# SOFTWARE REFERENCE MANUAL

# HDOS SYSTEM

*Chapter 4*

## HEATH ASSEMBLY LANGUAGE ASM



HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

595-2338
Printed in the United States of America

## TABLE OF CONTENTS

# WRITING ASSEMBLY LANGUAGE PROGRAMS

The Heath Assembly Language program (ASM) lets you use source (symbolic) programs using letters, numbers, and symbols that are meaningful as they are abbreviated in English statements. These source programs must be generated with the Heath Text Editor (EDIT). ASM assembles the source program into a listing and an object program in absolute binary format executable by your Computer.

This Manual assumes that you are already familiar with the writing of assembly language programs. Also, because of the many cross-references in this Section, we recommend that you read all of this Section to get a good "feel" for ASM.

ASM is designed to produce programs which run in an H8/H89 system; therefore, it assembles 8080 symbolic assembly code. When it is used with 16K of memory, ASM provides for approximately 350 user-defined symbols.

This Software reference Manual presumes that you have read the Operation Manual and are familiar with the 8080 instruction set, I/O formats, memory formats, and front panel configuration. A thorough knowledge of these facts is vital to efficient assembly language programming.

# THE CHARACTER SET

The Heath Assembly Language source program is composed of symbols, numbers, expressions, symbolic instructions, argument separators, assembly directives, and line terminators, all using ASCII characters. Those characters that are acceptable to ASM are listed below.

1. The letters A through Z (lower case letters are acceptable for quoted strings and comments).

2. The numerals 0 through 9.

3. The characters period (.) and dollar sign ($), which are considered alphabetic.

4. The symbols:

   : = % # ( ) , ; " ' + - _ ! ?

5. BLANKS and TABS.

# STATEMENTS

A source program is composed of a sequence of statements, designed to solve a problem. Each statement must be on a single line.

A statement is composed of up to four fields, identified by the order of appearance and separated by BLANKS and TABS. The four fields are:

LABEL OPCODE OPERAND COMMENT

The label on comment fields are optional. The opcode and operand fields are interdependent; either may be omitted, depending upon the contents of the other.

## The Label Field

The label field always starts in column one. A label is a user-defined symbol assigned the current value of the memory location counter. It is a symbolic means of referring to a specific memory location within a program. Most statements do not require a label. If you do not want a label, column one must be left blank or contain a TAB. Although the label is usually used to allow symbolic reference to the address of the labeled instruction, the SET and EQU pseudos make special use of the label field.

A label must start with an alphabetic character, and it consists entirely of alphabetic or numeric characters. The maximum length of a label is 7 characters. Note that the characters "$" and "." are considered alphabetic. Therefore, the following are valid labels.

```
A A3 . C9D4  START .. $END  END!PGM
```

For example, if the current location counter is set to 042 200 and the statement

```
START    MOV A,B
```

is the next statement, the assembler assigns the value 042 200 to the label START. Subsequent references to START refer to location 042 200.

## The Opcode Field

All statements (except the comment statements) must have an opcode field. The opcode field need not be located in any particular column. However, it must be separated from the label field by at least one blank or TAB. If no label is specified, the opcode field may start in or after column 2.

The opcode is either an instruction mnemonic or an assembler directive. When the entry in the opcode field is an instruction mnemonic, it specifies a machine operation to be performed on any following operands. When it is an assembler directive, it specifies certain functions or actions to be performed by the assembler during program assembly.

The opcode field is terminated by a blank, a TAB, or the end of a line.

## The Operand Field

The operand field follows the opcode field and must be separated from it by at least one blank or TAB. Not all opcodes require operands. The operand contains information used by a machine instruction or, in the case of assembler directives (pseudo opcodes or pseudo ops), it contains information to be used by the pseudo op.

Operands may be symbols, expressions, or numbers. When multiple operands appear with a statement, each is separated from the next by a comma. An operand may be followed by a comment.

The operand field is terminated by a blank or TAB when followed by a comment, or by the end of a line when the operand ends the assembly statement. For example,

```
START   MOV A,B   THIS IS A COMMENT
```

The TAB between START and MOV terminates the label field; the blank between MOV and A,B terminates the opcode field and begins the operand field. The comma separates the operands A and B and the TAB terminates the operand field and begins the comment field.

## The Comment Field

The comment field follows the operand field, or the opcode field if no operand field is present. It must be separated from its preceding field by at least one blank or TAB. The comment field is not processed by the assembler and it is designed to contain documentary information. The comment field is optional and may contain any printing ASCII character. All other characters, even those with special significance to the assembler, are ignored by the assembler when used in the comment field.

A statement with an asterisk (*) in column one is taken as a comment statement and is not otherwise processed by the assembler. A totally blank line is also taken as a comment.

## Format Control

The format of an assembly language program is controlled by the blank and TAB characters. Format control is primarily used to produce a program which is easily read. Format control has no effect on the assembly process of the source program. The following two statements are interpreted identically. The first one uses blanks and the second uses TABS.

```
START MOV A,B  THIS IS A COMMENT
START    MOV A,B    THIS IS A COMMENT
LAB  OPCODE  OPERAND      COMMENT
```

# OPERAND EXPRESSIONS

Except when the opcode is a machine instruction requiring that an 8080 register be specified as the operand, all operand fields may be coded as operand expressions. Such operand expressions are made up of integers, symbols, a special origin symbol, and character strings which may be combined, using certain operators. The operand may also be the origin symbol. The expressions are said to be made up of operators and tokens. No parentheses are allowed nor is any operator precedence recognized. Therefore, evaluation is strictly left to right. The result of any expression must fall between −32,767 and 65,534.

## Operators

ASM recognizes 5 operators. They are:

| | |
|---|---|
| + | Addition of an integer arithmetic expression. |
| − | Subtraction of an integer arithmetic expression. |
| * | Multiplication of an integer arithmetic expression. |
| / | Division of an integer arithmetic expression. |
| − | (unary) negation of a standard integer arithmetic expression. |

Note, the unary minus is valid only as the first character in an expression. The following are examples of legitimate assembler operand expressions.

```
3+5
−2        (unary)
1+2*3          = 3*3 = 9
```

Note that the last example evaluates to 9 rather than 7, as the **assembler does not recognize any operator precedence.** Therefore, it evaluates the expression from left to right.

# Tokens

Heath Assembly Language recognizes four different tokens: integers, symbols, character strings, and the origin symbol. Each of these tokens has the limitations described in the following sections.

## INTEGERS

Decimal integers ranging from 0 to 65,535 are allowed, but no decimal place may be specified. The radix of an integer expressions is assumed to be decimal. However, you may specify binary, octal, offset octal, decimal, or hexadecimal. Specify them by using a post-radix symbol following the integer expression.

| | |
|---|---|
| B | Binary |
| O or Q | Octal |
| D | Decimal |
| H | Hexadecimal |
| A | Offset Octal |

For example:

| EXPRESSION | RADIX | DECIMAL VALUE |
|---|---|---|
| 000 00011B | Binary | 3 |
| 160Q | Octal (also 160O) | 112 |
| 3200 | Decimal (also 3200D) | 3200 |
| 77000A | Offset octal | 16128 |
| 021AH | Hexadecimal | 282 |

| LEGAL INTEGER EXPRESSIONS | ILLEGAL INTEGER EXPRESSIONS | COMMENTS |
|---|---|---|
| 232 | 232.1 | Decimals may not be specified |
| 10111B | 226B | Not a binary number |
| 177Q | 888 | Not an octal number |
| A1FH | 21C | No hex radix specified |

If an integer expression evaluates to less than −32,767, or greater than 65,534, an error code is flagged.

## SYMBOLS

An expression may contain any user defined symbol. Although most symbols do not need to be defined sequentially before the referencing statement, some pseudo operators require all their operand symbols to be defined in earlier statements in the program. Such operators are said to require "pass one evaluation" and are documented in "The 8080 Opcodes" (Page 4-10). All symbols must consist of legal ASM characters.

### The # Symbol

If the pound sign (#) is the first character in an expression, the expression is evaluated as a 16-bit expression. After the expression is evaluated, the resultant value is masked to an 8-bit equivalent. Once this is done, a 16-bit operand may be referenced in an instruction requiring 8 bits without causing an overflow (V) error. For example:

```
MVI     H,ADDR/256
MVI     L,#ADDR         (HL) = 16 bit address
```

In this example, the first line of code loads the H and L register pair (16-bit register) with the binary value associated with the label "ADDR" divided by 256. The second line of code immediately loads the L register (an 8-bit register) with the lower 8-bits of the binary value equated to the symbol ADDR in the symbol table. This process does not cause an overflow error, as the 16-bit binary equivalent of ADDR is masked to the least significant 8-bits before it is moved into the 8-bit L register.

## CHARACTER STRING

A character string consisting of one or two legal characters may be used as a token in an ASM expression. Such a character string is enclosed in a single quote (apostrophe). For example:

| | |
|---|---|
| 'A' | The character A (Value 101Q) |
| 'GL' | The character string GL (Value 107 114A) |
| ' " ' | The character quotation mark (Value 042Q) |

## THE ORIGIN SYMBOL (*)

The current value of the origin counter may be referenced with the special symbol asterisk (*). NOTE: The assembler decides from the expression context whether the asterisk (*) represents the origin counter or is the multiplication operator. For example, the program

```
      ORG    10
A     EQU    ***
```

defines the symbol A to have the value 100. The first statement, "ORG 10," sets the origin counter to the value 10. In the second statement, the label A is equated with the first asterisk, which the assembler presumes to be the symbol for the origin counter. This is multiplied by the third symbol, which the assembler also presumes to be the origin symbol. However, the middle asterisk is taken as the multiplication operator.

## THE 8080 OPCODES†

Heath Assembly Language supports the standard 8080 machine opcodes. A review of the 8080 instruction set is presented on the following pages. Included in this review is a discussion of instruction and data formats, addressing modes, conditions flags, the symbols or abbreviations used in describing the 8080 instruction set, and the discussion of the format used to describe each instruction.

The 8080 instruction set includes five different types of instructions:

- **Data Transfer Group** — move data between registers or between memory and registers.
- **Arithmetic Group** — add, subtract, increment, or decrement data in registers or in memory.
- **Logical Group** — AND, OR, EXCLUSIVE-OR, compare, rotate, or complement data in registers or in memory.
- **Branch Group** — conditional and unconditional jump instructions, subroutine call instructions, and return instructions.
- **Stack, I/O and Machine Control Group** — includes I/O instructions, as well as instructions for maintaining the stack and internal control flags.

†Portions of this section are reprinted with the primission of Intel Corporation (Copyright, 1976).

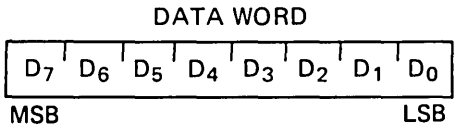
# Terms, Symbols, & Nomenclature

## INSTRUCTION AND DATA FORMATS

Memory for the 8080 is organized into 8-bit quantities called bytes. Each byte has a unique 16-bit binary address corresponding to its sequential position in memory.

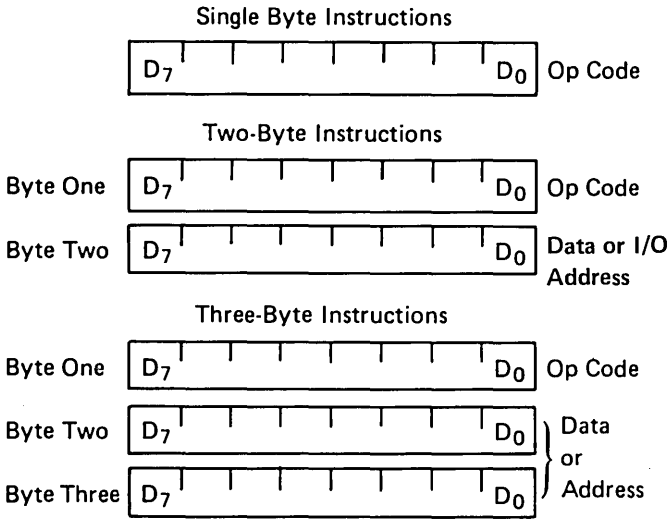
The 8080 can directly address up to 65,535 bytes of memory, which may consist of both read-only memory (ROM) elements and random-access memory (RAM) elements (read/write memory).

Data in the 8080 is stored in the form of 8-bit binary integers:




When a register or data word contains a binary number, it is necessary to establish the order in which the bits of the number are written. In the Intel 8080, BIT 0 is referred to as the **Least Significant Bit (LSB)**, and BIT 7 (of an 8-bit number) is referred to as the **Most Significant Bit (MSB).**

The 8080 program instructions may be one, two, or three bytes in length. Multiple byte instructions must be stored in successive memory locations; the address of the first byte is always used as the address of the instructions. The exact instruction format will depend on the particular operation to be executed.



## ADDRESSING MODES

Often, the data that is to be operated on is stored in memory. When multi-byte numeric data is used, the data, like instructions, is stored in successive memory locations with the least significant byte first, followed by increasingly significant bytes. The 8080 has four different modes for addressing data stored in memory or in registers:

- **Direct** — Bytes 2 and 3 of the instruction contain the exact memory address of the data item (the low-order bits of the address are in byte 2, the high-order bits in byte 3).

- **Register** — Specifies the register or register pair in which the data is located.

- **Register Indirect** — Specifies a register pair which contains the memory address where the data is located (the high-order bits of the address are in the first register of the pair, the low-order bits in the second).

- **Immediate** — Contains the data itself. This is either an 8-bit quantity or a 16-bit quantity (least significant byte first, most significant byte second).

Unless directed by an interrupt or branch instruction, the execution of instructions proceeds through consecutively increasing memory locations. A branch instruction can specify the address of the next instruction to be executed in one of two ways:

- **Direct** — The branch instruction contains the address of the next instruction to be executed. (Except for the "RST" instruction, byte 2 contains the low-order address and byte 3 the high-order address.)

- **Register Indirect** — The branch instruction indicates a register pair which contains the address of the next instruction to be executed. (The high-order bits of the address are in the first register of the pair, the low-order bits in the second.)

The RST instruction is a special 1-byte call instruction (usually used during interrupt sequences). RST includes a 3-bit field; program control is transferred to the instruction whose address is eight times the contents of this 3-bit field.

## CONDITION FLAGS

There are five condition flags associated with the execution of instructions on the 8080. They are Zero, Sign, Parity, Carry, and Auxiliary Carry, and are each represented by a 1-bit register in the CPU. A flag is "set" by forcing the bit to 1; and "reset" by forcing the bit to 0.

Unless indicated otherwise, when an instruction affects a flag, it affects it in the following manner.

Zero: If the result of an instruction has the value 0, this flag is set. Otherwise it is reset.

Sign: If the most significant bit of the result of the operation has the value 1, this flag is set. Otherwise it is reset.

Parity If the modulo 2 sum of the bits of the result of the operation is 0 (i. e., if the result has even parity), this flag is set. Otherwise it is reset (i. e., if the result has odd parity).

Carry: If the instruction resulted in a carry (from addition), or a borrow (from subtraction or a comparison) out of the high-order bit, this flag is set. Otherwise it is reset.

Auxiliary Carry: If the instruction caused a carry out of bit 3 and into bit 4 of the resulting value, the auxiliary carry is set. Otherwise it is reset. This flag is affected by single precision additions, subtractions, increments, decrements, comparisons, and logical operations, but is principally used with additions and increments preceding a DAA (Decimal Adjust Accumulator) instruction.

### Symbols and Abbreviations

The following symbols and abbreviations are used in the subsequent description of the 8080 instructions:

| SYMBOLS | MEANING |
|---|---|
| accumulator | Register A |
| addr | 16-bit address quantity |
| data | 8-bit data quantity |
| data 16 | 16-bit data quantity |
| byte 2 | The second byte of the instruction |
| byte 3 | The third byte of the instruction |
| port | 8-bit address of an I/O device |
| r, r1, r2 | One of the registers A,B,C,D,E,H,L |
| DDD, SSS | The bit pattern designating one of the registers A, B, C, D, E, H, L (DDD = destination, SSS = source): |

| DDD or SSS | REGISTER NAME |
|---|---|
| 111 | A |
| 000 | B |
| 001 | C |
| 010 | D |
| 011 | E |
| 100 | H |
| 101 | L |

rp One of the register pairs:

B represents the B, C pair with B as the high-order register and C as the low-order register;

D represents the D, E pair with D as the high-order register and E as the low-order register;

H represents the H, L pair with H as the high-order register and L as the low-order register;

SP represents the 16-bit stack pointer register.

RP The bit pattern designating one of the register pairs B, D, H, SP:

| RP | REGISTER PAIR |
|---|---|
| 00 | B-C |
| 01 | D-E |
| 10 | H-L |
| 11 | SP |

rh The first (high-order) register of a designated register pair.

rl The second (low-order) register of a designated register pair.

PC 16-bit program counter register (PCH and PCL are used to refer to the high-order and low-order 8-bits respectively).

SP 16-bit stack pointer register (SPH and SPL are used to refer to the high-order and low-order 8-bits respectively).

rm Bit m of the register r (bits are numbered 7 through 0 from left to right).

Z, S, P, Cy, AC The condition flags:

Zero,
Sign,
Parity,
Carry,
and Auxiliary Carry,
respectively.

NOTE, ASM recognizes the E as well as the Z defining the zero bit. Therefore, JZ (jump zero) or JE (jump equal) are both valid op-codes.

( ) The contents of the memory location or registers enclosed in the parentheses.

← "Is transferred to"

∧ Logical AND

⊻ Exclusive OR

∨ Inclusive OR

\+ Addition

\- Two's complement subtraction

* Multiplication

⟷ "Is exchanged with"

— The one's complement (e. g., $\overline{A}$)

n The restart number 0 through 7

NNN The binary representation 000 through 111 for restart number 0 through 7, respectively.

**Description Format**

The following pages provide a detailed description of the instruction set of the 8080. Each instruction is described in the following manner:

1. The ASM format, consisting of the opcode and operand fields, is printed in **BOLDFACE** on the left side of the first line.

2. The name of the instruction is enclosed in parentheses at the center of the first line.

3. The next line(s) contain a symbolic description of the operation of the instruction.

4. This is followed by a narrative description of the operation of the instruction.

5. The following line(s) contain the binary fields and patterns that comprise the machine instruction.

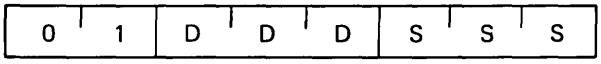
6. The last two lines contain incidental information about the execution of the instruction. The number of machine cycles and states required to execute the instruction are listed first. If the instruction has two possible execution times, as in a conditional jump, both times will be listed, separated by a slash. Next, any significant data addressing modes (see "Addressing Modes," Page 4-12) are listed. The last line lists any of the five Flags that are affected by the execution of the instruction.

## Data Transfer Group

This group of instructions transfers data to and from registers and memory. **Condition flags are not affected** by any instruction in this group.

**MOV r1, r2** (Move Register)

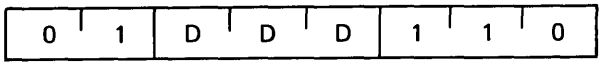
(r1)← (r2)
The content of register r2 is moved to register r1.




Cycles: 1 Addressing: register
States: 5 Flags: none

**MOV r, M** (Move from memory)

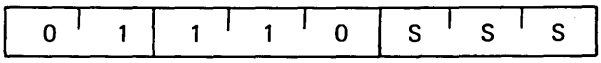
(r) ← ((H) (L))
The content of the memory location whose address is in registers H and L is moved to register r.




Cycles: 2 Addressing: reg. indirect
States: 7 Flags: none

**MOV M, r** (Move to memory)

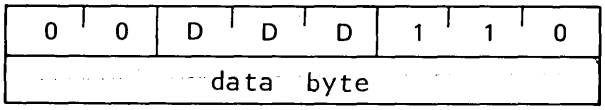
((H) (L)) ← (r)
The content of register r is moved to the memory location whose address is in registers H and L.




Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

**MVI r, data** (Move to register immediate)

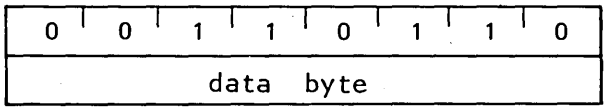
(r) ← (byte 2)
The content of byte 2 of the instruction is moved to register r.




Cycles: 2
States: 7
Addressing: immediate
Flags: none

**MVI M, data** (Move to memory immediate)

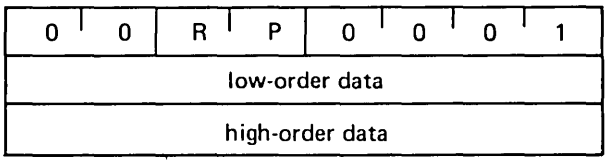
((H) (L)) ← (byte 2)
The content of byte 2 of the instruction is moved to the memory location whose address is in registers H and L.




Cycles: 3
States: 10
Addressing: immed./reg. indirect
Flags: none

**LXI rp, data 16** (Load register pair immediate)

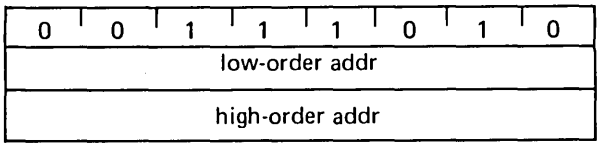
(rh) ← (byte 3),
(rl) ← (byte 2)
Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp. Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp.




Cycles: 3
States: 10
Addressing: immediate
Flags: none

**LDA addr** (Load Accumulator direct)

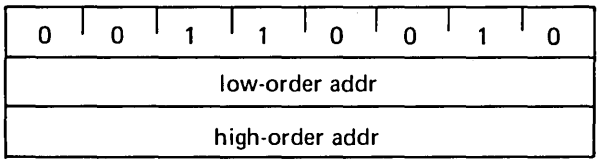
(A) ← ((byte 3) (byte 2))
The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register A.




Cycles: 4
States: 13
Addressing: direct
Flags: none

**STA addr** (Store accumulator direct)

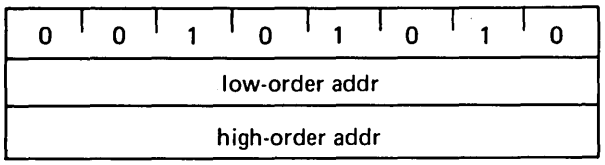
((byte 3) (byte 2)) ← (A)
The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.




Cycles: 4
States: 13
Addressing: direct
Flags: none

**LHLD addr** (Load H and L direct)

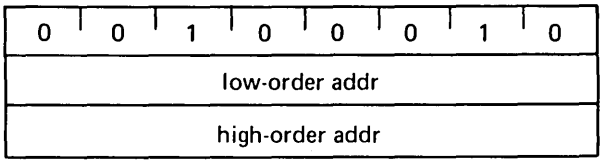
(L) ← ((byte 3) (byte 2))
(H) ← ((byte 3) (byte 2) + 1)
The content of the memory location whose address is specified in byte 2 and byte 3 of the instruction is moved to register L. The content of the memory location at the succeeding address is moved to register H.




Cycles: 5 Addressing: direct
States: 16 Flags: none

**SHLD addr** (Store H and L direct)

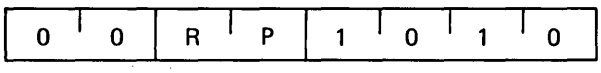
((byte 3) (byte 2)) ← (L)
((byte 3) (byte 2) + 1) ← (H)
The content of register L is moved to the memory location whose address is specified in byte 2 and byte 3. The content of register H is moved to the succeeding memory location.




Cycles: 5 Addressing: direct
States: 16 Flags: none

**LDAX rp** (Load accumulator indirect)

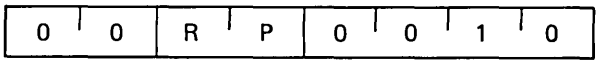
(A) ← ((rp))
The content of the memory location whose address is in the register pair rp is moved to register A. NOTE: Only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.




Cycles: 2 Addressing: reg. indirect
States: 7 Flags: none

**STAX rp** (Store accumulator indirect)

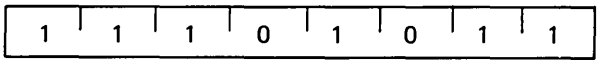
((rp)) ← (A)
The content of register A is moved to the memory location whose address is in the register pair rp. NOTE: Only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.




Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

**XCHG** (Exchange H and L with D and E)

(H) ⟷ (D)
(L) ⟷ (E)
The contents of registers H and L are exchanged with the contents of registers D and E.




Cycles: 1
States: 4
Addressing: register
Flags: none

## Arithmetic Group

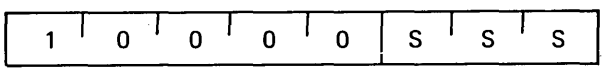
This group of instructions performs arithmetic operations on data in registers and memory.

**Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Carry, and Auxiliary Carry flags according to the standard rules.**

All subtraction operations are performed via two's complement arithmetic and set the carry flag to one to indicate a borrow and clear it to indicate no borrow.

**ADD r** (Add Register)

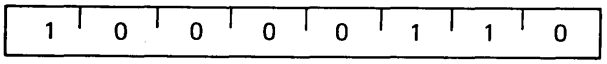
(A) ← (A) + (r)
The content of register r is added to the content of the accumulator. The result is placed in the accumulator.




Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

**ADD M** (Add memory)

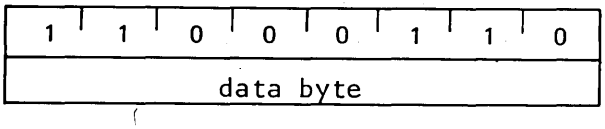
(A) ← (A) + ((H) (L))
The content of the memory location whose address is contained in the H and L registers is added to the content of the accumulator. The result is placed in the accumulator.




Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

**ADI DATA** (add immediate)

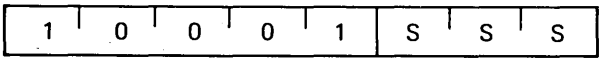
(A) ← (A) + (byte 2)
The content of the second byte of the instruction is added to the content of the accumulator. The result is placed in the accumulator.




Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

**ADC r** (Add Register with carry)

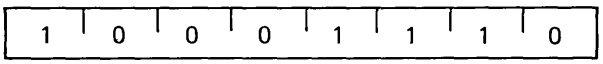
(A) ← (A) + (r) + (CY)
The content of register r and the content of the carry bit are added to the content of the accumulator. The result is placed in the accumulator.




Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

**ADC M** (Add memory with carry)

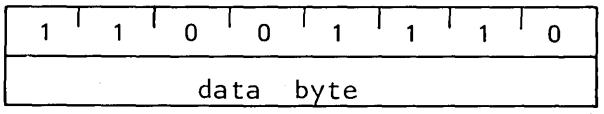
(A) ← (A) + ((H) (L)) + (CY)
The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are added to the accumulator. The result is placed in the accumulator.




Cycles: 2 Addressing: reg. indirect
States: 7 Flags: Z,S,P,CY,AC

**ACI data** (Add immediate with carry)

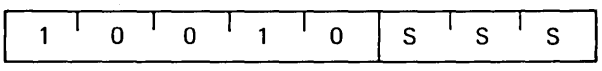
(A) ← (A) + (byte 2) + (CY)
The content of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.




Cycles: 2 Addressing: immediate
States: 7 Flags: Z,S,P,CY,AC

**SUB r** (Subtract Register)

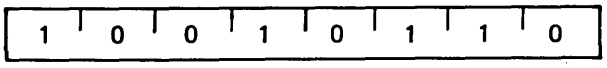
(A) ← (A) — (r)
The content of register r is subtracted from the content of the accumulator. The result is placed in the accumulator.




Cycles: 1 Addressing: register
States: 4 Flags: Z,S,P,CY,AC

**SUB M** (Subtract memory)

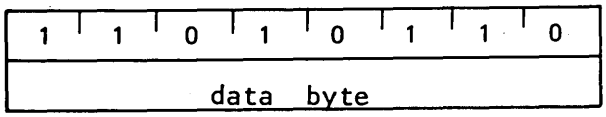
(A) ← (A) — ((H) (L))
The content of the memory location whose address is contained in the H and L registers is subtracted from the content of the accumulator. The result is placed in the accumulator.




Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

**SUI DATA** (Subtract immediate)

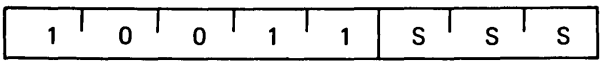
(A) ← (A) — (byte 2)
The content of the second byte of the instruction is subtracted from the content of the accumulator. The result is placed in the accumulator.




Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

**SBB r** (Subtract Register with borrow)

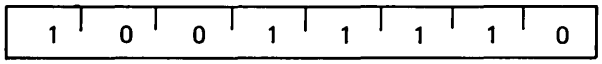
(A) ← (A) — (r) — (CY)
The content of register r and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.




Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

**SBB M** (Subtract memory with borrow)

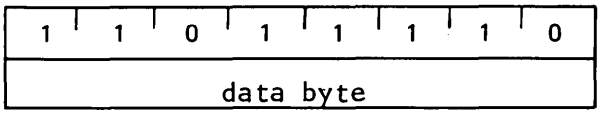
(A) ← (A) — ((H) (L)) — (CY)
The content of the memory location whose address is contained in the H and I registers and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.




Cycles: 2 Addressing: reg. indirect
States: 7 Flags: Z,S,P,CY,AC

**SBI data** (Subtract immediate with borrow)

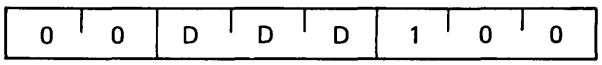
(A) ← (A) — (byte 2) — (CY)
The contents of the second byte of the instruction and the contents of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.




Cycles: 2 Addressing: immediate
States: 7 Flags: Z,S,P,CY,AC

**INR r** (Increment Register)

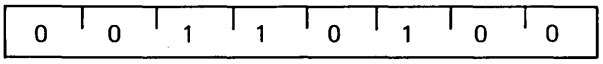
(r) ← (r) + 1
The content of register r is incremented by one. NOTE: All condition flags **except CY** are affected.




Cycles: 1 Addressing: register
States: 5 Flags: Z,S,P,AC

**INR M** (Increment memory)

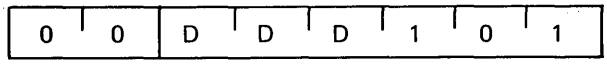
((H) (L)) ← ((H) (L)) + 1
The content of the memory location whose address is contained in the H and L registers is incremented by one. NOTE: All condition flags **except CY** are affected.




Cycles: 3 Addressing: reg. indirect
States: 10 Flags: Z,S,P,AC

**DCR r** (Decrement Register)

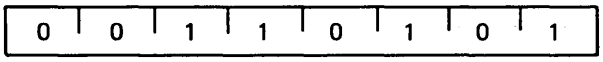
(r) ← (r) — 1
The content of register r is decremented by one. NOTE: All condition flags **except CY** are affected.




Cycles: 1 Addressing: register
States: 5 Flags: Z,S,P,AC

**DCR M** (Decrement memory)

((H) (L)) ← ((H) (L)) — 1
The content of the memory location whose address is contained in the H and L registers is decremented by one. NOTE: All condition flags **except CY** are affected.




Cycles: 3 Addressing: reg. indirect
States: 10 Flags: Z, S, P, AC

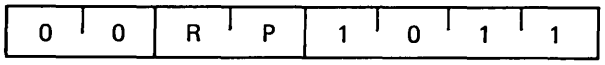
**INX rp** (Increment register pair)

(rh) (rl) ← (rh) (rl) + 1
The content of the register pair rp is incremented by one. NOTE: **No condition flags are affected.**


Cycles: 1 Addressing: register
States: 5 Flags: none

**DCX rp** (Decrement register pair)

(rh) (rl) ← (rh) (rl) — 1
The content of register pair rp is decremented by one. NOTE: **No condition flags are affected.**




Cycles: 1 Addressing: register
States: 5 Flags: none

**DAD rp** (Add register pair to H and L)

(H) (L) ← (H) (L) + (rh) (rl)
The content of register pair rp is added to the content of the register pair H and L. The result is placed in register pair H and L. NOTE: **Only the CY flag is affected.** It is set if there is a carry out of the double precision add; otherwise it is reset.


Cycles: 3 Addressing: register
States: 10 Flags: CY

**DAA** (Decimal Adjust Accumulator)

The eight-bit number in the accumulator is adjusted to form two 4-bit Binary-Coded-Decimal digits by the following process:

1. If the value of the least significant 4 bits of the accumulator is greater than 9 **or** if the AC flag is set, 6 is added to the accumulator.

2. If the value of the most significant 4 bits of the accumulator is now greater than 9, **or** if the CY flag is set, 6 is added to the most significant 4 bits of the accumulator.


Cycles: 1
States: 4
Flags: Z,S,P,CY,AC

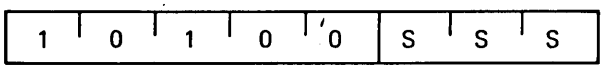
## Logical Group:

This group of instructions performs logical (Boolean) operations on data in registers and memory and on condition flags.

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Auxiliary Carry, and Carry flags according to the standard rules.

**ANA r** (AND Register)

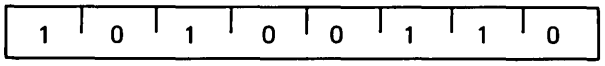
(A) ← (A) ∧ (r)
The content of register r is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**




Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

**ANA M** (AND memory)

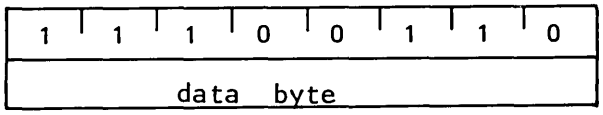
(A) ← (A) ∧ ((H) (L))
The contents of the memory location whose address is contained in the H and L registers is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**




Cycles: 2 Addressing: reg. indirect
States: 7 Flags: Z,S,P,CY,AC

**ANI data** (AND immediate)

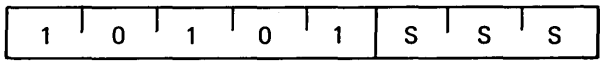
(A) ← (A) ∧ (byte 2)
The content of the second byte of the instruction is logically anded with the contents of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**




Cycles: 2 Addressing: immediate
States: 7 Flags: Z,S,P,CY,AC

**XRA r** (Exclusive OR Register)

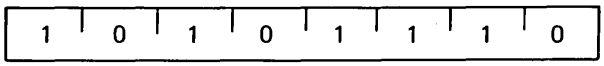
(A) ← (A) ⊻ (r)
The content of register r is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**




Cycles: 1 Addressing: register
States: 4 Flags: Z,S,P,CY,AC

**XRA M** (Exclusive OR Memory)

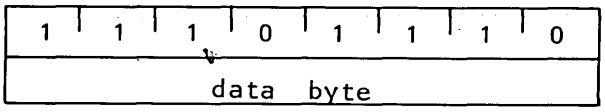
(A) ← (A) ⊻ ((H) (L))
The content of the memory location whose address is contained in the H and L registers is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**




Cycles: 2 Addressing: reg. indirect
States: 7 Flags: Z,S,P,CY,AC

**XRI data** (Exclusive OR immediate)

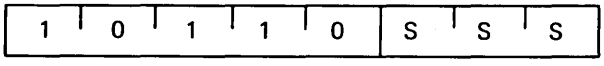
(A) ← (A) ⊻ (byte 2)
The content of the second byte of the instruction is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**




Cycles: 2 Addressing: immediate
States: 7 Flags: Z,S,P,CY,AC

**ORA r** (OR Register)

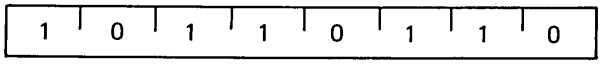
(A) ← (A) ∨ (r)
The content of register r is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**




Cycles: 1 Addressing: register
States: 4 Flags: Z,S,P,CY,AC

**ORA M** (OR memory)

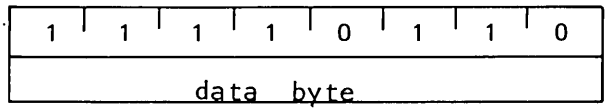
(A) ← (A) ∨ ((H) (L))
The content of the memory location whose address is contained in the H and L registers is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**




Cycles: 2 Addressing: reg. indirect
States: 7 Flags: Z,S,P,CY,AC

**ORI data** (OR Immediate)

(A) ← (A) ∨ (byte 2)
The content of the second byte of the instruction is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**




Cycles: 2 Addressing: immediate
States: 7 Flags: Z,S,P,CY,AC

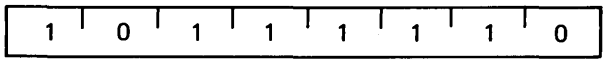
**CMP r** (Compare Register)

(A) — (r)
The content of register r is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A) = (r). The CY flag is set to 1 if (A) < (r).**


Cycles: 1 Addressing: register
States: 4 Flags: Z,S,P,CY,AC

**CMP M** (Compare memory)

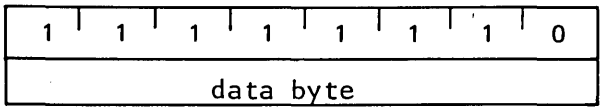
(A) — ((H) (L))
The content of the memory location whose address is contained in the H and L registers is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. The Z flag is set to 1 if (A) = ((H) (L)). The CY flag is set to 1 if (A) < ((H) (L)).




Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

**CPI data** (Compare immediate)

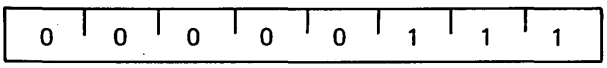
(A) — (byte 2)
The content of the second byte of the instruction is subtracted from the accumulator. The condition flags are set by the result of the subtraction. The Z flag is set to 1 if (A) = (byte 2). The CY flag is set to 1 if (A) < (byte 2).




Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

**RLC** (Rotate left)

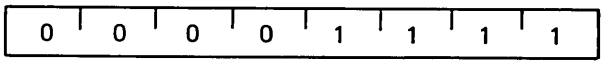
$(A_{n+1}) \leftarrow (A_n)$; $(A_0) \leftarrow (A_7)$
$(CY) \leftarrow (A_7)$
The content of the accumulator is rotated left one position. The low order bit and the CY flag are both set to the value shifted out of the high order bit position. **Only the CY flag is affected.**




Cycles: 1
States: 4
Flags: CY

**RRC** (Rotate right)

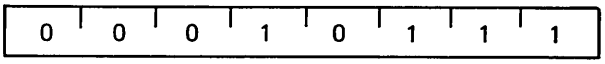
$(A_n) \leftarrow (A_{n-1})$; $(A_7) \leftarrow (A_0)$
$(CY) \leftarrow (A_0)$
The content of the accumulator is rotated right one position. The high order bit and the CY flag are both set to the value shifted out of the low order bit position. **Only the CY flag is affected.**




Cycles: 1
States: 4
Flags: CY

**RAL** (Rotate left through carry)

$(A_{n+1}) \leftarrow (A_n)$; $(CY) \leftarrow (A_7)$
$(A_0) \leftarrow (CY)$
The content of the accumulator is rotated left one position through the CY flag. The low order bit is set equal to the CY flag and the CY flag is set to the value shifted out of the high order bit. **Only the CY flag is affected.**




Cycles: 1
States: 4
Flags: CY

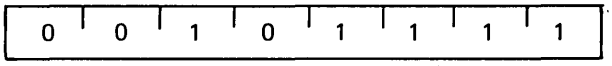
**RAR** (Rotate right through carry)

$(A_n) \leftarrow (A_{n+1})$; $(CY) \leftarrow (A_0)$
$(A_7) \leftarrow (CY)$
The content of the accumulator is rotated right one position through the CY flag. The high order bit is set to the CY flag and the CY flag is set to the value shifted out of the low order bit. **Only the CY flag is affected.**


Cycles: 1
States: 4
Flags: CY

**CMA** (Complement accumulator)

(A) ← (A)
The contents of the accumulator are complemented (zero bits become 1, one bits become 0). **No flags are affected.**




Cycles: 1
States: 4
Flags: none

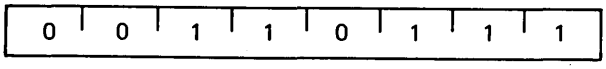
**CMC** (Complement carry)

(CY) ← ($\overline{CY}$)
The CY flag is complemented. **No other flags are affected.**


Cycles: 1
States: 4
Flags: CY

**STC** (Set carry)

(CY) ← 1
The CY flag is set to 1. **No other flags are affected.**




Cycles: 1
States: 4
Flags: CY

## Branch Group

This group of instructions alter normal sequential program flow. **Condition flags are not affected** by any instruction in this group.

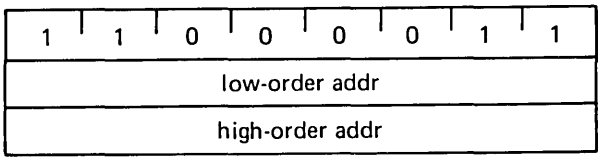
The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register PC (the

program counter). Conditional transfers examine the status of one of the four processor flags to determine if the specified branch is to be executed. The following conditions may be specified:

| CONDITION | CCC | OCTAL |
|---|---|---|
| NE or NZ — not zero (Z=0) | 000 | 0 |
| E or Z — zero (Z=1) | 001 | 1 |
| NC — no carry (CY = 0) | 010 | 2 |
| C — carry (CY = 1) | 011 | 3 |
| PO — parity odd (P = 0) | 100 | 4 |
| PE — parity even (P = 1) | 101 | 5 |
| P — plus (S = 0) | 110 | 6 |
| M — minus (S = 1) | 111 | 7 |

**JMP addr** (Jump)

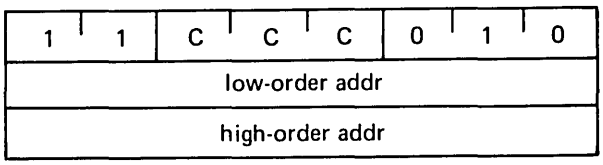
(PC) ← (byte 3) (byte 2)
Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.




Cycles: 3 Addressing: immediate
States: 10 Flags: none

**JNE JNC JPO JP**
**JE JC JPE JM** (Condition jump)

If (CCC),
(PC) ← (byte 3) (byte 2)
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction. Otherwise, control continues sequentially.

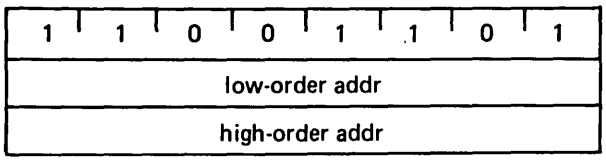



Cycles: 3 Addressing: immediate
States: 10 Flags: none

**CALL addr** (Call)

((SP) — 1) ← (PCH)
((SP) — 2) ← (PCL)
(SP) ← (SP) — 2
(PC) ← (byte 3) (byte 2)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

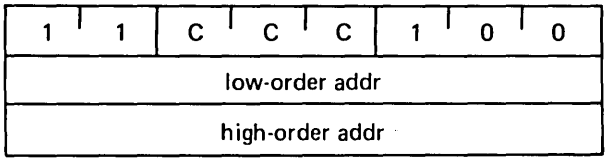



Cycles: 5
States: 17
Addressing: immediate/reg. indirect
Flags: none

**CNE CNC CPO CP**
**CE CC CPE CM** (Condition call)

If (CCC),
((SP) — 1) ← (PCH)
((SP) — 2) ← (PCL)
(SP) ← (SP) — 2
(PC) ← (byte 3) (byte 2)

If the specified condition is true, the actions specified in the CALL instruction (see above) are performed; otherwise, control continues sequentially.

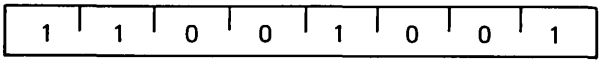



Cycles: 3/5
States: 11/17
Addressing: immediate/reg. indirect
Flags: none

**RET** (Return)

(PCL) ← ((SP)):
(PCH) ← ((SP)) + 1);
(SP) ← (SP) + 2;

The content of the memory location whose address is specified in register SP is moved to the low-order eight bits of register PC. The content of the memory location whose address is one more than the content of register SP is moved to the high-order eight bits of register PC. The content of register SP is incremented by 2.

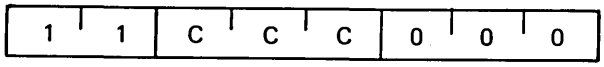



Cycles: 3 Addressing: reg. indirect
States: 10 Flags: none

**RNE RNC**
**RE RC** (Conditional return)

If (CCC),
(PCL) ← ((SP))
(PCH ← ((SP) + 1)
(SP) ← (SP) + 2

If the specified condition is true, the actions specified in the RET instruction (see above) are performed: otherwise, control continues sequentially.



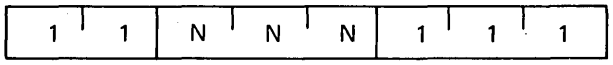

Cycles: 1/3 Addressing: reg. indirect
States: 5/11 Flags: none

**RST n** (Restart)

((SP) — 1) ← (PCH)
((SP) — 2) ← (PCL)
(SP) ← (SP) — 2
(PC) ← 8 * (NNN)

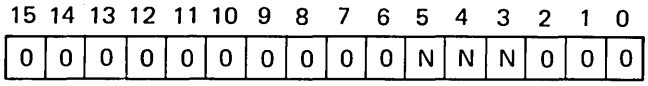
The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP.

The content of register SP is decremented by two. Control is transferred to the instruction whose address is eight times the content of NNN.




Cycles: 3
States: 11

Addressing: reg. indirect
Flags: none

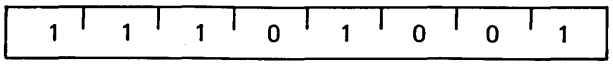



Program Counter After Restart

**PCHL** (Jump H and L indirect — move H and L to PC)

(PCH) ← (H)
(PCL) ← (L)

The content of register H is moved to the high-order eight bits of register PC. The content of register L is moved to the low-order eight bits of register PC.




Cycles: 1
States: 5

Addressing: register
Flags: none

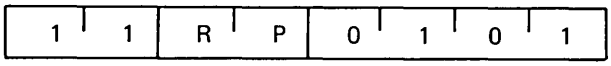
## Stack, I/O, and Machine Control Group

This group of instructions performs I/O, manipulates the Stack, and alters internal control flags. Unless otherwise specified, **condition flags are not affected by any instructions in this group.**

**PUSH rp** (Push)

((SP) — 1) ← (rh)
((SP) — 2) ← (rl)
(SP) ← (SP) — 2

The content of the high-order register of register pair (rp) is moved to the memory location whose address is one less than the content of register SP. The content of the low-order register of register pair (rp) is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. **NOTE: Register pair (rp) = SP may not be specified.**



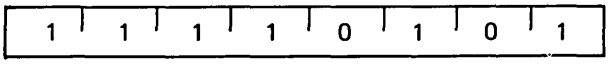

Cycles: 3
States: 11

Addressing: reg. indirect
Flags: none

**PUSH PSW** (Push processor status word)

$((SP) - 1) \leftarrow (A)$
$((SP) - 2)_0 \leftarrow (CY), ((SP) - 2)_1 \leftarrow 1$
$((SP) - 2)_2 \leftarrow (P), ((SP) - 2)_3 \leftarrow 0$
$((SP) - 2)_4 \leftarrow (AC), ((SP) - 2)_5 \leftarrow 0$
$((SP) - 2)_6 \leftarrow (Z), ((SP) - 2)_7 \leftarrow (S)$
$(SP) \leftarrow (SP) - 2$

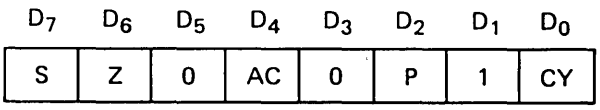
The content of register A is moved to the memory location whose address is one less than register SP. The contents of the condition flags are assembled into a processor status word and the word is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two.




Cycles: 3
States: 11

Addressing: reg. indirect
Flags: none

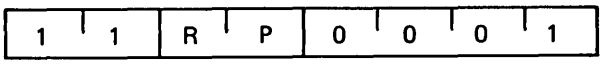
FLAG WORD



**POP rp** (Pop)

(rl) ← ((SP))
(rh) ← ((SP) + 1
(SP) ← (SP) + 2

The content of the memory location whose address is specified by the content of register SP is moved to the low-order register of register pair (rp). The content of the memory location whose address is one more than the content of register SP is moved to the high-order register of register pair (rp). The content of register SP is incremented by 2. **NOTE: Register pair (rp) = SP may not be specified.**

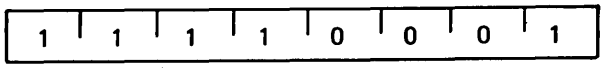



Cycles: 3 Addressing: reg. indirect
States: 10 Flags: none

**POP PSW** (Pop processor status word)

(CY) ← $((SP))_0$
(P) ← $((SP))_2$
(AC) ← $((SP))_4$
(Z) ← $((SP))_6$
(S) ← $((SP))_7$
(A) ← ((SP) + 1)
(SP) ← (SP) + 2

The content of the memory location whose address is specified by the content of register SP is used to restore the condition flags. The content of the memory location whose address is one more than the content of register SP is moved to register A. The content of register SP is incremented by 2.

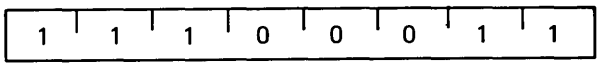



Cycles: 3 Addressing: reg. indirect
States: 10 Flags: Z,S,P,CY,AC

**XTHL** (Exchange stack top with H and L)

(L) ⟷ ((SP))
(H) ⟷ ((SP) + 1)

The content of the L register is exchanged with the content of the memory location whose address is specified by the content of register SP. The content of the H register is exchanged with the content of the memory location whose address is one more than the content of register SP.




Cycles: 5 Addressing: reg. indirect
States: 18 Flags: none

**SPHL** (Move HL to SP)

(SP) ← (H) (L)

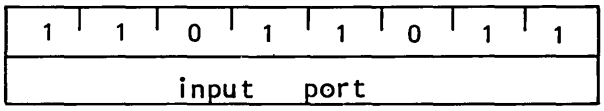
The contents of registers H and L (16 bits) are moved to register SP.


Cycles: 1 Addressing: register
States: 5 Flags: none

**IN port** (Input)

(A) ← (data)

The data placed on the eight bit bidirectional data bus by the specified port is moved to register A.

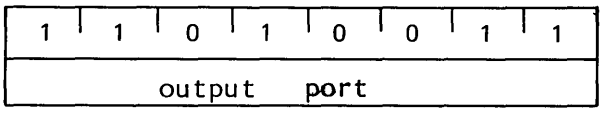



Cycles: 3 Addressing: direct
States: 10 Flags: none

**OUT port** (Output)

(data) ← (A)

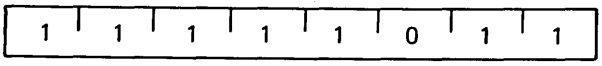
The content of register A is placed on the eight bit bidirectional data bus for transmission to the specified port.




Cycles: 3 Addressing: direct
States: 10 Flags: none

**EI** (Enable interrupt)

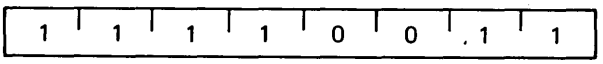
The interrupt system is enabled **following the execution of the next instruction.**




Cycles: 1
States: 4
Flags: none

**DI** (Disable interrupt)

The interrupt system is disabled **immediately following the execution of the DI instruction.**




Cycles: 1
States: 4
Flags: none

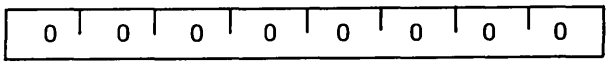
**HLT** (Halt)

The processor is stopped. The registers and flags are unaffected.


Cycles: 1
States: 7
Flags: none

**NOP** (No op)

No operation is performed. The registers and flags are unaffected.




Cycles: 1
States: 4
Flags: none

# PSEUDO OPCODES/ASSEMBLER DIRECTIVES

The Heath Assembly Language supports several assembler directives or, as they are more commonly known, pseudo opcodes or simply pseudo ops. These opcodes are called "pseudo" because they are coded as machine operations. But as their alternate name (assembler directives) indicates, they represent commands to ASM and are not translated as instructions. Some pseudo ops affect the operation of the assembler. Others cause the assembler to generate constants into the generated object code.

## Define Byte, DB

The DB pseudo defines byte contents. The DB pseudo is of the form:

```
Label DB iexp1, . . . . . ,iexpn
```

The integer expressions iexp1 through iexpn are expressions which evaluate to 8-bit values. For the DB pseudo, a long string can be substituted for an expression. The long string is a character string, delimited by single quotes ( ' ), containing one or more characters. You can enclose a quote ( ' ) within a string by coding it as two single quotes. Each of the expressions is converted into an 8-bit binary number and stored in sequential memory locations. A few examples of the DB pseudo are:

```
CR      EQU     15Q
LF      EQU     12Q
        DB      1
        DB      2,3,4
        DB      10,CR,LF,'H8 BASIC',0
```

In each case, the DB pseudo converts the expression into a single byte and stores it in the appropriate memory location. The DB pseudo recognized a character string as a series of expressions. Therefore, each character is converted into its ASCII binary equivalent and is stored in a sequential memory location.

## Define Space, DS

The defined space pseudo (DS) reserves a block of memory during assembly.

The form of the DS pseudo is:

```
LABEL  DS  iexp  COMMENT
```

This pseudo is used, for example, to set up a buffer area or to define any other storage area. The DS pseudo causes the assembler to reserve a number of bytes specified by the expression (iexp) in the operand. These bytes are not preset to any value. Therefore, you should not presume any special original contents. Programs using extensive buffer area should use the DS pseudo to declare this area. Using the DS pseudo significantly shortens the program load time. In the example

```
LINE   DS  80   80 character input line buffer
```

an 80-character input buffer is reserved by a single statement.

## Define Word, DW

The DW pseudo defines word constants. The form of the DW pseudo is:

```
LABEL   DW   iexpl , . . . . . , iexpn
```

The DW pseudo specifies one or more data words iexp through iexpn. Data words are **2-byte** values which are placed into memory space, low order byte first. NOTE: Strings greater than two characters long are not allowed when you are using the DW pseudo.

## Conditional Assembly Pseudo Operators

Frequently, you may want to write a program with certain portions of it that can be turned on or turned off. That is to say, when they are turned on, these portions of the program are assembled. If they are turned off, they are not assembled during that particular assembly. ASM contains three pseudos to aid in conditional assembly. They are:

IF ELSE and ENDIF

**IF**

The IF pseudo conditionally disables assembly of any statements following the IF pseudo operator. The form of the IF pseudo operator is:

```
IF    iexp
```

IF the expression (iexp) evaluates to zero, the statements following the IF pseudo are assembled. If the expression does not evaluate to zero (either negative or positive), any statements in the assembly source code following this expression are skipped until one of the three following pseudos are encountered. The ELSE, ENDIF and END pseudos are not skipped regardless of the value of the expression "iexp".

**ELSE**

The ELSE pseudo toggles the state of the assembly conditions. The ELSE pseudo is of the form:

```
ELSE
```

If the conditional assembly flag is set to skip assembling source code, it is changed so source code is now assembled. If lines of source code prior to encountering the ELSE pseudo are being assembled, those following the ELSE pseudo are skipped until an ELSE, ENDIF, or END is encountered. NOTE: The ELSE segment must appear after an IF statement, but before the associated ENDIF statement.

**ENDIF**

The ENDIF statement indicates the end of a block of source code designated for conditional assembly. The form of the ENDIF pseudo is:

```
ENDIF
```

Assembly resumes regardless of the current assembly state (assembling or skipping) when the ENDIF conditional assembly pseudo occurs.

## End Program, END

The END pseudo indicates the END of a program. The END pseudo takes the form:

```
END iexp
```

where iexp is the program entry point. The program entry point is the memory address where program execution begins. If the END statement is missing, the assembler generates one. If iexp is missing, an error is flagged and ASM uses 042 200A.

## Define Label, EQU

The Equate statement is used to assign an arbitrary value to a symbol. The form of the equate statement is:

```
LABEL    EQU    iexp
```

The equate statement is unique, as it must evaluate on pass one. For this reason, any symbols used within the expression "iexp" must be defined before the assembler encounters the EQU statements. The label is assigned the value of the integer expression "iexp". This label may not be redefined by subsequent use as a label in any other statement. For example,

```
START  EQU  *
```

The label START is set equal to the value of the memory location counter, or

```
START  EQU  100
```

The label START is set equal to 100.

NOTE: If you omit the label, an error is generated.

## Origin Statement, ORG

The Origin statement (ORG) sets the initial value of the memory location counter. The form of the origin statement is:

```
LABEL ORG iexp
```

The expression iexp must evaluate on pass one. Therefore, any symbols used within this expression must be defined before the assembler encounters this statement. When the assembler encounters the ORG statement, the memory location counter is set to the expression value. All subsequent object code generated by the assembler is placed in sequential memory locations, starting at the address given by the expression. It is legal to establish a new origin, either before or after a previous origin. If a label is present, it is given the value iexp. For example:

| | | | |
|---|---|---|---|
| BEGIN | ORG | 42200 A | The program is started at location 042 200 (offset octal) and the label BEGIN is assigned the offset octal value 040 200. This is the lowest address the user (programmer) should use. |
| BEGIN | ORG | START+256 | The memory location counter is set to the previously defined value of the label START +256. The label BEGIN also assumes this value. |

## Set Statement, SET

The SET statement assigns an arbitrary value to a desired symbol. The form of the SET statement is:

```
LABEL   SET   iexp
```

The SET pseudo op differs from the EQU pseudo op in that any label defined in a SET statement can be redefined in a following SET statement as many times as desired in the course of the program. The expression "iexp" must evaluate during pass one. Therefore, any symbols used within the expression "iexp" must be previously defined.

## Xtext Statement, XTEXT

The XTEXT statement is used to include the contents of another file in the assembly. The form of the XTEXT statement is:

```
XTEXT   <fname>
```

When the assembler encounters the XTEXT pseudo operation, it locates the specified file <fname>. <fname> must reside upon a disk device and should contain assembly language statements. Note that it may not contain an END statement, nor another XTEXT statement. The statements in <fname> are assembled into the program where the XTEXT statement was encountered. The XTEXT statement itself is normally listed, but the included statements from <fname> normally are not. The C listing control option is provided to cause them to be listed (see LON and LOF pseudo operations).

The file specification <fname> may specify a device code and an extension. If no extension is specified, ASM assumes an extension of ACM. The only device codes which may be specified are SYØ: and SY1:. You can only specify "SY1:" if you have a second drive installed on your system. If no device is specified, ASM uses the same device that the main source program resides on. If the named file is not found there, then ASM will try device SYØ:. If the named file still cannot be found, then ASM will flag the XTEXT statement with a 'U' error.

The XTEXT statement is normally used to include files containing symbol definitions and commonly used subroutines. For example, Heath provides a file intended to be used with XTEXT, "HDOS.ACM." HDOS.ACM contains symbolic definitions for various operating system function requests. For example, the symbol .EXIT is defined to have the value of 0 (zero). A program including the file HDOS.ACM can use this symbol in generating system requests. This is not only self-documenting, but should a future system revision change the system function codes, the programmer can convert over by simply changing the definitions in HDOS.ACM and reassembling all of his programs, since they all make use of the same definition file, HDOS.ACM.

You can also use XTEXT to include commonly used assembly language subroutines into a program. In this way, a programmer can avoid having to rewrite and redebug the same subroutine for each of his programs. An assembly language programmer will soon build an extensive library of utility subroutines, ready to be XTEXTed into any assembly language program.

## Listing Control

ASM provides a number of pseudo operators which affect the listing mode. They control paging, pagination, titles, and subtitles. The listing control pseudos are used to affect easily read documentation; they do not appear in the program listing.

### TITLE

The pseudo operator TITLE causes a new page title to be used. The form of the title pseudo op is:

```
TITLE    'new title'
```

Unless the assembler is already at the top of a page, a new page of the assembly listing is generated. This page is given the title contained in the string 'new title'.

### STL

The subtitle pseudo (STL) causes a new page subtitle to be set. The form of the subtitle pseudo is:

```
STL    'new subtitle'
```

The subtitle pseudo does not affect pagination. This is to say, it does not generate a new page but simply titles a subsection of the program. Subtitles are frequently used to indicate subroutines or major program modules.

### EJECT

The EJECT pseudo causes a new page to be started. The form of the eject pseudo is:

```
EJECT
```

When ASM processes an EJECT pseudo, the output device is instructed to move to the start of a new page during the listing.

### SPACE

The space pseudo leaves blank lines in the program listing. The form of the space pseudo is:

```
SPACE    iexp1,iexp2
```

During the assembly listing, iexp1 blank lines are left. If the optional expression iexp2 is specified, the assembler checks during a listing to see if the number of lines remaining on the page is greater than or less than iexp2. If there are less than iexp2 lines remaining on the page, the spacing function is skipped and a new page is started, as if an EJECT pseudo was encountered.

**LON (Listing on)**

The LON pseudo operator is used to turn-on listing options. The form of the LON pseudo is:

```
LON    CCC
```

Each option is represented by a single character. The characters for the desired options are supplied as CCC. The options and their default modes (if they are not specified) are:

L Master listing

If this option is enabled, all program lines are listed. If it is disabled, only lines containing errors are listed.
DEFAULT MODE: All program lines are listed (normally enabled; disable using LOF).

I Lists the IF-skipped lines. When this option is enabled, all lines skipped due to IF statements are listed (although they are not assembled).
DEFAULT MODE: The skip lines are not contained in the listing.

G Lists all generated bytes. When this option is enabled, all generated bytes appear on the listing. If more than three bytes are generated by a statement, new lines are generated in the listing to display these bytes. NOTE: The DB pseudo can produce many bytes when you are encoding a string. These are not normally listed.
DEFAULT MODE: Lists a maximum of the 3-bytes generated in each statement.

C Lists XTEXT-included lines. When this option is enabled, all lines included via the XTEXT pseudo operator are listed.
DEFAULT MODE: XTEXT lines are not listed.

### LOF (Listing off)

The LOF pseudo is identical to the LON pseudo except that the selected options are disabled. The form of the LOF pseudo is:

```
LOF    CCC
```

See LON, above, for a description of the control character CCC.

### ERRxx

ASM contains four conditional error pseudo operators. These are of the form:

```
ERRZR    iexp
ERRNZ    iexp
ERRPL    iexp
ERRMI    iexp
```

For each of these pseudo operators, the assembler tests the indicated expression. If the expression matches the expressed error condition, an error code is flagged in the listing. The errors associated with each of the conditional error pseudos are:

| | |
|---|---|
| ERRZR | tests for zero expression |
| ERRNZ | tests for non-zero expression |
| ERRPL | tests for positive expression |
| ERRMI | tests for negative expression |

These pseudo error tests are particularly useful when you make assumptions about the configuration of various program elements or expressions. You can encode these assumptions into ERRxx pseudos. Any change which casues the code to fail generates an error, flagging the programmer during the listing. For example,

```
LXI    H,AREA1
MOV    B,M              (B) = (AREA1)
INX    H
ERRNZ  AREA2-AREA1-1    Assume area 2 follows area 1
MOV    C,M              (C) = (AREA2)
```

If, when the program is assembled, AREA 1 and AREA 2 have been defined differently, an error flag would warn of this mistake.

# GENERATING THE ASSEMBLER

Before you can use the assembler, it must first be generated onto your system disk(s). You can do this by simply copying the file ASM.ABS from the system distribution disk. You will probably want to copy over all files with the .ACM (Assembler Common) extension, via *.ACM. The use of these will be discussed later. Copy the files from the distribution disk via ONECOPY or PIP, if you have a multiple-drive system. See the HDOS Manual, for more information.

## Using The Assembler

In order to use the assembler, you must prepare a source using a text editor, such as EDIT. To get you started, Heath has prepared some short assembly language programs which are in Appendix A.

When the source program is ready, type ASM in response to the HDOS prompt (>). HDOS interprets this command as RUN SYØ:ASM.ABS. If the assembler is on SY1:, then type RUN␣SY1:ASM. In either case, the assembler will type

```
HDOS Assembler Issue #104.00.00
*
```

Note that the issue number may be different, but an issue will be shown. The "*" is the assembler's prompt, asking you to enter a command line in the form

The "*" is the assembler's prompt, asking you to enter a command line in the form

```
<binary fname>,<listing fname>=<source fname>[/SWITCHa.../SWITCHn]
```

The <binary fname> specification tells ASM where to put the generated binary program. The default extension is .ABS. If you do not wish to generate a binary file, omit the file, but not the following comma.

The <listing fname> specification tells ASM where to put the assembly listing. The default extension is .LST. If you specify no listing file, ASM will not generate one. In that case, any program statements that contain errors will be listed on the system console.

The <source fname> specification tells ASM which file contains the assembly language source program. The default extension is .ASM. You must specify this file, it cannot be omitted. The device specified must be a disk (SYØ: or SY1:).

## Switches

There are several switches you may specify at the end of the command line. These switches are all optional, and you can combine any number of them. The legal switches are:

/LARGE

This switch tells ASM that the program you wish to assemble is large, and it should use all the available memory. Normally, when assembling, ASM speeds itself up by letting the operating system use a portion of RAM. However, if your program is so large that the assembler runs out of RAM, you will have to reassemble using the /LARGE switch. This switch causes ASM to use all the available RAM space for itself, with a slightly slower assembly as a result. For systems with only 12K of RAM, ASM will automatically use all of available memory; specifying /LARGE will have no effect.

/ERR

This switch causes ASM to write all program lines with errors in them to the console. Of course, the lines are also written in the normal fashion to the listing file. If no listing file is specified, error lines will be automatically written to the console regardless of the /ERR switch.

/PAGE:nn

ASM writes the source listing file formatted into pages, so that the program can be listed neatly on a printer or a hard-copy terminal. The /PAGE switch tells the assembler how many lines are to appear on a page. Note that this is not the size of the page itself; you will want to leave several lines to form a gap between the pages. Thus, for the standard page size of 66 lines, a specification of /PAGE:60 is about right. This is the default value, so only users with non-standard paper size need specify /PAGE.

/FORM:nn

When the assembler wishes to start a new page for the listing file, it writes an ASCII form feed character into the listing file. This causes an eject to a new page. If your hard copy device will not respond to a form feed in this way, you can use the /FORM:nn switch to have the assembler generate the proper number of line feeds to cause the page eject. The "nn" field is the size of a page (or "form") for your hard copy device. This must be larger than the specified /PAGE value (or default).

The standard size for most computer forms is 66 lines per page; thus, /FORM:66 should be specified. If, for example, you had paper that held 40 lines per page and you wished to print only on the top half of each page, you could specify /FORM:40/PAGE:20. This tells ASM that you want to print 20 lines per page, and that each page is 40 lines long. When the /FORM:nn switch is specified, the assembler writes the proper number of carriage-return line feeds to the listing file, instead of the form feed character.

/LON:ccc

The /LON switch is used to override the listing options specified (via the LON and LOF pseudo instructions) in the assembly language source code. "ccc" represents one or more listing options, discussed in the description of the LON pseudo instruction. A listing option selected by the /LON switch cannot be deselected by a LOF pseudo instruction in the program.

/LOF:ccc

The /LOF switch is used to override the listing options specifed (via the LOF and LOF pseudo instructions) in the assembly language source code. "ccc" represents one or more listing options, discussed in the description of the 'LOF' pseudo instruction. A listing option deselected by the /LOF switch cannot be selected by a LON pseudo instruction in the program.

## Command Line Examples

This section shows several example command lines, with a brief discussion of each. These lines all show assemblies of a sample program, DEMO.ASM

```
* DEMO,DEMO=DEMO ®
```

This command would cause the file SYØ:DEMO.ASM to be assembled, with the listing file written to SYØ:DEMO.LST, and the binary file written to SYØ:DEMO.ABS. Note that form feed characters will be used to separate the pages of the listing file.

```
* DEMO.XXX,TT:=DEMO.ASM/FORM:66 ®
```

This command causes the file SYØ:DEMO.ASM to be assembled, with the listing file written directly to the console terminal (deviceTT:). The binary file will be written to file SYØ:DEMO.XXX. This example assumes that the console terminal is a Decwriter II, without the form-feed option. Thus, the /FORM switch was specified so the assembler would space the paper correctly.

```
* ,LP:=DEMO/LOF:L ®
```

This command causes the SYØ:DEMO.ASM file to be assembled, with the listing file written directly to the line printer (device LP:). Since no /FORM switch was specified and the /PAGE switch was defaulted to /PAGE:60, the assembler will write pages of 60 lines (or less) to the line printer, separated by form feed characters. The user in this case wanted a listing of just the errors in his program, without listing all the correct statements. His use of the LOF:L switch specified that no lines were to be listed. Since lines containing errors are always listed on the listing file, the result will be a listing on the printer showing only lines with errors.

```
* =DEMO ®
```

This final example shows the user assembling the program SYØ:DEMO.ASM, and producing no binary or listing files. This form is useful to check a program for assembly errors since, in the absence of a listing file, all assembly errors are printed on the console. Note that no binary file will be generated.

## Errors

All errors detected by the Heath Assembly Language are flagged directly on the listing in the first three columns. One character is flagged for each error detected. If more than one error is detected, the second error character is placed in column 2 and the third error character is placed in column 3.

| CHARACTER | ERROR |
|---|---|
| U | An undefined symbol. The symbol name does not match any symbol in the symbol assignment table. Check for spelling errors or for a completely undefined symbol. |
| R | Illegal register specified. Two different errors can cause this message. A non-8080 register may have been specified, or the instruction was not meaningful for the register. For example, a register pair instruction which refers to a single register. |
| D | Label is doubly defined. The symbolic label has been defined twice in the source program. |
| A | Operand syntax error. The operand expression is improper. For example, it may evaluate to a number >65535, be a divide by zero, or be nonexistent. |
| V | Value exceeds eight bits. The result of an expression is greater than 255. This error is not flagged if the op-code called for a 16-bit operand such as an LXI instruction. |
| F | Format error. A pseudo-op requires a label that is not present in the source code. For example, an EQU pseudo-op requires a label. Too many characters in a label. |
| O | Unrecognized op-code. The op-code in this statement does not belong to the 8080 instruction set, nor does it belong to the ASM pseudo-op instruction code set. Check for spelling errors or for op-codes used from other microprocessor instruction sets. |

| CHARACTER | ERROR |
|---|---|
| P | Error generated by ERRxx pseudo or reference to a doubly defined label. Note the ERRxx pseudos are generated to flag the user when a test expression does not evaluate satisfactorily. |

NOTE: If an assembly generates a great number of errors, it is best to return to the Text Editor, correct as many errors as possible, and reassemble. The reassembly will frequently flag additional errors which are then obvious on the second assembly. If the errors are few, you may load the program and debug it using DBUG. However, this **does not** result in a correct listing.

# APPENDIX A

## *Assembly Language Interface*

## Introduction

The HDOS operating system offers a powerful and yet simple interface to assembly language programs. This section discusses the fundamental system commands necessary to execute a simple assembly language program. The advanced features and facilities of HDOS will be discussed elsewhere.

HDOS provides what is called the "environment" for an assembly language program. It loads the program into memory, sets up the stack, handles console and disk device I/O, and provides other services for the program. In return, a programmer must always remember that his program is not the only one running in the computer, the HDOS program is also running in the same machine. A programmer must: be careful not to write into memory locations reserved for HDOS, be sure his program does not destroy the program stack by loading the stack pointer; be sure his program does not turn off interrupts via the DI instruction (except for very short periods of time), and so forth.

Finally, it is important that assembly language programs use the support and facilities of HDOS rather than "doing it themselves." Using HDOS whenever possible serves two functions: first, it makes the program much more useful and flexible. For example, if your program uses the HDOS console driver rather than communicating directly to the console itself (via IN and OUT instructions), your program automatically takes advantage of the features of the HDOS console system (CTRL-S, CTRL-O, RUBOUT, CTRL-U, etc.) without any extra programming effort for you. Later, when Heath issues a new version of HDOS supporting new devices and/or new features, your program will automatically be able to take advantage of any new feature without having to be modified.

The second reason for using HDOS functions is system compatibility. As mentioned above, new releases of HDOS will be made available periodically. These new versions will fix known bugs, support new devices, and contain powerful new features. Programs which properly use HDOS functions will be able to run under the new versions of HDOS after being reassembled. Programs that "do it themselves" may fail to work under new HDOS releases.

## Writing Your Program

In order to successfully run your assembly language program under HDOS, you must follow the simple format shown in Figure 1. Your program must start with the three lines:

```
        TITLE   "some descriptive title"
        XTEXT   HDOS
        ORG     USERFWA
```

The TITLE statement causes an appropriate title to be printed on the assembly listing. The title you use is not important as long as it is meaningful to you. The XTEXT statement prepares the assembler for the HDOS commands you will be including in your program. These are discussed later in this section. Finally, the ORG statement tells the assembler to assemble your program into the user memory area.

After these three lines you will write your program. The last line in the program must be

```
        END     xxx
```

where xxx is a label in your program. When you run your program (via the RUN command), execution will begin at the label specified in the END statement.

```
        TITLE   "some meaningful title"
        XTEXT   HDOS
        ORG     USERFWA
XXX     (first line of executable code)
        (your program goes here, see Figures 2, 3, and 4 for examples)
        END     xxx
```

**Figure 1**

Required format for assembly language programs.

## Assembling Your Program

The first thing you must do to run an assembly language program is assemble it. This process translates the source language statements into the 8080A binary object codes. A sample program, DEMO.ASM is shown in Figure 2, Page 4-67. You should enter this through the editor. Once you have this program as a source file, you can assemble it. In HDOS command mode, type:

```
>RUN△ASM ↵
*DEMO,DEMO=DEMO ↵
```

Note that you must type the underlined characters; HDOS provides the characters shown without underlining. This command tells the assembler that you want to assemble the file SY0:DEMO.ASM, producing a listing file called SY0:DEMO.LST and producing a binary file SY0:DEMO.ABS. It is this binary file that contains the runable program. If you have a hard copy device, such as a Decwriter, you can copy the file DEMO.LST onto that device for reference during the remainder of this discussion. If you do not have a hard-copy device, you can refer to the listing of the file DEMO.ASM at the back of this section.

Note that the .ASM, .LST, and .ABS extensions are "defaults" provided by ASM. The assembler will use any specified extensions. Since ASM makes use of HDOS facilities for I/O, ASM is also device independent. For example, if you are assembling a program and want to produce the listing output on your "AT:" device, you need not write the listing file to the disk, copy it to "AT:" and then delete it. Instead, type:

```
*DEMO,AT:=DEMO ↵
```

and have the listing written directly to the "AT:" device.

## Executing Your Program

You must specify the starting address, or entry point, of your program in the END statement. Thus, in the program DEMO.ASM, the END statement says that execution is to start at the label ENTRY. When you type:

```
>RUN△DEMO ↵
```

HDOS will load the program into memory and start executing it at the label ENTRY.

## Returning to HDOS

When your program has finished executing, it must return control to HDOS so you can continue to use the operating system. Your program can do an orderly return to HDOS by executing the two instructions:

```
XRA     A
SCALL   .EXIT
```

which will cause control to return to HDOS. The SCALL .EXIT instruction will be the last one your program will execute.

The SCALL is a special HDOS assembler operation that generates a special two-byte call to the HDOS operating system. The symbol .EXIT indicates the particular type of request you want to make. In this case, you are telling HDOS that you are done executing.

Another way to return control to HDOS is to process CTRL-Cs within your program. In your program initialization, set up CTRL-C processing as follows:

```
        .
        .
        .
    LXI H,EXIT
    MVI A,003
    SCALL .CTLC
        .
        .
        .
```

The end of your program will have the exit routine:

```
        .
        .
        .
EXIT XRA A
SCALL .EXIT
```

A CTRL-C entered while your program is running will cause a return to HDOS.

If you have not dismounted or reset your system volume (see Chapter 1) typing CTRL-Z twice will return to HDOS immediately. However, if your program has a bug and cannot respond to CTRL-C or CTRL-Z, you should re-boot the entire system. This will re-initialize the system. You can run your program under DBUG and isolate the problem in a controlled environment.

# Memory Usage

HDOS uses memory locations both below and above your program. It is important that HDOS should know how much of the user memory area, starting at 42200A, your program will be using. In order to be as fast as possible, HDOS will use some of the user RAM area (that part directly below the resident HDOS code) for a work area if the running user program is not using it. Thus, if you are not going to use that RAM, HDOS should be informed so it can use the area. If you are going to use that RAM, HDOS should be informed so it will not use the area for itself.

When you type the command

```
>RUN△<fname> ↵
```

HDOS automatically computes the size of your program as it was assembled. This means that your program must not write into any memory location that you did not declare during the assembly with a DB, DW, or DS statement. For example, if your program needs a 500-byte memory area, you should not write your program in the form:

```
          .         .
          .         .
          .         .
WORK      EQU       *           500 BYTE WORK AREA STARTS HERE
          END       ENTRY
```

and then use the 500-bytes starting at the label WORK. In this case, HDOS would think that your program ended at the label WORK, and have no way of knowing that you were going to use 500 more bytes. You should code the program

```
          .         .
          .         .
          .         .
WORK      DS        500         500 BYTE WORK AREA
          END       ENTRY
```

In this case, HDOS will know that you will be using the 500 bytes at WORK because you declared them in the DS statement.

## Typing Lines and Characters

HDOS provides two commands for writing to the console terminal. These are .PRINT, and .SCOUT.

.PRINT

The .PRINT SCALL is used to print a line of text on the system console. Before you issue the .PRINT SCALL, you must load the address of the first byte of the line to be printed in the H and L registers. For example,

```
        LXI     H,LINE
        SCALL   .PRINT          PRINT THE MESSAGE
          .       .
          .       .
LINE    DB      12Q,'HI THER','E'+200Q
```

would cause the message
HI THERE
to be printed on the system console.

You have probably noticed that the DB statement in the above example contains more than just the character string 'HI THERE'. The first of these additions is the 12Q. This tells the assembler to start the message with the ASCII character 012 (octal). This is the ASCII "New Line" character. Instead of using the ASCII Carriage Return and Line Feed characters, HDOS uses the New Line character. (Note that New Line has the same octal code as Line Feed; since HDOS does not allow Line Feed characters, there is no confusion.) The New Line character causes a new line to be started on the output device. The rational behind the use of New Line instead of carriage return line feed is beyond the scope of this Manual; suffice it to say that the use of New Line gives HDOS a device-independent way to cause a new line to be started. The carriage return character should not be used; the line feed character will be interpreted as a New Line (since both are represented by 12Q).

The other item to note about the DB statement is the expression " 'E'+200Q". The .PRINT command prints the characters whose address is in the H and L registers until it prints a character with the parity (200Q) bit set. This character is the last one printed. Thus, in the example the expression " 'E'+200Q" was used to set the high-order bit on the last 'E' in the message so HDOS would stop typing at that point.

.SCOUT

Use the .SCOUT to type a single character on the console device. The character in the A register is printed on the console terminal. For example,

```
        MVI     A,'X'
        SCALL   .SCOUT          PRINT THE CHARACTER 'X'
```

The high-order bit (the parity bit) is ignored by .SCOUT.

## Reading From the Console

HDOS provides the .SCIN command for reading characters from the console terminal, and the one command .CONSL to control character echoing, backspace, and erase-line handling.

.SCIN

The .SCIN command is used to read a single character from the console device. If the 8080 "carry" flag is set after the SCALL instruction, it means that no character has been typed yet. If the carry flag is clear, then a character has been read and is in the A register. It does not matter if the carry flag is set or clear when you execute the SCALL .SCIN. For example,

```
READ    SCALL   .SCIN           READ A CHARACTER, IF ANY
        JC      READ            NO CHARACTER ENTERED, YET
        STA     CHAR            STORE CHARACTER READ IN MEMORY
```

.CONSL

The .CONSL command is used to set the mode of console input. There are two modes of input: line mode, and character mode.

When you are inputting in line mode, HDOS saves up the typed characters until you type a RETURN. This is done so HDOS can handle RUBOUT (character delete) and CTRL-U (line delete) functions. If HDOS were to give you the characters one by one as they were typed, it wouldn't be able to 'take them away again' if CTRL-U were typed. By saving them all up until you hit the RETURN, HDOS can handle and DELETEs and CTRL-Us that are typed. For example, if you were to type the four keys Y, E, S, and RETURN while your program was executing the example shown above, it would not receive any characters until you pressed the RETURN. The next four .SCIN commands would each return with one of the characters. The RETURN key gives the 012Q (New Line) character code. Thus, the four values read when you type YES (RETURN) are 131Q (Y), 105Q (E), 123Q (S) and 012Q (RETURN).

Line mode is very useful when you wish to input a line from the console, since HDOS provides the DELETE and CTRL-U functions for you automatically. For programs that need to read each character immediately after it is typed, there is 'character mode'. Inputting in character mode causes the typed character to be passed to your program immediately. If the user types RUBOUT or DELETE, the RUBOUT code (177Q) is passed to your program. If the user types CTRL-U, the CTRL-U code (025Q) is passed to your program. Character mode is more flexible than line mode, but it requires your program to handle the RUBOUT and CTRL-U keys.

The .CONSL command also allows you to turn character echoing on and off. If echoing is turned on, then each time the user strikes a character it is typed on the console automatically by HDOS. If echoing is turned off, the character is not typed on the console. If you wish the character to be visible, your program must type it itself, via the .SCOUT command.

To use the .CONSL command, code the following lines:

```
XRA     A
MVI     B,xxx           'xxx' = value discussed below
MVI     C,201Q
SCALL   .CONSL
```

where 'xxx' is

```
000Q FOR LINE MODE WITH ECHO
001Q FOR CHARACTER MODE WITH ECHO
200Q FOR LINE MODE WITHOUT ECHO
201Q FOR CHARACTER MODE WITHOUT ECHO
```

The default mode of HDOS console input is "line mode, with echo". You only need to use the SCALL .CONSL command if you wish some other mode of operation. You can change modes of operation as often as you like.

> NOTE: The system must be configured to accept tabs in order to run a demonstration program.
>
> See SET command in the HDOS Manual.

```
                TITLE   'DEMO.ASM - HEATH HDOS ASSEMBLY LANGUAGE DEMO'
                XTEXT   HDOS
                ORG     USERFWA

                STL     'DOCUMENTATION'
***             DEMO.ASM - HEATH HDOS ASSEMBLY LANGUAGE.
*
*               DEMO IS A SHORT AND SIMPLE PROGRAM USED TO DEMONSTRATE THE HDOS
*               ASSEMBLER, AND THE HDOS OPERATING SYSTEM.
*
*               THIS PROGRAM SIMPLY PRINTS TWO CODED LINES ON THE
*               SYSTEM CONSOLE TERMINAL.

ENTRY           LXI     H,MESA          (HL) = ADDRESS OF 1ST MESSAGE
                SCALL   .PRINT          PRINT FIRST MESSAGE
                LXI     H,MESB          (HL) = ADDRESS OF 2ND MESSAGE
                SCALL   .PRINT          PRINT 2ND MESSAGE

*               SEND A BELL TO THE TERMINAL

                MVI     A,07Q           (A) = ASCII BELL
                SCALL   .SCOUT          RING TERMINAL'S BELL

*               RETURN CONTROL TO HDOS OPERATING SYSTEM

                XRA     A
                SCALL   .EXIT           EXIT TO  OPERATING SYSTEM

*               MESSAGES FOR .PRINT SCALL'S

MESA            DB      12Q,'HI THERE, SPORTS FANS','!'+200Q
                LON     G               LIST THE BYTES OF THE NEXT MESSAGE
MESB            DB      12Q,'YOUR SYSTEM WORKS FINE','!'+200Q

                END     ENTRY           START EXECUTING AT 'ENTRY' LABEL
```

**Figure 2**

DEMO.ASM

```
            TITLE   'DEMO2.ASM - CONSOLE READ DEMO, LINE MODE'
            XTEXT   HDOS
            ORG     USERFWA

            STL     'WRITTEN 01/23/78'

***         DEMO2.ASM - CONSOLE INPUT DEMO, IN LINE MODE.
*
*           THIS IS A SIMPLE DEMONSTRATION PROGRAM THAT INPUTS LINES FROM
*           THE CONSOLE, AND TYPES THEM BACK AGAIN.
*
*           IF THE LAST LINE YOU ENTERED CONTAINED A PERIOD ('.') THEN
*           DEMO2 EXITS TO HDOS AFTER TYPING THE LINE.

***         TO RUN THIS PROGRAM, TYPE THE FOLLOWING:
*           (DO NOT TYPE COMMENTS IN PARENTHESIS)
*
*           >RUN ASM
*           *DEMO2,TT:=DEMO2                    (WRITES LISTING TO CONSOLE)
*           >RUN DEMO2
*           HI, I'M DEMO2!                      (DEMO2 TYPES THIS)
*           ABCD                                (YOU TYPE THIS)
*           ABCD                                (DEMO2 TYPES THIS)
*           IS ANYONE THERE?                    (YOU TYPE THIS)
*           IS ANYONE THERE?                    (DEMO2 TYPES THIS)
*           BYE BYE.                            (YOU TYPE THIS)
*           BYE BYE.                            (DEMO2 TYPES THIS)
*           >                                   (DEMO2 EXITS TO THE OPERATING SYSTEM)
            STL     'MAIN PROGRAM'
            EJECT                       START A NEW PAGE
ENTRY       LXI     H,DEMOA             EXECUTION STARTS HERE
            SCALL   .PRINT              PRINT 'HI!' MESSAGE

*           LOOP ECHOING LINES

ECHO        SCALL   .SCIN
            JC      ECHO                NO CHARACTER YET
            CPI     '.'
            JNE     ECHO1               NOT PERIOD CHARACTER
            STA     ENDFLAG             MAKE ENDFLAG NON-ZERO (A '.', IN FACT)
ECHO1       SCALL   .SCOUT              TYPE CHARACTER BACK
            LDA     ENDFLAG
            ANA     A
            JZ      ECHO                STILL MORE TO GO

*           HAVE SEEN '.'. WILL RETURN TO HDOS

            XRA     A
            SCALL   .EXIT               RETURN TO HDOS

DEMOA       DB      12Q,'HI, I'M DEMO2!',212Q
ENDFLAG     DB      0                   <>0 IF TO EXIT

            END     ENTRY
```

**Figure 4**

DEMO2.ASM

Note that although this program appears to be written to echo each character after it is typed, actually it echoes each line after the RETURN has been typed. This is because the program reads characters in line mode. HDOS holds the characters until you press the RETURN key, and then supplies them to the DEMO2 program. Thus, each line typed to this program appears twice: once when HDOS echoed it as it was being typed, and once when DEMO2.ASM types it.

```
            TITLE   'DEMO3.ASM - CONSOLE READ DEMO, CHARACTER MODE'
            XTEXT   HDOS
            ORG     USERFWA

            STL     'WRITTEN 01/23/78'

***         DEMO3.ASM - CONSOLE INPUT DEMO, IN CHARACTER MODE.
*
*           THIS IS A SIMPLE DEMONSTRATION PROGRAM THAT INPUTS CHARACTERS FROM
*           THE CONSOLE, AND TYPES THEM BACK AGAIN.
*
*           IF THE LAST CHARACTER YOU ENTERED CONTAINED A PERIOD ('.') THEN
*           DEMO3 EXITS TO HDOS AFTER TYPING THE CHARACTER.

***         TO RUN THIS PROGRAM, TYPE THE FOLLOWING:
*           (DO NOT TYPE COMMENTS IN PARENTHESIS)
*
*           >RUN ASM
*           *DEMO3,TT:=DEMO3                    (WRITES LISTING TO CONSOLE)
*           >RUN DEMO3
*            HI, I'M DEMO3!                     (DEMO3 TYPES THIS)
*           AABBCCDD                            (YOU TYPE ABCD, DEMO3 DUPLICATES IT)
*           XXYY..                              (YOU TYPE 'XY.', DEMO3 ECHOS IT)
*           >                                   (DEMO3 EXITS TO THE OPERATING SYSTEM)
            STL     'MAIN PROGRAM'
            EJECT                       START A NEW PAGE
ENTRY       LXI     H,DEMOA             EXECUTION STARTS HERE
            SCALL   .PRINT              PRINT 'HI!' MESSAGE

*           SETUP CHARACTER MODE. SINCE HDOS WILL ECHO
*           THE CHARACTERS, AND THEN DEMO3 WILL TYPE THEM, CHARACTERS WILL
*           BE DOUBLED ON THE SCREEN AS THEY ARE TYPED.

            XRA     A
            MVI     B,001Q              CHARACTER MODE WITH ECHO
            MVI     C,201Q
            SCALL   .CONSL

*           LOOP ECHOING LINES

ECHO        SCALL   .SCIN
            JC      ECHO                NO CHARACTER YET
            CPI     '.'
            JNE     ECHO1               NOT PERIOD CHARACTER
            STA     ENDFLAG             MAKE ENDFLAG NON-ZERO (A '.', IN FACT)
ECHO1       SCALL   .SCOUT              TYPE CHARACTER BACK
            LDA     ENDFLAG
            ANA     A
            JZ      ECHO                STILL MORE TO GO

*           HAVE SEEN '.'. WILL RETURN TO HDOS

            XRA     A
            SCALL   .EXIT               RETURN TO HDOS

DEMOA       DB      12Q,'HI, I''M DEMO3!';212Q
ENDFLAG     DB      0                   <>0 IF TO EXIT

            END     ENTRY
```

**Figure 5**
DEMO3.ASM

Note that this program is identical to DEMO2.ASM, except that this program inputs in character mode rather than line mode. This causes a big difference in the response the program makes when you type input to it. DEMO3.ASM echoes each character immediately after it is typed. This causes each character to be printed twice on the screen, once when HDOS echoed it and once when DEMO3.ASM typed it. As an exercise, modify this program to disable the automatic echoing (done by HDOS).

# INDEX