Introduction to Assembly Programming

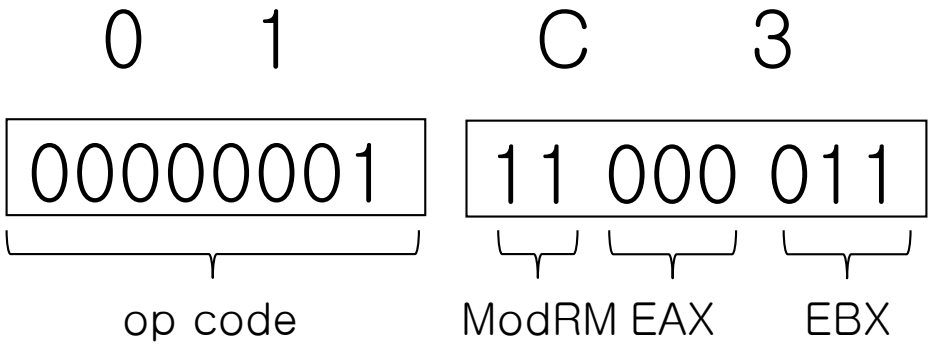
Assembly language

- Reasons for assembly programming
  - To improve performance
  - To learn how a particular CPU works
  - There are sections of code which must be written in assembly language
- 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



- 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*, 36H
    - *MOV AL*, 'A'
    - *MOV AL*, -129 ; not valid - -129 is not in the 8-bit signed range
    - *MOV AL*, 999 ; not valid - 999 will not fit into an 8-bit register

The *MOV* instruction

- *MOV reg, reg*
  - Copies from the second reg into the first one
  - Example
    - *MOV EAX, EBX*
    - *MOV EBX, DX* ; **not valid** - the two registers must be the
      ; same size

- 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*

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*

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

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

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

Directives

- The *equ* directive : to define a symbol

  symbol equ value

- The *%define* directive: to define constant macros

  %define SIZE 100

  mov eax, SIZE

- Data directives: to define room for memory

```
L1   db     0          ; byte labeled L1 with initial value 0
L2   dw     1000       ; byte labeled L2 with initial value 1000
L3   db     110101b    ; byte initialized to binary 110101
L4   db     12h        ; byte initialized to hex 12
```

Directives

```
L5   db      17o         ; byte initialized to octal 17
L6   dd      1A92h       ; double word initialized to hex 1A92
L7   resb    1           ; 1 uninitialized byte
L8   db      "A"         ; byte initialized to ASCII code for A (65)

L9   db      0, 1, 2, 3                      ; define 4 bytes
L10  db      "w","o","r","d", 0              ; define a C string "word"
L11  db      'word', 0                       ; same as L10

L12  times 100  db   0      ; equivalent to 100 (db 0)'s
L13  resw  100              ; reserve room for 100 words
```

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:

  $ 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
```