DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

CP/M ASSEMBLER (ASM)

USER'S GUIDE

Disclaimer

DIGITAL RESEARCH

Post Office Box 579, Pacific Grove, California 93950, (408) 649-3896

CP/M ASSEMBLER (ASM)
USER'S GUIDE

Disclaimer

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

Table of Contents

CP/M Assembler User's Guide

1. INTRODUCTION.

The CP/M assembler reads assembly language source files from the diskette, and produces 8080 machine language in Intel hex format. The CP/M assembler is initiated by typing

```
ASM filename
```

or

```
ASM filename.parms
```

In both cases, the assembler assumes there is a file on the diskette with the name

```
filename.ASM
```

which contains an 8080 assembly language source file. The first and second forms shown above differ only in that the second form allows parameters to be passed to the assembler to control source file access and hex and print file destinations.

In either case, the CP/M assembler loads, and prints the message

```
CP/M ASSEMBLER VER n.n
```

where n.n is the current version number. In the case of the first command, the assembler reads the source file with assumed file type "ASM" and creates two output files

```
filename.HEX
```

and

```
filename.PRN
```

the "HEX" file contains the machine code corresponding to the original program in Intel hex format, and the "PRN" file contains an annotated listing showing generated machine code, error flags, and source lines. If errors occur during translation, they will be listed in the PRN file as well as at the console

The second command form can be used to redirect input and output files from their defaults. In this case, the "parms" portion of the command is a three letter group which specifies the origin of the source file, the destination of the hex file, and the destination of the print file. The form is

```
filename.p1p2p3
```

where p1, p2, and p3 are single letters

p1: A,B, ..., Y designates the disk name which contains

```
                    the source file
p2:  A,B, ..., Y    designates the disk name which will re-
                    ceive the hex file
     Z              skips the generation of the hex file
p3:  A,B, ..., Y    designates the disk name which will re-
                    ceive the print file
     X              places the listing at the console
     Z              skips generation of the print file
```

Thus, the command

ASM X.AAA

indicates that the source file (X.ASM) is to be taken from disk A, and that the hex (X.HEX) and print (X.PRN) files are to be created also on disk A. This form of the command is implied if the assembler is run from disk A. That is, given that the operator is currently addressing disk A, the above command is equivalent to

ASM X

The command

ASM X.ABX

indicates that the source file is to be taken from disk A, the hex file is placed on disk B, and the listing file is to be sent to the console. The command

ASM X.BZZ

takes the source file from disk B, and skips the generation of the hex and print files (this command is useful for fast execution of the assembler to check program syntax).

The source program format is compatible with both the Intel 8080 assembler (macros are not currently implemented in the CP/M assembler, however), as well as the Processor Technology Software Package #1 assembler. That is, the CP/M assembler accepts source programs written in either format. There are certain extensions in the CP/M assembler which make it somewhat easier to use. These extensions are described below.

2. PROGRAM FORMAT.

An assembly language program acceptable as input to the assembler consists of a sequence of statements of the form

```
line#    label    operation    operand    ;comment
```

where any or all of the fields may be present in a particular instance. Each

~embly language statement is terminated with a carriage return and line feed (the line feed is inserted automatically by the ED program), or with the character "!" which is a treated as an end-of-line by the assembler (thus, multiple assembly language statements can be written on the same physical line if separated by exclaim symbols).

The line# is an optional decimal integer value representing the source program line number, which is allowed on any source line to maintain compatibility with the Processor Technology format. In general, these line numbers will be inserted if a line-oriented editor is used to construct the original program, and thus ASM ignores this field if present.

The label field takes the form

identifier

or

identifier:

and is optional, except where noted in particular statement types. The identifier is a sequence of alphanumeric characters (alphabetics and numbers), where the first character is alphabetic. Identifiers can be freely used by the programmer to label elements such as program steps and assembler directives, but cannot exceed 16 characters in length. All characters are significant in an identifier, except for the embedded dollar symbol ($) which can be used to improve readability of the name. Further, all lower case alphabetics become are treated as if they were upper case. Note that the ":" following the identifier in a label is optional (to maintain compatibility between Intel and Processor Technology). Thus, the following are all valid instances of labels

```
x       xy      long$name
x:      yxl:    longer$named$data:
XlY2    Xlx2    x234$5678$9012$3456:
```

The operation field contains either an assembler directive, or pseudo operation, or an 8080 machine operation code. The pseudo operations and machine operation codes are described below.

The operand field of the statement, in general, contains an expression formed out of constants and labels, along with arithmetic and logical operations on these elements. Again, the complete details of properly formed expressions are given below.

The comment field contains arbitrary characters following the ";" symbol until the next real or logical end-of-line. These characters are read, listed, and otherwise ignored by the assembler. In order to maintain compatability with the Processor Technology assembler, the CP/M assembler also treat statements which begin with a "*" in column one as comment statements, which are listed and ignored in the assembly process. Note that the Processor

Technology assembler has the side effect in its operation of ignoring the characters after the operand field has been scanned. This causes an ambiguous situation when attempting to be compatible with Intel´s language, since arbitrary expressions are allowed in this case. Hence, programs which use this side effect to introduce comments, must be edited to place a ";" before these fields in order to assemble correctly.

The assembly language program is formulated as a sequence of statements of the above form, terminated optionally by an END statement. All statements following the END are ignored by the assembler.

3. FORMING THE OPERAND.

In order to completely describe the operation codes and pseudo operations, it is necessary to first present the form of the operand field, since it is used in nearly all statements. Expressions in the operand field consist of simple operands (labels, constants, and reserved words), combined in properly formed subexpressions by arithmetic and logical operators. The expression computation is carried out by the assembler as the assembly proceeds. Each expression must produce a 16-bit value during the assembly. Further, the number of significant digits in the result must not exceed the intended use. That is, if an expression is to be used in a byte move immediate instruction, then the most significant 8 bits of the expression must be zero. The restrictions on the expression significance is given with the individual instructions.

3.1. Labels.

As discussed above, a label is an identifier which occurs on a particular statement. In general, the label is given a value determined by the type of statement which it precedes. If the label occurs on a statement which generates machine code or reserves memory space (e.g, a MOV instruction, or a DS pseudo operation), then the label is given the value of the program address which it labels. If the label precedes an EQU or SET, then the label is given the value which results from evaluating the operand field. Except for the SET statement, an identifier can label only one statement.

When a label appears in the operand field, its value is substituted by the assembler. This value can then be combined with other operands and operators to form the operand field for a particular instruction.

3.2. Numeric Constants.

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. The radix indicators are

B	binary constant (base 2)
O	octal constant (base 8)

Q	octal constant (base 8)
D	decimal constant (base 10)
H	hexadecimal constant (base 16)

Q is an alternate radix indicator for octal numbers since the letter O is easily confused with the digit 0. Any numeric constant which does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is thus composed as a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. That is binary constants must be composed of 0 and 1 digits, octal constants can contain digits in the range 0 - 7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A (10D), B (11D), C (12D), D (13D), E (14D), and F (15D). Note that the leading digit of a hexadecimal constant must be a decimal digit in order to avoid confusing a hexadecimal constant with an identifier (a leading 0 will always suffice). A constant composed in this manner must evaluate to a binary number which can be contained within a 16-bit counter, otherwise it is truncated on the right by the assembler. Similar to identifiers, imbedded "$" are allowed within constants to improve their readability. Finally, the radix indicator is translated to upper case if a lower case letter is encountered. The following are all valid instances of numeric constants

1234	1234D	1100B	1111$0000$1111$0000B
1234H	0FFEH	3377O	33$77$22Q
3377o	0fe3h	1234d	0ffffh

3.3. Reserved Words.

There are several reserved character sequences which have predefined meanings in the operand field of a statement. The names of 8080 registers are given below, which, when encountered, produce the value shown to the right

A	7
B	0
C	1
D	2
E	3
H	4
L	5
M	6
SP	6
PSW	6

(again, lower case names have the same values as their upper case equivalents). Machine instructions can also be used in the operand field, and evaluate to their internal codes. In the case of instructions which require operands, where the specific operand becomes a part of the binary bit pattern of the instruction (e.g, MOV A,B), the value of the instruction (in this case MOV) is the bit pattern of the instruction with zeroes in the optional fields (e.g, MOV produces 40H).

When the symbol "$" occurs in the operand field (not imbedded within identifiers and numeric constants) its value becomes the address of the next instruction to generate, not including the instruction contained withing the current logical line.

3.4. String Constants.

String constants represent sequences of ASCII characters, and are represented by enclosing the characters within apostrophe symbols (´). All strings must be fully contained within the current physical line (thus allowing "!" symbols within strings), and must not exceed 64 characters in length. The apostrophe character itself can be included within a string by representing it as a double apostrophe (the two keystrokes ´´), which becomes a single apostrophe when read by the assembler. In most cases, the string length is restricted to either one or two characters (the DB pseudo operation is an exception), in which case the string becomes an 8 or 16 bit value, respectively. Two character strings become a 16-bit constant, with the second character as the low order byte, and the first character as the high order byte.

The value of a character is its corresponding ASCII code. There is no case translation within strings, and thus both upper and lower case characters can be represented. Note however, that only graphic (printing) ASCII characters are allowed within strings. Valid strings are

```
´A´     ´AB´     ´ab´   ´c´
´´´´    ´a´´´    ´´´´´´ ´´´"´
´Walla Walla Wash.´
´She said ´´Hello´´ to me.´
´I said "Hello" to her.´
```

3.5. Arithmetic and Logical Operators.

The operands described above can be combined in normal algebraic notation using any combination of properly formed operands, operators, and parenthesized expressions. The operators recognized in the operand field are

a + b	unsigned arithmetic sum of a and b
a - b	unsigned arithmetic difference between a and b
+ b	unary plus (produces b)
- b	unary minus (identical to 0 - b)
a * b	unsigned magnitude multiplication of a and b
a / b	unsigned magnitude division of a by b
a MOD b	remainder after a / b
NOT b	logical inverse of b (all 0´s become 1´s, 1´s become 0´s), where b is considered a 16-bit value

a AND b	bit-by-bit logical and of a and b
a OR b	bit-by-bit logical or of a and b
a XOR b	bit-by-bit logicl exclusive or of a and b
a SHL b	the value which results from shifting a to the left by an amount b, with zero fill
a SHR b	the value which results from shifting a to the right by an amount b, with zero fill

In each case, a and b represent simple operands (labels, numeric constants, reserved words, and one or two character strings), or fully enclosed parenthesized subexpressions such as

```
10+20        10h+37Q      L1 /3     (L2+4) SHR 3
('a' and 5fh) + '0'       ('B'+B) OR (PSW+M)
(1+(2+c)) shr (A-(B+1))
```

Note that all computations are performed at assembly time as 16-bit unsigned operations. Thus, -1 is computed as 0-1 which results in the value 0ffffh (i.e., all 1's). The resulting expression must fit the operation code in which it is used. If, for example, the expression is used in a ADI (add immediate) instruction, then the high order eight bits of the expression must be zero. As a result, the operation "ADI -1" produces an error message (-1 becomes 0ffffh which cannot be represented as an 8 bit value), while "ADI (-1) AND 0FFH" is accepted by the assembler since the "AND" operation zeroes the high order bits of the expression.

3.6. Precedence of Operators.

As a convenience to the programmer, ASM assumes that operators have a relative precedence of application which allows the programmer to write expressions without nested levels of parentheses. The resulting expression has assumed parentheses which are defined by the relative precedence. The order of application of operators in unparenthesize expressions is listed below. Operators listed first have highest precedence (they are applied first in an unparenthesized expression), while operators listed last have lowest precedence. Operators listed on the same line have equal precedence, and are applied from left to right as they are encountered in an expression

```
* / MOD SHL SHR
     - +
     NOT
     AND
    OR XOR
```

Thus, the expressions shown to the left below are interpreted by the assembler as the fully parenthesize expressions shown to the right below

```
a * b + c                   (a * b) + c
a + b * c                   a + (b * c)
a MOD b * c SHL d           ((a MOD b) * c) SHL d
```

```
a OR b AND NOT c + d SHL e        a OR (b AND (NOT (c + (d SHL e))))
```

Balanced parenthesized subexpressions can always be used to override the assumed parentheses, and thus the last expression above could be rewritten to force application of operators in a different order as

```
(a OR b) AND (NOT c) + d SHL e
```

resulting in the assumed parentheses

```
(a OR b) AND ((NOT c) + (d SHL e))
```

Note that an unparenthesized expression is well-formed only if the expression which results from inserting the assumed parentheses is well-formed.

4. ASSEMBLER DIRECTIVES.

Assembler directives are used to set labels to specific values during the assmbly, perform conditional assembly, define storage areas, and specify starting addresses in the program. Each assembler directive is denoted by a "pseudo operation" which appears in the operation field of the line. The acceptable pseudo operations are

ORG	set the program or data origin
END	end program, optional start address
EQU	numeric "equate"
SET	numeric "set"
IF	begin conditional assembly
ENDIF	end of conditional assembly
DB	define data bytes
DW	define data words
DS	define data storage area

The individual pseudo operations are detailed below

4.1. The ORG directive.

The ORG statement takes the form

```
label   ORG    expression
```

where "label" is an optional program label, and expression is a 16-bit expression, consisting of operands which are defined previous to the ORG statement. The assembler begins machine code generation at the location specified in the expression. There can be any number of ORG statements within a particular program, and there are no checks to ensure that the programmer is not defining overlapping memory areas. Note that most programs written for the CP/M system begin with an ORG statement of the form

```
ORG 100H
```

which causes machine code generation to begin at the base of the CP/M transient program area. If a label is specified in the ORG statement, then the label is given the value of the expression (this label can then be used in the operand field of other statements to represent this expression).

4.2. The END directive.

The END statement is optional in an assembly language program, but if it is present it must be the last statement (all subsequent statements are ignored in the assembly). The two forms of the END directive are

```
label   END
label   END   expression
```

where the label is again optional. If the first form is used, the assembly process stops, and the default starting address of the program is taken as 0000. Otherwise, the expression is evaluated, and becomes the program starting address (this starting address is included in the last record of the Intel formatted machine code "hex" file which results from the assembly). Thus, most CP/M assembly language programs end with the statement

```
END 100H
```

resulting in the default starting address of 100H (beginning of the transient program area).

4.3. The EQU directive.

The EQU (equate) statement is used to set up synonyms for particular numeric values. the form is

```
label   EQU   expression
```

where the label must be present, and must not label any other statement. The assembler evaluates the expression, and assigns this value to the identifier given in the label field. The identifier is usually a name which describes the value in a more human-oriented manner. Further, this name is used throughout the program to "parameterize" certain functions. Suppose for example, that data received from a Teletype appears on a particular input port, and data is sent to the Teletype through the next output port in sequence. The series of equate statements could be used to define these ports for a particular hardware environment

```
TTYBASE   EQU   10H       ;BASE PORT NUMBER FOR TTY
TTYIN     EQU   TTYBASE   ;TTY DATA IN
TTYOUT    EQU   TTYBASE+1;TTY DATA OUT
```

At a later point in the program, the statements which access the Teletype could appear as

```
        IN      TTYIN       ;READ TTY DATA TO REG-A
        ...
        OUT     TTYOUT      ;WRITE DATA TO TTY FROM REG-A
```

making the program more readable than if the absolute i/o ports had been used. Further, if the hardware environment is redefined to start the Teletype communications ports at 7FH instead of 1ØH, the first statement need only be changed to

```
TTYBASE   EQU   7FH     ;BASE PORT NUMBER FOR TTY
```

and the program can be reassembled without changing any other statements.

4.4. The SET Directive.

The SET statement is similar to the EQU, taking the form

```
label   SET   expression
```

except that the label can occur on other SET statements within the program. The expression is evaluated and becomes the current value associated with the label. Thus, the EQU statement defines a label with a single value, while the SET statement defines a value which is valid from the current SET statement to the point where the label occurs on the next SET statement. The use of the SET is similar to the EQU statement, but is used most often in controlling conditional assembly.

4.5. The IF and ENDIF directives.

The IF and ENDIF statements define a range of assembly language statements which are to be included or excluded during the assembly process. The form is

```
IF   expression
statement#1
statement#2
   ...
statement#n
ENDIF
```

Upon encountering the IF statement, the assembler evaluates the expression following the IF (all operands in the expression must be defined ahead of the IF statement). If the expression evaluates to a non-zero value, then statement#1 through statement#n are assembled; if the expression evaluates to zero, then the statements are listed but not assembled. Conditional assembly is often used to write a single "generic" program which includes a number of possible run-time environments, with only a few specific portions of the program selected for any particular assembly. The following program segments for example, might be part of a program which communicates with either a Teletype or a CRT console (but not both) by selecting a particular value for TTY before the assembly begins

```
TRUE     EQU   0FFFFH      ;DEFINE VALUE OF TRUE
FALSE    EQU   NOT TRUE    ;DEFINE VALUE OF FALSE
;
TTY      EQU   TRUE        ;TRUE IF TTY, FALSE IF CRT
;
TTYBASE  EQU   10H         ;BASE OF TTY I/O PORTS
CRTBASE  EQU   20H         ;BASE OF CRT I/O PORTS
         IF    TTY         ;ASSEMBLE RELATIVE TO TTYBASE
CONIN    EQU   TTYBASE     ;CONSOLE INPUT
CONOUT   EQU   TTYBASE+1   ;CONSOLE OUTPUT
         ENDIF
;
         IF    NOT TTY     ;ASSEMBLE RELATIVE TO CRTBASE
CONIN    EQU   CRTBASE     ;CONSOLE INPUT
CONOUT   EQU   CRTBASE+1   ;CONSOLE OUTPUT
         ENDIF
         ...
         IN     CONIN      ;READ CONSOLE DATA
         ...
         OUT    CONOUT     ;WRITE CONSOLE DATA
```

In this case, the program would assemble for an environment where a Teletype is connected, based at port 10H. The statement defining TTY could be changed to

```
TTY      EQU   FALSE
```

and, in this case, the program would assemble for a CRT based at port 20H.

4.6. The DB Directive.

The DB directive allows the programmer to define initialize storage areas in single precision (byte) format. The statement form is

```
label    DB    e#1, e#2, ..., e#n
```

where e#1 through e#n are either expressions which evaluate to 8-bit values (the high order eight bits must be zero), or are ASCII strings of length no greater than 64 characters. There is no practical restriction on the number of expressions included on a single source line. The expressions are evaluated and placed sequentially into the machine code file following the last program address generated by the assembler. String characters are similarly placed into memory starting with the first character and ending with the last character. Strings of length greater than two characters cannot be used as operands in more complicated expressions (i.e., they must stand alone between the commas). Note that ASCII characters are always placed in memory with the parity bit reset (0). Further, recall that there is no translation from lower to upper case within strings. The optional label can be used to reference the data area throughout the remainder of the program. Examples of valid DB statements are

```
data:   DB   0,1,2,3,4,5
        DB   data and 0ffh,5,377Q,1+2+3+4
signon: DB   'please type your name',cr,lf,0
        DB   'AB' SHR 8, 'C', 'DE' AND 7FH
```

4.7. The DW Directive.

The DW statement is similar to the DB statement except double precision (two byte) words of storage are initialized. The form is

```
label   DW   e#1, e#2, ..., e#n
```

where e#1 through e#n are expressions which evaluate to 16-bit results. Note that ASCII strings of length one or two characters are allowed, but strings longer than two characters disallowed. In all cases, the data storage is consistent with the 8080 processor: the least significant byte of the expression is stored forst in memory, followed by the most significant byte. Examples are

```
doub:   DW   0ffefh,doub+4,signon-$,255+255
        DW   'a', 5, 'ab', 'CD', 6 shl 8 or llb
```

4.8. The DS Directive.

The DS statement is used to reserve an area of uninitialized memory, and takes the form

```
label   DS   expression
```

where the label is optional. The assembler begins subsequent code generation after the area reserved by the DS. Thus, the DS statement given above has exactly the same effect as the statement

```
label:  EQU  $    ;LABEL VALUE IS CURRENT CODE LOCATION
        ORG  $+expression    ;MOVE PAST RESERVED AREA
```

5. OPERATION CODES.

Assembly language operation codes form the principal part of assembly language programs, and form the operation field of the instruction. In general, ASM accepts all the standard mnemonics for the Intel 8080 microcomputer, which are given in detail in the Intel manual "8080 Assembly Language Programming Manual." Labels are optional on each input line and, if included, take the value of the instruction address immediately before the instruction is issued. The individual operators are listed breifly in the following sections for completeness, although it is understood that the Intel manuals should be referenced for exact operator details. In each case,

e3	represents a 3-bit value in the range 0-7 which can be one of the predefined registers A, B, C, D, E, H, L, M, SP, or PSW.
e8	represents an 8-bit value in the range 0-255
e16	represents a 16-bit value in the range 0-65535

which can themselves be formed from an arbitrary combination of operands and operators. In some cases, the operands are restricted to particular values within the allowable range, such as the PUSH instruction. These cases will be noted as they are encountered.

In the sections which follow, each operation codes is listed in its most general form, along with a specific example, with a short explanation and special restrictions.

5.1. Jumps, Calls, and Returns.

The Jump, Call, and Return instructions allow several different forms which test the condition flags set in the 8080 microcomputer CPU. The forms are

```
JMP  e16    JMP L1       Jump unconditionally to label
JNZ  e16    JMP L2       Jump on non zero condition to label
JZ   e16    JMP 100H     Jump on zero condition to label
JNC  e16    JNC L1+4     Jump no carry to label
JC   e16    JC  L3       Jump on carry to label
JPO  e16    JPO $+8      Jump on parity odd to label
JPE  e16    JPE L4       Jump on even parity to label
JP   e16    JP  GAMMA    Jump on positive result to label
JM   e16    JM  al       Jump on minus to label

CALL e16    CALL S1      Call subroutine unconditionally
CNZ  e16    CNZ  S2      Call subroutine if non zero flag
CZ   e16    CZ   100H    Call subroutine on zero flag
CNC  e16    CNC  S1+4    Call subroutine if no carry set
CC   e16    CC   S3      Call subroutine if carry set
CPO  e16    CPO  $+8     Call subroutine if parity odd
CPE  e16    CPE  S4      Call subroutine if parity even
CP   e16    CP   GAMMA   Call subroutine if positive result
CM   e16    CM   b1$c2   Call subroutine if minus flag

RST  e3     RST  0       Programmed "restart", equivalent to
                         CALL 8*e3, except one byte call
```

RET	Return from subroutine
RNZ	Return if non zero flag set
RZ	Return if zero flag set
RNC	Return if no carry
RC	Return if carry flag set
RPO	Return if parity is odd
RPE	Return if parity is even
RP	Return if positive result
RM	Return if minus flag is set

5.2. Immediate Operand Instructions.

Several instructions are available which load single or double precision registers, or single precision memory cells, with constant values, along with instructions which perform immediate arithmetic or logical operations on the accumulator (register A).

MVI e3,e8	MVI B,255	Move immediate data to register A, B, C, D, E, H, L, or M (memory)
ADI e8	ADI 1	Add immediate operand to A without carry
ACI e8	ACI ØFFH	Add immediate operand to A with carry
SUI e8	SUI L + 3	Subtract from A without borrow (carry)
SBI e8	SBI L AND 11B	Subtract from A with borrow (carry)
ANI e8	ANI $ AND 7FH	Logical "and" A with immediate data
XRI e8	XRI 1111$ØØØØB	"Exclusive or" A with immediate data
ORI e8	ORI L AND 1+1	Logical "or" A with immediate data
CPI e8	CPI ´a´	Compare A with immediate data (same as SUI except register A not changed)
LXI e3,e16	LXI B,1ØØH	Load extended immediate to register pair (e3 must be equivalent to B,D,H, or SP)

5.3. Increment and Decrement Instructions.

Instructions are provided in the 8Ø8Ø repetoire for incrementing or decrementing single and double precision registers. The instructions are

INR e3	INR E	Single precision increment register (e3 produces one of A, B, C, D, E, H, L, M)
DCR e3	DCR A	Single precision decrement register (e3 produces one of A, B, C, D, E, H, L, M)
INX e3	INX SP	Double precision increment register pair (e3 must be equivalent to B,D,H, or SP)
DCX e3	DCX B	Double precision decrement register pair (e3 must be equivalent to B,D,H, or SP)

5.4. Data Movement Instructions.

Instructions which move data from memory to the CPU and from CPU to memory are given below

MOV e3,e3	MOV A,B	Move data to leftmost element from right-most element (e3 produces one of A,B,C D,E,H,L, or M). MOV M,M is disallowed
LDAX e3	LDAX B	Load register A from computed address (e3 must produce either B or D)
STAX e3	STAX D	Store register A to computed address (e3 must produce either B or D)
LHLD e16	LHLD L1	Load HL direct from location e16 (double precision load to H and L)
SHLD e16	SHLD L5+x	Store HL direct to location e16 (double precision store from H and L to memory)
LDA e16	LDA Gamma	Load register A from address e16
STA e16	STA X3-5	Store register A into memory at e16
POP e3	POP PSW	Load register pair from stack, set SP (e3 must produce one of B, D, H, or PSW)
PUSH e3	PUSH B	Store register pair into stack, set SP (e3 must produce one of B, D, H, or PSW)
IN e8	IN Ø	Load register A with data from port e8
OUT e8	OUT 255	Send data from register A to port e8
XTHL		Exchange data from top of stack with HL
PCHL		Fill program counter with data from HL
SPHL		Fill stack pointer with data from HL
XCHG		Exchange DE pair with HL pair

5.5. Arithmetic Logic Unit Operations.

Instructions which act upon the single precision accumulator to perform arithmetic and logic operations are

ADD e3	ADD B	Add register given by e3 to accumulator without carry (e3 must produce one of A, B, C, D, E, H, or L)
ADC e3	ADC L	Add register to A with carry, e3 as above
SUB e3	SUB H	Subtract reg e3 from A without carry, e3 is defined as above
SBB e3	SBB 2	Subtract register e3 from A with carry, e3 defined as above
ANA e3	ANA 1+1	Logical "and" reg with A, e3 as above
XRA e3	XRA A	"Exclusive or" with A, e3 as above
ORA e3	ORA B	Logical "or" with A, e3 defined as above
CMP e3	CMP H	Compare register with A, e3 as above
DAA		Decimal adjust register A based upon last arithmetic logic unit operation
CMA		Complement the bits in register A
STC		Set the carry flag to 1

CMC		Complement the carry flag
RLC		Rotate bits left, (re)set carry as a side effect (high order A bit becomes carry)
RRC		Rotate bits right, (re)set carry as side effect (low order A bit becomes carry)
RAL		Rotate carry/A register to left (carry is involved in the rotate)
RAR		Rotate carry/A register to right (carry is involved in the rotate)
DAD e3	DAD B	Double precision add register pair e3 to HL (e3 must produce B, D, H, or SP)

5.6. Control Instructions.

The four remaining instructions are categorized as control instructions, and are listed below

HLT	Halt the 8080 processor
DI	Disable the interrupt system
EI	Enable the interrupt system
NOP	No operation

6. ERROR MESSAGES.

When errors occur within the assembly language program, they are listed as single character flags in the leftmost position of the source listing. The line in error is also echoed at the console so that the source listing need not be examined to determine if errors are present. The error codes are

D	Data error: element in data statement cannot be placed in the specified data area
E	Expression error: expression is ill-formed and cannot be computed at assembly time
L	Label error: label cannot appear in this context (may be duplicate label)
N	Not implemented: features which will appear in future ASM versions (e.g., macros) are recognized, but flagged in this version)
O	Overflow: expression is too complicated (i.e., too many pending operators) to computed, simplify it
P	Phase error: label does not have the same value on two subsequent passes through the program

R	Register error: the value specified as a register is not compatible with the operation code
V	Value error: operand encountered in expression is improperly formed

Several error message are printed which are due to terminal error conditions

NO SOURCE FILE PRESENT	The file specified in the ASM command does not exist on disk
NO DIRECTORY SPACE	The disk directory is full, erase files which are not needed, and retry
SOURCE FILE NAME ERROR	Improperly formed ASM file name (e.g., it is specified with "?" fields)
SOURCE FILE READ ERROR	Source file cannot be read properly by the assembler, execute a TYPE to determine the point of error
OUTPUT FILE WRITE ERROR	Output files cannot be written properly, most likely cause is a full disk, erase and retry
CANNOT CLOSE FILE	Output file cannot be closed, check to see if disk is write protected

7. A SAMPLE SESSION.

The following session shows interaction with the assembler and debugger in the development of a simple assembly language program.

```
ASM SORT   assemble SORT.ASM

CP/M ASSEMBLER - VER 1.0

015C next free address
003H USE FACTOR % of table used 00 TO FF (hexadecimal)
END OF ASSEMBLY

DIR SORT.*

SORT      ASM source file
SORT      BAK backup from last edit
SORT      PRN print file (contains tab characters)
SORT      HEX machine code file
A>TYPE SORT.PRN

                              source line

machine code location ;     SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
                      ;     START AT THE BEGINNING OF THE TRANSIENT PROGRAM AR
 0100                       ORG     100H
      generated machine code
 0100 214601      SORT:     LXI     H,SW     ;ADDRESS SWITCH TOGGLE
 0103 3601                  MVI     M,1      ;SET TO 1 FOR FIRST ITERATION
 0105 214701                LXI     H,I      ;ADDRESS INDEX
 0108 3600                  MVI     M,0      ;I = 0
                  ;
                  ;         COMPARE I WITH ARRAY SIZE
 010A 7E          COMP:     MOV     A,M      ;A REGISTER = I
 010B FE09                  CPI     N-1      ;CY SET IF I < (N-1)
 010D D21901                JNC     CONT     ;CONTINUE IF I <= (N-2)
                  ;
                  ;         END OF ONE PASS THROUGH DATA
 0110 214601                LXI     H,SW     ;CHECK FOR ZERO SWITCHES
 0113 7EB7C20001            MOV A,M! ORA A! JNZ SORT ;END OF SORT IF SW=0
                  ;
 0118 FF                    RST 7            ;GO TO THE DEBUGGER INSTEAD OF REB
                  ; truncated
                  ;         CONTINUE THIS PASS
                  ;         ADDRESSING I, SO LOAD AV(I) INTO REGISTERS
 0119 5F16002148CONT:       MOV E,A! MVI D,0! LXI H,AV! DAD D! DAD D
 0121 4E792346              MOV C,M! MOV A,C! INX H! MOV B,M
                  ;         LOW ORDER BYTE IN A AND C, HIGH ORDER BYTE IN B
                  ;
                  ;         MOV H AND L TO ADDRESS AV(I+1)
 0125 23                    INX     H
                  ;
                  ;         COMPARE VALUE WITH REGS CONTAINING AV(I)
 0126 965778239E            SUB M! MOV D,A! MOV A,B! INX H! SBB M   ;SUBTRACT
                  ;
                  ;         BORROW SET IF AV(I+1) > AV(I)
 012B DA3F01                JC      INCI     ;SKIP IF IN PROPER ORDER
                  ;
                  ;         CHECK FOR EQUAL VALUES
 012E B2CA3F01              ORA D! JZ INCI ;SKIP IF AV(I) = AV(I+1)
```

```
0132 56702B5E           MOV D,M! MOV M,B! DCX H! MOV E,M
0136 712B722B73         MOV M,C! DCX H! MOV M,D! DCX H! MOV M,E
                  ;
                  ;     INCREMENT SWITCH COUNT
013B 21460134           LXI H,SW! INR M
                  ;
                  ;     INCREMENT I
013F 21470134C3INCI:    LXI H,I! INR M! JMP COMP
                  ;
                  ;     DATA DEFINITION SECTION
0146 00           SW:   DB      0         ;RESERVE SPACE FOR SWITCH COUNT
0147              I:    DS      1         ;SPACE FOR INDEX
0148 050064001EAV:      DW      5,100,30,50,20,7,1000,300,100,-32767
000A =            N     EQU     ($-AV)/2          ;COMPUTE N INSTEAD OF PRE
015C   equate value     END
A>TYPE SORT.HEX

:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:07014000470134C30A01006E
:10014800050064001E00320014000700E8032C01BB
:0401580064000180BE
:0000000000
```

machine code in HEX format

```
A>DDT SORT.HEX   start debug run

16K DDT VER 1.0
NEXT  PC
015C 0000   default address (no address on END statement)
-XP

P=0000 100   Change PC to 100

-UFFFF   untrace for 65535 steps
                                                                 abort with rubout
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI  H,0146*0100
-T10   trace 10₁₆ steps

C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI  09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JNC  0119
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0110 LXI  H,0146
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0113 MOV  A,M
C1Z0M1E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0114 ORA  A
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0115 JNZ  0100
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M*010B
-A10D

010D  JC 119   change to a jump on carry               stopped at 10BH
0110
```

```
-XP
P=010B 100    reset program counter back to beginning of program
-T10    trace execution for 10H steps

C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI  09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JC   0119    Altered instructi
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0119 MOV  E,A
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011A MVI  D,00
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011C LXI  H,0148
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=011F DAD  D
C0Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=0120 DAD  D
C0Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=0121 MOV  C,M
C0Z0M1E0I0 A=00 B=0005 D=0000 H=0148 S=0100 P=0122 MOV  A,C
C0Z0M1E0I0 A=05 B=0005 D=0000 H=0148 S=0100 P=0123 INX  H
C0Z0M1E0I0 A=05 B=0005 D=0000 H=0149 S=0100 P=0124 MOV  B,M*0125
-L100
                                                    Automatic
0100  LXI  H,0146                                   breakpoint
0103  MVI  M,01
0105  LXI  H,0147
0108  MVI  M,00
010A  MOV  A,M          list some code
010B  CPI  09           from 100H
010D  JC   0119
0110  LXI  H,0146
0113  MOV  A,M
0114  ORA  A
0115  JNZ  0100
-L

0118  RST  07           list more
0119  MOV  E,A
011A  MVI  D,00
011C  LXI  H,0148
- abort list with rubout
-G,118    start program from current PC (0125H) and run in real time to 11BH

*0127  stopped with an external interrupt 7 from front panel (program was
-T4    look at looping program in trace mode            looping indefinitely)

C0Z0M0E0I0 A=38 B=0064 D=0006 H=0156 S=0100 P=0127 MOV  D,A
C0Z0M0E0I0 A=38 B=0064 D=3806 H=0156 S=0100 P=0128 MOV  A,B
C0Z0M0E0I0 A=00 B=0064 D=3806 H=0156 S=0100 P=0129 INX  H
C0Z0M0E0I0 A=00 B=0064 D=3806 H=0157 S=0100 P=012A SBB  M*012B
-D148
                                    data is sorted, but program doesn't stop.
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D.,.........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
-G0  return to CP/M

DDT SORT.HEX  reload the memory image

16K DDT VER 1.0
NEXT  PC
015C 0000
-XP

P=0000 100  set PC to beginning of program

-L10D  list bad opcode

010D  JNC  0119
0110  LXI  H,0146
- abort list with rubout

-A10D  assemble new opcode

010D  JC 119

0110

-L100  list starting section of program

0100  LXI  H,0146
0103  MVI  M,01
0105  LXI  H,0147
0108  MVI  M,00
- abort list with rubout

-A103  change "switch" initialization to 00

0103  MVI M,0

0105

-^C  return to CP/M with ctl-C (G0 works as well)

SAVE 1 SORT.COM  save 1 page (256 bytes, from 100H to 1FFH) on disk in case
                                                              we have to reload later

A>DDT SORT.COM  restart DDT with
                               saved memory image
16K DDT VER 1.0
NEXT  PC
0200 0100 "COM" file always starts with address 100H
-G  run the program from PC=100H

*0118  programmed stop (RST7) encountered
-D148
                                              data properly sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D.,.........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

-G0  return to CP/M
```

```
ED SORT.ASM↲ make changes to original program
  ctl-Z
*N,0^Z0TT↲ find next ",0"
        MVI     M,0     ;I = 0
*-↲ up one line in text
        LXI     H,I     ;ADDRESS INDEX
*-↲ up another line
        MVI     M,1     ;SET TO 1 FOR FIRST ITERATION
*KT↲ kill line and type next line
        LXI     H,I     ;ADDRESS INDEX
*I↲ insert new line
        MVI     M,0     ;ZERO SW
*T↲
        LXI     H,I     ;ADDRESS INDEX
*NJNC^Z0T↲
        JNC*T↲
        CONT    ;CONTINUE IF I <= (N-2)
*-2DIC^Z0LT↲
        JC      CONT    ;CONTINUE IF I <= (N-2)
*E↲
              source from disk A
               hex to disk A
ASM SORT.AAZ↲ skip prn file

CP/M ASSEMBLER - VER 1.0

015C next address to assemble
003H USE FACTOR
END OF ASSEMBLY

DDT SORT.HEX↲ test program changes

16K DDT VER 1.0
NEXT  PC
015C 0000
-G100↲

*0118
-D148↲
                                data sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D.,.........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

- abort with rubout

-G0↲ return to CP/M - program checks OK.
```