be referred to any number of times and any-where in the assembly program, even before the line in which it is declared using it in A-instruction
- The name of a user defined symbol can be any sequence of alphabets, digits, underscore, dot, dollar sign and a colon. However, the name must not begin with a digit
- The naming convention is to use uppercase alphabets for labels and lower case alphabets for variables



Branching Example: Understanding Labels

```
// Program: ifelsev2.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 //@8
3 D;JGT // If R0>0 goto 8

4 @R1
5 M=0
6 @END //@10
7 0;JMP
  (POSITIVE)
8 @R1
9 M=1
  (END)
10 @END //@10
11 0;JMP
```

Referring to a label

declaring a label

Referring to a label

- These are user-defined symbols, which serve to label destinations of goto commands
- Declared by (xxx) directive
- So @xxx refer to the instruction number following the declaration
- A label can be declared only once and can be referred to any number of times and any-where in the assembly program, even before the line in which it is declared
- The name of a user defined symbol can be any sequence of alphabets, digits, underscore, dot, dollar sign and a colon. However, the name must not begin with a digit
- The naming convention is to use uppercase alphabets for labels and lower case alphabets for variables



Branching Example : Resolving Labels

```

// Program: ifelsev2.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 //@8
3 D;JGT // If R0>0 goto 8

4 @R1
5 M=0
6 @END //@10
7 0;JMP
  (POSITIVE)
8 @R1
9 M=1
  (END)
10 @END //@10
11 0;JMP

```



Label resolution rules:

- Label declarations are not translated, are ignored, so generate no code and are called pseudo-commands
- Each reference to a label is translated, i.e., replaced with a reference to the instruction number following that label's declaration

ROM	
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	



Use of Predefined Symbols and Labels

Use of Pre-defined symbols:

(R0, R1)

```
0 @0
1 D=M

2 @8
3 D;JGT

4 @1
5 M=0
6 @10
7 0;JMP

8 @1
9 M=1

10 @10
11 0;JMP
```

ifelsev0.asm



```
0 @R0
1 D=M

2 @8
3 D;JGT

4 @R1
5 M=0
6 @10
7 0;JMP

8 @R1
9 M=1

10 @10
11 0;JMP
```

ifelsev1.asm



Use of Label symbols:

(POSITIVE, END)

```
0 @R0
1 D=M

2 @POSITIVE
3 D;JGT

4 @R1
5 M=0
6 @END
7 0;JMP
(POSITIVE)
8 @R1
9 M=1
(END)
10 @END
11 0;JMP
```

ifelsev2.asm



Running an Assembly Program in CPU Emulator





Use of Variables



Variables

- Variable is an abstraction of a container, that has a name, a value and an associated address inside RAM
- You can say that it is a named memory location
- In high level languages we also have a type associated with a variable, but in Hack machine/assembly language, we have only 16 bit values of a variable
- So in Hack assembly language, a variable is user-defined symbol **xxx** appearing in the program that is not predefined and is not defined elsewhere using the **(xxx)** directive.
- **All variables are assigned unique memory addresses by the Hack Assembler, starting at RAM address 16 (0x0010)**



Example: Using Variables

```
//Program: swap.asm
//flips the values of RAM[0] and RAM[1]
//temp = R1
// R1 = R0
//R0 = temp
// temp = R1
@R1
D=M
@temp
M=D
// R1 = R0
@R0
D=M
@R1
M=D
// R0 = temp
@temp
D=M
@R0
M=D
(END)
@END
0 ; JMP
```

symbol used for
the first time
(variable created)

symbol used again

@temp:

- Any symbol **xxx** appearing in the Hack assembly program that is not predefined and is not defined elsewhere using the **(xxx)** directive is treated as a variable. Each variable is assigned a unique memory address starting at RAM address 16 (0x0010)
- Since **@temp** is the first occurrence of the symbol **temp**, not declared as a label elsewhere using **(temp)**, so this qualifies it to be a variable
- So each occurrence of this variable **temp** in the program inside an A-instruction will be translated into **@16**
- So you first declare/creates a variable using an A-instruction **@temp** and then assign it a value using C-instruction **M=D**



Example: Resolving Variables

```

//Program: swap.asm
//flips the values of RAM[0] and RAM[1]
//temp = R1
// R1 = R0
//R0 = temp
@R1
D=M
@temp
M=D // temp = R1

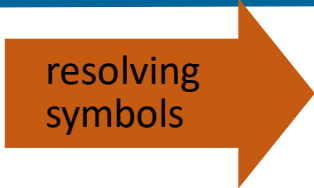
@R0
D=M
@R1
M=D // R1 = R0

@temp
D=M
@R0
M=D // R0 = temp
(END)
@END
0 ; JMP

```

Symbolic variable

Symbolic label



Symbol resolution rules:

- A reference to a symbol that has no corresponding label declaration is treated as a reference to a variable
- Variables are allocated to the RAM from address 16 onward (say n), and the generated code is @n
- Here we have only one variable, so that is allocated RAM address 16. If there are more they will be allocated address 17, 18, and so on

In other words: variables are allocated to RAM[16] onward.

ROM

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	



Implications of Using Symbols

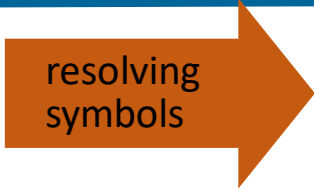
```

//Program: swap.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

```

Symbolic variable

Symbolic label



Implications:

Symbolic code is easy to read and debug

ROM

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

The program has become Relocatable Code:

- You can take this program and load it into memory, not necessarily to address zero, as long as you remember the base address of memory where this program is loaded
- This is very important when several such programs are loaded and running inside the memory



Running an Assembly Program in CPU Emulator





Example : maxv1.asm

```
0  @0
1  D=M //D = First no
2  @1
3  D=D-M //D = First no - Second no
4  @10
5  D;JGT // if D>0 (first is greater) goto address 10
6  @1
7  D=M // D = second number (which is max)
8  @12
9  0;JMP // if D<0 (second is greater) goto address 12
10 @0
11 D=M // D = first number (which is max)
12 @2
13 M=D // M[2] = D (max number)
14 @14
15 0;JMP
```

0	5
1	3
2	5



Running an Assembly Program in CPU Emulator





Iteration



Iteration Example

Pseudo
Code:

```

// Computes RAM[1] = 1 + 2 + 3 ... + n
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:

```

// Computes RAM[1] = 1 + 2 + 3 ... + n
// Usage: put a number (n) in RAM[0]
@R0
D=M
@n      // First variable at RAM addr 16
M=D     // n = R0
@i      // second variable at RAM addr 17
M=1    // i = 1
@sum    // third variable at RAM addr 18
M=0    // sum = 0
. . . .

```

ROM

0	@0
1	D=M
2	@16 // @n
3	M=D
4	@17 // @i
5	M=1
6	@18 // @sum
7	M=0
8	...
9	
10	
11	
12	
13	
14	
15	
32767	



Variables are allocated to consecutive RAM locations from address 16 onwards



Iteration Example

```
// Computes RAM[1] = 1 + 2 + 3 ... + n
n = R0
i = 1
sum = 0
LOOP:
  if i > n goto STOP
  sum = sum + i
  i = i + 1
  goto LOOP
STOP:
  R1 = sum
```

Pre-defined symbols:

(R0, R1)

Label symbols:

(LOOP, STOP, END)

Variable symbols:

(n, i, sum)

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



Iteration





Pointers and Arrays



Pointers and Arrays: Example

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

Observations:

- Variables that store memory addresses like **arr** in this example are called pointers
- Abstraction of arrays exist only in high level languages. In machine language there is no abstraction of arrays. Rather array is a segment of memory of which we know the base address of this segment and the length of the array that programmer has declared
- Arrays are implemented as a block of memory registers and in order to access these memory registers one after the other, we need a variable that holds the current address
- There is nothing special about pointer variables, except that their values are interpreted as addresses



Pointers and Arrays: Example

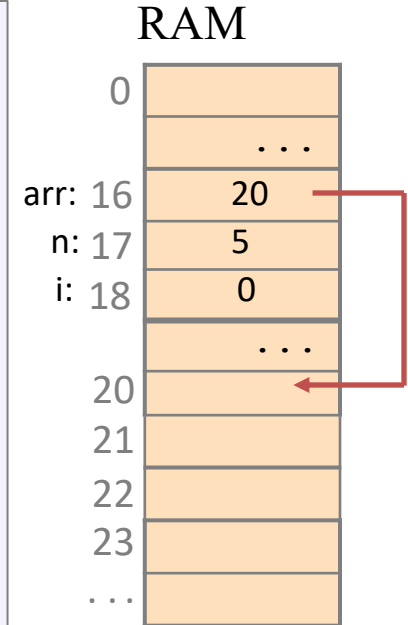
```
// for (i=0; i<n; i++)
//     arr[i] = -1;
// Let us initialize arr=20, n=5, i=0

// arr = 20  A pointer var pointing to RAM[20]
@20
D=A
@arr
M=D

// n = 5  A data var containing value 5
@5
D=A
@n
M=D

// i = 0 A data var containing value 0
@i
M=0

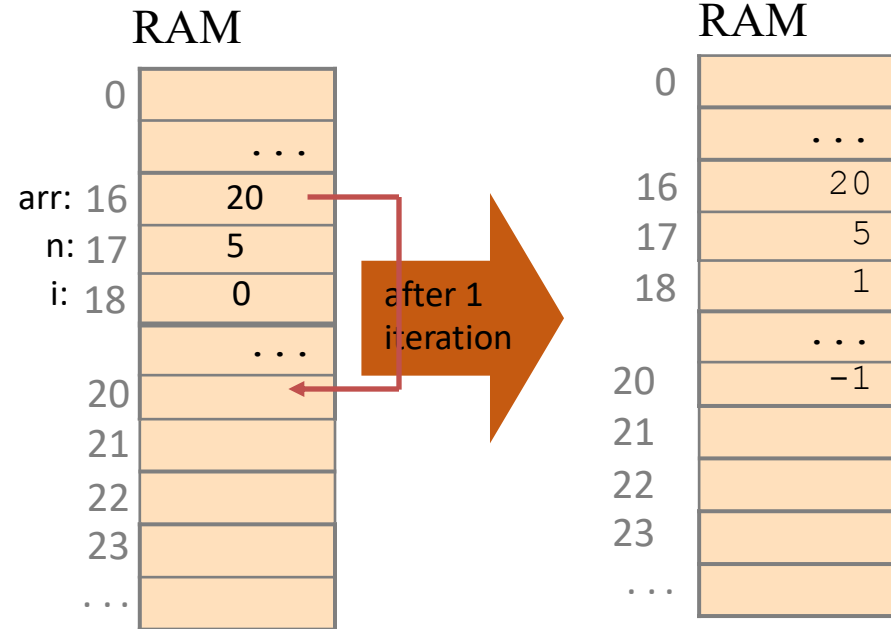
// Loop code continues on next slide...
(LLOOP)
```





Pointers and Arrays: Example

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

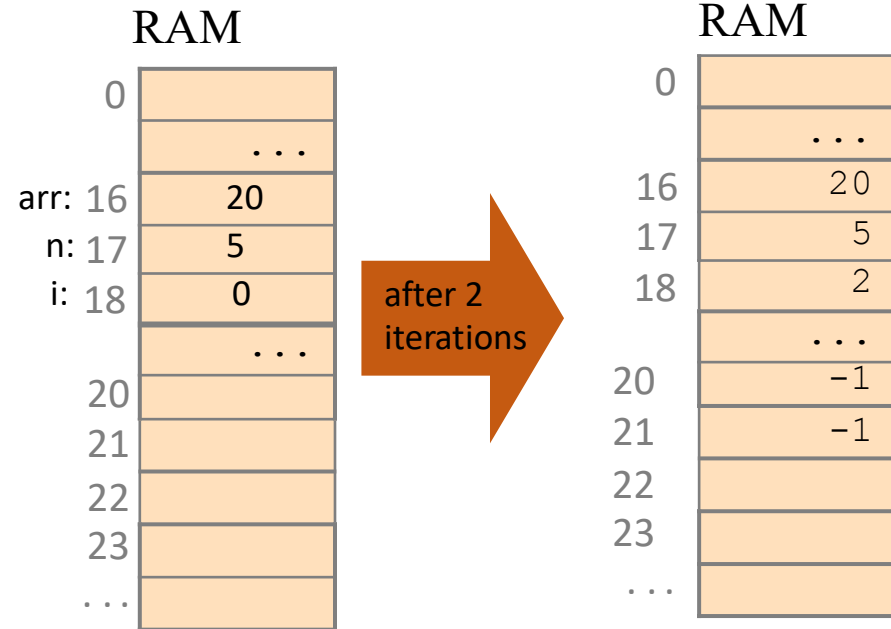


- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like `A=expression`
- Typical Pointer Semantics: Set the address register to the contents of some memory register



Pointers and Arrays: Example

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

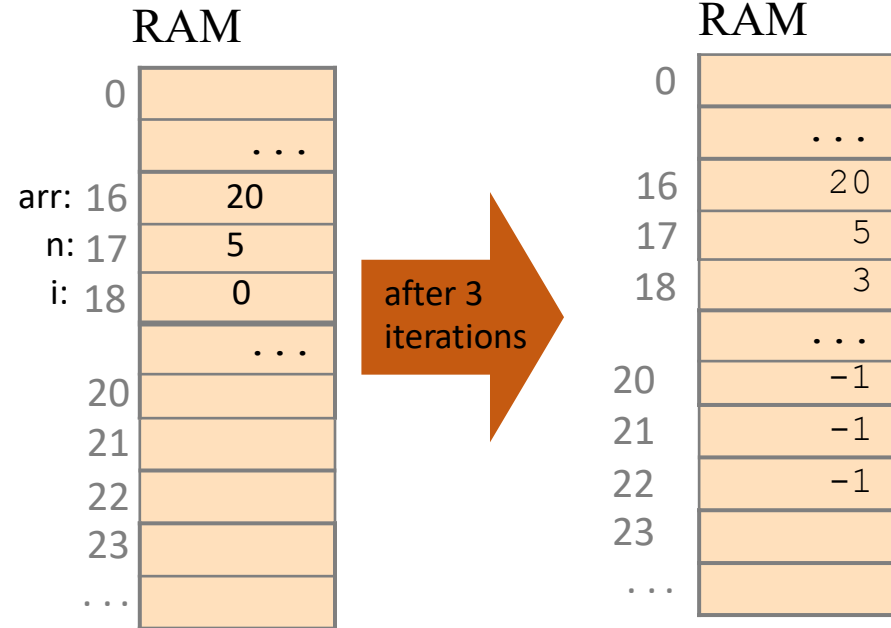


- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like `A=expression`
- Typical Pointer Semantics: Set the address register to the contents of some memory register



Pointers and Arrays: Example

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

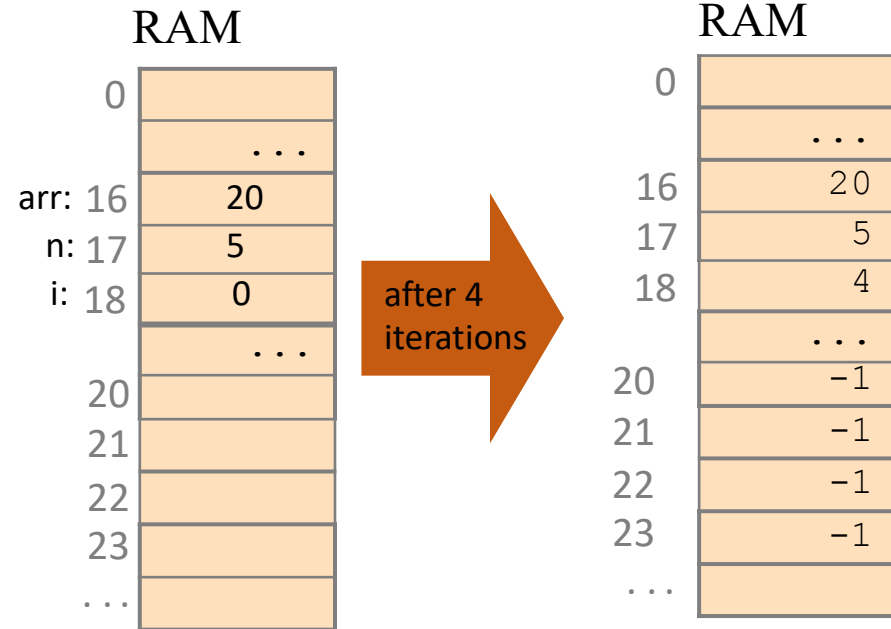


- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like `A=expression`
- Typical Pointer Semantics: Set the address register to the contents of some memory register



Pointers and Arrays: Example

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

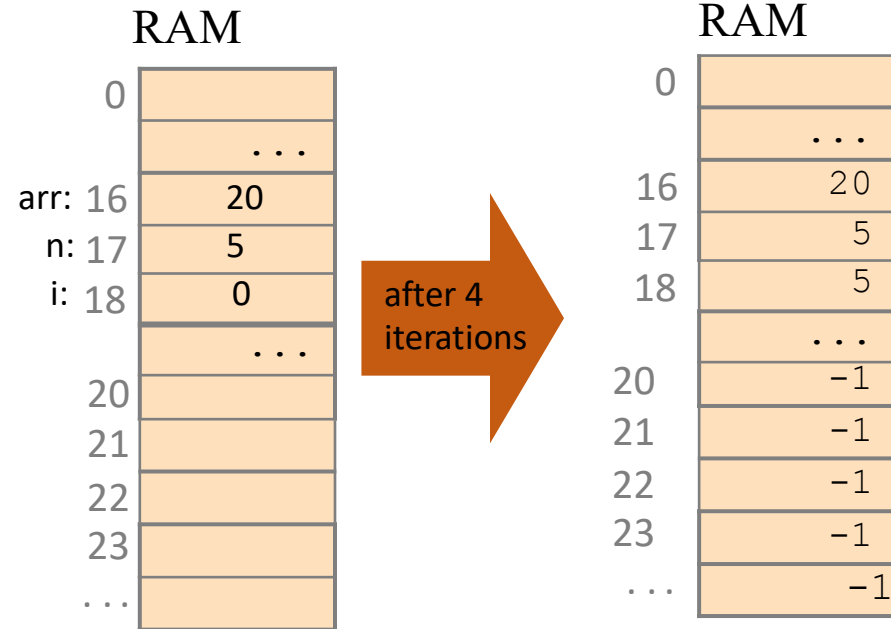


- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like $A = \text{expression}$
- Typical Pointer Semantics: Set the address register to the contents of some memory register



Pointers and Arrays: Example

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



- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like `A=expression`
- Typical Pointer Semantics: Set the address register to the contents of some memory register



Manipulating Arrays using Pointers





Things To Do



Coming to office hours does NOT mean you are academically weak!