Assembly Language Programming
String, Procedure and Macro

Outline

- The 8086 String instructions
  - Moving a String
  - Using compare string byte to check password
- Writing and using procedures
  - The CALL and RET instructions
  - The 8086 Stack
  - Using PUSH and POP
  - Passing parameters to and from procedures
  - Writing and debugging program containing procedures
  - Reentrant and Recursive procedures
  - Writing and Calling Far procedures
  - Accessing a procedure
- Writing and using Assembler Macros
  - Comparison Macros and Procedures
  - Defining and calling a Macro without parameters
  - Passing parameters to Macros

The 8086 String instructions

- A string is the series of bytes stored in successive memory locations.
- Word processor or text editor programs can be used to create strings.
- These programs have facility to search through the text.

Moving a String(contd.)

- Definition:
  - You have a string of ASCII characters in successive memory locations in data segment, and you want to move the string to some new location in the data segment.
- Basic pseudo code:

REPEAT

MOVE BYTE FROM SOURCE STRING

TO DESTINATION STRING

UNTIL ALL BYTES MOVED

Moving a String(contd.)

- The basic pseudo code doesn’t help much in understanding how the algorithm will be implemented.
- Expanded code:

```
INITIALIZE SOURCE POINTER, SI
INITIALIZE DESTINATION POINTER, DI
INITIALIZE COUNTER, CX
REPEAT
        COPY BYTE FROM SOURCE TO DESTINATION
        INCREMENT SOURCE POINTER
        INCREMENT DESTINATION POINTER
        DECREMENT COUNTER
UNTIL COUNTER=0
```

Using compare string byte to check password(contd.)

- Definition:
  - We want to compare a user entered password to the correct password stored in the memory. If the passwords do not match we want to sound an alarm and If the passwords matches we will allow access to computer for that user.
- Need:
  - REPEAT-UNTIL
  - Compare String instruction CMPS

Using compare string byte to check password - Code

- **Code:**

```
INITIALIZE PORT DEVICE FOR OUTPUT
INTIALIZE SOURCE POINTER-SI
INITALIZE DESTINATION POINTER-DI
INITIALIZE COUNTER-CX
REPEAT
        COMPARE SOURCE BYTE WITH DESTINATION BYTE
        INCREMENT SOURCE POINTER
        INCREMENT DESTINATION POINTER
        DECREMENT COUNTER
UNTIL (STRING BYTES NOT EQUAL) OR (CX=0)
IF STRING BYTES NOT EQUAL THEN
        SOUND ALARM
STOP
ELSE DO NEXT MAINLINE INSTRUCTION
```

Writing and using procedures

- Avoid writing the same sequence of instruction again and again.
- Write it in a separate subprogram and call that subprogram whenever necessary.
- For that CALL instruction is used.

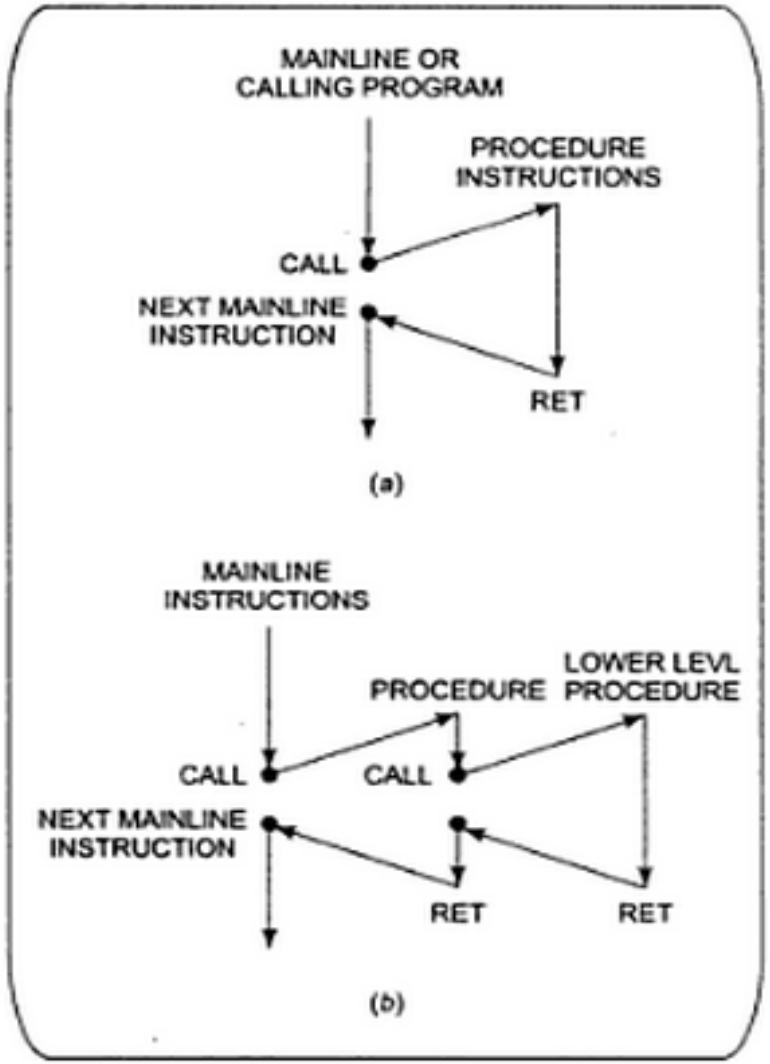
The CALL and RET instructions(contd.)

The CALL Instruction:

- Stores the address of the next instruction to be executed after the CALL instruction to stack. This address is called as the return address.
- Then it changes the content of the instruction pointer register and in some cases the content of the code segment register to contain the starting address of the procedure.

The CALL and RET instructions(contd.)

Chart for CALL and RET instruction →



The CALL and RET instructions(contd.)

Types of CALL instructions:

- DIRECT WITHIN-SEGMENT NEAR CALL: produce the starting address of the procedure by adding a 16-bit signed displacement to the contents of the instruction pointer.
- INDIRECT WITHIN-SEGMENT NEAR CALL: the instruction pointer is replaced with the 16-bit value stored in the register or memory location.
- THE DIRECT INTERSEGMENT FAR CALL: used when the called procedure is in different segment. The new value of the instruction pointer is written as bytes 2 and 3 of the instruction code. The low byte of the new IP value is written before the high byte.
- THE INDIRECT INTERSEGMENT FAR CALL: replaces the instruction pointer and the contents of the segment register with the two 16-bit values from the memory.

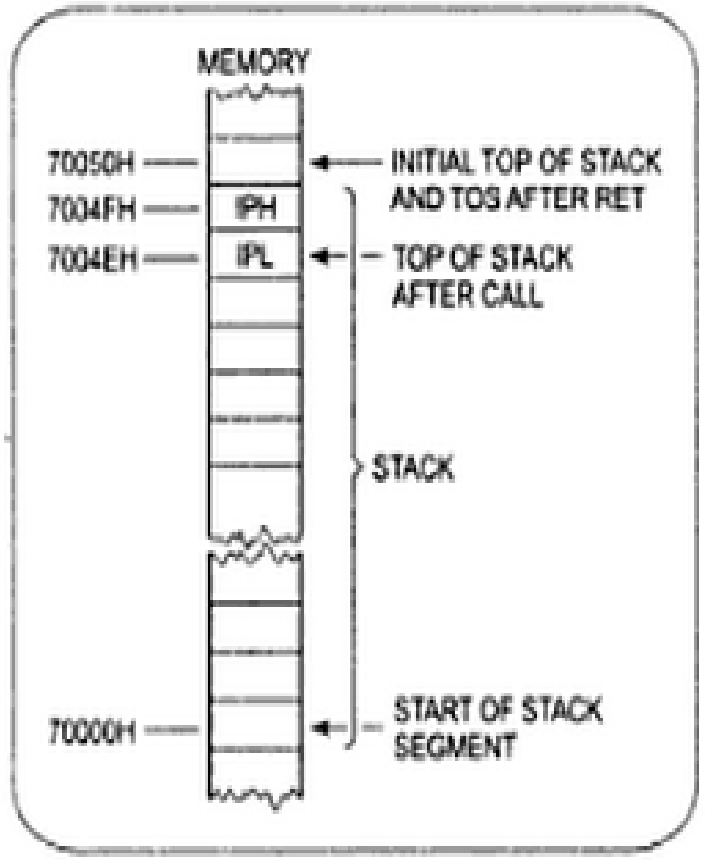
The CALL and RET instructions

The 8086 RET instruction:

- When 8086 does near call it saves the instruction pointer value after the CALL instruction on to the stack.
- RET at the end of the procedure copies this value from stack back to the instruction pointer (IP).

The 8086 Stack

- Section of memory you set aside for storing return addresses.
- Also used to store the contents of the registers for the calling program while a procedure executes.
- Hold data or address that will be acted upon by procedures.



Using PUSH and POP

- The PUSH register/memory instruction decrements the stack pointer by 2 and copies the contents of the specified 16-bit register or memory location to memory at the new top-of-stack location.
- The POP register/memory instruction copies the word on the top-of-stack to the specified 16-bit register or memory location and increments the stack pointer by 2.

Passing parameters to and from procedures

Major ways of passing parameters to and from a procedure:

- In register
- In dedicated memory locations accessed by name
- With pointers passed in registers
- With the stack

Writing and debugging programs containing procedures

- Carefully workout the overall structure of the program and break it down into modules which can easily be written as procedures.
- Simulate each procedure with few instructions which simply pass test values to the mainline program. This is called as dummy or stubs.
- Check that number of PUSH and POP operations are same.
- Use breakpoints before CALL, RET and start of the program or any key points in the program.

Reentrant and Recursive procedures

- **Reentrant procedures:** The procedure which can be interrupted, used and “reentered” without losing or writing over anything.
- **Recursive procedure:** It is the procedure which call itself.

Writing and Calling Far procedures

- It is the procedure that is located in a segment which has different name from the segment containing the CALL instruction.

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

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

Accessing Procedure

Accessing a procedure in another segment

- Put mainline program in one segment and all the procedures in different segment.
- Using FAR calls the procedures can accessed as discuss above.

Accessing procedure and data in separate assembly module

- Divide the program in the series of module.
- The object code files of each module can be linked together.
- In the module where variables or procedures are declared, you must use PUBLIC directive to let the linker know that it can be accessed from other modules.
- In a module which calls procedure or accesses a variable in another module, you must use the EXTERN directive.

Writing and using Assembler Macros

Comparison Macros and Procedures

- A big advantage of using procedures is that the machine codes for the group of instruction in the procedures needs to be loaded in to main memory only once.
- Disadvantage using the procedures is the need for the stack.
- A macro is the group of instruction we bracket and give a name to at the start of the program.
- Using macro avoids the overhead time involved in calling and returning from a procedures.
- Disadvantage is that this will make the program take up more memory than using a procedure.

Defining and calling a Macro without parameters

```
PUSH-ALL  MACRO
          PUSHF
          PUSH AX
          PUSH BX
          PUSH CX
          PUSH DX
          PUSH BP
          PUSH SI
          PUSH DI
          PUSH DS
          PUSH ES
          PUSH SS
ENDM
```

Passing parameters to Macros

- The words NUMBER, SOURCE and DESTINATION are called as the dummy variables. When we call the macro, values from the calling statements will be put in the instruction in place of the dummies.

```
MOVE_ASCII MACRO NUMBER, SOURCE, DESTINATION
        MOV CX, NUMBER          ;Number of characters to be moved In CX
        LEA SI, SOURCE          ;Point SI at ASCII source
        LEA DI, DESTINATION     ;Point DI at ASCII destination
        CLD                     ;Autoincrement pointers after move
REP MOVSB                       ;Copy ASCII string to new location
  ENDM                          ;
```