Machine Language – Introduction to Computer (計算機概論)

Winston H. Hsu (徐宏民)
National Taiwan University, Taipei

November 14, 2022

Office: R512, CSIE Building
Communication and Multimedia Lab (通訊與多媒體實驗室)
http://winstonhsu.info

The majority of the slides are from the chapter 4 in [Nisan and Schocken] and few from the reference [Harris]

Administrative Matters

- Today's lecture
  - Section 4.1 – 4.3 [Nisan and Schocken]

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:

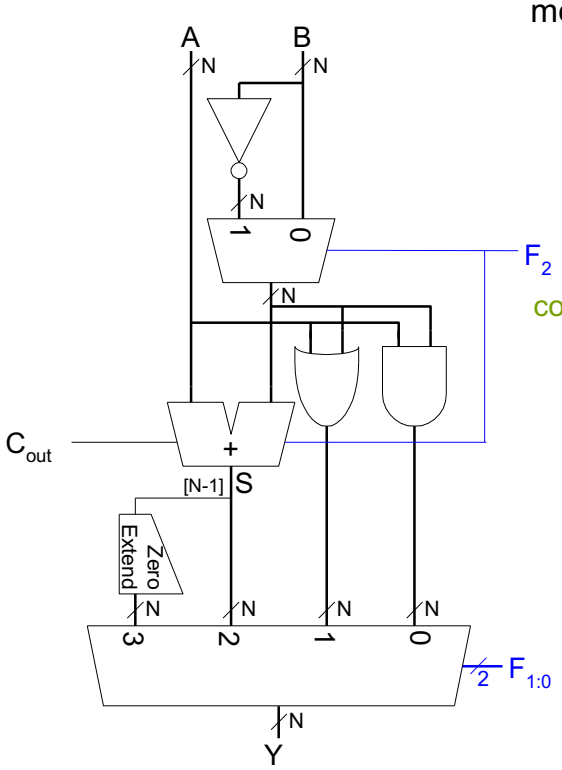
```
1010 0001 0010 1001
```

```
ADD R1, R2, R9
```

  - Binary version (machine codes)
  - Symbolic version (assembly)
- 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

ALU Design (Recap)

ALU combines a variety of mathematical and logical operations into a single unit and forms the heart of most computer systems.



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Also need memory (register) access and program control

Machine Language Example

- For example, the language designer can decide that
  - the operation code 1010 will be represented by the mnemonic (助記符) **add** and
  - that the registers of the machine will be symbolically referred to using the symbols R0, R1, R2, and so forth.
  - Using these conventions, one can specify machine language instructions either directly, as
  - **1010** 0011 0001 1001,
  - or symbolically, as, say, ADD R3,R1,R9.
  - **Assembly** (language) to **binary codes** by **assembler**

High-Level Language vs. Machine Language

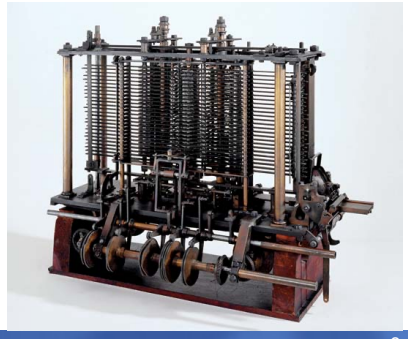
- As opposed to high-level languages, whose basic design goals are generality and power of expression,
- the goal of machine language's design is direct execution in, and total control of, a given hardware platform.
- High-level language: C, C++, C#, F#, Python, Java, Pascal, Perl, FORTRAN, COBOL, etc.
- Machine language – the fine line where hardware and software meet; **we still use them now**!!
- How if some functions not supported in the hardware?

Outline

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

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was the only legitimate child of the poet Lord Byron (拜倫)
- Search wikipedia "Augusta Ada King-Noel, Countess of Lovelace"

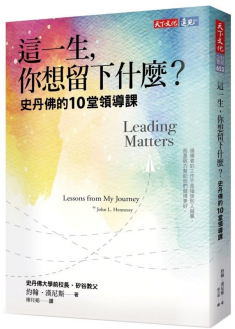


Assembly Language

- Instructions: commands in a computer's language
  - **Assembly language**: human-readable format of instructions
  - **Machine language**: computer-readable format (1's and 0's)
  - **Assembler** does the translation (from readable to binary)
- Example: MIPS architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco
- Once you've learned one architecture, it's easy to learn others

John Hennessy

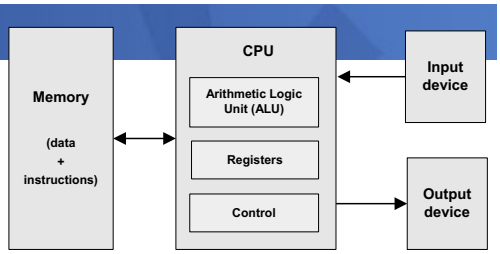
- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Co-invented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold



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:

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]**

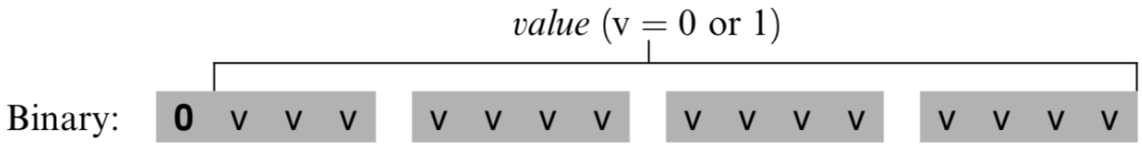
Processing: **ALU**, capable of computing various functions

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.

A-instruction (Hack Machine ONLY)



- The A-instruction for three different purposes.
  - Providing the only way to enter a constant into the computer under program control
  - Setting the stage for a subsequent C-instruction designed to manipulate a certain data memory location, by first setting A to the address of that location.
  - Setting the stage for a subsequent C-instruction that specifies a jump, by first loading the address of the jump destination to the A register.

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`)

Coding example:

```
@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`)

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

Addressing Mode

- **Addressing modes** – ways of specifying the address of the required memory word; many kinds for complex instructions, few here
- Direct addressing: express a specific address or use a symbol that refers to a specific address

```
LOAD R1,67      //  R1←Memory[67]
```

- Immediate addressing: this form of addressing is used to load constants

```
LOADI R1,67     //  R1←67
```

- Indirect addressing: the instruction specifies a memory location that holds the required address.

```
// Translation of x=foo[j] or x=*(foo+j):
ADD R1,foo,j    // R1←foo+j
LOAD* R2,R1     // R2←Memory[R1]
STR R2,x        // x←R2
```


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}

the three-character string **A+D**, taken as a whole, is treated as a single assembly mnemonic, designed to code a single ALU operation.

Exercise: Implement the following tasks using Hack commands:

- Set `D` to `A-1`
- Set both `A` and `D` to `A + 1`

```
AD=A+1
```

- Set `D` to `19`

```
@19
D=A
```

- Set both `A` and `D` to `A + D`
- Set `RAM[5034]` to `D - 1`

```
@5034
M=D-1
```

- Set `RAM[53]` to `171`

```
@171
D=A
@53
M=D
```

- Add `1` to `RAM[7]`, and store the result in `D`.

```
@7
D=M+1
```

C language

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

Hack machine language

```
// Adds 1+...+100.
       @i      // i refers to some mem. location.
       M=1     // i=1
       @sum    // sum refers to some mem. location.
       M=0     // sum=0   // RAM[sum] = 0
 (LOOP)
       @i
       D=M     // D=i
       @100              // register A= 100
       D=D-A   // D=i-100
       @END              // register A= @END
       D;JGT   // If (i-100)>0 goto END
       @i
       D=M     // D=i
       @sum              // register A= @sum
       M=D+M   // sum=sum+i
       @i
       M=M+1   // i=i+1
       @LOOP             // register A= @LOOP
       0;JMP   // Goto LOOP
  (END)
       @END
       0;JMP   // Infinite loop
```

What are the corresponding binary codes?

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	3012
sum	4500
q	3812
arr	20561

(All symbols and values are arbitrary examples)

Exercise: Implement the following tasks using Hack commands:

- sum = 0
- j = j + 1

```
@j
M=M+1
```

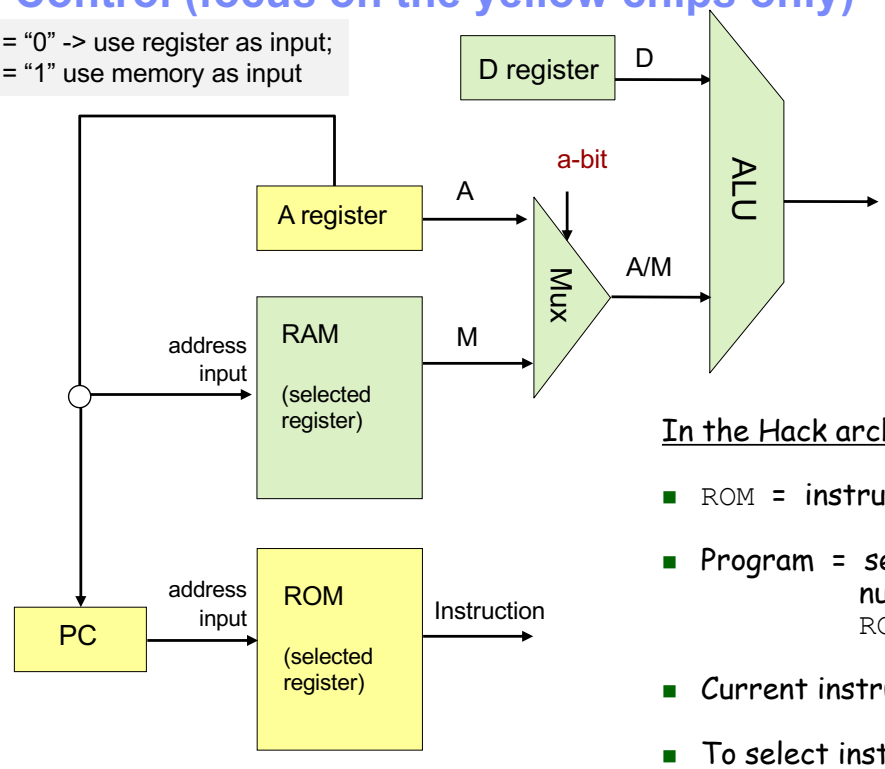
- q = sum + 12 − j
- arr[3] = −1
- arr[j] = 0

```
@j
D=A
@arr
A=D+A
M=0
```

- arr[j] = 17
- etc.

Control (focus on the yellow chips only)

a = "0" -> use register as input;
a = "1" use memory as input



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

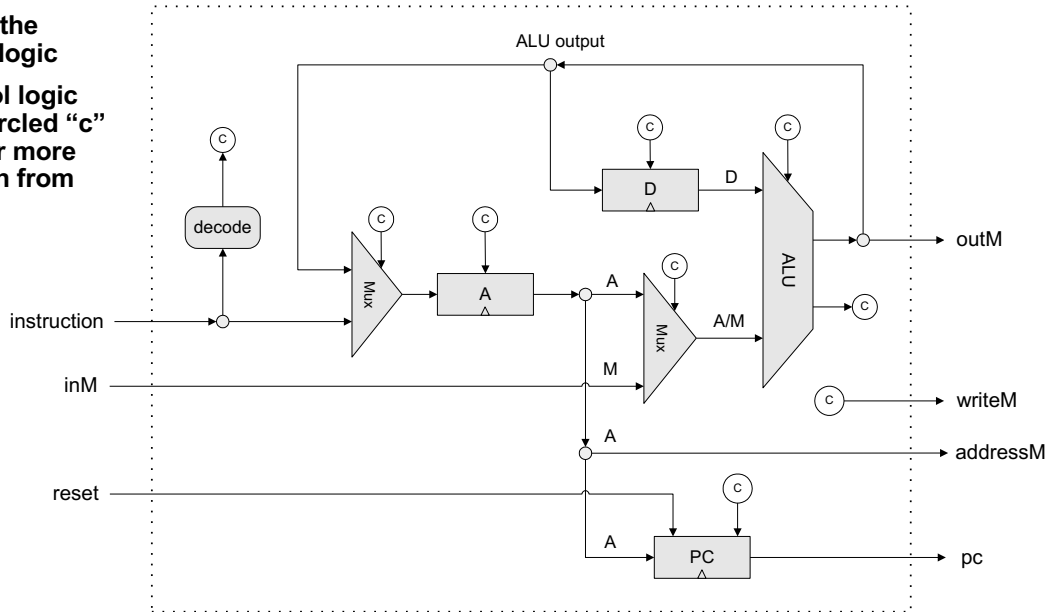
CPU implementation

dest = comp; jump

	comp								dest			jump			
binary:	1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

Chip diagram:

- Includes most of the CPU's execution logic
- The CPU's control logic is hinted: each circled "c" represents one or more control bits, taken from the instruction
- The "decode" bar does not represent a chip, but rather indicates that the instruction bits are decoded somehow.



Cycle:

- Execute
- Fetch

Execute logic:

- Decode
- Execute

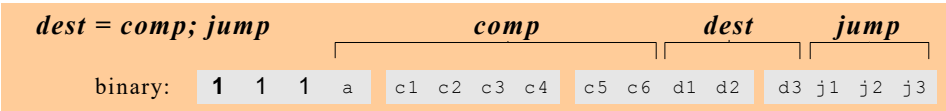
Fetch logic:

If there should be a jump,
  set PC to A
else set PC to PC+1

Resetting the computer:

Set reset to 1,
then set it to 0.

The C-instruction



7-bit pattern can potentially code 128 different functions

(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* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

C-instruction Examples

- Suppose we want,
- To have the ALU compute **D-1**,the current value of the D register minus 1. Can be done by issuing the instruction 111**0 0011 10**00 0000
- To compute the value of D|M, we issue the instruction 111**1 0101 01**00 0000.
- What's
  - 0000 0000 0000 0111
  - 1111 1101 1101 1000

C-instruction Examples (Jump)

Logic

```
if Memory[3]=5 then goto 100
else goto 200
```

Implementation

```
@3
D=M     // D=Memory[3]
@5
D=D-A   // D=D-5
@100
D;JEQ   // If D=0 goto 100
@200
0;JMP   // Goto 200
```

the three-character string **D-A**, taken as a whole, is treated as a single assembly mnemonic, designed to code a single ALU operation.

Coding examples (practice)

Exercise: Implement the following tasks using Hack commands:

- goto 50

```
@50
0;JMP
```

- if D==0 goto 112

```
@112
D;JEQ
```

- if D<9 goto 507
- if RAM[12] > 0 goto 50

```
@12
D=M
@50
D;JGT
```

- if sum>0 goto END
- if x[i]<=0 goto NEXT.

What are the corresponding binary codes?

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 , MD , A , 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.

Symbol table:

sum	2200
x	4000
i	6151
END	50
NEXT	120

(All symbols and values in are arbitrary examples)

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
```

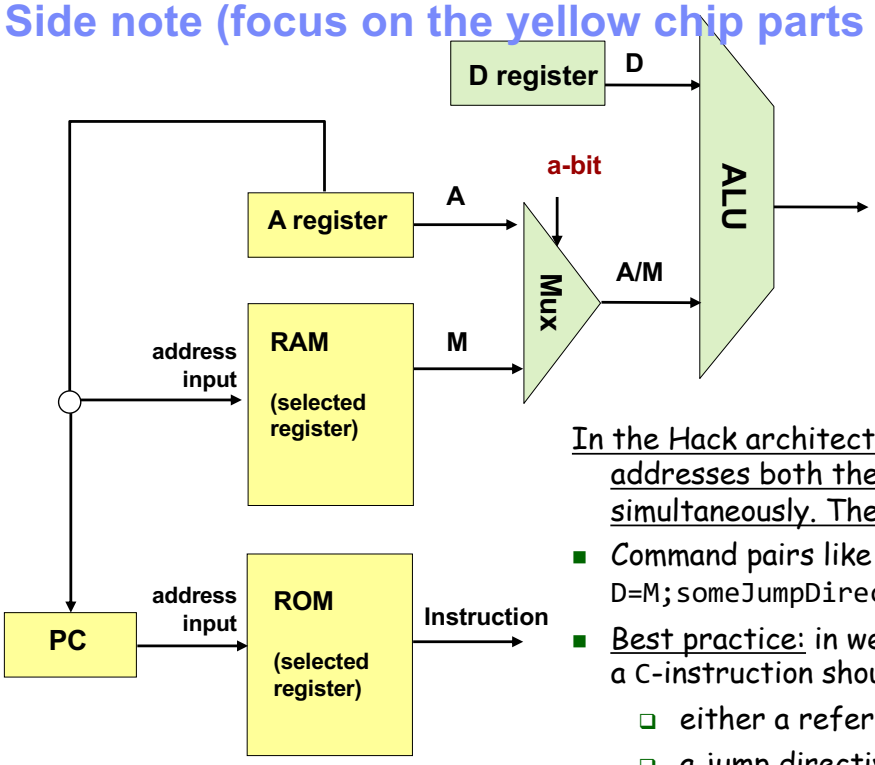
Hack:

```
(LOOP)
      D ← 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 (focus on the yellow chip parts only)



In the Hack architecture, the A register addresses both the RAM and the ROM, simultaneously. Therefore:

- Command pairs like `@addr` followed by `D=M;someJumpDirective` 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 language code:

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

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
```

Hack assembly convention:

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

Symbols in Hack assembly programs

Symbols created by Hack programmers and code generators:

- 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 (within the program, **XXX** is called "label")
- 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 "automatically" assigned a unique RAM address, starting at RAM address 16

  By convention, Hack programmers use lower-case and upper-case letters for variable names and labels, respectively.

Predefined symbols:

- I/O pointers: The symbols **SCREEN** and **KBD** are "automatically" predefined to refer to RAM addresses 16384 and 24576, respectively (base addresses of the Hack platform's *screen* and *keyboard* memory maps)
- Virtual registers: covered in future lectures.
- VM control registers: covered in future lectures.

Q: Who does all the "automatic" assignments of symbols to RAM addresses?

A: The *assembler*, which is the program that translates symbolic Hack programs into binary Hack program. As part of the translation process, the symbols are resolved to RAM addresses. (more about this in future lectures)

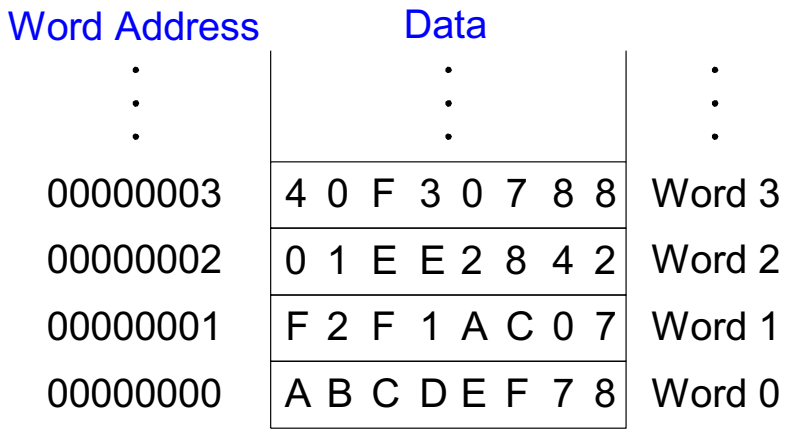
```
// Typical symbolic
// Hack code, meaning
// not important
   @R0
   D=M
   @INFINITE_LOOP
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @addr
   M=D
(LOOP)
   @addr
   A=M
   M=-1
   @addr
   D=M
   @32
   D=D+A
   @addr
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP
```


Perspective

- Hack is a simple machine language
- User friendly syntax: **D=D+A** instead of **ADD D,D,A (varies from different hardware platforms)**
- 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**, a set of frequently used commands defined in advance (like function) that assembler can translate, can be easily developed.
- A Hack assembler is needed and will be discusses and developed later in the course. **Who writes the first assembler?**

Word-Addressable Memory

- Each 32-bit data word has a unique address



Word-Addressable Memory (Another Assembly Language)

- Memory read called ***load***
- **Mnemonic:** *load word* (lw)
- **Format:**

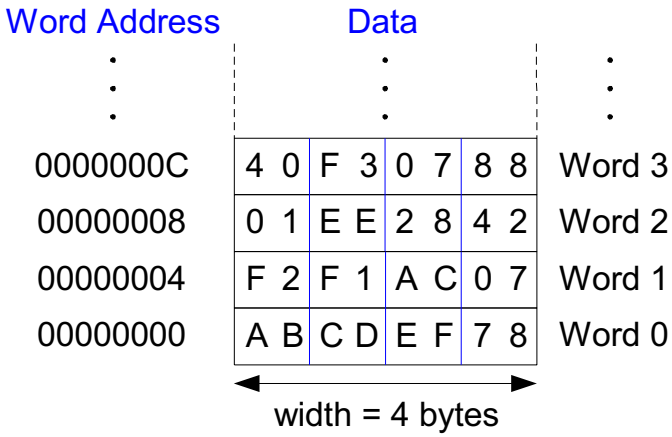
  lw $s0, 5($t1)

- **Address calculation:**
  - add *base address* ($t1) to the *offset* (5)
  - address = ($t1 + 5)
- **Result:**
  - $s0 holds the value at address ($t1 + 5)

Any register may be used as base address

Byte-Addressable Memory

- Each data byte has unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address increments by 4



Reading Byte-Addressable Memory

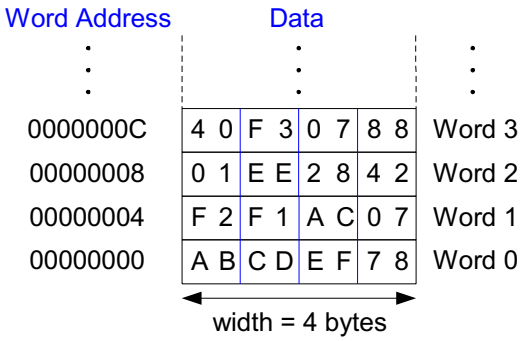
- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)
- MIPS is byte-addressed, not word-addressed

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into $s3.
- $s3 holds the value 0xF2F1AC07 after load

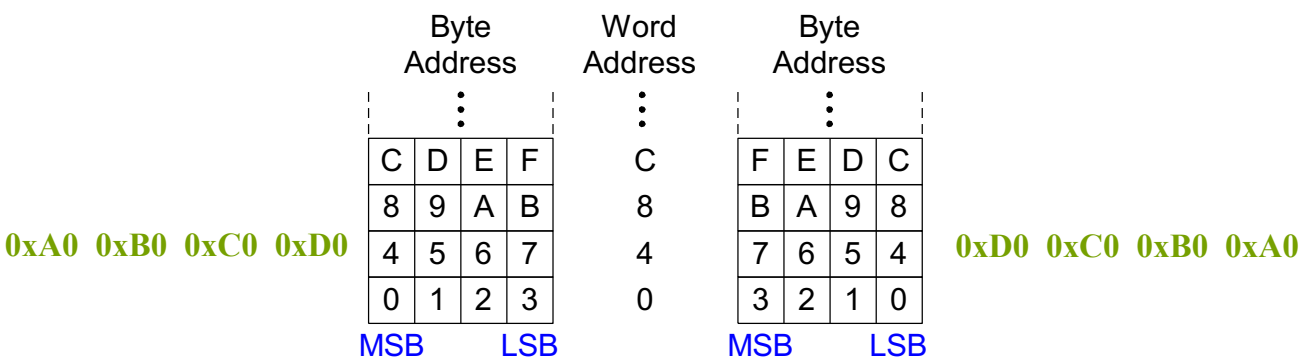
MIPS assembly code

```
lw $s3, 4($0)  # read word at address 4 into $s3
```



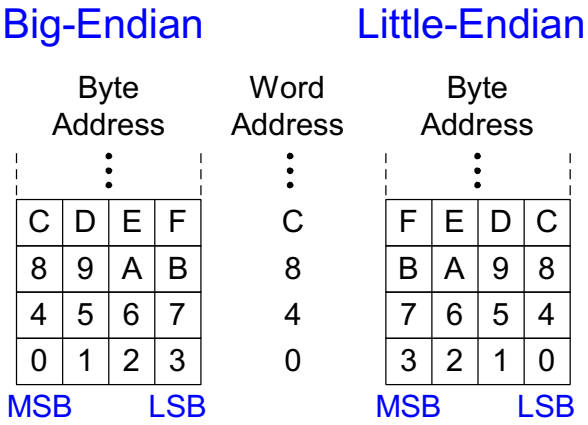
Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end (Intel)
- **Big-endian:** byte numbers start at the big (most significant) end (Sun)
- **Word address** is the **same** for big- or little-endian
- For example, **0xA0 0xB0 0xC0 0xD0**



Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels (葛利佛遊記)*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!



Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains 0x23456789
- After following code runs on big-endian system, what value is `$s0`?
- In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
- Little-endian: 0x00000067