Strings, Procedures and Macros

The 8086 String Instructions

Moving a String:

REPEAT
MOVE BYTE FROM SOURCE STRING
TO DESTINATION STRING
UNTIL ALL BYTES MOVED

Contd..

- The more detailed algorithm is as follows:

```
INITIALIZE SOURCE POINTER, SI
INITIALIZE DESTINATION POINTER, DI
INITIALZE COUNTER, CX
REPEAT
        COPY BYTE FROM SOURCE TO

        DESTINATION
        INCREMENT SOURCE POINTER
        INCREMENT DESTINATION POINTER
        DECREMENT COUNTER
UNTIL COUNTER = 0
```

Contd..

- The single 8086 string instruction ***MOVSB***, will perform all the actions in the REPEAT-UNTIL loop
- The ***MOVSB*** instruction will copy a byte from the location pointed to by the SI register to a location pointed to by DI register
- It will then automatically increment/decrement SI and DI to point to the next source location

Contd..

- If you add a special prefix called the repeat prefix (***REP***) in front of the MOVSB instruction, the MOVSB instruction will be repeated and CX decremented until CX is counted down to zero

- In order for the MOVSB instruction to work correctly, the source index register, SI, must contain the offset of the start of the source string and the destination index register, DI, must contain the offset of the start of the destination location

Contd..

- Also, the number of string elements to be moved must be loaded into the CX register
- The string instructions will automatically increment or decrement the pointers after each operation, depending on the state of the ***direction flag DF***

Contd..

- If the direction flag is cleared with a ***CLD*** instruction, then the pointers in SI and DI will be automatically be incremented after each string operation

- If the direction flag is set with an ***STD*** instruction, then the pointers in SI and DI will be automatically be decremented after each string operation

Contd..

Note:

- For String instructions, an offset in DI is added to the segment base represented by the number in the ES register to produce a physical address

- If DS and ES are initialized with the same value, then SI and DI will point to locations in the same segment

Example

```
DATA SEGMENT
        TEST_MSG DB 'HELLO'
                DB 100 DUP(?)
        NEW_LOC DB 5 DUP(0)
DATA ENDS

CODE SEGMENT
        ASSUME CS:CODE, DS:DATA, ES:DATA
START:  MOV AX, DATA
                MOV DS, AX
                MOV ES, AX
```

Contd..

```
            LEA  SI, TEST_MSG
            LEA  DI, NEW_LOC
            MOV CX, 05
            CLD
REP  MOVSB
      MOV AH, 4CH
      INT 21H
      CODE ENDS
      END START
```

Contd..

Note:

- The ***MOVSW*** instruction can be used to move a string of words; Depending on the state of the direction flag, SI and DI will automatically be incremented or decremented by 2 after each word move

- If the REP prefix is used, CX will be decremented by 1 after each word move, so CX should be initialized with the number of words in the string

Advantage of using MOVSB/MOVSW:

- The string instruction is much more efficient than using a sequence of standard instructions, because the 8086 only has to fetch and decode the REP MOVSB instruction once

- A standard instruction sequence such as MOV, MOV, INC, INC, LOOP etc would have to be fetched and decoded each time around the loop

Using the CMPSB (Compare String Byte)

- The ***CMPSB*** instruction will compare the byte pointed to by SI with the byte pointed to by DI and set the flags according to the result

- It also increment/decrement the pointers SI and DI (depending on DF) to point to the next string elements

Contd..

- The ***REPE*** prefix in front of this instruction tells the 8086 to decrement the CX register after each compare, and repeat the CMPSB instruction if the compared bytes were equal ***AND*** CX is not yet decremented down to zero

Example

```
DATA SEGMENT
        STR1 DB 'HELLO'
        STR2 DB 'HELLO'
DATA ENDS

CODE SEGMENT
        ASSUME CS:CODE, DS:DATA, ES:DATA
START:          MOV AX, DATA
                MOV DS, AX
                MOV ES, AX
```

Contd..

```
                LEA  SI, STR1
                LEA  DI, STR2
                MOV CX, 05
                CLD
  REPE      CMPSB
                 JNE NOTEQUAL
                ; DISPLAY EQUAL
                JMP EXIT
NOTEQUAL:
                ; DISPLAY NOT EQUAL
     EXIT:  MOV AH, 4CH
                INT 21H
             CODE ENDS
            END START
```

Writing and using Procedures

- To avoid writing the sequence of instructions in the program each time you need them, you can write the sequence as a separate subprogram called a procedure
- You use the CALL instruction to send the 8086 to the starting address of the procedure in memory

Contd..

- A ***RET*** instruction at the end of the procedure returns execution to the next instruction in the main line
- Procedures can even be nested

8086 CALL and RET Instructions

- A CALL instruction in the mainline program loads the Instruction Pointer and in some cases also the code segment register with the starting address of the procedure
- At the end of the procedure, a RET instruction sends execution back to the next instruction after the CALL in the mainline program

The CALL Instruction Overview

- The 8086 CALL instruction performs 2 operations when it executes
- First, it stores the address of the instruction after the CALL instruction on the stack
- This address is called the return address because it is the address that execution will return to after the procedure executes

Contd..

- If the CALL is to a procedure in the same code segment, then the call is ***near***, and only the instruction pointer contents will be saved on the stack

- If the CALL is to a procedure in another code segment, the call is ***far*** ; In this case both the instruction pointer and the code segment register contents will be saved on the stack

Contd..

- Second operation of the CALL instruction is to change the contents of the instruction pointer and, in some cases, the contents of the code segment register to contain the starting address of the procedure

Direct within-segment NEAR CALL

Opcode	***DispLow***	***DispHigh***

Operation:

IP ← Disp16 -- (SP) ← Return Link

- The starting address of the procedure can be anywhere in the range of **-32,768** bytes to **+32,767** bytes from the address of the instruction after the CALL

Indirect within-segment NEAR CALL

Opcode	mod 010 r/m		

Operation:

IP ← Reg16 --(SP) ← Return Link

IP ← Mem16 --(SP) ← Return Link

Direct Inter-Segment FAR CALL

Opcode	Offset-low	Offset-high	Seg-low	Seg-high

Operation:
CS ← SegBase
IP ← Offset

Indirect Inter-Segment FAR CALL

Opcode	***Mem-low***	***Mem-high***

Operation:
CS ← SegBase
IP ← offset

- New value of IP and CS is got from memory

Contd..

- The first word from memory is put in the instruction pointer and the second word from memory is put in the code segment register

The 8086 RET Instruction

- A return at the end of the procedure copies this value from the stack back to the instruction pointer to return execution to the calling program
- When the 8086 does a ***far*** call, it saves the contents of both the instruction pointer and the code segment register on the stack

Contd..

- A RET instruction at the end of the procedure copies these values from the stack back into the IP and CS register to return execution to the mainline program

Contd..

Opcode

Operation:

- Intra-segment return
- Inter-segment return

The 8086 Stack

- 8086 lets us to set aside entire 64 KB segment of memory as a stack
- The stack segment register is used to hold the upper 16 bits of the starting address you give to the stack segment
- 8086 produces the physical address for a stack location by adding the offset contained in the SP register to the stack segment base address represented by the 16-bit number in the SS register

Contd..

- SP register is automatically decremented by 2 before a word is written to the stack
- This means that at the start of your program you must initialize the SP register to point to the top of the memory you set aside as a stack rather than initializing it to point to the bottom location

Contd..

```
STACK_SEG  SEGMENT  STACK
                   DW   40  DUP(0)
STACK_TOP  LABEL  WORD
STACK_SEG ENDS

CODE SEGMENT
        ASSUME CS:CODE,
SS:STACK_SEG
```

Contd..

```
START:
MOV   AX, STACK_SEG
MOV SS, AX
LEA SP, STACK_TOP
                    .
                    .
CODE ENDS
END START
```

Passing Parameters to and from Procedures

- Often when we call a procedure, we want to make some data values or addresses available to the procedure
- Likewise, we often want a procedure to make some processed data values or addresses available to the main program

Contd..

- The 4 major ways of passing parameters to and from a procedure are:
  1. In Registers
  2. In dedicated memory locations accessed by name
  3. With pointers passed in registers
  4. With the stack

Contd..

We use a simple BCD-to-Binary program to demonstrate all the 4 types of parameter passing techniques

1. Passing Parameters in Registers

```
DATA SEGMENT
        BCD_INPUT DB 17H
        BIN_VALUE DB ?
DATA ENDS

STACK_SEG SEGMENT
                        DW 40 DUP(0)
                TOP_STACK LABEL WORD
STACK_SEG ENDS
```

Contd..

```
CODE  SEGMENT
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
START:     MOV AX, DATA
           MOV DS, AX
           MOV AX, STACK_SEG
           MOV SS, AX
           MOV SP, OFFSET  TOP_STACK
           ; SAME AS LEA SP, TOP_STACK
```

Contd..

```
MOV AL, BCD_INPUT
CALL BCD_TO_BIN
MOV BIN_VALUE, AL

MOV AH, 4CH
INT 21H
```

Contd..

```
BCD_TO_BIN PROC NEAR
        PUSHF
        PUSH BX
        PUSH CX
        ; DO THE CONVERSION
        MOV BL, AL
        AND BL, 0FH
        AND AL, 0F0H
        MOV CL, 04H
        SHR AL, CL
        MOV BH, 0AH
```

Contd..

```
        MUL BH
        ADD AL, BL
        ; End Of Conversion; Binary Result In AL

        POP CX
        POP BX
        POPF
        RET
BCD_TO_BIN ENDP
CODE ENDS
END START
```

Contd..

Note that we don't push the AX register because we are using it to pass a value to the procedure and expecting the procedure to pass a different value back to the calling program in it

Contd..

- The ***disadvantage*** of using registers to pass parameters is that the number of registers limits the number of parameters you can pass
- You can't, for example, pass an array of 100 elements to a procedure in registers

2. Passing Parameters in General Memory

We can directly access the parameters by name from the procedure

Contd..

```
DATA SEGMENT
        BCD_INPUT DB 17H
        BIN_VALUE DB ?
DATA ENDS

STACK_SEG SEGMENT
                DW 40 DUP(0)
        TOP_STACK LABEL WORD
STACK_SEG ENDS
```

Contd..

```
CODE  SEGMENT
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
START:      MOV AX, DATA
            MOV DS, AX
            MOV AX, STACK_SEG
            MOV SS, AX
            MOV SP, OFFSET  TOP_STACK
            ; SAME AS LEA SP, TOP_STACK
```

Contd..

```
CALL BCD_TO_BIN

MOV AH, 4CH
INT 21H
```

Contd..

```
BCD_TO_BIN PROC NEAR
        PUSHF
        PUSH AX
        PUSH BX
        PUSH CX
        MOV AL, BCD_INPUT
        ; DO THE CONVERSION
        MOV BL, AL
        AND BL, 0FH
        AND AL, 0F0H
        MOV CL, 04H
        SHR AL, CL
        MOV BH, 0AH
```

Contd..

```
        MUL BH
        ADD AL, BL
        ; End Of Conversion; Binary Result In AL
        MOV  BIN_VALUE, AL
        POP CX
        POP BX
        POP AX
        POPF
        RET
BCD_TO_BIN ENDP
CODE ENDS
END START
```

Contd..

- The limitation of this method is that this procedure will always look to the memory location named BCD_INPUT to get it data and will always put its result in the memory location called BIN_VALUE
- In other words, the way it is written we cant easily use this procedure to convert a BCD number in some other memory location

3. Passing Parameters Using Pointers

- A parameter-passing method which overcomes the disadvantage of using data item names directly in a procedure is to use registers to pass the procedure pointers to the desired data
- This pointer approach is more versatile because you can pass the procedure pointers to data anywhere in memory
- You can pass pointers to individual values or pointers to arrays or strings

Contd..

```
DATA SEGMENT
        BCD_INPUT DB 17H
        BIN_VALUE DB ?
DATA ENDS

STACK_SEG SEGMENT
                        DW 40 DUP(0)
                TOP_STACK LABEL WORD
STACK_SEG ENDS
```

Contd.

```
CODE  SEGMENT
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
START:      MOV AX, DATA
            MOV DS, AX
            MOV AX, STACK_SEG
            MOV SS, AX
            MOV SP, OFFSET  TOP_STACK
            ; SAME AS LEA SP, TOP_STACK
```

Contd..

```
MOV SI, OFFSET BCD_INPUT
        ; or LEA SI, BCD_INPUT
MOV DI, OFFSET BIN_VALUE
        ; or LEA DI, BIN_VALUE
CALL BCD_TO_BIN

MOV AH, 4CH
INT 21H
```

Contd..

```
BCD_TO_BIN PROC NEAR
        PUSHF
        PUSH AX
        PUSH BX
        PUSH CX
        MOV AL, [SI]
        ; DO THE CONVERSION
        MOV BL, AL
        AND BL, 0FH
        AND AL, 0F0H
        MOV CL, 04H
        SHR AL, CL
        MOV BH, 0AH
```

Contd..

```
        MUL BH
        ADD AL, BL
        ; End Of Conversion; Binary Result In AL
        MOV  [DI], AL
        POP CX
        POP BX
        POP AX
        POPF
        RET
BCD_TO_BIN ENDP
CODE ENDS
END START
```

4. Passing Parameters Using the Stack

- To pass parameters to a procedure using the stack, we push the parameters on the stack somewhere in the mainline program before we call the procedure
- Instructions in the procedure then read the parameters from the stack as needed

Contd..

- Likewise, parameters to be passed back to the calling program are written to the stack by instructions in the procedure and read off the stack by instructions in the mainline program

Contd..

```
DATA SEGMENT
        BCD_INPUT DB 17H
        BIN_VALUE DB ?
DATA ENDS

STACK_SEG SEGMENT
                        DW 40 DUP(0)
                TOP_STACK LABEL WORD
STACK_SEG ENDS
```

Contd..

```
CODE  SEGMENT
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
START:        MOV AX, DATA
              MOV DS, AX
              MOV AX, STACK_SEG
              MOV SS, AX
              MOV SP, OFFSET  TOP_STACK
              ; SAME AS LEA SP, TOP_STACK
```

Contd..

```
MOV  AL, BCD_INPUT
PUSH  AX
CALL  BCD_TO_BIN
POP  AX
MOV BIN_VALUE, AL

MOV AH, 4CH
INT 21H
```

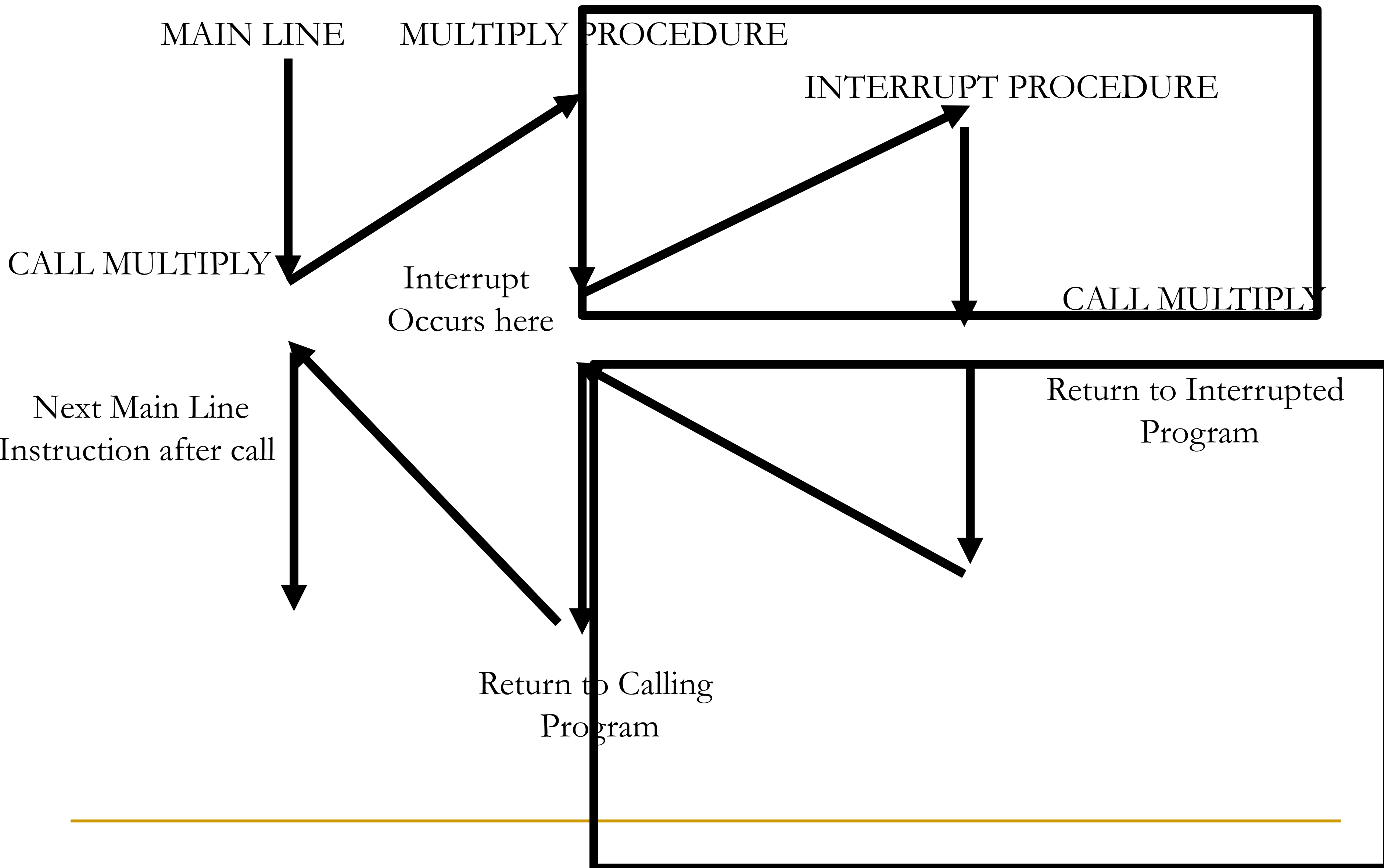
Contd..

```
BCD_TO_BIN PROC NEAR
        PUSHF
        PUSH AX
        PUSH BX
        PUSH CX
        PUSH BP
        MOV BP, SP
        MOV AX, [BP+12]
        ; DO THE CONVERSION
        MOV BL, AL
        AND BL, 0FH
        AND AL, 0F0H
        MOV CL, 04H
        SHR AL, CL
        MOV BH, 0AH
```

Contd..

```
        MUL BH
        ADD AL, BL
        ; End Of Conversion; Binary Result In AL
        MOV  [BP+12], AX
        POP BP
        POP CX
        POP BX
        POP AX
        POPF
        RET
BCD_TO_BIN ENDP
CODE ENDS
END START
```

REENTRANT PROCEDURES



Contd..

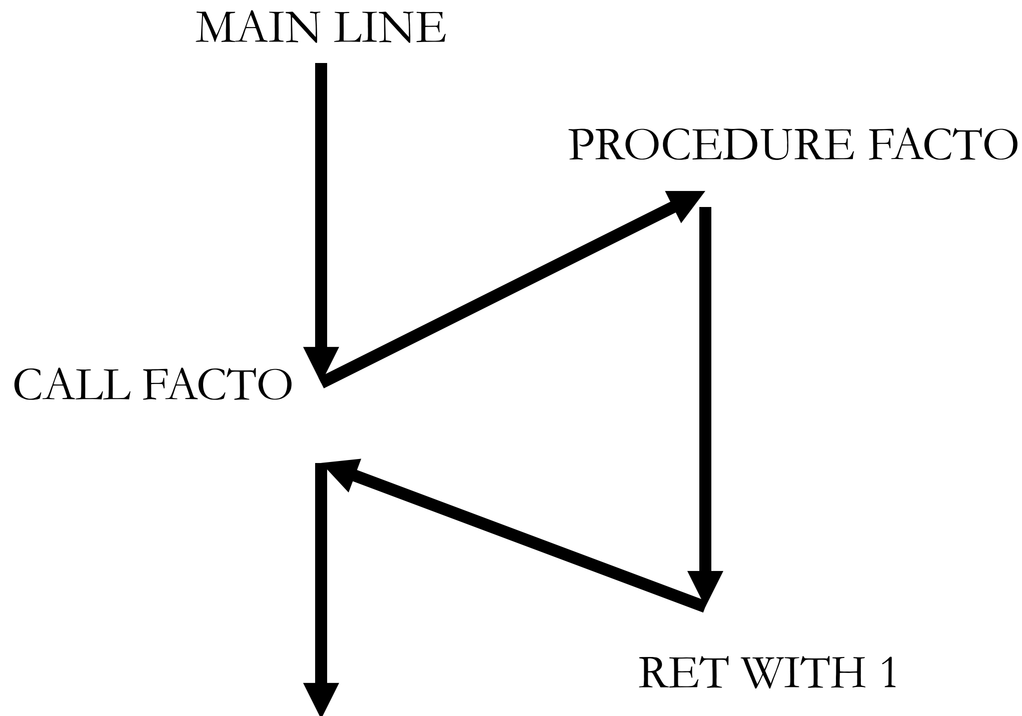
- To be reentrant, a procedure must first of all push the flags and all registers used in the procedure
- Also, to be reentrant, a program should use only registers or the stack to pass parameters

Recursive Procedures

- A recursive procedure is a procedure which calls itself
- Recursive procedures are often used to work with complex data structures.

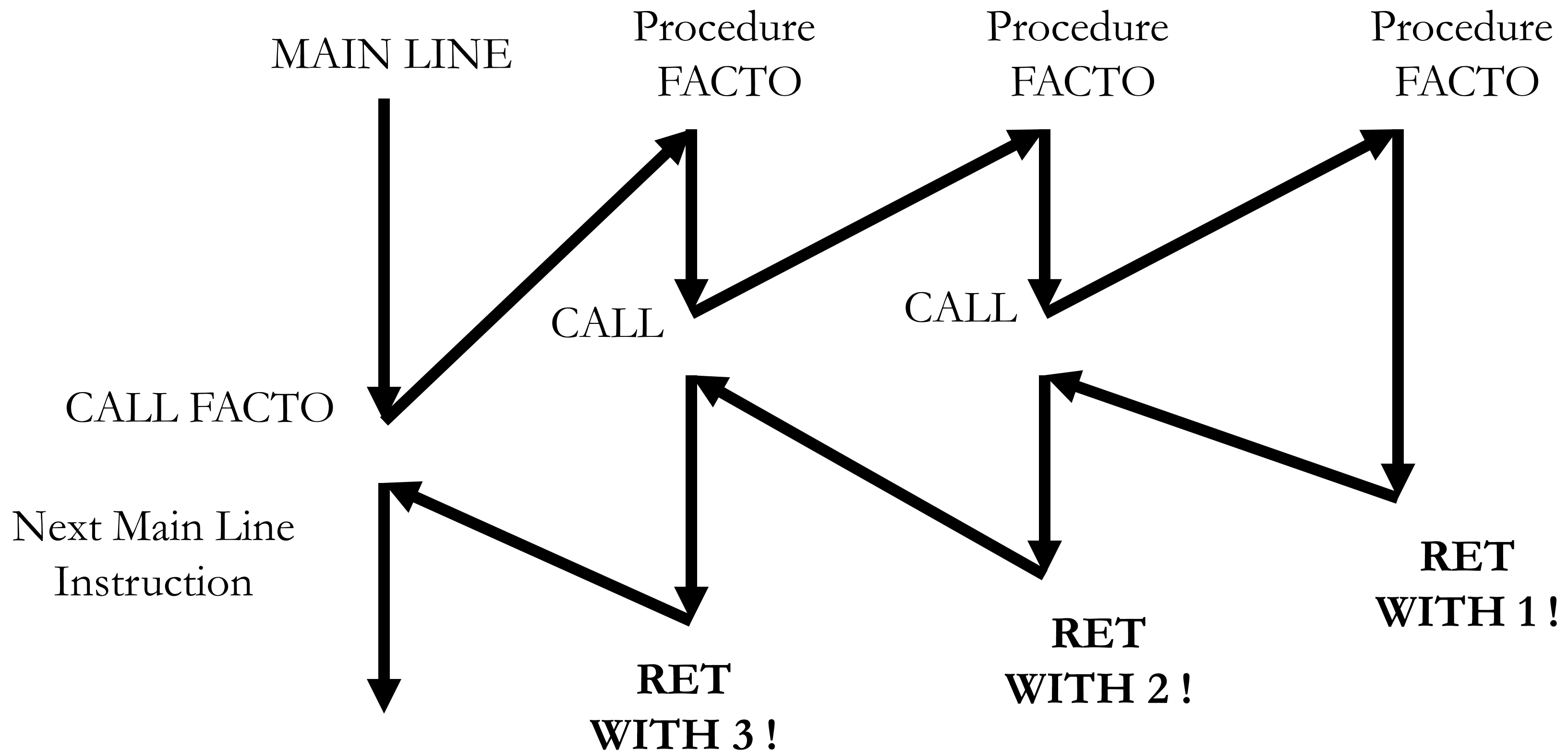
Recursive Procedure Example

- Flow diagram for N=1



Contd..

- Flow diagram for N=3



Contd..

```
PROCEDURE FACTO
        IF N = 1
                FACTORIAL = 1
                RET
        ELSE
                REPEAT
                        DECREMENT   N
                        CALL FACTO
                UNTIL N=1
                MULTIPLY (N-1)! X PREVIOUS  N
                RET
```

Contd..

```
DATA SEGMENT
        N   DB   06H
        FACT    DW   ?
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
```

Contd..

```
START:
        MOV AX, DATA
        MOV DS, AX

        MOV AX, 1

        MOV BL, N
        MOV BH, 0

        CALL FACTORIAL

        MOV FACT, AX

        MOV AH, 4CH
        INT 21H
```

Contd..

```
FACTORIAL PROC
    CMP BX, 1
    JE L1

    PUSH BX
    DEC BX
    CALL FACTORIAL

    POP BX
    MUL BX

  L1:RET
    FACTORIAL ENDP

    CODE ENDS
    END START
```

Writing and Calling Far Procedures

- A FAR procedure is one that is located in a segment which has a different name from the segment containing the CALL instruction
- To get the starting address of a far procedure, the 8086 must change the contents of both the Code Segment register and the Instruction Pointer

Contd..

- At the end of the far procedure, both the contents of the code segment register and the contents of the instruction pointer must be popped off the stack to return to the calling program

Contd..

```
CODE   SEGMENT
       ASSUME  CS:CODE, DS:DATA,
SS:STCK_SEG
              :
       CALL  MULTIPLY_32
              :
CODE ENDS
```

Contd..

```
PROCEDURES  SEGMENT
      MULTIPLY_32  PROC  FAR
            ASSUME CS:PROCEDURES
                  :
                  :
      MULTIPLY_32  ENDP
PROCEDURES  ENDS
```

Contd..

- If a procedure is in a different segment from the CALL instruction, you must declare it far with the FAR Assembler Directive

- Also you must put an ASSUME statement in the procedure to tell the assembler what segment base to use when calculating the offsets of the instructions in the procedure

Accessing a Procedure and Data in a Separate Assembly Module

- The best way to write a large program is to divide it into a series of modules
- Each module can be individually written, assembled, tested and debugged
- The object code files for the modules can then be linked together

Contd..

- In the module where a variable or procedure is declared, you must use the ***PUBLIC*** directive to let the linker know that the variable or procedure can be accessed from other modules
- The statement ***PUBLIC DISPLAY***, for example, tells the linker that a procedure or variable named ***DISPLAY*** can be legally accessed from another assembly module

Contd..

- In a module which calls a procedure or accesses a variable in another module, you must use the ***EXTRN*** directive to let the assembler know that the procedure or variable is not in this module
- The statement ***EXTRN DISPLAY:FAR***, ***SECONDS:BYTE*** tells the linker that ***DISPLAY*** is a far procedure and ***SECONDS*** is a variable of type byte located in another assembly module

Contd..

To summarize, a procedure or variable declared PUBLIC in one module will be declared EXTRN in modules which access the procedure or variable

Defining and Calling a Macro Without Parameters

Macro Definition:

MACRO-NAME MACRO

; ***MACRO DEFINITION***

;

ENDM

Invoking a Macro:

MACRO-NAME

Example

Macro Definition:

```
CLRSCR MACRO
        MOV AH, 00H
        MOV AL, 02H
        INT 10H
ENDM
```

Macro Invocation:

```
        CLRSCR
```

Passing Parameters to Macros

Example:

Macro definition:

```
SETCURSOR MACRO X, Y
        MOV DL, Y
        MOV DH, X
        MOV BH, 00H
        MOV AH, 02H
        INT 10H
ENDM
```

Contd..

- If the macro invocation statement is **SETCURSOR 12, 40** then the macro will be replaced by:

```
MOV DL, 40
MOV DH, 12
MOV BH, 00H
MOV AH, 02H
INT 10H
```

END OF UNIT 2