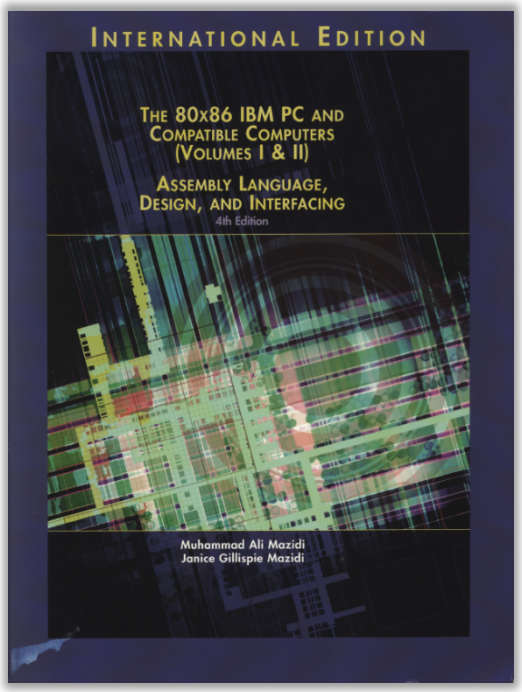
Lecture 04: Assembly Language Programming (1)

Reference Book:

The 80x86 IBM PC and Compatible Computers

Chapter 2

Assembly Language Programming



Programming Languages

- Machine language
  - Binary code for CPU but not human beings
  - *Low-level language*: deals with the internal structure of a CPU
  - Hard to program, poor portability but very efficient
- BASIC, Pascal, C, Fortran, Perl, TCL, Python, ...
  - *High-level languages:* do not have to be concerned with the internal details of a CPU
  - Easy to program, good portability but less efficient

How to convert your program in low/high-level languages into machine language?

Assembly Language Programs

- A series of *statements*
  - Assembly language instructions
    - Perform the real work of the program
  - Directives *(pseudo-instructions)*
    - Give instructions for the *assembler* program about how to translate the program into machine code.
- Consists of multiple segments
  - But CPU can access only one data segment, one code segment, one stack segment and one extra segment (*Why?*)

Form of an statement

- [label:] mnemonic [operands] [;comment]
  - *label* is a reference to this statement
    - Rules for names: each label must be unique; letters, 0-9, (?), (.), (@), (_), and ($); first character cannot be a digit; less than 31 characters
  - ":" is needed if it is an instruction otherwise omitted
  - ";" leads a comment, the assembler omits anything on this line following a semicolon

Shell of a Real Program

- Full segment definition (old fashion)
  - See an example later
- Simplified segment definition

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           DB      52H
DATA2           DB      29H
SUM             DB      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AL,DATA1        ;get the first operand
                MOV     BL,DATA2        ;get the second operand
                ADD     AL,BL           ;add the operands
                MOV     SUM,AL          ;store the result in location SUM
                MOV     AH,4CH          ;set up to return to DOS
                INT     21H             ;
MAIN            ENDP
                END     MAIN            ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

Model Definition

- The **MODEL** directive
  - Selects the size of the memory model
  - SMALL: code <=64KB, data <=64KB
  - MEDIUM: data <=64KB, code >64KB
  - COMPACT: code<=64KB, data >64KB
  - LARGE: data>64KB but single set of data<64KB, code>64KB
  - HUGE: data>64KB, code>64KB
  - TINY: code + data<64KB

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
        .MODEL SMALL
        .STACK 64
        .DATA
DATA1   DB   52H
DATA2   DB   29H
SUM     DB   ?
        .CODE
MAIN    PROC FAR          ;this is the program entry point
        MOV  AX,@DATA     ;load the data segment address
        MOV  DS,AX        ;assign value to DS
        MOV  AL,DATA1     ;get the first operand
        MOV  BL,DATA2     ;get the second operand
        ADD  AL,BL        ;add the operands
        MOV  SUM,AL       ;store the result in location SUM
        MOV  AH,4CH       ;set up to return to DOS
        INT  21H          ;
MAIN    ENDP
        END  MAIN         ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

Simplified Segment Definition

- Simplified segment definition
  - **.CODE, .DATA, .STACK**
  - Only three segments can be defined
  - Automatically correspond to the CPU's CS, DS, SS
  - DOS determines the CS and SS segment registers automatically. DS (and ES) has to be manually specified.

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           DB      52H
DATA2           DB      29H
SUM             DB      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AL,DATA1        ;get the first operand
                MOV     BL,DATA2        ;get the second operand
                ADD     AL,BL           ;add the operands
                MOV     SUM,AL          ;store the result in location SUM
                MOV     AH,4CH          ;set up to return to DOS
                INT     21H             ;
MAIN            ENDP
                END     MAIN            ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

Segments All at a Glance

- Stack segment
- Data segment
  - Data definition
- Code segment
  - Write your statements
  - Procedures definition
    - label **PROC** [**FAR**|**NEAR**]
    - label **ENDP**
  - Entrance proc should be FAR

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           DB      52H
DATA2           DB      29H
SUM             DB      ?
                .CODE
MAIN            PROC    FAR             ;this is the program entry point
                MOV     AX,@DATA        ;load the data segment address
                MOV     DS,AX           ;assign value to DS
                MOV     AL,DATA1        ;get the first operand
                MOV     BL,DATA2        ;get the second operand
                ADD     AL,BL           ;add the operands
                MOV     SUM,AL          ;store the result in location SUM
                MOV     AH,4CH          ;set up to return to DOS
                INT     21H             ;
MAIN            ENDP
                END     MAIN            ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

Quiz

- Can you tell some differences between assembly program and C program?

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
        .MODEL SMALL
        .STACK 64
        .DATA
DATA1   DB    52H
DATA2   DB    29H
SUM     DB    ?
        .CODE
MAIN    PROC  FAR             ;this is the program entry point
        MOV   AX,@DATA        ;load the data segment address
        MOV   DS,AX           ;assign value to DS
        MOV   AL,DATA1        ;get the first operand
        MOV   BL,DATA2        ;get the second operand
        ADD   AL,BL           ;add the operands
        MOV   SUM,AL          ;store the result in location SUM
        MOV   AH,4CH          ;set up to return to DOS
        INT   21H             ;
MAIN    ENDP
        END   MAIN            ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

procedure { FAR / NEAR

```
1   #include<stdio.h>
2   #include<conio.h>
3   void main()
4   {
5   int num,i;
6   printf("Enter a number below 100\n");
7   scanf("%d",&num);
8   for(i=1;i<100;i++)
9   {
10      printf("%d\n",i);
11      if(i==num)
12      break;
13  }
14  getch();
15  }
16
```

funition

Full Segment Definition

- Full segment definition
  - *label* **SEGMENT**
  - *label* **ENDS**
  - You name those labels
  - as many as needed
  - DOS assigns CS, SS
  - Program assigns DS (manually load data segments) and ES

```
DaSeg1 segment
    str1 db 'Hello World! $'
DaSeg1 ends

StSeg segment
    dw   128  dup(0)
StSeg ends

CoSeg segment
    start   proc far
        assume cs:CoSeg, ss:StSeg

        mov ax, DaSeg1      ; set segment registers:
        mov ds, ax
        mov es, ax

        call subr        ;call subroutine

        mov ah, 1         ; wait for any key....
        int 21h

        mov ah, 4ch       ; exit to operating system.
        int 21h
   start endp

    subr proc

        mov dx, offset str1
        mov ah, 9
        int 21h           ; output string at ds:dx

        ret
    subr endp

CoSeg ends

      end start   ; set entry point and stop the assembler.
```

Program Execution

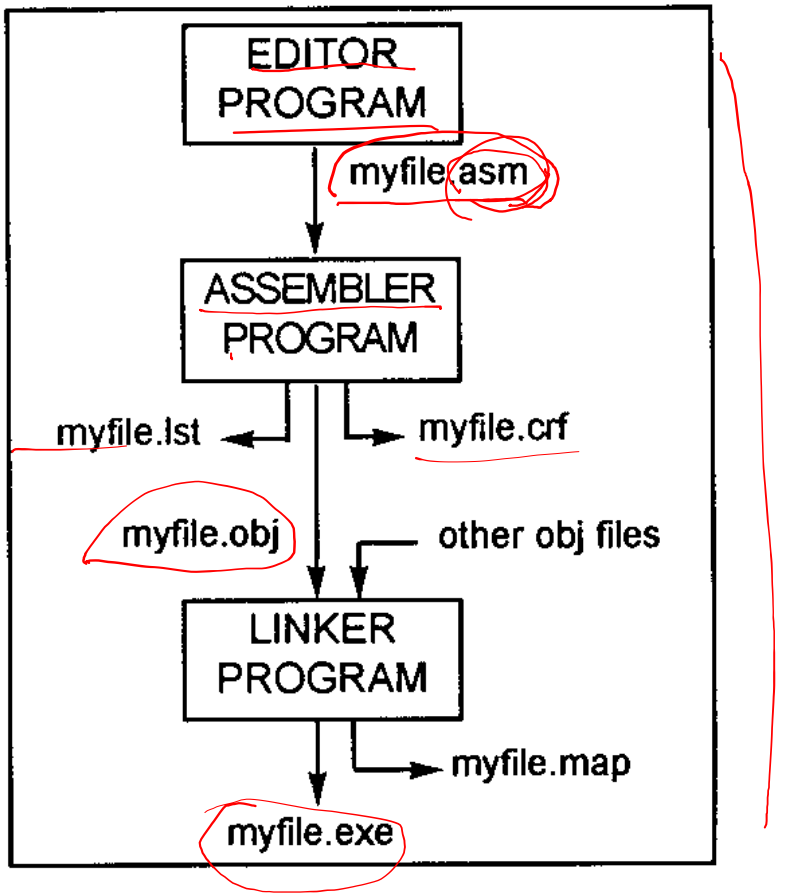
- Program starts from the entrance
  - Ends whenever calls 21H interruption with AH = 4CH
- Procedure caller and callee
  - **CALL** *procedure*
  - **RET**

```
DaSeg1 segment
    str1 db 'Hello World! $'
DaSeg1 ends

StSeg segment
    dw   128  dup(0)

CoSeg segment
start   proc far
    assume cs:CoSeg, ss:StSeg

        mov ax, DaSeg1      ; set segment registers:
        mov ds, ax
        mov es, ax

        call subr       ;call subroutine

        mov ah, 1           ; wait for any key....
        int 21h

        mov ah, 4ch         ; exit to operating system.
        int 21h
start endp

subr proc

        mov dx, offset str1
        mov ah, 9
        int 21h             ; output string at ds:dx

        ret
subr endp

CoSeg ends

        end start   ; set entry point and stop the assembler.
```

Build up Your Program

C>MASM A:MYFILE.ASM <enter>

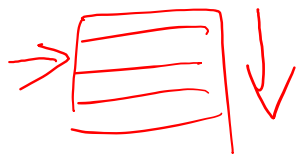
C>LINK A:MYFILE.OBJ <enter>



Control Transfer

- C statement vs assembly instruction

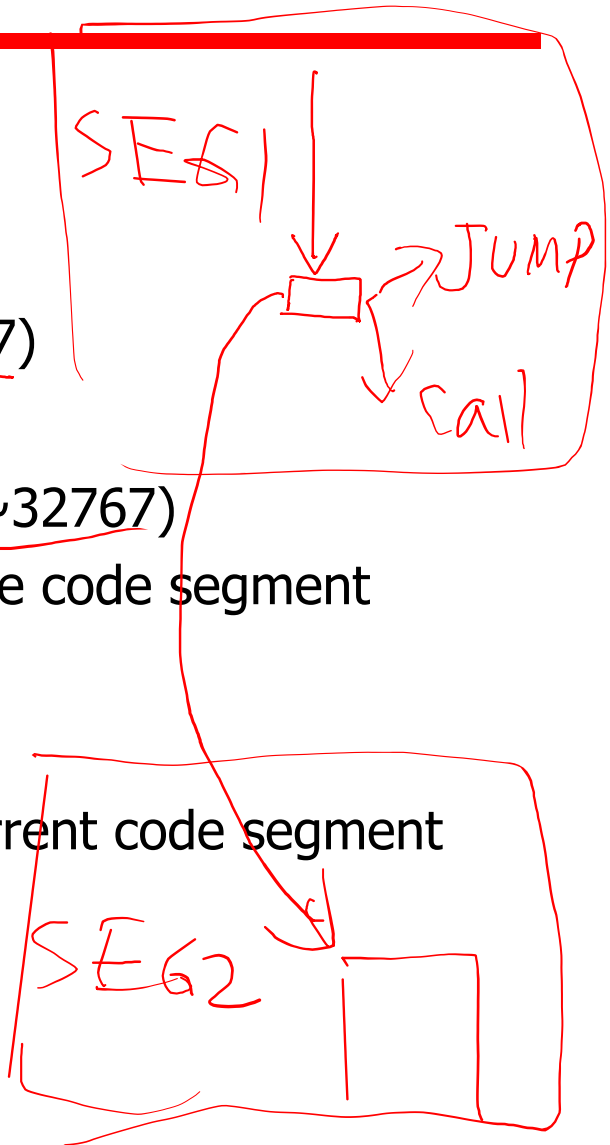
for switch
while if/else goto



← unconditional jump
← conditional jump
call

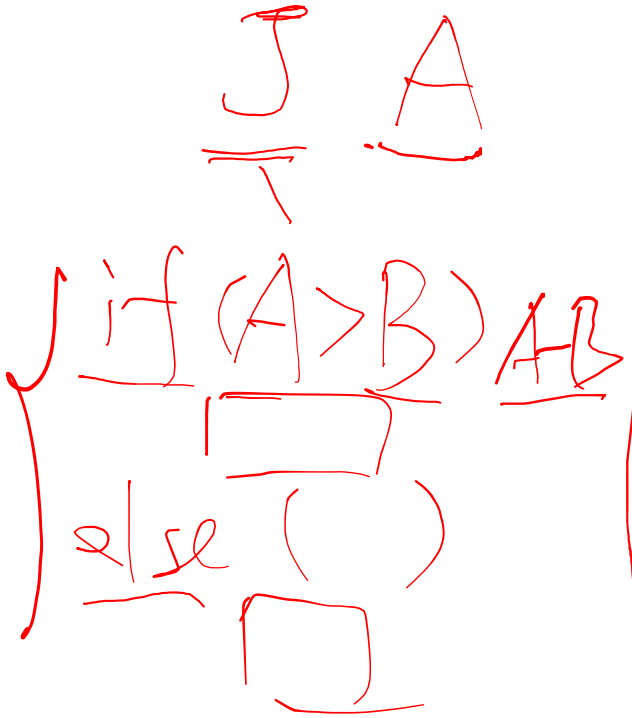
Control Transfer Instructions

- Range
  - **SHORT**, *intrasegment*
    - IP changed: one-byte range (-128~127)
  - **Near**, *intrasegment*
    - IP changed: two-bytes range (-32768~32767)
    - If control is transferred within the same code segment
  - **FAR**, *intersegment*
    - CS and IP all changed
    - If control is transferred outside the current code segment
- Jumps
- CALL statement



Conditional Jumps

- Jump according to the value of the flag register
- Short jumps



Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

Unconditional Jumps

- JMP [SHORT|NEAR|FAR PTR] *label*
- Near by default

Subroutines & CALL Statement

- Range
  - **NEAR**: procedure is defined within the same code segment with the caller
  - **FAR**: procedure is defined outside the current code segment of the caller
- **PROC** & **ENDP** are used to define a subroutine
- **CALL** is used to call a subroutine
  - **RET** is put at the end of a subroutine
  - *Difference between a far and a near call?*

Calling a NEAR proc

- ✓The CALL instruction and the subroutine it calls are in the same segment.
- ✓Save the current value of the IP on the stack.
- ✓load the subroutine's offset into IP (nextinst + offset)

Calling Program	Subroutine	Stack

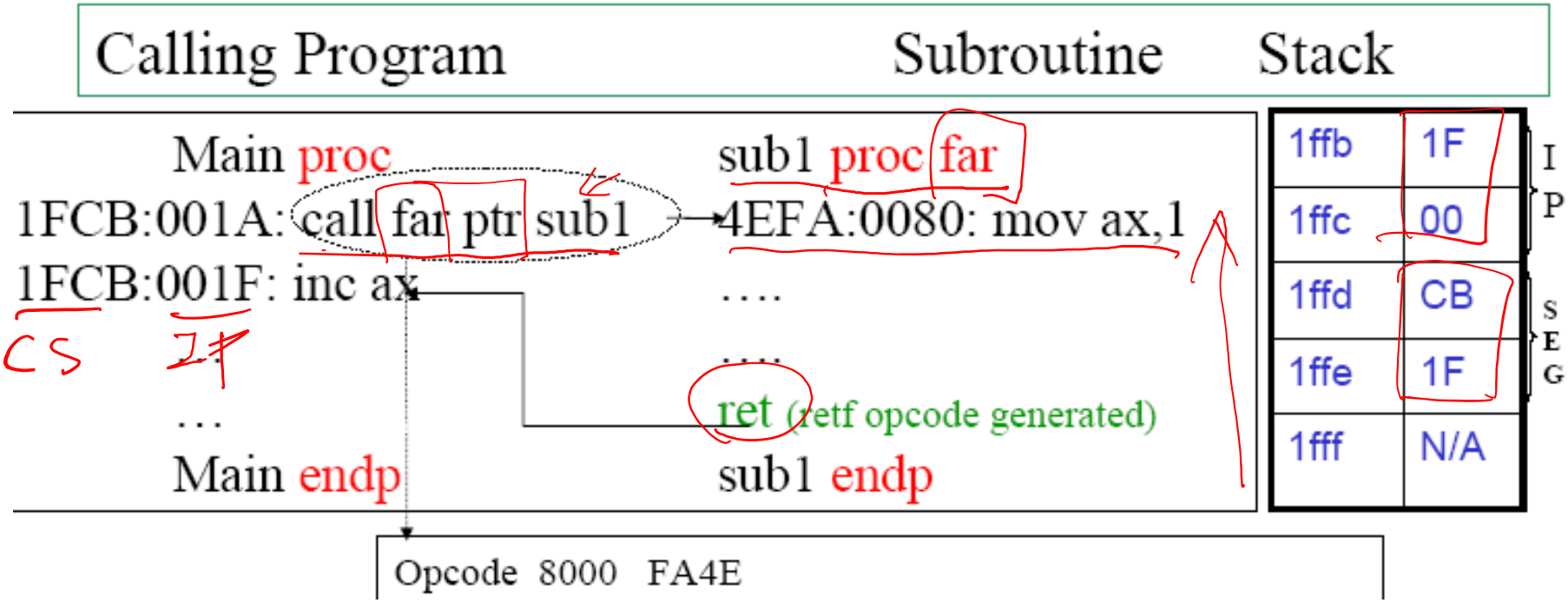
```
Main proc
001A: call sub1
001D: inc ax
.
Main endp
```

```
sub1 proc
0080: mov ax,1
...
ret
sub1 endp
```

1ffd	1D
1ffe	00
1fff	(not used)

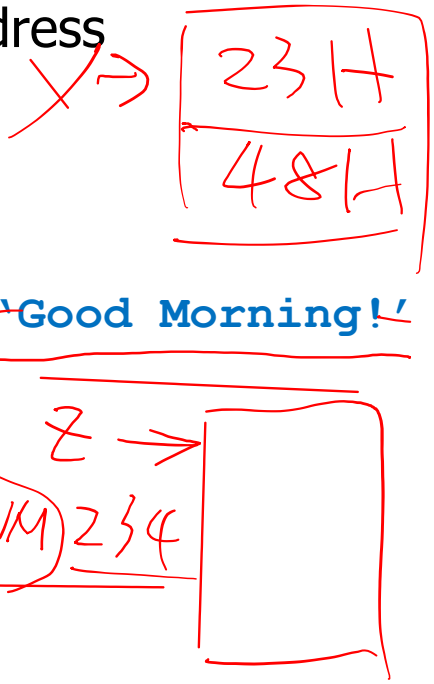
Calling a FAR proc

- ✓The CALL instruction and the subroutine it calls are in the "Different" segments.
- ✓Save the current value of the CS and IP on the stack.
- ✓Then load the subroutine's CS and offset into IP.



Data Types & Definition

- CPU can process either 8-bit or 16 bit ops
  - What if your data is bigger?
- Directives
  - **ORG:** indicates the beginning of the offset address
    - E.g., `ORG 10H`
  - Define variables:
    - **DB:** allocate byte-size chunks
      - E.g., `x DB 12` | `y DB 23H,48H` | `Z DB 'Good Morning!'` | `str DB "I'm good!"`
    - **DW, DD, DQ**
  - **EQU:** define a constant
    - E.g., `NUM EQU 234`
  - **DUP:** duplicate a given number of characters
    - E.g., `x DB 6 DUP(23H)` | `y DW 3 DUP(0FF10H)`



More about Variables

- For variables, they may have names
  - E.g., `luckyNum DB 27H, time DW 0FFFFH`
- Variable names have three attributes:
  - **Segment value**
  - **Offset address**

  } Logical address
  - **Type:** how a variable can be accessed (e.g., DB is byte-wise, DW is word-wise)
- Get the segment value of a variable
  - Use **SEG** directive (E.g., `MOV AX, SEG luchyNum`)
- Get the offset address of a variable
  - Use **OFFSET** directive, or **LEA** instruction
  - E.g., `MOV AX, OFFSET time`, or `LEA AX, time`

Load Effective Address

More about Labels

- Label definition:
  - Implicitly:
    - E.g., `AGAIN: ADD AX, 03423H`
  - Use **LABEL** directive:
    - E.g.,
```
AGAIN LABEL FAR
      ADD AX, 03423H
```
- Labels have three attributes:
  - **Segment value:**
  - **Offset address:**

  } Logical address

  - **Type:** range for jumps, NEAR, FAR

More about the PTR Directive

- Temporarily change the type (range) attribute of a variable (label)
  - To guarantee that both operands in an instruction match
  - To guarantee that the jump can reach a label
- E.g.,

```
DATA1   DB   10H,20H,30H   ;
DATA2   DW   4023H,0A845H
.....
MOV   BX, WORD PTR DATA1 ; 2010H -> BX
MOV   AL, BYTE PTR DATA2 ; 23H -> AL
MOV   WORD PTR [BX], 10H ; [BX],[BX+1]←0010H
```

- E.g.,

```
JMP FAR PTR aLabel
```

More about the PTR Directive

- E.g.,

```
DATA1   DB   10H,20H,30H   ;
DATA2   DW   4023H,0A845H
.....
MOV   BX, WORD PTR DATA1 ; 2010H -> BX
MOV   AL, BYTE PTR DATA2 ; 23H -> AL
MOV   WORD PTR [BX], 10H ; [BX],[BX+1]←0010H
```

MOV [BX], 10H

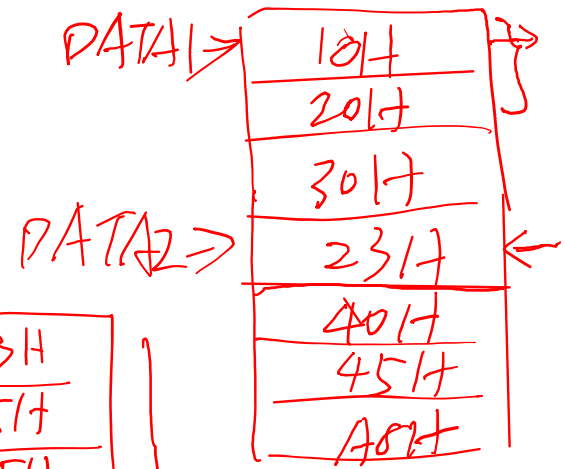
L-W

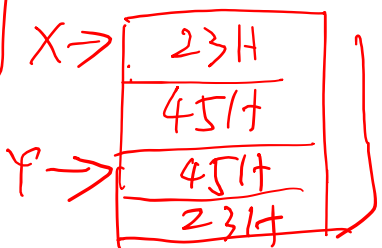
- E.g.,

```
JMP FAR PTR aLabel
```

X DB 23H, 45H

Y DW 2345H





High

.COM Executable

- One segment in total
  - Put data and code all together
  - Less than 64KB

```
TITLE   PROG2-4 COM PROGRAM TO ADD TWO WORDS
PAGE   60,132
CODSG          SEGMENT
               ORG    100H
               ASSUME   CS:CODSG,DS:CODSG,ES:CODSG
;——THIS IS THE CODE AREA
PROGCODE       PROC   NEAR
               MOV    AX,DATA1        ;move the first word into AX
               MOV    SUM,AX          ;move the sum
               MOV    AH,4CH          ;return to DOS
               INT    21H
PROGCODE       ENDP
;——THIS IS THE DATA AREA
DATA1          DW     2390
DATA2          DW     3456
SUM            DW     ?
;-------------------------------
CODSG          ENDS
               END    PROGCODE
```

```
TITLE          PROG2-5 COM PROGRAM TO ADD TWO WORDS
PAGE           60,132
CODSG          SEGMENT
               ASSUME  CS:CODSG,DS:CODSG,ES:CODSG
               ORG    100H
START:         JMP    PROGCODE        ;go around the data area
;——THIS IS THE DATA AREA
DATA1          DW     2390
DATA2          DW     3456
SUM            DW     ?
;——THIS IS THE CODE AREA
PROGCODE:      MOV    AX,DATA1        ;move the first word into AX
               ADD    AX,DATA1        ;add the second word
               MOV    SUM,AX          ;move the sum
               MOV    AH,4CH
               INT    21H
;-------------------------------
CODSB          ENDS
               END    START
```