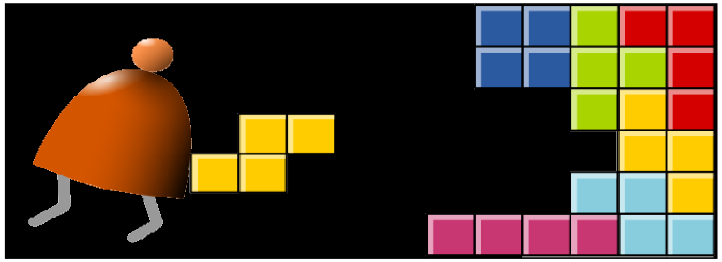
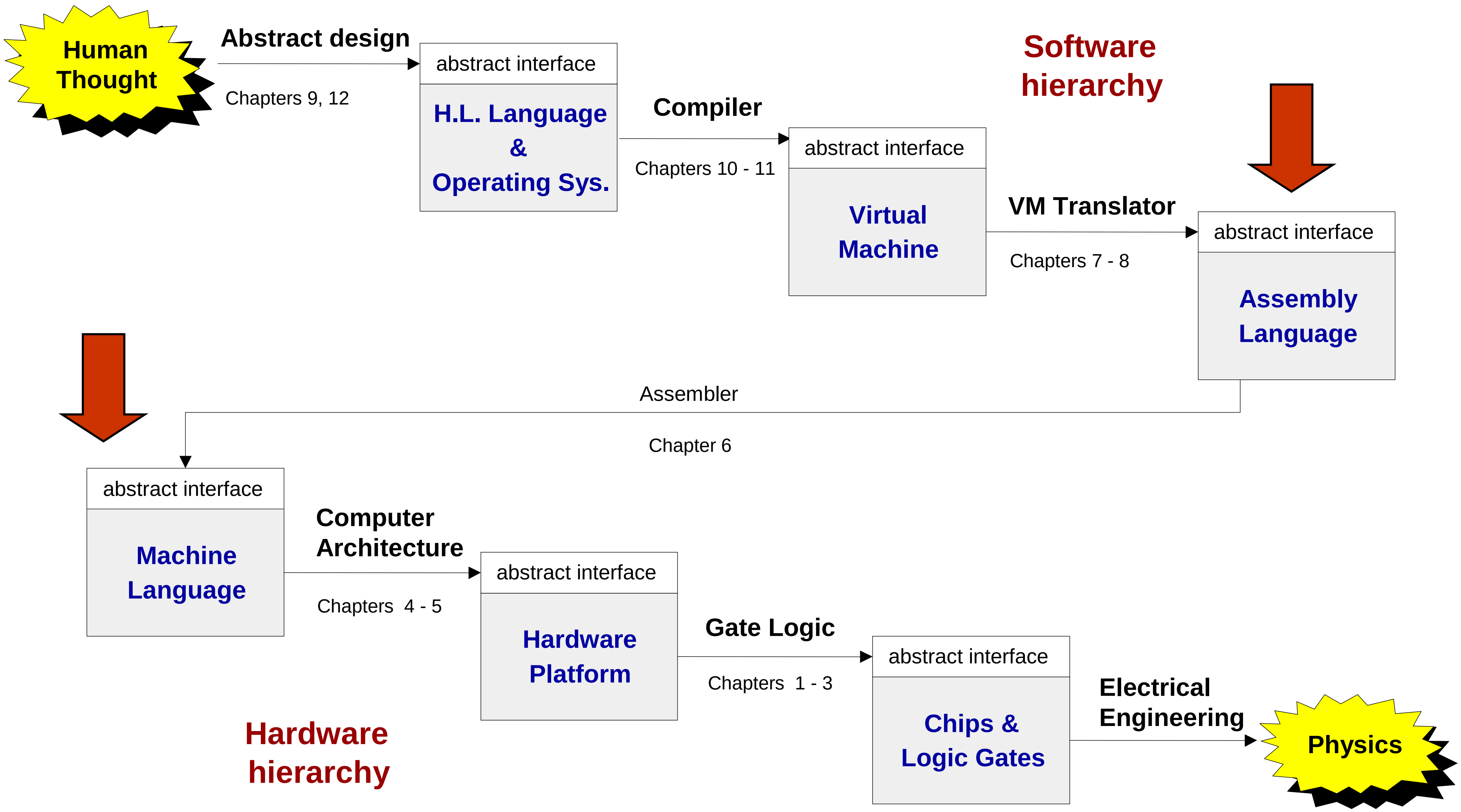
Machine Language



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Machine language

Abstraction – implementation duality:

- Machine language (= instruction set) can be viewed as a programmer-oriented abstraction of the hardware platform
- The hardware platform can be viewed as a physical means for realizing the machine language abstraction

Another duality:

- Binary version
- Symbolic version

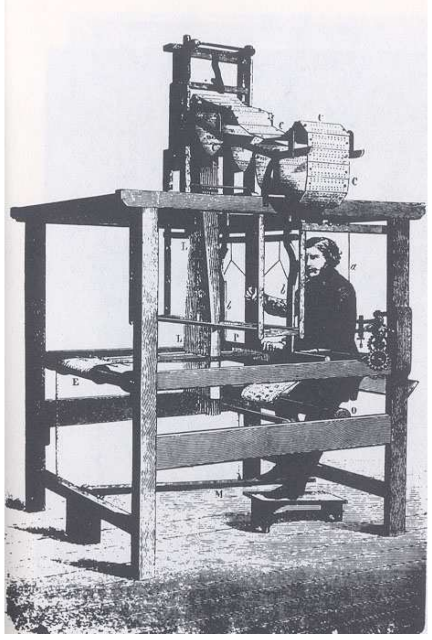
Loose definition:

- Machine language = an agreed-upon formalism for manipulating
  a *memory* using a *processor* and a set of *registers*
- Same spirit but different syntax across different hardware platforms.

Binary and symbolic notation

`1010 0001 0010 1011`

`ADD R1, R2, R3`



Jacquard loom

(1801)

Evolution:

- Physical coding
- Symbolic documentation
- Symbolic coding
- Translation and execution
- Requires a *translator*.

Augusta Ada King, Countess of Lovelace

(1815-1852)

Lecture plan

- Machine languages at a glance
- The Hack machine language:
  - Symbolic version
  - Binary version
- Perspective

(The assembler will be covered in lecture 6).

Typical machine language commands (a small sample)

```
// In what follows R1,R2,R3 are registers, PC is program counter,
// and addr is some value.

ADD R1,R2,R3       // R1 ← R2 + R3

ADDI R1,R2,addr    // R1 ← R2 + addr

AND R1,R1,R2       // R1 ← R1 and R2 (bit-wise)

JMP addr           // PC ← addr

JEQ R1,R1,addr     // IF R1 == R2 THEN PC ← addr ELSE PC++

LOAD R1, addr      // R1 ← RAM[addr]

STORE R1, addr     // RAM[addr] ← R1

NOP                // Do nothing

// Etc. – some 50-300 command variants
```

The Hack computer

A 16-bit machine consisting of the following elements:

Data memory: **RAM** – an addressable sequence of registers

Instruction memory: **ROM** – an addressable sequence of registers

Registers: **D**, **A**, **M**, where **M** stands for **RAM[A]**

Processing: **ALU**, capable of computing various functions

Program counter: **PC**, holding an address

Control: The **ROM** is loaded with a sequence of 16-bit instructions, one per memory location, beginning at address 0. Fetch-execute cycle: later

Instruction set: Two instructions: A-instruction, C-instruction.

The A-instruction

```
@value      // A ← value
```

Where *value* is either a number or a symbol referring to some number.

Used for:

- Entering a constant value
(**A** = **value**)

```
@17      // A = 17
D = A    // D = 17
```

- Selecting a **RAM** location
(**register** = **RAM[A]**)

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

- Selecting a **ROM** location
(**PC** = **A**)

Later

```
@17      // A = 17
JMP      // fetch the instruction
         // stored in ROM[17]
```

The C-instruction (first approximation)

dest = *x* + *y*

dest = *x* - *y*

dest = *x*

dest = 0

dest = 1

dest = -1

x = {A, D, M}

y = {A, D, M , 1}

dest = {A, D, M, MD, A, AM, AD, AMD, null}

Exercise: Implement the following tasks using Hack commands:

- Set **D** to **A-1**
- Set both **A** and **D** to **A + 1**
- Set **D** to **19**
- Set both **A** and **D** to **A + D**
- Set **RAM[5034]** to **D - 1**
- Set **RAM[53]** to **171**
- Add **1** to **RAM[7]**, and store the result in **D**.

The C-instruction (first approximation)

dest = *x* + *y*

dest = *x* - *y*

dest = *x*

dest = 0

dest = 1

dest = -1

x = {A, D, M}

y = {A, D, M , 1}

dest = {A, D, M, MD, A, AM, AD, AMD, null}

Symbol table:

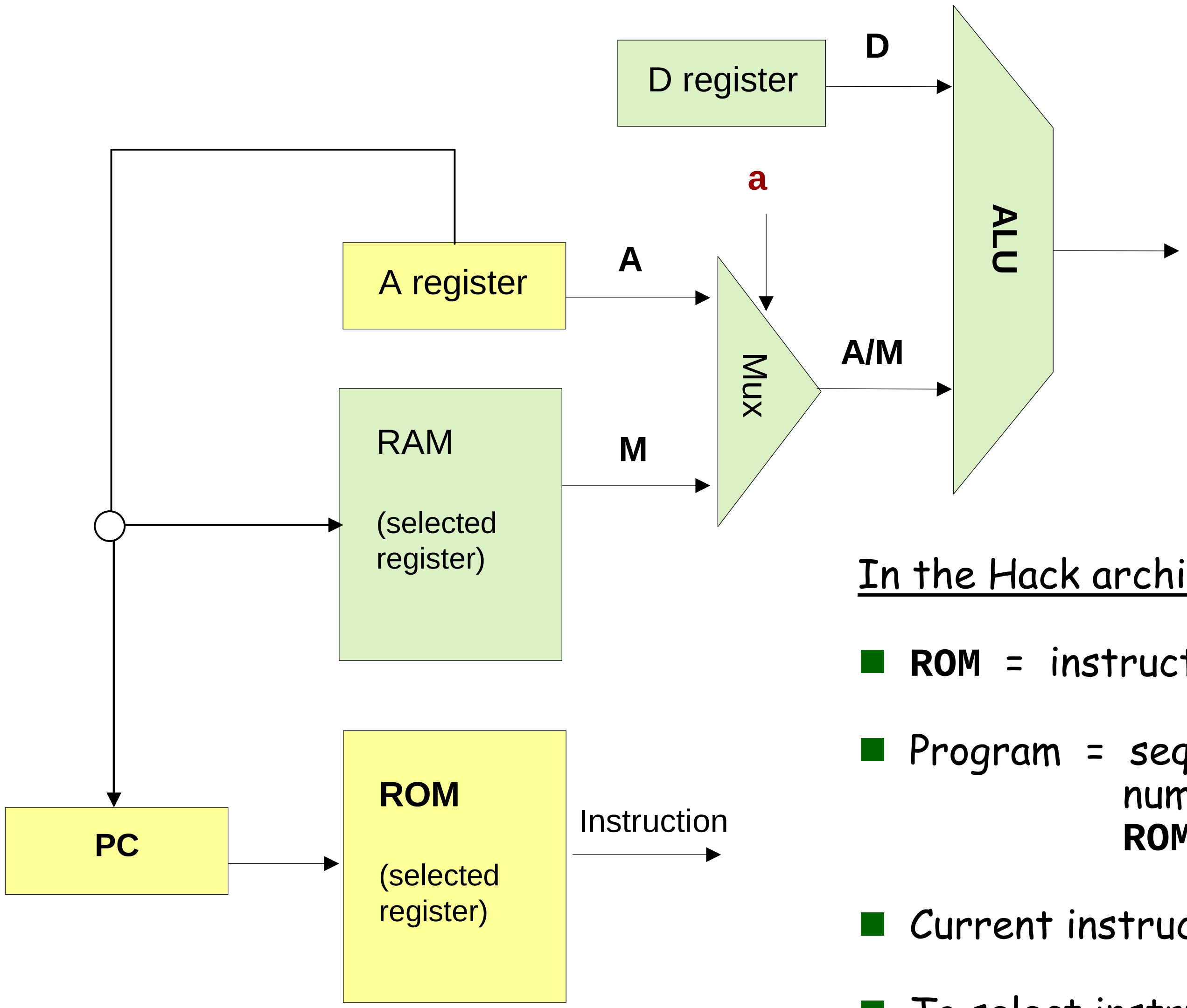
j	17
sum	22
q	21
arr	16

(All symbols and values in are arbitrary examples)

Exercise: Implement the following tasks using Hack commands:

- `sum = 0`
- `j = j + 1`
- `q = sum + 12 – j`
- `arr[7] = 0`
- `Etc.`

Control (focus on the yellow chips only)



In the Hack architecture:

- **ROM** = instruction memory
- Program = sequence of 16-bit numbers, starting at **ROM[0]**
- Current instruction = **ROM[PC]**
- To select instruction *n* from the **ROM**, we set **A** to *n*, using the instruction **@n**

Coding examples (practice)

Exercise: Implement the following tasks using Hack commands:

- `GOTO 50`
- `IF D == 0 GOTO 112`
- `IF D < 9 GOTO 507`
- `IF RAM[12] > 0 GOTO 50`
- `IF sum > 0 GOTO END`
- `IF x[i] <= 0 GOTO NEXT.`

Hack commands:

@value // set A to value

dest = comp ; jump // "dest = " is 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 = **M** , **D** , **MD** , **A** , **AM** , **AD** , **AMD**, or null

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

All conditional jumps refer to the current value of D.

Symbol table:

sum	200
x	4000
i	151
END	50
NEXT	120

(All symbols and values in are arbitrary examples)

C-instruction syntax (final version)

```
dest = comp ; jump          // comp is mandatory
                            // dest and jump are optional
```

Where:

comp is one of:

```
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 is one of:

```
null, M, D, MD, A, AM, AD, AMD
```

jump is one of:

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

IF logic – Hack style

High level:

```
if condition {
   code block 1}
else {
   code block 2}
code block 3
```

Hack:

```
   D ← not condition
   @IF_TRUE
   D;JEQ
   code block 2
   @END
   0;JMP
(IF_TRUE)
   code block 1
(END)
   code block 3
```

Hack convention:

- True is represented by -1
- False is represented by 0

WHILE logic – Hack style

High level:

```
while condition {
    code block 1
}
Code block 2
```

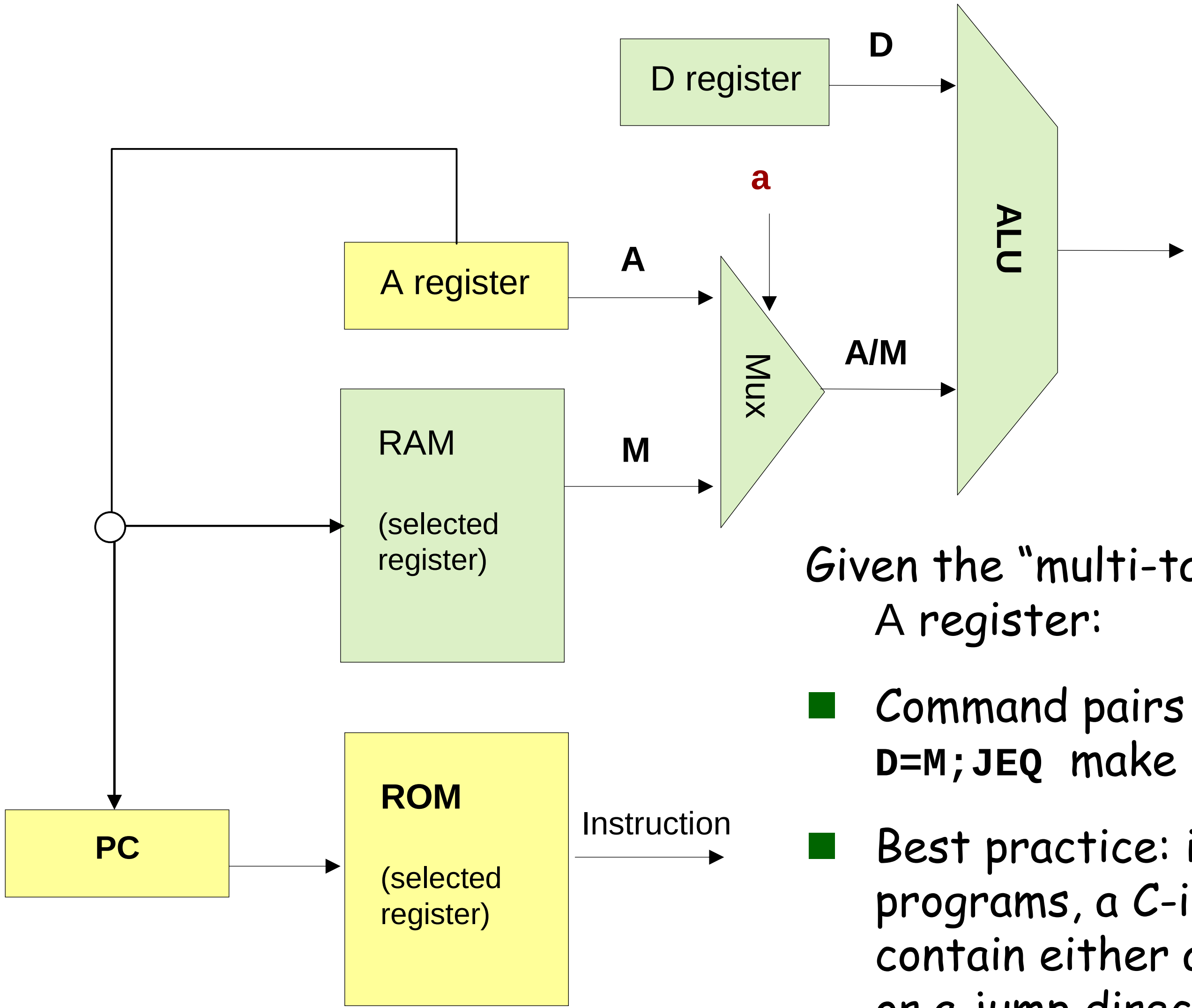
Hack:

```
(LOOP)
     D ← not condition)
     @END
     D;JEQ
     code block 1
     @LOOP
     0;JMP
(END)
     code block 2
```

Hack convention:

- True is represented by -1
- False is represented by 0

Side note



Given the "multi-tasking" nature of the A register:

- Command pairs like `@100` followed by `D=M;JEQ` make no sense
- Best practice: in well-written Hack programs, a C-instruction should contain either a reference to `M`, or a jump directive, but not both.

Complete program example

C:

```
// Adds 1+...+100.
 into i = 1;
 into sum = 0;
 while (i <= 100){
    sum += i;
    i++;
 }
```

Hack:

```
// Adds 1+...+100.
      @i      // i refers to some memo. location
      M=1     // i=1
      @sum    // sum refers to some memo. location
      M=0     // sum=0
(LOOP)
      @i
      D=M      // D = i
      @100
      D=D-A    // D = i - 100
      @END
      D;JGT    // If (i-100) > 0 got END
      @i
      D=M      // D = i
      @sum
      M=D+M    // sum += i
      @i
      M=M+1    // i++
      @LOOP
      0;JMP    // Got LOOP
 (END)
      @END
      0;JMP    // Infinite loop
```

<u>Hack assembly convention:</u>

- Variables: lower-case
- Labels: upper-case
- Commands: upper-case

Demo
CPU emulator

Lecture plan

- Symbolic machine language
- <u>Binary machine language</u>

A-instruction

Symbolic: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.

value (v = 0 or 1)

Binary:

0	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v

C-instruction

Symbolic: *dest*=*comp*;*jump* // Either the *dest* or *jump* fields may be empty.

comp *dest* *jump*

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

(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
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	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

j1 (*out* < 0)	j2 (*out* $= 0$)	j3 (*out* > 0)	**Mnemonic**	**Effect**
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* $= 0$ jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* $\neq 0$ jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

Symbols (user-defined)

- Label symbols: Used to label destinations of goto commands. Declared by the pseudo command **(XXX)**. This directive defines the symbol **XXX** to refer to the instruction memory location holding the next command in the program
- Variable symbols: Any user-defined symbol **xxx** appearing in an assembly program that is not defined elsewhere using the "**(xxx)**" directive is treated as a variable, and is assigned a unique memory address by the assembler, starting at RAM address 16
- By convention, label symbols are upper-case and variable symbols are lower-case.

```
   @R0
   D=M
   @INFINITE_LOOP
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @addr
   M=D
(LOOP)
   @addr
   A=M
   M=-1
   @addr
   D=M
   @32
   D=D+A
   @addr
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP
```

Symbols (pre-defined)

- Virtual registers: **R0**,…, **R15** are predefined to be 0,…,15
- I/O pointers: The symbols **SCREEN** and **KBD** are predefined to be 16384 and 24576, respectively (base addresses of the *screen* and *keyboard* memory maps)
- Predefined pointers: the symbols **SP**, **LCL**, **ARG**, **THIS**, and **THAT** are predefined to be 0 to 4, respectively.

```
   @R0
   D=M
   @INFINITE_LOOP
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @addr
   M=D
(LOOP)
   @addr
   A=M
   M=-1
   @addr
   D=M
   @32
   D=D+A
   @addr
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP
```

Perspective

- Hack is a simple machine language
- User friendly syntax: **D=D+A** instead of **ADD D,D,A**
- Hack is a "½-address machine": it normally takes two commands to get something done: **A**-command to address, **C**-command to process
- A Macro-language can be easily developed
- A Hack assembler is needed and will be discusses and developed later in the course.

End-note: a macro machine language (optional, can be implemented rather easily)

Assignment:

1. **x = constant** (e.g. **x = 17**)
2. **x = y**
3. **x = 0 , x = 1, x = -1**

Arithmetic / logical:

4. **x = y op z**
   where **y**, **z** are variables or constants and
   op is some ALU operation like **+**, **-**, **and**, **or**, etc.

Control:

5. **GOTO s**
6. **IF cond GOTO s**
   where **cond** is an expression **(x op y) {=|<|>|…} {0|1}**
   e.g. **IF x+17 > 0 goto loop**

White space or comments:

7. White space: ignore
8. **//** comment to the end of the line: ignore.