MPF-I

**EXPERIMENT MANUAL
(SOFTWARE/HARDWARE)**

MPF-IP

EXPERIMENT MANUAL
(SOFTWARE/HARDWARE)

MPF-IP EXPERIMENT MANUAL

TABLE OF CONTENTS

Introduction to Designing Microcomputer Programs

A computer program is an organized series of instructions. The central processing unit will perform a series of logical actions to obtain the desired result.

Before a program is executed by CPU it must be stored in memory in binary form. This type of program is called a "machine language program". This is the only type of language the computer understands. The machine language program is usually represented by Hexadecimal digits. For example, the 8-bit instruction 1010 1111B (B represents binary) in the Z80 CPU it can be replaced by 0AFH (H represents hexadecimal). Interpreting a machine language program is extremely difficult and time consuming for the user. the microprocessor manufacturer divides the CPU instructions into several categories according to their functions. The CPU instructions and registers are usually represented by symbols called "mnemonics". For example, the Z80 CPU instruction 70H can be represented by the mnemonic code LD A,L (Load Data into register a from register L). A program written in mnemonic codes is called an "assembly language program." Before an assembly language program can be executed by the CPU, it must be translated into machine language by a special software program called an "Assembler".

Normally a program is written in assembly language. The main advantage of assembly language program over machine language programming is that assembly language programming is much faster to code, the mnemonics makes it much easier for the user to remember the instruction set, and normally the assembler will contain a self-diagnostic package for debugging programs. The main disadvantage of assembly language programs is that it requires an assembler and microcomputer development system. these two items are very costly. With the MPF-IP microcomputer the user has to translate assembly programs into machine level programs by hand before executing programs.

A. Problem Analysis

The software program of a simple problem may be easily designed with a well-defined flowchart. It may also be obtained by revising some existing programs or combining some simple routines. The design of a more complicated programs, such as monitor programs, system control programs or a special purpose program, are usually started after some detailed analysis of the problem has been made. Problem analysis and solution requires a good understanding of the following:

(1)   Characteristic and requirements of the problem
(2)   Conditions which are known
(3)   Input information format and how it is converted
(4)   Output data format and how it is converted
(5)   Type of data and how precise it is
(6)   Execution speed required
(7)   CPU instructions and performance
(8)   Memory size
(9)   The possibility that the problem can be solved
(10)   Methods to solve the problem
(11)   Evaluation of the program
(12)   How the resultant program will be disposed
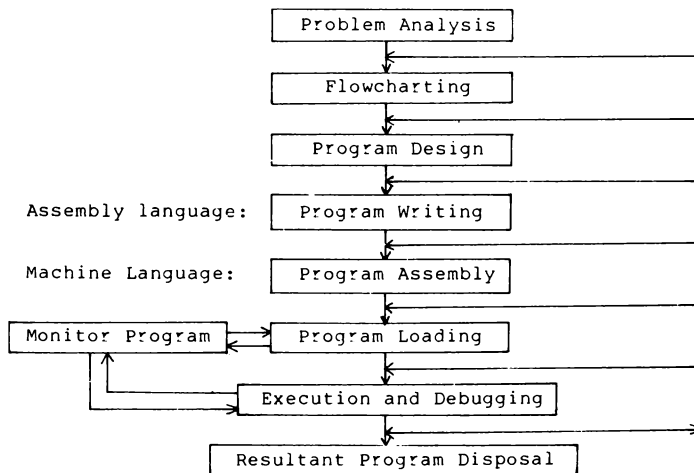
Figure   A-1

B. Flowchart

   A flowchart can be used to indicate the behavior of algorithms by
suitable graphs. Once the complete flowchart has been completed, a
full picture of the programmer's thought processes in reaching a
solution to the problem may be followed.   Flowcharts are especially
important in program-debugging.   It is an important part of the
finished program. It may help other people to understand the exact
algorithm used by the programmer.

   Two levels of flowcharts are often desirable:

System flowchart -- showing the general flow of the program

Detailed flowchart -- providing details that are of interest mainly
                      to the programmer.

Usually, a complicated program is introduced using a system flowchart
outlining the program, and then a detailed flowchart is presented.
The advantage of a flowchart is that it emphasizes the sequential
nature of steps by using arrows pointing from each step to its
successor. Various symbols are used to indicate the operation that
is to be performed at each step.   Figure 2-A-2 gives some standard
symbols used in flowcharts:

Process	Div 16
Decision	HL ← 0 A ← 16H
Manual Operation	DV0
Input/Output	RL E,D HL ← HL+HL HL ← HL−BC
Connector	Yes — NC — No
Terminal Interrupt	HL ← HL+BC
Label	DV1
Flow Line	CCF DEC A

Process

Decision

Manual
Operation

Input/Output

Connector

Terminal
Interrupt

Label

Flow Line

Div 16

HL ← 0
A ← 16H

DV0

RL E,D
HL ← HL+HL
HL ← HL−BC

Yes   NC   No

HL ← HL+BC

DV1

CCF
DEC A

NZ — YES

No

RL E,D

RET

Figure   A-2

6A
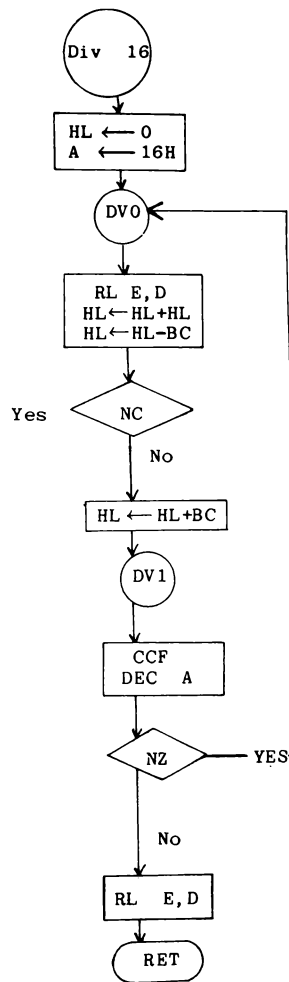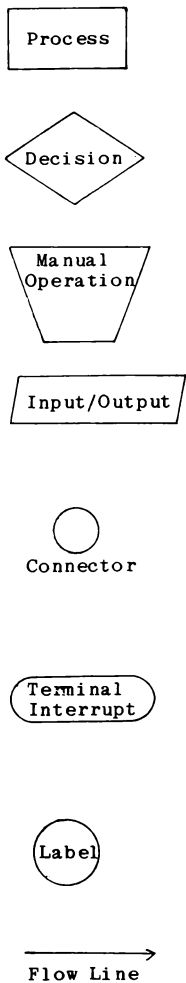
C. Program Design

There are many types of programs. Programs for mathematical equations, conversion of input and output signals, coding and decoding of the program data, peripheral device drives, etc. are example of simple programs. Assembler, monitor and system control programs or special purpose applications are examples of more complicated programs. The following items are usually considered in program design:

(1) Acquisition of input signals or data
(2) Generation or conversion of output signals and data
(3) Logical analysis and calculations in the main program
(4) Relation between the main program and subroutines
(5) Use of internal registers
(6) Memory allocation of the main program
(7) Memory allocation of subroutines
(8) Memory allocation of data tables and indexed addressing methods
(9) System initialization and constants in the program
(10) Definition of the variables in the program
(11) Consideration of timing sequences and program execution speed
(12) Limitations of memory size
(13) Length and precision of data
(14) Availability of documents and references
(15) Other special items

D. Program Writing

In this book, the programs are written mainly in assembly language. Here only the format of the assembly language program is given.

A statement in the program is composed of four parts : Label, Opcode, Operand and Comment. An example is shown below

```
LABEL   OPCODE & OPERAND           COMMENT
-----   -------------------        ---------


DTB4    LD      B,16
DB3     SRL     H
        RR      L
        RR      D
        RR      E                  ; ROTATE HL DE RIGHT
        LD      A,H
        CALL    DB1
        LD      H,A                ; CORRECT H
        LD      A,L
        CALL    DB4
        LD      L,A                ; BINARY CORRECT L
        DJNZ    DB3
        RET

   BINARY CORRECT ROUTINE
DB4     BIT     7,A
        JR      Z,DB1              ; IF BIT 7 OF A = 1, SUB FROM 30H
        SUB     30H
DB1     BIT     3,A
        JR      Z,DB2              ; IF BIT 3 OF A = 1, SUB FROM 03H
        SUB     3
DB2     RET
```

Sometimes, a program statement without a comment is not easy to understand. The comments in the statements are very important especially for a complicated program. Statements with a label and comment field are more convenient for calling and debugging.

E. Program Assembly

Using the resident assembler in a microcomputer system is an effective way to assemble the source program. However, a beginner or a proram designer not familiar with the microcomputer development system must assemble his/her program by hand. The usual procedure for hand assembly is:

(1) Translate each instruction (mnemonic) into the machine code by looking it up in the conversion table. The comment field of each statement is ignored.

(2) After deciding the starting address of the program. Assign an appropriate address to the first byte of each instruction. The exact number of bytes needed must be reserved including space for instructions such as JR, DJNZ, and destination addresses of instructions JP, CALL, etc.

(3) Calculate the relative displacement and put it in the assembled program. A simple formula for calculating the relative displacement is:

displacement   = (destination address) − (next instruction address)

If the calculated result is positive, then it is the desired value. If the calculated result is negative, then subtract the result from 100H (i.e. take its 2's complement) and the final result is taken as the operand of this instruction. For instance, in the program listed above, the instruction DJNZ DB3 at address 0014H is first translated into 10xx and then the xx value is calculated.

    xx = 0002H (destination address) − 1016H (next instruction's address)
        = −14H (negative value)
    xx = 100H − 14H = 0ECH

Therefore, the instruction DJNZ DB3 must be translated into 10EC. In addition, the instruction JR Z, DB 1 at address 0019H is first translated into 28xx, and then the xx value is calculated.

    xx = 001DH (destination address) − 001BH (next instruction's address)
        = 2 H

The instruction JR Z, DB 1 must be translated into 2802.

The translated machine language is given below:

```
        Machine
Address Language Label  Opcode & Operand Comment
------- -------- -----  ---------------- -------
                 ; ** 4 DIGIT BCD TO BINARY CONVERTION ROUTINE **
                 ; EXTRY : BCD DATA IN HL
                 ; EXIT : BINARY DATA IN DE
                 ; REGISTER CHANGED : AF BC DE HL

0000    0610    DTB4    LD    B,16    ; B = BIT COUNT
0002    CB3C    DB3     SRL   H
```

```
0004    CB1D            RR      L
0006    CB1A            RR      D
0008    CB1B            RR      E       ; ROTATE HL DE RIGHT
000A    7C              LD      A,H
000B    CD1D00          CALL    DB1
000E    67              LD      H,A     ; CORRECT H
000F    7D              LD      A,L
0010    CD1700          CALL    DB4
0013    6F              LD      L,A     ; BINARY CORRECT L
0014    10EC            DJNZ    DB3
0016    C9              RET
                ;
                ; BINARY CORRECT ROUNTINE
0017    CB7F    DB4     BIT     7,A
0019    2802            JR      Z,DB1   ; IF BIT 7 OF A = 1, SUB FROM 30H
001B    D630            SUB     30H
001D    CB5F    DB1     BIT     3,A
001F    2802            JR      Z,DB2   ; IF BIT 3 OF A = 1, SUB FROM 03H
0021    D603            SUB     3
0023    C9      DB2     RET
```

Experiment 1
Data-Transfer Experiment

Purposes:

1. To familiarize the user with the function of data-transfer instruction
2. To practise setting the initial value of data
3. To practise assembling, loading and executing a program

Time required: 4 hours

I. Theorectical Background:

1. Most of the data-transfer operation is accomplished by using LD (load) instructions. Data can be transferred in group of 8 bits or 16 bits. Also, instructions such as EX, EXX, PUSH and POP can be used to transfer 16-bit data. Instructions such as LDI and LDIR can be used to transfer blocks of data by moving a series of bytes.

2. A LD instruction must have two operands. The first operand represents the location where data will be stored (register or memory section). This is called its "destination". The second operand represents the original location of the data to be transferred. This is called the "source". For instance, LD A,B indicates that data in register B will be transferred to register A. Register A is the "destination" and Register B is the "source".

3. Data transfer instructions can be used in the following ways:

```
(1) register <- register   e.g.  LD   A,B     ; LD HL,BC
(2) register <- memory     e.g.  LD   A,(HL)  ; POP AF
(3) register <- immediate data
                           e.g.  LD   A,25H   ; LD HL,125AH
(4) memory <- register     e.g.  LD   (HL),A  ; PUSH BC
(5) memory <- memory       e.g.  LDD          ; LDIR
(6) memory <- immediate data
                           e.g.  LD   (HL),5BH
```

II. Experiment 1-1

Write an assembly language program to set the contents of the registers as follows : A=0, B=1, C=2, D=3, E=4, H=5, L=6 (use 8-bit LD instruction to transfer one byte of data each time).

Step 1   Write the assembly language program in the following blank
         form.   The last instruction is RST 38H which returns control
         of the MPF-IP to the monitor program after executing the whole
         program.

Step 2   Key in the source program using the text editor or input
         mchine code directly to the MPF-IP.

Step 3   Use the two-pass assembler to assemble the source code to
         machine code without modifying the default values.   Then
         key in G F B 0 0 and ⌐←┘ to execute the program.

Step 4   Examine the contents in the A, B, C, D, E, H, L
         registers. If the values are not stored properly, repeat
         from step 1.

* Table --- from Chinese MPF-IP Manual p.12

Memory Address	Machine Language	Assembly Language
1800H	3E00	LD A,0
FF		RST 38H

III. Experiment 1-2

Write an assembly language program to set the contents of
registers as follows: B=12, C=34, D=56, E=78, H=9, L=A (use 16-
bit LD instruction to transfer two bytes of data each time).

Step 1   Same as the step 1 in Experiment 1-1.

Step 2   Same as the step 2 in Experiment 1-1.

Step 3   Press G F B 0 0 and ⌐←┘ to execute the program.

Step 4   Use the v key to check contents of each register.

2

A 16-bit data is composed of two bytes of data. The high-order byte is stored in the higher ordered memory address and the low-order byte is stored in the lower ordered memory address. For instance, the 16-bit data 1234H is stored in addresses 1820H - 1821H in the following way:

16-bit data	memory contents	memory address

```
16-bit data       memory contents       memory address

  ┌──────┐                   ┌──────┐     FB00H (lower order address)
  ╭──────────╮ low order     │      │
 1 2    3 4   byte           │ 3  4 │
        ╰──────────────────→ │      │
   ╭──────────────╮          │      │
   │ high order byte         │ 1  2 │     FB01H (higher order address)
   ╰────────────────────────→│      │
                             └──────┘
```
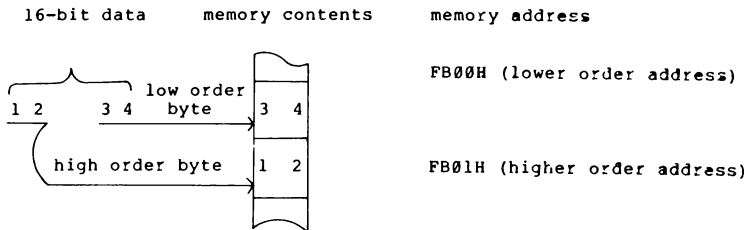
Address	Machine Language	Assembly Language
FB00H FB01H	013412	LD BC,1234H
	FF	RST 38H

Example 1-1 : THE USE OF A LOOP

Write a program to clear the contents of memory addresses FA00H - FA1FH.

Explanation:

(1) If we use an 8-bit LD instruction to transfer the data to each destination, the single load instruction would be executed for 32 (20H) times. It is more convenient to use the loop method in the program.

(2) Use register B is generally used as a loop counter. Set register B to 20H before the loop is executed. Use HL as a memory address pointer, and set the starting address FA00H to HL. HL is incremented by one and B is decremented by one for each loop. If B=0, then all loops have been executed; otherwise, run the loop again.

(3) The program is given below:

```
         Machine
Address  Language  Label  Opcode & Operand  Comment
-------  --------  -----  ----------------  -------
F800               LD      B,20H    ; Set loop counter equal to 32
                   LD      HL,0FA00H; Set HL equal to the starting address
                                    ; of memory to be cleared
                   XOR     A        ; Set A=0
         LOOP      LD      (HL),A   ; Load 0 into the memory address
                                    ; pointed to  by HL
                   INC     HL       ; Increment HL by 1
                   DEC     B        ; Decrement HL by 1
                   JR      NZ,LOOP  ; If B not = 0, return to LOOP
     FF            RST     38H      ; Return to the monitor program
```

IV. Experiment 1-3

Enter the program in the Example 1-1, assemble the source code to
machine code, execute the program. Then check if the contents of the
memory range from FA00H through FA1FH has been cleared.

V. Experiment 1-4 :

Write an assembly language program to set the contents of memory
address FA50H - FA7FH as follows: 0, 1, 2, 3, ...F.

(HINT: Change the loop counter and the value of the starting address.
register A is incremented by '1' in the next loop)

ADDRESS	MACHINE LANGUAGE	LABEL	OPCODE & OPERAND

4

Experiment 2
Basic Applications of Arithmetic and Logic Operation Instructions

Purposes:

1. To familiarize the user with the arithmetic and logic operation instructions
2. To understand the memory addressing modes
3. To understand the meaning of the register status flag
4. To practise arranging data for CPU registers and memory sections

Time Required: 4 hours

I. Theoretical Background:

1. 8-bit arithmetic and logic operation instructions:

The 8-bit arithmetic and logic operations in the Z80 CPU are performed in register A (accumulator). Registers A, B, C, D, E, H, and L can be used as operands in conjunction with register A in the LD instructions. If data are transferred between memory and register A, the memory address can be pointed to by HL, IX or IY registers. The meaning of the following instructions are given in the right-side comment field:

(1) ADD A   ; Data in register A is added to itself, i.e. the data is doubled shifted left one bit.

(2) ADC B   ; Register B and the carry flag are added to register A.

(3) SUB C   ; Data in register C is subtracted from register A.

(4) SBC (HL)   ; Subtract the data in the memory address pointed to by HL and the contents of the carry flag from register A.

(5) AND D   ; Logical "AND" of register D and register A.

(6) OR 0FH   ; Logical "OR" of data 0FH and register A .

(7) XOR A   ; Exclusive "OR" register A and itself. (Since register A is equal to register A, the result is zero).

(8) INC H   ; Increment the contents of register H by 1.

(9) INC (IX)   ; Increment the contents of the memory address
                   pointed to by register IX by 1.

(10) DEC C   ; Decrement the contents of register C by 1.

(11) DEC (IY+3)
                 ; The sum of the contents of register IY and
                   3 is used as the memory address pointer.
                   Decrement the contents of memory address
                   IY +3.

2. Data Addressing Mode

In the above assembly language instructions, the addressing modes used
can be summarized below.   Other addressing modes can be found in the
Z80 CPU technical manual.

(1) Register Addressing

Example:

In the instruction ADC A,B, ADC is the opcode which represents what
kind of operation will be performed.   The character A in the right
means that the data will be added to A.   The character B at the far
right means that the data to be added to A is taken from register B.

            (2) Register Indirect Addressing

            A 16-bit register is used to store the memory address.
            Example: In the instruction SBC A,(HL) , (HL) does not
                      mean that HL will be subtracted from register A.
                      Instead, the CPU takes the 16-bit data contained
                      in HL as the memory address and then accesses the
                      8-bit data stored in this memory address.   The
                      8-bit data pointed to by HL is finally subtracted
                      from register A.   IX and IY are called index
                      registers.   When a memory address is pointed
                      to by IX or IY, an 8-bit byte which is less than
                      +127 but larger than -128 can be added to this
                      register.

            For instance, the following two instructions can be used to
            add the data stored in the memory address pointed to by IX
            to the 8-bit data stored in the memory address pointed to by
            IX+2.   The result is stored in register A.

                      LD   A,(IX)
                      ADD   A,(IX+2)

6

(3) Immediate Addressing

Example : OR ØFH. On the right-hand side of the opcode OR,
a hexadecimal number, ØFH, is given. It means that
the number ØFH is logically ORed with the contents
of register A. Therefore, the data is part of the
instruction which is stored in memory. The CPU
fetches the data by using the program counter (PC)
as a reference address. The following instructions
are examples of immediate addressing.

```
LD      B,8
ADD     A,44H
SUB     A,ØA4H
```

3. Status Flags

After a logical or arithmetic operation is finished, the
result will be stored in register A and some of the status flags
(Carry, Overflow, Change Sign, Zero Result, Parity) will also be
affected. These status flags will be stored in the flip flops in
the Z-8Ø CPU. These flip flops form a register called the Flag
Register. The data in this register can be moved to memory, like
data in other registers, by specific instructions (PUSH instruction).
Some of the status flags are given below.

(1) Carry Flag

This flag is the carry from highest order bit of the
Accumulator. The carry flag will be set in either a signed or
unsigned addition where the result is larger than an 8-bit
munber. This flag is also set if a borrow is generated during
a subtraction instruction. The carry flag can be used as
a condition for jump, call, or return instructions. The carry
flag also serves as an important linkage in multi-byte arith-
metic operations. Three 8-bit data can be connected as a
24-bit data by using carry flag and four 8-bit data can be
connected as a 32-bit data.

(2) Overflow/Parity Flag

When signed two's complement arithmetic operations are
performed, this flag represents overflow. The Z-8Ø overflow
flag indicates that the signed two's complement number in
the accumulator has exceeded the maximum possible (+127) or
is less than minimum possible (-128).

7

When an arithmetic operation is performed in the Z80-CPU, the number in register A can be assumed to be unsigned data (0 - 255) or signed data (-128 - +127). Thus, either the carry flag or the overflow flag can be affected by the arithmetic operation. The programmer decides which interpretation is desired. The following arithmetic operations are described on the right-hand side.

```
      10101100          <- unsigned number 172 or signed number -84
  +)  11101000          <- unsigned number 232 or signed number -24
      --------------
  1 <- 10010100         <- unsigned number 148 with carry or signed
                           number -108 but no overflow


      01001010          <- signed or unsigned number 74
  +)  01000010          <- signed or unsigned number 66
      --------------
  0 <- 10001100         <- unsigned number 140 but no carry, or
                           signed number -116 but overflow has
                           occurred and the result becomes negative
          change sign
```

For logical operations in the Z80-CPU, this flag is set if the parity of the 8-bit result in the accumulator is even. This flag is very useful in checking for parity errors occurring during data transmission. Since carry and overflow will never occur in logical operations, the parity and overflow status can be stored in the same flip flop. This flip flop is called the P/V flag. By testing this flip flop the programmer can check overflow after arithmetic operations and check parity after logical operations.
operations.

(3) Zero Flag

If register A is zero after a logical or aritnmetic operation, this status will be registered in a flip flop called zero flag. The zero flag can be used as a condition for branch instructions. It is very useful in program looping.

(4) Sign Flag

If the leftmost bit (bit 7) of register A is 1 after a logical or arithmetic operation, the number in register A is interpreted as a negative number. The sign flag is then set to 1. This flag will be ignored if the programmer has assigned the data as unsigned numbers.

8

(5) The other flags designed for BCD arithmetic are not important for the programmer. The bit positions of the flags discussed above are shown below:

S	Z				P/V		C

Sign   Zero   Parity   Carry
                                or Overflow
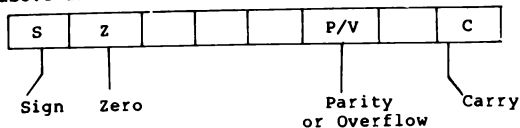
In microcomputers, it is usual to represent the contents of the flag register by two hexadecimal digits. The reader has to express this two-digit data with an 8-bit binary number. By referring to the bit positions in the flag register, the reader can obtain the status of the flag. For instance, if the flag register is 3CH, then the sign is positive, the value is non-zero, the parity is even or there is overflow has occurred but there is no carry. To know which flags will be affected by an instruction, the reader has to refer to the assembly language manual. Not all instructions will affect the status flags.

II. Example of Experiments

1. The following program can be used to add the contents of register D and register E together. The result will be stored in the register pair HL. Load the program into MPF-IP and then execute it. Record the result.

```
ORG    0FB00H  ; Starting Address <- 0FB00H
LD     A,E     ; A <- E
ADD    A,D     ; A <- A + D
LD     L,A     ; L <- A
LD     A,0     ; A <- 0
ADC    A,0     ; A <- A + 0 + Carry
LD     H,A     ; H <- A
RST    38H     ; Return to Monitor
```

Preset Value		Result of Program Execution				
Register		Register	Flag			
D	E	HL	Sign	Zero	P/V	Carry
5AH	A6H					
46H	77H					

9

2. The following program can be used to add the 16-bit data in memory
   addresses FA00H - FA01H to the 16-bit value in the register pair
   DE. The result will be stored in the register pair HL. Load the
   program into MPF-IP and execute it. Discuss the result obtained.

Preset values of memory: (FA01H)   =   ,(FA00H) =
Preset value of register DE pair   =   ,

```
        ORG     0F800H          ; Starting address <- 0F800H
        LD      A,(0FA00H)      ; A <- (FA00H)
        ADD     A,E             ; A <- A + E
        LD      L,A             ; L <- A
        LD      A,(0FA01H)      ; A <- (FA01H)
        ADC     A,D             ; A <- A + D + Carry
        LD      H,A             ; H <- A
        RST     38H             ; Return to monitor.
```

Result:

Preset values of memory: (FA01H)   =_____,(FA00H) = _____
Preset value of register DE pair   =_____,   _____
result   HL   =_____,   _____
                    Carry   =_____,   _____
                    Zero   =_____,   _____
                    Overflow   =_____,   _____
                    Sign   =_____,   _____

3. Revise the above program for a subtraction operation.

4. The following program can be used to add the 32-bit data in
   memory addresses 0FA00H - 0FA03H to the 32-bit data in memory
   addresses FA04H - 0FA07H. The result will be stored in memory
   addresses 0FA08H - 0FA0BH. The higher-order byte is stored in a
   higher address (This is conventional in microcomputer programming)

```
        ORG     0FB00H
        LD      B,4
        LD      IX,0FA00H
        AND     A
LOOP    LD      A,(IX)
        ADC     A,(IX+4)
        LD      (IX+8),A
        INC     IX
        DEC     B
        JP      NZ,LOOP
        RST     38H
```

Result of program testing:

Preset memory contents: (ØFAØ3H − ØFAØØH) = _____

                                   (ØFAØ7H − FAØ4H) = _____

results of program execution: (ØFAØBH ⇄ ØFAØ8H) = _____

                                   Flag Register   = _____

5. If the instruction ADC A,(IX+4) is replaced by SBC A, (IX+4),
then the above program can be used for a subtraction operation.
If the instruction DAA is inserted immediately after the ADC or
SBC instruction, then the program becomes a program for decimal
addition or subtraction. Load the revised program to MPF-IP and
test it.

Experiment 3
Binary Addition and Subtraction

Purposes:

1. To understand how an addition or subtraction operation is performed on a microcomputer.
2. To familiarize the reader with software programming techniques.

Time Required: 4 hours

I. Theoretical Background:

1. In this experiment, we only discuss unsigned binary integer addition and subtraction. For a N-bit binary number, its range is < $0, 2^{-1}$ >. For instance, if N = 8, the range is < 0,255 >; if N = 16, the range is < 0,65535 >. If the range of the numbers are expressed by hexadecimal digits, the ranges are < 0,FFH > and < 0,FFFFH >, respectively. If the sum of an addition operation is larger than the maximum value that can be represented by N bits, then carry is generated and the carry flag is set. In the subtraction operation, if the subtrahend is more than the minuend, a borrow is generated and the carry flag is set in the high order byte. The set carry bit indicates an incorrect result.

Example 3-1:

Single byte addition and subtraction.

Addition: 7FH + ADH = 12CH

```
     01111111 -> 7FH
  +) 10101101 -> ADH
  --------------------
    100101100 -> 12CH
```

Carry

Subtraction: 7FH - ADH   Subtraction: ADH - 7FH = 2EH

```
     01111111                    10101101
  -) 10101101                 -) 01111111
  -------------               -------------
    111010010                   000101110
```

12

```
                    Borrow                          Borrow
            The answer is incorrect          The answer is correct
            (   CY = 1 )                     (   CY = 0 )
```

Example 3-2

Three-byte addition and subtraction

Addition: 6A7CBDH + 4B65ACH = B5E269H

```
              6A          7C          BD
              4B          65          AC
          +  | 0 |<--  +  | 1 |<--  + | 0 |   <- Carry
          -------      -------      -------
          | 0 | B5     | 0 | E2     | 1 | 69
          Carry        Carry        Carry
```
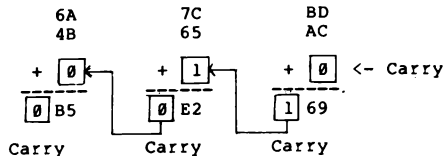
Subtraction: 854372H - 69ACBFH =

```
              85          43          72
            - 69        - AC        - BF
          ------      ------      ------
            0  1B       1  97       1  B3
          - | 1 |<--  - | 1 |<--  - | 0 |   <- Borrow
          ------      -------     -------
            0  1A     | 1 | 96    | 1 | B3
          Borrow        Borrow      Borrow
```
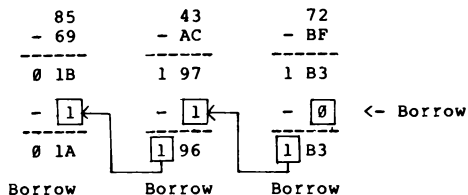
The borrow of the highest-order byte is 0, thus the answer is
correct. In multi-byte subtraction, the correctness of the result
depends upon the borrow of the highest-order byte. If the borrow
is 1, then the result is incorrect.

2. Order of data stored in memory:

    The conventional way of storing multi-byte data in memory is:
    the lowest order byte is stored in the lowest address and the
    highest order byte is stored in the highest address. The address
    of the multi-byte data is usually expressed by its lowest address.
    For beginning atstance, the number 7325H is stored beginning at
    memory address A in the following way:

```
        address A   | 25 |  <- low-order byte

        A + 1       | 73 |  <- high-order byte
```

If the starting address of 4 three-byte numbers stored in memory
is A, the data and their addresses can be shown as follows :

```
Address        A   | 56 |⎫
                   | 7C |⎬   987C56H
                   | 98 |⎭
             A + 3 | 43 |⎫
                   | 69 |⎬   AD6943H
                   | AD |⎭
             A + 6 | BC |⎫
                   | 01 |⎬   2501BCH
                   | 25 |⎭
             A + 9 | 78 |⎫
                   | 95 |⎬   439578H
                   | 43 |⎭
            A + 12 | 21 |⎫
                   | 96 |⎬   .......
                   |    |⎭
```

3. Design of Addition/Subtraction Programs:

The data used in addition/subtraction operation are stored in
memory according to the conventional method given above. The
starting address of the augend/minuend is stored in index
register IX. The starting address of addend/subtrahend is
stored in index register IY. The byte-number of the data is
stored in register B. First, clear CY and load the augend/
minuend into the accumulator. Then, use the indexed addressing
mode instruction ADC (SBC) to proceed with the addition/
subtraction operation. The result is stored in the original
address of the augend/minuend. Increment the index registers
and compare register B with zero. Repeat the load augend, add,
store increment cycle until the B register equals zero. Finally,
test the carry flag to check if the result is correct. The
only difference between the addition program and subtraction
program is that the instruction ADC is used for addition
operation and the instruction SBC is used for subtraction op-
eration. The flowcharts and programs are given below for com-
parison:

14

Addition:

```
        ┌─────────────┐
        │   Start     │
        └─────────────┘
               │
        ┌─────────────┐
        │ Clear Carry │
        └─────────────┘
               │
        ┌──────────────────┐
        │ Load Byte-number │
        │ Into Register B  │
        └──────────────────┘
               │
        ┌──────────────┐
        │  A <- ( IX ) │
        └──────────────┘
               │
        ┌──────────────────┐
        │  A<-A+(IY)+CY    │    ADC
        └──────────────────┘    Instruction
               │
        ┌──────────────┐
        │  ( IX ) <- A │
        └──────────────┘
               │
        ┌──────────────┐
        │  IX <- IX + 1│
        │  IY <- IY + 1│
        └──────────────┘
               │
  B = Ø   ◇ B <- B - 1 ◇
        └──────────────┘
               │  B = Ø
        ┌─────────────┐
        │    End      │
        └─────────────┘
```

Subtraction:

```
        ┌─────────────┐
        │   Start     │
        └─────────────┘
               │
        ┌─────────────┐
        │ Clear Carry │
        └─────────────┘
               │
        ┌──────────────────┐
        │ Load Byte-number │
        │ Into Register B  │
        └──────────────────┘
               │
        ┌──────────────┐
        │  A <- ( IX ) │
        └──────────────┘
               │
        ┌──────────────────┐
        │  A<-A-(IY)-CY    │    SBC
        └──────────────────┘    Instruction
               │
        ┌──────────────┐
        │  IX ) <- A   │
        └──────────────┘
               │
        ┌──────────────┐
        │  IX <- IX + 1│
        │  IY <- IY + 1│
        └──────────────┘
               │
  B = Ø   ◇ B <- B - 1 ◇
        └──────────────┘
               │  B = Ø
        ┌─────────────┐
        │    End      │
        └─────────────┘
```
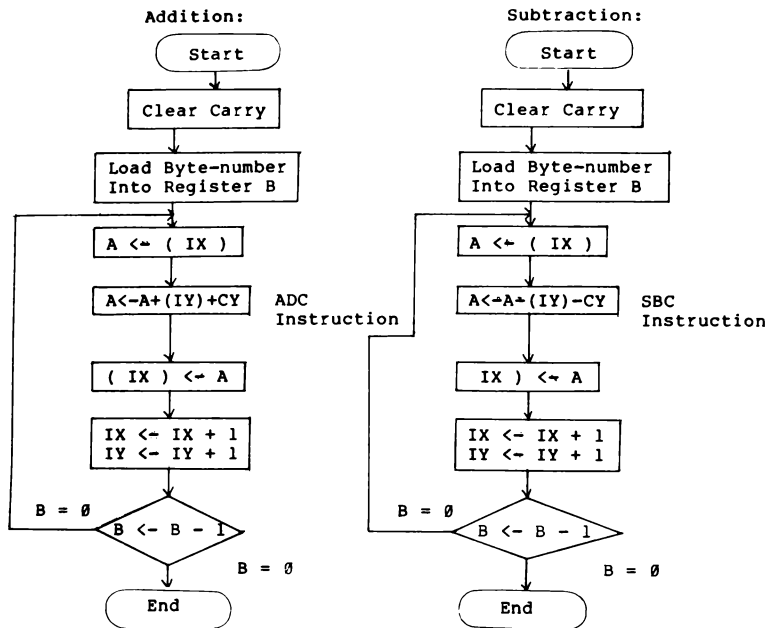
The following block diagram is given to demonstrate data
transfer in an addition operation.

15

LD A,(IX)

```
      ┌──┐                    ┌──┐
      │BD│  A ◄──────────────│BD│  <-  (IX)
      ├──┤                    ├──┤
      │AC│  (IY)◄──┐          │7C│
ADC   └──┘         │          ├──┤
Instruction        │          │6A│
      +  ┌─┐  CY    │          └──┘
         │∅│        │
         └─┘        │
      ─────────────┐│
   CY ↔> 1 69    A │
```

After executing the instruction
LD (IX), A, the contents of A
are stored in (IX).

```
      ┌──┐
      │69│  <-  (IX)
      ├──┤
      │7C│
      ├──┤
      │6A│                    ┌──┐
      └──┘                    │AC│
                              ├──┤         Starting address
                              │65│         of the addend
                              ├──┤         <- (IY)
      ┌──┐                    │4B│
      │AC│  <- (IY)           └──┘
      ├──┤
      │65│
      ├──┤
      │4B│
      └──┘
```
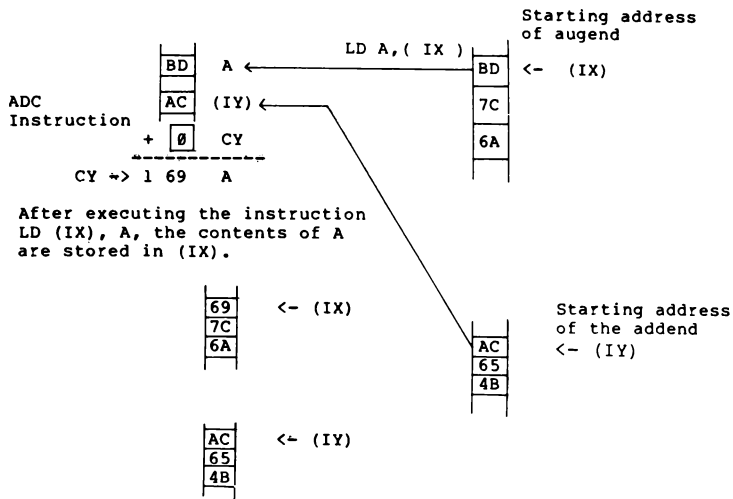
Instruction   INC IX increases the value of IX by one.
              In the comment field the incrementation
              of IX can be shown as IX   IX + 1
              INC IY leads to IY <- IY + 1
In each of frames showing the results of an instruction
step the current value pointed to by the index registers
are indicated by

                    index register
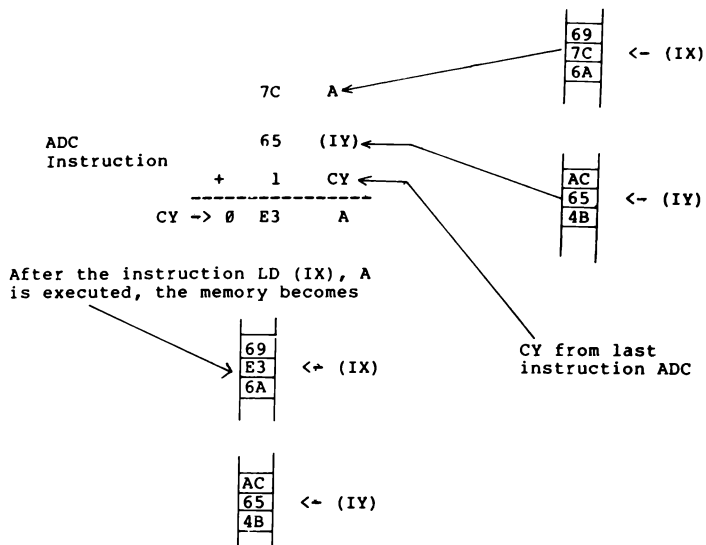
```
                                                    ┌────┐
                                                    │ 69 │
                                                    │ 7C │  <- (IX)
                                                    │ 6A │
                                                    └────┘
                        7C    A ◄─────────

        ADC             65   (IY)◄────────
        Instruction                                 ┌────┐
                    +    1    CY ◄──────────         │ AC │
                   ──────────────────                │ 65 │  <- (IY)
                   CY -> 0   E3    A                 │ 4B │
                                                     └────┘

   After the instruction LD (IX), A
   is executed, the memory becomes
                                 ┌────┐
                                 │ 69 │                CY from last
                              ──►│ E3 │  <- (IX)       instruction ADC
                                 │ 6A │
                                 └────┘

                                 ┌────┐
                                 │ AC │
                                 │ 65 │  <- (IY)
                                 │ 4B │
                                 └────┘


   When B = 0, the program execution is completed
   and the memory becomes

                                 ┌────┐
                                 │ 69 │
          CY 0   B5   A          │ E3 │
                                 │ B5 │
                                 └────┘

                                      <- (IX)


                                 ┌────┐
                  \              │ AC │
                                 │ 65 │
                                 │ 4B │
                                 └────┘

                                      <- (IY)
```
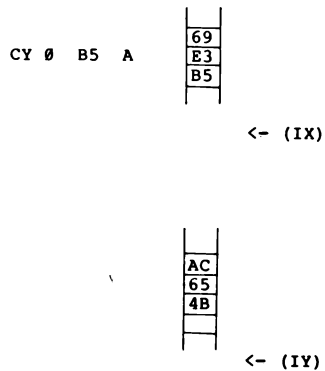
The addition program is given below. By replacing the instruction ADC A, (IY) by SBC A, (IY), the addition program becomes a subtraction program.

```
1. *** MPF-IP EXAMPLE PROGRAM ***
2. 3-BYTE ADDITION ( UNSIGNED INTEGER )
3. ENTRY ; AUGEND ADDRESS IN IX,
4.         ADDEND ADDRESS IN IY.
5. EXIT  : SUM IN AUGEND ADDRESS
6.
7. ADD3  : XOR A ; CLEAR CARRY FLAG
8.          LD B, 3 ; BYTE NUMBER IN B
9. ADDLP : LD A, (IX)
10.         ADC A, (IY)
11.         LD (IX), A
12.         INC IX
13.         INC IY
14.         DJNE ADDLP
15.         RET
```

4. Programming Technique:

From the above examples (3-1 and 3-2), we can see that the multibyte addition/subtraction operation can be accomplished by repeating the single-byte addition/subtraction operation, that is, by the loop operation of single-byte addition/subtraction. In the above program, register B is used as a loop counter. If the byte-number is 4, then 4 is loaded into B initially. Register B is decremented by 1 after each loop operation. The loop ends when B = 0. The instruction DJNZ is used for conditional jump. When B = 0, the program no longer executes the jump operation. Since ADC and SBC instructions are used in the programs, the CY is included in each addition/subtraction operation. Therefore, before the first byte addition/subtraction operation, the carry flag must be cleared (instruction XOR A). The index registers IX and IY are used as address pointers. By incrementing IX and IY, the CPU can access multibyte values stored in memory.

II. Student Exercises:

1. Load the above addition program into MPF-IP and store it on magnetic tape.

2. Replace the last instruction RET in the program by RST 38H. Load the following data into memory. The starting addresses of augend and addend are assigned as F900H and FA00H, respectively.Execute the program and record the result in the following table.

Augend	Addend	Answer	Check
793865H	ABCEDFH	CY =	
009543H	AB1236H	CY =	
954717H	003390H	CY =	

3. Replace the ADC instruction by the SBC instruction. Assign the starting addresses of minuend and subtrahend as F900H and FA00H, respectively. Execute the program and record the results obtained.

Minuend	Subtrahend	Answer	Check
683147H	336700H		
5935ABH	5877FFH		
049677H	F65B79H		

4. Express the data in the above two tables as five-byte data. Change the byte-counter to the proper value and execute the addition/subtraction program.

5. Write a program to add the 7-byte data in memory addresses FA00H – FA06H to the 7-byte data in memory addresses F900H – F906H and then subtract the 7-byte data in memory addresses F940H – F946H from the sum. The final result must be stored in memory with the starting address F900H.

Experiment 3-1:

The carry/borrow flag is used to indicate whether a carry/borrow is generated during an arithmetic or logical operation. If a carry/borrow is generated, then the flag is set to 1. Otherwise, the flag is zero. The carry flag is represented by bit 0 of the flag register.

REG.F
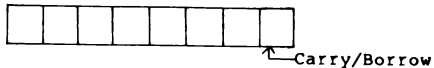
Carry/Borrow

In other words, the contents of the F register will be an even number if a carry/borrow is generated during the arithmetic or logical operation. If register F is an odd number, then no carry/borrow has been generated. Load the following program into MPF-IP. Execute every instruction by using the Single Instruction method. Observe the variations of register F and record the results in the table.

Address	Machine Language		Assembly Language	
FB00H	AF	1	XOR A	A,CY <-- 0
FB01H	3E7F	2	LD A,7FH	A <-- 7FH
FB03H	C6AD	3	ADD 7,ADH	CY,A <-- A + ADH
FB05H	C623	4	ADD A,23H	CY,A <-- A + 23H
FB07H	D613	5	SUB A,13H	CY,A <-- A - 13H
FB09H	D6B3	6	SUB A,B3H	CY,A <-- A - B3H
FB0BH	D615	7	SUB A,15H	CY,A <-- A - 15H
FB0DH	AF	8	XOR A	A,CY <-- 0
FB0EH	3E7F	9	LD A,7FH	A <-- 7FH
FB10H	CEAD	10	ADC A,ADH	CY,A <-- A + A DH + CY
FB12H	CE23	11	ADC A,23H	CY,A <-- A + 23H + CY
FB14H	DE13	12	SBC A,13H	CY,A <-- A - 13H - CY
FB16H	DEB3	13	SBC A,B3H	CY,A <-- A - B3H - CY
FBFBH	DE15	14	SBC A,15H	CY,A <-- A - 15H - CY
FB1AH	76	15	HALT	

| INSTRUCTION | (3) | (4) | (5) | (6) | (7) |

BEFORE EXECUTION

| A | A | A | A | A |
| 7 F | □ □ | □ □ | □ □ | □ □ |

AFTER EXECUTION

| + A D | + 2 3 | − 1 3 | − B 3 | − 1 5 |

| □□ □□ | □□ □□ | □□ □□ | □□ □□ | □□ □□ |
| CY A | CY A | CY A | CY A | CY A |

| (10) | (11) | (12) | (13) | (14) |

| CY | CY | A | A | A |
| □ □ | □ | □ □ | □ □ | □ □ |

| A | A | | | |
| □ □ | □ □ | − 1 3 | − B 3 | − 1 5 |

| + A D | + 2 3 | − CY | − CY | − □CY |

| □□ □□ | □□ □□ | □□ □□ | □□ □□ | □□ □□ |
| CY A | CY A | CY A | CY A | CY A |

21

Experiment 3-2:

Referring to the operation for of 3-byte addition in **example** 3-2, write a basic addition program using only three kinds of instructions: XOR A, LD A,(nn) and ADD A,(nn). Assume that the memory addresses of the addend, augend and sum are assigned as follows:

FB20H			FB23H			FB26H	
FB21H			FB24H			FB27H	
FB22H			FB25H			FB28H	

| Augend | Addend | Sum |
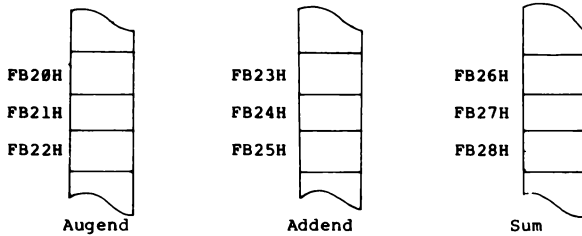
Explanation: In the above example, we see the following rules of addition:

(1) The addition operation moves from the low-order byte to the high-order byte, the carry generated in the low-order byte addition is added to the next higher order byte.

(2) The addition operation is executed with the aid of the accumulator. Its result is also stored in the accumulator. Thus to add two bytes together, one byte must be loaded into the accumulator first (using the LD A,(nn$_1$) instruction). The other byte is then added to the accumulator (using the ADD A,(nn$_2$) instruction or the ADC A,(nn$_2$) instruction). The final result is stored in an assigned memory address (using the LD(nn$_3$),A instruction).

Experiment 4
Branch Instructions and Program Loops

Purposes:

1. To familiarize the reader with the applications of conditional
   and unconditional branch instructions.
2. To familiarize the reader with the technique of designing
   program loops.
3. To practise using status flags in decision-making.

Time Required: 4 hours

I. Theoretical Background:

      1. Program Counter:

         The program counter (PC) is an important 16-bit register in
the CPU. When the voltage level of the RESET pin (pin 26) of the
CPU drops to 0 and then rises to 1 (by pressing the RS key), the
PC will be cleared to 0000H. The program execution is then
started from address 0000H according to the clock pulses supplied
by the system hardware. Once the CPU has fetched one byte of each
instruction from memory, the PC will be incremented by one
automatically. (The internal control circuit in the CPU
determines how many bytes are contained in the instruction after
the CPU has fetched the first byte of the instruction. The
instruction will be executed only when the PC has been
incremented by the number of bytes in the instruction). Usually,
the program is fetched from the memory instruction by the instruc-
tion for execution, starting from the low memory address.

      2. Branch Instructions:

         At any address, the PC can be changed to another address if the
programmer doesn't want the program executio to continue se-
quentially (For instance, when there is no memory beyond that
address or the program is not stored in that area). The program
then jumps to another address and continue its execution. For
example, the following assembly language means that the PC will
be changed to 1828H after this instruction has been executed,
and the program execution continues from address 1828H.

      LD PC, 1828H (This instruction is illegal in Z80 assembly
                language.)

23

Actually, in assembly language, JP (Jump) is used to indicate the change in sequence of program execution. The instruction has the same meaning as LD PC, F028H

```
JP  F028H
```

3. Conditional Branch Instructions:

A conditional branch instruction performs the jump operation if some specified conditions are met. These conditions are all dependent on the data in the flag register. This function makes the microcomputer capable of responding to various external conditions. It is also an indispensable tool for designing program loops. The actions of the following instructions are described in the comments to the right of the instruction:

```
CP   10H        ; Compare the accumulator with 10H and
                  set the proper flag.

JP   Z,   F028H ; If the zero flag is set, i.e. A = 10H,
                  then jump to address F028H and continue
                  the program execution.

JP   C,   245AH ; If the carry flag is set, i.e. A < 10H,
                  then jump to 245AH to execute other
                  program.

ADD  A,B        ; Otherwise, i.e. A > 10, continue the
                  program execution.
```

The condition of a conditional branch instruction is written after JP:

```
(1) JP  C, XXXX    ; If there is a carry, or carry flag = 1,
                     then jump to XXXX.

(2) JP  NC, XXXX   ; If there is no carry, or carry flag = 0
                     then jump to XXXX.

(3) JP  Z, XXXX    ; If zero flag = 1, or the result of previous
                     operation is zero, then jump to XXXX.

(4) JP  NZ, XXXX   ; If zero flag = 0, then jump to XXXX.

(5) Jp  PE, XXXX   ; If parity flag = 1 (even parity), or there
                     and an overflow in the previous arithmetic
                     operation, then jump to XXXX.

(6) JP  PO, XXXX   ; If P/V flag = 0 (odd parity or no overflow)
                     then jump to XXXX.
```

```
(7) JP  P, XXXX      ; If sign flag = 0 (the sign of result of
                       previous operation is positive) then jump
                       to XXXX.

(8) JP  M, XXXX      ; If sign flag = 1 (negative) then jump to
                       XXXX.
```

4. Jump Relative:

To reduce the memory space occupied by the program and also
reduce the cost of the microcomputer system, the Z80
microcomputer can use relative addresses to specify the
displacement of a program jump. Since most displacements
in a jump are within the rage between +127 and -128, a one
byte number can be used to indicate this displacement. One byte
of memory is saved for each jump operation compared with the
two-byte absolute address in JP instructions. The operations of
the following instructions are described in the commands to the
right of the instruction.

```
    JR 10H           ; Jump forward 10H (16) locations from the
                       present program counter (the address of the
                       next instruction). Actually, the address
                       of the next instruction to be executed is
                       obtained by adding 10H to the present PC.

    JR C,FOH         ; If carrry flag = 1, then jump backward 10H
                       (16) locations from the present program
                       counter.  Since the leftmost bit of FOH
                       is 1, it is recognized as a negative
                       number (its 2's complement is 10H).

    JR NC,7FH        ; If carry flag = 0, than jump forward 127
                       locations (maximum value)

    JR Z,80H         ; If zero flag = 1, i.e. the result of the
                       previous operation is zero, then jump
                       backward 128 locations. 80H (-128) is
                       the minimum negative number that can be
                       used in a relative address.
```

From the above examples, we can see that a positive relative
address means jumping forward. The largest displacement then
is 7FH (+127). A negative relative address means jumping backward.
Its largest displacement is 80H (-128). The displacement is
always measured from the address of the next instruction's op code.
Relative jumps can be unconditional or conditional. The
conditional jump depends on the status of the carry or zero flag.
In the Z80 system, the data in the sign or P/V flag cannot be
used as the condition of a relative jump.

5. Program Loop:

One of the important advantages of a computer is that it can
repeat the steps in a repetitive task as many times as is
necessary to complete the task. This is accomplished by
using a program loop. Looping is a very powerful tool in
program design. A basic program loop must contain the following:

(1) A loopss counter preset with the number of loops
    to be executed. Usually, a CPU register is used as
    a loop counter. Of course, memory can also be used
    as a counter.

(2) The loop counter is decremented by 1 after one cycle of the
    loop has been executed. After each cycle the value of the
    loop counter must be checked. If the counter is not 0, then
    the loop repeats until the loop counter equals to 0.

The following program can be used to add the 8-bit data in memory
addresses 1900H - 190FH and store the result in the DE
register pair. This is a typical application of a program loop.

```
    LD C,10H        ; Use register C as the loop counter. Since
                      sixteen bytes data are to be added together,
                      10H is preset in C.

    XOR A           ; Clear the accumulator

    LD HL,1900H     ; Use the HL register pair as the address pointer.
                      The contents of the memory pointed to by HL
                      will be added to register A.
                      The first address is 1900H.

    LD D,A          ; Register D is used to store the carry
                      generated during the addition operation.
                      Clear Register D.

XX  ADD A,(HL)      ; Add the contents of the memory address pointed
                      to by HL to Register A. This instruction will
                      be repeated 16 times. XX is assigned
                      as the label of this instruction's address.

    INC HL          ; Increment HL by 1. The new HL points to the
                      next byte in data memory to be added to
                      Register A.

    JR NC,YY        ; If no carry is generated, jump to address
                      YY to continue program execution.

    INC D           ; If a carry is generated, add this carry to
                      Register D.
```

26

```
YY  DEC C              ; Decrement register C by 1.

    JR NZ,XX           ; If the result is not zero (zero flag = 0),
                         the program loop has not finished.  Jump to
                         XX to repeat the loop.

    LD E,A             ; If zero flag = 1, then all data have been
                         added together. Load A into E, the answer
                         will be stored in the DE register pair.

    END
```

There are various methods of designing a program loop. Try to
design the program loops described in the following illustrations.

II. Example Experiments:

1. A program loop with a loop number of less than 256 : If the loop
   number is less than 256, register B is recommended as the loop
   counter. At the end of the loop, the DJNZ instruction can be used
   to decrement register B. If the result is not zero, jump to the
   assigned location using the relative jump method to continue the
   program execution. Try to analyze the following program and
   verify its function by loading it into the MPF-IP and executing it.

```
          ORG      0F000H
          LD       HL,0FA00H
          LD       B,20H
   ┌─→LOOP  LD       (HL),A
   │      INC      HL
   └─────DJNZ     LOOP
          RST      38H
```

Experimental result:

(1) Preset register A to 0 and then execute the above program
    Results:
       Contents of memory addresses F900H – F91FH:
       Contents of memory address F920H:

(2) Preset register A to 55H and execute the above program.
    Results:

(3) Preset register A to 64H and replace the second instruction
    LD B,20H by the instruction LD B,0 . Execute the program again.
    Results:
       Contents of memory addresses F900H – F9FFH:

27

Discussion:

2. Nested loops:

In a more complicated program, a loop can be totally nested or embedded inside another loop. The following program can be used to divide the 256 bytes of data stored in memory into 19 groups. The starting address of the memory is F900H. Put the contents of each group of data in the form of a hexadecimal number:

0.....(1st set), 1.....(2nd set), 2.....(3rd set),.....F..... (19th set).

```
                       LD        HL,0F9FFH
                       LD        C,0FH
         ┌──────→ LOOP2  LD      B,10H
         │  ┌───→ LOOP1  LD      (HL),C
         │  │           DEC       HL
         │  │  Small loop ╱ DJNZ  LOOP1
         │  │           DEC       C
         │Large loop   ╱JP        NZ,LOOP2
         └──┴──────────←  RST     38H
```
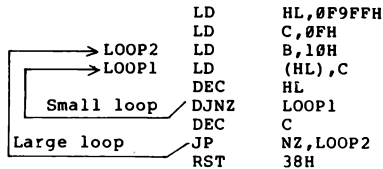
(1) Translate the above program into machine language and then load it into the MPF-IP. Execute the program.
    Results:

(2) Revise the above program such that the 19 bytes of the first group are all "F", and the 16 bytes of the last group are all "0".

3. A program loop with loop number larger than 256: If the loop number is larger than 256, a 19-bit register can be used as the loop counter. But, in the Z80 system, incrementing or decrementing a 16-bit register can not affect the status flag. Thus, some auxiliary instruction is used to determine whether the loop counter is zero. The following program is supposed to be able to set all data in RAM F980H - FAFFH to AAH. Try to find the errors in this program and correct them. Load the correct program into the MPF-IP and record the result of the program execution.

```
        ORG     ØFØØØH
        LD      BC,Ø18ØH
        LD      HL,ØF98ØH
LOOP    LD      (HL),ØAAH
        INC     HL
        DEC     BC
        JR      NZ,LOOP
        HALT
```

4. A program 'loop without a down counter : A program loop need not
   use a down counter. The function of the down counter can be
   replaced by using an up counter or using the method of address
   comparision or data comparison. Study the method used in the
   following program loops. Load the programs into MPF-IP and execute
   them.

   (1) Move the data string in the memory (RAM) section with
       starting address FAØØH to the memory (RAM) section with
       starting address F9ØØH. The movement will be terminated when
       data ØFFH is found.

```
        ORG     ØFØØØH
        LD      HL,ØFAØØH
        LD      DE,ØF9ØØH
LOOP    LD      A,(HL)
        LD      (DE),A
        CP      ØFFH
        JR      Z,EXIT
        INC     HL
        INC     DE
        JR      LOOP
EXIT    RST     38H
```

   (2) Replace all the data stored in the memory section starting
       from the address pointed to by HL to the address pointed to
       by DE by their corresponding 2's complement. In testing
       the program, the values of HL and DE must be preset first.
       The value of HL must be larger than that of DE.

```
        ORG     ØFØØØH
LOOP    LD      A,(HL)
        NEG
        LD      (HL),A
        INC     HL
        AND     A
        SBC     HL,DE
        ADD     HL,DE
        JR      NZ,LOOP
```

Experiment 5
Stack and Subroutines

Purposes:
1. To understand the meaning and applications of the stack.
2. To understand the designing techniques and applications of subroutines.

Time Required: 4 hours

I. Theoretical Background

1. Stack: In program design, a stack is recognized as a memory
section which has only one port for input and output. Data are
written in or retrieved from stack via this port. The first
data placed in stack is said to be at the bottom of stack.
The data most recently placed in stack is said to be at the
top of stack. Thus, a stack is also called a last-in
first-out memory. A stack can be constructed by hardware shift
registers or general RAMs. In the Z80 microcomputer system,
the programmer can assign a region of RAM as the stack. To define
a stack at the top of RAM, the highest address of RAM is incre-
mented by 1 and then loaded into the stack pointer (SP) in the
CPU. The following program and diagrams illustrate the operation
of stack.

Instruction Number	Instruction	Comment
(1)	LD SP,0FEA0H	; Stack pointer is set to 0FEA0H, i.e. the RAM section with address less than or equal to FEA0H is assigned as stack.
(2)	DEC SP	; Decrement SP by 1. Stack pointer is at FE9FH, i.e. at the bottom of stack.
(3)	LD (SP),H	; Load the contents of register H into memory (RAM) address FE9FH.
(4)	DEC SP	; Decrement SP by 1 again.
(5)	LD (SP), L	; Place the contents of L at the top of stack (i.e. above H).
(6)	DEC SP	
(7)	LD (SP),A	; Place the contents of A at the top of stack (i.e. above L).
(8)	DEC SP	
(9)	LD (SP), F	; Place the contents of F at the top of stack (i.e. above A).

.
.
.
.

30

(10)	LD C, (SP)	; Pop one byte of data from the top of stack and move it to register C.
(11)	INC SP	; Increment SP by 1. SP is moved towards the top of the stack.
(12)	LD B,(SP)	; Pop data from the top of stack.
(13)	INC SP	; Increment SP by 1 again.
(14)	LD E,(SP)	; Pop data from the top of stack and move it to register E.
(15)	INC SP	
(16)	LD D,(SP)	; Pop data from the top of stack and move it to register D. This data is the first one that is stored in stack.
(17)	INC SP	; SP is at the initial value.

RAM   RAM


```
SP assigned by                        F    SP assigned by                 C
the 9th instruction ->   | F |             the 10th instruction ->  | F |
                                      A    SP assigned by                 B
                         | A |             the 12th instruction ->  | A |
                                      L    SP assigned by                 E
                         | L |             the 14th instruction ->  | L |
SP assigned by                        H    SP assigned by                 D
the 3rd instruction ->   | H |             the 16th instruction ->  | H |
Initial value of SP                        SP assigned by
            FEA0H ->                       the 17th instruction ->
```
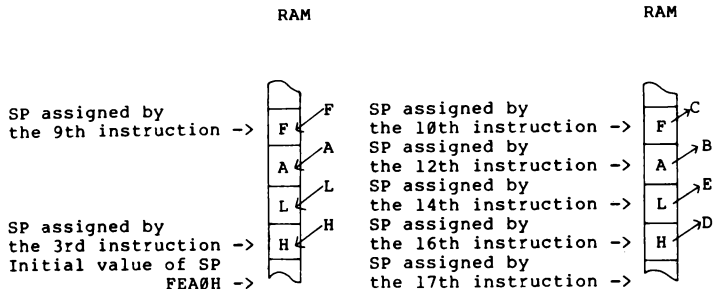
Push data onto the stack   Pop data from the stack

From the above illustrations of stack operation, we can see that
data can be stored in RAM by using SP as the pointer. SP is
decremented by 1 whenever one-byte of data is stored in and the
stack area becomes larger. The SP will be incremented by 1 whenever
one-byte data is retrieved from the stack area and the stack area
becomes smaller. The process of decrementing SP (pushing data onto
stack) or incrementing SP (popping data out of stack) can be
accomplished automatically by special hardware design. A stack can
also be used to store a 16-bit address (or data). In the Z80/8085
system, there are instructions to push a 16-bit register pair onto
stack and pop a 16-bit data out of stack. During each operation, SP
is decremented or incremented by 2. The following program is
equivalent in function to that of the program given above.

```
         LD SP,  ØFEAØH   ; Same as 1st instruction.
         PUSH    HL       ; Same as no. (2)(3)(4)(5) instructions.
         PUSH    AF       ; Same as no. (6)(7)(8)(9) instructions.
         POP     BC       ; Same as no. (1Ø)(11)(12)(13) instructions.
         POP     DE       ; Same as no. (14)(15)(16)(17) instructions.
```

Instructions PUSH and POP can be used to temporily store data in registers and also used to transfer register data. An example is given below.

```
         PUSH    BC
         POP     IX       ; Move the 16-bit data in BC to IX
         PUSH    HL
         AND     A
         SBC     HL,DE    ; Compare HL with DE to generate status
                          ; flags. The value of HL is kept
                          ; unchanged.
         POP     HL
```

It is a very important that the number of PUSH instructions be equal to the number of POP instructions in the stack operation.

2. Subroutine:

Programs for arithmetic (addition, subtraction, multiplication or division), keyboard and display control, etc are often used as part of a large program in practical applications. If the programmer rewrites these small programs everytime he needs them, the whole program would be very tedious to write. To save memory space for the program and reduce errors, subroutines are often used in a large program. Instructions CALL and RET are used to manipulate the subroutines. The subroutines can be executed unconditionally or according to the conditions of flags. The instruction CALL in the main program is used to call the subroutine. Its function consists of two operations which are illustrated below.

```
    ┌── CALL     ØFA38H  ;  Call the subroutine stored in address ØFA38H.
    │
Equivalent to
    │
    │   ┌ PUSH   PC      ; Push the current program counter onto
    └───┤                  stack.
        └ JP     ØFA38H  ; Jump to address FA38H and continue the
                          program execution.
```

RET instruction does'nt need an operand (1 byte instruction), it is the same as 'POP PC' instruction.

```
    ┌─RET              ; Return to original program and continue
    │                    to execute.
Equivalent to         ; Retrieve 16-bit data in stack and load into
    │                    PC, then ececute program according PC
    │                  ; contents.
    └─POP PC
```

Calling a subroutine is an important step in a program. Subroutines in a program can be in a nested form that is a subroutine can be another subroutine. The relationship is shown below:

Main Program
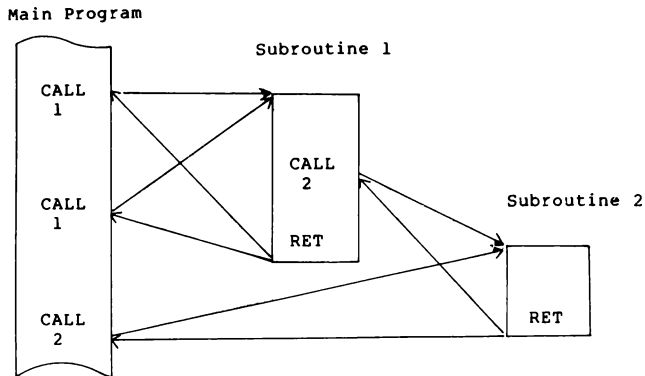
Usually, subroutines are written by a specialist. The user only has to understand its calling ˌrocedure . If the subroutine is written by the user himself, the following items must be considered in the design:
    (1) An easily-remembered name must be chosen for the subroutine.
    (2) How to get the data required in the subroutine before executing the subroutine.
    (3) How to express the result after executing the subroutine.
    (4) Which register will be changed after executing the subroutine.
    (5) How much memory will be occupied by the subroutine and how much time is needed for the CPU to execute the subroutine.

The following must also be considered when a subroutine is called by the main program:

(1) Registers that should not be changed by the execution of the
    subroutine must be pushed onto stack before calling the
    subroutine.
(2) How the results obtained from the subroutine execution will
    be transmitted by the main routine (the calling routine).

The following listing is a sample subroutine named MADD.
It can be used for multi-byte BCD addition.

```
                            1 ; *** MULTIBYTE BCD ADDITION ROUTINE ***
                            2 ; ENTRY: HL POINTS TI LOW ORDER BYTE OF AUGEND
                            3 ;          DE POINTS TO LOW ORDER BYTE OF ADDEND
                            4 ;           B = BYTE NUMBER, 1 BYTE = 2 BCD DIGIT
                            5 ;'EXIT : IX POINTS TO LOW ORDER BYTE OF RESULT
                            6 ; REG. CHANGE : AF,B,HL,DE,IX
                            7 ; MEMORY USED : 15 BYTES
                            8
 FB00     AF               9 MADD    XOR A   ; CLEAR CARRY FLAG
 FB01     1A              10 MADD1   LD A,(DE)
 FB02     86              11         ADD A,(HL)
 FB03     27              12         DAA
 FB04     DD7700          13         LD (IX),A
 FB07     13              14         INC DE
 FB08     23              15         INC HL
 FB09     DD23            16         INC IX
 FB0B     10F4            17         DJNZ MADD1
 FB0D     C9              18         RET
```

Two 4-byte BCD data are stored in the memory with starting
addresses at 0FA00H and 0FA40H, respectively. To add the BCD data
together and store the result in RAM address FA08H, subroutine
MADD is called by the following procedure:

```
          LD      B,  4           ; Set Byte Number = 4 .
          LD      HL, 0FA00H      ; Hl points to the address of augend.
          LD      DE, 0FA40H      ; DE points to the address of addend.
          LD      IX, 0FA08H      ; IX points to the address of sum.
          CALL    MADD
```

34

II. Example Experiment:

(1) Using the instructions for stack operation, write a routine to move the data in HL, DE and BC to HL', BC' and DE', respectively. Load the program into MPF-IP and execute it.

(2) In the following program, a small loop is embedded in a large loop. The function of this program is to shift all the 8-bit the data in bytes in the address FA11H - FA20H left four bits. Use register B as the loop counter for both small and large loops. Load the program into MPF-IP and execute it. Discuss the reason why register B can be used as the counter for both loops.

F800		1		ORG 0F800H
F800	0621	2		LD B,21H
F802	21001A	3		LD HL,0FA40H
F805	C5	4	LOOP1	PUSH BC
F806	7E	5		LD A,(HL)
F807	0604	6		LD B,4
F809	87	7	LOOP2	ADD A,A
F80A	10FD	8		DJNZ LOOP2
F80C	77	9		LD (HL),A
F80D	23	10		INC HL
F80E	C1	11		POP BC
F80F	10F4	12		DJNZ LOOP1
F811	76	13		HALT

(3) By calling the subroutine given in part I (multi-byte BCD addition routine), write a program to add two 8-byte data stored in memory FA00H and FA08H. The result must be stored in the 8-byte memory starting at 0FA40H.

(4) Revise the above program for BCD subtraction or multi-byte binary addition/subtraction. Test the program and record the method of revision used.

(5) Write a subroutine to change the 16-bit data in HL to its 2's complement. Write a main program to change the data in IX and IY to their 2's complements. Load the program into MPF-IP and test it.

(6) By using the above routine for complementing the HL register pair, write a program to subtract DE from the data in IY and store the result in IY.

Experiment 6
Rotate, Shift Instructions, and
Multiplication Routines

Purposes:

      1. To understand the use of Rotate and Shift instructions
      2. To understand the designing techniques and uses of a binary
      multiplication subroutine.

Time Required: 4 - 8 hours

I. Theoretical Background:

      1. The 9-bit data formed by the carry flag and 8-bit data in a
      register or memory can be shifted one bit left or right by
      ROTATE or SHIFT instructions. The ROTATE and SHIFT instructions
      are mainly used for multiplication and division. We multiply
      a number by rotating and shifting left the bits that constitute
      a number, while a division operation is done by rotating or
      shifting right the bits that constitute a number. There are
      many ways to rotate or shift the bits of a number. So, there
      are 13 different types of ROTATE and SHIFT instructions. Please
      refer to the MPF-IP User's Manual, Appendix C. The mnemonic
      codes of these instructions are described below.

        (1) If the leftmost character of an instruction is "R", it is a
        "ROTATE" instruction. Such instructions can be used to rotate
        the 9-bit data (formed by 8-bit data and carry flag) left or
        right one bit, e.g. RLCA, RL, RRA, etc.
        If the leftmost character is "S", then it is a "SHIFT"
        instruction. All the 9-bits of the data are shifted left or
        right by one bit. The bit shifted out from one side will
        not be moved in from other side. Examples of such instructions
        are SAL and SRL.

        (2) If the second character from the left is "R", it means "shift
        right" or "rotate right". Instructions RR, SRL, RRCA, etc.
        are examples.
        If the second character in the left is "L", it means "shift
        left" or "rotate left". Instructions RL, SLA, RLCA, etc. are
        the examples.

        (3) The meaning of the third character is more complicated, but
        it can be summarized as follows:

(a) In ROTATE instructions:
The third character "C" represents the circular rotation
of 8-bit data, carry flag is not included. The third
character (or the fourth character) "A" means that this
instruction is operated with the accumulator.
Instructions RLA, RRA, RLCA and RRCA are examples.
The third character "D" indicates the shift operation
on decimal or hexadecimal numbers, for example, RLD and
RRD. These instructions are designed to rotate the
memory pointed to by HL left or right one digit (4 bits)
The digit entering from the left or right direction comes
from bit 0 - bit 3 of the accumulator. The digit moving
out from the other side is sent to bit 0 - bit 3 of the
accumulator.

(b) In SHIFT instructions:
The third character "A" indicates "Arithmetic Shift".
Binary data shifted left means multiplying it by 2 .
Binary data shifted right means dividing it by 2 . Two
of these instructions are SLA and SRA. Because bit 7 is
assigned as "sign bit" and the sign of the data is not
changed by these operations, the leftmost bit (bit 7)
must be kept unchanged.
The third character "L" means "logical shift".
Instruction SRL is an example. In these operations, a
"0" is always moved to bit 7 from the left direction.

2. Binary Multiplication:

    The operation of unsigned binary multiplication can be
accomplished by shifting the binary number left or by a program
loop of addition. An example of binary multiplication by
hand-calculation is illustrated below.

```
        0101←—<- Multiplicand ->——→ 0101
      X 1011←—<-  Multiplier  ->——→X 1011
      --------                       --------
        0101←                       →0101
       0101←                        → 0000
       0000←                        → 0101    partial
    +) 0101←                        → 0101    product
    -----------                     ------------
      0110111   <--- Answer --->     0110111
```

37
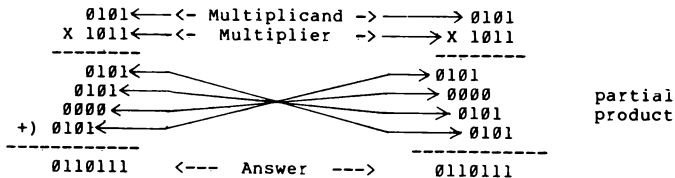
In the above calculation, one bit of the multiplier is checked. If that bit is 1, the multiplicand is copied as the partial product. If that bit is 0, 0000 is given instead. The position of the partial product is arranged such that the least significant bit of the multiplicand is aligned with the bit of the multiplier being checked. In this example, multiplicand and multiplier are both 4-bit data. Thus, it is necessary to repeat the operations of checking, shifting and addition four times. Similarly, the operations must be repeated 8 times for 8-bit data multiplication and 16 times for 16-bit data multiplication. In the left-hand side calculation given above, the bit-checking process starts from the least significant bit of the multiplier. In the right-hand side calculation, the bit-checking process starts from the most significant bit. But the results of the two calculations are identical. The program of binary multiplication for microcomputers can be designed by a method similar to the above calculation.

Example: Multiply the 8-bit data in register E by the 8-bit data in register A. The product is stored in the HL register pair.

Answer: Specific registers have been assigned to store multiplicand, multiplier and product according to the characteristics of the Z80 instruction set. Using the calculation algorithm given in the right-hand side of the above example, the program is designed as follows.

1. In the above hand calculation, the bit-checking process starts from the least significant bit. A program loop can be employed in the example. The multiplier is 8-bits long, thus the loop number is equal to 8 . In every loop execution, the bit being checked (in register A) can be shifted into the carry flag by the RLCA instruction. Then, according to the condition of the carry flag, we can decide what will (or will not) be done next.

2. If the first bit checked (the leftmost bit) is 1, the partial result is actually obtained by shifting the multiplicand left (n-1) bits, where n is the number of bits in the multiplier. The other partial results are obtained by shifting the partial products left (n-2) bits, (n-3) bits,........, etc. In this example, no other registers are required to store the partial results. Each partial result can be added directly to the HL register pair.

3. From the above description, we can see that the partial products must be shifted left (n-1) bits, (n-2) bits, (n-3) bits,...,etc. Since the bit-checking is also moving left in the process, we can generate a new intermediate result by immediately adding each partial product to the previous intermediate result. This method is more efficent and is used in the following program flowchart.

4. Register Assignments:

```
┌─────────────┬─────────────┐
│      D      │      E      │ <- Multiplicand
└─────────────┴─────────────┘
              ┌─────────────┐
              │      A      │ <- Multiplier
              └─────────────┘
┌─────────────┬─────────────┐
│      H      │      L      │ <- Answer
└─────────────┴─────────────┘
```

5. Program Flowchart :

```
   ( START )
      │
   ┌──────────┐
   │  B <- 8  │      <- Set B as the loop counter. For an 8-bit
   └──────────┘         multiplier, B is set to 8
      │
   ┌──────────┐
   │  D <- 0  │      <- 16-bit addition will be performed.
   └──────────┘         First clear D.
      │
   ┌──────────┐
   │  HL <- 0 │      <- Set the initial value of answer to 0 .
   └──────────┘
      │
   ┌──────────┐
   │    HL    │      <- Shift the intermediate result left one
   │    <-    │         bit. The first shift process is invalid.
   └──────────┘         Thus the first partial product will be
      │                 shifted left n-1 bits after the loop is
      │                 executed once.
   ┌──────────┐
   │    A     │      <- The leftmost bit of the multiplier is
   └──────────┘         moved to the carry flag for testing.
      │
=0    ◇
  ◇ CARRY? ◇            If the leftmost bit of the multiplier
      │  =1             is 1,  the multiplicand is added to
   ┌──────────────┐     the intermediate result.
   │ HL <- HL + DE│     Otherwise, the addition is ignored.
   └──────────────┘
      │
B=0   ◇
  ◇ B <- B - 1 ◇     <- Check if the program loop has been
      │   B=0           completed. If it is, stop execution.
   ( EXIT )             Otherwise, repeat the loop operation.
```

39

```
LOC OBJCODE STMT SOURCE STATEMENT              ASM 3.0

                 1;***MULTIPLY***

                 2:ENTRY:

                 3        ;MULTIPLER IN E

                 4        ;MULTIPLICAND IN A

                 5;EXIT:

                 6        ;PRODUCT IN HL

                 7;REG   CHANGE   : B,D,HL

                 8;MEMORY BYTE    : 14

                 9;EXECUTION TIME :<395 CLOCK / 221.2 uS.

                 10;

                 11MP8:

FB00  0608       12MULTI  LD B,8        ;SET BYTE COUNTER=8

FB02  1600       13       LD D,0

FB04  62         14       LD H,D

FB05  6A         15       LD L,D         ;CLEAR D,HL REGISTER

FB06  29         16 LOOP  ADD HL,HL      ;SHIFT HL LEFT

FB07  07         17       RLCA           ;ROTATE BIT 7 OF "A" INTO

                                         ;CARRY FLAG

FB08  3001       18       JR NC,NADD     ;TEST CARRY FLAG

FB0A  19         19       ADD HL,DE      ;ADD DE TO HL

FB0B  10F9       20 NADD  DJNZ LOOP      ;END?

FB0D  C9         21       RET
```

II. Example Experiments:

1. The following program can be used to shift the 32-bit data stored in the HL and DE register pairs, which are adjacent, right one bit (or divide the data by 2). Load the program into MPF-IP and test it. Next, revise the program such that it can be used to shift the 32-bit data left one bit (or multiply it by 2).

```
ORG     0F800H
SRA     H
RR      L
RR      D
RR      E
RST     38H
```

2. Write a program to shift the 32-bit data, stored in RAM addresses FA00H - FA03H, left five bits (or multiply it by 20H). Load the program into MPF-IP and test it. The starting address of the program is assigned as FB00H.

3. Using the RLD instruction, write a program to shift the BCD data, stored in RAM addresses FA00H - FA03H, left four bits. The starting address is assigned as F830H. Load the program into MPF-IP and test it.

4. The following program can be used to multiply the 16 bit data stored in the DE register pair by the contents of register A. Load the program into MPF-IP and test it. Compare this program with the program given in Theoretical Background. Discuss the advantages and disadvantages of this program.

```
MPY8    LD      BC,800H
        LD      H,C
        LD      L,C
M1      ADD     HL,HL
        RLA
        JR      NC,M2
        ADD     HL,DE
        ADC     A,C
M2      DJNE    M1
        RST     38H
```

5. Write a program to multiply the 32-bit data stored in RAM addresses FA00H - FA03H by the 32-bit data stored in RAM addresses FA04H - FA07H. The product must be stored in RAM addresses FA08H - FA0FH.

41

Experiment 7
Binary Division Routine

Purposes:

1. To understand how to write a binary division subroutine for
   a microcomputer.
2. To familiarize the reader with the technique of software
   programming.

Time Required: 4 - 8 hours

I. Theoretical Background:

1. Binary division by hand-calculation:

The following example will be used to illustrate the detailed
procedure of binary division.
Divide 11101101 by 00010100
(1) Write the dividend on the right-hand side, divisor on the
    left-hand side, and put the quotient above the divisor.

```
              ┌──────────┐
              │          │      <- Quotient
              ├──────────┤
              │ 11101101 │      <- Dividend (237)
              └──────────┘
    ┌──────────┐
    │ 00010100 │   <- Divisor (20)
    └──────────┘
```

(2) Shift the dividend and the quotient left one bit.

```
              ┌──────────┐
              │ 0        │      <- Quotient (Answer)
              ├──────────┤
              │ 11101101 │      <- Dividend
              └──────────┘
         00010100                <- Divisor
```

To compare the dividend and the divisor, place seven zeros
after the divisor in the columns beneath the dividend. It
can then be seen that the dividend is smaller than the
divisor. Therefore put "0" in the position of quotient.

(3) Continue to test if the dividend is less than the divisor
    with each shift. If the dividend is still less than the
    divisor, then put a "0" in the quotient. Otherwise, put a
    "1" in the quotient and the divisor is subtracted from
    the dividend. In this example, the dividend and the quo-
    tient must be shifted left five bits before a "1" can
    be put in the quotient. Thus four "0"s and one "1"
    are put in the quotient in the following way.

```
                    ┌─────────┐
                    │ 00001   │   <- Quotient (when the dividend
                    └─────────┘                is larger than
                    ┌─────────┐
                    │ 11101101│   <- Dividend the divisor "1" is
                    └─────────┘                put in the quotient.
                 ┌───────────┐
                 │ 00010100  │  <- Divisor
                 └───────────┘
```

(4) Subtract the divisor from the dividend.
    The difference becomes the dividend.

```
                    ┌─────────┐
                    │ 00001   │   <- Quotient (Answer)
                    └─────────┘
                    ┌─────────┐
                    │ 01001101│   <- Dividend after subtraction
                    └─────────┘
                 ┌───────────┐
                 │ 00010100  │  <- Divisor
                 └───────────┘
```

(5) The dividend and the quotient are shifted left two bits,
    then a "1" is put in the quotient.

```
                    ┌─────────┐
                    │ 0000101 │   <- Quotient (Answer)
                    └─────────┘
                    ┌─────────┐
                    │ 01001101│   <- Dividend
                    └─────────┘
                 ┌───────────┐
                 │ 00010100  │  <- Divisor
                 └───────────┘
```

(6) Subtract the divisor from the dividend.
    The difference becomes the dividend.

```
                    ┌─────────┐
                    │ 0000101 │   <- Quotient (Answer)
                    └─────────┘
                    ┌─────────┐
                    │ 00100101│   <- Dividend after subtraction
                    └─────────┘
                 ┌───────────┐
                 │ 00010100  │  <- Divisor
                 └───────────┘
```

(7) Both dividend and quotient are shifted one bit again. Since
    the dividend is not less than the divisor, put "1" in the
    quotient.

```
                    ┌─────────┐
                    │ 00001011│   <- Quotient (Answer)
                    └─────────┘
                    ┌─────────┐
                    │ 00100101│   <- Dividend
                    └─────────┘
                 ┌───────────┐
                 │ 00010100  │  <- Divisor
                 └───────────┘
```

(8) Subtract the divisor from the dividend, the remainder is
    placed in the position of the dividend.

```
                    ┌─────────┐
                    │ 00001011│   <- Quotient (11)
                    └─────────┘
                    ┌─────────┐
                    │ 00010001│   <- Remainder (17)
                    └─────────┘
                 ┌───────────┐
                 │ 00010100  │  <- Divisor
                 └───────────┘
```

(9) If the remainder is not zero, the division process can be
    continued, but the result will contain fractions.

2. Division Program Design:

For the above algorithm, three memory locations are required to
store the dividend, divisor and quotient.

Example : Write a program to divide the 16-bit data in the DE
          register pair by the 16-bit data in the BC register pair.
          The result (quotient) must be stored in the HL register
          pair and the remainder in the DE register pair.

Solution: The register assignment has been given in the problem
          description. The HL register pair can be used as the
          working register for 16-bit arithmetic subtraction. Shift
          the 16-bit data in DE left one bit to the HL register
          pair. Compare HL with BC. If HL is not less than BC, then
          subtract BC from HL and the carry flag is set to 1
          automatically. Otherwise, no subtraction operation is
          performed and the carry flag will be 0 . Since the
          right-most bit of DE is now empty, the carry flag is then
          moved to this position.

The flowchart and the assembly language program are given below.

```
                    ╭─────────╮
                    │  START  │
                    ╰─────────╯
                         │
    ┌────────────────────────────────────────────┐
    │  Counter A = 16                             │
    └────────────────────────────────────────────┘
                         │
    ┌────────────────────────────────────────────┐
    │  Subtraction working                        │
    │  register HL = 0                            │
    └────────────────────────────────────────────┘
                         │
    ┌────────────────────────────────────────────┐
    │          CARRY <- 0                         │
    └────────────────────────────────────────────┘
                         │
    ┌────────────────────────────────────────────┐
    │  Connect HL & DE side by side,              │
    │  then shift left one bit                    │
    └────────────────────────────────────────────┘
                         │
    ┌────────────────────────────────────────────┐
    │  Compare HL with BC. If HL is               │
    │  larger than BC, then subtract BC           │
    │  from HL. Otherwise, no subtraction.        │
    └────────────────────────────────────────────┘
                         │
    ┌────────────────────────────────────────────┐
    │  If the subtraction is performed            │
    │  then set CARRY equal to 1 . Otherwise      │
    │  CARRY is equal to 0 . Move CARRY into      │
    │  the right-most bit of DE                   │
    └────────────────────────────────────────────┘
                         │
                    ◇ A <- A - 1 ◇
                         │
    ┌────────────────────────────────────────────┐
    │  Store the result                           │
    └────────────────────────────────────────────┘
                         │
                    ╭─────────╮
                    │   END   │
                    ╰─────────╯
```

45
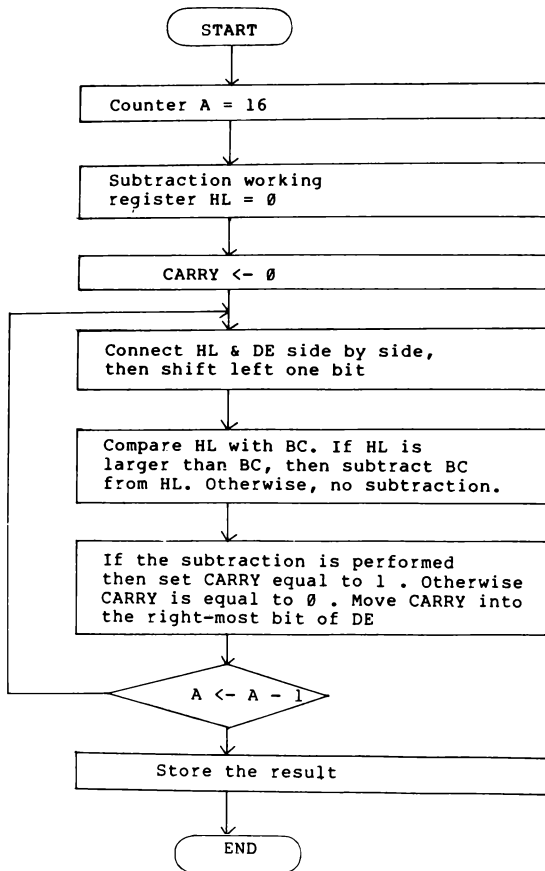
```
                          1 ; *** MPF-IP EXAMPLE PROGRAM 008 ***
                          2 ;16 BIT DIVISION ROUTINE
                          3 ;ENTRY:DIVIDEND IN 'DE'
                          4 ;      :DIVISOR IN 'BC'
                          5 ;EXIT :RESULT IN 'HL'
                          6 ;      :REMAINDER IN 'DE'
                          7 ;REG. CHANG  :AF,DE,HL
                          8
FB00    AF                9 DIV16   XOR A          ;CLEAR CARRY FLAG
FB01    67               10         LD H,A
FB02    6F               11         LD L,A         ;HL=0
FB03    3E10             12         LD A,16        ;A = 16,LOOP COUNTER
                         13
                         14 DV0   ;HL&DE 4 BYTE ROTATE LEFT 1 BIT
FB05    CB13             15         RL E              ;SHIFT LEFT,STORE PARTIAL RESULT
                         16                             IN BIT 0
FB07    CB12             17         RL D
FB09    ED6A             18         ADC HL,HL      ;ROTATE HL LEFT
                         19
                         20         IF HL GREAT THAN BC, SUBTRACT FROM BC
FB0B    ED42             21         SBC HL, BC     ;HL = HL - BC
FB0D    3001             22         JR NC,DV1
FB0F    09               23         ADD HL,BC      ;IF NEGATIVE,RESTORE HL
                         24
FB10    3F               25 DV1     CCF            ;PARTIAL RESULT IN CARRY FLAG
FB11    3D               26         DEC A
FB12    20F1             27         JR NZ,DV0
                         28
FB14    EB               29         EX DE HL
FB15    ED6A             30         ADC HL,HL      ;STORE LAST BIT OF RESULT
FB17    C9               31         RET
```

(1) Statement 10 and 11 of the program can be replaced by
    instruction LD HL,0 . But this instruction occupies 3 bytes
    memory and takes 10 clock cycles to execute. Instead, in
    this example, LD H,A and LD L,A are used (A is cleared to
    zero by statement 9). They occupy 2 bytes of memory and can
    be executed in 8 clock cycles.

(2) Addition and subtraction instructions can be used for "shift
    left" or "rotation" operations. In this example, instructions
    ADC HL,HL is identical with rotating the 16-bit data in HL
    pair left one bit (The bit moved to the carry flag comes from
    the leftmost bit of register D). The functions of the
    following instructions are described on the right-hand side.

```
ADD    A,A    ; Shift register A left one bit;
              or multiply A by 2.

ADC    A,A    ; Rotate A left one bit

ADD    HL,HL  ; Shift HL left one bit; or double it.

ADC    HL,HL  ; Rotate HL left one bit.

ADD    IX,IX  ; Shift IX left one bit; or double it.

ADD    IY,IY  ; Shift IY left one bit; or double it.
```

II. Illustrations of Experiments :

1. Load the above program into MPF-IP and then store it on audio tape.

2. Replace the last instruction (RET) in the above division subroutine by RST 38H and execute it. Record the obtained results in the following table.

Dividend	Divisor	Answer	Remainder	Check
8686H	0020H			
FFFFH	0003H			
5A48H	0142H			
0H	0142H			
1234H	0H			

3. Modify the above program such that the division process can be continued until a 16-bit fractional quotient is obtained.

4. Using the above program as a subroutine, write a main program to divide the data in RAM addresses FA00H - FA01H by the data in RAM addresses FA04H - FA05H. The result (quotient) must be stored in addresses FA00H - FA01H.

5. Write a program to divide the 4-byte data stored in addresses FA00H - FA03H by the 4-byte data stored in the memory address pointed to by the HL register pair. The result (quotient) must be in addresses FA00H - FA03H. The remainder must be stored stored in addresses FA04H - FA07H.

47

Experiment 8
Binary-to-BCD Conversion Program

Purposes:

       1. To understand the programming techniques of binary-to-BCD conversion and its applications.
       2. To understand the relation between subroutines and the main program.
       3. To familiarize the reader with the technique of program writing.

Time Required: 4 hours

I. Theoretical Background:

       1. Methods of binary-to-BCD conversion:

           There are several methods for binary-to-BCD conversion. The method given below will be very neat because it uses the DAA instruction. Two memory sections are assigned to store binary and BCD data, respectively. The memory addresses for BCD data are initially cleared to zero. The following process of shifting and checking data is repeated until all binary data bits are shifted left completely: shift the binary data left one bit, and its leftmost bit is automatically transferred to CARRY. The BCD data is then doubled and its rightmost bit-position is filled with the CARRY of binary data.
       The flowchart will be:

       (1) Preparation:
           Store the binary data in RAM with a starting address of FA00H. Assign register D as the byte counter for the binary data, and register E as byte counter for the BCD data. (Since the bit number of the BCD data may be larger than that of the binary data, the value of E is usually not less than that of D).

       (2) Clear the RAM section (starting address at FA08H) for the BCD data.

       (3) Shift the binary data (stored in RAM with starting address at FA00H) left one bit. The leftmost bit is automatically transferred to CARRY Flag.

       (4) Add CARRY to the BCD data (starting address at FA08H) and then double the BCD data.

48

(5) Check if all the bits of binary data have been shifted out of
            the original memory section. If not, repeat step (3). If yes,
            it is end of the program.

        The actual assembly language program is listed below.

```
                      1 ;*** MPF-IP EXAMPLE PROGRAM 001***
                      2 ;MULTIBYTE BINARY TO BCD CONVERTION
                      3 ;ENTRY:BINARY DATA STORED IN ADDR. 1A00H
                      4 ;EXIT :BCD DATA STORED IN ADDR. 1A08H
                      5 ;REGISTER USE
                      6 ; D  CONTAINS BYTE NUMBER OF BINARY DATA
                      7 ; E  CONTAINS BYTE NUMBER OF BCD DATA
                      8 ; A  BCD DATA WORKING REGISTER
                      9 ; B  LOOP COUNTER
                     10 ; C  BINARY BIT NUMBER
                     11
FB00                 12        ORG 0FB00H
                     13 BINBCD:
                     14 ;CLEAR BCD DATA BUFFER
FB00    AF           15 CLEAR  XOR A     ;A=0
FB01    43           16        LD B,E    ; B=BCD BYTE NUMBER
FB02    21081A       17        LD HL,1A08H
FB05    77           18 CLR    LD (HL),A ;CLEAR MEMORY
FB06    23           19        INC HL    ;NEXT ADDRESS
FB07    10FC         20        DJNZ CLR
                     21
                     22 ;CALCULATE BIT NUMBER
FB09    7A           23        LD A,D    ;A=BYTE NUMBER
FB0A    87           24        ADD A,A
FB0B    87           25        ADD A,A
FB0C    87           26        ADD A,A   ;A=A*8
FB0D    4F           27        LD  C,A   ;C=BIT NUMBER
                     28
                     29 LOOP:
                     30 ;SHIFT BINARY DATA LEFT
FB0E    2E00         31        LD L,0     ;HL=1A00=BINARY STARTING ADDRESS
FB10    42           32        LD B,D
FB11    CB16         33 SHLB   RL (HL)
FB13    23           34        INC HL
FB14    18FB         35        DJNZ SHLB
                     36
                     37 ;ADD CARRY & DOUBLE BCD DATA
FB16    2E08         38        LD L 8     ;HL=1A08=BCD STARTING ADDRESS
FB18    43           39        LD B,E
FB19    7E           40 BCDADJ LD A,(HL)
FB1A    8F           41        ADC A,A
FB1B    27           42        DAA
```

                                          49

```
FB1C    77      43      LD (HL),A
FB1D    23      44      INC HL
FB1E    10F9    45      DJNZ BCDADJ
                46
FB20    0D      47      DEC C
FB21    20EB    48      JR NZ,LOOP
FB23    FF      49      RST 38H
```

2. Assembly Language Programming Technique.

    (a) Multiply (or divide) a piece of binary data by a fixed number:

        Of course, the standard multiplication (or division) subroutine can be used to multiply (or divide) a binary number by a constant. However, a simple multiplication (or division) can be easily accomplished by shifting, additions or subtraction operations. For instance, in the above program, if the byte number of the binary data is known, then the bit number of the data can be easily obtained by multiplying the byte number by 8. In statements 22 – 27, instruction ADD A,A is used three times for multiplying the data in register D by 8 and then storing the result in register C. If the multiplier is not an exponential of 2, then addition or subtraction instructions must also be used.

Example: Multiply the data in D register by 6 and then store the result in register A. The program can be designed as follows.

```
        LD      A,D     ; A = D
        ADD     A,A     ; A = 2 * D
        ADD     A,D     ; A = 3 * D
        ADD     A,A     ; A = 6 * D
```

(b) Addressong method for memory on the same page:

        A memory address can be pointed to indirectly by a register pair (16 bits). To change a memory address pointed to by a required pair within the same page (each page contains 256 bytes), only a change in the low-order byte of the register pair is required. For instance, in the program listed above, the binary and BCD data are stored on the same page of memory (page 1AH). Since statement 1A assigns the contents of register H as 1AH, only a change in the contents of register L is required to change the pointed address in statements 31 and 38.

II. Example Experiments:

1. Load the binary-to-BCD conversion program listed in part I into MPF-IP and then store it on audio tape for future applications.

2. Test the above program:

   First, store the byte numbers of binary and BCD data in registers D and E, respectively. Next, load the binary data into RAM, with a starting address at FA00H. Record the obtained result and check if it is correct.

Binary	Hexadecimal	BCD	registers D & E
1000000000	0200H		D = 2, E = 3
	FFFFH		D = 2, E = 3
	10000H		D = 3, E = 4
	5A48347FH		D = 4, E = 6
	2^{32}		D = 8, E = 0AH
	2^{63}		D = 8, E = 0AH
	$2^{64} - 1$		D = 8, E = 0AH

3. Change the above program to a subroutine format (Replace the last instruction RST 38H by RET). Using this subroutine, write a program to convert the contents of the DE register pair into a BCD number and then store the converted BCD data in the HL register pair. The contents of the DE register pair will not be changed after the program execution. Test the program and write down the complete program in the blanks below.

4. Write a program to multiply the binary data in register E (<20H) by 7 and store the result in register A.

Experiment 9
BCD-to-Binary Conversion Program

Purpuses:

   1. To understand the methods of BCD-to-Binary conversion.
   2. To familiarize the reader with programming technique.

Time Required: 4 - 8 hours

I. Theoretical Background:

   1. Methods of BCD-to-Binary conversion:

   There are also several method for BCD-to-Binary conversion. In this experiment, the simple yet efficient method of shifting and checking is used. The RAMs used for storing the binary and BCD data are adjacent (in a row with the low-order digit on the right side). The BCD data is stored on the left-hand side and the converted binary data is stored on the right-hand side. The conversion procedure is given as follows.

   (1) Assign the bit number of the binary number as N for N program loops.

   (2) Shift the connected data right one bit.

   (3) Check the left-most bit of each digit (4 bits). If the checked bit is 1, then subtract 3 from the corresponding digit.

   (4) Repeat step (2) & (3) N times. The conversion process is then completed.

   2. Principle of the checking process :

   The real purpose of steps (2) & (3) of the above method is to divide the BCD number by 2 and put the remainder in the memory.The principle is illustrated in the following figure.
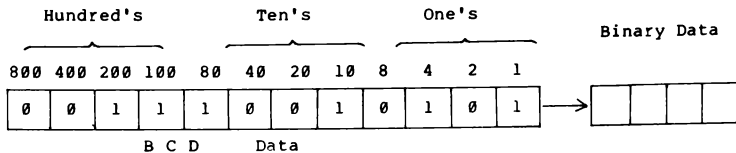
(1) Each BCD digit contains 4 bits. Shifting the 4 bits of a digit right one bit will divide this digit by 2. For instance, the leftmost digit of the ten's four bits represents 80 if it is "1". If this bit is shifted right, then it represents 40, that is, half of its original value.

(2) If a "1" is shifted from high a order digit to a lower order digit, the value is reduced to 5 (or 50, 500, ---, etc). However, the resulting BCD code will interprete this bit as 8 (or 80, 800, ---, etc). Thus 3 (or 30, 300, ---, etc) must be subtracted from the resulting BCD number.

(3) The conversion method can be illustrated by the following hand-calculation.
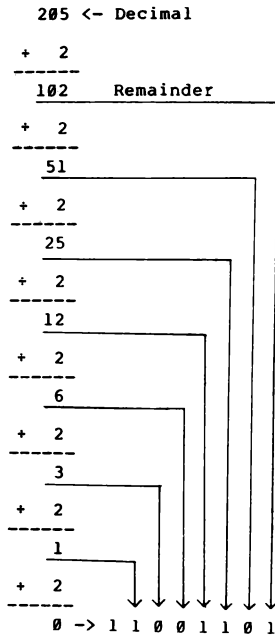
```
    205 <- Decimal          2 | 205  --- 1     BIT 0

  +   2                     2 | 102  --- 0     BIT 1
  ------
    102    Remainder        2 |  51  --- 1     BIT 2

  +   2                     2 |  25  --- 1     BIT 3
  ------
     51                     2 |  12  --- 0     BIT 4

  +   2                     2 |   6  --- 0     BIT 5
  ------
     25                     2 |   3  --- 1     BIT 6

  +   2                            1           BIT 7
  ------
     12                        1100   1101
                                C      D
  +   2
  ------
      6

  +   2
  ------
      3

  +   2
  ------
      1

  +   2
  ------
      0 -> 1 1 0 0 1 1 0 1
```

53

3. BCD-to-Binary conversion program:

Once the conversion method is decided, it is very easy to design the program. The following program can be used to convert 5-byte (or 10-digit) BCD data stored in RAM into 4-byte binary data. Since the largest value of 4-byte binary data is 4,294,967,295, the BCD number to be converted can not exceed this value. In RAM, the memory of addresses FA00H - FA03H are reserved for storing the binary data (lowest-order byte in FA00H). The memory of addresses FA04H - FA08H are assigned to store the BCD data. Sample programs for BCD-to-Binary conversion and Binary-to BCD conversion are listed below for reference.

```
                    1 ;*** MPF-IP EXAMPLE PROGRAM 007 ***
                    2
                    3 ; 10 DIGIT BCD TO BINARY CONVERSION
                    4 ; ENTRY: BCD DATA IN RAM FA04H TO FA08H
                    5 ;        : MAX. BCD DATA IS (4294967295)
                    6 ; EXIT : BINARY DATA IN RAM FA00H TO FA03H
                    7 ; REG. CHG : AF,HL,BC
 FB00               8         ORG FB00H
 FB00    0E20       9         LD C,32        ;PRESET CONV. LOOP = 32
                 FB DBLP:
                   11 ; DECIMAL DIVID BY 2
 FB02    0605      12         LD B,5         ;BCD BYTE COUNT = 5
 FB04    AF        13         XOR A          ;CLEAR CARRY FLAG
 FB05    2108FA    14         LD HL 0FA08H   ;HL POINT TO LEFT BYTEL
 FB08    7E        15 COR0    LD A,(HL)      ;TRANSFER DATA TO A REG.
 FB09    1F        16         RRA            ;ROTATE RIGHT
 FB0A    F5        17         PUSH AF        ;SAVE CARRY FLAG
                   18 ;* BCD DIVID CORRECTION
 FB0B    CB7F      19         BIT 7,A        ;TEST BIT 7
 FB0D    2802      20         JR Z,COR1'     ;NO CORRECT IF BIT 7 = 0
 FB0F    D630      21         SUB 30H        ;SUBTRACT FROM 30H IF BIT 7 = 1
 FB11    CB5F      22 COR1    BIT 3,A        ;TEST BIT 3
 FB13    2802      23         JR Z,COR2
 FB15    D603      24         SUB 3
                   25
 FB17    77        26 COR2    LD (HL),A      ;STORE TO MEMORY
 FB18    2B        27         DEC HL         ;NEXT BYTE
 FB19    F1        28         POP AF         ;RESTORE CARRY FLAG
 FB1A    18EC      29         DJNZ COR0'     ;DONE LOOP
                   30
                   31 ;ROTATE BINARY RIGHT
 FB1C    0604      32         LD B,4         ;BINARY BYTE = 4
 FB1E    CB1E      33 SHR4    RR (HL)
 FB20    2B        34         DEC HL
 FB21    18FB      35         DJNZ SHR4
                   36
 FB23    0D        37         DEC C
 FB24    20DC      38         JR NZ,DBLP
 FB26    C9        39         RET
```

```
                   40 *E
                   41 ;4 BYTE BINARY TO BCD CONVERSION
                   42 ;   ENTRY:BINARY DATA STORE IN ADDR.  FA00H TO FA03H
                   43 ;   EXIT :BCD DATA STORE IN ADDR. FA04H TO FA08H
                   44 ;   REG. CHANG : AF,BC,HL
                   45
                   46 BINBCD:
                   47 ;CLEAR BCD DATA BUFFER
FB27   2104FA      48         LD HL FA04H
FB2A   0605        49         LD B,5
FB2C   3600        50 CLEAR LD (HL),0
FB2E   23          51         INC HL
FB2F   10FB        52         DJNZ CLEAR
                   53
FB31   0E20        54         LD C,32
                   55 LOOP
                   56 ;SHIFT BINARY DATA LEFT
FB33   68          57         LD L,B        ; HL=FA00=BINARY STARTING ADDRESS
FB34   0604        58         LD B,4
FB36   AF          59         XOR A
FB37   CB16        60 SHLB  RL (HL)
FB39   23          61 INC HL
FB3A   10FB        62         DJNZ SHLB
                   63
                   64 ;ADD CARRY & DOUBLE BCD DATA
FB3C   0605        65         LD B,5
FB3E   7E          66 BCDADJ LD A,(HL)
FB3F   8F          67         ADC A,A
FB40   27          68         DAA
FB41   77          69         LD (HL),A
FB42   23          70         INC HL
FB43   10F9        71         DJNZ BCDADJ
                   72
FB45   0D          73         DEC C
FB46   20EB        74         JR NZ, LOOP
FB48   C9          75         RET
```

        0 ASSEMBLY ERRORS

II. Example Experiments:

1. Load the two subroutines for BCD-to-Binary anb Binary-to-BCD conversion into MPF-IP and then store them on audio tape for future application.

2. Replace the last instruction RET of the above subroutines by RST 38H so that control of the microcomputer MPF-IP will be returned to monitor after program execution. Load an arbitrary 5-byte BCD number in RAM address FA04H - FA08H. Convert this BCD data into binary data by using the above program. Check if the result is correct.

3. By a method similar to that described in part I (Theoretical Background), write a program to convert the 4-digit BCD data into binary data : The processing must be held within CPU registers and the result will be stored in the DE register pair.

	Assigned Decimal Number	Converted Binary Number	Re-converted Decimal Number
1			
2			
3			
4			
5			

4. Using the binary multiplication routine and the routines for conversion between binary and BCD data, write a program for decimal multiplication. The decimal multiplier and multiplicand must be stored in the HL and DE register pairs, respectively. The result must be stored in RAM addresses FA04H - FA08H. The data in HL and DE must be unchanged after program execution.

57

Experiment 10
Square-Root Program

Purposes:

     1. To understand how the microcomputer calculates the square root
       of a binary number.
     2. To practice microcomputer programming.

Time Required: 4 – 8 hours

I. Theoretical Background:

     1. Calculating square roots of binary numbers by hand:

     There are several methods for calculating the square root of a
binary number. The following method for hand-calculation can be
easily converted into a microcomputer program. This method is
illustrated by calculating the square root of 01010001 (or 81):

     (1) Each of the following blocks represents the position for
        storing data. The original binary number is stored in Y
        block, the number 01 is permanently stored in P block. X
        and R blocks are prestored with 0.

                      X   Y

X	Y
	01010001
	01000000

                      R   P

     (2) Subtract the number formed by the R & P blocks from the number
        formed by the X and Y blocks. If.the result is non-negative,
        then put 1 at the rightmost position in the R block and shift
        the original data in the R block left one bit. If the result is
        negative, then restore the original data in the X & Y blocks
        and shift the data in R left one bit. In this example, the
        result of subtraction is positive. Thus, the following
        result is obtained.

```
     X              Y
  ┌─────────┬─────────────┐
  │         │ 00010001    │
  └─────────┴─────────────┘

  ┌─────────┬───┬───┐
  │         │ 1 │ 01│
  └─────────┴───┴───┘
     R          P
```

(3) Shift the data in the X & Y blocks left two bits.

```
     X              Y
  ┌─────────┬────┬──────────┐
  │         │ 00 │ 01000100 │
  └─────────┴────┴──────────┘

  ┌─────────┬───┬───┐
  │         │ 1 │ 01│
  └─────────┴───┴───┘
     R          P
```

(4) Since the number in the X and Y blocks after the shift process
    is still less than that in the R and P blocks, thus the data
    in the R block must be shifted left one bit and a "0" is put
    in the rightmost position. The data in the X and Y blocks
    remains unchanged.

(5) Shift the data in the X and Y blocks left two bits.

```
     X              Y
  ┌─────────┬──────┬──────────┐
  │         │ 0001 │ 00010000 │
  └─────────┴──────┴──────────┘

  ┌─────────┬────┬───┐
  │         │ 10 │ 01│
  └─────────┴────┴───┘
     R           P
```

(6) The new data in the X and Y blocks is still less than the R
    and P block. Thus, shift the data in the R block left one bit
    again. An "0" is put in the rightomst position of the R block.
    The data in the X and Y blocks is also shifted left two bits.

```
     X              Y
    ┌──────────┬────┐
    │   000100 │ 01 │
    ├──────────┼────┤
    │      100 │ 01 │
    └──────────┴────┘
     R              P
```

(7) The number in the X and Y blocks is not less than that in the
    R and P blocks. Subtract the number in the R and P blocks from
    the number in the X and Y blocks. Shift the data in R left one
    bit and put a "1" in the left-most bit-position.

```
     X              Y
    ┌──────────┬────┐
    │   090000 │ 00 │
    ├──────────┼────┤
    │     1001 │ 01 │
    └──────────┴────┘
     R              P
```

(8) Shift data in the X and Y blocks left two bits. Since the
    the orginal data in the Y blocks has been shifted out compl-
    etly, the final result is given in the R block.

```
     X              Y
    ┌──────────────┐
    │  00000000    │
    ├──────────┬────┤
    │     1001 │ 01 │
    └──────────┴────┘
     R              P
```

(9) If the original data in the Y block is not the square root
    of some integral binary number, then the above method may be
    continued to find the fractional part of the square root.

60

2. Square root routine

The square root routine can be designed by the method described above. A subroutine for calculating the square root of a 16-bit piece of data is illustrated below.

Example: Find the square root of a 16-bit piece of data stored in the BC register pair. The calculation must be continued till the fractional part of the solution contains 8 bits. The integral part of the solution will be stored in register D, while the fractional part will be stored in register E.

Solution: The CPU registers are assigned as follows:

```
  X             Y
+-------+-----+-------+-----+
| H   L | A       C |
+-------+-----+-------+-----+

+-------+-----+-----+
| D   E | N   |
+-------+-----+-----+
  R           P
```
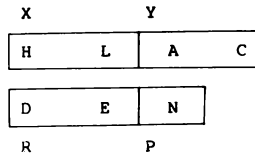
The original data is stored in registers A and C (Y block). The HL register pair is used as the working area of subtraction operation. The answer will be stored in the DE register pair (R block). The data in the P block is a fixed number, its left-most two bits are 01, i.e. the data in the P block may be written as 01000000B (40H). The program and its flowchart are given below.

61

```
        ┌─────────────┐
        │    ENTRY    │
        └─────────────┘
               │
        ┌─────────────┐
        │  A     B    │
        │  B     16   │
        └─────────────┘
               │
    ┌──────────────────────┐
    │  HL    DE    0       │
    └──────────────────────┘
               │
    ┌──────────────────────┐
    │  HLA   HLA   DEN     │
    └──────────────────────┘
               │
          ╱─────────╲
         ╱  CARRY?   ╲
         ╲           ╱
          ╲─────────╱
               │ C
    ┌──────────────────────┐
    │  HLA    HLA + DEN    │
    └──────────────────────┘
               │
    ┌──────────────────────┐
    │     CY     CY        │
    └──────────────────────┘
               │
    ┌──────────────────────┐
    │   ROTATE DE LEFT     │
    │    WITH  CARRY       │
    └──────────────────────┘
               │
    ┌──────────────────────┐
    │    SHIFT HLAC        │
    │   LEFT TWO BIT       │
    └──────────────────────┘
               │
          ╱─────────╲
         ╱  B   B  1 ╲
         ╲           ╱
          ╲─────────╱
               │ B=0
        ┌─────────────┐
        │    EXIT     │
        └─────────────┘
```
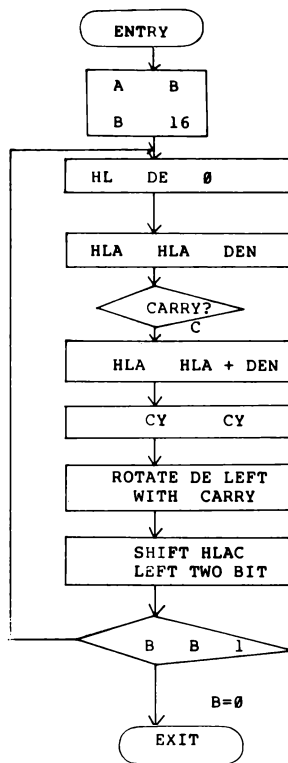
```
                        1 ; *** MPF-IP EXAMPLE PROGRAM 009 ***
                        2 ; 16 BIT SQUARE ROOT ROUTINE
                        3 ;ENTRY: BINARY DATA IN 'BC'
                        4 ;EXIT : RESULT IN 'D'(INTEGER)
                        5 ;                 'E' (FRACTION)
                        6 ;REG. CHANG.AF,BC,DE,HL
FB00   78               7 SQRT16 LD A,B          ;A&C = ENTRY DATA
FB01   0610             8        LD B,16         ;LOOP COUNTER
FB03   2100FB           9        LD HL,0         ;HL:WORKING AREA
FB06   54              10        LD D,H
FB07   5C              11        LD E,H          ;DE=0,RESULT PRESET TO 0
FB08   D640            12 SQ0    SUB 40H         ;A=A-40H,40H IS A FIXED DATA
FB0A   ED52            13        SBC HL,DE       ;HL=HL-DE
FB0C   3004            14        JR NC,SQ1       ;IS HL > DE ?
FB0E   C640            15        ADD A,40H
FB10   ED5A            16        ADC HL,DE       ;IF NOT, RESTORE A&HL
FB12   3F              17 SQ1    CCF             ;PARTIAL RESULT IN CARRY FLAG
FB13   CB13            18        RL E            ;STORE PARTIAL RESULT
FB15   CB12            19        RL D            ;   & SHIFT 'DE' (RESULT) LEFT
                       20        ;'HL.A C' 4 BYTE SHIFT LEFT TWICE
FB17   CB21            21        SLA C
FB19   17              22        RLA
FB1A   ED6A            23        ADC HL,HL
FB1C   CB21            24        SLA C
FB1E   17              25        RLA
FB1F   ED6A            26        ADC HL,HL
                       27
FB21   10E5            28        DJNZ SQ0        ;DONE LOOP
FB23   C9              29        RET

       0 ASSEMBLY ERRORS
```

II. Example Experiments:

   1. Load the above program onto MPF-IP and then store it in audio
    tape for future applications.

   2. Replace the last instruction (RET) by RST 38H. Prestore a 16-bit
    data in the BC register pair and then execute the square root
    program. Write down the result obtained.

Data Prestored in BC	Result of Program Execution	Check
0051H		
0000H		
FFFFH		
4000H		

3. Revise the above program such that it can be used for calculating the square root of a 32-bit piece of data. Store the original data in the BC and IX registers. The answer will be stoed in the DE register pair. Only the integial part of the square root is required.

4. Using the square root routine and binary multiplication routine, write a program for finding the absolute value of the vector formed by two mutual perpendicular vectors. The length of each vector component can be represented by an 8-bit binary number. These two numbers are stored in the H and L registers, respectively. The result of the program execution will be stored in register D.

$$(D) = \sqrt{ (H)^2 + (L)^2 }$$

64

Experiment 11
Introduction To MPF-IP Display

Purposes:

1. To understand how to use subroutines of the monitor program for designing display pattern.

2. To understand how the display is designed.

3. To understand the structure and characteristics of a matrix-form keyboard.

Time Required: 4 hours

I. Theoretical Background:
The structure and the characteristics of the MPF-IP display and keyboard are discussed in detail in Chapter 8 of the MPF-IP User's Manual. In Chapter 5, a number of useful monitor subroutines are listed for users' reference. In this experiment, the sample programs will use some of the monitor subroutines to control the display.

Example 1:

Display 'HELP US' until the SPACE key is pressed. Once the
SPACE key is pressed, the CPU will be halted and the red LED
lamp will get lighted. The program is shown as below:

```
                              EXP11.1
   LOC    OBJ CODE M STMT SOURCE STATEMENT


                         1   ;Display 'HELP US', HALT when
                         2   ;space key is pressed.
                         3   ;
   FB00                  4            ORG      0FB00H
   FB00    210EFB        5            LD       HL,HELP
   FB03    CD8608        6            CALL     PRTMES
   FB06    CD4602        7   DISP     CALL     SCAN
   FB09    FE20          8            CP       20H
   FB0B    20F9          9            JR       NZ,DISP
   FB0D    76           10            HALT
                        11   ;
   FB0E    20202020     12   HELP     DEFM     '        '
   FB14    48454C50     13            DEFM     'HELP US'
   FB1B    0D           14            DEFB     0DH
                        15   ;
                        16   PRTMES   EQU      0886H
                        17   SCAN     EQU      0246H
                        18            END
```

Example 2:

Flash 'HELP US'.
Because the execution time of the SCAN1 subroutine is 15.67 micro-second, to cause the display flashes 'HELP US' and then blank out alternately, the display buffer pointer - IX - should be changed after the SCAN1 has been executed 32 times. The program is as follows:

EXP11.2

LOC   OBJ CODE M STMT SOURCE STATEMENT

		1	;Flash	'HELP US':
FB00		2		ORG	0FB00H
FB00	CDB909	3		CALL	CLEAR
FB03	211CFB	4		LD	HL,HELP
FB06	CDCA09	5		CALL	MSG
FB09	DD212CFF	6		LD	IX,DISPBF
FB0D	21D06F	7		LD	HL,BLANK
FB10	E5	8		PUSH	HL
FB11	0620	9	LOOP1	LD	B,32
FB13	CD9B02	10	LOOP2	CALL	SCAN1
FB16	10FB	11		DJNZ	LOOP2
FB18	DDE3	12		EX	(SP),IX
FB1A	18F5	13		JR	LOOP1
		14	;		
FB1C	20202020	15	HELP	DEFM	' '
FB22	48454C50	16		DEFM	'HELP US'
FB29	0D	17		DEFB	0DH
		18	;		
		19	BLANK	EQU	6FD0H
		20	CLEAR	EQU	09B9H
		21	DISPBF	EQU	0FF2CH
		22	MSG	EQU	09CAH
		23	SCAN1	EQU	029BH
		24		END	

If you want to change the frequency at which the display flashes, you can achieve that by changing the times SCAN1 is called. If you intend to change the pattern to be displayed, you can alter the operand following the DEFM pseudo-op.

Example 3:

Display the ASCII code of the key pressed.
Fill "FF" into each memory location of the memory range from
FF2C to FF54, the display will blank out when the program is
executed.   When a key is pressed, the ASCII code of the key
pressed will be displayed.   The user may compare it with the
ASCII code table provided in the appendix of MPF-IP User's
Manual.

```
                            EXP11.3
    LOC    OBJ CODE M STMT SOURCE STATEMENT


                       1   ;Display ASCII code:
    FB00                2            ORG      0FB00H
    FB00    CDB909      3            CALL     CLEAR
    FB03    DD212CFF    4   LOOP     LD       IX,DISPBF
    FB07    CD4602      5            CALL     SCAN
    FB0A    CDB909      6            CALL     CLEAR
    FB0D    CD9A0A      7            CALL     HEX2
    FB10    18F1        8            JR       LOOP
                        9   ;
                       10   CLEAR    EQU      09B9H
                       11   DISPBF   EQU      0FF2CH
                       12   HEX2     EQU      0A9AH
                       13   SCAN     EQU      0246H
                       14            END
```

Example 4:

Display the position code of the key pressed. When a key is pressed, the position code of the key pressed will be displayed. The user may compare it with the position code table provided in the appendix of MPF-IP User's Manual. The program is as follows:

EXP11.4

LOC   OBJ CODE M STMT SOURCE STATEMENT

```
                       1   ;Display position code:
                       2   ;
FB00                   3           ORG     0FB00H
FB00   CDB909          4           CALL    CLEAR
FB03   DD212CFF        5           LD      IX,DISPBF
FB07   0603            6   LOOP1   LD      B,3
FB09   CD9B02          7   LOOP2   CALL    SCAN1
FB0C   30F9            8           JR      NC,LOOP1
FB0E   10F9            9           DJNZ    LOOP2
FB10   CD9B02          10  LOOP3   CALL    SCAN1
FB13   38FB            11          JR      C,LOOP3
FB15   CDB909          12          CALL    CLEAR
FB18   CD9A0A          13          CALL    HEX2
FB1B   CD9903          14          CALL    DECSP
FB1E   18E7            15          JR      LOOP1
                       16  ;
                       17  CLEAR   EQU     09B9H
                       18  DISP    EQU     0FF84H
                       19  DECSP   EQU     0399H
                       20  DISPBF  EQU     0FF2CH
                       21  HEX2    EQU     0A9AH
                       22  SCAN1   EQU     029BH
                       23          END
```

Exercises

1. Example 1:

   (a) If you want to change program in Example 1 so that the MPF-IP will display "HELP US" until the "Q" key is pressed, how will you change the program?
   (b) Try to write a program so that "NESBITT" will be displayed until the carriage return key is pressed. After the carriage return key is pressed, the red LED will get illuminated.

2. Example 2:

   (a) Save the program in Example 2 on casette tape.
   (b) Execute the program and examine the results.
   (c) Change the instruction "LD B,32" to "LD B,50H", and then execute the modified program. Examine the results. Explain why the results of the modified program are different from that of the original program?
   (d) Change the instruction "LD B,32" to "LD B,5", and then execute the modified program. Examine the results.
   (e) Modify the program by changing the contents of the memory so that "HELP US" is displayed for two seconds on the display and then blanked out for two seconds alternately.

3. Example 3:

   (a) Save the program in Example 2 on casette tape.
   (b) Execute the program and examine the results.
   (c) Why does the cursor "⋀" appear after the ASCII code? How to eliminate the cursor?

4. Example 4:

   (a) Save the program in Example 2 on casette tape.
   (b) Execute the program and examine the results.
   (c) Why is the displayed position code not followed by the cursor?
   (d) What does the position code mean?
   (You can figure this out by examining the detailed functions of the subroutines SCAN1 and SCAN2.)

Experiment 12
Fire-Loop Game

Purpose:

   1. To understand how to use a subroutine contained in the
      monitor program

   2. To familiarize the reader with programming techniques.

Time Required: 4 hours

I. Theoretical Background:

   1. Monitor Program:

      After the microcomputer is powered on, it will execute
      programs from the designated address. Besides some initia-
      lization task (e.g. setting 8255 or selecting I/O mode), a
      special software program called monitor is used to monitor
      the presence of data or commands from peripheral devices
      (e.g. a keyboard, an external switch, a button, a sensor,
      etc.) If no signal is monitored, then the scanning process
      continues (using the looping method to search) until a
      signal input is detected. If an input signal is detected,
      the input signal is then analyzed and the microcomputer
      jumps to the service routine to perform the job assigned by
      the input signal. After this service routine has been
      executed, the microcomputer returns to scan the peripheral
      devices.

      Since MPF-IP is a general-purpose microcomputer, it has a
      monitor. The main function of this monitor is to respond to
      key presses on the keyboard and to display necessary data.
      Tracing the monitor program will improve your programming
      skill.

   2. Fig. 12-1 is the flowchart of the Fire Loop.

```
                    ┌─────────┐
                    │  START  │
                    └─────────┘
                         │
        ┌────────────────▼──────────┐
        │  ┌──────────────────────┐ │
        │  │ HL ◄──── TABLE       │ │
        │  │ IX ◄──── DISPBF      │ │
        │  └──────────────────────┘ │
        │            │              │
        │  ┌──────────────────────┐ │
        │  │ Call CLEAR           │ │
        │  └──────────────────────┘ │
        │            │              │
        │         ◄──▼──►           │
   Yes ◄       Is REPEAT     ►      │
        │       code FF?            │
              └──────┬──────┘
                     │ No
        ┌────────────▼──────────┐
        │ Get back E and        │
        │ use E as an offest     │
        │ to calculate the       │
        │ location of the        │
        │ selected digit         │
        └────────────┬──────────┘
        ┌────────────▼──────────┐
        │ Get display pattern    │
        │ and put in display     │
        │ buffer                 │
        └────────────┬──────────┘
        ┌────────────▼──────────┐
        │ Call SCAN1             │
        └────────────┬──────────┘
               ◄─────▼─────►
  ─Yes ◄      Is C=02H?      ►
               └─────┬─────┘
                     │ no
        ┌────────────▼──────────┐
        │ Key pressed and        │
        │ Save key-code in C     │
        └────────────┬──────────┘
        ┌────────────▼──────────┐
        │ Test if the SPACE key  │
        │ is pressed, if yes,    │
        │ decrement HL to        │
        │ get the same data for  │
        │ display then it look   │
        │ like STOP, otherwise,  │
        │ get next pattern.      │
        └────────────┬──────────┘
                ┌────▼────┐
                │  STOP   │
                └─────────┘
```

Fig. 12-1 The Flowchart of Fire Loop

```
                              1  ;
                              2  ;Segment illuminates one by one until KEY-SPACE
                              3  ;is pressed.Any other key will resume looping
                              4  ;again.
                              5  ;
FB00                          6          ORG     0FB00H
FB00    213CFB                7  INI     LD      HL,TABLE
FB03    DD212CFF              8          LD      IX,DISPBF
FB07    CDB909                9  LOOP    CALL    CLEAR   ;Clear display buffer.
FB0A    5E                   10          LD      E,(HL)  ;Get the digit-select data.
FB0B    1C                   11          INC     E       ;Test repeat code:FF.
FB0C    28F2                 12          JR      Z,INI   ;If yes, go to INI.
FB0E    1D                   13          DEC     E       ;Otherwise, get back E.
FB0F    7B                   14          LD      A,E
FB10    87                   15          ADD     A,A
FB11    5F                   16          LD      E,A
                             17  ;The following 9 instructions get display
                             18  ;pattern and put into display buffer.
FB12    1600                 19          LD      D,0
FB14    DD19                 20          ADD     IX,DE
FB16    23                   21          INC     HL
FB17    7E                   22          LD      A,(HL)
FB18    DD7700               23          LD      (IX),A
FB1B    23                   24          INC     HL
FB1C    DD23                 25          INC     IX
FB1E    7E                   26          LD      A,(HL)
FB1F    DD7700               27          LD      (IX),A
FB22    DD212CFF             28          LD      IX,DISPBF
FB26    0603                 29          LD      B,SPEED ;Using B registr as SCAN1
                             30                          ;counter.
                             31  ;The following 4 instructions display the pattern
                             32  ;for B times.
FB28    CD9B02               33  LIGHT   CALL    SCAN1
FB2B    3801                 34          JR      C,NSCAN
FB2D    4F                   35          LD      C,A     ;Key pressed, save key_code
                             36                          ;in C. Note that, reg C̄ will
                             37                          ;not be changed until next
                             38                          ;key is pressed.
FB2E    10F8                 39  NSCAN   DJNZ    LIGHT
FB30    79                   40          LD      A,C
FB31    FE02                 41          CP      02H     ;Test KEY-SPACE of SCAN1.
FB33    2803                 42          JR      Z,STOP  ;If yes, decrement HL to get
                             43                          ;the same pattern for display.
                             44                          ;Then it looks like stop.
                             45                          ;Otherwise, get next pattern.
FB35    23                   46          INC     HL
FB36    23                   47          INC     HL
FB37    23                   48          INC     HL
FB38    2B                   49  STOP    DEC     HL
FB39    2B                   50          DEC     HL
FB3A    18CB                 51          JR      LOOP
                             52  ;
                             53  CLEAR   EQU     09B9H
                             54  DISPBF  EQU     0FF2CH
                             55  SCAN1   EQU     029BH
                             56  SPEED   EQU     3
                             57  ;
FB3C    07                   58  TABLE   DEFB    7       ;DIGIT 8
```

LOC	OBJ CODE	M STMT	SOURCE STATEMENT		
FB3D	FEFF	59	DEFW	0FFFEH	;SEG_a
FB3F	08	60	DEFB	8	;DIGIT 9
FB40	FEFF	61	DEFW	0FFFEH	;SEG_a
FB42	09	62	DEFB	9	;DIGIT 10
FB43	FEFF	63	DEFW	0FFFEH	;SEG_a
FB45	0A	64	DEFB	10	;DIGIT 11
FB46	FEFF	65	DEFW	0FFFEH	;SEG_a
FB48	0B	66	DEFB	11	;DIGIT 12
FB49	FEFF	67	DEFW	0FFFEH	;SEG_a
FB4B	0C	68	DEFB	12	;DIGIT 13
FB4C	FEFF	69	DEFW	0FFFEH	;SEG_a
FB4E	0C	70	DEFB	12	;DIGIT 13
FB4F	FDFF	71	DEFW	0FFFDH	;SEG_b
FB51	0C	72	DEFB	12	;DIGIT 13
FB52	FBFF	73	DEFW	0FFFBH	;SEG_c
FB54	0C	74	DEFB	12	;DIGIT 13
FB55	F7FF	75	DEFW	0FFF7H	;SEG_d
FB57	0B	76	DEFB	11	;DIGIT 12
FB58	F7FF	77	DEFW	0FFF7H	;SEG_g
FB5A	0A	78	DEFB	10	;DIGIT 11
FB5B	F7FF	79	DEFW	0FFF7H	;SEG_d
FB5D	09	80	DEFB	9	;DIGIT 10
FB5E	F7FF	81	DEFW	0FFF7H	;SEG_d
FB60	08	82	DEFB	8	;DIGIT 9
FB61	F7FF	83	DEFW	0FFF7H	;SEG_d
FB63	07	84	DEFB	7	;DIGIT 8
FB64	F7FF	85	DEFW	0FFF7H	;SEG_d
FB66	07	86	DEFB	7	;DIGIT 8
FB67	EFFF	87	DEFW	0FFEFH	;SEG_e
FB69	07	88	DEFB	7	;DIGIT 8
FB6A	DFFF	89	DEFW	0FFDFH	;SEG_f
FB6C	FF	90	DEFB	0FFH	;REPEAT CODE.
		91	END		

3. Further Expriments

(a) Load the above program into MPF-IP and then store it on audio
tape for future applications.
Test this program and record the display response.

(b) Write a program to make the Fire-Loop illuminate counterclockwise.

(c) Change the contents of FB32.   Then pressing space key
will not respond as before. Why?

(d) Change the contents of FB27 and the display will change. Why?

(e) Write a program that will cause the segments to move in a pattern
of your choice.

(f) Write a program to display "HELP US" for 20 secs, then
play the "Fire-Loop Game" 20 times.   Then display "HELP
US", and play fire-loop game over and over again.

Experiment 13
Stop-Watch

Purpose:

    1. To illustrate how to use monitor subroutines.

    2. To practise programming skills.

Time Required: 2 hours

  I. Theoretical Background:

  1. The object of this experiment is to design a 2/100
    second-based stop-watch. Actually, this is only roughly
    accurate.   The   total   execution time of the   SCAN1
    subroutine is 16.184 msec, plus the time required to
    perform the delay loop, results in the counter to be added
    by 2 each time all the instructions of the program are
    executed.   The accuracy varies with the system clock and
    the number of instructions used in the keyboard/display
    scan subroutine.

    2. The demonstration program calls two monitor subroutines SCAN1
    and HEX2 which are located at 29BH and 0A9AH respectively.

    3. The counting procedure is halted by depressing a key. This
    is done by checking the result of SCAN1 routine.

```
                    ┌─────────────┐
                    │    START    │
                    └─────────────┘
                           │
                           ▼
                  ┌──────────────────┐
                  │ CALL CLEAR       │
                  │ Subroutine       │
                  └──────────────────┘
                           │
                           ▼
                  ┌──────────────────┐
                  │ 1.Set display    │
                  │   buffer pointer │
                  │ 2.Load intial    │
                  │   display data   │
                  └──────────────────┘
                           │
                           ▼
                  ┌──────────────────┐
           ┌─────▶│      Loop        │◀─────┐
           │      └──────────────────┘      │
           │                │               │
           │                ▼               │
           │      ┌──────────────────┐      │
           │      │ Call SCAN1, scan │      │
           │      │   keyboard/disp  │    YES│
           │      └──────────────────┘      │
           │                │               │
           │                ▼               │
           │             ╱──────╲           │
           │            ╱ Is key ╲          │
           │            ╲pressed?╱──────────┘
           │             ╲──────╱
           │                │ NO
           │                ▼
           │      ┌──────────────────────┐
           │      │ 1. Increment the     │
           │      │        counter by 2  │
           │      │ 2. DECIMAL adjust the│
           │      │      counter contents│
           │      └──────────────────────┘
           │                │
           │                ▼
           │      ┌──────────────────┐
           │      │ 1. Call HEX2     │
           └──────│ 2. Store pattern │
                  │    into buffer   │
                  └──────────────────┘
```
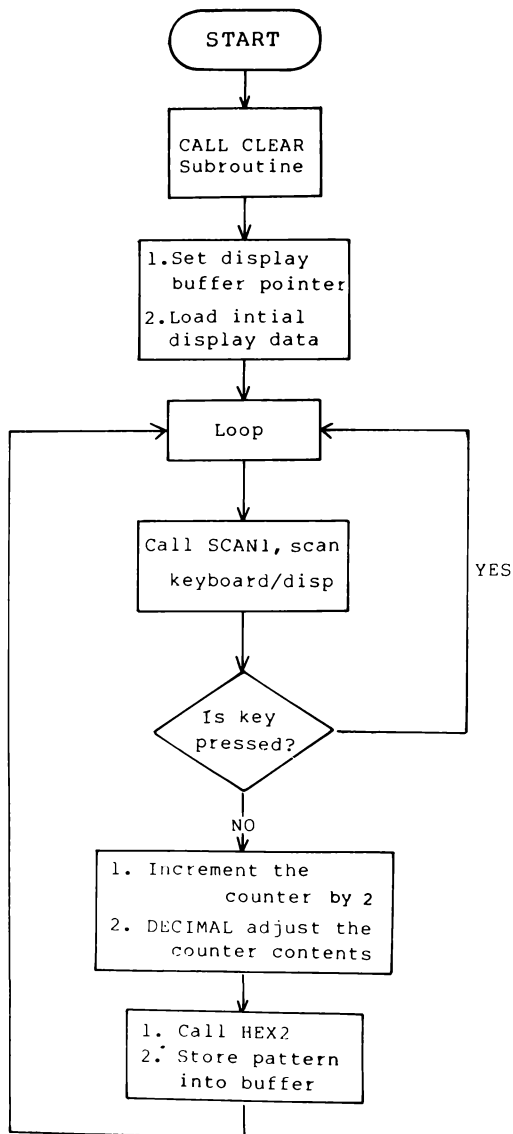
Fig   13-1   Flowchart of stop watch

```
                          1  ;STOP-WATCH
FB00                      2          ORG     0FB00H
FB00   DD212CFF           3          LD      IX,DISPBF ;Initial display pointer.
FB04   0E00               4          LD      C,0       ;Initial MIN in C.
FB06   110000             5          LD      DE,0      ;Initial SEC & 2/100 SEC
                          6                            ;in DE.
FB09   CD9B02             7  LOOP     CALL    SCAN1     ;Display for 15.6 m sec .
FB0C   30FB               8          JR      NC,LOOP   ;If any key pressed, then
                          9                            ;looping the same pattern.
                         10                            ;Otherwise, increment 2/100
                         11                            ;sec.
FB0E   CDB909            12          CALL    CLEAR
FB11   7B                13          LD      A,E
FB12   C602              14          ADD     A,2
FB14   27                15          DAA
FB15   5F                16          LD      E,A
FB16   7A                17          LD      A,D
FB17   CE00              18          ADC     A,0
FB19   27                19          DAA
FB1A   57                20          LD      D,A
FB1B   D660              21          SUB     60H       ;If SEC=60, then set SEC=0 and
                         22                            ;increment MIN by 1.
FB1D   2007              23          JR      NZ,BFUPDT
FB1F   1600              24          LD      D,0
FB21   79                25          LD      A,C
FB22   C601              26          ADD     A,1
FB24   27                27          DAA
FB25   4F                28          LD      C,A
                         29  BFUPDT:
FB26   2138FF            30          LD      HL,DISPBF+12
FB29   79                31          LD      A,C       ;Convert MIN to display
FB2A   CD3BFB            32          CALL    PA        ;format, and put them
                         33                            ;into display buffer.
FB2D   7A                34          LD      A,D       ;Convert SEC to display
FB2E   CD3BFB            35          CALL    PA        ;format, and put them
                         36                            ;into display buffer.
FB31   7B                37          LD      A,E       ;Convert 2/100 SEC to dis_
FB32   CD3BFB            38          CALL    PA        ;play format, and put them
                         39                            ;into display buffer.
FB35   0601              40  DELAY    LD      B,1
FB37   10FE              41          DJNZ    $
FB39   18CE              42          JR      LOOP
                         43  PA:
FB3B   2284FF            44          LD      (DISP),HL
FB3E   CD9A0A            45          CALL    HEX2
FB41   CD9903            46          CALL    DEC
FB44   2A84FF            47          LD      HL,(DISP)
FB47   23                48          INC     HL
FB48   23                49          INC     HL
FB49   C9                50          RET
                         51  CLEAR    EQU     09B9H
                         52  DISP     EQU     0FF84H
                         53  DISPBF   EQU     0FF2CH
                         54  DEC      EQU     0399H
                         55  HEX2     EQU     0A9AH
                         56  SCAN1    EQU     029BH
                         57          END
```

II. Illustration of the Experiments

(1) Load the program and GO!

(2) Press the $\boxed{\text{RESET}}$ $\boxed{\text{CONTROL}}$ and $\boxed{\text{SHIFT}}$ keys. Watch how the MPF-IP respond? Why?

(3) Note that the program will loop continuously. How can the execution of the program be interrupted?

(4) Users are encouraged to modify the program:

    a. Build a 1/10 second based stop watch.

    b. Display all zeros at the beginning, start the stop watch by depressing an arbitrary key or the user defined key.

    c. Build a stop key.

(5) Check the timing on the display with your watch for one minute. Perhaps, there is an error. Try to find the reasons for the error and note them.

Experiment 14
Designing a Clock Using Software

Purposes:

1. To practise calculating the clock cycle of a program.

2. To construct a software driven digital clock.

Time Required: 4 hours.

I. Theoretical Background:

1. This is an example of using the software delay to build a digital clock.

2. All the timing is based on the system clock, which is
3.579545 MHz ÷ 2 = 1.789772 MHz
So that 1 cycle is about 0.55873 micro-seconds.

3. The total number of cycles in ONE LOOP has been carefuly calculated.

4. The cycle count calculation is given as follow:

CLEAR   :   1041 T

MSG   :   4956 T

BFUPDT :   1747 T

SCAN1   :   28898 T

TMUPDT :   258 T

The total number of counts is 1800610.

and

0.56 usec x 1800610 ÷ 1.008 sec

5. Flowchart of clock

80

```
                    ┌─────────────┐
                    │    ENTRY    │
                    └─────────────┘
                           │
              ┌────────────▼────────────┐
              │ Set time buffer to      │
              │ 000000                  │
              └─────────────────────────┘
                           │
              ┌────────────▼────────────┐
              │ 1.Get time base         │
              │ 2.Call READLN           │
              └─────────────────────────┘
                           │
                      ◇────▼────◇
            YES ─────< Is A=20H? >
                      ◇─────────◇
                           │ NO
                      ◇────▼────◇
             ┌───────< Is A=0DH? >──── NO ──────┐
             │        ◇─────────◇                │
             │             │ YES                  │
             │  ┌──────────▼──────────┐           │
             │  │ Load time base      │           │
             │  │ to time buffer      │           │
             │  └─────────────────────┘           │
             │             │                      │
             │  ┌──────────▼──────────┐           │
             └─▶│      1 second       │◀──────────┤
                └─────────────────────┘           │
                           │                      │
              ┌────────────▼────────────┐         │
              │ 1. Clear display buffer │         │
              │ 2. Update display buffer│         │
              └─────────────────────────┘         │
                           │                      │
              ┌────────────▼────────────┐         │
              │ Time delay compensation │         │
              └─────────────────────────┘         │
                           │                      │
              ┌────────────▼────────────┐         │
              │ B  <--   61             │          │
              └─────────────────────────┘         │
                           │                      │
              ┌────────────▼────────────┐         │
              │ Call SCAN1  B<--B-1     │◀────┐   │
              └─────────────────────────┘     │   │
                           │                  │   │
                      ◇────▼────◇             │   │
                     < Is B=0? >──── NO ──────┘   │
                      ◇─────────◇                 │
                           │ YES                  │
              ┌────────────▼────────────┐         │
              │ Update the contents     │─────────┘
              │ of time buffer          │
              └─────────────────────────┘
```
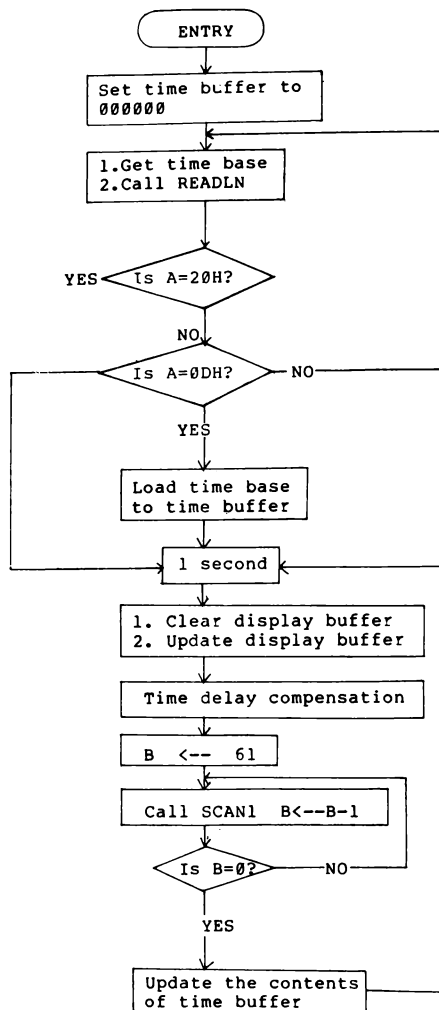
Fig. 13-1 Flowchart of clock

Time Update Flowchart

```
        ( TMUPDT )
            │
            ▼
┌───────────────────────┐
│ 1. Load timing data   │
│    pointer to DE       │
│ 2. Load timing maximum │
│    range table pointer │
│    to HL               │
└───────────────────────┘
            │
            ▼
┌───────────────────────┐
│ 1. Load counter B with 3 │
│    for hour, minute, second │
│    updating. 2. set carry bit │
│    for adding          │
└───────────────────────┘
            │
            ▼◄──────────────────────┐
┌───────────────────────┐           │
│ 1. Update timing, then │          │
│    check maximum range,│          │
│    if reached then carry │        │
│    propagated          │          │
│ 2. Point to next timing │         │
│    item    B<--B-1     │          │
└───────────────────────┘           │
            │                       │
            ▼                       │
         ╱ B=0? ╲ ──── NO ──────────┘
         ╲      ╱
            │
           YES
            │
            ▼
         ( RET )
```

82
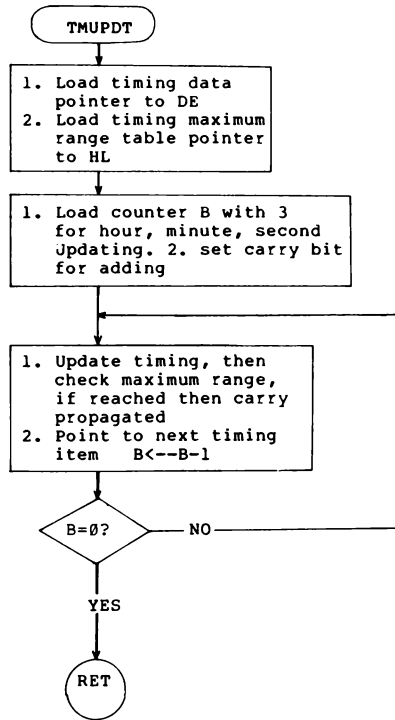
Display buffer Update Flowchart

```
        ( BFUPDT )
              |
┌─────────────────────────────┐
│ 1. Load display buffer      │
│    pointer to HL            │
│ 2. Load time buffer pointer │
│    to DE                    │
└─────────────────────────────┘
              |
     ┌─────────────────┐
     │ B  <--- 3       │
     │ Loop Counter    │
     └─────────────────┘
              |
              v
     ┌─────────────────┐
     │ 1. A <--(DE)    │
     │ 2. CALL HEX2    │
     │ 3. CALL DECSP   │
     │ 4. CALL SPACE1  │
     └─────────────────┘
              |
     ┌─────────────────┐
     │ HL <-- HL + 1   │
     │                 │
     │  B <-- B - 1    │
     └─────────────────┘
              |
              v
         < B = 0 ? >─── NO ──┐
              |              (back to loop)
             YES
              |
           ( RET )
```
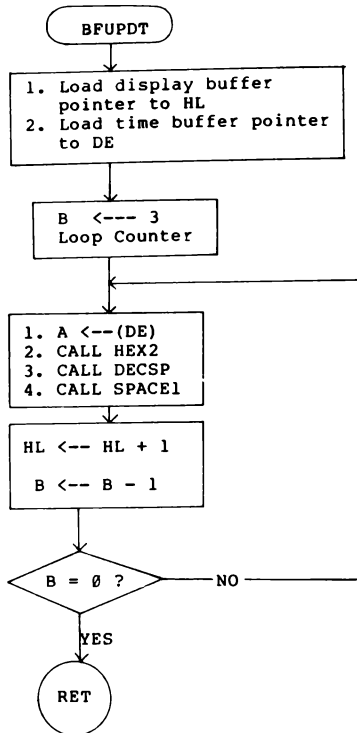
```
                          1   ;
                          2   ;A software driven digit clock.
                          3   ;
FB00                      4             ORG      0FB00H
                          5   START:
FB00   CDB909             6             CALL     CLEAR
FB03   0603               7             LD       B,3
FB05   2175FB             8             LD       HL,HOUR
FB08   3600               9   LOOP1     LD       (HL),0     ;This loop initial
FB0A   23                10             INC      HL         ;the time buffer.
FB0B   10FB              11             DJNZ     LOOP1
FB0D   217BFB            12             LD       HL,FORMAT
FB10   CDCA09            13             CALL     MSG
FB13   CDD409            14             CALL     READLN     ;Get time base.
FB16   2811              15             JR       Z,MAIN     ;If input line contain
                         16                                 ;only <CR>,then jump to
                         17                                 ;MAIN.Otherwise, get HOUR
                         18                                 ;, MIN and SEC.
FB18   CDDF08            19             CALL     CHKHEX
FB1B   0603              20             LD       B,3
FB1D   2175FB            21             LD       HL,HOUR
FB20   E5                22   LOOP2     PUSH     HL
FB21   CDE508            23             CALL     GETHL
FB24   E1                24             POP      HL
FB25   77                25             LD       (HL),A
FB26   23                26             INC      HL
FB27   10F7              27             DJNZ     LOOP2
                         28   MAIN:
FB29   CDB909            29             CALL CLEAR
FB2C   CD59FB            30             CALL     BFUPDT
FB2F   063E              31             LD       B,62
FB31   DD212CFF          32             LD       IX,DISPBF
FB35   CD9B02            33   LOOP3     CALL     SCAN1
FB38   10FB              34             DJNZ     LOOP3
FB3A   CD3FFB            35             CALL     TMUPDT
FB3D   18EA              36             JR       MAIN
                         37   ;
                         38   ;Time_buffer is updated here.
                         39   ;This routine takes almost the same time
                         40   ;in any condition.
                         41   ;
                         42   TMUPDT:
FB3F   217AFB            43             LD       HL,MAXTAB+2
FB42   1177FB            44             LD       DE,SEC
FB45   C5                45             PUSH     BC
FB46   0603              46             LD       B,3
FB48   37                47             SCF                 ;Set carry fag: force
                         48                                 ;add 1.
FB49   1A                49   TMINC     LD       A,(DE)
FB4A   CE00              50             ADC      A,0
FB4C   27                51             DAA
FB4D   12                52             LD       (DE),A
FB4E   96                53             SUB      (HL)       ;Compare with data in
                         54                                 ;MAX_TAB. If the result is
                         55                                 ;less than that,the follow-
                         56                                 ;ing loop will be null.
FB4F   3801              57             JR       C,COMPL
FB51   12                58             LD       (DE),A
```

84

```
FB52   3F          59  COMPL   CCF
FB53   2B          60          DEC     HL
FB54   1B          61          DEC     DE
FB55   10F2        62          DJNZ    TMINC
FB57   C1          63          POP     BC
FB58   C9          64          RET
                   65  ;
                   66  ;Display buffer is updated here.
                   67  ;This routine takes the same time in
                   68  ;any condition.
                   69  ;
                   70  BFUPDT:
FB59   2138FF      71          LD      HL,DISPBF+12 ;Set display buffer
                   72                                ;pointer.
FB5C   2284FF      73          LD      (DISP),HL
FB5F   1175FB      74          LD      DE,TMBF
FB62   0603        75          LD      B,3
FB64   1A          76  LOOP4   LD      A,(DE)
FB65   CD9A0A      77          CALL    HEX2
FB68   CD9903      78          CALL    DECSP
FB6B   CD950A      79          CALL    SPACE1
FB6E   13          80          INC     DE
FB6F   10F3        81          DJNZ    LOOP4
FB71   CD9903      82          CALL    DECSP
FB74   C9          83          RET
                   84  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                   85  TMBF:
FB75               86  HOUR    DEFS    1
FB76               87  MIN     DEFS    1
FB77               88  SEC     DEFS    1
FB78   24          89  MAXTAB  DEFB    24H
FB79   60          90          DEFB    60H
FB7A   60          91          DEFB    60H
FB7B   54494D45    92  FORMAT  DEFM    'TIME BASE='
FB85   0D          93          DEFB    0DH
                   94  CLEAR   EQU     09B9H
                   95  CURSOR  EQU     0A79H
                   96  DISP    EQU     0FF84H
                   97  MSG     EQU     09CAH
                   98  DECSP   EQU     0399H
                   99  CHKHEX  EQU     08DFH
                  100  DISPBF  EQU     0FF2CH
                  101  HEX2    EQU     0A9AH
                  102  READLN  EQU     09D4H
                  103  GETHL   EQU     08E5H
                  104  SCAN1   EQU     029BH
                  105  SPACE1  EQU     0A95H
                  106          END
```

II. Illustrations of the Experiments

1. Load the program

2. When the program is executed, the MPF-IP display will first
   display TIME BASE =⋏ , prompting the user to enter hour,
   minute, second. Hour, minute, second should be separated by
   the space key, and followed by a carriage return. When the
   carriage return key is pressed, the clock begins clicking.
   If the user does not key in time, the clock begins from 0
   hour, 0 minute, 0 second.

   Example: The clock is to begin counting from 10:30:00, then
            type in

   1) [G] [F] [B] [0] [0] [←——]   The display will show

       TIME BASE = ⋏

   2) Type in 10 30 00   The display will show

       10 30 00

   3) The clock begins counting.

3. Modify the program to improve the accuracy of the clock.
   Make the MPF-IP a 12-hour clock, and display "AM" or "PM"
   when the time is shown.

Experiment 15
Telephone Tone Simulation

Purposes:

1. To simulate a telephone ring.

2. To familiarize the reader with the application of 'tone' subroutine.

Time requirod: 4 hours.

I. Theoretical Background:

1. The telephone ring can be simulated as a repeating 1 second tone with 2 seconds silence.

2. This tone is a frequency shift keying signal modulated by two 20HZ square waves (half-period of 25 m sec).
The low & high states of this 20HZ signal correspond to 320HZ and 480HZ, so that it takes 8 & 12 cycles respectively.

3. In the following program, register C controls the frequency of the sound and register pair HL controls the length of the sound.

   a. Low frequency: C = 211, HL = 8, so the period is

   $(44 + 13 \times 211) \times 2 \times 0.56 = 3121$ micro-sec.

   frequency : $f = 1/3121 = 320Hz$

   length of the sound: 3121 micro-sec x 8 = 25m sec.

   b. High frequency: C = 140, HL = 12, so the period is

   $(44 + 13 \times 140) \times 2 \times 0.56 = 2087$ micro-sec

   frequency: $1/2087 = 480HZ.$

   length of the sound: 2087 micro-sec x 2 = 25m sec.

4. Output Circuit of tone
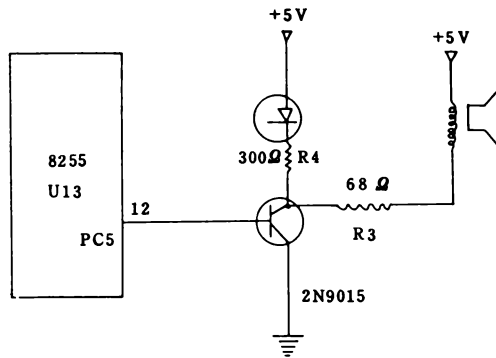
The output of the tone is sent via PC5 of 8255, 2N9015, R3, to the
speaker. When the voltage of PC5 is low, the transistor will
conduct; when the voltage of PC7 is high, the transistor will
nonconduct. By means of the transistor conducts and nonconducts,
the speaker will make sound.

5. Flowchart of Telephone Tone
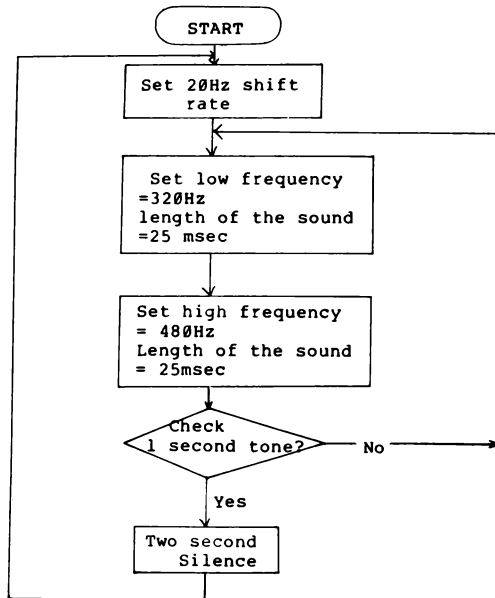
Flowchart of a telephone tone simulation

6. Telephone Tone Program

```
                         1  ;TELEPHONE TONE
FB00                     2          ORG     0FB00H
FB00     3E14            3  RINGBK  LD      A,20        ;20HZ freq shift rate
                         4                              ;so that 1 sec has 20 loops.
FB02     08             5  RING    EX      AF,AF'      ;Save to A'
FB03     0ED3           6          LD      C,211
FB05     210800         7          LD      HL,8
FB08     CD7408         8          CALL    TONE        ;320HZ, .25 m sec
FB0B     0E8C           9          LD      C,140
FB0D     210C00        10          LD      HL,12
FB10     CD7408        11          CALL    TONE        ;480HZ, 25 m sec
FB13     08            12          EX      AF,AF'      ;Retrieve from A'
FB14     3D            13          DEC     A           ;Decrement 1 count
FB15     20EB          14          JR      NZ,RING
FB17     0150C3        15          LD      BC,50000
FB1A     CD1FFB        16          CALL    DELAY       ;Silent 2 sec
FB1D     18E1          17          JR      RINGBK
                       18  ;Delay subroutine: (BC)*40 micro_sec
                       19  ;based on the 1.79 MHZ system clock
FB1F     E3            20  DELAY   EX      (SP),HL ;19 states
FB20     E3            21          EX      (SP),HL ;19
FB21     EDA1          22          CPI             ;16
FB23     E0            23          RET     PO      ;5
FB24     18F9          24          JR      DELAY   ;12
                       25  ;
                       26  TONE    EQU     0874H
                       27          END
```

II. Example and Practice Experiments

1. Load the above program into MPF-IP and then execute it.

2. Execute the program and listen to it. Does it sound like a
   telophone ring? If it doesn't, try to modify the frequency
   of the tone.

3. Try to simulate the telephone busy tone

   Hint: The busy tone can be simulated as follows: a repeating
         0.5 second 400HZ tone with 0.5 seconds of silence.

90

Experiment 16
Microcomputer Organ

Purposes:

    1. To enable the part of the Microprofessor to simulate an electronic organ.

    2. To familiarize the reader with the application of the keyboard -scaning routine.

Time Rquired: 4 hours

I. Theoretical Background:

    1. This experiment converts the MPF-IP into a simple electronic organ.

    2. When a key is pressed, the speaker will generate a tone corresponding to this key. This tone will not terminate until the key is released.

    3. Acceptable keyboard: key Ø — key F.

       If other keys are entered, the response is unpredictable.

    4. Key Mapping To Tones

Q	W	E	R	T	Y	U
$\dot{C}$	$\dot{D}$	$\dot{E}$	$\dot{F}$	$\dot{G}$	$\dot{A}$	$\dot{B}$

A	S	D	F	G	H	J
C	D	E	F	G	A	B

V	B	N	M
$\underset{.}{F}$	$\underset{.}{G}$	$\underset{.}{A}$	B

5. An octave ranges from a C to a B. The octave is divided into 5 full-
   tone and 2 half-tones, which equals to 12 half-tones, as follows:

   C #C D #D E F #F G #G A #A B

   The next octave is just twice the frequency of the current one,
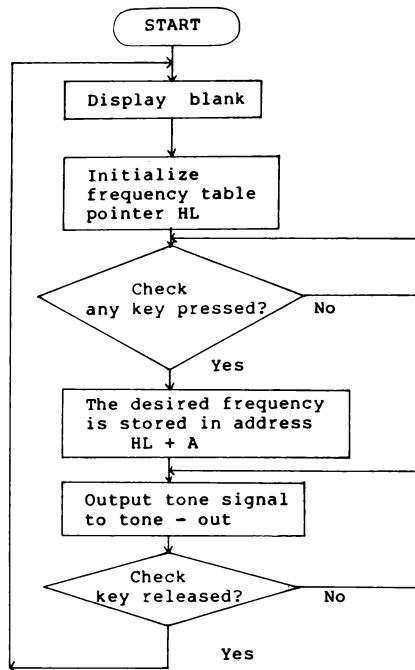   There is a logarithmic relationship between each half-tone.
   The frequency of each half-tone can be calculated by multiplying
   the last one by 2 ** (1/12), which is approximately 1.059.
   .
   For example, if the frequency of E is 503HZ, then the
   frequency of F is equal to

         503HZ x 1.059 = 532HZ.

Flowchart of orgran

```
                          1  ;MICROCOMPUTER ORGAN
FB00                      2          ORG     0FB00H
FB00   DD21D06F           3  START   LD      IX,BLANK
FB04   CD4D02             4          CALL    SCAN2   ;Display blank,return
                          5                          ;when any key is pressed
                          6                          ;A reg contains the key_code
FB07   2134FB             7          LD      HL,FREQTAB ;Based address of
                          8                             ;frequency table
                          9  ;
                         10  ;After routine SCAN2, A reg contain the code of the
                         11  ;key pressed. Using this code as table offset.
                         12  ;The desired frequency is stored in addresss HL+A
FB0A   D641              13          SUB     41H     ;Get offset of FREQTAB
FB0C   85                14          ADD     A,L     ;Add A to HL
FB0D   6F                15          LD      L,A
FB0E   3EDF              16          LD      A,0DFH
                         17  HALF:
FB10   D392              18          OUT     (KIN),A ;Output tone singal to
                         19                          ;tone out
FB12   46                20          LD      B,(HL)  ;Get the frequency from
                         21                          ;FREQTAB.
FB13   00                22  DELAY   NOP
FB14   00                23          NOP
FB15   00                24          NOP
FB16   10FB              25          DJNZ    DELAY
FB18   EE20              26          XOR     20H     ;Complement bit 5 of A.
FB1A   4F                27          LD      C,A
FB1B   AF                28          XOR A
FB1C   D380              29          OUT     (80H),A ;Activate the first 8 columns
                         30                          ;of the keyboard matrix.
FB1E   D381              31          OUT     (81H),A ;Activate next 8 columns of
                         32                          ;the keyboard matrix.
FB20   D382              33          OUT     (82H),A ;Activate the last 4 columns
                         34                          ;of the keyboard matrix.
FB22   DB92              35          IN      A,(KIN) ;Check whether this key is
                         36                          ;pressed or not. If any key
                         37                          ;is presssed, the corresponding
                         38                          ;matrix row input must be low.
FB24   F6F8              39          OR      0F8H
FB26   3C                40          INC     A       ;If A is 11111111B, increase A
                         41                          ;by one will make a zero and
                         42                          ;set zero flag.
FB27   3EFF              43          LD      A,0FFH
FB29   D380              44          OUT     (80H),A ;Disable the first 8 columns
                         45                          ;of the keyboard matrix and
                         46                          ;digits.
FB2B   D381              47          OUT     (81H),A ;Disable next 8 columns of
                         48                          ;the keyboard matrix and
                         49                          ;digits
FB2D   D382              50          OUT     (82H),A ;Disable the last 4 columns
                         51                          ;of the keyboard matrix and
                         52                          ;digits.
FB2F   79                53          LD      A,C     ;Restore A from reg C.
FB30   28CE              54          JR      Z,START ;If all key released,restart.
                         55                          ;Otherwise,continue this
                         56                          ;frequency.
FB32   18DC              57          JR      HALF
                         58
```

```
                          59   FREQTAB:
FB34   A8                 60          DEFB   0A8H   ;Key A
FB35   E0                 61          DEFB   0E0H   ;Key B
FB36   00                 62          DEFB   00     ;Key C
FB37   85                 63          DEFB   85H    ;Key D
FB38   42                 64          DEFB   42H    ;Key E
FB39   7E                 65          DEFB   7EH    ;Key F
FB3A   70                 66          DEFB   70H    ;Key G
FB3B   64                 67          DEFB   64H    ;Key H
FB3C   00                 68          DEFB   00     ;Key I
FB3D   59                 69          DEFB   59H    ;Key J
FB3E   00                 70          DEFB   00     ;Key K
FB3F   00                 71          DEFB   00     ;Key L
FB40   B2                 72          DEFB   0B2H   ;Key M
FB41   C8                 73          DEFB   0C8H   ;Key N
FB42   00                 74          DEFB   00     ;Key O
FB43   00                 75          DEFB   00     ;Key P
FB44   54                 76          DEFB   54H    ;Key Q
FB45   3E                 77          DEFB   3EH    ;Key R
FB46   96                 78          DEFB   96H    ;Key S
FB47   37                 79          DEFB   37H    ;Key T
FB48   2C                 80          DEFB   2CH    ;Key U
FB49   FB                 81          DEFB   0FBH   ;Key V
FB4A   4A                 82          DEFB   4AH    ;Key W
FB4B   00                 83          DEFB   00     ;Key X
FB4C   31                 84          DEFB   31H    ;Key Y
FB4D   00                 85          DEFB   00     ;Key Z
                          86   ;
                          87   BLANK   EQU   6FD0H
                          88   KIN     EQU   92H
                          89   SCAN2   EQU   024DH
                          90           END.
```

II. Example and Practice Expeiments

    1. Load the above program into MPF–IP and then store it on
       audio tape.

    2. Execute the program. When a key is pressed, the speaker
       will generate a tone coesponding to this key.

       Are these tones accurate?

    3. Try to play a song using this organ.

    4. Extend this program so that more keys of the keyboard can
       be used as input keys of the organ.

Experiment 17
Music Box Simulation

Purposes:

    1. To construct a music box.

    2. To familiarize the reader with programming techniques.

Time Required: 4 hours.

I. Theoretical Background:

    1. This experiment generates a song using programming techniques.

    2. There are two tables (frequency-table & song-table) in this program, which is described below:

        a. Frequency-table

        Every element of this table has 2 bytes, the 1st byte is the frequency parameter and the 2nd byte is the number of half-periods in a unit-time duration.

        One octave ranges from C to B. It is divided into 5 full-tones and 2 half-tones, which equals 12 half-tones, as follows:

        C #C D #D E F #F G #G A #A B

        The next octave is just twice the frequency of the current one, and there is a logarithmic relationship between each half-tone. So that the frequency of each half-tone can be calculated by multiplying the last tone by $2**1/12$, which is approximately 1.059.

        b. Song-Table:

        Each element of this table has 2 bytes:

        The 1st byte contains the code of the NOTE or REST or command of REPEAT or STOP. These codes are:

            bit 7 ---- STOP
            bit 6 ---- REPEAT
            bit 5 ---- REST
            bit 4-0 ---- NOTE CODE

The 2nd byte contains the counts of the unit-time, i.e. the NOTE length.

3. A flowchart of music box simulation is given below:

```
                    ┌─────────────┐
                    │   START     │
                    └──────┬──────┘
                           │
      ┌────────────────────┤
      │          ┌─────────▼─────────┐
      │          │    Initialize     │
      │          │ Song-Table pointer│
      │          │        IX         │
      │          └─────────┬─────────┘
      │                    │
      │          ┌─────────▼─────────┐◄──────────────────┐
      │          │  Get note data    │                   │
      │          └─────────┬─────────┘                   │
      │                    │                             │
      │               ╱────▼────╲                        │
      │              ╱   Check    ╲                       │
      │     Repeat  ╱ Stop, play or ╲    Stop             │
      └────────────◄    repeat?      ►──────┐             │
                    ╲              ╱        │             │
                     ╲────┬───────╱      ┌──▼───┐         │
                          │              │ Halt │         │
                          │ Play         └──────┘         │
                          │                               │
                ┌─────────▼─────────┐                     │
                │    Initialize     │                     │
                │ frequency table   │                     │
                │   pointer HL      │                     │
                └─────────┬─────────┘                     │
                          │                               │
                    ┌─────▼─────┐                         │
                    │ Tone out  │                         │
                    └─────┬─────┘                         │
                          │                               │
                 ┌────────▼────────┐                      │
                 │  Increment IX   │──────────────────────┘
                 └─────────────────┘
```

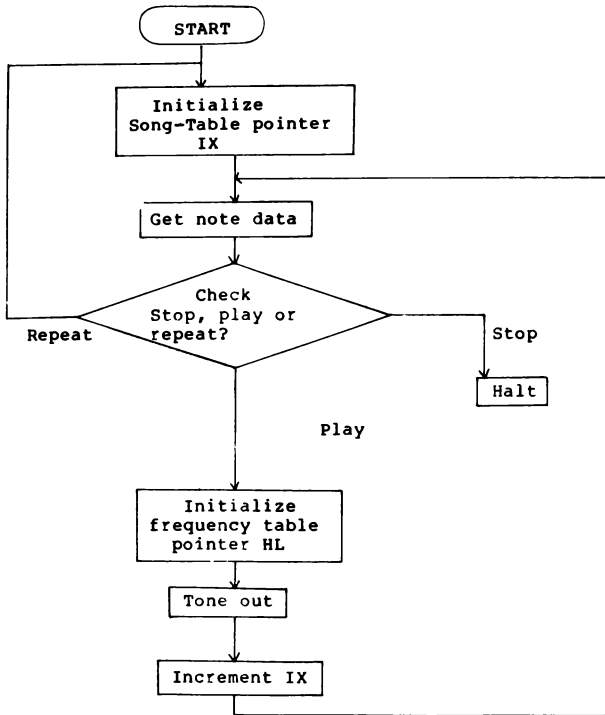Fig 17-1   Flowchart of music box simulation

```
                              1  ;
                              2  ;MUSIC BOX
                              3  ;
FB00                          4          ORG     0FB00H
FB00    DD2100F9              5  START   LD      IX,SONG  ;Initial song_table
                              6                           ;pointer.
FB04    DD7E00               7.  FETCH   LD      A,(IX)   ;Get note data
FB07    87                    8          ADD     A,A      ;Each note data have 2
                              9                           ;bytes.
FB08    3830                 10          JR      C,STOP   ;Stop?
FB0A    FA00FB               11          JP      M,START  ;Repeat?
FB0D    0E00                 12          LD      C,0      ;Reset tone_bit.
FB0F    CB77                 13          BIT     6,A      ;Reset?
FB11    2002                 14          JR      NZ,PLAY
FB13    CBE9                 15          SET     5,C      ;Set tone_bit
FB15    E63F                 16  PLAY    AND     3FH      ;Mask out note data.
FB17    213BFB               17          LD      HL,FRQTAB
FB1A    85                   18          ADD     A,L
FB1B    6F                   19          LD      L,A      ;Locate pointer in FRQTAB.
FB1C    5E                   20          LD      E,(HL)   ;Counts of loop per HALF_
                             21                           ;PERIOD delay.
FB1D    23                   22          INC     HL
FB1E    56                   23          LD      D,(HL)   ;Counts of HALF_PERIODS
                             24                           ;per UNIT_TIME.
FB1F    DD23                 25          INC     IX
FB21    DD6600               26          LD      H,(IX)   ;Counts of UNIT_TIME for
                             27                           ;this note.
FB24    3EFF                 28          LD      A,0FFH
                             29  ;
                             30  ;The following loop runs for one note or rest:
                             31  ;
FB26    6A                   32  TONE    LD      L,D
FB27    D392                 33  UNIT    OUT     (92H),A  ;Bit 5 is tone_out.
FB29    43                   34          LD      B,E
FB2A    00                   35  DELAY   NOP
FB2B    00                   36          NOP
FB2C    00                   37          NOP
FB2D    10FB                 38          DJNZ    DELAY
FB2F    A9                   39          XOR     C        ;If C=00H then reset.
                             40                           ;If C=00H then tone_out
FB30    2D                   41          DEC     L
FB31    20F4                 42          JR      NZ,UNIT
FB33    25                   43          DEC     H
FB34    20F0                 44          JR      NZ,TONE
                             45  ;
                             46  ;The current note has ended, increment pointer
                             47  ;next.
                             48  ;
FB36    DD23                 49          INC     IX
FB38    18CA                 50          JR      FETCH
FB3A    76                   51  STOP    HALT
                             52  ;
                             53  FRQTAB:
                             54  ;
                             55  ;1st byte:counts of DELAY loop per HALF_PERIOD.
                             56  ;2nd byte:counts of HALF_PERIOD per UNIT-TIME.
                             57  ;OCTAVE 3
FB3B    E118                 58          DEFW    18E1H    ;Code 00 , G
```

98

LOC	OBJ CODE	M STMT	SOURCE STATEMENT		
FB3D	D41A	59	DEFW	1AD4H	;Code 01 , #G
FB3F	C81B	60	DEFW	1BC8H	;Code 02 , A
FB41	BD1D	61	DEFW	1DBDH	;Code 03 , #A
FB43	B21E	62	DEFW	1EB2H	;Code 04 , B
		63	;OCTAVE 4		
FB45	AB20	64	DEFW	20A8H	;Code 05 , C
FB47	9F22	65	DEFW	229FH	;Code 06 , #C
FB49	9624	66	DEFW	2496H	;Code 07 , D
FB4B	8D26	67	DEFW	268DH	;Code 08 , #D
FB4D	8529	68	DEFW	2985H	;Code 09 , E
FB4F	7E2B	69	DEFW	2B7EH	;Code 0A , F
FB51	7723	70	DEFW	2E77H	;Code 0B , #F
FB53	7031	71	DEFW	3170H	;Code 0C , G
FB55	6A33	72	DEFW	336AH	;Code 0D , #G
FB57	6437	73	DEFW	3764H	;Code 0E , A
FB59	5E3A	74	DEFW	3A5EH	;Code 0F , #A
FB5B	593D	75	DEFW	3D59H	;Code 10 , B
		76	;OCTAVE 5		
FB5D	5441	77	DEFW	4154H	;Code 11 , C
FB5F	4F45	78	DEFW	454FH	;Code 12 , #C
FB61	4A49	79	DEFW	494AH	;Code 13 , D
FB63	464D	80	DEFW	4D46H	;Code 14 , #D
FB65	4252	81	DEFW	5242H	;Code 15 , E
FB67	3E57	82	DEFW	573EH	;Code 16 , F
FB69	3B5C	83	DEFW	5C3BH	;Code 17 , #F
FB6B	3762	84	DEFW	6237H	;Code 18 , G
FB6D	3467	85	DEFW	6734H	;Code 19 , #G
FB6F	316E	86	DEFW	6E31H	;Code 1A , A
FB71	2E74	87	DEFW	742EH	;Code 1B , #A
FB73	2C7B	88	DEFW	7B2CH	;Code 1C , B
		89	;OCTAVE 6		
FB75	2982	90	DEFW	8229H	;Code 1D , C
FB77	278A	91	DEFW	8A27H	;Code 1E , #C
FB79	5592	92	DEFW	9255H	;Code 1F , D
		93	;		

```
 94 ;1st byte,bit 7,6,5 & 4-0  : stop, repeat, rest, note
 95 ;         Code of stop   : 80H
 96 ;         Code of repeat : 40H
 97 ;         Code of rest   : 20H
 98 ;2nd byte, note lenth: counts of UNIT_TIME
 99 ;(N*0.77 sec).
100 ;
101 ;JINGLE BELL: (TRUNCATED)
```

```
F900                    102  SONG     ORG    0F900H
F900      09            103           DEFB   9
F901      04            104           DEFB   4
F902      09            105           DEFB   9
F903      04            106           DEFB   4
F904      09            107           DEFB   9
F905      06            108           DEFB   6
F906      20            109           DEFB   20H
F907      02            110           DEFB   2
F908      09            111           DEFB   9
F909      04            112           DEFB   4
F90A      09            113           DEFB   9
F90B      04            114           DEFB   4
F90C      09            115           DEFB   9
F90D      06            116           DEFB   6
```

```
FB3D    FEFF            59           DEFW   0FFFEH   ;SEG_a
FB3F    08              60           DEFB   8        ;DIGIT 9
FB40    FEFF            61           DEFW   0FFFEH   ;SEG_a
FB42    09              62           DEFB   9        ;DIGIT 10
FB43    FEFF            63           DEFW   0FFFEH   ;SEG_a
FB45    0A              64           DEFB   10       ;DIGIT 11
FB46    FEFF            65           DEFW   0FFFEH   ;SEG_a
FB48    0B              66           DEFB   11       ;DIGIT 12
FB49    FEFF            67           DEFW   0FFFEH   ;SEG_a
FB4B    0C              68           DEFB   12       ;DIGIT 13
FB4C    FEFF            69           DEFW   0FFFEH   ;SEG_a
FB4E    0C              70           DEFB   12       ;DIGIT 13
FB4F    FDFF            71           DEFW   0FFFDH   ;SEG_b
FB51    0C              72           DEFB   12       ;DIGIT 13
FB52    FBFF            73           DEFW   0FFFBH   ;SEG_c
FB54    0C              74           DEFB   12       ;DIGIT 13
FB55    F7FF            75           DEFW   0FFF7H   ;SEG_d
FB57    0B              76           DEFB   11       ;DIGIT 12
FB58    F7FF            77           DEFW   0FFF7H   ;SEG_g
FB5A    0A              78           DEFB   10       ;DIGIT 11
FB5B    F7FF            79           DEFW   0FFF7H   ;SEG_d
FB5D    09              80           DEFB   9        ;DIGIT 10
FB5E    F7FF            81           DEFW   0FFF7H   ;SEG_d
FB60    08              82           DEFB   8        ;DIGIT 9
FB61    F7FF            83           DEFW   0FFF7H   ;SEG_d
FB63    07              84           DEFB   7        ;DIGIT 8
FB64    F7FF            85           DEFW   0FFF7H   ;SEG_d
FB66    07              86           DEFB   7        ;DIGIT 8
FB67    EFFF            87           DEFW   0FFEFH   ;SEG_e
FB69    07              88           DEFB   7        ;DIGIT  8
FB6A    DFFF            89           DEFW   0FFDFH   ;SEG_f
FB6C    FF              90           DEFB   0FFH     ;REPEAT CODE.
        91           END
```

```
F90E    20              117              DEFB    20H
F90F    02              118              DEFB    2
F910    09              119              DEFB    9
F911    04              120              DEFB    4
F912    0C              121              DEFB    0CH
F913    04              122              DEFB    4
F914    05              123              DEFB    5
F915    04              124              DEFB    4
F916    07              125              DEFB    7
F917    04              126              DEFB    4
F918    09              127              DEFB    9
F919    08              128              DEFB    8
F91A    20              129              DEFB    20H
F91B    08              130              DEFB    8
F91C    80              131              DEFB    80H
                        132      ;
                        133      ;The following data are codes of song 'GREEN SLEEVES'
                        134      ;The user can put them at the SONG-TABLE, i.e. from
                        135      ;0F900 it will play until 'RESET' key is pressed.
                        136      ;
                        137      ;
                        138      ;F900    07 08 0A 10 0C 08 0E 10    10 04 0E 04 0C 10 09 08
                        139      ;F910    05 10 07 04 09 04 0A 10    07 08 07 10 06 04 07 04
                        140      ;F920    09 10 06 08 02 10 07 08    0A 10 0C 08 0E 10 10 04
                        141      ;F930    0E 04 0C 10 09 08 05 10    07 04 09 04 0A 08 09 08
                        142      ;F940    07‵08 06 08 04 08 06 08    07 10 20 08 11 10 11 08
                        143      ;
                        144      ;F950    11 10 10 04 0E 04 0C 10    09 08 05 10 07 04 09 04
                        145      ;F960    0A 10 07 08 07 10 06 04    07 04 09 10 06 08 02 10
                        146      ;F970    20 08 11 10 11 08 11 10    10 04 0E 04 0C 10 09 08
                        147      ;F980    05 10 07 04 09 04 0A 08    09 08 07 08 06 08 04 08
                        148      ;F990    06 08 07 18 20 10 40
                        149      ;
                        150              END
```