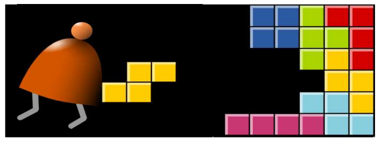
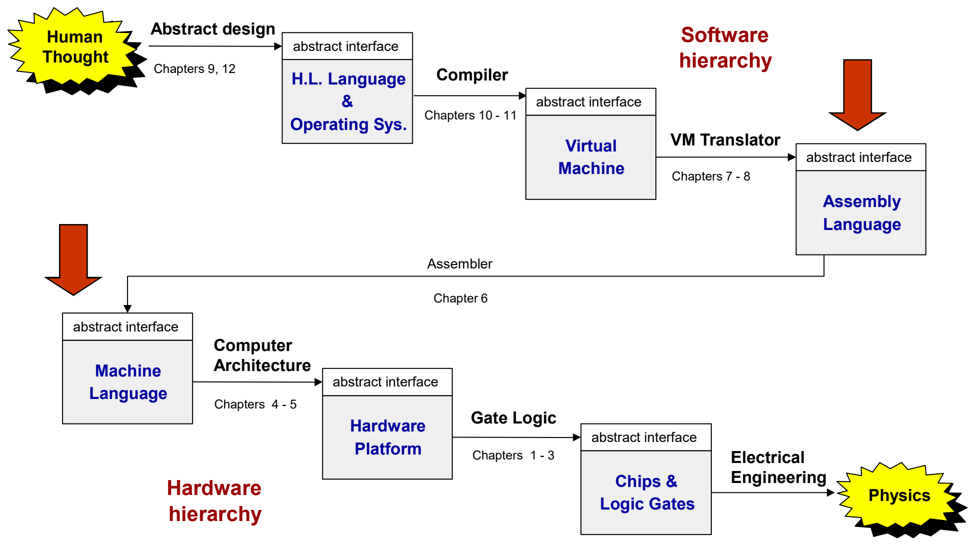
Machine (Assembly) Language



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:

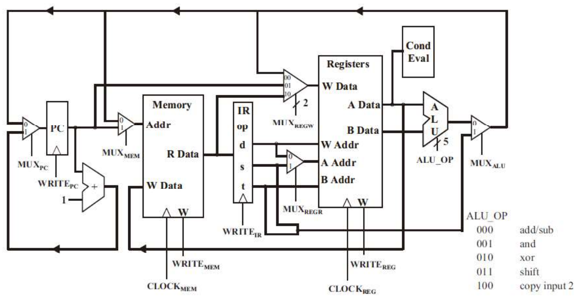


Machine language

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

#	Operation	Fmt	Pseudocode
0:	halt	1	`exit(0)`
1:	add	1	`R[d] ← R[s] + R[t]`
2:	subtract	1	`R[d] ← R[s] - R[t]`
3:	and	1	`R[d] ← R[s] & R[t]`
4:	xor	1	`R[d] ← R[s] ^ R[t]`
5:	shift left	1	`R[d] ← R[s] << R[t]`
6:	shift right	1	`R[d] ← R[s] >> R[t]`
7:	load addr	2	`R[d] ← addr`
8:	load	2	`R[d] ← mem[addr]`
9:	store	2	`mem[addr] ← R[d]`
A:	load indirect	1	`R[d] ← mem[R[t]]`
B:	store indirect	1	`mem[R[t]] ← R[d]`
C:	branch zero	2	`if (R[d] == 0) pc ← addr`
D:	branch positive	2	`if (R[d] > 0) pc ← addr`
E:	jump register	1	`pc ← R[t]`
F:	jump and link	2	`R[d] ← pc; pc ← addr`



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: 0001 0001 0010 0011 (machine code)
- Symbolic version ADD R1, R2, R3 (assembly)

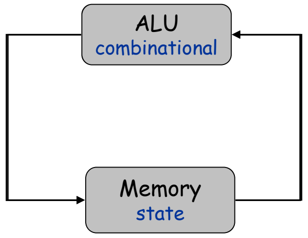
Machine language

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.

Lecture plan

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

(The assembler will be covered in chapter 6).

Typical machine language commands (3 types)

- ALU operations
- Memory access operations

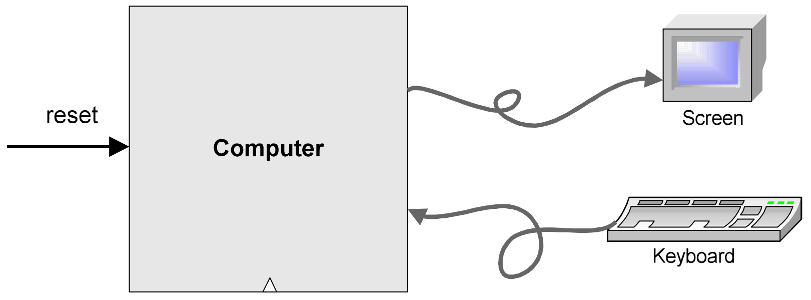
  (addressing mode: how to specify operands)
  - Immediate addressing, LDA R1, 67 // R1=67
  - Direct addressing, LD R1, 67 // R1=M[67]
  - Indirect addressing, LDI R1, R2 // R1=M[R2]
- Flow control operations

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,R2,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:

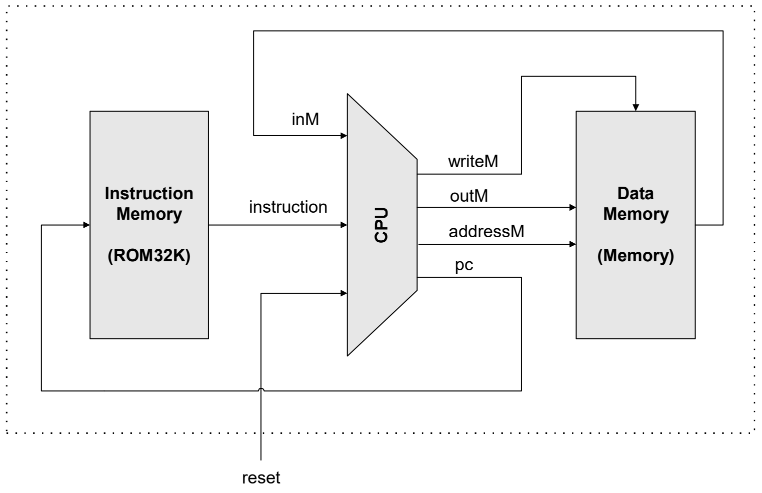


The Hack computer

- The ROM is loaded with a Hack program
- The reset button is pushed
- The program starts running

The Hack computer

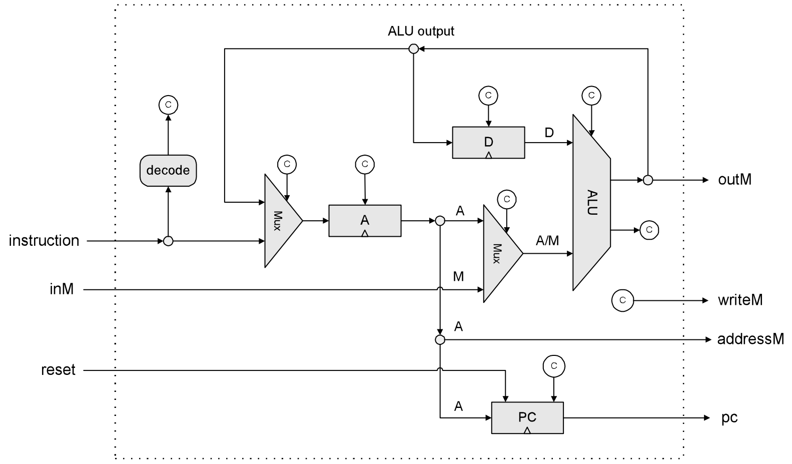
A 16-bit machine consisting of the following elements:



Both memory chips are 16-bit wide and have 15-bit address space.

The Hack computer (CPU)

A 16-bit machine consisting of the following elements:



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.

Why A-instruction?

In TOY, we store address in the instruction (fmt #2). But, it is impossible to pack a 15-bit address into a 16-bit instruction. So, we have the A-instruction for setting addresses if needed.

Example:

```
@21
```

Effect:

- Sets the A register to 21
- RAM[21] becomes the selected RAM register M

The A-instruction

```
@value      // A ← value
```

Used for:

- Entering a constant value (**A = value**)
- Selecting a **RAM** location (**register = RAM[A]**)
- Selecting a **ROM** location (**PC = A**)

Coding example:

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

```
@17    // A = 17
D = M  // D = RAM[17]
M = -1 // RAM[17]=-1
```

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

The C-instruction

dest = comp ; jump

Both dest and jump are optional.

First, we compute something.

Next, optionally, we can store the result, or use it to jump to somewhere to continue the program execution.

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, A, D, M, MD, AM, AD, AMD**

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

Compare to zero. If the condition holds, jump to ROM[A]

The C-instruction

```
dest = comp ; jump
```

- Computes the value of comp
- Stores the result in dest
- If (the condition jump compares to zero is true), goto the instruction at ROM[A].

The C-instruction

```
dest = comp ; jump
```

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

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

Example: set the D register to -1

```
D = -1
```

Example: set RAM[300] to the value of the D register minus 1

```
@300
M = D-1
```

Example: if ((D-1) == 0) goto ROM[56]

```
@56
D-1; JEQ
```

Hack programming reference card

Hack commands:

A-command: @value // set A to value

C-command: dest = comp ; jump // 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 = M, D, A, MD, AM, AD, AMD, or null

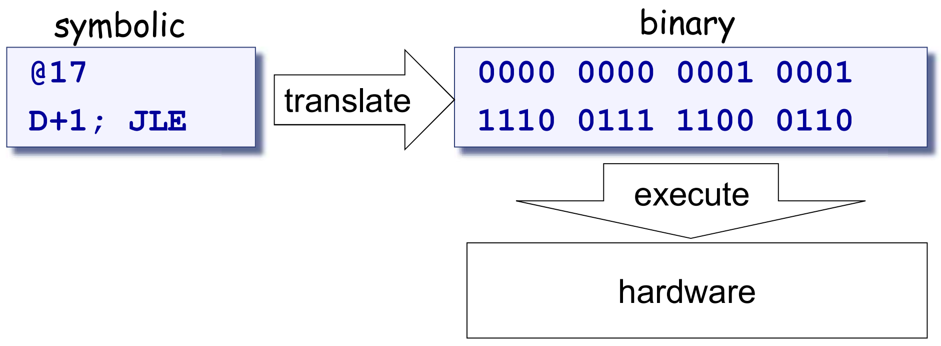
jump = JGT , JEQ , JGE , JLT , JNE , JLE , JMP, or null

In the command dest = comp; jump, the jump materialzes if (comp jump 0) is true. For example, in D=D+1,JLT, we jump if D+1 < 0.

The Hack machine language

Two ways to express the same semantics:

- Binary code (machine language)
- Symbolic language (assembly)



```
@17
D+1; JLE
```

```
0000 0000 0001 0001
1110 0111 1100 0110
```

The A-instruction

symbolic

`@value`

- *value* is a non-negative decimal number <= 2^{15}-1 or
- A symbol referring to such a constant

binary

`0value`

- *value* is a 15-bit binary number

Example

`@21`

`0000 0000 0001 0101`

The C-instruction

symbolic

dest = *comp* ; *jump*

binary

$111A\ C_1C_2C_3C_4\ C_5C_6\ D_1D_2\ D_3J_1J_2J_3$

opcode

not used

comp

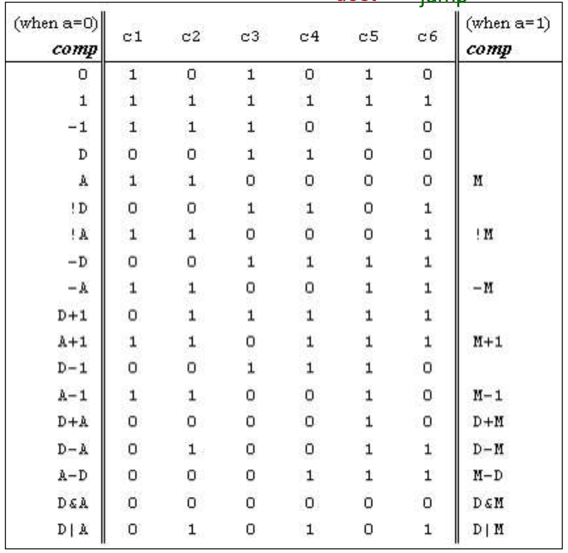
dest

jump

The *C*-instruction

$111A\ C_1C_2C_3C_4\ C_5C_6\ D_1D_2\ D_3J_1J_2J_3$

comp dest jump



(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

The *C*-instruction

$111A\ C_1C_2C_3C_4\ C_5C_6\ D_1D_2\ D_3J_1J_2J_3$

comp dest jump

	A	D	M	
dest	*d*	*d*	*d*	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
DM	0	1	1	D register and RAM[A]
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
ADM	1	1	1	A register, D register, and RAM[A]

The *C*-instruction

$$\texttt{111A}\ \underbrace{C_1C_2C_3C_4\ C_5C_6}_{\text{comp}}\ \underbrace{D_1D_2\ D_3}_{\text{dest}}\underbrace{J_1J_2J_3}_{\text{jump}}$$

jump	< *j*	= *j*	> *j*	effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	Unconditional jump

Hack assembly/machine language

Source code (example)

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i    // if i>RAM[0] goto WRITE
   D=M
   @R0
   D=D-M
   @WRITE
   D;JGT
   @i    // sum += i
   D=M
   @sum
   M=D+M
   @i    // i++
   M=M+1
   @LOOP // goto LOOP
   0;JMP
(WRITE)
   @sum
   D=M
   @R1
   M=D  // RAM[1] = the sum
(END)
   @END
   0;JMP
```

assemble

Hack assembler or CPU emulator

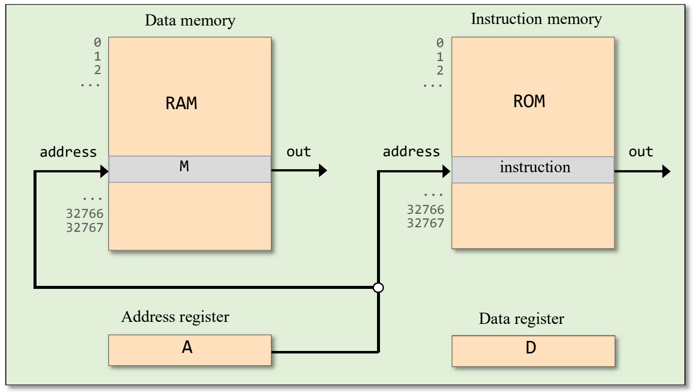
Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

We will focus on writing the assembly code.

Working with registers and memory

- D: data register
- A: address/data register
- M: the currently selected memory cell, M=RAM[A]



Hack programming exercises

Exercise: Implement the following tasks using Hack commands:

1. Set `D` to `A-1`
2. Set both `A` and `D` to `A + 1`
3. Set `D` to `19`
4. `D++`
5. `D=RAM[17]`
6. Set `RAM[5034]` to `D - 1`
7. Set `RAM[53]` to `171`
8. Add `1` to `RAM[7]`, and store the result in `D`.

Hack programming exercises

Exercise: Implement the following tasks using Hack commands:

1. Set `D` to `A-1`
2. Set both `A` and `D` to `A + 1`
3. Set `D` to `19`
4. D++
5. D=RAM[17]
6. Set `RAM[5034]` to `D - 1`
7. Set `RAM[53]` to `171`
8. Add `1` to `RAM[7]`, and store the result in `D`.

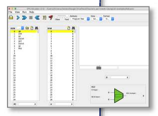
```
1. D = A-1
2. AD=A+1
3. @19
   D=A
4. D=D+1
5. @17
   D=M
6. @5034
   M=D-1
7. @171
   D=A
   @53
   M=D
8. @7
   D=M+1
```

A simple program: add two numbers (demo)

`Add.asm` (example)

```
// Computes: RAM[2] = RAM[0] + RAM[1]
// D = RAM[0]
@0
D=M
// D = D + RAM[1]
@1
D=D+M
// RAM[2] = D
@2
M=D
```

Load into the CPU emulator



Binary

```
0000000000000000
1000010010001101
0000000000000001
1010011001100001
0000000000000010
1110010010010011
```

Execute

When loading a symbolic program into the CPU emulator, the emulator translates it into binary code (using a built-in assembler)

Terminate properly

- To avoid malicious code, you could terminate your program with an infinite loop, such as

```
@6
0; JMP
```

```
00: @0
01: D=M
02: @1
03: D=D+M
04: @2
05: M=D
```

Built-in 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

- R0, R1, …, R15 : virtual registers
- SCREEN and KBD : base address of I/O memory maps
- Others: used in the implementation of the Hack Virtual Machine
- Note that Hack assembler is case-sensitive, R5 != r5

Branching

```
// Program: branch.asm
// if R0>0
//    R1=1
// else
//    R1=0
```

Branching

```
// Program: branch.asm
// if R0>0
//    R1=1
// else
//    R1=0


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

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

   @R1
   M=0             // R1=0
   @10
   0; JMP          // go to end

   @R1
   M=1             // R1=1

   @10
   0; JMP
```

Branching

```
// Program: branch.asm
// if R0>0
//    R1=1
// else
//    R1=0


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

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

   @R1
   M=0             // R1=0
   @10
   0; JMP          // go to end

   @R1
   M=1             // R1=1

   @10
   0; JMP
```

Branching with labels

```
// Program: branch.asm
// if R0>0
//    R1=1
// else
//    R1=0


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

   @POSTIVE        <= refer a label
   D; JGT          // If R0>0 goto 8

   @R1
   M=0             // R1=0
   @END
   0; JMP          // go to end
(POSTIVE)          <= declare a label
   @R1
   M=1             // R1=1
(END)
   @10
   0; JMP
```

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
12	
13	
14	
15	
16	

IF logic – Hack style

High level:

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

Hack:

```
   D ← 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

Coding examples (practice)

Exercise: Implement the following tasks using Hack commands:

1. goto 50
2. if D==0 goto 112
3. if D<9 goto 507
4. if RAM[12] > 0 goto 50
5. if sum>0 goto END
6. if x[i]<=0 goto NEXT.

Coding examples (practice)

Exercise: Implement the following tasks using Hack commands:

1. goto 50
2. if D==0 goto 112
3. if D<9 goto 507
4. if RAM[12] > 0 goto 50
5. if sum>0 goto END
6. if x[i]<=0 goto NEXT.

```
1. @50
   0; JMP
2. @112
   D; JEQ
3. @9
   D=D-A
   @507
   D; JLT
4. @12
   D=M
   @50
   D; JGT
5. @sum
   D=M
   @END
   D; JGT
6. @i
   D=M
   @x
   A=D+M
   D=M
   @NEXT
   D; JLE
```

variables

```
// Program: swap.asm
// temp = R1
// R1 = R0
// R0 = temp
```

variables

```
// Program: swap.asm
// 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
```

- When a symbol is encountered, the assembler looks up a symbol table
- If it is a new label, assign a number (address of the next available memory cell) to it.
- For this example, temp is assigned with 16.
- If the symbol exists, replace it with the number recorded in the table.
- With symbols and labels, the program is easier to read and debug. Also, it can be relocated.

Hack program (exercise)

Exercise: Implement the following tasks using Hack commands:

1. `sum = 0`
2. `j = j + 1`
3. `q = sum + 12 – j`
4. `arr[3] = -1`
5. `arr[j] = 0`
6. `arr[j] = 17`

Hack program (exercise)

Exercise: Implement the following tasks using Hack commands:

1. `sum = 0`
2. `j = j + 1`
3. `q = sum + 12 – j`
4. `arr[3] = -1`
5. `arr[j] = 0`
6. `arr[j] = 17`

```
1. @sum
   M=0
2. @j
   M=M+1
3. @sum
   D=M
   @12
   D=D+A
   @j
   D=D-M
   @q
   M=D
```

```
4. @arr
   D=M
   @3
   A=D+A
   M=-1
5. @j
   D=M
   @arr
   A=D+M
   M=0
```

```
6. @j
   D=M
   @arr
   D=D+M
   @ptr
   M=D
   @17
   D=A
   @ptr
   A=M
   M=D
```

WHILE logic – Hack style

High level:

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

Hack:

```
(LOOP)
     D ← condition
     @END
     D;JNE
     code block 1
     @LOOP
     0;JMP
(END)
     code block 2
```

Hack convention:

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

Complete program example

C language code:

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

Hack assembly convention:

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

Complete program example

Pseudo code:

```
i = 1;
sum = 0;
LOOP:
   if (i>100) goto END
   sum += i;
   i++;
   goto LOOP
END:
```

Hack assembly convention:

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

Demo
CPU emulator

Hack assembly code:

```
// Adds 1+...+100.
      @i      // i refers to some RAM location
      M=1     // i=1
      @sum    // sum refers to some RAM 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 goto 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
```

Example

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

```
Pseudo code:
```

Example

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

```
Pseudo code:

i = 0

(LOOP)
   if (i-n)>=0 goto END
   arr[i] = -1
   i++
   goto LOOP
(END)
```

Example

```
// for (i=0; i<n; i++)
//      arr[i] = -1;
   @i
   M=0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @END
   D; JGE

   @arr
   D=M
   @i
   A=D+M
   M=-1

   @i
   M=M+1

   @LOOP
   0; JMP
(END)
```

```
Pseudo code:

i = 0

(LOOP)
   if (i-n)>=0 goto END
   arr[i] = -1
   i++
   goto LOOP
(END)
```

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": any operation that needs to operate on the RAM must be specified using two commands: an **A**-command to address the RAM, and a subsequent **C**-command to operate on it
- A Macro-language can be easily developed
  - D=D+M[XXX] => @XXX followed by D=D+M
  - GOTO YYY => @YYY followed by 0; JMP
- A Hack assembler is needed and will be discusses and developed later in the course.