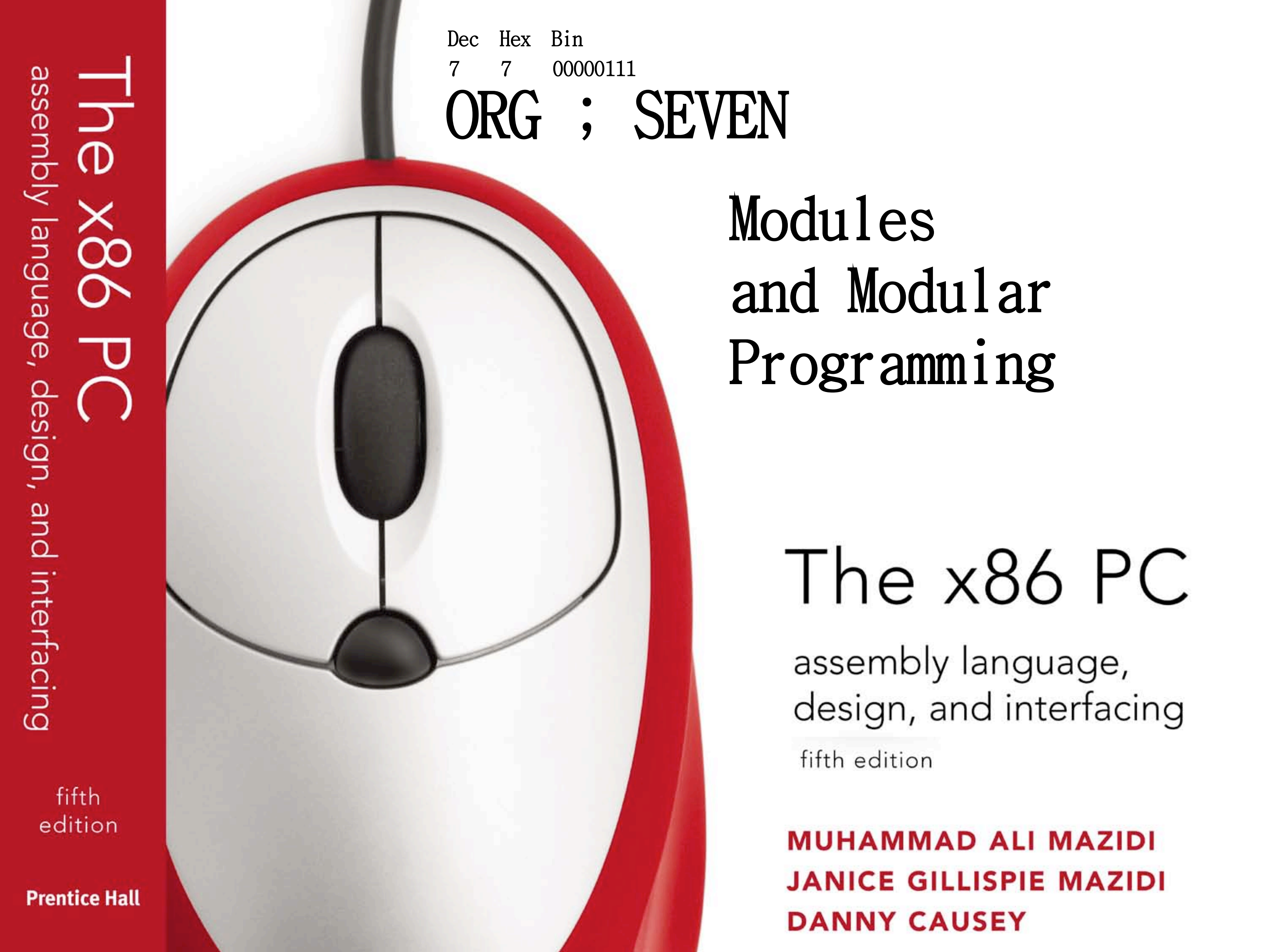
Dec Hex Bin
7 7 00000111

ORG ; SEVEN

Modules and Modular Programming



OBJECTIVES
this chapter enables the student to:

- Discuss the advantages of modular programming.
- Break large programs into modules, code modules and calling programs.
- Declare names that are defined externally via the EXTRN directive.
- Link subprograms together into one executable program.
- Code segment directives to link data, code, or stack segments from different modules into one segment.
- Code programs using the full segment definitions.

OBJECTIVES (*cont*)

this chapter enables the student to:

- List the various methods of passing parameters to modules and discuss the advantages and disadvantages of each.
- Code programs passing the parameters via registers, memory, or stack.

7.1: WRITING AND LINKING MODULES
why modules?

- Modules make projects more manageable, and have other advantages, such as:
  - 1. Each module can be written, debugged, and tested individually.
  - 2. Failure of one module does not stop the entire project.
  - 3. The task of locating and isolating any problem is easier and less time consuming.
  - 4. One can use the modules to link with high-level languages such as C/C++, C#, or Visual Basic.
  - 5. Parallel development shortens considerably the time required to complete a project.

7.1: WRITING AND LINKING MODULES
writing modules

- An efficient way to develop software is to treat each subroutine as a separate program (or module) with a separate filename.
  - Each can be assembled and tested, and brought together (linked) to make a single program.
- To link the modules, certain Assembly language directives must be used.
  - Most widely used are EXTRN (external) and PUBLIC.

EXTRN directive

- EXTRN is used to notify the assembler and linker that certain names and variables not defined in the present module are defined externally elsewhere.
  - In absence of EXTRN, the assembler would show an error, since it cannot find where the names are defined.

```
EXTRN name1:type                    ;each name in a separate EXTRN
EXTRN name2:type                    ;or
EXTRN name1:type,name2:type         ;many listed in the same EXTRN
```

  - External procedure names can be NEAR, FAR, or PROC.
  - Data name types, with the number of bytes indicated in parentheses:
    - BYTE (1); WORD (2); DWORD (4).
    - FWORD (6); QWORD (8); TBYTE (10).

7.1: WRITING AND LINKING MODULES
PUBLIC directive

- Names or parameters defined as EXTRN (defined outside the present module), must be defined as PUBLIC in the module where they are defined.
  - Defining a name as PUBLIC allows the assembler and linker to match it with its EXTRN counterpart(s).

```
PUBLIC name1      ;each name can be in a separate directive
PUBLIC name2

PUBLIC name1, name2 ;or many can be listed
                    ;in the same PUBLIC
```

- Example 7-1, on page 197, demonstrates that for every EXTRN definition there is a PUBLIC directive defined in another module.

7.1: WRITING AND LINKING MODULES
END directive in modules

- Note the program entry/exit points of the in Ex. 7-1.
  - The entry point is MAIN & the exit point is "END MAIN".

```
MAIN        PROC   FAR
            . . .
            CALL    SUBPROG1
            CALL    SUBPROG2
            . . .
            MOV     AH,4CH
            INT     21H
MAIN        ENDP
            END     MAIN
```

Modules called by the main module have the END directive with no label or name after it.

See the entire program listing on page 197 of your textbook.

7.1: WRITING AND LINKING MODULES
linking modules into one executable unit

- Assume modules assembled & saved separately: EXAMPLE1.OBJ, PROC1.OBJ, and PROC2.OBJ.
  - To link in MASM to generate a single executable file:

```
C>LINK EXAMPLE1.OBJ + PROC1.OBJ + PROC2.OBJ
```

linking modules into one executable unit

- Program 7-1 shows how the EXTRN and PUBLIC directives can also be applied to data variables.

```
TITLE PROG7-1MM DEMONSTRATES MODULAR PROGRAMMING
PAGE 60,132
      EXTRN  SUBPROG1:FAR
      EXTRN  SUBPROG2:FAR
      PUBLIC VALUE1, VALUE2, SUM, PRODU
            .MODEL SMALL
            .STACK 64
            .DATA
VALUE1      DW  2050
VALUE2      DW  500
SUM         DW  2 DUP (?)
PRODUCT     DW  2 DUP (?)
;-----------------------------
            .CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      CALL  SUBPROG1
```

The main module contains a **data** segment and a **stack** segment, but the subroutine modules do *not*.

Each module *can* have its own data and stack segment.

See the entire program listing on page 198 of your textbook.

linking modules into one executable unit

- Each subroutine was declared with **EXTRN** indicating they would be defined in another file.

```
TITLE PROG7-1MM DEMONSTRATES MODULAR PROGRAMMING
PAGE 60,132
      EXTRN SUBPROG1:FAR
      EXTRN SUBPROG2:FAR
      PUBLIC VALUE1, VALUE2, SUM, PRODU
            .MODEL SMALL
            .STACK 64
            .DATA
VALUE1      DW 2050
VALUE2      DW 500
SUM         DW 2 DUP (?)
PRODUCT     DW 2 DUP (?)
;------------------------------
            .CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      CALL  SUBPROG1
```

External subroutines were defined as **FAR** in this case.

In the files where each is defined, it is declared as PUBLIC, so other programs can call it.

See the entire program listing on page 198 of your textbook.

7.1: WRITING AND LINKING MODULES
linking modules into one executable unit

- Each subroutine was declared with **EXTRN** indicating they would be defined in another file.

```
TITLE PROG7-1MM DEMONSTRATES MODULAR PROGRAMMING
PAGE 60,132
      EXTRN SUBPROG1:FAR
      EXTRN SUBPROG2:FAR
      PUBLIC VALUE1, VALUE2, SUM, PRODUCT
            .MODEL SMALL
            .STACK 64
            .DATA
VALUE1      DW 2050
VALUE2      DW 500
SUM         DW 2 DUP (?)
PRODUCT     DW 2 DUP (?)
;-----------------------------
            .CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      CALL  SUBPROG1
```

In the main module, the names **VALUE1**, **VALUE2**, **SUM**, and **PRODUCT** were defined as **PUBLIC**, so that other programs could access these data items.

See the entire program listing on page 198 of your textbook.

- In the subprograms, these data items were declared as **EXTRN** and **PUBLIC**.

```
TITLE         PROG7-1M3  PROGRAM
PAGE          60,132
              EXTRN  VALUE1:WORD
              EXTRN  VALUE2:WORD
              EXTRN  PRODUCT:WORD
              PUBLIC
              .MODEL
              .CODE
SUBPROG2      PROC
```

VALUE1, **VALUE2**, **SUM**, and **PRODUCT** were declared as **EXTRN**.

```
TITLE  PROG7-1M2  PROGRAM TO
PAGE   60,132
       EXTRN  VALUE1:WORD
       EXTRN  VALUE2:WORD
       EXTRN  SUM:WORD
       PUBLIC  SUBPROG1
              .MODEL  SMALL
              .CODE
SUBPROG1      PROC    FAR
              SUB     BX,BX
```

See the program module listings on page 199 of your textbook.

7.1: WRITING AND LINKING MODULES analysis of Program 7-1

- The three programs would be linked together as:

```
C>LINK PROG7-1MM.OBJ + PROG7-1M2 + PROG7-1M3
```

  - The linker program resolves external references by matching PUBLIC and EXTRN names.
    - It searches files specified by LINK for external subroutines.

7.1: WRITING AND LINKING MODULES analysis of Program 7-1

- Example 7-2, page 200, shows the shell of modular programs using simplified segment definition.
  - In simplified segment definition, procedures will default to NEAR for small or compact models and to FAR for medium, large, or huge models.
- In the main module of 7-2, MAIN has a colon after it, and is used for the first executable instruction.
  - This is the entry point of the program, and the exit is indicated by the same label, in the END directive.
    - No program can have more than one entry, and one exit point.
- Program 7-2, page 201 - 202, is the same as 7-1, rewritten for the full segment definition.

7.1: WRITING AND LINKING MODULES
SEGMENT directive

- The *complete* segment definition, used widely in modular programming is:

```
name  SEGMENT alignment combine type class name
```

  - The `alignment` field indicates whether a segment should start on a byte, word, paragraph, or page boundary.
    - Default alignment is PARA, to start on a paragraph boundary.
  - The `combine` type field indicates to the linker whether segments of the same type should be linked together.
    - Typical options for combine type are STACK or PUBLIC.
  - The `class name` field has four options:

    'CODE'; 'STACK'; 'DATA'; 'EXTRA'.

7.1: WRITING AND LINKING MODULES
complete SEGMENT directive definition

- The following stack segment definition in the main module will eliminate the "***Warning: no stack segment.***" message generated by the linker:

```
name   SEGMENT      PARA STACK 'STACK'
```

7.1: WRITING AND LINKING MODULES
complete data & code segment definitions

- The following data segment definition can be used if no other module has defined any data segment:

```
name   SEGMENT      PARA 'DATA'
```

  - If any other module has defined a data segment then PUBLIC should be placed between PARA and 'DATA'.

- Code and data segment definitions to combine segments from different modules:

```
name   SEGMENT      PARA PUBLIC 'CODE'
name   SEGMENT      PARA PUBLIC 'DATA'
```

  - Example 7-4, page 204, rewrites Example 7-2, on page 200, to define segments using the complete segment definition.

7.2: SOME VERY USEFUL MODULES
binary (hex) to ASCII(decimal) conversion

- The result of arithmetic operations is in binary.
  - To display the result in decimal, the number is first converted to decimal, then each digit is tagged with 30H to put it in ASCII form to displayed or print.
- The following example converts 34DH to decimal.

$$
\begin{aligned}
34DH &= (3 \times 16^2) + (4 \times 16^1) + (D = 13 \times 16^0) \\
&= (3 \times 256) + (4 \times 16) + (13 \times 1) \\
&= 768 + 64 + 13 \\
&= 845
\end{aligned}
$$

7.2: SOME VERY USEFUL MODULES binary (hex) to ASCII(decimal) conversion

- The result of arithmetic operations is in binary.
  - To display the result in decimal, the number is first converted to decimal, then each digit is tagged with 30H to put it in ASCII form to displayed or print.
- Another method divides a hex number repeatedly by 10 (0AH), storing each remainder, until the quotient is less than **10**.

```
34DH / A = 84 remainder 5
84H  / A = 8  remainder 4
8 (< A, so the process stops)
```

  - Taking the remainders in reverse order gives **845** decimal.

7.2: SOME VERY USEFUL MODULES
binary(hex)to ASCII(decimal) conversion

- Program 7-3, page 206, shows the conversion process for a word-sized (16-bit) number using the method of repeated division demonstrated above.
  - As each decimal digit (the remainder) is placed in DL, it is tagged with 30H to convert it to ASCII.

```
MOV    DS,AX
MOV    BX,10          ;BX=10 THE DIVISOR
MOV    SI,OFFSET ASCNUM  ;SI = BEGINNING OF ASCII ST
ADD    SI,5           ;ADD LENGTH OF STRING
DEC    SI             ;SI POINTS TO LAST ASCII DIGIT
MOV    AX,BINNUM      ;LOAD BINARY (HEX) NUMBER
SUB    DX,DX          ;DX MUST BE 0 IN WORD DIVISION
DIV    BX             ;DIVIDE HEX NUMBER BY 10 (BX=10)
OR     DL,30H         ;TAG '3' TO MAKE IT ASCII
MOV    [SI],DL              ;MOVE THE ASCII DIGIT
DEC    SI             ;DECREMENT POINTER
CMP    AX,0           ;CONTINUE LOOPING WHILE AX >0
```

It is then placed in a memory area called ASCNUM.

7.2: SOME VERY USEFUL MODULES
ASCII(decimal) to binary(hex) conversion

- An mathematical example of converting the decimal number 482 to hex:

```
482 / 16² = 482 / 256 = 1
482 - (1 × 256) = 226     226 / 16¹ = 226 / 16 = 14 = E
226 - (14 × 16) = 2
482 decimal = 1E2  hexadecimal
```

- Since a computer works in binary arithmetic, it would use a different method:
  - First the 30H would be masked off each ASCII digit.
  - Then each digit is multiplied by a weight (a power of 10) such as 1, 10, 100, or 1000, then added together to get the final hex (binary) result.

ASCII(decimal) to binary(hex) conversion

- Program 7-4, on page 207 of your book, converts an ASCII number to binary.
  - It assumes the maximum size of the decimal number to be 65535, making the maximum hex result is FFFFH.
    - A 16-bit word.
- First, a user types '482' through the keyboard, yielding 343832, the ASCII version of 482.
  - The following steps are the performed:

```
2 × 1     =       =     2
8 × 10    =   80  =    50H
4 × 100   =  400  =   190H
                      1E2 hexadecimal
```

7.2: SOME VERY USEFUL MODULES
ASCII(decimal) to binary(hex) conversion

- Program 7-4, on page 207 of your book, converts an ASCII number to binary.
  - It begins with the least significant digit, masks off the 3, and multiplies it by its weight factor.
    - Register CX holds the weight. (for the least significant digit)
  - For the next digit CX becomes 10 (0AH), for the next it becomes 100 (64H), etc.
    - The program assumes that the least significant ASCII digit is in the highest memory location of the data.
- Programs 7-3 and 7-4, pages 206 & 207, written and tested with sample data, can be changed from programs into modules, and called by any program.

7.2: SOME VERY USEFUL MODULES
binary-to-ASCII module

- Program 7-5, page 208, is the modularized version of Program 7-3, seen on page 206.

```
TITLE       PROG7-5 BINARY TO D
PAGE   60,132
;this module converts a binary
; then makes it displayable (AS
;CALLING PROGRAM SETS
; AX = BINARY VALUE TO BE CONVERTED TO ASCII
; SI = OFFSET ADDRESS WHERE ASCII VALUE IS TO BE
       .MODEL SMALL
       PUBLIC B2ASC_CON
       .CODE
B2ASC_CON PROC FAR
       PUSHF                ;STORE REGS CHANGED BY TH
       PUSH BX
```

The procedure is declared **PUBLIC**, so it can be called by another program.

7.2: SOME VERY USEFUL MODULES
binary-to-ASCII module

- Program 7-6, page 208, is the modularized version of Program 7-4, seen on page 207.

```
TITLE        PROG7-6 ASCII TO B
PAGE   60,132
;this module converts any ASCI
;CALLING PROGRAM SETS
;  SI = OFFSET OF ASCII STRING
;  BX = STRING LENGTH - 1 (USE
;THIS MODULE SETS
;  AX = BINARY NUMBER
            .MODEL SMALL
            EXTRN TEN:WORD
            PUBLIC ASC2B_CON
            .CODE
ASC2B_CON   PROC FAR
            PUSH                       STORE RE
```

All values are declared **EXTERNAL,** since data will be provided by the calling program.

TEN is defined in the calling program.

This module must return to the caller and not OS.

7.2: SOME VERY USEFUL MODULES
calling module

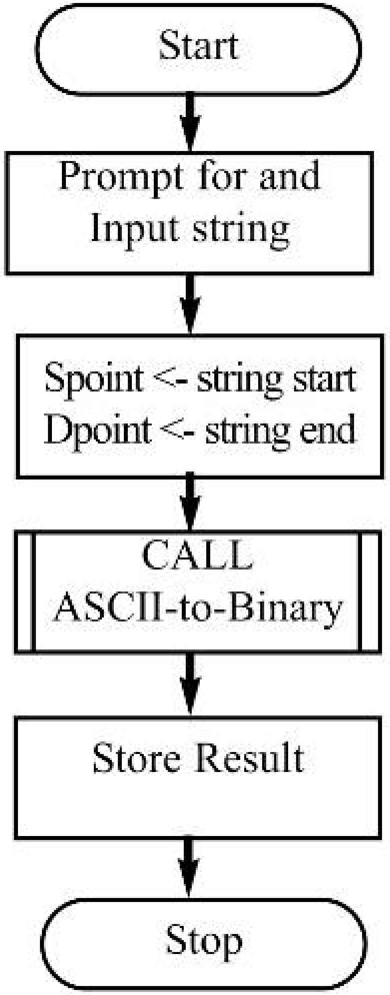
- Program 7-7, page 209, shows the calling program for the module that converts ASCII to binary.

```
        EXTRN  ASC2B_CON:FAR
MAIN    PROC   FAR
        MOV    AX,@DATA
        MOV    DS,AX
        MOV    AH,09                     ;DISPLA
        MOV    DX,OFFSET PROMPT1
        INT    21H
        MOV    AH,0AH                    ;INPUT
        MOV    DX,OFFSET ASC_AREA
        INT    21H
        MOV    SI,OFFSET ASC_NUM
        MOV    BH,00
        MOV    BL,ACT_LEN
        DEC    BX
        CALL   ASC2B_CON
        MOV    BINNUM,AX ;SAVE THE BINARY (H
        MOV    AH,4CH
        INT    21H    ;GO BACK TO OS
MAIN    ENDP
```

The program sets up the data segment, then **inputs ASCII data** from the keyboard, **places it in memory**, then **calls the routine** to convert the number to binary.

Finally, the hex result is **stored in memory**.

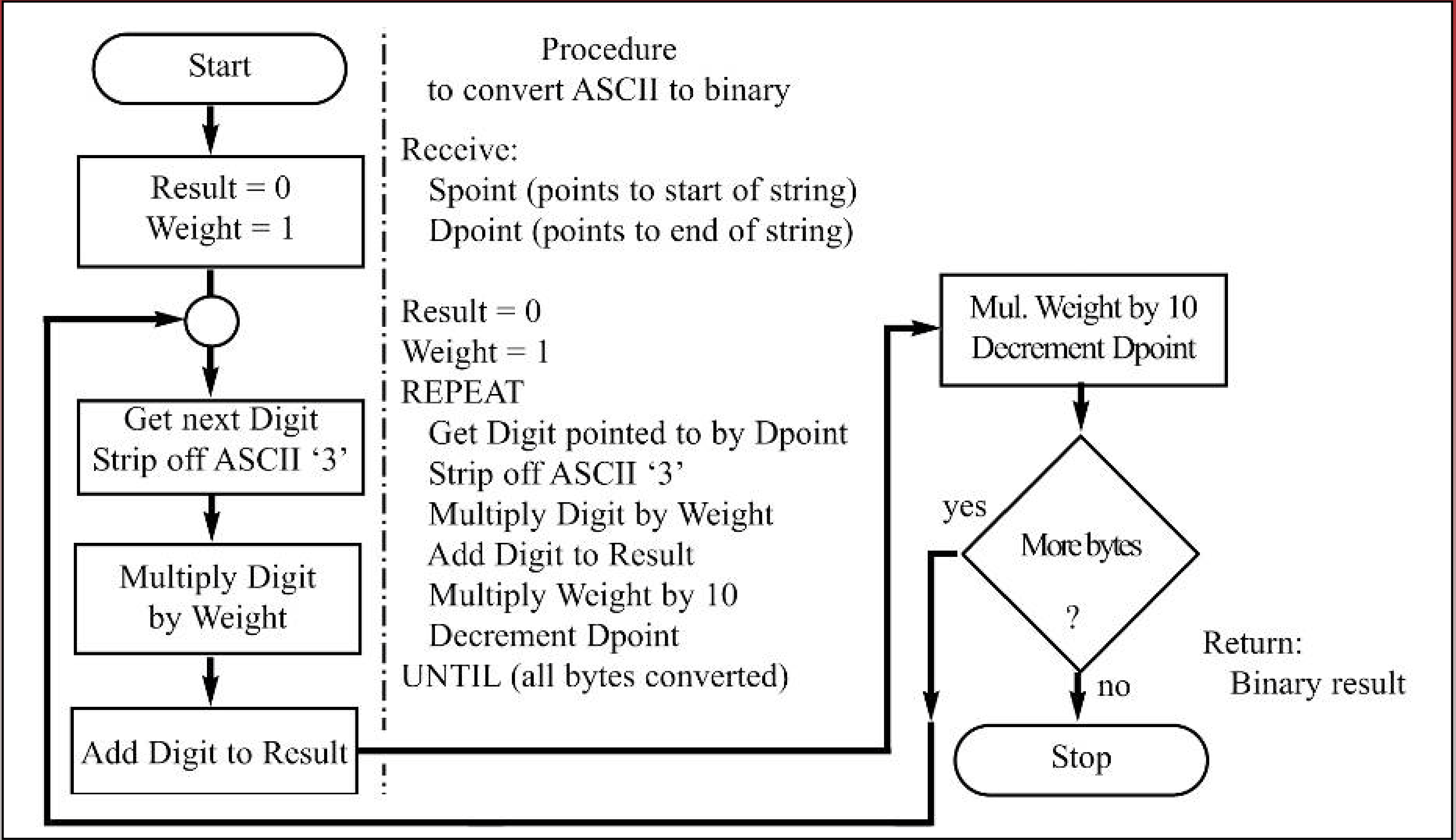
7.2: SOME VERY USEFUL MODULES
calling module flowchart – main module



Main Module

Prompt for string
Input string
Spoint <- points to start of string
Dpoint <- points to end of string
CALL procedure to convert ASCII to binary
    Send: Spoint, Dpoint
    Return: binary result
Store binary result

7.2: SOME VERY USEFUL MODULES calling module flowchart - procedure



7.3: PASSING PARAMETERS AMONG MODULES
passing parameters via registers

- Parameters can be passed from one module to another through registers, memory, or the stack.
  - Fixed values, variables, arrays of data, memory pointers.
- When there is a need to pass parameters among various modules, one *could* use CPU registers.
  - The programmer must clearly document the registers used for the incoming data & registers expected to have the result after the execution of the subroutine.
    - The limited number of CPU registers a major limitation associated with this method of parameter passing.

7.3: PASSING PARAMETERS AMONG MODULES
passing parameters via memory

- OS and IBM BIOS frequently pass parameters via memory by defining an area of RAM and passing parameters to these locations.
  - There must be universal agreement as to addresses of the memory area, to ensure modules can be run on the hardware & software of various companies.
  - The only reason that BIOS & OS use memory area for passing parameters is because IBM & Microsoft worked closely to decide on the memory addresses.
- The *most widely* used method of passing parameters is via the stack, making parameters both register *and* memory independent.

7.3: PASSING PARAMETERS AMONG MODULES
passing parameters via the stack

- The stack is a very critical part of every program.
  - Playing with it can be risky.
- When a module is called, the stack holds the return address, where the program returns after execution.
  - If the stack contents are altered, the program can **crash**.

7.3: PASSING PARAMETERS AMONG MODULES
passing parameters via the stack

- Program 7-8, page 212, demonstrates this method of parameter passing, written with the following requirements.
  – The main module gets three word-sized operands from the data segment, stores them on the stack, and calls the subroutine.
  – The subroutine gets the operands from the stack, adds them together, holds the result in a register, and returns control to the main module.
  – The main module stores the result of the addition.

stack contents analysis for Program 7-8

- Step-by-step analysis of stack pointer and contents:
  - Assume the stack pointer has the value SP = 17FEH.

```
MOV     DS,AX
PUSH    VALUE3          ;SAVE VALUE3 ON STACK
PUSH    VALUE2          ;SAVE VALUE2 ON STACK
PUSH    VALUE1          ;SAVE VALUE1 ON STACK
CALL    SUBPROG6        ;CALL THE ADD ROUTINE
MOV     RESULT,AX       ;STO
```

See the program module listings on page 212 of your textbook.

  - VALUE3 = 25F1H is pushed and SP = 17FC.
    - Low byte to low address and high byte to high address.
  - VALUE2 = 1979H is pushed, then SP = 17FA.
  - VALUE1 = 3F62H is pushed, then SP = 17F8.
  - CALL SUBPROG6 is a FAR call, so, both CS & IP are pushed onto the stack, making SP = 17F4.

7.3: PASSING PARAMETERS AMONG MODULES
stack contents analysis for Program 7-8

- In the subprogram module, register **BP** is saved by PUSHing **BP** onto the stack, making SP = 17F2.

```
        PUSH    BP          ;SAVE BP
        MOV     BP,SP             ;SET BP FOR INDEXING
        MOV     AX,[BP]+6         ;MOV VALUE1 TO AX
        MOV     CX,[BP]+8         ;MOV VALUE2 TO CX
        MOV     DX,[BP]+10        ;MOV VALUE3 TO DX
        ADD     AX,CX             ;ADD VALUE2 TO VALUE1
        ADC     BX,00             ;KEEP THE CARRY IN BX
        ADD     AX,DX             ;ADD VALUE3
        ADC     BX,00             ;KEEP THE CARRY IN BX
        POP     BP          ;RESTORE BP BEFO
        RET     6           ;RETURN AND ADD 6 T
UBPROG6         ENDP
```

In the subprogram, **BP** is used to access values in the stack.

7.3: PASSING PARAMETERS AMONG MODULES
stack contents analysis for Program 7-8

- **SP** is first copied to **BP**, as only **BP** can be used in indexing mode with the stack segment (SS) register.

```
      PUSH   BP      ;SAVE BP
      MOV    BP,SP         ;SET BP FOR INDEXING
      MOV    AX,[BP]+6     ;MOV VALUE1 TO AX
      MOV    CX,[BP]+8     ;MOV VALUE2 TO CX
      MOV    DX,[BP]+10    ;MOV VALUE3 TO DX
      ADD    AX,CX         ;ADD VALUE2 TO VALUE1
      ADC    BX,00         ;KEEP THE CARRY IN BX
      ADD    AX,DX         ;ADD VALUE3
      ADC    BX,00         ;KEEP THE CARRY IN BX
      POP    BP      ;RESTORE BP BEFO
      RET    6       ;RETURN AND ADD 6 T
UBPROG6      ENDP
```

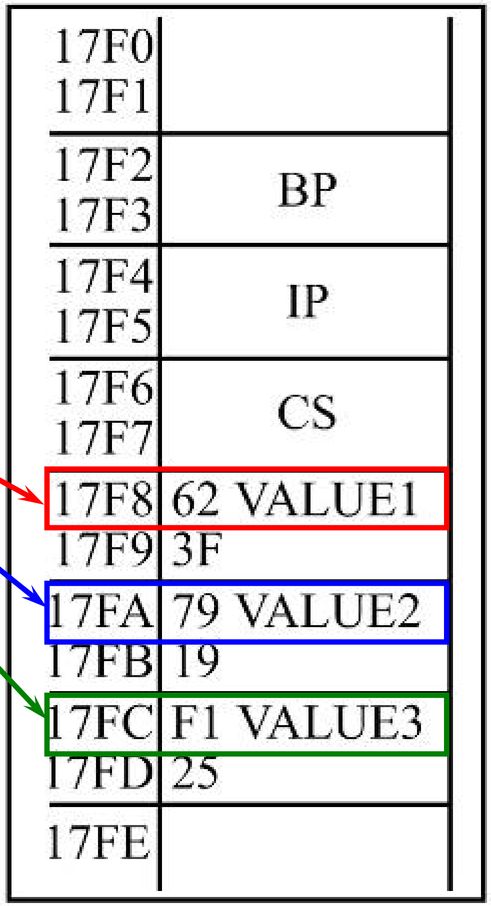
"MOV AX,[SP+4]" will cause an error.

7.3: PASSING PARAMETERS AMONG MODULES
stack contents analysis for Program 7-8

- MOV AX,[BP]+6 loads **VALUE1** into AX.
  - [BP]+6=17F2+6=17F8, exactly where VALUE1 is located.

```
MOV    AX,[BP]+6     ;MOV VALUE1 TO AX
MOV    CX,[BP]+8     ;MOV VALUE2 TO CX
MOV    DX,[BP]+10    ;MOV VALUE3 TO DX
```

  - BP+8=17F2+8=17FA, where VALUE2 is located.
  - BP+10=17F2H+10=17FCH, where VALUE3 is located.



Program 7-8: Stack Contents Diagram

7.3: PASSING PARAMETERS AMONG MODULES
stack contents analysis for Program 7-8

- After all parameters are brought into the CPU by the present module & processed (in this case added), the module restores the original **BP** contents by **POP**ping **BP** from stack.
  - SP = 17F4.

```
POP     BP        ;RESTORE BP BEFORE RETURNING
RET     6         ;RETURN AND ADD 6 TO SP TO BYPAS
```

7.3: PASSING PARAMETERS AMONG MODULES
stack contents analysis for Program 7-8

- **RET 6** is a new instruction.
  - "RET n" instruction means first to POP CS:IP (IP only if the CALL was NEAR) off the top of the stack and then add *n* to the SP.

```
ADC     BX,00                                    BX
POP     BP       ;RESTORE BP BEFORE RETURNING
RET     6        ;RETURN AND ADD 6 TO SP TO BYPAS
OG6     ENDP
```

  - After popping CS and IP off the stack, the stack pointer is incremented four times, making SP = 17F8.
    - Adding 6 to bypass the six locations of the stack where the parameters are stored makes SP = 17FEH, its original value.

7.3: PASSING PARAMETERS AMONG MODULES
stack contents analysis for Program 7-8

- If the program had a RET instruction, instead of the "RET 6“, every time this subprogram is executed it will cause the stack to lose six locations.
  – If this practice of losing some area of the stack continues, eventually the stack could be reduced to a point where the program would run out of stack and crash.

Dec	Hex	Bin
7	7	00000111

ENDS ; SEVEN

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