MICROPERIPHERALS

A Division of Pertec Computer Corporation

6741 VARIEL AVENUE - CANOGA PARK - CALIFORNIA 91303 - PHONE: (213) 348-1391

TELEX/TWX 910-494-2788

OPERATOR'S MANUAL

iCOM MACRO ASSEMBLER

iCOM MACRO ASSEMBLER

The iCOM Macro Assembler, used in conjunction with the iCOM Text Editor and FDOS-II on the iCOM Floppy Disk System provides the programmer the necessary tools for rapid, efficient software development.

The following text is intended as a guide and reference for those already experienced in Assembly Language programming.

Section I deals with the 8080 Assembly Language instructions required by the iCOM Assembler to produce executable object code.

Section II discusses the psuedo-instructions used by the iCOM Assembler to assist the programmer with his programming task.

Section III describes the macro capability of the iCOM Assembler, a feature which facilitates greater ease and efficiency in software development.

ERRATA SHEET

Make the following changes to the label examples:

Page	Was	Is
	Label	Label
44	SPRT:	SPRT
45	SPRT:	SPRT
46	name:	name
46	LOAD:	LOAD
47	PSMG:	PSMG
48	MDEC:	MDEC

iCOM MACRO ASSEMBLER

8080 ASSEMBLY LANGUAGE INSTRUCTIONS

CONTENTS

OPERATION OF THE iCOM MACRO ASSEMBLER

Execution of the Assembler is accomplished from FDOS by the command:

```
ASMB,INPUT-FILE-NAME,OUTPUT-FILE-NAME,PASS(Cr)
```

This command assumes (1) the diskette in drive Ø is a systems diskette, (2) the input file (INPUT-FILE-NAME) is present on a diskette in drive and consists of 8080 source code, and (3) sufficient space exists on the output diskette to accomodate the resulting object code file (OUTPUT-FILE-NAME) or source listing file, if either is requested.

PASS determines the type of output generated, as follows:

PASS VALUE	OUTPUT
2	Source listing on the list device
3	Executable object code in hexidecimal format on the output file
4	Both a source listing and object file
5	Source listing on the output file

SECTION I

8080 ASSEMBLY LANGUAGE INSTRUCTIONS FOR THE iCOM MACRO ASSEMBLER

The following statement format is required by the iCOM Macro Assembler to produce object code which is to be executed.

I. STATEMENT MNEMONICS

An instruction consists of up to four parts, or FIELDS. They are:

A. LABEL - (Field 1) The instruction's name. Used to reference the instruction.

B. CODE - (Field 2) Defines operation to be performed by instructions.

C. OPERAND - (Field 3) Provides address or data information needed by the CODE field.

D. COMMENT - (Field 4) Used for programmer's clarification, but is ignored by the Assembler. Using COMMENTS makes the operator's program more readable by describing each operation in the program.

EXAMPLE:

```
LABEL   CODE    OPERAND        COMMENT

START:  LXI     SP,STACK       ; Set stack pointer

STEND:  DB      20H            ; Create one byte data
                               ; constant
STACK:  EQU     Ø1FFFH         ; Top of stack

        MVI     A,2ØH          ; Set A to ASCII space
```

A. LABEL FIELD

Only alphanumeric characters, or one of the special characters listed below may be used as the first character of a label. The label may be up to five characters long, and a colon (:) must follow the last character.

EXAMPLE: Special Characters

```
@ At sign

? Question mark
```

EXAMPLE: Valid Label Fields

```
LABEL:

F14F:

@JMP:

?MVI:
```

Instructions which may not be used as LABELS are operation codes, pseudo-instructions and register names defined within the Assembler, (described in Section II).

If a label has more than five characters, only the first five will be recognized:

```
INSTRUCTION:  will be read as INSTR:
```

Labels serve as instruction addresses and cannot be duplicated. One instruction, however, may have more than one label, as follows:

EXAMPLE:

```
LBLØ1:                    ; First label

LBLØ2:    MVI  A,2ØH      ; Second label

          ADD  B

          JZ   LBLØ1

          ADD  C

          JNZ  LBLØ2
```

Each JMP instruction will cause program control to be transferred to the same MVI instruction.

B. CODE FIELD

The CODE field contains a code which identifies the machine operation to be performed. These are referred to as Op CODES and include such instructions as ADD, SUBTRACT, JUMP, etc. For example, the JUMP instruction is identified by the letters JUM. These letters must appear in the CODE field to identify the instruction as JUMP, and there must be at least one space following the CODE field.

```
EXAMPLE:     BEGIN:  JMP     START

INCORRECT:   BEGIN:  JMPSTART
```

C. OPERAND FIELD

The OPERAND field contains information used together with the CODE field to define the operation to be performed by the instruction. The OPERAND field may be absent or may consist of one item, or two items separated by a comma, depending upon the CODE field.

Four types of information that may be entered as items of an OPERAND field, may be specified in the following nine ways:

OPERAND FIELD

INFORMATION REQUIRED:	WAYS OF SPECIFYING INFORMATION:
Register	Hexadecimal Data
Register Pair	Decimal Data
Immediate Data	Octal Data
16 bit Memory Address	Binary Data
	Program Counter ($)
	ASCII Constant
	Labels assigned values
	Labels of instructions
	Expressions

WAYS OF SPECIFYING INFORMATION

1. HEXADECIMAL DATA--Each hexadecimal number must be followed by a letter "H" and must begin with a numeric digit.

   EXAMPLE:

```
LABEL   CODE  OPERAND   COMMENT
START:  MVI   A,0FFH    ; Load Register A with the hexi-
                        ; decimal value FF
```

2. DECIMAL DATA--Each decimal number may optionally be followed by the letter "D" (decimal), or may stand alone.

EXAMPLE:

```
   LABEL      CODE      OPERAND   COMMENT

   START:     MVI       A,255     ; Load register A with
                                  ; the value 255 (FF hex)
```

3. OCTAL DATA--Each octal number must be followed by one of the letters "O" or "Q".

EXAMPLE:

```
   START:     MVI       A,377O    ; Load accumulator with
                                  ; octal value 377
```

4. BINARY DATA--Each binary number must be followed by the letter "B".

EXAMPLE:

```
   START:     MVI  111B,11111111B   ; Load register A
                                    ; with FF
```

5. CURRENT PROGRAM COUNTER--Specified as the character $ and is equal to the address of the current instruction.

EXAMPLE:

```
   BEGIN:     JMP       $+9
```

The instruction causes program control to be transferred to the address 9 bytes beyond where the JMP instruction is loaded.

6. ASCII CONSTANT--One or more ASCII characters enclosed in single quotes. Two successive single quotes must be used to represent one single quote within an ASCII constant.

EXAMPLE:

```
   CHARS:     MVI       A,'*'     ; Load A register with
                                  ; 8-bit ASCII repre-
                                  ; sentation of an as-
                                  ; terisk

   CHARS:     DB        '*''*'    ; Set data string at
                                  ; CHARS to the ASCII
                                  ; representation of
                                  ; *'*
```

7. LABELS ASSIGNED VALUES--Labels that have been assigned a numeric value by the Assembler are built in and are always active.

LABEL assigned to	NUMERIC represent	REGISTER
B	0	B
C	1	C
D	2	D
E	3	E
H	4	H
L	5	L
M	6	Memory ref.
A	7	Register A

EXAMPLE: If DATUM has been equated to the hexadecimal F8H, all the following instructions load the D register with the hexidecimal value F8.

```
LABEL     CODE       OPERAND     COMMENT

A1:       MVI        D,DATUM
A2:       MVI        2,F8H
A3:       MVI        2,DATUM
```

8. LABELS OF INSTRUCTION--Labels which appear in the LABEL field of another instruction.

EXAMPLE:

```
BEGIN:    JMP        START       ; Jump to instruc-
                                 ; tion at START
START:    MVI        A,20H
```

9. EXPRESSIONS--Arithmetic and logical expressions involving data types 1 - 8 connected by the arithmetic operators + (addition), - (unary minus and subtraction), *(multiplication), /(division), MOD (modulo), logical operators NOT, AND, OR, XOR, SHR (shift right), SHL (shift left), and left and right parentheses.

. All operators treat their arguments as 16-bit quantities, and generate 16-bit quantities as their result.

. The operator + produces the arithmetic sum of its operands.

. The operator - produces the arithmetic difference of its operand when used as subtraction, or the arithmetic negative of its operand when used as unary minus.

- The operator * produces the arithmetic product of its operands.

- The operator / produces the arithmetic integer quotient of its operands, discarding any remainder.

- The operator MOD produces the integer remainder obtained by dividing the first operand by the second.

- The operator NOT complements each bit of its operand.

- The operator AND produces the bit-by-bit logical AND of its operands.

- The operator OR produces the bit-by-bit logical OR of its operands.

- The operator XOR produces the bit-by-bit logical EXCLUSIVE-OR of its operands.

- The SHR and SHL operators are linear shifts which cause the first operand to be shifted, either right or left, respectively, by the number of bit positions specified by the second operand. Zeroes are shifted into the high-order or low-order bits, respectively, of the first operand.

The programmer must insure that the result generated by any operation fits the requirements of the operation being coded. For instance, the operand of an MVI instruction must be an 8-bit value.

```
   EXAMPLE:              MVI   A,NOT 0
```

The example shown here is an invalid instruction because NOT 0 produces the 16-bit hexadecimal number FFFF.

```
   EXAMPLE:              MVI A,NOT 0 AND OFFH
```

This instruction is valid since the most significant 8 bits of the result are going to be 0, and the result can be represented in 8 bits.

An instruction mnemonic in parentheses is a legal expression of an optional field. Its value is the encoding of the instruction. The following example shows the instruction loading the hexadecimal address (16-bit address of the label LOC shifted right 8 bits) into the A register.

```
      LABEL     CODE        OPERAND

      LOC:      MVI         A,LOC SHR 8

   EXAMPLE:  Instruction will load the value
             18+(16/2)=18+8=26(IAH)

      SHIFT:    MVI         D,18+10H/2


   EXAMPLE:  Instruction defines a byte of value C3H
             (encoding of a JMP instruction) at
             location INSTR.

      INSTR:    DB          (JMP)
```

Operators cause expressions to be evaluated in this order:

1. Parenthesized expressions
2. *,/, MOD, SHL, SHR
3. +,- (unary and binary)
4. NOT
5. AND
6 OR XOR

<u>PARENTHESIZED EXPRESSIONS</u>-- The most deeply parenthesized expressions are evaluated first.

```
   EXAMPLE:  The instruction:  MVI A,(18+10H)/2
             Value to be loaded: (18+8)/2=13 into A
             register.
```

<u>MOD, SHL, SHR, NOT, AND, OR, XOR</u>-- All must be separated from their operands by at least one blank space.

```
   EXAMPLE:     MVI         A,DATUM AND0FH   is invalid

                MVI         A,DATUM AND   0FH is valid
```

The following four types of information may be specified using any number of all of the above nine data specifications.

1. A register, or code indicating memory reference, may utilize all of the above nine except the current program counter and labels of instruction to specify the register or memory reference. However, specifications must evaluate to a number, 0-7, as follows:

VALUE	REGISTER
0	B
1	C
2	D
3	E
4	H
5	L
6	Memory reference
7	A (Accumulator)

EXAMPLE:

LABEL	CODE	OPERAND
INS1:	MVI	REG4,0FFH
INS2:	MVI	4H,2EH
INS3:	MVI	8/2,2EH

If REG4 has been equated to 7, the above instruction will load the value FFH into register 4 (H register).

2. REGISTER PAIRS--Used to serve as the source or destination in a data operation.

REGISTER PAIR SPECIFICATION

Specification	Register Pair
B	Registers B & C
D	Registers D & E
H	Registers H & L
PSW	Program status word and Register A
SP	16-bit stack pointer register

3. IMMEDIATE DATA--To be used as a data item.

EXAMPLE:

```
LABEL   CODE  OPERAND    COMMENT

START:  MVI   C,DATA    ; Load the H register with the
                        ; value of DATA
```

4. 16-BIT ADDRESS--Label of another instruction in memory.

EXAMPLE:

```
LABEL  CODE  OPERAND   COMMENT

BEGIN  JMP   START     ; Jump to the instruction at
                       ; START

       JMP   OE800H    ; Jump to address E800H
```

D. COMMENT FIELD

A single rule governing this field is: comments must begin with the semicolon (;). Comment fields may also appear alone on a line.

EXAMPLE:

```
BEGIN: MVI   C,OADH    ; Comment here
                       ;
                       ; Another comment here
                       ;
```

II. <u>DATA STATEMENTS</u>

The three data statements are:

DB - Define Byte(s) of Data

DW - Define Word (2 bytes) of data

DS - Define Storage (bytes)

Data statements define the ways in which data is specified in, and received by, a program. An 8-bit byte contains one of the 256 possible combinations of zeros and ones, and any specified combination may be interpreted in several ways. The code 1FH may be interpreted, for instance, as a machine instruction (Rotate Accumulator Right Through Carry), as a hexadecimal value 1FH=31D, or as the bit pattern 00011111.

Arithmetic instructions assume that the data bytes upon which they operate are in two's complement format. The result of the operation performed is also two's complement.

A. <u>DB</u> -- Define Byte(s) of Data

FORMAT:	LABEL	CODE	OPERAND
	LABEL:	DB	String

"String" may be a list of:

1. Arithmetic and logical expressions using any of the arithmetic and logical operations which evaluate to 8-bit data quantities.

2. Strings of ASCII characters surrounded by quotation marks.

The 8-bit value of each expression, or the 8-bit ASCII representation of each character is stored in the next available byte of memory beginning with the byte addressed by LABEL. The most significant bit of each ASCII character is =0.

EXAMPLE:	INSTRUCTION	CODE	OPERAND	ASSEMBLED DATA (Hex)
	DATUM:	DB	OFFH	FF
	STRNG:	DB	'ABC'	414243
	NFVAL:	DB	-05H	FB

B. DW -- Define word (2 bytes) of data

```
FORMAT:       LABEL   CODE   OPERAND

              ADDRS:  DW     LIST
```

"List" is the expression(s) which evaluate to 16-bit data quantities. The least significant 8 bits of the expression are stored in the lower address memory byte (LABEL) and the most significant 8 bits are stored in the next higher addressed byte (LABEL+1). (It is standard procedure to reverse the order of the high and low address bytes when storing addresses in memory.)

EXAMPLE:	INSTRUCTION	CODE	OPERAND	ASSEMBLED DATA (Hex)
	ADDR1:	DW	START	00E8
	ADDR2:	DW	0F4C1H	C1F4
	ADDR3:	DW	4FC2H,4FC3H	C24FC34F

START is the label at E800H. Data are stored with the least significant 8 bits first.

C. DS -- Define Storage (bytes)

```
FORMAT:       VALU:        DS     exp
```

"exp" represents a n arithmetic or logical expression.

The value of exp specifies the number of memory bytes to be reserved for data storage. Data values are not assembled into these bytes; the programmer must not assume a data byte to be zero, for instance.

EXAMPLE: The first instruction is assembled VALUE, the second instruction is assembled at memory location VALUE+10.

LABEL	CODE	OPERAND	COMMENT
VALU:	DS	OAH	; Reserve next 10 bytes
	DS	150	; Reserve next 150 bytes

III. CARRY BIT INSTRUCTIONS

Carry bit instructions operate directly upon the carry bit, and each occupies one byte.

FORMAT:

	LABEL	CODE	OPERAND	MACHINE CODE
A. STC -- Set Carry	Label	STC	---	37
B. CMC -- Complement Carry	Label	CMC	---	3F

A. STC -- Set Carry
The carry bit is set to one. Condition bits affected is CARRY only.

B. CMC -- Complement Carry
If the carry bit is 0, it is set to 1. If the carry bit is 1, it is reset to 0. Condition bits affected are CARRY.

IV. SINGLE REGISTER INSTRUCTIONS

Single register instructions operate on a single register, or memory location. If a memory reference is specified, the memory byte addressed by the H and L registers is operated upon. The H register holds the most significant 8 bits of the address; the L register holds the least significant 8 bits of the address.

The four single register instructions are:

INR - Increment Register or Memory

DCR - Decrement Register or Memory

CMA - Complement Accumulator

DAA - Decimal Adjust Accumulator

FORMAT:

CODE	OPERAND	MACHINE CODE

A. INR -- Increment Register or memory.

The specified register or memory byte is incremented by one. Condition bits affected are ZERO, SIGN, PARITY, AUXILIARY CARRY.

EXAMPLE: If register A contains FEH, the instruction INR A will cause register A to contain FFH.

CODE	OPERAND	MACHINE CODE
INR	A	3C
INR	B	04
INR	C	0C
INR	D	14
INR	E	1C
INR	H	24
INR	L	2C
INR	M	34

B. DCR -- Decrement Register or Memory.

The specified register or memory byte is decremented by one.

CODE	OPERAND	MACHINE CODE
DCR	A	3D
DCR	B	05
DCR	C	0D
DCR	D	15
DCR	E	1D
DCR	H	25
DCR	L	2D
DCR	M	35

C. CMA -- Complement Accumulator.

Each bit of the contents of the accumulator is complemented, producing one's complement.

CODE	OPERAND	MACHINE CODE
CMA	---	2F

CMA (continued)

EXAMPLE: If the accumulator contains F0H, the instruction CMA will cause the accumulator to contain 0FH.

```
Accumulator = 1 1 1 1 0 0 0 0=F0H
_____________
Accumulator = 0 0 0 0 1 1 1 1=0FH
```

D. DAA -- Decimal Adjust Accumulator

The 8-bit hexadecimal number in the accumulator is adjusted to form two 4-bit binary-coded decimal digits by the following 2-step procedure.

FORMAT:

CODE	OPERAND	MACHINE CODE
DAA	---	27

1. If the least significant four bits of the accumulator represents a number greater than 9, or if the auxiliary carry bit is equal to one, the accumulator is incremented by six. If neither of these conditions exist, no incrementing occurs.

2. If the most significant four bits of the accumulator now represent a number greater than 9, or if the normal carry bit is equal to one, the most significant four bits of the accumulator are incremented by six. If neither of these conditions exist, no incrementing occurs.

If a carry out of the least significant four bits occurs during step #1, the auxiliary carry bit is set. If not, it is unaffected.

If a carry out of the most significant four bits occurs during step #2, the normal carry bit is set. If not, it is unaffected.

The DAA instruction is used when adding decimal numbers. It is the ONLY instruction whose operation is affected by the auxiliary carry bit. Condition bits which are affected are ZERO, SIGN, PARITY, CARRY and AUXILIARY CARRY.

EXAMPLE: If the accumulator contains 9BH, and both carry bits equal 0, the DAA instruction will operate in the following manner:

1. Bits 0-3 are greater than 9, and 6 is added to the accumulator. This addition will generate a carry out of the lower four bits, setting the auxiliary bit.

```
Bit Number:  7 6 5 4 3 2 1 0

Accumulator: 1 0 0 1 1 0 1 1 = 9BH
+ 6 =                0 1 1 0
             _______________

             1 0 1 0|0 0 0 1 = A1H
                    ↓
                    Auxiliary Carry = 1
```

2. Bits 4-7 are now greater than 9, and 6 is added to these bits. This addition will generate a carry-out of the upper four bits, setting the carry bit.

```
Bit Number:  7 6 5 4 3 2 1 0

Accumulator =1 0 1 0 0 0 0 1 = A1H

+ 6 =        0 1 1 0________

1)           0 0 0 0 0 0 0 1
↓
Carry = 1
```

The accumulator will now contain 1, and both carry bits will be = 1.

V. NOP INSTRUCTION

FORMAT:

LABEL	CODE	OPERAND	MACHINE CODE
Label	NOP	---	00

The NOP instruction occupies one byte. No operation occurs. Execution continues with the next sequential instruction, and no condition bits are affected.

VI. DATA TRANSFER INSTRUCTIONS

Data transfer instructions transfer data between registers or between memory and registers. The three data transfer instructions are:

MOV - Move

STAX- Store Accumulator

LDAX- Load Accumulator

A. MOV INSTRUCTION

One byte of data is moved from the source register to the destination register. The data replaces the contents of the destination register. The source register remains unchanged. No condition bits are affected.

FORMAT:

LABEL	CODE	OPERAND
Label	MOV	dst,src

EXAMPLE:

```
CODE  OPERAND  COMMENT

MOV   A,C      ; Move contents of the
               ; C register to the
               ; A register

MOV   M,A      ; Move contents of
               ; the A register to
               ; the memory byte
               ; specified by the
               ; contents of the
               ; H and L register
               ; pair
```

CODE	OPERAND	MACHINE CODE	CODE	OPERAND	MACHINE CODE
MOV	A,B	78	MOV	C,A	4F
MOV	A,C	79	MOV	C,B	48
MOV	A,D	7A	MOV	C,D	4A
MOV	A,E	7B	MOV	C,E	4B
MOV	A,H	7C	MOV	C,H	4C
MOV	A,L	7D	MOV	C,L	4D
MOV	A,M	7E	MOV	C,M	4E
MOV	B,A	47	MOV	D,A	57
MOV	B,C	41	MOV	D,B	50
MOV	B,D	42	MOV	D,C	51
MOV	B,E	43	MOV	D,E	53
MOV	B,H	44	MOV	D,H	54
MOV	B,L	45	MOV	D,L	55
MOV	BM	46	MOV	D,M	56

MOV INSTRUCTION (continued)

CODE	OPERAND	MACHINE CODE	CODE	OPERAND	MACHINE CODE
MOV	E,A	5F	MOV	L,A	6F
MOV	E,B	58	MOV	L,B	68
MOV	E,C	59	MOV	L,C	69
MOV	E,D	5A	MOV	L,D	6A
MOV	E,H	5C	MOV	L,E	6B
MOV	E,L	5D	MOV	L,H	6C
MOV	E,M	5E	MOV	L,M	6E
MOV	H,A	67	MOV	M,A	77
MOV	H,B	60	MOV	M,B	70
MOV	H,C	61	MOV	M,C	71
MOV	H,D	62	MOV	M,D	72
MOV	H,E	63	MOV	M,E	73
MOV	H,L	65	MOV	M,H	74
MOV	H,M	66	MOV	M,L	75

B. STAX STORE ACCUMULATOR

The contents of the accumulator are stored in the memory location addressed by registers B and C, or by registers D and E. No condition bits are affected.

FORMAT:

CODE	OPERAND	MACHINE CODE
STAX	B	02
STAX	D	12

C. LDAX LOAD ACCUMULATOR

The contents of the memory location addressed by registers B and C, or by registers D and E, replace the contents of the accumulator. No condition bits are affected.

LDAX	B	0A
LDAX	D	1A

VII. REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS

Eight instructions operate on the accumulator, using a byte fetched from another register or memory. These instructions occupy one byte. They are:

A. ADD - Add register or memory to accumulator
B. ADC - Add register or memory to accumulator with carry
C. SUB - Subtract register or memory from accumulator
D. SBB - Subtract register or memory from accumulator with borrow
E. ANA - Logical AND register or memory with accumulator
F. XRA - Logical EXCLUSIVE-OR register or memory with accumulator
G. ORA - Logical OR register or memory with accumulator
H. CMP - Compare register or memory with accumulator

These instructions operate on the accumulator using the byte in the register specified. If a memory reference is specified, the instructions use the byte in the memory location addressed by registers H and L. The H register holds the most significant 8 bits of the address, and the L register holds the least significant 8 bits of the address. The specified byte will remain unchanged by any of the instructions in this category. The result replaces the contents of the accumulator.

A. ADD - Add Register or Memory to Accumulator

The specified byte is added to the contents of the accumulator using two's complement arithmetic. Condition bits affected are: CARRY, SIGN, ZERO, PARITY, AUXILIARY CARRY.

FORMAT:

CODE	OPERAND	MACHINE CODE
ADD	A	87
ADD	B	80
ADD	C	81
ADD	D	82
ADD	E	83
ADD	H	84
ADD	L	85
ADD	M	86

B. ADC - Add Register or Memory to Accumulator with Carry

The specified byte plus the content of the carry bit is added to the contents of the accumulator. Condition bits affected are: CARRY, SIGN, ZERO, PARITY, AUXILIARY CARRY.

ADC	A	8F
ADC	B	88
ADC	C	89
ADC	D	8A
ADC	E	8B
ADC	H	8C
ADC	L	8D
ADC	M	8E

C. SUB - Subtract Register or Memory from Accumulator

The specified byte is subtracted from the accumulator using two's complement arithmetic. If there is no overflow out of the high-order bit position, (a borrow did not occur) the carry bit is set. If a borrow did occur, the carry bit is reset. Condition bits affected are CARRY, SIGN, ZERO, PARITY, AUXILIARY CARRY.

FORMAT:

CODE	OPERAND	MACHINE CODE
SUB	A	97
SUB	B	90
SUB	C	91
SUB	D	92
SUB	E	93
SUB	H	94
SUB	L	95
SUB	M	96

D. SBB - Subtract Register or Memory from Accumulator with Borrow

The carry bit is internally added to the contents of the specified byte. The value is then subtracted from the accumulator using two's complement arithmetic.

This instruction is used when performing subtractions. It adjusts the result of subtracting two bytes when a previous subtraction has produced a negative result. Condition bits affected are: CARRY, SIGN, ZERO, PARITY, and AUXILIARY CARRY.

SBB	A	9F
SBB	B	98
SBB	C	99
SBB	D	9A
SBB	E	9B
SBB	H	9C
SBB	L	9D
SBB	M	9E

E. ANA - Logical AND Register or Memory with Accumulator

The specified byte is logically ANDed, bit-by-bit, with the contents of the accumulator. The carry bit is reset to zero. The logical AND function of two bits is one if both the bits equal one.

ANA	A	A7
ANA	B	A0
ANA	C	A1
ANA	D	A2
ANA	E	A3
ANA	H	A4
ANA	L	A5
ANA	M	A6

F. XRA - Logical EXCLUSIVE-OR Register or Memory with Zero Accumulator

The specified byte is EXCLUSIVE-ORed, bit-by-bit with the contents of the accumulator. The carry bit is reset to zero. The EXCLUSIVE-OR function of two bits equals one if the values of the bits are different. Condition bits affected are CARRY, ZERO, SIGN and PARITY.

FORMAT:

CODE	OPERAND	MACHINE CODE
XRA	A	AF
XRA	B	A8
XRA	C	A9
XRA	D	AA
XRA	E	AB
XRA	H	AC
XRA	L	AD
XRA	M	AE

G. ORA - Logical OR Register or Memory with Accumulator

The specified byte is logically ORed, bit-by-bit, with the contents of the accumulator. The carry bit is reset to zero. The logical OR function of two bits equals zero if both the bits equal zero. Condition bits affected are CARRY, ZERO, SIGN and PARITY.

ORA	A	B7
ORA	B	B0
ORA	C	B1
ORA	D	B2
ORA	E	B3
ORA	H	B4
ORA	L	B5
ORA	M	B6

H. CMP - Compare Register or Memory with Accumulator

The specified byte is compared to the contents of the accumulator. The comparison is performed by internally subtracting the contents of the specified register from the accumulator, leaving both unchanged, and setting the condition bits according to the result. The zero bit is set if the quantities are equal, and reset if they are not. Since a subtract operation occurs, the carry bit is set if there is no overflow out of bit 7, indicating that the contents of the specified register are greater than the contents of the accumulator. If there is overflow out of bit 7, the carry bit is reset. If the two quantities to be compared differ in sign, the sense of the carry bit is reversed. Condition bits affected are CARRY, ZERO, SIGN, PARITY and AUXILIARY CARRY.

CMP	A	BF
CMP	B	B8
CMP	C	B9
CMP	D	BA
CMP	E	BB
CMP	H	BC
CMP	L	BD
CMP	M	BE

VIII. ROTATE ACCUMULATOR INSTRUCTIONS

When specifying instructions which rotate the contents of the accumulator, no memory locations, or other registers, are referenced. The four Rotate Accumulator Instructions are:

A. RLC - Rotate Accumulator Left
B. RRC - Rotate Accumulator Right
C. RAL - Rotate Accumulator Left through Carry
D. RAR - Rotate Accumulator Right through Carry

	FORMAT: CODE	OPERAND	MACHINE CODE
A. RLC - Rotate Accumulator Left	RLC	---	07
B. RRC - Rotate Accumulator Right	RRC	---	0F
C. RAL - Rotate Accumulator Left through Carry	RAL	---	17

A. RLC - Rotate Accumulator Left

The carry bit is set equal to the high-order bit of the accumulator. The contents of the accumulator are rotated one bit position to the left, and the high-order bit is transferred to the low-order bit position of the accumulator, and to the carry bit. Condition bit affected is CARRY.

B. RRC - Rotate Accumulator Right

The carry bit is set equal to the low-order bit of the accumulator. The contents of the accumulator are rotated one bit position to the right, with the low-order bit being transferred to the high-order bit position of the accumulator, and to the carry bit. Condition bit affected is CARRY.

C. RAL - Rotate Accumulator Left through Carry

The contents of the accumulator are rotated one bit position to the left. The high-order bit of the accumulator replaces the carry bit, and the carry bit replaces the low-order bit of the accumulator. Condition bit affected is CARRY.

D. <u>RAR</u> - Rotate Accumulator Right through Carry

The contents of the accumulator are rotated one bit position to the right. The low-order bit of the accumulator replaces the carry bit, and the carry bit replaces the high-order bit of the accumulator. Condition bit affected is CARRY.

FORMAT:

CODE	OPERAND	MACHINE CODE
RAR	---	1F

IX. REGISTER PAIR INSTRUCTIONS

Register pair instructions operate on pairs of registers. The eight register pair instructions are:

A. PUSH - Push Data onto Stack
B. POP - Pop Data off Stack
C. DAD - Double Add
D. INX - Increment Register Pair
E. DCX - Decrement Register Pair
F. XCHG - Exchange Registers
G. XTHL - Exhange Stack
H. SPHL - Load SP from H and L

A. PUSH - Push Data Onto Stack

The contents of the specified register pair are saved in the two bytes of memory indicated by the stack pointer (SP). The contents of the first register are saved at the memory address one less than the address indicated by the stack pointer. The contents of the second register are saved at the address two less than the address indicated by the stack pointer. If register pair PSW is specified, the first byte of information saved holds the settings of the five condition bits. Condition bits saved are CARRY, ZERO, SIGN, PARITY and AUXILIARY CARRY.

After the data has been saved, stack pointer is decremented by two. No condition bits are affected.

FORMAT:

CODE	OPERAND	MACHINE CODE
PUSH	PSW	F5
PUSH	B	C5
PUSH	D	D5
PUSH	H	E5

EXAMPLE:
Using S Z 0 A 0 P 1 C

Bit		
7	S	- State of sign bit
6	Z	- State of Zero bit
5	0	- Always 0
4	A	- State of Auxiliary Carry Bit
3	0	- Always 0
2	P	- State of Parity bit
1	1	- Always 1
0	C	- State of Carry bit

B. POP - Pop Data Off Stack

FORMAT:

CODE	OPERAND	MACHINE CODE
POP	PSW	F1
POP	B	C1
POP	D	D1
POP	H	E1

The contents of the specified register pair are restored from two bytes of memory indicated by the stack pointer SP. The byte of data at the memory address indicated by SP is loaded into the second register of the register pair. The byte of data at the address one greater than the address indicated by SP is loaded into the first register of the pair. If PSW is specified, the byte of data indicated by the contents of the stack pointer is used to restore the A register and the byte of data indicated by the contents of the s tack pointer, plus one, is used to restore the values of the five condition bits using the example described in (A) PUSH. The five condition bits affected are CARRY, ZERO, SIGN, PARITY and AUXILIARY CARRY. If pair PSW is not specified no condition bits are affected. After the data is restored, the stack pointer is incremented by two. C ondition bits affected are POP PSW.

C. DAD - Double Add

DAD	B	09
DAD	D	19
DAD	H	29
DAD	SP	39

The 16-bit number in the specified register pair is added to the 16-bit number held in the H and L registers, using two's complement arithmetic. The result replaces the contents of the H and L registers. If a carry out of bit 16 results from the DAD operation, the carry bit is set to 1. Condition bit affected is CARRY.

FORMAT:

D. <u>INX</u> - Increment Register Pair

The 16-bit number is held in the specified register pair and is incremented by one. No condition bits are affected.

CODE	OPERAND	MACHINE CODE
INX	B	03
INX	D	13
INX	H	23
INX	SP	33

E. <u>DCX</u> - Decrement Register Pair

The 16-bit number held in the specified register pair is decremented by one. No condition bits are affected.

CODE	OPERAND	MACHINE CODE
DCX	B	0B
DCX	D	1B
DCX	H	2B
DCX	SP	35

F. <u>XCHG</u> - Exchange Registers

CODE	OPERAND	MACHINE CODE
XCHG	---	EB

The 16 bits of data held in the H and L registers are exchanged with the 16 bits of data held in the D and E registers. No condition bits are affected.

G. <u>XTHL</u> - Exchange Stack with H and L

CODE	OPERAND	MACHINE CODE
XTHL	---	E3

The contents of the L register are exchanged with the contents of the memory byte whose address resides in the stack pointer. The contents of the H register are exchanged with the contents of the memory byte whose address is one greater than the one held in the stack pointer. No condition bits are affected.

H. <u>SPHL</u> - Load SP from H and L

CODE	OPERAND	MACHINE CODE
SPHL	---	F9

The 16 bits of data held in the H and L registers replace the contents of the stack pointer. The contents of the H and L registers are not changed. No condition bits are affected.

X. IMMEDIATE INSTRUCTIONS

The remaining iCOM assembly language instructions perform operations using a byte-, or bytes, of data which are part of the instruction itself. Listed below are those ten instructions and their definitions.

Instructions in this section occupy two or three bytes of data. The LXI occupies 3 bytes, and the remaining instructions occupy two bytes. Except for the LXI and MVI instructions, all instructions in this section operate on the accumulator or the memory byte specified by the contents of the H and L register pair, using one byte of immediate data. The result replaces the contents of the accumulator.

The ten IMMEDIATE instructions are:

A. LXI - Load Register Pair Immediate
B. MVI - Move Immediate Data
C. ADI - Add Immediate to Accumulator
D. ACI - Add Immediate to Accumulator with Carry
E. SUI - Subtract Immediate From Accumulator
F. SBI - Subtract Immediate from Accumulator with Borrow
G. ANI - AND Immediate with Accumulator
H. XRI - EXCLUSIVE-OR Immediate with Accumulator
I. ORI - OR immediate with Accumulator
J. CPI - Compare Immediate with Accumulator

A. LXI - Load Register Pair Immediate

FORMAT:

CODE	OPERAND	MACHINE CODE
LXI	B,data	01
LXI	D,data	11
LXI	H,data	21
LXI	SP,data	31

The LXI instruction operates on the specified register pair, using two bytes of immediate data. The third byte of the instruction (most significant 8 bits of the 16-bit immediate data) is loaded into the first register of the specified pair and the second byte of the instruction (the least significant 8 bits of the 16-bit immediate data) is loaded into the second register of the specified pair. If SP is specified as the register pair, the second byte of the instruction replaces the least significant 8 bits of the stack pointer, and the third byte of the instruction replaces the most significant 8 bits of the stack pointer. No condition bits are affected.

The immediate data for LXI is a 16-bit quantity. All other immediate instructions require an 8-bit data value.

FORMAT:

	CODE	OPERAND	MACHINE CODE
B. <u>MVI</u> - Move Immediate Data	MVI	A,data	3E
	MVI	B,data	06
	MVI	C,data	0E
	MVI	D,data	16
	MVI	E,data	1E
	MVI	H,data	26
	MVI	L,data	2E
	MVI	M,data	36

The MVI instruction operates on the specified register using one byte of immediate data. If a memory reference is specified, the instruction operates on the memory location addressed by registers H and L. The H register holds the most significant 8 bits and the L register holds the least significant 8 bits of the address.

The byte of immediate data is stored in the specified register, or memory byte. No condition bits are affected.

	CODE	OPERAND	MACHINE CODE
C. <u>ADI</u> - Add Immediate to Accumulator	ADI	data	C6

The byte of immediate data is added to the contents of the accumulator using two's complement arithmetic. Condition bits which are affected are CARRY, SIGN, ZERO, PARITY and AUXILIARY CARRY.

	CODE	OPERAND	MACHINE CODE
D. <u>ACI</u> - Add Immediate to Accumulator with Carry	ACI	data	CE

The byte of immediate data is added to the contents of the accumulator, plus the contents of the carry bit. Condition bits affected are CARRY, SIGN, ZERO, PARITY, and AUXILIARY CARRY.

	CODE	OPERAND	MACHINE CODE
E. <u>SUI</u> - Subtract Immediate from Accumulator with Borrow	SUI	data	D6

The byte of immediate data is subtracted from the contents of the accumulator using two's complement arithmetic. In this subtraction operation, the carry bit is set, indicating a borrow, provided there is no overflow from the high-order bit position. It is reset if there is an overflow. Condition bits affected are CARRY, SIGN, ZERO, PARITY, and AUXILIARY CARRY.

FORMAT:

		CODE	OPERAND	MACHINE CODE
F.	<u>SBI</u> - Subtract Immediate from Accumulator with Borrow	SBI	data	DE
G.	<u>ANI</u> - AND Immediate with Accumulator	ANI	data	E6
H.	<u>XRI</u> - EXCLUSIVE-OR Immediate with Accumulator	XRI	data	EE
I.	<u>ORI</u> - OR Immediate with Accumulator	ORI	data	F6

F. <u>SBI</u> - Subtract Immediate from Accumulator with Borrow

The carry bit is internally added to the byte of immediate data. The value is then subtracted from the accumulator using two's complement arithmetic. The SBI instruction is best utilized when performign multibyte subtractions. In this subtraction operation, the carry bit is set if there is no overflow from the high-order position, and it is reset if there is an overflow. Condition bits affected are CARRY, SIGN, ZERO, PARITY, and AUXILIARY CARRY.

G. <u>ANI</u> - AND Immediate with Accumulator

The byte of immediate data is logically ANDed with the contents of the accumulator. The carry bit is reset to zero. Condition bits affected are CARRY, ZERO, SIGN and PARITY.

H. <u>XRI</u> - EXCLUSIVE-OR Immediate with Accumulator

The byte of immediate data is EXCLUSIVE-ORed with the contents of the accumulator. The carry bit is set to zero. Condition bits affected are CARRY, ZERO, SIGN and PARITY.

I. <u>ORI</u> - OR Immediate with Accumulator

The byte of immediate data is logically ORed with the contents of the accumulator. The result is stored in the accumulator. The carry bit is reset to zero, and the zero, sign and parity bits are set according to the result. Condition bits affected are CARRY, ZERO, SIGN and PARITY.

FORMAT:

	CODE	OPERAND	MACHINE CODE
J. CPI - Compare Immediate with Accumulator	CPI	data	FE

The byte of immediate data is compared to the contents of the accumulator. The comparison is performed by internally subtracting the data from the accumulator using two's complement arithmetic, leaving the accumulator unchanged, but setting the condition bits by the result.

For instance, the zero bit is set if the quantities are equal, and reset if they are not equal.

In the CPI instruction a subtract operation is performed. The carry bit is set if there is no overflow from bit 7, indicating the immediate data is greater than the contents of the accumulator. The carry bit will be reset if there is overflow.

If the two quantities to be compared differ in sign, the sense of the carry bit is reversed. Condition bits affected are CARRY, ZERO, SIGN, PARITY and AUXILIARY CARRY.

XI. DIRECT ADDRESSING INSTRUCTIONS

The instructions listed below reference memory by a two-byte address which is part of the instruction. All instructions in this category occupy three bytes. The least significant byte of the address occupies the second byte of the instruction. The most significant byte occupies the third byte of the instruction. The four Direct Addressing Instructions are:

A. STA - Store Accumulator Direct
B. LDA - Load Accumulator Direct
C. SHLD - Store H and L Direct
D. LHLD - Load H and L Direct

FORMAT:

	CODE	OPERAND	MACHINE CODE
A. <u>STA</u> - Store Accumulator Direct The contents of the accumulator replace the byte at the memory address which is formed by combining OK and LOW ADD (byte two of the instruction). No condition bits are affected.	STA	adr	32
B. <u>LDA</u> - Load Accumulator Direct The byte at the memory address, which is formed by combining HI ADD and LOW ADD, replaces the contents of the accumulator. No condition bits are affected.	LDA	adr	3A
C. <u>SHLD</u> - Store H and L Direct The contents of the L register are stored at the memory address, formed by combining HI ADD and LOW ADD. Contents of the H register are stored at the next higher memory address. No condition bits are affected.	SHLD	adr	22
D. <u>LHLD</u> - Load H and L Direct The byte at the memory address formed by concatenating HI ADD with LOW ADD replaces the contents of the L register. The byte at the next higher memory address replaces the contents of the H register. No condition bits are affected.	LHLD	adr	2A

XII. JUMP INSTRUCTIONS

There are ten jump instructions, listed below. These instructions alter the normal execution sequence and each occupies either one or three bytes. The 3-byte instructions cause a transfer of program control. For instance, if the specified condition is true, program execution will continue at the memory address formed by combining the 8 bits of HI ADD (third byte) and the 8 bits of LOW ADD (second byte). If the specified condition is false, program execution will resume with the next sequential instructions.

Jump instruction addresses are stored in memory with the low-order byte first. The ten jump instructions are:

A. PCHL - Load Program Counter
B. JMP - Jump
C. JC - Jump if Carry
D. JNC - Jump if No Carry
E. JZ - Jump if Zero
F. JNZ - Jump if Not Zero
G. JM - Jump if Minus
H. JP - Jump if Positive
I. JPE - Jump if Parity Even
J. JPO - Jump if Parity Odd

FORMAT:

CODE	OPERAND	MACHINE CODE
PCHL	---	E9
JMP	adr	C3
JC	adr	DA

A. PCHL - Load Program Counter

The contents of the H register replace the most significant 8 bits of the program counter. The contents of the L register replace the least significant 8 bits of the program counter. The program then executes at the address contained in the H and L registers. No condition bits are affected.

B. JMP - Jump

Program execution continues at specified memory address. No condition bits are affected.

C. JC - Jump if Carry

Program execution continues at the specified memory address, if the carry bit is one. No condition bits are affected.

		FORMAT:		
		CODE	OPERAND	MACHINE CODE
D.	<u>JNC</u> - Jump if No Carry Program execution continues at the specified memory address, if the carry bit is zero. No conditions bits are affected.	JNC	adr	D2
E.	<u>JZ</u> - Jump if Zero Program execution continues at the specified memory address, if the zero bit is one. No condition bits are affected.	JZ	adr	CA
F.	<u>JNZ</u> - Jump is Not Zero Program execution continues at specified memory address, if the zero bit is zero. No condition bits are affected.	JNZ	adr	C2
G.	<u>JM</u> - Jump if Minus If the sign bit is one (negative result) program execution continues at the specified memory address. No condition bits are affected.	JM	adr	FA
H.	<u>JP</u> - Jump if Positive Program execution continues at the specified memory address, if the sign bit is zero. No condition bits are affected.	JP	adr	F2
I.	<u>JPE</u> - Jump if Parity Even If the parity bit is one (even parity), program execution continues at the specified memory address. No condition bits are affected.	JPE	adr	EA
J.	<u>JPO</u> - Jump if Parity Odd If the parity bit is zero (odd parity), program execution continues at the specified memory address. No condition bits are affected.	JPO	adr	E2

XIII. CALL SUBROUTINE INSTRUCTIONS

There are nine call subroutine instructions which operate much like the JMP instructions in that they cause the transfer of program control. In addition, they cause a return address to be pushed onto the stack for use by the RETURN instructions. (See Section XIV., Return from Subroutine Instructions).

Call subroutine instructions occupy three bytes of memory. Call instructions are stored in memory with the low-order byte first. Subroutines may be called under specified conditions. If the condition is true, a return address is pushed onto the stack and program execution continues at memory address formed by combining the 8 bits of HI ADD and 8 bits of LOW ADD. If the specified condition is false, program execution continues with the next sequential instruction.

The nine call subroutine instructions are:

A. CALL - Call
B. CC - Call if Carry
C. CNC - Call if No Carry
D. CZ - Call if Zero
E. CNZ - Call if Not Zero
F. CM - Call if Minus
G. CP - Call if Plus
H. CPE - Call if Parity Even
I. CPO - Call if Parity Odd

	FORMAT:		
	CODE	OPERAND	MACHINE CODE
A. <u>CALL</u> - Call A CALL operation is unconditionally performed to the specified address. No condition bits are affected.	CALL	adr	CD
B. <u>CC</u> - Call if Carry If the carry bit is one, a CALL operation is performed to the specified address. No condition bits are affected.	CC	adr	DC
C. <u>CNC</u> - Call if No Carry If the carry bit is zero, a call operation is performed to the specified address. No condition bits are affected.	CNC	adr	D4

FORMAT:

		CODE	OPERAND	MACHINE CODE
D.	CZ - Call if Zero If the zero bit is one, a call operation is performed to specified address. No condition bits are affected.	CZ	adr	CC
E.	CNZ - Call if Not Zero If the zero bit is zero, a call operation is performed to the specified address. No condition bits are affected.	CNZ	adr	C4
F.	CM - Call if Minus Call operation is performed to specified address if sign bit is one. No condition bits are affected.	CM	adr	FC
G.	CP - Call if Plus A call operation is performed to the specified address if the sign bit is zero. No condition bits are affected.	CP	adr	F4
H.	CPE - Call if Parity Even A call operation is performed to the specified address, if the parity bit is one. No condition bits are affected.	CPE	adr	EC
I.	CPO - Call if Parity Odd If the parity bit is zero, a call operation is performed to the specified address. No condition bits are affected.	CPO	adr	E4

XIV. RETURN FROM SUBROUTINE INSTRUCTIONS

The nine RETURN instructions listed below are used to return from subroutines by popping the last address saved on the stack into the program counter. This causes a transfer of program control to that address. Return instructions occupy one byte.

Return operations are performed upon specified conditions. If the specified condition is true, a return operation is performed. If it is not true, program execution continues with the next sequential instruction.

The nine return instructions are:

A. RET - Return
B. RC - Return if Carry
C. RNC - Return if No Carry
D. RZ - Return if Zero
E. RNZ - Return if Not Zero
F. FM - Return if Minus
G. RP - Return if Plus
H. RPE - Return if Parity Even
I. RPO - Return if Parity Odd

FORMAT:

	CODE	OPERAND	MACHINE CODE
A. RET - Return A return operation is unconditionally performed. Execution normally proceeds with the instruction immediately following the last call instruction. No condition bits are affected.	RET	---	C9
B. RC - Return if Carry If the carry bit is one, a return operation is performed, and no condition bits are affected.	RC	---	D8
C. RNC - Return if No Carry If the zero bit is one, a return operation is performed. No condition bits are affected.	RNC	---	D0
D. RZ - Return if Zero A return operation is performed if the zero bit is one. No condition bits are affected.	RZ	---	C8

		FORMAT:		
		CODE	OPERAND	MACHINE CODE
E.	RNZ - Return if Not Zero If the zero bit is zero, a return operation is performed. No condition bits are affected.	RNZ	---	C0
F.	RM - Return if Minus If the sign bit is one (minus result) a return operation is performed. No condition bits are affected.	RM	---	F8
G.	RP - Return if Plus If the sign bit is zero (positive result), a return operation is performed. No condition bits are affected.	RP	---	F0
H.	RPE - Return if Parity Even If the parity bit is one (even parity), a return operation is performed. No condition bits are affected.	RPE	---	E8
I.	RPO - Return if Parity Odd If the parity bit is zero (odd parity), a return operation is performed. No condition bits are affected.	RPO	---	E0

XV. RST (RESTART) INSTRUCTION

The RST instruction, a special purpose subroutine jump, occupies one byte.

FORMAT:

CODE	OPERAND	MACHINE CODE
RST	0	C7
RST	1	CF
RST	2	D7
RST	3	DF
RST	4	E7
RST	5	EF
RST	6	F7
RST	7	FF

The operand expression must evaluate to a number in the range 0 - 7. The contents of the program counter are pushed onto the stack, providing a return address for later use by a RETURN instruction. Program execution continues at memory address: OPERAND X 8

Normally, this instruction is used in conjunction with up to eight 8-byte routines in the lower 64 words of memory, to provide interrupts processing. The interrupt mechanism causes a specified RST instruction to be executed, and transfers control to a subroutine. For example, RST 1, when executed, would cause program execution to continue at memory location 8.

RETURN then causes the original program to continue execution at the location of the interrupt. No condition bits are affected.

XVI. INTERRUPT FLIP-FLOP INSTRUCTIONS

Interrupts operate directly upon the Interrupt Enable flip-flop INTE. These instructions occupy one byte. The two interrupt instructions are:

A. EI - Enable Interrupts
B. DI - Disable Interrupts

FORMAT:

	CODE	OPERAND	MACHINE CODE
A. EI - Enable Interrupts	EI	---	FB
B. DI - Disable Interrupts	DI	---	F3

A. EI - Enable Interrupts

The EI instruction sets the INTE flip-flop, enabling the CPU to recognize and respond to interrupts. No condition bits are affected.

B. DI - Disable Interrupts

The DI instruction resets the INTE flip-flop, causing the CPU to ignore all interrupts. No condition bits are affected.

XVII. INPUT/OUTPUT INSTRUCTIONS

The input and output instructions cause data to be input or output from the microprocessor. Instructions in this category occupy two bytes. They are:

A. IN - Input

B. OUT -Output

FORMAT:

	CODE	OPERAND	MACHINE CODE
A. IN - Input	IN	data	DB
B. OUT - Output	OUT	data	D3

A. IN - Input

An eight-bit data byte is read from the input port specified by the operand and replaces the contents of the accumulator. No condition bits are affected.

B. OUT - Output

The contents of the accumulator are output to the output port specified by the operand. No condition bits are affected.

XVIII. HLT - HALT INSTRUCTION

CODE	OPERAND	MACHINE CODE
HLT	---	76

The HLT instruction occupies one byte.

The program counter is incremented to the address of the next sequential instruction. The CPU then enters the STOPPED state. There is no further action until an interrupt occurs.

If the interrupt system is disabled and an HLT instruction is executed, the microprocessor must be powered down and repowered to continue operation. No condition bits are affected.

SECTION II

PSEUDO INSTRUCTIONS FOR

iCOM MACRO ASSEMBLER

Pseudo-instruction, which are recognized by the Assembler, are sritten the same way as the machine instructions, discussed in Section I, Items III through XVIII. However, the pseudo-instruction does not cause any object code to be generated. Instead it provides the assembler with data for future use while generating object code.

The six-psuedo instructions are:

A. ORG - Origin
B. EQU - Equate
C. SET - Set
D. END - End of Assembly
E. IF and ENDIF - Conditional Assembly
F. MACRO and ENDM - Macro definition

Pseudo-instruction names are not followed by a colon, as are labels. The pseudo-instructions which do require names in the label field are:

MACRO

EQU

SET

Optional labels may be used in the label fields of the remaining pseudo-instructions, as are used on machine instructions.

	FORMAT:	
	CODE	OPERAND
A. ORG - Origin		
The assembler's location counter is set to the value of a 16-bit memory address expression. The first instruction generated after an ORG statement is assembled at the expression, exp, and so forth. If no ORG appears before the first instruction in the program, assembly will begin at location 0.	ORG	exp
B. EQU - Equate	EQU	exp
The assembler assigns name the value of exp. Subsequently when the name is encountered in the assembly, this value of exp will be used. The EQU symbol may not be redefined. The name in the LABEL field may appear only once for the EQU symbol.		
C. SET - Set	SET	exp
A name in the label field is required. Identical to the EQU equation, the SET instruction differs only in that symbols may be defined more than once. The value of exp will always be used in the assembly until changed by a new SET instruction.		
D. END - End of Assembly	END	---
The end of the program is signified by use of the END statement. Only one END statement may appear in the assembly and is the last statement input. Object program and listing of the source program may now begin. END is a required statement.		

E. IF and ENDIF - Conditional Assembly	IF
	statements
	ENDIF
		exp
		
		--- 	

The assembler evaluates exp, and if evaluated to zero, the statements between IF and ENDIF are disregarded. If not zero, the statements are assembled as if the IF and ENDIF did not exist.		

FORMAT:

CODE	OPERAND
MACRO	list
Statements	
ENDM	---

F. MACRO and ENDM - Macro Definition

Name in the label field is required. For a complete explanation of programming with macros, see Section III following this section.

The assembler accepts statements between MACRO and ENDM as the definition of the macro "name". When name is encountered in the code field, the assembler substitutes the specification in the operand field for occurrences of "list" in the macro definition. The statements are then assembled.

The pseudo-instruction MACRO may not appear in the list of statements between MACRO and ENDM. Macros may not be used to define other macros.

SECTION III

MACRO PROGRAMMING FOR THE iCOM MACRO ASSEMBLER

Macros provide an important tool which can increase the efficiency and readability of the program. Its compiler capabilities make the assembly program much more powerful in that large programs may be divided into segments for separate testing. In addition, macros provide the programmer extensive analyzing capabilities in debugging. When the user becomes fully familiar with the use of macros, he will find he has a valuable means for tailoring programming to his particular needs.

The user will therefore utilize macro programming to decrease debugging time, reduce the drudgery of often-repeated groups of instructions, and reduce duplication of efforts between programmers.

A. OPERATION

The macro name and its representative instructions are selected by the programmer. The macro name, or symbol specified to the assembler, appears in the code field and represents a group of instructions.

EXAMPLE: This macro will print the contents of the accumulator, after shifting it, to the right one bit, and a zero will shift to the high order bit position. This macro will be called SHPRT, and is defined by writing the following instructions:

```
LABEL   CODE     OPERAND    COMMENT

SHPRT:  MACRO
        RRC                 ; Rotate accumulator right
        ANI      7FH        ; Clear high order bit
        MOV C,A             ;
        CALL     CO         ; Output to console
        ENDM
```

The macro may be referenced later in the program by using this instruction:

```
LABEL CODE   OPERAND   COMMENT

      LDA    TEMP      ; Load Accumulator
      SHPRT            ;
```

This would be the same as writing:

```
      LDA    TEMP      ; Load Accumulator
      RRC
      ANI    7FH
      MOV    C,A
      CALL   CO
```

As demonstrated above, three aspects of macros are immediately available to the programmer:

1. DEFINITION
2. REFERENCE
3. EXPANSION

1. Definition specifies the sequence the instructions will take. SHPRT is used to specify the four instructions in the code field. Each macro need be specified only once in the program.

```
EXAMPLE: LABEL CODE   OPERAND

         SHPRT: MACRO
                RRC
                ANI   7FH
                MOV   C,A
                CALL  CO
                ENDM
```

2. Reference specifies the macro at a point in the program, and the macro may be referenced in any number of statements by inserting the macro name in the code field:

```
LDA   TEMP
SHPRT           ; Macro referenced
STA   TEMP
```

3. Expansion is the complete instruction sequence represented by the macro reference. The macro expansion will be present in its machine language equivalent and will be generated by the assembler in the object program:

```
LDA   TEMP
RRC             ; Macro referenced
ANI   7FH
MOV   C,A
CALL  CO
STA   TEMP
```

B. MACRO TERMS AND IMPLEMENTATION

A macro must first be defined, then referenced, and each reference must have an equivalent expansion. Each of the three aspects of a macro is discussed below.

```
FORMAT:

LABEL  CODE   OPERAND
```

1. MACRO DEFINITION

The macro definition indicates to the assembler that the symbol "name" is the equivalent to the group of statements residing between the pseudo instructions MACRO and ENDM. The macro definition does not produce assembled data in the object program. The macro body, or group of statements, may be assembly language instructions, pseudo-instructions except MACRO or ENDM, comments or references to other macros.

```
name:  MACRO  list
(.....macro body.....)
       ENDM
```

Expressions indicating parameters specified by a macro reference is called "list". These expressions are replaced in the macro body and serve to designate the location of macro parameters. "list" expressions are called "dummy parameters".

This macro takes the memory address of the label specified by the macro reference, loads the address into the H register and loads the least significant 8 bits into the B register.

```
LOAD:  MACRO  ADDR
       LXI    H,ADDR
       MVI    B,ADDR AND OFFH
       ENDM
```

The reference:

```
       LOAD   LABEL
```

Equivalent to the expansion:

```
       LXI    H,LABEL
       MVI    B,LABEL AND OFFH
```

The reference:

```
       LOAD   INST
```

Equivalent to the expansion:

```
       MVI    H,INST
              B,INST AND OFFH
```

MACRO and END statements tell the assembler than when LOAD appears in the code field the characters in the operand field are to be substituted wherever the symbol ADDR appears in the macro body, and the LXI and MVI instructions are inserted into the statements and assembled.

2. MACRO REFERENCE

The name of a macro appears in the label field of the MACRO pseudo-instruction. A list of expressions is substituted in the operand field, using the first string of "list" to replace every occurrence of the first dummy parameter in the macro body, the second to replace every second occurrence, etc.

If the parameters appearing in the macro reference are fewer than in the definition, a null string is substituted for the remaining expressions. If more parameters appear in the reference than the definition, the extra parameters are ignored.

```
EXAMPLE:            FORMAT:

                    LABEL CODE   OPERAND         COMMENT

Using the macro     PMSG: MACRO  P1,P2,P3        ; Comment
definition:               LXI    H,P2
                          MVI    B,P1            ; Comment
                          CALL
                          ENDM

Reference:                PMSG   MSG1,CNT,ADDR   ; Print message
                                                 ; on device X


Equivalent to
Expansion:                LXI    H,MSG1
                          MVI    B,CNT
                          CALL   ADDR            ; Print message
                                                 ; on device X

Reference:                PMSG   MSG2,NUMB,ADDR2

Equivalent to
Expansion:                LXI    H,MSG2
                          MVI    B,NUMB
                          CALL   ADDR2
```

3. MACRO EXPANSION

Macro expansion is the result of substituting the macro parameters into the macro body. The expansion statements are assembled into the assembler just as it assembles other statements. For instance, each statement derived from expansion of the macro must be a legal assembler statement.

EXAMPLE:	FORMAT:		
	LABEL	CODE	OPERAND
Using the macro definition:	MDEC:	MACRO	P1
		DCX	P1
		ENDM	
Reference:		MDEC	H
Result is legal expansion:		DCX	H
However, using reference:		MDEC	L
Results in illegal expansion:		DCX	L

This will be flagged as an error.

C. LABELS AND NAMES

Two terms are used to determine how references, definitions and expansions of macros are used.

GLOBAL:	A symbol is globally defined if its value is known and can be referenced by any statement in the program, regardless of whether the statement is the result of expansion of a macro.
LOCAL:	A symbol is locally defined if its value is known and can be referenced only within a specific macro expansion.

1. INSTRUCTION LABELS

A symbol may normally appear in the label field of only one instruction. However, if a label appears in the macro body it will be generated any time the macro is referenced. Macros are treated as local labels to avoid multiple-label conflicts.

To generate a global label, the programmer must type two colons following the label in the macro definition. This global label may be generated just once since it is unique in the program.

EXAMPLE:

FORMAT:

Definition:

If two references to MAC1 appear in a program, CONTU will be a local label and each JMP CONTU instruction refers to the label generated within its own expansion:

If CONTU had been followed, in the macro definition, by two colons (::) CONTU would be generated as a global label by the first reference to MAC1, and the second reference would be flagged as an error.

```
LABEL   CODE   OPERAND

MAC1    MACRO
CONTU:  macro  body
        JMP    CONTU
        ENDM

CONTU:  MAC1
        CONTU
        macro  body
        JMP    CONTU

          .
          .
          .
CONTU:  MAC1
        CONTU
        macro  body
        JMP    CONTU
```

2. EQUATE Names

Equate names on statements within a macro are always local, and are always defined within the expansion in which they are generated.

```
EXAMPLE:                            FORMAT:
                                                         ASSM.
                                    LABEL  CODE   OPERAND DATA

Macro definition:                   MAC2   MACRO
                                    VALU   EQU    40H
                                           DB     VALU
                                           ENDM

Valid program:                      VALU   EQU    0FFH    FF
                                    DB1:   DB     VALU
                                           MAC2
                                    VALU   EQU    40H
                                           DB     VALU    40
                                    DB2:   DB     VALU    FF
```

VALU is defined first globally with the value FF. The reference to VALU at DB1 therefore produces a byte equal to FF.

MAC2 is the macro reference which generates the symbol VALU and is defined only within the macro expansion with the value 40. The reference to VALU by the second statement produces a byte equal to 40.

The reference to VALU at DB2 refers to the global definition of VALU, because the VALU statement ends the macro expansion. The statement at DB2 therefore produces a byte qual to the value FF.

3. SET Names

If a SET statement is generated by a macro, and the name has previously been defined globally by another SET statement, the generated statement changes the global value of the name thereafter. If the SET statement's name had not previously been defined, the name is defined locally and applies only in the current macro expansion.

```
    EXAMPLE:                    FORMAT:
                                                          ASSEM.
                                LABEL CODE   OPERAND    DATA

    Macro Definition:           MAC3  MACRO
                                SYMBL SET    16
                                      DB     SYMBL
                                      ENDM


    Valid Program Section:      SYMBL SET    32D
                                DB1:  DB     SYMBL      20
                                      MAC2
                                SYMBL SET    16D
                                      DB     SYMBL      10
                                DB2:  DB     SYMBL      10
```

SYMBL is first defined globally with the value 32. This causes the reference at DB1 to produce a byte of 20H. The macro reference MAC2 resets the global value to 10H, causing the second statement to produce a value of 10H. The value of SYMBL remains equal to 10H, as indicated by the reference at DB2.

```
    EXAMPLE:                          MAC2
                                SYMBL SET    16
                                      DB     SYMBL      10
                                DB3:  DB     SYMBL      **ERROR**
```

The statement at DB3 is invalid because SYMBL is unknown globally.

D. MACRO PARAMETER SUBSTITUTION

Macro parameters value is assigned prior to expansion, when the macro is referenced. Evaluation can be delayed by enclosing a parameter in quotation marks so that the character string will appear in the macro body. The string will be evaluated at the occurrence of macro expansion.

EXAMPLE: FORMAT:

```
LABEL CODE  OPERAND
```

Macro MAC 3 is defined at beginnign of program:

```
MAC3  MACRO P1
LABL  SET   0
DB    P1
ENDM
```

The value of LABL is set to 5 by writing SET prior to the first reference to MAC3.

```
LABL  SET   5
```

Macro Reference:

```
      MAC3  LABL
```

This causes assembler to evaluate LABL and to substitute the value 5 for parameter P1.

or: Macro Reference:

```
      MAC3  "LABL"
```

Assembler evaluates expression "LABL", producing the characters LABL as the value of parameter P1. Expansion is now produced. P1 now produces the value 0 because LABL is altered by the first statement of the expansion.

Expansion produced:

```
LABL  SET   0        ;
      DB    LABL     ; Assembles
                     ; as 0
```

APPENDIX

iCOM MACRO ASSEMBLER

MNEMONIC INDEX

STATEMENT	OPERATION	TYPE INSTRUCTION	PAGE NO.
ACI	Add Immediate to Accumulator with Carry	Immediate Instruction	28
ADC	Add Register/Memory to Accumulator with Carry	Register/Memory to Accumulator	19
ADD	Add Register or Memory to Accumulator	Register/Memory to Accumulator	19
ADI	Add Immediate to Accumulator	Immediate Instruction	28
ANA	Logical AND Register or Memory with Accumulator	Register/Memory to Accumulator Instruc.	20
ANI	AND Immediate with Accumulator	Immediate Instruction	29
CALL	Call	Call Subroutine Instruc.	34
CNZ	Call if Not Zero	Call Subroutine Instruc.	35
CZ	Call if Zero	Call Subroutine Instruc.	35
CC	Call if Carry	Call Subroutine Instruc.	34
CM	Call if Minus	Call Subroutine Instruc.	35
CMA	Complement Accumulator	Single Register Instruc.	14
CMC	Complement Carry	Carry Bit Instruction	13
CMP	Compare Register or Memory with Accumulator	Register/Memory to Accumulator Instruction	21
CNC	Call if No Carry	Call Subroutine Instruc.	34
CP	Call if Plus	Call Subroutine Instruc.	35
CPE	Call if Parity Even	Call Subroutine Instruc.	35
CPO	Call if Parity Odd	Call Subroutine Instruc.	35
CPI	Compare Immediate with Accumulator	Immediate Instruction	30

STATEMENT	OPERATION	TYPE INSTRUCTION	PAGE NO.
DB	Define Byte	Data Statement	11
DAA	Decimal Adjust Accumulator	Single Register Instruc.	15
DW	Define Word	Data Statement	12
DAD	Double Add	Register Pair Instruc.	25
DS	Define Storage	Data Statement	12
DCR	Decrement Register or Memory	Single Register Instruc.	14
DCX	Decrement Register Pair	Register Pair Instruction	26
DI	Disable Interrupts	Interrupt Flip-Flop Inst.	39
EI	Enable Interrupts	Interrupt Flip-Flop Inst.	39
END	End of Assembly	Pseudo-Instruction	42
ENDM	End Macro Statement	Pseudo-Instruction	43
EQU	Equate	Pseudo-Instruction	42
HLT	Halt	Halt Instruction	40
IF and ENDIF	Conditional Assembly	Pseudo-Instruction	42
IN	Input	Input/Output Instruction	40
INR	Increment Register or Memory	Single Register Instruc.	14
INX	Increment Register Pair	Register Pair Instruction	26
JMP	Jump	Jump Instruction	32
JZ	Jump if Zero	Jump Instruction	33
JNZ	Jump if Not Zero	Jump Instruction	33
JP	Jump if Positive	Jump Instruction	33
JM	Jump if Minus	Jump Instruction	33
JC	Jump if Carry	Jump Instruction	33

STATEMENT	OPERATION	TYPE INSTRUCTION	PAGE NO.
JNC	Jump if No Carry	Jump Instruction	33
JPE	Jump if Parity Even	Jump Instruction	33
JPO	Jump if Parity Odd	Jump Instruction	33
LDA	Load Accumulator Direct	Direct Addressing Instruc.	31
LDAX	Load Accumulator	Data Transfer Instruction	18
LHLD	Load H and L Direct	Direct Addressing Instruc.	31
LXI	Load Register Pair Immediate	Immediate Instruction	27
MACRO and ENDM	Macro Definition	Pseudo-Instructions	43
MOV	Move	Data Transfer Instruction	17
MVI	Move Immediate Data	Immediate Instruction	28
NOP	No Operation	NOP Instruction	16
ORA	Logical OR Register or Memory with Accumulator	Register/Memory to Accumulator Instruction	21
ORI	OR Immediate with Accumulator	Immediate Instruction	29
ORG	Origin	Pseudo-Instruction	42
OUT	Output	Input/Output Instructions	40
PCHL	Load Program Counter	Jump Instruction	36
POP	Pop Data Off Stack	Register Pair Instruction	25
PUSH	Push Data Onto Stack	Register Pair Instruction	24
RAL	Rotate Accumulator Left Through Carry	Rotate Accumulator Instruc.	22
RAR	Rotate Accumulator Right through Carry	Rotate Accumulator Instruc.	23
RLC	Rotate Accumulator Left	Rotate Accumulator Instruc.	22

STATEMENT	OPERATION	TYPE INSTRUCTION	PAGE NO.
RRC	Rotate Accumulator Right	Rotate Accumulator Instruc.	22
RET	Return	Return from Subroutine	36
RZ	Return if Zero	Return from Subroutine	36
RNZ	Return if Not Zero	Return from Subroutine	37
RP	Return if Plus	Return from Subroutine	37
RM	Return if Minus	Return from Subroutine	37
RC	Return if Carry	Return from Subroutine	36
RNC	Return if No Carry	Return from Subroutine	36
RPE	Return if Parity Even	Return from Subroutine	37
RPO	Return if Parity Odd	Return from Subroutine	37
RST	Restart	Restart Instruction	38
SET	Set	Pseudo-Instruction	42
SPHL	Load SP from H and L	Register Pair Instruction	26
SHLD	Store H and L Direct	Direct Addressing Instruc.	31
STA	Store Accumulator Direct	Direct Addressing Instruc.	31
STAX	Store Accumulator	Data Transfer Instruction	18
STC	Set Carry	Carry Bit Instruction	13
SUB	Subtract Register or Memory from Accumulator	Register/Memory to Accumulator Instruction	20
SUI	Subtract Immediate from Accumulator	Immediate Instruction	28
SBB	Subtract Register or Memory from Accumulator With Borrow	Register/Memory to Accumulator Instruction	20
SBI	Subtract Immediate from Accumulator with Borrow	Immediate Instruction	29

ADDENDUM APPENDIX

iCOM MACRO ASSEMBLER

ERROR MESSAGES

The iCOM Macro Assembler detects errors by indicating a single-letter code on the output listing. If multiple errors occur in a single line of code, only the first error is indicated.

CODE	DEFINITION	EXAMPLE

B	Balance Error--Parentheses in an expression or quotes in a string are unbalanced.	ERROR: `ORG $/256+1)*256-$`
		`DB 'A`
		CORRECTION:
		`ORG (256+1)*256-$`
		`DB 'A'`

E	Expression Error--Poorly constructed expression due to missing operator, omitted comma or misspelled opcode.	ERROR: `ORG ($/256+1)256-$` CORRECTION: `ORG ($/256+1)*256-$`

F	Format Error--Error in format of a statement, usually caused by a missing or extraneous operand.	ERROR: `MOV A`
		`MOV A,B,C`
		CORRECTION:
		`MOV A,B`

I	Illegal Character--Illegal ASCII character is present in the statement or a numeric character is too large for the base of the number in which it occurs.	ERROR: `MVI A,02B`
		`ADI A,79Q`
		CORRECTION:
		`MVI A,00000010B`
		`ADI A,77Q`

M	Multiple Definition--Symbol or macro is defined more than once. M occurs on all definitions of and references to the multiply-defined symbol. Symbols must be unique in the first five characters.	ERROR: `LOCATION1: NOP`
		`LOCATION2: NOP`
		CORRECTION:
		`LOC1: NOP`
		`LOC2: NOP`

N	Nesting Error--ENDIF, ENDM, or END statements improperly nested. IF statement must precede statements which appear in the program, followed by ENDIF.	ERROR: `ENDIF`
		CORRECTION:
		`IF (expression)`
		`statements`
		`ENDIF ---`

P	Phase Error--Value of an element being defined has changed between pass one and pass two of the assembly.	

CODE	DEFINITION	EXAMPLE
P	(Continued) During pass one, BEGIN is undefined when ORG is encountered. Assembler assumes it to be at location zero and begins assembling the program at zero. During pass two BEGIN is equal to 5. The location assigned to every label in the program will then be increased by 5, producing the P error.	ERROR: ORG BEGIN statements BEGIN EQU 5 statements CORRECTION: BEGIN EQU 5 statements ORG BEGIN
Q	Questionable Syntax--Omission or misspelled opcode.	ERROR: MVO A,B CORRECTION: MOV A,B
R	Register Error--Register specified for an operation is invalid for the operation.	ERROR: INR 9 CORRECTION: INR 7

S	Stack Overflow--Assembler's internal expression evaluation stack is too large for available memory. Causes include using excessively long character strings, excessive nested macros, excessive nested IF statements, or too complex expresisons.
	
	Nested IF statement occurs between another IF/ENDIF pair.
		ERROR: IF expression
		IF expression
		IF expression
		statements
		ENDIF
		ENDIF
		ENDIF
		
		CORRECTION:
		IF expression
		IF expression
		statements
		ENDIF
		ENDIF

T	Table Overflow--Assembler's symbol table space is exhausted, caused by using excessive symbols in one assembly, or by accumulating more macro text than the assembler can store in available memory. To correct, add memory or reduce the number of labels.	
U	Undefined Identifier--Symbol used in an operand field has never been defined by appearing in the label field of another instruction	ERROR: JMP LAB1 CORRECTION: JMP LAB1 statements LAB1 instruction

CODE	DEFINITION	EXAMPLE
V	Illegal Value--Value of an operand or expression exceeds range required for a specific expression. The MVI instruction, for example, must be in the number range 0 to 255.	ERROR: MVI A,257 CORRECTION: MVI A,255