SYLLABUS

UNIT - II

Implementing standard Program Structures in 8086 Assembly language: Simple sequence programs, jumps flags and conditional jumps, if-then if-then-else multiple if-then-else programs, while do programs, repeat-until programs, instruction timing and delay loops

Strings and procedures: The 8086 string instructions, writing and using procedures; **assembler directives.**

Chapter 4 - Implementing Standard Program Structures in 8086 Assembly Language

Outline

- Simple sequence programs
  - Finding the average of two numbers
  - Converting two ASCII codes to packed BCDs
  - Debugging assembly language programs
- Jumps, flags and conditional jumps
  - The 8086 unconditional jump instructions
  - The 8086 conditional jump instructions
- If-then, if-then-else, multiple if-then-else programs
- While-do programs
- Repeat-until programs
- 8086 addressing modes
- The 8086 Loop instructions
- Instruction timing and delay loops

Simple sequence programs

- **There two programs that we will discuss:**

  1. **Finding the average of two numbers**
  2. **Converting two ASCII codes to packed BCDs**

Finding the average of two numbers(contd.)

- Some common steps that can be followed are:
- **Defining the problem and writing the algorithm**
  - problem definition is simple find average of two numbers
- **Setting up the data structure**

  You need to ask the following questions:
  - Will the data be in memory or register?
  - Is the data of type byte, word or double word?
  - How many data items are there?
  - Does the data represents only positive numbers, or does it represents positive and negative numbers?
  - How the data is structured?
  - **Let's assume for this example that the data is all in memory, that the data Is of type byte, and that the data represents only positive numbers in the range 0 to OFFH.**

Finding the average of two numbers(contd.)

Initialization checklist

- initialize the data segment register
- Do this using MOV AX, DATA and MOV DS, AX instructions.

Choosing instructions to implement the Algorithm

- choose which instructions are needed to implement the algorithm
- For this problem ADD will be used to add two numbers
- DIV will be used to divide the addition by 2

Finding the average of two numbers

```
                                  ; 8086 PROGRAM     F4-01.ASM
                        ;ABSTRACT : This program averages two temperatures
                                  ; named HI_TEMP and LO_TEMP and puts the
                                  ; result in the memory location AV_TEMP.
                        ;REGISTERS : Uses DS, CS, AX, BL
                        ;PORTS     : None used

0000                    DATA    SEGMENT
0000  92                        HI_TEMP  DB  92H    ; Max temp storage
0001  52                        LO_TEMP  DB  52H    ; Low temp storage
0002  ??                        AV_TEMP  DB   ?     ; Store average here
0003                    DATA    ENDS

0000                    CODE    SEGMENT
                                ASSUME CS:CODE, DS:DATA
0000  B8 0000s          START:  MOV  AX, DATA       ; Initialize data segment
0003  8E D8                     MOV  DS, AX
0005  A0 0000r                  MOV  AL, HI_TEMP    ; Get first temperature
0008  02 06 0001r               ADD  AL, LO_TEMP    ; Add second to it
000C  B4 00                     MOV  AH, 00H        ; Clear all of AH register
000E  80 D4 00                  ADC  AH, 00H        ; Put carry in LSB of AH
0011  B3 02                     MOV  BL, 02H        ; Load divisor in BL register
0013  F6 F3                     DIV  BL             ; Divide AX by BL. Quotient in AL,
                                                    ; and remainder in AH
0015  A2 0002r                  MOV  AV_TEMP, AL    ; Copy result to memory
0018                    CODE    ENDS
                                END  START
```

Converting two ASCII codes to packed BCDs

- Defining the problem and writing the algorithm
- The data structure and initialization list
- Masking with the AND instruction
- Moving a nibble with the ROTATE instruction
- Combining bytes or words with the ADD or the OR instruction

Converting two ASCII codes to packed BCDs

Defining the problem and writing the algorithm

- The ASCII codes for the numbers 0 through 9 are 30H through 39H. The lower nibble of the ASCII codes contains the 4-bit BCD code for the decimal number represented by the ASCII code.
- For many applications, we want to convert the ASCII code to its simple BCD equivalent. We can do this by Simply replacing the 3 in the upper nibble of the byte with four 0's.
- For example, suppose we read in 00111001 binary or 39H, the ASCII code for 9. If we replace the upper 4 bits with 0 s. we are left with 00001001 binary or 09H. The lower 4 bits then contain 1001 binary, the BCD code for 9. **Numbers represented as one BCD digit per byte are called unpacked BCD.**
- For applications in which we are going to perform mathematical operations on the BCD numbers, **we usually combine two BCD digits in a single byte. This form is called packed BCD**, Figure 4-2 shows examples of ASCII, unpacked BCD and packed BCD,.

Converting two ASCII codes to packed BCDs

Defining the problem and writing the algorithm

ASCII	5	0011	0101	= 35H
ASCII	9	0011	1001	= 39H
UNPACKED BCD	5	0000	0101	= 05H
UNPACKED BCD	9	0000	1001	= 09H
UNPACKED BCD 5 MOVED TO UPPER NIBBLE		0101	0000	= 50H
PACKED BCD	59	0101	1001	= 59H

FIGURE 4-2 ASCII, unpacked BCD, and packed BCD examples.

Converting two ASCII codes to packed BCDs

The data structure and initialization list

- For this example program, let's assume that the ASCII code for 5 was received and put In the BL register, and the second ASCII code was received and left in the AL register, Since we are not using memory for data in this program.
- We do not need to declare a data segment or initialize the data segment register.

Converting two ASCII codes to packed BCDs

Masking with the AND instruction

- The first operation in the algorithm is to convert a number in ASCII form to Its unpacked BCD equivalent.
- This is done by replacing the upper 4 bits of the ASCII byte with four 0's.
- The 8086 AND instruction can be used to do this operation,when a 1 or a 0 is ANDed with a 0, the result is always a 0.
- ANDing a bit with a 0 is called **masking** that bit because the previous state of the bit is hidden or masked.
- To mask 4 bits in a word, then, all you do is AND each bit you want to mask with a 0. A bit ANDed with a 1, remember is not changed.

Converting two ASCII codes to packed BCDs

Masking with the AND instruction

- For this example the first ASCII number is in the BL register. So we can just AND an immediate number with this register to mask the desired bits.
- The upper 4 bits of the immediate number should be 0's because these correspond to the bits we want to mask in BL.
- The lower 4 bits of the immediate number should be 1s because we want to leave these bits unchanged. The immediate number, then, should be 00001111 binary or OFH.
- The instruction to convert the first ASCII number is AND BL,OFH. When this instruction executes, it will leave the desired unpacked BCD in BL.

ASCII 5	0011 0101
MASK	0000 1111
RESULT	0000 0101

FIGURE 4-3 Effects of ANDing with 1's and 0's.

Converting two ASCII codes to packed BCDs

Moving a nibble with the ROTATE instruction

- The next action in the algorithm is to move the 4 BCD bits in the first unpacked BCD byte to the upper nibble position in the byte. We need to do this so that the 4 BCD bits are in the correct position for packing with the second BCD nibble.
- We are effectively doing here is swapping or exchanging the top nibble with the bottom nibble of the byte.
- The 8086 has a wide variety of rotate and shift instructions, For now, let's look at the rotate instructions, There are two Instructions, **ROL and RCL,** which rotate the bits of a specified **Operand to the left**.

Converting two ASCII codes to packed BCDs

Moving a nibble with the ROTATE instruction

- For **ROL Instruction**, each bit in the specified register or memory location is rotated 1 bit position to the left.
- The bit that was the MSB is rotated around into the LSB position, The old MSB is also copied to the carry flag.
- For the **RCL Instruction**, each bit of the specified register or memory location is also rotated 1 bit position to the left.
- However, the bit that was in the MSB position is moved to the carry flag and the bit that was in the carry flag is moved into the LSB position.

Converting two ASCII codes to packed BCDs

Moving a nibble with the ROTATE instruction

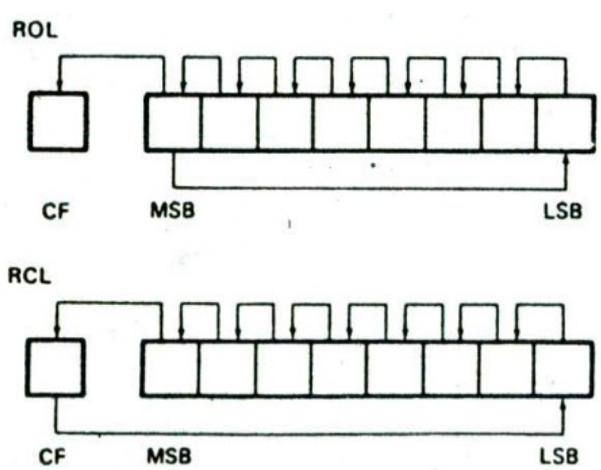


FIGURE 4-4 ROL instruction and RCL instruction operations for byte operands.

Converting two ASCII codes to packed BCDs

Combining bytes or words with the ADD or the OR instruction

- The **ADD** instruction adds the contents of a specified source to the contents of a specified destination and leaves the result in the specified destination.
- For the example program here, the instruction **ADD AL,BL** can be used to combine the two BCD nibbles.
- Another way to combine the two nibbles is with the OR instruction.
- This instruction **OR**s each bit in the specified source with the corresponding bit in the specified destination.
- The result of the **OR**ing is left in the specified destination.
- **OR**ing a bit with a 0 leaves the bit unchanged. To set a bit in a word to a 1. then, all you have to do is OR that bit with a word which has a 1 in that bit position and 0's in all the other bit positions.

Converting two ASCII codes to packed BCDs

- Similar steps can be performed to solve the given problem.

```
                                  ; 8086 PROGRAM F4-05.ASM
                     ;ABSTRACT    : Program produces a packed BCD byte from 2 ASCII-encoded digits
                                  ; The first ASCII digit (5) is loaded in BL.
                                  ; The second ASCII digit (9) is loaded in AL.
                                  ; The result (packed BCD) is left in AL
                     ;REGISTERS   ; Uses CS, AL, BL, CL
                     ;PORTS       : None used

0000                 CODE    SEGMENT
                             ASSUME CS:CODE
0000  B3 35          START:  MOV   BL, '5'    ; Load first ASCII digit into BL
0002  B0 39                  MOV   AL, '9'    ; Load second ASCII digit into AL
0004  80 E3 0F               AND   BL, 0FH    ; Mask upper 4 bits of first digit
0007  24 0F                  AND   AL, 0FH    ; Mask upper 4 bits of second digit
0009  B1 04                  MOV   CL, 04H    ; Load CL for 4 rotates required
000B  D2 C3                  ROL   BL, CL     ; Rotate BL 4 bit positions
000D  0A C3                  OR    AL, BL     ; Combine nibbles, result in AL
000F                 CODE    ENDS
                             END START
```

Debugging Assembly Language Programs

- **Very carefully define the problem** you are trying to solve with the program **and workout the best algorithm** you can.
- **Write and test each sections of the program** as you go, **instead of writing the larger program all at once**.
- **If a program or program section does not work**, first recheck the algorithm to make sure that it really does what you want it to.
- **If the algorithm seems correct**, check to make sure that you have used the **correct instructions to implement** the algorithm.
- If you are hand coding the program this is the next place to check. It is very **easy to get bit wrong** when you are constructing the instruction codes.
- If you are not finding the problem in algorithm, instruction codes or coding then now it's the time to use debugger.
- For longer programs single step approach is tedious rather **put breakpoints** at the victim functions you want to check.

Jumps, Flags and Conditional jumps

- **The real power of a computer comes** from **its ability to choose between two or more sequences of actions based on some condition**, **repeat a sequence of Instructions as long as some condition exists**, or **repeat a sequence of instructions until some condition exists.**
- **Flags indicate whether some condition is present or not.**
- **Jump Instructions are used to tell the computer the address to fetch its next instruction from.**
- **Jump instructions** are used to tell the computer the address of the next instruction to be executed. They change the flow of the program in the desired direction.
- Two types of jump instructions
  - Conditional instructions
  - Unconditional instructions

Jumps, Flags and Conditional jumps

When the 8086 fetches and decodes an **Unconditional Jump instruction**, it always goes to the specified jump destination. You might use this type of Jump Instruction at the end of a program so that the entire program runs over and over, as shown In Figure 4-6.

When the 8086 fetches and decodes a **Conditional Jump instruction**, It **evaluates the -state of a specified flag to** determine whether to fetch its next instruction from the jump destination location or to fetch its next instruction from the next sequential memory location.

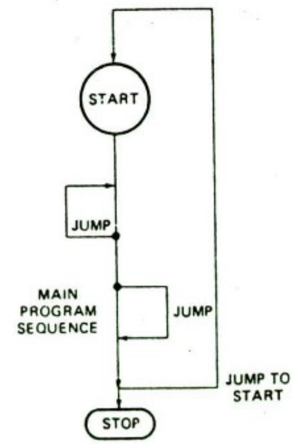


FIGURE 4-6 Change in program flow that can be caused by jump instructions.

The 8086 Unconditional Jump Instructions(contd.)

- Jumps to the desired location without any condition
- **JMP** instruction is used for this purpose.
- When 8086 executes **JMP** instruction, it loads new number into instruction pointer register and in some cases it also loads the number into code segment register.

The 8086 Unconditional Jump Instructions(contd.)

Program for Backward JMP

```
                      ; 8086 PROGRAM     F4-08.ASM
               ;ABSTRACT  : This program illustrates a "backwards" jump
               ;REGISTERS : Uses CS, AL
               ;PORTS     : None used

0000           CODE   SEGMENT
                      ASSUME  CS:CODE
0000  04 03    BACK:  ADD AL, 03H  ; Add 3 to total
0002  90              NOP          ; Dummy instructions to represent those
0003  90              NOP          ; Instructions jumped back over
0004  EB FA           JMP BACK     ; Jump back over instructions to BACK label
0006           CODE   ENDS
                      END
```

FIGURE 4-8 List file of program demonstrating "backward" JMP.

The 8086 Unconditional Jump Instructions(contd.)

Program for Forward JMP

```
                         ; 8086 PROGRAM     F4-09.ASM
                  ;ABSTRACT  : This program illustrates a "forwards" jump
                  ;REGISTERS : Uses CS, AX
                  ;PORTS     : None used

0000              CODE    SEGMENT
                          ASSUME CS:CODE
0000  EB 03 90            JMP THERE      ; Skip over a series of instructions
0003  90                  NOP            ; Dummy instructions to represent those
0004  90                  NOP            ; Instructions skipped over
0005  B8 0000     THERE:  MOV AX, 0000H  ; Zero accumulator before addition instructions
0008  90                  NOP            ; Dummy instruction to represent continuation of execution
0009              CODE    ENDS
                          END
```

FIGURE 4-9 List file of program demonstrating "forward" JMP.

The 8086 Unconditional Jump Instructions

- **Unconditional Jump instruction – Type Overview**
  - Jump within segments – direct
  - Jump within segments - indirect
  - Inter segment or group – direct
  - Inter segment or group - indirect
- **The direct near and short type jump instructions**
  - It can cause the next instruction to be fetched from anywhere **in the current code segment**.
  - It **adds 16-bit signed displacement** contained in the instruction to the instruction pointer register.

The 8086 Unconditional Jump Instructions

- If the **JMP destination is in the same code segment**, the 8086 only has to change the contents of the instruction pointer. This type of jump is referred to as a **near, or intrasegment**, jump.
- If the **JMP destination is in a code segment which has a different name from the segment** in which the JMP instruction is located, the 8086 has to change the contents of both CS and IP to make the jump. This type of jump Is referred to as **a Far, or intersegment**, jump.
- Near and far jumps are further described as either **direct or indirect.**
- If the destination address for the jump is **specified directly as part of the instruction,** then the jump is described as **direct**. You can have a direct near jump or a direct far jump.
- If the **destination address for the jump is contained in a register or memory location**, the jump Is referred to as indirect, because the 8086 has to go to the specified register or memory location to get the required destination address.

JMP = Jump

Within segment or group, IP relative—near and short

Opcode	DispL	DispH

Opcode	Clocks	Operation
E9	15	IP ← IP + Disp16
EB	15	IP ← IP + Disp8 (Disp8 sign-extended)

Within segment or group, Indirect

Opcode	mod 100 r/m	mem-low	mem-high

Opcode	Clocks	Operation
FF	11	IP ← Reg16
FF	18+EA	IP ← Mem16

Inter-segment or group, Direct

Opcode	offset-low	offset-high	seg-low	seg-high

Opcode	Clocks	Operation
EA	15	CS ← segbase IP ← offset

Inter-segment or group, Indirect

Opcode	mod 101 r/m		

Opcode	Clocks	Operation
FF	24+EA	CS ← segbase IP ← offset

FIGURE 4-7 8086 Unconditional Jump instructions. (*Intel Corporation*)

The 8086 Conditional Flags

- **The carry flag:** if addition of two 8-bit numbers is greater than 8-bits then the carry flag will be set to **1 to indicate the final carry produced by the addition.**
- **The parity flag:** indicates whether the word has even number of 1s or odd number of 1s. The **flag is set as 1 if the lower 8 bits of the destination address contains even number of 1s** which is known as even parity.
- **The Auxiliary Carry Flag:** it is used in BCD Addition and Subtraction. If **the carry is produced then lower 2 bytes are added.**
- **The Zero Flag:** set **if the result of arithmetic operation is zero.**
- **The sign Flag:** used to indicate the sign of the number. **MSB 0 means +ve and 1 means –ve**
- **The overflow Flag:** if the result of **signed operation is too large to fit in then it will set.**

The 8086 Conditional Jump instructions

- These are the instruction that will change the flow only when some conditions are met.

MNEMONIC	CONDITION TESTED	"JUMP IF . . ."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
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 OF) =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

If-then, if-then-else, multiple if-then-else programs(contd.)

- **IF-THEN PROGRAMS**

Structure:

IF Condition THEN
Action.

- This structure says that **IF the stated condition is found to be true**, **the series of actions following THEN-will be executed**, **If the condition is false, execution will skip over the actions after the THEN** and proceed with the next mainline instruction.
- **The Simple IF-THEN is Implemented with a Conditional Jump instruction**. In some cases an instruction to set flags is needed before the Conditional Jump instruction.
- **Conditional jump can only be to a location in the range of — 128 bytes to + 127 bytes from the address after the Conditional Jump instruction.**
- If you are not sure whether the destination will be in range, the Instruction sequence shown in Figure 4-1 b will always work. In this sequence, the Conditional Jump instruction only has to jump over the JMP instruction. The JMP Instruction used to get to the **label THERE** can jump to anywhere in the code segment.or even to another code segment.

If-then, if-then-else, multiple if-then-else programs(contd.)

```
        CMP AX, BX    ; Compare to set flags
        JE  THERE     ; If equal then skip correction
        ADD AX, 0002H ; Add correction factor
THERE:  MOV CL, 07H   ; Load count
```

(a)

```
        CMP AX, BX    ; Compare to set flags
        JNE FIX       ; If not equal do correction
        JMP THERE     ; If equal then skip correction
FIX:    ADD AX, 0002H ; Add correction factor

THERE: MOV CL, 07H ; Load count
```

(b)

FIGURE 4-11 Programming conditional jumps. (a) Destinations closer than ±128 bytes. (b) Destinations further than ±128 bytes.

The 8086 IN and OUT Instructions

- The 8086 has two types of input instruction, fixed-**port** and **variable-port.**
- The fixed-port instruction has the format **IN AL, port or IN AX, port**. The term **port** in these Instructions **represents an 8-bit port address** to be put directly in the instruction. The instruction **IN AX,04H**., for example, will copy a word from port 04H to the AX register.
- The variable-port input instruction has the format **IN AL,DX or IN AX,DX.** When using the variable-port Input Instruction, you must first put the address of the desired port in the DX register. If, for example, you load DX with FFF8H and then do an IN AL,DX, the 8086 will copy a byte of data from port FFF8H to the AL register.
- The variable-port input instruction has two major advantages.

➢First, up to 65,536 different Input ports can be specified with the 16-bit port address In DX.

➢Second, the port address can be changed as a program executes by simply putting a different number in DX

The 8086 IN and OUT Instructions

- The 8086 also has a fixed-port output Instruction and a variable-port output instruction.
- The device used for parallel input and output ports on the SDK-86 board and in many microcomputers is the Intel 8255.
- As shown in the block diagram in Figure 4-13, the **8255** basically contains **three 8-bit ports and a control register**.
- Each of the ports and the control register will have a separate address, so you can write to them or read from them.
- The addresses for the ports and control registers for the two 8255s on an SDK-86 board, for example, are as follows:

PORT 2A	FFF8H	PORT 1A	FFF9H
PORT 2B	FFFAH	PORT 1B	FFFBH
PORT 2C	FFFCH	PORT 1C	FFFDH
CONTROL2	FFFEH	CONTROL1	FFFFH

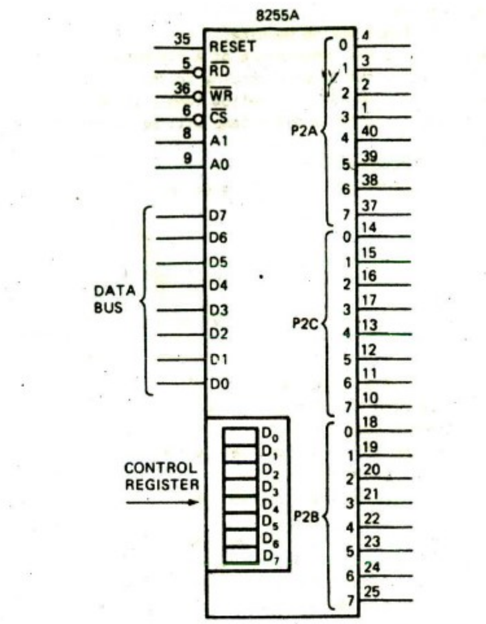


FIGURE 4-13 Block diagram of SDK-86 board's 8255A port.

FIGURE 4-13 Block diagram of SDK-86 board's 8255A port.

If-then, if-then-else, multiple if-then-else programs(contd.)

- **IF-THEN-ELSE PROGRAMS**

  Structure:

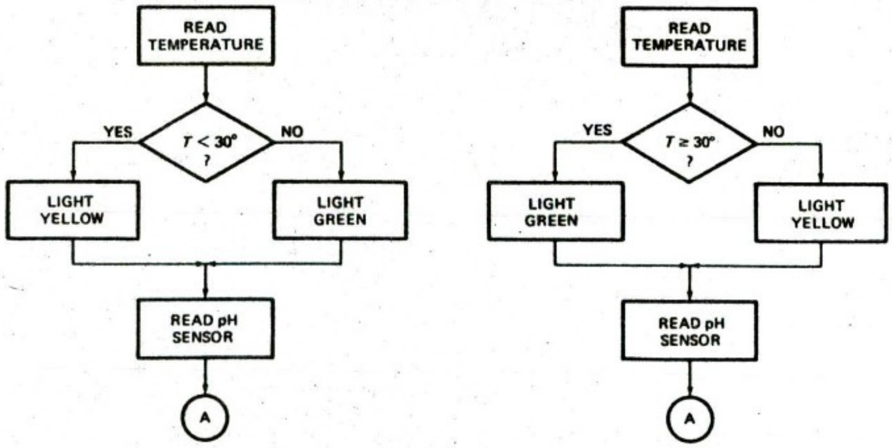
  IF Condition THEN

  Action

  ELSE

  Action.

Figure 3-3b shows the flowchart and pseudocode for this structure.




```
READ TEMPERATURE
IF TEMPERATURE < 30° THEN
    LIGHT YELLOW LAMP
ELSE
    LIGHT GREEN LAMP
READ pH SENSOR
```

(a)

```
READ TEMPERATURE
IF TEMPERATURE ≥ 30° THEN
    LIGHT GREEN LAMP
ELSE
    LIGHT YELLOW LAMP
READ pH SENSOR
```

(b)

FIGURE 4-12 Flowcharts and pseudocode for two ways of expressing algorithm for printed-circuit-board-making machine. (*a*) Temperature below 30° test. (*b*) Temperature above 30° test.

FIGURE 4-14 List file for printed-circuit-board-making machine program. (a) Below 30° version.

```
                                  ; 8086 PROGRAM   F4-14A.ASM
                        ;ABSTRACT : Program section for PC board making machine.
                                  ; This program section reads the temperature of a cleaning bath
                                  ; solution and lights one of two lamps according to the
                                  ; temperature read. If the temp <30°C, a yellow lamp will be
                                  ; turned on. If the temp is ≥30°C, a green lamp will be turned on.
                        ;REGISTERS: Uses CS, AL, DX
                        ;PORTS    : Uses FFF8H - temperature input
                                  ; FFFAH - lamp control output (yellow=bit 0, green=bit 1)

0000                    CODE     SEGMENT
                                 ASSUME CS:CODE
                        ;inftialize SDK-86 port FFFAH as output port, FFF8H as input port
0000  BA FFFE                    MOV DX, 0FFFEH      ; Point DX to port control register
0003  B0 99                      MOV AL, 99H         ; Load control word to initialize ports
0005  EE                         OUT DX, AL          ; Send control word to port control register

0006  BA FFF8                    MOV DX, 0FFF8H      ; Point DX at input port
0009  EC                         IN  AL, DX          ; Read temp from sensor on input port
000A  3C 1E                      CMP AL, 30          ; Compare temp with 30°C
000C  72 03                      JB  YELLOW          ; IF temp <30 THEN light yellow lamp
000E  EB 0A 90                   JMP GREEN           ; ELSE light green lamp
0011  B0 01             YELLOW:  MOV AL, 01H         ; Load code to light yellow lamp
0013  BA FFFA                    MOV DX, 0FFFAH      ; Point DX at output port
0016  EE                         OUT DX, AL          ; Send code to light yellow lamp
0017  EB 07 90                   JMP EXIT            ; Go to next mainline instruction
001A  B0 02             GREEN:   MOV AL, 02H         ; Load code to light green lamp
001C  BA FFFA                    MOV DX, 0FFFAH      ; Point DX at output port
001F  EE                         OUT DX, AL          ; Send code to light green lamp
0020  BA FFFC           EXIT:    MOV DX, 0FFFCH      ; Next mainline instruction
0023  EC                         IN  AL, DX          ; Read ph sensor
0024                    CODE     ENDS
                                 END
```

(a)

```
000A  3C 1E                CMP AL, 30      ; Compare temp with 30°C
000C  73 03                JAE GREEN       ; IF temp ≥30 THEN light green lamp
000E  EB 0A 90             JMP YELLOW      ; ELSE light yellow lamp
0011  B0 02        GREEN:  MOV AL, 02H     ; Load code to light green lamp
0013  BA FFFA              MOV DX, 0FFFAH  ; Point DX at output port
0016  EE                   OUT DX, AL      ; Send code to light green lamp
0017  EB 07 90             JMP EXIT        ; Go to next mainline instruction
001A  B0 01        YELLOW: MOV AL, 01H     ; Load code to light yellow lamp'
001C  BA FFFA              MOV DX, 0FFFAH  ; Point DX at output port
001F  EE                   OUT DX, AL      ; Send code to light yellow lamp
0020  BA FFFC      EXIT:   MOV DX, 0FFFCH  ; Next mainline instruction
0023  EC                   IN  AL, DX      ; Read ph sensor
0024               CODE    ENDS
                           END .
```

(b)

FIGURE 4-14 List file for printed-circuit-board-making machine program. (b) Program section for above 30° version.

If-then, if-then-else, multiple if-then-else programs

- **MULTIPLE IF-THEN-ELSE ASSEMBLY PROGRAMS**
  - Structure:

    IF Condition THEN

        Action.

    ELSE IF Condition THEN

        Action.

    ELSE

        Action.

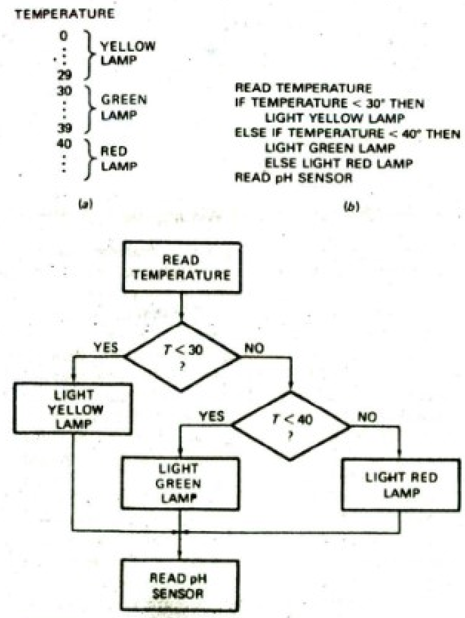


FIGURE 4-15 **Algorithm for three-lamp printed-circuit-board-making machine.** (*a*) Condition list. (*b*) Pseudocode. (*c*) Flowchart.

```
                                 ; 8086 PROGRAM F4-16.ASM
                       ;ABSTRACT  : This program section reads the temperature of a cleaning bath
                                  ; solution and lights one of three lamps according to the
                                  ; temperature read. If the temp < 30°C, a yellow lamp will be
                                  ; turned on. If the temp ≥ 30° and < 40°, a green lamp will be
                                  ; turned on. Temperatures ≥ 40° will turn on a red lamp.
                       ;REGISTERS : Uses CS, AL, DX
                       ;PORTS      : Uses FFF8H - temperature input
                                  ;      FFFAH - lamp control output, yellow=bit 0, green=bit 1, red=bit 2
0000                   CODE    SEGMENT
                               ASSUME CS:CODE
                       ;initialize port FFFAH for output and port FFF8H for input
0000  BA FFFE                  MOV DX, OFFFEH     ; Point DX to port control register
0003  B0 99                    MOV AL, 99H        ; Load control word to set up output port
0005  EE                       OUT DX, AL         ; Send control word to control register

0006  BA FFF8                  MOV DX, OFFF8H     ; Point DX at input port
0009  EC                       IN  AL, DX         ; Read temp from sensor on input port
000A  BA FFFA                  MOV DX, OFFFAH     ; Point DX at output port
0000  3C 1E                    CMP AL, 30         ; Compare temp with 30°C
000F  72 0A                    JB  YELLOW         ; IF temp < 30 THEN light yellow lamp
0011  3C 28                    CMP AL, 40         ; ELSE compare with 40°
0013  72 0C                    JB  GREEN          ;  IF temp < 40 THEN light green lamp
0015  B0 04            RED:    MOV AL, 04H        ;  ELSE temp ≥ 40 so light red lamp
0017  EE                       OUT DX, AL         ; Send code to light red lamp
0018  EB 0A 90                 JMP EXIT           ; Go to next mainline instruction
001B  B0 01            YELLOW: MOV AL, 01H        ; Load code to light yellow lamp
001D  EE                       OUT DX, AL         ; Send code to light yellow lamp
001E  EB 04 90                 JMP EXIT           ; Go to next mainline instruction
0021  B0 02            GREEN:  MOV AL, 02H        ; Load code to light green lamp
0023  EE                       OUT DX, AL         ; Send code to light green lamp
0024  BA FFFC          EXIT:   MOV DX, OFFFCH     ; Next mainline instruction
0027  EC                       IN  AL, DX         ; Read ph sensor
0028                   CODE    ENDS
                               END
```

FIGURE 4-16 List file for three-lamp printed-circuit-board-making machine program.

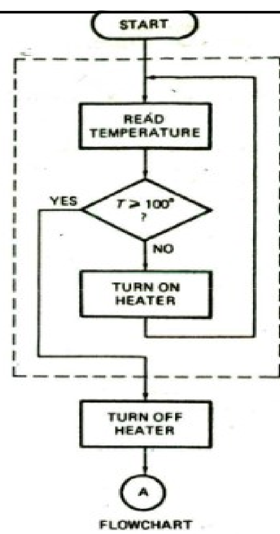
WHILE-DO Programs

- This structure is useful in executing a number of instructions repeatedly till some condition is satisfied.
- **WHILE-DO PROGRAMS**
  - Structure:

    WHILE some condition is present DO

    Action.

    Action.



FLOWCHART

(a)

READ TEMPERATURE
WHILE TEMPERATURE < 100° DO
TURN HEATER ON
TURN HEATER OFF

PSEUDOCODE

(b)

FIGURE 4-17 Flowchart and pseudocode for heater control program

```
                              ; 8086 PROGRAM   F4-18A.ASM
                   ;ABSTRACT  : Program turns heater off if temperature ≥ 100°C
                              ; and turns heater on if temperature < 100°C.
                   ;REGISTERS : Uses CS, DX, AL
                   ;PORTS     : Uses FFF8H - temperature data input
                              ;       FFFAH - MSB for heater control output, 0=off, 1=on
0000               CODE    SEGMENT
                           ASSUME  CS:CODE
                   ; Initialize port FFFAH for output, and port FFF8H for input
0000  BA FFFE                 MOV  DX, 0FFFEH    ; Point DX to port control register
0003  B0 99                   MOV  AL, 99H       ; Control word to set up output port
0005  EE                      OUT  DX, AL        ; Send control word to port

0006  BA FFF8      TEMP_IN:   MOV  DX, 0FFF8H    ; Point at input port
0009  EC                      IN   AL, DX        ; Input temperature data
000A  3C 64                   CMP  AL, 100       ; If temp ≥ 100 then
000C  73 08                   JAE  HEATER_OFF    ; turn heater off
000E  B0 80                   MOV  AL, 80H       ; else load code for heater on
0010  BA FFFA                 MOV  DX, 0FFFAH    ; Point DX to output port
0013  EE                      OUT  DX, AL        ; Turn heater on
0014  EB F0                   JMP  TEMP_IN       ; WHILE temp < 100 read temp again
0016  B0 00        HEATER_OFF:MOV  AL, 00        ; Load code for heater off
0018  BA FFFA                 MOV  DX, 0FFFAH    ; Point DX to output port
001B  EE                      OUT  DX, AL        ; Turn heater off
001C               CODE       ENDS
                              END
```

(*a*)

```
0006  BA FFF8      TEMP_IN:    MOV  DX, 0FFF8H   ; Point DX at input port
0009  EC                       IN   AL, DX       ; Read in temperature data
000A  3C 64                    CMP  AL, 100      ; If temp < 100° then
000C  72 03                    JB   HEATER_ON    ; turn heater on
000E  EB 09 90                 JMP  HEATER_OFF   ; else temp ≥100 so turn heater off
0011  B0 80        HEATER_ON:  MOV  AL, 80H      ; Load code for heater on
0013  BA FFFA                  MOV  DX, 0FFFAH   ; Point DX at output port
0016  EE                       OUT  DX, AL       ; Turn heater on
0017  EB ED                    JMP  TEMP_IN      ; WHILE temp < 100° read temp again
0019  B0 00        HEATER_OFF: MOV  AL, 00       ; Load code for heater off
001B  BA FFFA                  MOV  DX, 0FFFAH   ; Point DX at output port
001E  EE                       OUT  DX, AL       ; Turn heater off
001F               CODE        ENDS
                               END
```

(*b*)

FIGURE 4-18 List file for heater control program. (*a*) First approach. (*b*) Improved version of WHILE-DO section of program.

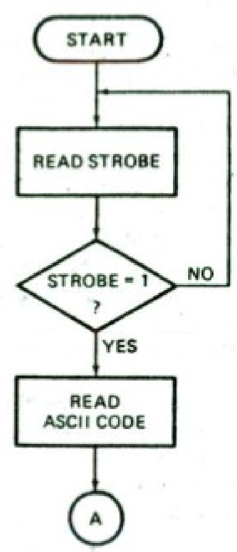
REPEAT-UNTIL Programs

- This structure can be used to loop through number of instructions until some condition is met. **If the condition in the until is true then the loop will break** and the immediate instruction will be executed.
- REPEAT-UNTIL PROGRAMS
  - Structure:

    REPEAT

    Action

    UNTIL some condition is present.



FLOWCHART

(a)

```
REPEAT
    READ KEYPRESSED STROBE
UNTIL STROBE = 1
READ ASCII CODE FOR KEY PRESSED
```

PSEUDOCODE

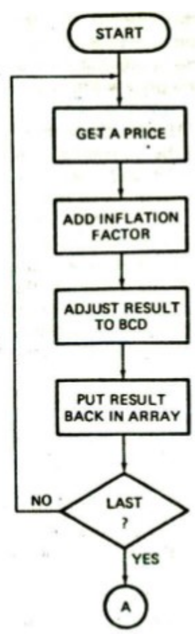
(b)

```
                                ; 8086 PROGRAM F4-20C.ASM
                      ;ABSTRACT  : Program to read ASCII code after a strobe signal
                                 ; is sent from a keyboard
                      ;REGISTERS : Uses CS, DX, AL
                      ;PORTS     : Uses FFFAH - strobe signal input on LSB
                                 ;      FFF8H - ASCII data input port

0000                  CODE         SEGMENT
                                   ASSUME CS:CODE
0000  BA FFFA                      MOV DX, OFFFAH    ; Point DX at strobe port
0003  EC              LOOK_AGAIN:  IN  AL, DX        ; Read keyboard strobe
0004  24 01                        AND AL, 01        ; Mask extra bits and set flags
0006  74 FB                        JZ  LOOK_AGAIN    ; If strobe is low then keep looking
0008  BA FFF8                      MOV DX, OFFF8H    ; else point DX at data port
000B  EC                           IN  AL, DX        ; Read in ASCII code
000C                  CODE         ENDS
                                   END
```

(c)

FIGURE 4-20 Flowchart, pseudocode, and assembly language for reading ASCII code when a strobe is present. (a) Flowchart. (b) Pseudocode. (c) List file of program.



FLOWCHART

(a)

REPEAT
    GET A PRICE FROM ARRAY
    ADD INFLATION FACTOR
    ADJUST RESULT TO CORRECT BCD
    PUT RESULT BACK IN ARRAY
UNTIL ALL PRICES ARE INFLATED

PSEUDOCODE

(b)

```
                                      ; 8086 PROGRAM  : F4-21C.ASM
                              ;ABSTRACT  : Program adds an inflation factor to a series of prices
                                         ; in memory. It copies the new price over the old price.
                              ;REGISTERS : Uses DS, CS, AX, BX, CX
                              ;PORTS     : None used

0000                          ARRAYS  SEGMENT
0000  20 28 15 26 19 27 16 +          COST     DB     20H, 28H, 15H, 26H, 19H, 27H, 16H, 29H
      29
0008  36 55 27 42 38 41 29 +          PRICES   DB     36H, 55H, 27H, 42H, 38H, 41H, 29H, 39H
      39
0010                          ARRAYS  ENDS

0000                          CODE    SEGMENT
                                      ASSUME CS:CODE, DS:ARRAYS
0000  B8 0000s                START:  MOV  AX, ARRAYS   ; Initialize data segment
0003  8E D8                           MOV  DS, AX       ; register
0005  8D 1E 0008r                     LEA  BX, PRICES   ; Initialize pointer
0009  B9 0008                         MOV  CX, 0008H    ; Initialize counter
000C  8A 07                   DO_NEXT: MOV  AL, [BX]    ; Copy a price to AL
000E  04 03                           ADD  AL, 03H      ; Add inflation factor
0010  27                              DAA               ; Make sure result is BCD
0011  88 07                           MOV  [BX], AL     ; Copy result back to memory
0013  43                              INC  BX           ; Point to next price
0014  49                              DEC  CX           ; Decrement counter
0015  75 F5                           JNZ  DO_NEXT      ; If not last, go get next
0017                          CODE    ENDS
                                      END  START
```

(c)

FIGURE 4-21 Adding a constant to a series of values in memory. (a) Flowchart. (b) Pseudocode. (c) List file of program.

```
                                      ; 8086 PROGRAM F4-23.ASM
                              ;ABSTRACT  : Program adds a profit factor to each element in a
                                         ; COST array and puts the result in an PRICES array.
                              ;REGISTERS : Uses DS, CS, AX, BX, CX
                              ;PORTS     : None used

      = 0015                  PROFIT   EQU     15H        ; profit = 15 cents
0000                          ARRAYS   SEGMENT
0000  20 28 15 26 19 27 16 +           COST     DB  20H, 28H, 15H, 26H, 19H, 27H, 16H, 29H
      29
0008  08*(00)                          PRICES   DB  8 DUP(0)
0010                          ARRAYS   ENDS

0000                          CODE    SEGMENT
                                      ASSUME  CS:CODE, DS:ARRAYS
0000  B8 0000s                START:  MOV  AX, ARRAYS      ; Initialize data segment
0003  8E D8                           MOV  DS, AX          ; register
0005  B9 0008                         MOV  CX, 0008H       ; Initialize counter
0008  BB 0000                         MOV  BX, 0000H       ; Initialize pointer
000B  8A 87 0000r             DO_NEXT: MOV  AL, COST[BX]   ; Get element [BX] from COST
000F  04 15                           ADD  AL, PROFIT      ; Add the profit to value
0011  27                              DAA                  ; Decimal adjust result
0012  88 87 0008r                     MOV  PRICES[BX], AL  ; Store result in PRICES at [BX]
0016  43                              INC  BX              ; Point to next element in arrays
0017  49                              DEC  CX              ; Decrement the counter
0018  75 F1                           JNZ  DO_NEXT         ; If not last element, do again
001A                          CODE    ENDS
                                      END  START
```

FIGURE 4-23 List file of "price-calculating" program.

8086 Addressing Modes

- **Single Index:** Contents of BX, BP, SI, DI is added directly to displacement to generate effective address.
- **Double Index:** Contents of BX or BP register is first added with SI or DI and then the result is added to Displacement to generate the effective address.

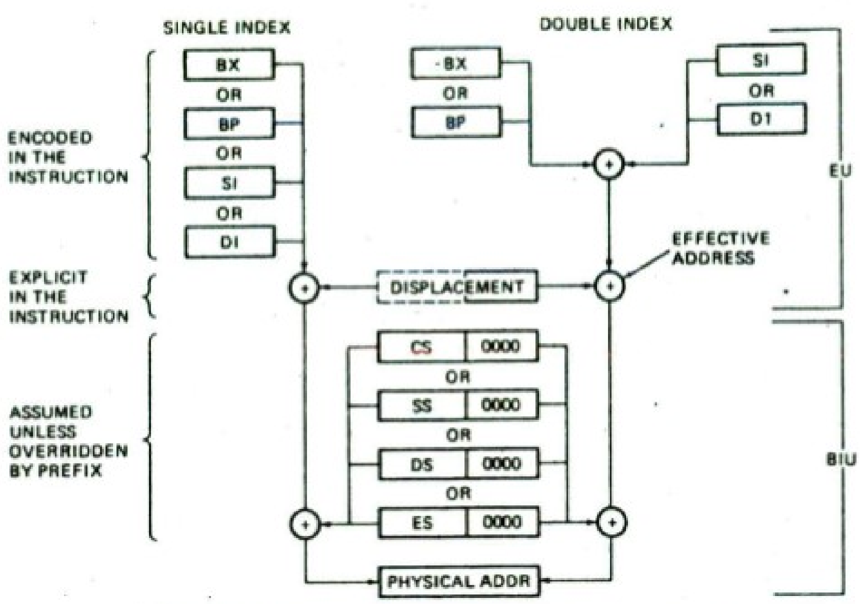


FIGURE 4-24 Summary of 8086 addressing modes.

The 8086 Loop instructions

- These are the instructions that are used to do some sequence specific number of times.
- They are basically conditional jumps which have format LOOP Label.
- LOOP instructions decrements the CX register but do not affect the ZF.
- The LOOPNE/LOOPNZ Label instruction decrements the CX by 1 and if CX != 0 and ZF = 0 the instruction will cause a jump on the specified label.

LOOP	Loop until CX = 0
LOOPE/LOOPZ	Loop if zero flag set and CX ≠ 0
LOOPNE/LOOPNZ	Loop if zero flag not set and CX ≠ 0
JCXZ	Jump if CX = 0

FIGURE 4-26 8086 LOOP instructions.

Instruction Timing and Delay Loops(contd.)

- The rate at which 8086 instructions are executed is determined by a crystal-controlled clock with a frequency of a few megahertz.
- Each instruction takes a certain number of clock cycles to execute. The **MOV** register, register instruction, for example, requires **2 clock cycles** to execute, and the **DAA** instruction requires **4 clock cycles**. The **JNZ** instruction requires **16 clock cycles** if it does the Jump, but it requires only 4 clock cycles tilt doesn't do the Jump.
- With this, you can calculate how long it takes to execute an instruction or series of instructions.
- For example, if you are running an 8086 with a 5-MHz clock, then each clock cycle takes (5 MHz) or 0.2 µs.
- An instruction which takes **4 clock cycles**, then, will take 4 clock cycles x 0.2 µs/clock cycle or 0.8 µs to execute.

Instruction Timing and Delay Loops(contd.)

- Program loops introduce the delay between instructions.
- Calculate number of clock cycles to produce the delay. E.g. 8086 with **5 MHz clock** the time for one clock cycle is 1/5 micro seconds or **0.2 micro seconds.**
- Next determine how many clock cycles needed in the loop.
- Now, suppose that you want to create a delay of 1 ms or 1000 µs with a delay loop.
- If you divide the 1000 µs desired by the 0.2 µs per clock cycle, you get the number of clock cycles required to produce the desired delay.
- For this example you need a total of 1000/0.2 or 5000 processor clock cycles to produce the desired delay. We will call this **number C_T.**

Instruction Timing and Delay Loops(contd.)

- The next step is to write the number of clock cycles required for each instruction next to that instruction as shown in Figure 4-27a.
- The number of clock cycles for the instructions **which execute Only Once** will only contribute to the total once. Instructions which only enter, the calculation once are often called **overhead**. We will represent the number of cycles of overhead with the symbol C_0 .
- Next you determine how many clock cycles required for the loop. The two **NOPs** in the loop require a total of **6 clock cycles.**
- The LOOP instruction requires 17 clock cycles If it does the Jump back to KILL_TIME, but it requires only 5 clock cycles when it exits the loop.

Instruction Timing and Delay Loops(contd.)

```
                                     ;   Clock Cycles
            MOV CX, N                ;     4     = Co
KILL_TIME:  NOP                      ;     3
            NOP                      ;     3     = CL
            LOOP KILL_TIME           ;  17 or 5
```

(a)

$$C_T = C_O + N\,(C_L) - 12$$

$$N = \frac{C_T - C_O + 12}{C_L} = \frac{5000 - 4 + 12}{23} = 218$$

(b)

FIGURE 4-27 Delay loop program and calculations. (a) Program. (b) Calculations.

Instruction Timing and Delay Loops

Note about using delay loops for timing:

- The BIU and the EU are asynchronous, so **for some instruction sequences an extra clock cycle may be required.**
- **The no of clock cycles required to read a word from memory** or write a word on memory **depends on whether the first byte of the word is at even address or at odd address.**
- The no of clock cycles required to read a byte from memory or write a byte on memory depends **on the addressing mode used to address the byte.**
- If a given microcomputer system is designed to insert **WAIT** states during each memory access, this will **increase the no of clock cycles required** for each memory access.

- **Strings and Procedures**
  - The 8086 string instructions
  - Writing and using procedures
- **Assembler Directives**

8086 String Instructions

- **A string is a series of bytes or words stored** in successive memory locations. Often a string consists of **a series of ASCII character codes.**
- When you use a **word processor or text editor program**, you are actually creating a string of this sort as you type in **a series of characters.**
- One important feature of a **word processor is the ability to move a sentence or group of sentences from one place in the text to another.**
- Doing this involves moving a siring of ASCII characters from one place in memory to another.
- The 8086 Move String instruction, **MOVS** allows you to do operations such as this very easily.

8086 String Instructions

- Another important feature of most **word processors is the ability to search through the text looking for a given word or phrase.**
- The 8086 Compare String instruction, **CMPS,** can be used to do operations of this type. In a similar manner, the 8086 **SCAS** instruction can be used to search a string to see whether it contains a specified character.

8086 String Instructions- Moving a String

- Suppose that you have a string of ASCII characters in successive memory locations in the data segment and you want to move the string to some new sequence of locations in the data segment.
- To help you visualize this, take a look at the strings we Set up in the data segment in **Figure 5.1b, p. 96.** to test our program.
- The statement TEST_MESS DB 'TIS TIME FOR A NEW HOME' sets side 23 bytes of memory and gives the first memory location the name TEST_MESS.

```
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
```

ALP with **MOVS**

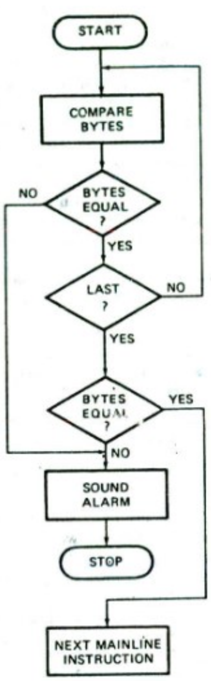
```
                                            ; 8086 PROGRAM F5-01.ASM
                                   ;ABSTRACT : Program moves a string from the location TEST_MESS
                                            ; to the location NEW_LOC.
                                   ;REGISTERS ; Uses CS, DS, ES, SI, DI, AX, CX
                                   ;PORTS    ; None used

0000                               DATA  SEGMENT
0000  54 49 53 20 54 49 4D +             TEST_MESS DB 'TIS TIME FOR A NEW HOME' ; String to move
      45 20 46 4F 52 20 41 +
      20 4E 45 57 20 48 4F +
      4D 45
0017  64*(??)                                  DB  100 DUP(?)          ; Stationary block of text
007B  17*(00)                            NEW_LOC  DB  23. DUP(0)          ; String destination
0092                               DATA  ENDS

0000                               CODE  SEGMENT
                                         ASSUME CS:CODE, DS:DATA, ES:DATA

0000  B8 0000s                     START:MOV AX, DATA        ; Initialize data segment register
0003  8E D8                              MOV DS, AX
0005  8E C0                              MOV ES, AX          ; Initialize extra segment register
0007  8D 36 0000r                        LEA SI, TEST_MESS   ; Point SI at source string
000B  8D 3E 007Br                        LEA DI, NEW_LOC     ; Point DI at destination location
000F  B9 0017                            MOV CX, 23          ; Use CX register as counter
0012  FC                                 CLD                 ; Clear direction flag so pointers autoincrement
                                                             ; after each string element is moved
0013  F3> A4                       REP   MOVSB               ; Move string bytes until all moved

0015                               CODE ENDS
                                         END START
```




```
REPEAT
     COMPARE SOURCE BYTE WITH DESTINATION BYTE
UNTIL (BYTES NOT EQUAL) OR (END OF STRING)
IF BYTES NOT EQUAL THEN
     SOUND ALARM
     STOP
ELSE DO NEXT MAINLINE INSTRUCTION
```

(*b*)

```
INITIALIZE PORT DEVICE FOR OUTPUT
INITIALIZE SOURCE POINTER – SI
INITIALIZE 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
```

(*c*)

FIGURE 5-2 Flowchart and pseudocode for comparing strings program. (*a*) Flowchart. (*b*) Initial pseudocode. (*c*) Expanded pseudocode.

ALP with **CMPS**

```
                                      ; 8086 PROGRAM F5-03.ASM
                           ;ABSTRACT  : This program inputs a password and sounds an alarm
                                      ; if the password is incorrect
                           ;REGISTERS : Uses CS, DS, ES, AX, DX, CX, SI, DI
                           ;PORTS     : Uses FFFAH - Port 2B on SDK-86,for alarm output

0000                       DATA  SEGMENT
0000  46 41 49 4C 53 41 46 +       PASSWORD    DB   'FAILSAFE'      ; Password
      45
      = 0008                       STR_LENGTH  EQU  ($ - PASSWORD) ; Compute length of string
0008  08*(00)                      INPUT_WORD  DB   8 DUP(0)       ; Space for user password input
0010                       DATA  ENDS

0000                       CODE  SEGMENT
                                 ASSUME CS:CODE, DS:DATA, ES:DATA
0000  B8 0000s                         MOV  AX, DATA
0003  8E D8                            MOV  DS, AX            ; Initialize data segment register
0005  8E C0                            MOV  ES, AX            ; Initialize extra segment register
0007  BA FFFE                          MOV  DX, OFFFEH        ; These next three instructions
000A  B0 99                            MOV  AL, 99H           ; set up an output port on
000C  EE                               OUT  DX, AL            ; the SDK-86 board
000D  8D 36 0000r                      LEA  SI, PASSWORD      ; Load source pointer
0011  8D 3E 0008r                      LEA  DI, INPUT_WORD    ; Load destination pointer
0015  B9 0008                          MOV  CX, STR_LENGTH    ; Load counter with password length
0018  FC                               CLD                    ; Increment DI & SI
0019  F3> A6               REPE        CMPSB                  ; Compare the two string bytes
001B  75 03                            JNE  SOUND_ALARM       ; If not equal, sound alarm
001D  EB 08 90                         JMP  OK                ;  else continue
0020  B0 01                SOUND_ALARM:MOV  AL, 01            ; To sound alarm, send a 1
0022  BA FFFA                          MOV  DX, OFFFAH        ; to the output port whose
0025  EE                               OUT  DX, AL            ; address is in DX
0026  F4                               HLT                    ; and HALT.
0027  90                   OK:         NOP                    ; Program continues if password is OK
0028                       CODE        ENDS
                                       END
```

FIGURE 5-3 Assembly language program for comparing strings.

writing and using Procedures

- Often when writing programs we will find that we need to **use a particular sequence of instructions at several different points in a program.**
- 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.**
- Each time you need to execute the sequence of instructions contained in the procedure, you use the **CALL** instruction to send the 8086 to the starting address of the procedure in memory.

writing and using Procedures

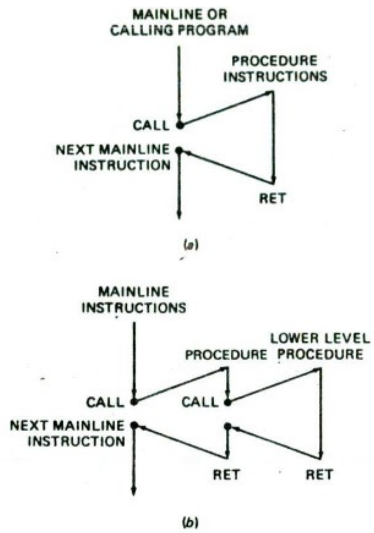


FIGURE 5-4 Program flow to and from procedures. (a) Single procedures. (b) Nested procedures.

The 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.
- The next instruction fetched will be the first instruction 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 **RET** instruction does this by **loading the Instruction pointer** and If necessary, the code segment register with the address of the next instruction after the CALL instruction.
- The 8086 **CALL Instruction performs two 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.

The 8086 CALL and RET Instructions

- 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.
- **The 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.
- This **function of the CALL instruction is very similar to the operation of the JMP instructions .**

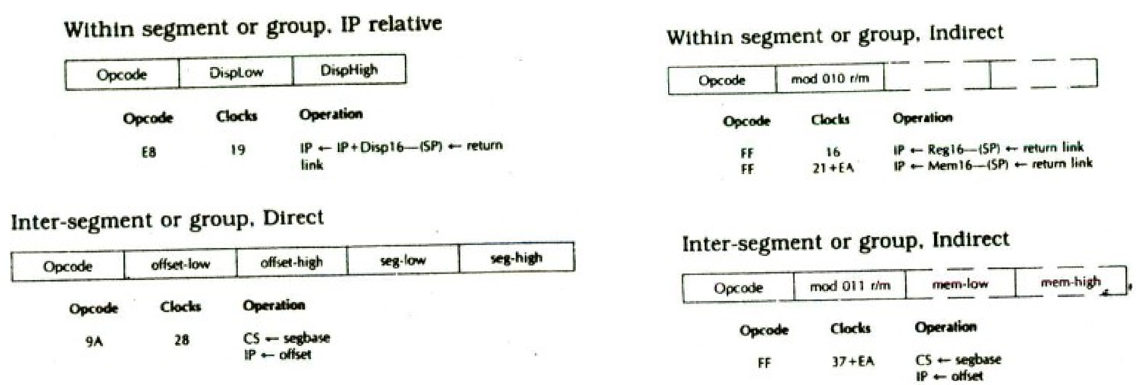
The 8086 CALL and RET Instructions

- Similar to JMP instruction here also we have

➢ **DIRECT WITHIN-SEGMENT NEAR CALL**

➢ **THE INDIRECT WITHIN-SEGMENT NEAR CALL**

➢ **THE DIRECT/INDIRECT INTERSEGMENT FAR CALL**



The 8086 CALL and RET Instructions

RET = Return from Subroutine

Opcode

Opcode	Clocks	Operation
C3	8	intra-segment return
CB	18	inter-segment return

Return and add constant to SP

Opcode	DataL	DataH

Opcode	Clocks	Operation
C2	12	intra-segment ret and add
CA	17	inter-segment ret and add

The 8086 Stack

- The **stack is a section of memory you set aside for storing return addresses.**
- The stack is also **used to save the contents of registers** for the calling program while a procedure executes.
- A third use of the stack is **to hold data or addresses** that will be acted upon by a procedure.
- The 8086 lets you set aside up to an entire **64-Kbyte segment of memory as a stack.**
- The **stack pointer register is used to hold the offset** of the last word written on the stack. The 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.**

The 8086 Stack

- An important point about the operation of the stack is that the **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.

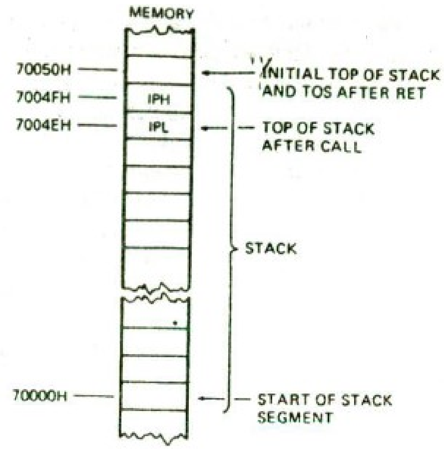


FIGURE 5-7 Stack diagram showing how the return address is pushed onto the stack by CALL.

Using PUSH and POP to Save Register Contents

- It is very common to want to use registers both in the mainline program and in a procedure without the two uses Interfering with each other.
- The PUSH and POP instructions make this very easy to do.
- 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.
- This will decrement the stack pointer by 2 and copy the Contents of the CX register to the stack where the stack pointer now points
- The **POP register/memory instruction** copies a word from the top of the stack to the specified 16-bit register or memory location and **increments the stack pointer by 2.**
- This will copy a word from the top of the stack to the CX register and increment the stack pointer by 2.
- After a POP, the stack pointer will point to the next word on the stack.

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.
- **These addresses or data values passed back** and forth between the mainline and the procedure are commonly **called parameters.**
- The four 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

Passing Parameters to and from Procedures

PASSING PARAMETERS IN REGISTERS

```
4596 = (4 x 1000) + (5 x 100) + (9 x 10) + (6 x 1)

   1 = 0001H  therefore    6 = 6 x 0001H = 0006H
  10 = 000AH  therefore   90 = 9 x 000AH = 005AH
 100 = 0064H  therefore  500 = 5 x 0064H = 01F4H
1000 = 03E8H  therefore 4000 = 4 x 03E8H = 0FA0H
                        ----               -----
                        4596             = 11F4H
```

FIGURE 5-13 BCD-to-binary algorithm.

•The units position has a value of 1 in hex, so multiplying this by 6 units gives **0006H.**

•The tens position has a value of 1010 binary. or OAI-l. Multiplying this value by 9. the number of tens, gives **005AH.**

•The value of the hundreds position in the BCD number is 01100100 binary, or 64H. When you multiply this value by 5, the number of hundreds, you get **01F4H**.

•When you multiply the hex value of the thousands position, O3E8H, by 4 (the number of thousands), you get **0FA0H.**

•Adding up the results for the four digits gives **11F4H** or 0001000111110100, which is the binary equivalent of **4596 BCD**

Passing Parameters to and from Procedures

PASSING PARAMETERS IN REGISTERS

- The algorithm for this program is the simple sequence of operations
  - Separate nibbles
  - Save lower nibble (don't need to multiply by 1)
  - Multiply upper nibble by OAH
  - Add lower nibble to result of multiplication
- Figure 5-14, p. 110, shows our first version of a procedure to Convert a two-digit packed BCE) number to its binary equivalent. The BCD number is copied from memory to the AL register and then passed to the procedure in the AL register.
- We start the procedure by pushing the flag register and the other registers we use in the procedure.

Passing Parameters to and from Procedures

PASSING PARAMETERS IN REGISTERS

- **Example Program FIGURE 5-14 Page No 110**

Passing Parameters to and from Procedures

PASSING PARAMETERS IN MEMORY

- In this procedure we first **push** the flags and all the registers used in the procedure.
- We then copy the BCD number into AL with the MOV AL, BCD_INPUT Instruction.
- From here on, the procedure is the same as the previous version until we reach the point where we want to pass the binary result back to the calling program.
- Here we use the MOV BINVALUE, AL instruction to copy the result directly to the dedicated memory location we set aside for it.
- To complete the procedure, we **pop** the flags and registers and return to the main program.

Passing Parameters to and from Procedures

PASSING PARAMETERS IN MEMORY

- **Example Program FIGURE 5-15 Page No 111**

Passing Parameters to and from Procedures

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.
- In the main program, before we call the procedure, we use the MOV SI,OFFSET BCD_INPUT instruction to set up the SI register as a pointer to the memory location BCD_INPUT.
- We also use the MOV DI,OFFSET BIN_VALUE Instruction to set up the DI register as a pointer to the memory location named BIN_VALUE.
- In the procedure, the MOV AL,[SI] Instruction will copy the byte pointed to by SI Into AL.
- Likewise, the MOV [DI],AL Instruction later In the procedure will copy the byte from AL to the memory location pointed to by DI.

Passing Parameters to and from Procedures

PASSING PARAMETERS USING POINTERS

- **Example Program   FIGURE 5-16 Page No 112**

Passing Parameters to and from Procedures

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.
- 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.
- A simple example will best show you how this works.
- Figure 5-17, p. 114, shows a version of our BCD_BIN procedure which uses the stack for passing the BCD number to the procedure and for passing the binary value back to the calling program.
- To save space here, we assume that previous instructions in the mainline program set up a stack segment. Initialized the stack segment register, and initialized the stack pointer.

Passing Parameters to and from Procedures

PASSING PARAMETERS ON STACK

- **Example Program FIGURE 5-16 Page No 114**

Reentrant and Recursive Procedures

REENTRANT PROCEDURES

- The 8086 has a signal Input which allows a signal from some external device to interrupt the normal program execution sequence and call a specified procedure.
- In our electronics factory, for example. a temperature sensor in a flow-solder machine could be connected to the interrupt input.
- If the temperature gets too high. the sensor sends an interrupting signal to the 8086. The 8086 will then stop whatever it is doing and go to a procedure which takes whatever steps are necessary to cool down the solder bath.
- This procedure is called an **Interrupt service procedure.**
- When the interrupt occurs, execution goes to the Interrupt service procedure. The interrupt service procedure then calls the multiply procedure when it needs it.
- The **RET Instruction** at the end of the multiply procedure returns execution to the interrupt service procedure. A special return instruction at the end of the interrupt service procedure returns execution to the multiply procedure where it was executing when the-interrupt occurred

Reentrant and Recursive Procedures

REENTRANT PROCEDURES

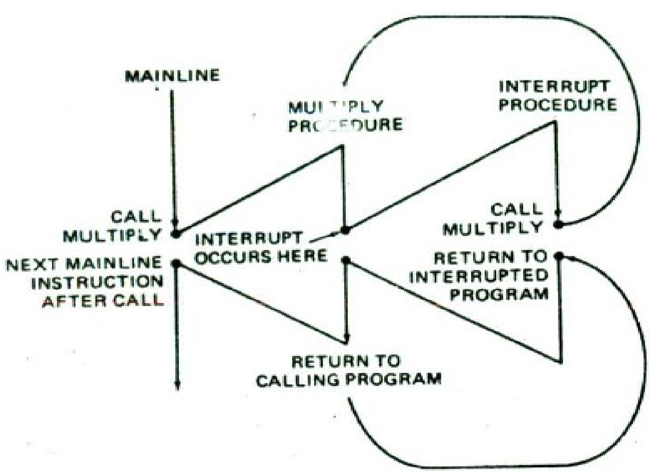


FIGURE 5-20 Program execution flow for reentrant procedure.

Reentrant and Recursive Procedures

RECURSIVE PROCEDURES

- A Recursive procedure is a procedure **which calls itself.**
- This seems simple enough, but the question you may be thinking is, "Why would we want a procedure to call itself?'
- The answer is that certain types of problems, such as **choosing the next move in a computer chess program, can best be solved with a recursive procedure.**
- Recursive procedures are often **used to work with complex data structures called trees.**
- We usually write recursive procedures in a high-level language such as C or Pascal, except in those cases where we need the speed gained by writing in assembly language.

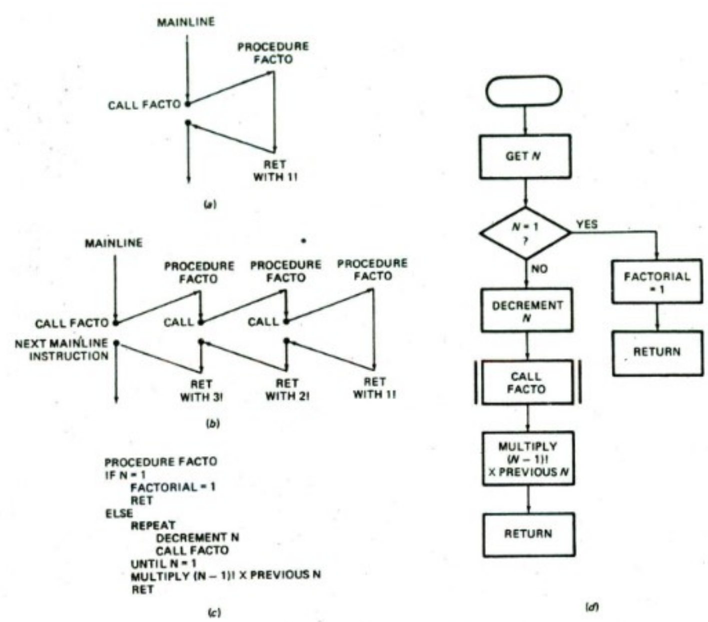


FIGURE 5-21 Algorithm for program to compute factorial for a number N between 1 and 8. (a) Flow diagram for N = 1. (b) Flow diagram for N = 3. (c) Pseudocode. (d) Flowchart.

Assembler Directives

▪ASSUME ▪DB—Define Byte ▪DD—Define Doubleword ▪DQ.—Define Quadword ▪DT—Define Ten Bytes ▪DW—Define Word ▪END—End Program ▪ENDP—End Procedure ▪EQU—Equate ▪EVEN—Align on Even Memory Address ▪EXTRN ▪GLOBAL—Declare Symbols as PUBLIC or EXTRN ▪GROUP—Group-Related Segments	▪INCLUDE—Include Source Code from File ▪LABEL ▪LENGTH—Not Implemented in IBM MASM ▪NAME ▪OFFSFT ▪ORG—Originate ▪PROC—Procedure ▪PTR—Pointer ▪Public ▪SEGMENT ▪SHORT ▪TYPE

UNIT-2 ENDS

THANK YOU