Introduction to Assembly Programming

상명대학교 컴퓨터과학과

Assembly language

- Reasons for assembly programming
  - To improve performance
  - There are sections of code which must be written in assembly language
  - To learn how a particular CPU works
- Assembly language is
  - Machine dependent
  - But it follows universal format

Machine language vs. Assembly language

- Ex) To add *EAX* and *EBX* registers together and store the result back into *EAX*

- Machine language: numerical-coded instruction

$$\begin{array}{cc} 0\quad 1 & C \qquad 3 \\ \underbrace{\boxed{00000001}}_{\text{op code}} & \boxed{\underbrace{11}_{\text{ModRM}}\ \underbrace{000}_{\text{EAX}}\ \underbrace{011}_{\text{EBX}}} \end{array}$$

- Assembly language

  add eax, ebx

The four field format

- Assembly language program consists of lines
- Every statement in a line consists of four fields
  - label field
  - mnemonic (opcode) field
  - operand field
  - comment field

```
LABEL     OPCODE     OPERANDS      COMMENT
DATA1       db       00001000b     ;Define DATA1 as decimal 8
START:      mov      eax, ebx      ;Copy ebx to eax
```

The four field format

- Label field
  - specifies the target of a jump instruction
  - jump is the same as *goto*

- Mnemonic (opcode) field
  - an instruction specifier
  - *MOV*, *ADD*, *SUB*, etc
  - the word mnemonic suggests that it makes the machine code easy to remember

The four field format

- Operand field
  - objects on which the instruction is operating
  - Each instruction have a fixed number of operands (0 ~ 3)
    - *ADD* takes two operands, *JMP* takes one operand, …
  - if there is more than one operand, they are separated by commas
  - Operands can have the following types: register, memory, immediate, implied

- Comment field
  - contains documentation
  - It begins with semicolon
  - documentation is especially important in assembly language, because it is hard to read
  - A line may consist of nothing but a comment

Example program

```
      ;
      ; Greatest common divisor program
      ;
      MOV EDX, 0      ; 0 is the only Edlinas input port
      IN EAX,[DX]     ; Get the user's first input
      MOV ECX, EAX    ; Get the input out of harm's way
      IN EAX,[DX]     ; Get the user's second input
      MOV EDX, EAX    ; Use EDX for the larger of the two inputs
ORD:  SUB EAX, ECX    ; Use EAX as a working copy of EDX
      JZ GCD          ; When equality is obtained we are done.
      JNS NXT         ; We want EDX to be larger. No swap needed
      MOV EAX, ECX    ; Swap EDX and ECX (Takes three MOV's)
      MOV ECX, EDX
NXT:  MOV EDX, EAX    ; If there was no swap then EDX = EDX-ECX
      JMP ORD         ; End of the loop
GCD:  MOV EAX, EDX    ; The GCD is in EDX
      MOV EDX, 1      ; We need EDX for the output port number
      OUT [DX],EAX    ; Display the answer to the user
      RET
```

The *MOV* instruction

- *MOV* reg, imm
  - Reg stands for register
  - Imm stands for immediate value
  - Example
    - *MOV EAX*, 54
    - *MOV AX*, 036H
    - *MOV AL*, 'A'
    - *MOV AL*, -129 ; not valid
    - *MOV AL*, 999 ; not valid

The *MOV* instruction

- *MOV reg, reg*
  - Copies from the second reg into the first one
  - Example
    - *MOV EAX, EBX*
    - *MOV EBX, DX* ; **not valid**

- Ambiguity problem: *MOV BL, AH*
  - *AH* : register '*AH*' or hex number 'A'
  - NASM requires all hex numbers begin with one of the digits 0, 1, …, 9. -> *AH* is the name of register!

Addition Instruction

- *ADD reg, imm*
  - Add the immediate value *imm* to the register *reg*
  - Example
    - *ADD BL*, 10 ; let *BL* = *BL* + 10

- *ADD reg, reg*
  - Add the contents of the second register to the first one
  - Example
    - *ADD BL, AL* ; let *BL* = *BL* + *AL* - ***AL* is not changed !!**

Subtraction Instruction

- *SUB reg, imm*
  - Subtract the immediate value from the register
  - Example
    - *SUB BL*, 10 ; let *BL* = *BL* - 10
- *SUB reg, reg*
  - Subtract the contents of the second register from the first
  - Example
    - *SUB BL, AL* ; let *BL* = *BL* – *AL* , *AL* is not changed

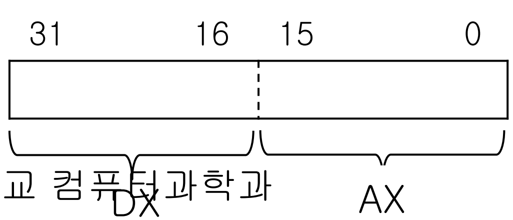
Multiplication Instruction

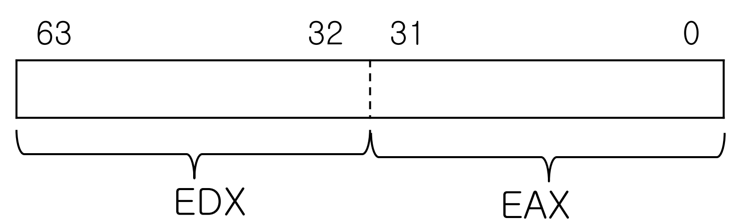
- *MUL reg*
  - Multiplier is in *reg*
  - Multiplicand is always in *A* register and result are always in *A* and *D* register
  - *MUL* command has **no immediate form**
  - Example
    - *MUL BH* ; let *AX* = *AL* x *BH*
    - *MUL BX* ; let *DX*:*AX* = *AX* x *BX*
    - *MUL EBX* ; let *EDX*:*EAX* = *EAX* x *EBX*

P × Q

multiplicand
곱해지는 수/피승수
A 레지스터

Multiplier
곱하는 수/승수
오퍼랜드에 표기





Division Instruction

- *DIV reg*
  - *DIV* resemble *MUL* syntax; and is essentially its inverse

	dividend	remainder	quotient
32-bit form	*EDX:EAX*	*EDX*	*EAX*
16-bit form	*DX:AX*	*DX*	*AX*
8-bit form	*AX*	*AH*	*AL*

P ÷ Q

dividend
나뉘는수/피젯수
A와 D 레지스터

divisor
나누는수/젯수
오퍼랜드에 표ㄱ

  - Example
    - When $AX = 17$, $BH = 3$

      DIV BH ; $AH = 2$ and $AL = 5$

Constants

- Numeric constants

```
mov     ax,100          ; decimal
mov     ax,0a2h         ; hex
mov     ax,$0a2         ; hex again: the 0 is required
mov     ax,0xa2         ; hex yet again
mov     ax,777q         ; octal
mov     ax,777o         ; octal again
mov     ax,10010011b    ; binary
```

- Character constants
  - A character constant consists of up to four charact ers enclosed in either single or double quotes.
  - A character constant with more than one character will be arranged with little−endian order in mind

```
mov eax,'abcd'
```

Constants

- String constants
  - only acceptable to some pseudo−instructions

```
db     'hello'                  ; string constant
db     'h','e','l','l','o'      ; equivalent character constants
dd     'ninechars'              ; doubleword string constant
dd     'nine','char','s'        ; becomes three doublewords
db     'ninechars',0,0,0        ; and really looks like this
```

- Floating-point constants

```
dd     1.2                      ; an easy one
dq     1.e10                    ; 10,000,000,000
dq     1.e+10                   ; synonymous with 1.e10
dq     1.e-10                   ; 0.000 000 000 1
dt     3.141592653589793238462 ; pi
```

Pseudo-Instructions and Directives

- To instruct the assembler to do something or inform the assembler of something
  - Define constants
  - Define memory to store data into
  - Group memory into segments
  - Conditionally include source code
  - Include other files

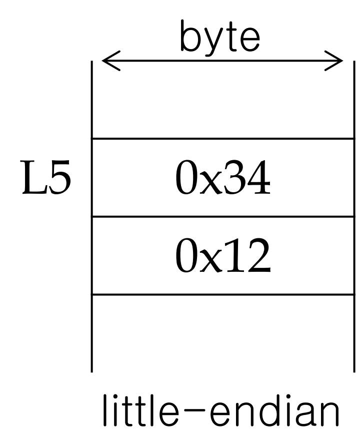
Pseudo-Instructions

- DB and friends: declaring initialized data

```
L1  db   0x55               ; byte labeled L1 with initial value 0x55
L2  db   0x55, 0x56, 0x57   ; three bytes in succession
L3  db   'a', 0x55          ; character constants are OK
L4  db   'hello', 13, 10, '$'   ; so are string constants

L5  dw   0x1234    ; 0x34  0x12
L6  dw   'a'       ; 0x61  0x00 (it's just a number)
L7  dw   'ab'      ; 0x61 0x62  (character constant)
L8  dw   'abc'     ; 0x61 0x62 0x63 0x00 (string)

L9  dd  0x12345678   ; 0x78  0x56  0x34   0x12
L10 dd 1.234567e20   ; floating point constant
L11 dq 1.234567e20   ; double-precision float
```



Pseudo-Instructions

- RESB and friends: declaring uninitialized data

  buffer *resb* 64 *; reserve 64 bytes*

  wordvar *resw* 1 *; reserve a word*

  doublevar *resd* 2 *; reserve two doubles*

  realarray resq 10 *; array of 10 reals*

- TIMES: prefix to repeat instructions or data

  zerobuf times 64 db 0 *; reserve 64 bytes*

  buf times 100 resb 1 ; same as *resb 100*

  buffer db 'hello, world' ; to make the total length of

  times 64-($-buffer) db ' ' ; *buffer* up to 64

- EQU: defining constants

  ten equ 10

Directives

- SECTION or SEGMENT
  - Defines and changes segments
  - Segment names: *.text* *.data* *.bss*

- GLOBAL
  - Exports symbols to other modules

    global _main

    _main:

    ; som code

- EXTERN
  - Imports symbols from other modules

    extern _printf, _scanf

Hello, World

- ***Hello World (using ld):***

```
        section  .data
msg     db    'Hello, world!',0x0A
len     equ  $ - msg              ;length of hello string.

        section  .text
        global _start             ;must be declared for linker (ld)
_start:                           ;we tell linker where is entry point
        mov eax, 4                ;system call number (sys_write)
        mov ebx, 1                ;file descriptor (stdout)
        mov ecx, msg              ;message to write
        mov edx, len              ;message length
        int 0x80                  ;call kernel

        mov eax, 1                ;system call number (sys_exit)
        xor ebx, ebx              ;exit status of this program
        int 0x80
```

Hello, World

- To produce *hello.o* object file:

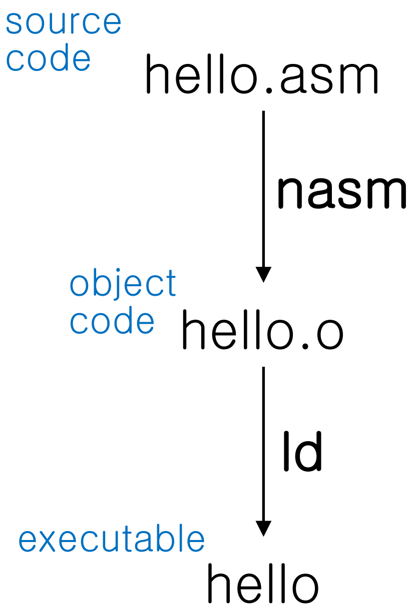
  $ nasm -f elf hello.asm

- To produce *hello.lst* list file too:

  $ nasm -f elf hello.asm –l hello.lst

- To produce *hello* ELF executable:

  $ ld -s -o hello hello.o



hello.lst

```
 1                                               section .data
 2
 3  00000000 48656C6C6F2C20776F-     msg      db      "Hello, world!",0xA
 4  00000009 726C64210A
 5                                   len      equ     $ - msg
 6
 7                                            section .text
 8                                            global _start
 9                                   _start:
10  00000000 B804000000                       mov     eax,4
11  00000005 BB01000000                       mov     ebx,1
12  0000000A B9[00000000]                     mov     ecx,msg
13  0000000F BA0E000000                       mov     edx,len
14  00000014 CD80                             int     0x80
15
16  00000016 B801000000                       mov     eax,1
17  0000001B 31DB                             xor     ebx,ebx
18  0000001D CD80                             int     0x80
```