Chapter TWO

Assembly Language Programming

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY

The x86 PC
assembly language, design, and interfacing
fifth edition

Prentice Hall

OBJECTIVES

this chapter enables the student to:

- Explain the difference between Assembly language instructions and pseudo-instructions.
- Identify the segments of an Assembly program.
- Assemble, link, and run a simple Assembly program.
- Code control transfer instructions such as conditional jump and unconditional jump and call.
- Data directives for binary, hex, decimal, or ASCII.
- Write an Assembly language program using either the simplified/full segment definition.
- Explore the use of the (MASM and emu8086) assemblers.

2.0: ASSEMBLY LANGUAGE

- An Assembly language program is a series of statements, or lines.
  - Either Assembly language instructions, or statements called *directives*.
    - Directives (pseudo-instructions) give directions to the assembler about how it should translate the Assembly language instructions into machine code.
- Assembly language instructions consist of four fields:
  `[label:] mnemonic [operands][;comment]`
  - Brackets indicate that the field is optional.
    - Do not type in the brackets.

2.1: DIRECTIVES AND A SAMPLE PROGRAM
assembly language instructions

[label:] mnemonic [operands][;comment]

- Examples of directives are **DB**, **END**, and **ENDP**.

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
model definition

- After the first two comments is the MODEL directive.
  - This directive selects the size of the memory model.

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
model definition

- Among the options for the memory model are SMALL, MEDIUM, COMPACT, and LARGE.

```
.MODEL SMALL     ;this directive defines the model as small
.MODEL MEDIUM    ;the data must fit into 64K bytes
                 ;but the code can exceed 64K bytes of memory
.MODEL COMPACT   ;the data can exceed 64K bytes
                 ;but the code cannot exceed 64K bytes
.MODEL LARGE     ;both data and code can exceed 64K
                 ;but no single set of data should exceed 64K
.MODEL HUGE      ;both code and data can exceed 64K
                 ;data items (such as arrays) can exceed 64K
.MODEL TINY      ;used with COM files in which data and code
                 ;must fit into 64K bytes
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
segment definition

- Every line of an Assembly language program must correspond to one an x86 CPU segment register.
  - CS (code segment); DS (data segment).
  - SS (stack segment); ES (extra segment).
- The simplified segment definition format uses three simple directives: ".CODE" ".DATA" ".STACK“
  - Which correspond to the CS, DS, and SS registers.

```
.STACK      ;marks the beginning of the stack segment
.DATA       ;marks the beginning of the data segment
.CODE       ;marks the beginning of the code segment
```

  - The stack segment defines storage for the stack.
  - The data segment defines the data the program will use.
  - The code segment contains Assembly language instructions.

2.1: DIRECTIVES AND A SAMPLE PROGRAM
stack segment

- This directive reserves 64 bytes of memory for the stack:

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
data segment

- The data segment defines three data items:
  - DATA1, DATA2, and SUM.

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The first line of the segment after the .CODE directive is the PROC directive.

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- *A procedure* is a group of instructions designed to accomplish a specific function.
  - A code segment is organized into several small procedures to make the program more structured.
- Every procedure must have a name defined by the PROC directive.
  - Followed by the assembly language instructions, and closed by the ENDP directive.
    - The PROC and ENDP statements must have the same label.
  - The PROC directive may have the option FAR or NEAR.
    - The OS requires the entry point to the user program to be a FAR procedure.

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The program loads **AL** & **BL** with **DATA1** & **DATA2**, ADDs them together, and stores the result in **SUM**.

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The first instructions, "**MOV AX,@DATA**" & "**MOV DS, AX**" initialize DS register

```
;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 OS
              INT    21H            ;
MAIN          ENDP
              END    MAIN           ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The last instructions, "**MOV AH,4CH**" & "**INT 21H**" return control to the operating system.

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- The last two lines end the procedure & program.
  - The label for **ENDP(MAIN)** matches the label for **PROC**.

```
;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 OS
              INT    21H          ;
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.1: DIRECTIVES AND A SAMPLE PROGRAM
code segment definition

- It is handy to keep a sample shell & fill it in with the instructions and data for your program.

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
; USING SIMPLIFIED SEGMENT DEFINITION
              .MODEL SMALL
              .STACK 64
              .DATA
              ;
              ;place data definitions here
              ;
              .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
              ;
              ;place code here
              ;
              MOV    AH,4CH       ;set up to
              INT    21H          ;return to OS
MAIN          ENDP
              END    MAIN         ;this is the program exit point
```

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

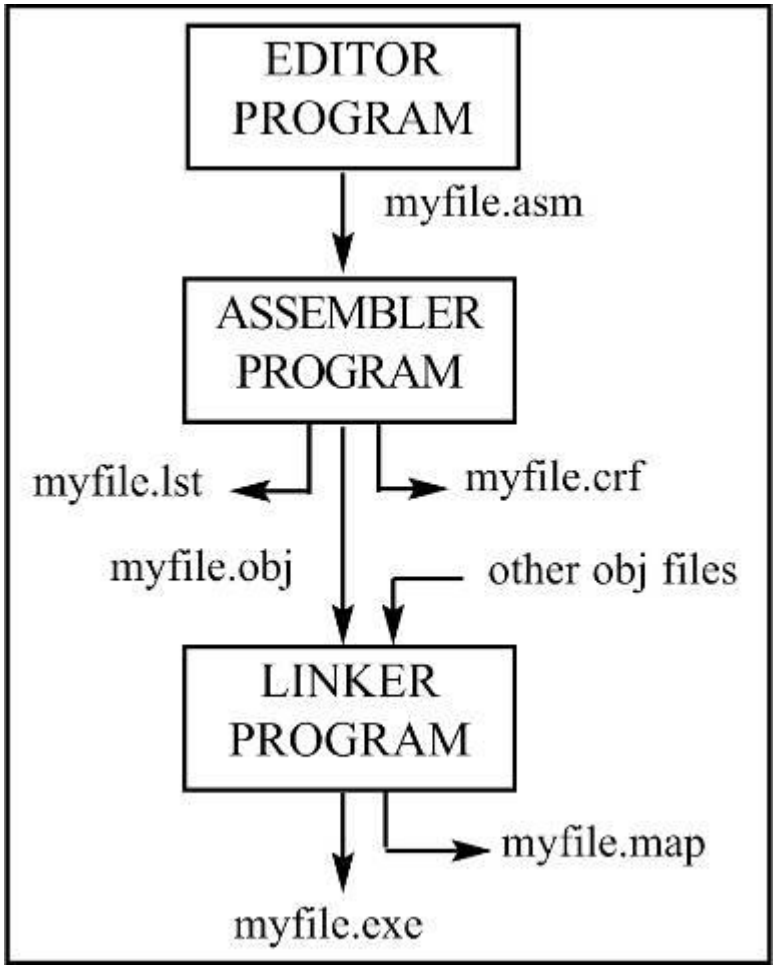
- MASM & LINK are the assembler & linker programs.
  - Many editors or word processors can be used to create and/or edit the program, and produce an ASCII file.
  - The steps to create an executable Assembly language program are as follows:

Step	Input	Program	Output
1. Edit the program	keyboard	editor	myfile.asm
2. Assemble the program	myfile.asm	MASM or TASM	myfile.obj
3. Link the program	myfile.obj	LINK or TLINK	myfile.exe

2.2: ASSEMBLE, LINK, AND RUN A PROGRAM

Before feeding the ".obj" file into LINK, all syntax errors must be corrected.

Fixing these errors will not guarantee the program will work as intended, as the program may contain conceptual errors.



2.4: CONTROL TRANSFER INSTRUCTIONS FAR and NEAR

- In the sequence of instructions, it is often necessary to transfer program control to a different location.
  - If control is transferred to a memory location within the current code segment, it is NEAR.
    - Sometimes called intrasegment. (within segment)
    - the IP is updated and CS remains the same
  - If control is transferred outside the current code segment, it is a FAR jump.
    - Or intersegment. (between segments)
    - both CS and IP have to be updated to the new values

2.4: CONTROL TRANSFER INSTRUCTIONS conditional jumps

- Conditional jumps have mnemonics such as JNZ (jump not zero) and JC (jump if carry).
  - In the conditional jump, control is transferred to a new location if a certain condition is met.
  - The flag register indicates the current condition.
- For example, with "JNZ label", the processor looks at the zero flag to see if it is raised.
  - If not, the CPU starts to fetch and execute instructions from the address of the label.
  - If ZF = 1, it will not jump but will execute the next instruction below the JNZ.

2.4: CONTROL TRANSFER INSTRUCTIONS conditional jumps

Table 2-1: 8086 Conditional Jump Instructions

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

Note: "Above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- All conditional jumps are short jumps.
  - The address of the target must be within -128 to +127 bytes of the IP.
- The conditional jump is a two-byte instruction.
  - One byte is the opcode of the J condition.
  - The second byte is a value between 00 and FF.
    - An offset range of 00 to FF gives 256 possible addresses.
- In a jump backward, the second byte is the 2's complement of the displacement value

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- To calculate the target address, the second byte is added to the IP of the instruction after the jump.

```
1067:0000 B86610          MOV   AX,1066
1067:0003 8ED8    MOV     DS,AX
1067:0005 B90500          MOV   CX,0005
1067:0008 BB0000          MOV   BX,0000
1067:000D 0207    ADD     AL,[BX]
1067:000F 43              INC   BX
1067:0010 49              DEC   CX
1067:0011 75FA    JNZ     000D
1067:0013 A20500          MOV   [0005],AL
1067:0016 B44C    MOV     AH,4C
1067:0018 CD21    INT     21
```

  - "JNZ AGAIN" was assembled as "JNZ 000D", and 000D is the address of the instruction with the label AGAIN.
    - "JNZ 000D" has the opcode 75 and the target address FA.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- This is followed by "MOV SUM,AL", which is located beginning at offset address 0013.

```
1067:0000 B86610          MOV   AX,1066
1067:0003 8ED8    MOV     DS,AX
1067:0005 B90500          MOV   CX,0005
1067:0008 BB0000          MOV   BX,0000
1067:000D 0207    ADD     AL,[BX]
1067:000F 43              INC   BX
1067:0010 49              DEC   CX
1067:0011 75FA    JNZ     000D
1067:0013 A20500          MOV   [0005],AL
1067:0016 B44C    MOV     AH,4C
1067:0018 CD21    INT     21
```

  - The IP value of MOV,0013, is added to FA to calculate the address of label AGAIN, and the carry is dropped.
    - FA is the 2's complement of -6.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- Calculate a forward jump target address by adding the IP of the following instruction to the operand.
  - The displacement value is positive, as shown.

```
0005        8A 47 02 AGAIN:   MOV   AL,[BX]+2
0008        3C 61             CMP   AL,61H
000A        72 06             JB    NEXT
000C        3C 7A             CMP   AL,7AH
000E        77 02             JA    NEXT
0010        24 DF             AND   AL,0DFH
0012        88 04   NEXT:     MOV   [SI],AL
```

  - "**JB NEXT**" has the opcode **72**, the target address **06** and is located at **IP = 000A** and **000B**.
    - The jump is 6 bytes from the next instruction, is IP = 000C.
    - Adding gives us 000CH + 0006H = 0012H, which is the exact address of the **NEXT** label.

2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- For conditional jumps, the address of the target address can never be more than -128 to +127 bytes away from the IP associated with the instruction following the jump.
  - Any attempt is made to violate this rule will generate a "`relative jump out of range`" message.

2.4: CONTROL TRANSFER INSTRUCTIONS unconditional jumps

- An unconditional jump transfers control to the target location label unconditionally, in the following forms:
  - **SHORT JUMP** - in the format "JMP SHORT label".
    - A jump within -128 to +127 bytes of memory relative to the address of the current IP, opcode EB.
  - **NEAR JUMP** - the default, has the format "JMP label".
    - A jump within the current code segment, opcode E9.
    - The target address can be any of the addressing modes of direct, register, register indirect, or memory indirect:
  - **Direct JUMP** - exactly like the short jump.
    - Except that the target address can be anywhere in the segment in the range +32767 to -32768 of the current IP.

2.4: CONTROL TRANSFER INSTRUCTIONS unconditional jumps

- An unconditional jump transfers control to the target location label unconditionally, in the following forms:
  - **Register indirect JUMP** - target address is in a register.
    - In "JMP BX", IP takes the value BX.
  - **Memory indirect JMP** - target address is the contents of two memory locations, pointed at by the register.
    - "JMP [DI]" will replace the IP with the contents of memory locations pointed at by DI and DI+1.
  - **FAR JUMP** - in the format "JMP FAR PTR label". register.
    - A jump out of the current code segment
    - IP and CS are both replaced with new values.

2.4: CONTROL TRANSFER INSTRUCTIONS
CALL statements

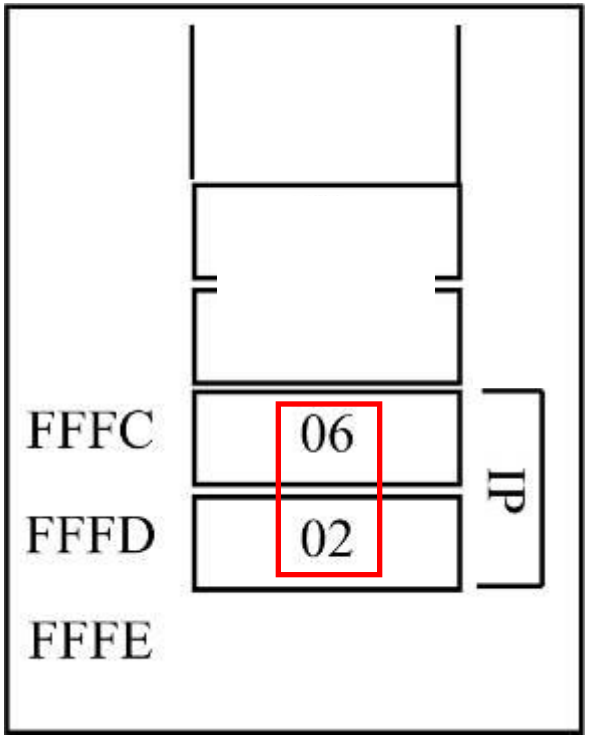
- The CALL instruction is used to call a procedure, to perform tasks that need to be performed frequently.
  - The target address could be in the current segment, in which case it will be a NEAR call or outside the current CS segment, which is a FAR call.
- The microprocessor saves the address of the instruction following the call on the stack.
  - To know where to return, after executing the subroutine.
    - In the NEAR call only the IP is saved on the stack.
    - In a FAR call both CS and IP are saved.

2.4: CONTROL TRANSFER INSTRUCTIONS
CALL statements

- For control to be transferred back to the caller, the last subroutine instruction must be RET (return).
  - For NEAR calls, the IP is restored.
  - For FAR calls, CS & IP are restored.
- Assume SP = FFFEH:

```
12B0:0200   BB1295   MOV BX,9512
12B0:0203   E8FA00   CALL 0300
12B0:0206   B82F14   MOV AX,142F
```

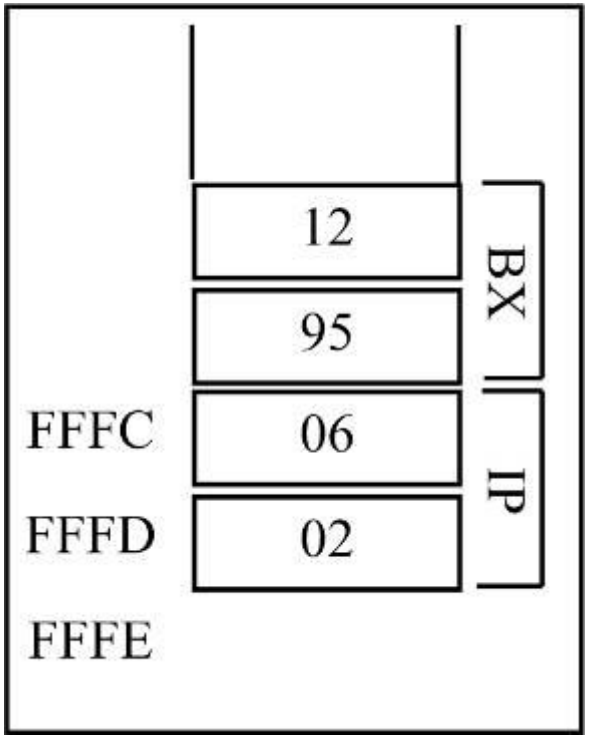
  - Since this is a NEAR call, only IP is saved on the stack.
    - The IP address **0206**, which belongs to the "**MOV AX,142F**" instruction, is saved on the stack.



2.4: CONTROL TRANSFER INSTRUCTIONS
short jumps

- The last instruction of the called subroutine must be a RET instruction that directs the CPU to POP the top 2 bytes of the stack into the IP and resume executing at offset address 0206.
  - The number of PUSH and POP instructions (which alter the SP) **must match**.
    - For every **PUSH** there must be a **POP**.

```
12B0:0300   53   PUSH BX
12B0:0301   ...  ...... ..
........    ...  ...... ..
12B0:0309   5B   POP BX
12B0:030A   C3   RET
```



2.5: DATA TYPES AND DATA DEFINITION

- DUP will duplicate a given number of characters.
- DW is used to allocate memory 2 bytes (one word)
- EQU associates a constant value with a data label.
  - Defines a constant without occupying a memory location.
- The DD directive is used to allocate memory locations that are 4 bytes (two words) in size.
  - Data is converted to hex & placed in memory locations
    - Low byte to low address and high byte to high address.
- DQ is used to allocate memory 8 bytes (four words) in size, to represent any variable up to 64 bits wide

2.5: DATA TYPES AND DATA DEFINITION

- DQ is used to allocate memory 8 bytes (four words) in size, to represent any variable up to 64 bits wide
- DT is used for memory allocation of packed BCD numbers.
  - This directive allocates 10 bytes.
    - A maximum of 18 digits can be entered.
  - The "H" after the data is not needed.

2.6: FULL SEGMENT DEFINITION
→ segment definition

```
;FULL SEGMENT DEFINITION
      ;--- stack segment ---
name1 SEGMENT
      DB    64 DUP (?)
name1 ENDS
      ;--- data segment ---
name2 SEGMENT
;place data definitions here
name2 ENDS
      ;--- code segment ---
name3 SEGMENT
MAIN  PROC  FAR
      ASSUME ...
      MOV   AX,name2
      MOV   DS,AX
      ...
MAIN  ENDP
name3 ENDS
      END   MAIN
```

```
;SIMPLIFIED FORMAT
.MODEL  SMALL
.STACK    64
;
;
;--------------------
. DATA
;place data definitions here
;
;--------------------
.CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      ...
      ...
MAIN  ENDP
      END   MAIN
```

Figure 2-8

ENDS ; TWO

The x86 PC

assembly language, design, and interfacing

fifth edition

Prentice Hall

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY