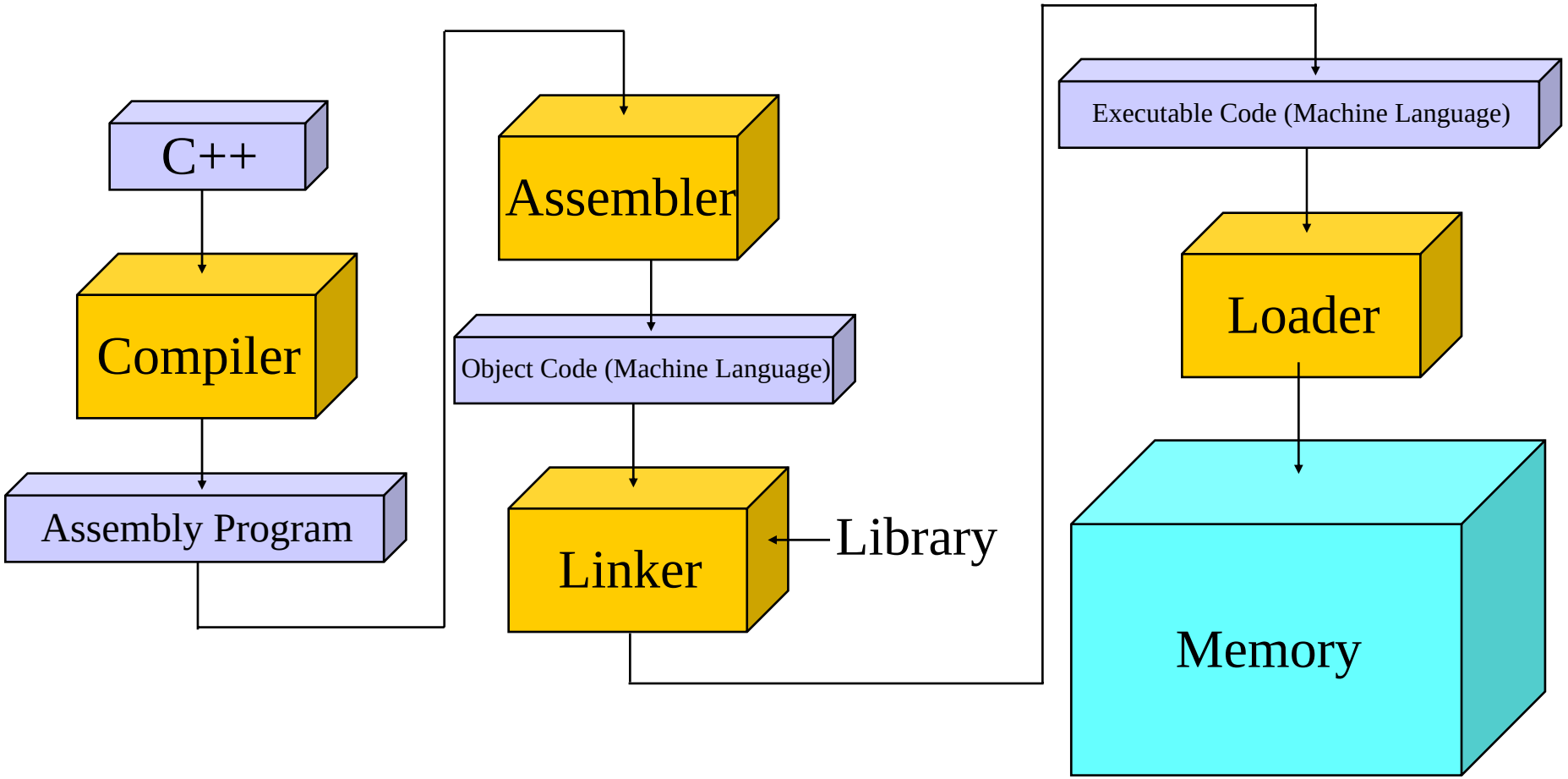
Running a Program

- Running a program involves several steps:

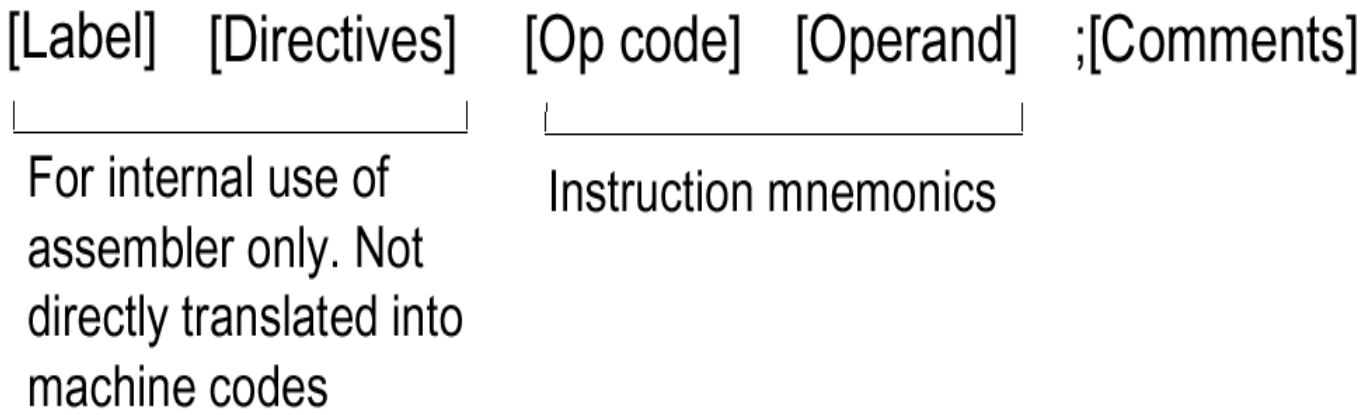


Step	Input	Program	Output
Edit the program	Keyboard	Editor	myfile.asm
Assemble the program	myfile.asm	MASM or TASM	myfile.obj
Link the program	myfile.obj	LINK or TLINK	myfile.exe

Programming with the assembler

An **assembler** serves to convert **instruction mnemonics** (e.g. MOV AX,BX) into machine codes.

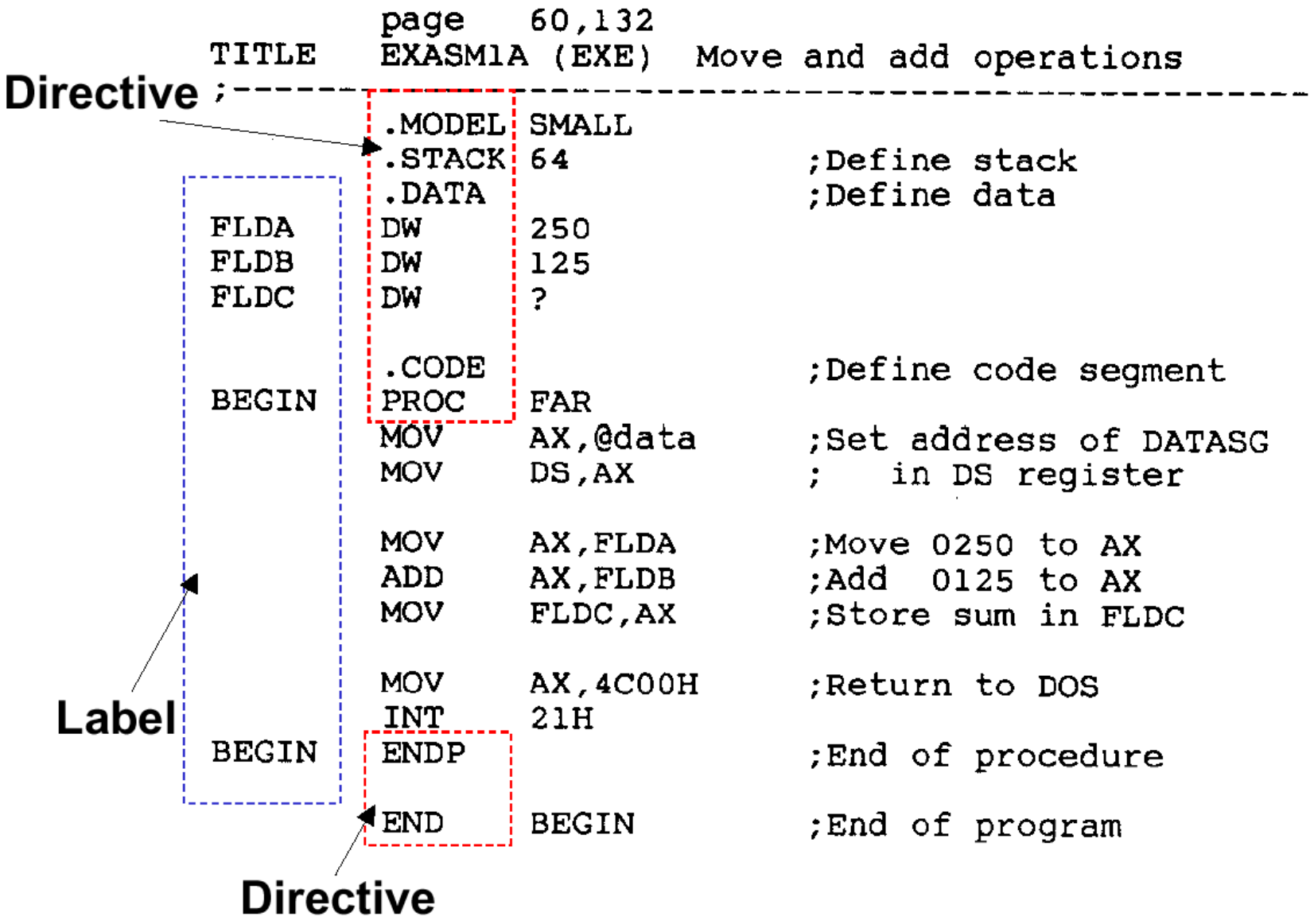
The assembler also allows the use of **labels** and **assembler directives** that make the assembly program easy to be organized understood.



Assembler

- Reads and Uses Directives
  - Directives give instructions to the assembler but do not produce machine language instructions
    - .data, . . .
- Produce Machine Language
- Creates Object File

Example program



Language Components of MASM

- Reserved words
- Identifiers
- Predefined symbols
- Constants
- Expressions
- Operators
- Data types
- Registers
- Statements

Reserved Words

- instructions
  - operations the processor can execute
- directives
  - give commands to the assembler
- operators
  - used in expressions such as product+2
- predefined symbols
  - return info to your program such as @data

reserved words are not case sensitive, except for the predefined symbols

Identifiers

- a name that you invent and attach to a definition
- can represent variables, constants, procedure names, segment names, and user-defined data types such as structures
  - cannot exceed 247 characters
  - the first character can be alphabetic (A-Z), or @, _ $?
  - the other characters can by any of the above characters or a decimal digit (0 - 9)

it's best to avoid starting an identifier with @, because many predefined symbols begin with @

Some Predefined Symbols

- @code
  - returns the name of the code segment
- @data
  - returns the name of the data segment
- @Model
  - returns the selected memory model

3.1.1 Constants and expressions

- **Numeric literal**
  - A combination of digits and other optional parts: a sign, a decimal point, and an exponent
  - 5, 5.5, -5.5, 26.E+05
  - **Integer constants**: end with a radix symbol that identifies the numeric base.
    H, h = hexadecimal
    Q, q, O, o = octal
    D, d = decimal
    B, b = binary
  - 26, 1Ah, 1101b, 36q, 2BH, 42Q, 36D, 47d, **0F6h**
  - When a **hexadecimal constant** begins with a letter, it must contain a leading zero.

Integer Constants

- an integer constant is a series of one or more numerals followed by an optional radix specifier

  mov ax, 25

  mov ax, 0b3h

  - 25 and 0b3h are integer constants..h is a radix specifier
  - the default radix is decimal
    - b is for binary
    - d is for decimal
    - h is hex
    - q for octal
  - hex constants must start with a decimal digit…if necessary, add a leading zero

Symbolic Constant Expressions

- You can create symbolic integer constants using the EQU directive

```
column EQU 80 ; Constant - 80
row    EQU 25 ;  Constant - 25
screen EQU column * row
; Constant - 2000
```

- you cannot change these!

3.1.1 Constants and expressions

- **Character or string constant**
  - 'ABC', 'X'
  - "This is a test"
  - '4096'
  - "This isn't a test"
  - 'Say "hello" to bill.'

Data Allocation Directives

db	define byte
dw	define word (2 bytes)
dd	define double word (4 bytes)
dq	define quadword (8 bytes)
dt	define tenbytes
equ	equate, assign numeric expression to a name

Examples:

```
db 100 dup (?)    define 100 bytes, with no initial values for bytes
db "Hello"        define 5 bytes, ASCII equivalent of "Hello".
maxint   equ      32767
count    equ      10 * 20      ; calculate a value (200)
```

3.4.1 Define byte (DB)

[name] DB initval [,initval]...

- **Allocates storage for one or more 8-bit (byte) values.**
- **Name is optional**
- **Multiple initializers**

```
char1      db    'A'          char2  db    'A'-10
signed1    db    -128         signed2 db   +127
unsigned1 db     255          myval db     ?
List       db    10, 20, 30, 40
```

3.4.1 Define byte (DB)

- **Characters and integer are the same**

```
char    db    'A'
hex     db    41h
Dec     db    65
bin     db    01000001b
Oct     db    101q

list1   db    10, 32, 41h, 00100010b
list2   db    0Ah, 20h, 'A', 22h
```

```
DATA1   DB      25
DATA2   DB      10001001b
DATA3   DB      12h
        ORG  0010h
DATA4   DB    “2591”
        ORG   0018h
DATA5   DB    ?
```

This is how data is initialized in the data segment

```
0000      19
0001      89
0002      12
0010      32 35 39 31
0018      00
```

3.4.1 Define byte (DB)

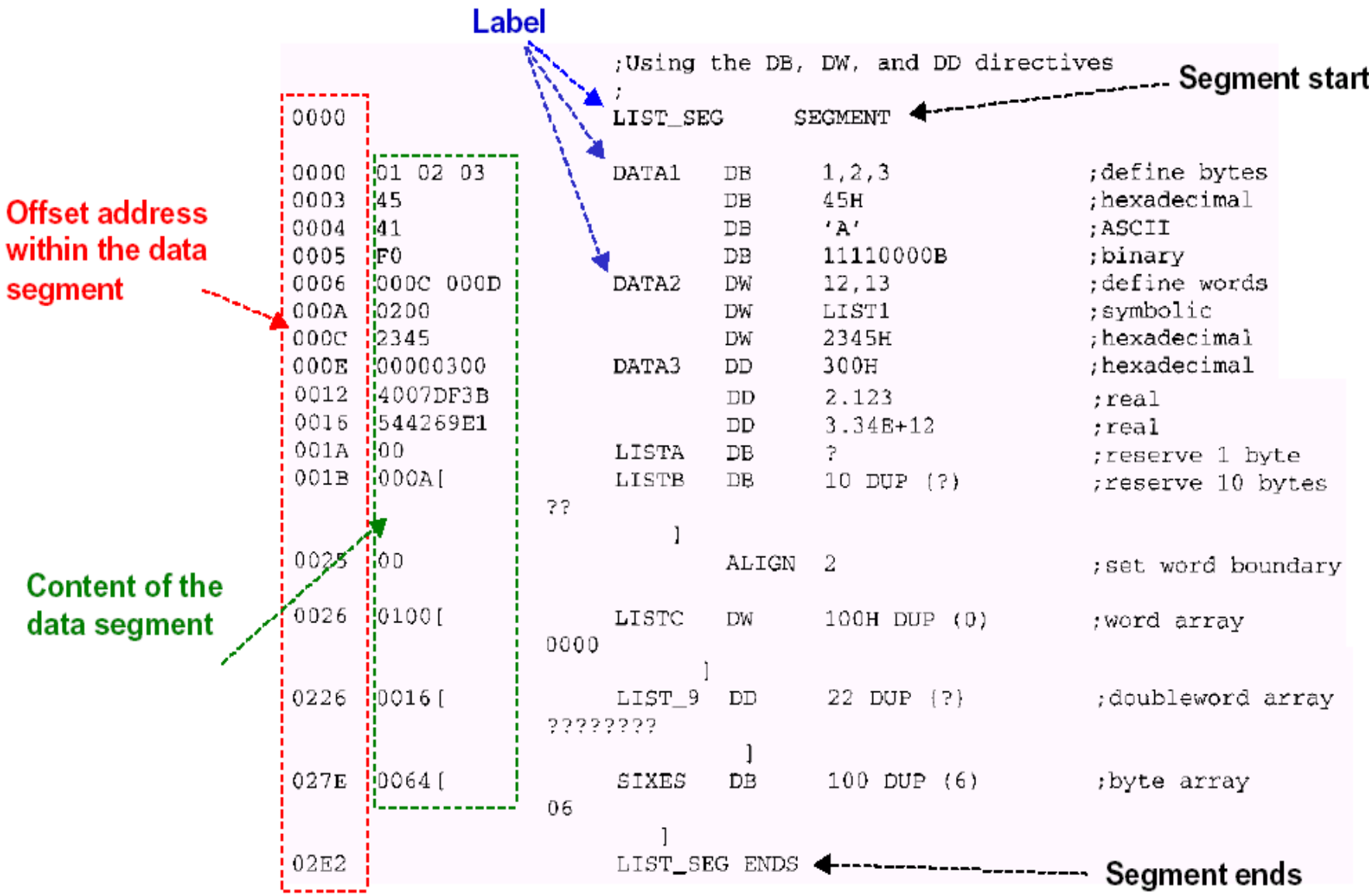
- **DUP operator**
  - With DUP, you can repeat one or more values when allocating storage.
  - Especially useful when allocating space for a string or array

```
db 20 dup (0)          ;20 bytes, all equal to zero
db 20 dup (?)          ;20 bytes, uninitialized
db 4 dup ("ABC")       ;12 bytes: "ABCABCABCABC"
db 4096 dup (0)        ;4096-byte buffer, all zero
```

  - Can also be **nested**

```
aTable db 4 dup ( 3 dup (0), 2 dup ('X') )
aMatrix dw 3 dup (4 dup (0) )
```

Directives: SEGMENT, DB, DW, DUP




```
                        ;Using the DB, DW, and DD directives
                        ;
0000                    LIST_SEG      SEGMENT

0000  01 02 03          DATA1   DB      1,2,3               ;define bytes
0003  45                        DB      45H                 ;hexadecimal
0004  41                        DB      'A'                 ;ASCII
0005  F0                        DB      11110000B           ;binary
0006  000C 000D          DATA2   DW      12,13               ;define words
000A  0200                      DW      LIST1               ;symbolic
000C  2345                      DW      2345H               ;hexadecimal
000E  00000300          DATA3   DD      300H                ;hexadecimal
0012  4007DF3B                  DD      2.123               ;real
0016  544269E1                  DD      3.34E+12            ;real
001A  00                LISTA   DB      ?                   ;reserve 1 byte
001B  000A[             LISTB   DB      10 DUP (?)          ;reserve 10 bytes
??
                        ]
0025  00                        ALIGN   2                   ;set word boundary

0026  0100[             LISTC   DW      100H DUP (0)        ;word array
0000
                        ]
0226  0016[             LIST_9  DD      22 DUP (?)          ;doubleword array
????????
                        ]
027E  0064[             SIXES   DB      100 DUP (6)         ;byte array
06
                        ]
02E2                    LIST_SEG ENDS
```

Notes: The segment address has not been specified yet !

Memory Models

.model *mname* define a memory model for a program. The memory model will affect the size and number of the code, data segments. The memory model also affects the default procedure calls generated by the assembler (*near* calls for tiny,small, compact; *far* calls for medium, large, huge).

Model name can be:

tiny	code, data combined <= 64K
small	1 code, 1 data; code < = 64k, data <= 64k
medium	data <= 64k, code any size, multiple code segs, 1data
compact	code <= 64k, data any size, mulitple data segs, 1 code
large	code, data any size; multiple code, data segs
huge	same as large, but single array can be > 64k.
flat	no segments, all 32-bit addresses for code, data. Protected mode only

Target Processor Directives

.586 allow all nonpriviledged pentium instructions

.486 allow all nonpriviledged 486 instructions

other directives for processor types are similar:

.386

.286

.186

.8086

In the lab, simply use *.586* to access all of the Pentium instructions.

Assembler directives

An assembly program may consist of one or more segments, e.g. code segment, data segment, stack segment. The **SEGMENT** directive is used to define these segments.

```
[label]   [Directives]                    [Op code]  [Operand]  ;[Comments]
DATA      SEGMENT PARA 'DATA'
MSG         DB 'Hello World !$'
DATA      ENDS

STK       SEGMENT PARA STACK 'STACK'
            DW 127 DUP (?)
TOS         DW ?
STK       ENDS
```

Other Directives

- ASSUME
  - tells the assembler which logical segment to use for each physical segment
    - remember that just because I named my data segment DATA_HERE, doesn't tell the assembler that that's my data segment!

```
ASSUME DS:DATA_HERE,
CS:CODE_HERE
```

  - segment registers other than the code segment must still be initialized

```
mov ax, DATA_HERE
mov ds, ax
```

Directives: ASSUME, ORG

```
CODE    SEGMENT PARA 'CODE'
          ASSUME CS:CODE, DS:DATA, SS:STACK
          ORG 100H
START:    MOV AX, DATA
          ...   ...
          ...   ...
CODE      ENDS
          END START
```

ASSUME tells the assembler the purpose of each segment.

"`ASSUME CS:CODE`" says that the segment "`CODE`" is associated with CS.

ORG set the offset address for the first instruction in "CODE", i.e. the instruction "MOV AX, DATA" will be stored at CS:`0100H`.

The option "`PARA`" specifies the memory alignment. It says that the segment will start at the next 16-byte boundary.

Directives: PROC, ENDP

PROC defines a *procedure* or *subroutine*. It appears in pair with **ENDP**.

```
ADDEM  PROC   NEAR          ;start procedure
              ADD BX,CX
              ADD BX,DX
              MOV AX,BX
              RET
ADDEM  ENDP                 ;end procedure
```

NEAR indicates that "ADDEM" resides in the same code segment as the program that "calls" it. The opposite **FAR** means that the procedure may reside in another code segment.

To call this procedure,

```
CALL   ADDEM
```

or

```
CALL   FAR ADDEM
```

Masm Assembler Directives

end *label*	end of program, lable is entry point
proc far|near	begin a procedure; far, near keywords specify if procedure in different code segment (far), or same code segment (near)
endp	end of procedure
page	set a page format for the listing file
title	title of the listing file
.code	mark start of code segment
.data	mark start of data segment
.stack	set size of stack segment

MASM Directives

- .TITLE
  - give the title of the program
- .DOSSEG
  - use the MSDOS segment order
- .MODEL small
  - use a small memory model
- .8086
  - 8086/88 instructions only
- .STACK 0100h
  - start stack segment and specify size
- .DATA
  - start data segment
- .CODE
  - start code segment
- END
  - tells the assembler to STOP reading…any instructions after this will be ignored
  - an optional label/address tells the assembler what to use as the program entry point

3.2 Sample hello program

Directive	Description
end	End of program assembly; **Label**
endp	End of procedure
page	Set a page format for the listing file
proc	Begin procedure
title	Title of the listing file
.code	Mark the beginning of the code segment
.data	Mark the beginning of the data segment
.model	Specify the program's memory model
.stack	Set the size of the stack segment

Simplified segment directives

```
          .MODEL small
          .STACK 64                ;Define stack

          .DATA                    ;Define data
FLDA      DW 250
FLDB      DW ?

          .CODE                    ;Define code
BEGIN     PROC FAR
            MOV AX, @data
            ...
            ...
BEGIN     ENDP                     ;End of procedure

          END                      ;End of program
```

"`.MODEL`" defines the whole memory model.
"`.CODE`", "`.DATA`" and "`.STACK`" are used to start the respective segments.

"`.MODEL`" specifies the *scale* of the whole memory model being defined.
Six different model types: `tiny`, `small`, `medium`, `compact`, `large`, `huge`

```
        .MODEL SMALL
        .STACK 64
        .DATA
DATA1   DB  52h
DATA2   DB 29h
SUM     DB ?
        .CODE
MAIN    PROC FAR
        MOV AX,@DATA
        MOV DS,AX
        MOV AL,DATA1
        MOV BL,DATA2
        ADD AL,BL
        MOV SUM,AL
        MOV AH,4Ch
        INT 21h
MAIN     ENDP
         END MAIN
```

Example program

```
        .MODEL  SMALL
        .CODE
        ORG     100H
BEGIN:  JMP     SHORT MAIN
; ---------------------------------------------
BYTEA   DB  64H                     ;Data items
BYTEB   DB  40H
BYTEC   DB  16H
WORDA   DW  4000H
WORDB   DW  2000H
WORDC   DW  1000H
; ---------------------------------------------
MAIN    PROC    NEAR                ;Main procedure:
        CALL    B10ADD              ;Call ADD routine
        CALL    C10SUB              ;Call SUB routine
        MOV     AH,4CH
        INT     21H                 ;Exit
MAIN    ENDP
;       Examples of ADD bytes:
;       ----------------------
B10ADD  PROC
        MOV     AL,BYTEA
        MOV     BL,BYTEB
        ADD     AL,BL               ;Register-to-register
        ADD     AL,BYTEC            ;Memory-to-register
        ADD     BYTEA,BL            ;Register-to-memory
        ADD     BL,10H              ;Immediate-to-register
        ADD     BYTEA,25H           ;Immediate-to-memory
        RET
B10ADD  ENDP
;       Examples of SUB words:
;       ----------------------
C10SUB  PROC
        MOV     AX,WORDA
        MOV     BX,WORDB
        SUB     AX,BX               ;Register-from-register
        SUB     AX,WORDC            ;Memory-from-register
        SUB     WORDA,BX            ;Register-from-memory
        SUB     BX,1000H            ;Immediate-from-register
        SUB     WORDA,256H          ;Immediate-from-memory
        RET
C10SUB  ENDP

        END     BEGIN
```