SECTION 4

HEATH ASSEMBLY LANGUAGE

HASL-8

TABLE OF CONTENTS

WRITING H8 ASSEMBLY LANGUAGE PROGRAMS

The Heath Assembly Language program (HASL-8) 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 (TED-8).

Heath Assembly Language assembles the source program into a listing and an object program in absolute binary format executable by the H8 Computer. The listing and the object program are produced after two passes through the assembler (the assembler must read the source tape twice before it is able to produce the absolute binary output). If there are sufficient memory locations in the host computer, the binary program may be stored directly in memory. If there is not room, as is often the case when you are assembling large programs, the object program is written onto a mass storage device. During the second pass, HASL-8 produces a complete octal/symbolic listing of the assembled program. This listing is especially useful for documentation and debugging purposes.

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 HASL-8. If you are not familiar with assembly language programming, we recommend that you study material such as the Heathkit Continuing Education "8080 Programming" course.

HASL-8 is designed to produce programs which run in an H8 system; therefore, it assembles 8080 symbolic assembly code. It requires about 8K of memory (depending upon the number of symbols defined during assembly) and one mass storage unit. Separately controllable mass storage readers and recorders may be necessary for large assemblies. This can take the form of one paper tape/reader punch or two independent cassette recorders. When used with 8K of memory, HASL-8 provides for approximately 250 user-defined symbols.

This Software Reference Manual presumes that you have read the H8 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 HASL-8 are listed below.

1. The letters A through Z (lower case letters are acceptable for quoted strings and comments, provided that HASL-8 is configured properly in accordance with the software configuration guide).

2. The numerals 0 through 9.

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

4. The symbols

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

   LINE FEED AND CARRIAGE RETURN

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 by separating characters. The four fields are:

LABEL OPCODE OPERAND COMMENT

The label and 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. 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 040 100 and the statement

```
.START    MOV A,B
```

is the next statement, the assembler assigns the value 040 000 to the label .START. Subsequent references to .START refer to location 040 100.

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. 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 or by 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. 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 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 space 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 blank 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. 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 character. Format control is primarily used to produce a program which is easily read. Format control has no affect on the assembly process of the source program, and because HASL-8 uses compression techniques, the use of multiple blanks does not take up extra memory space. The following two statements are identical with the exception that the first one does not use any tabs and the second one uses tabs.

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

The TABs were converted to the appropriate number of blanks by TED-8. Therefore, HASL-8 sees no TAB characters.

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

HASL-8 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
```

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 112O)	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 HASL-8 characters.

The # Symbol

If the pound sign (#) is the first charcter 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 HASL-8 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 10n 114A)
" "	The character quotation mark (Value 042Q)

THE ORIGIN SYMBOL, ORG

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.

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

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.

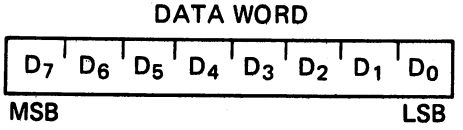
Terms, Symbols, & Nomenclature

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.

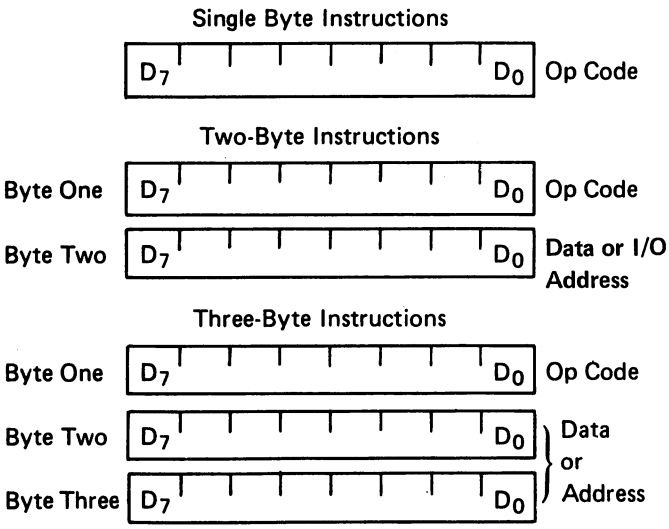
The 8080 can directly address up to 65,536 bytes of memory, which may consist of both read-only memory (ROM) elements and random-access memory (RAM) elements (read/write memory).

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: HASL-8 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., (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 HASL-8 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.

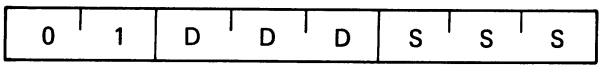
6. The last two lines contain incidental information about the execution of the instruction. The number of machine cycles and states required to execute the instruction are listed first. If the instruction has two possible execution times, as in a conditional jump, both times will be listed, separated 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)

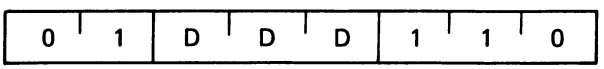
(r1)← (r2)
The content of register r2 is moved to register r1.



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

MOV r, M (Move from memory)

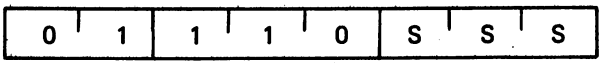
(r) ← ((H) (L))
The content of the memory location whose address is in registers H and L is moved to register r.



Cycles: 2
States: 7
Addressing: reg. indirect
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.

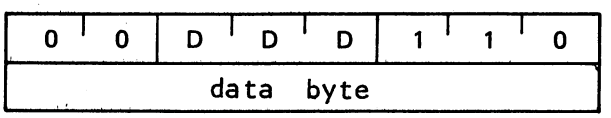


Cycles: 2
States: 7

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.

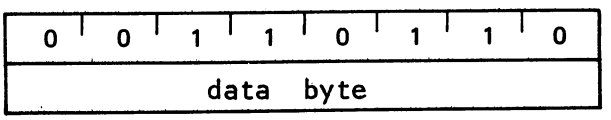


Cycles: 2
States: 7

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.

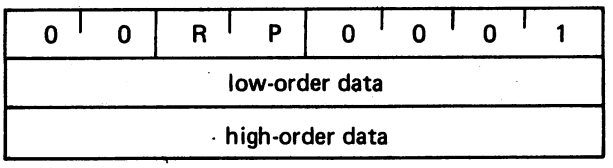


Cycles: 3
States: 10

Addressing: immed./reg. indirect
Flags: none

LXI rp, data 16 (Load register pair immediate)

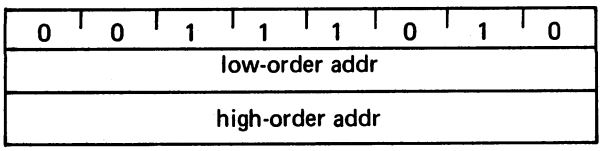
(rh) ← (byte 3),
(rl) ← (byte 2)
Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp. Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp.



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

LDA addr (Load Accumulator direct)

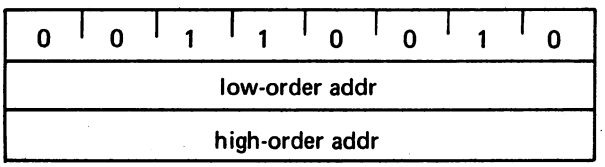
(A) ← (byte 3) (byte 2)
The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register A.



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

STA addr (Store accumulator direct)

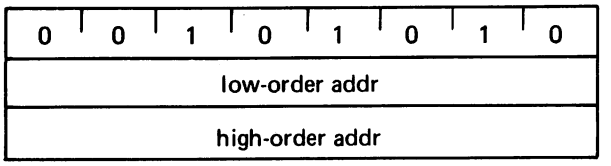
((byte 3) (byte 2)) ← (A)
The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.



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

LHLD addr (Load H and L direct)

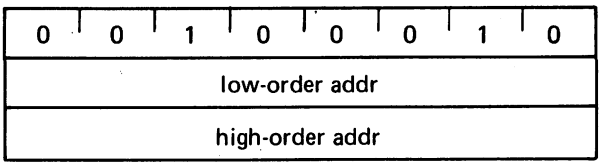
(L) ← ((byte 3) (byte 2))
(H) ← ((byte 3) (byte 2) + 1)
The content of the memory location whose address is specified in byte 2 and byte 3 of the instruction is moved to register L. The content of the memory location at the succeeding address is moved to register H.



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

SHLD addr (Store H and L direct)

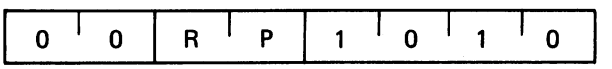
((byte 3) (byte 2)) ← (L)
((byte 3) (byte 2) + 1) ← (H)
The content of register L is moved to the memory location whose address is specified in byte 2 and byte 3. The content of register H is moved to the succeeding memory location.



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

LDAX rp (Load accumulator indirect)

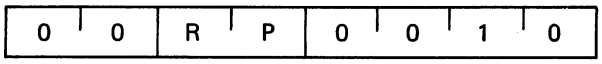
(A) ← ((rp))
The content of the memory location whose address is in the register pair rp is moved to register A. NOTE: Only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.



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

STAX rp (Store accumulator indirect)

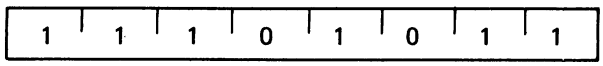
((rp)) ← (A)
The content of register A is moved to the memory location whose address is in the register pair rp. NOTE: Only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.



Cycles: 2 Addressing: reg. indirect
States: 7 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 Addressing: register
States: 4 Flags: none

Arithmetic Group

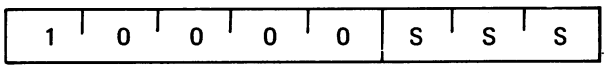
This group of instructions performs arithmetic operations on data in registers and memory.

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)

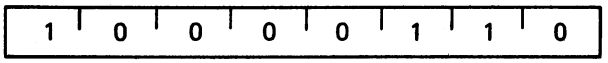
(A) ← (A) + (r)
The content of register r is added to the content of the accumulator. The result is placed in the accumulator.



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

ADD M (Add memory)

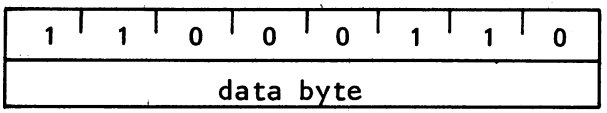
(A) ← (A) + ((H) (L))
The content of the memory location whose address is contained in the H and L registers is added to the content of the accumulator. The result is placed in the accumulator.



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

ADI DATA (add immediate)

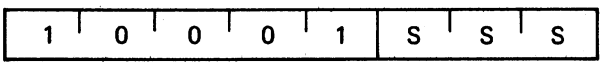
(A) ← (A) + (byte 2)
The content of the second byte of the instruction is added to the content of the accumulator. The result is placed in the accumulator.



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

ADC r (Add Register with carry)

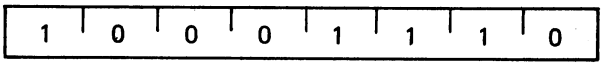
(A) ← (A) + (r) + (CY)
The content of register r and the content of the carry bit are added to the content of the accumulator. The result is placed in the accumulator.



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

ADC M (Add memory with carry)

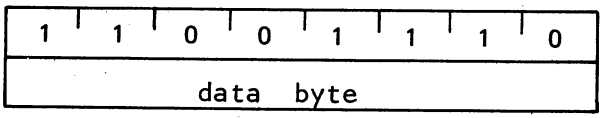
(A) ← (A) + ((H) (L)) + (CY)
The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are added to the accumulator. The result is placed in the accumulator.



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

ACI data (Add immediate with carry)

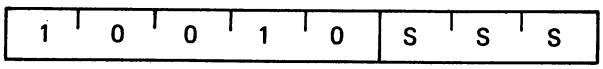
(A) ← (A) + (byte 2) + (CY)
The content of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.



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

SUB r (Subtract Register)

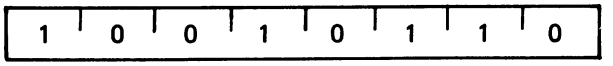
(A) ← (A) — (r)
The content of register r is subtracted from the content of the accumulator. The result is placed in the accumulator.



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

SUB M (Subtract memory)

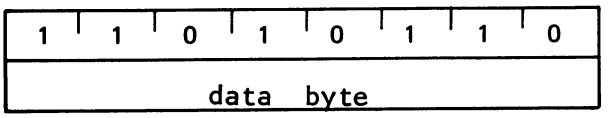
(A) ← (A) — ((H) (L))
The content of the memory location whose address is contained in the H and L registers is subtracted from the content of the accumulator. The result is placed in the accumulator.



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

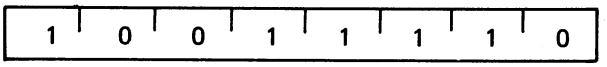
SBB r (Subtract Register with borrow)

(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)

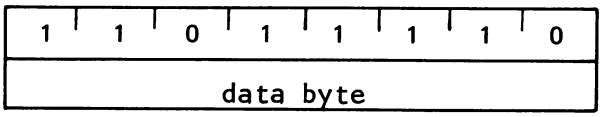
(A) ← (A) — ((H) (L)) — (CY)
The content of the memory location whose address is contained in the H and I registers and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.



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

SBI data (Subtract immediate with borrow)

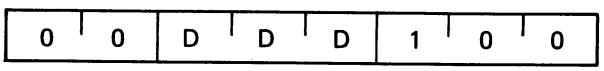
(A) ← (A) — (byte 2) — (CY)
The contents of the second byte of the instruction and the contents of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.



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

INR r (Increment Register)

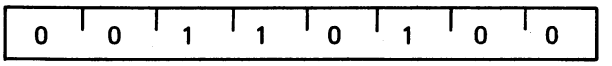
(r) ← (r) + 1
The content of register r is incremented by one. NOTE: All condition flags **except CY** are affected.



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

INR M (Increment memory)

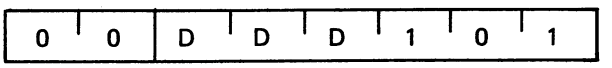
((H) (L)) ← ((H) (L)) + 1
The content of the memory location whose address is contained in the H and L registers is incremented by one. NOTE: All condition flags **except CY** are affected.



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

DCR r (Decrement Register)

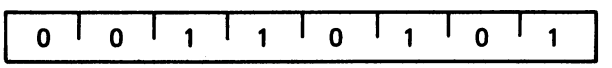
(r) ← (r) — 1
The content of register r is decremented by one. NOTE: All condition flags **except CY** are affected.



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

DCR M (Decrement memory)

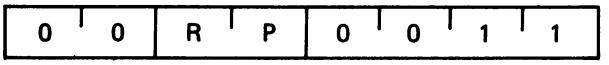
((H) (L)) ← ((H) (L)) — 1
The content of the memory location whose address is contained in the H and L registers is decremented by one. NOTE: All condition flags **except CY** are affected.



Cycles: 3 Addressing: reg. indirect
States: 10 Flags:

INX rp (Increment register pair) 0(0,2,4,6)3

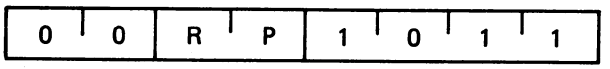
(rh) (rl) ← (rh) (rl) + 1
The content of the register pair rp is incremented by one. NOTE: **No condition flags are affected.**



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

DCX rp (Decrement register pair)

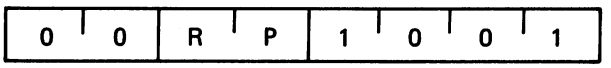
(rh) (rl) ← (rh) (rl) — 1
The content of register pair rp is decremented by one. NOTE: **No condition flags are affected.**



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.



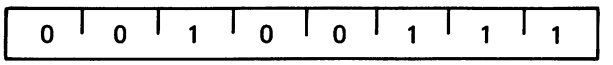
Cycles: 3 Addressing: register
States: 10 Flags: CY

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

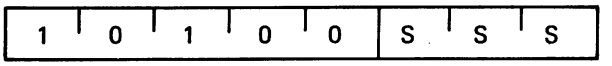
Logical Group:

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)

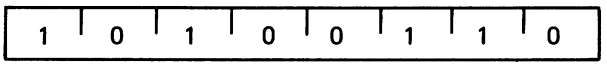
(A) ← (A) ∧ (r)
The content of register r is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**



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

ANA M (AND memory)

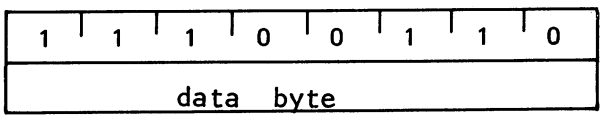
(A) ← (A) ∧ ((H) (L))
The contents of the memory location whose address is contained in the H and L registers is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**



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

ANI data (AND immediate)

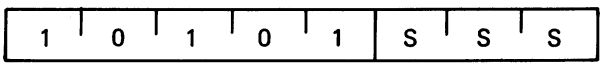
(A) ← (A) ∧ (byte 2)
The content of the second byte of the instruction is logically anded with the contents of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



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

XRA r (Exclusive OR Register)

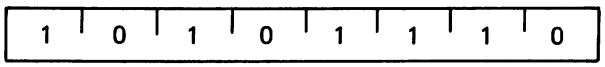
(A) ← (A) ⊻ (r)
The content of register r is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



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

XRA M (Exclusive OR Memory)

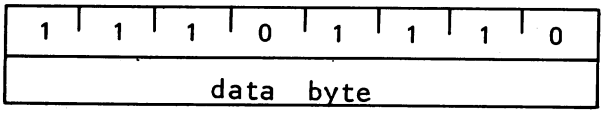
(A) ← (A) ⊻ ((H) (L))
The content of the memory location whose address is contained in the H and L registers is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



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

XRI data (Exclusive OR immediate)

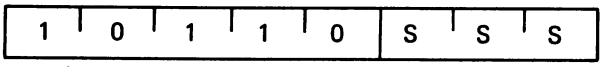
(A) ← (A) ⊻ (byte 2)
The content of the second byte of the instruction is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



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

ORA r (OR Register)

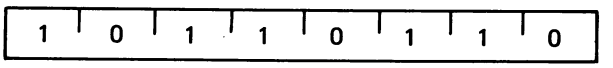
(A) ← (A) ∨ (r)
The content of register r is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



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

ORA M (OR memory)

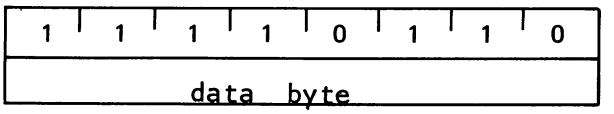
(A) ← (A) ∨ ((H) (L))
The content of the memory location whose address is contained in the H and L registers is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



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

ORI data (OR Immediate)

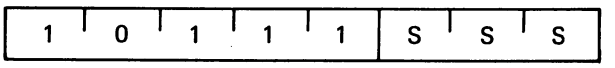
(A) ← (A) ∨ (byte 2)
The content of the second byte of the instruction is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



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

CMP r (Compare Register)

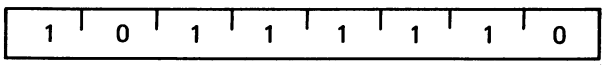
(A) — (r)
The content of register r is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A) = (r). The CY flag is set to 1 if (A) < (r).**



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

CMP M (Compare memory)

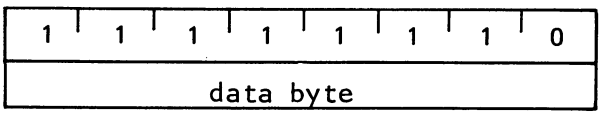
(A) — ((H) (L))
The content of the memory location whose address is contained in the H and L registers is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. The Z flag is set to 1 if (A) = ((H) (L)). The CY flag is set to 1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



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

CPI data (Compare immediate)

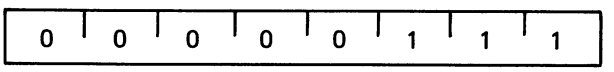
(A) — (byte 2)
The content of the second byte of the instruction is subtracted from the accumulator. The condition flags are set by the result of the subtraction. The Z flag is set to 1 if (A) = (byte 2). The CY flag is set to 1 if (A) < (byte 2).



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

RLC (Rotate left)

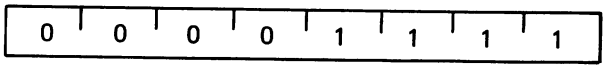
$(A_{n+1}) \leftarrow (A_n)$; $(A_0) \leftarrow (A_7)$
$(CY) \leftarrow (A_7)$
The content of the accumulator is rotated left one position. The low order bit and the CY flag are both set to the value shifted out of the high order bit position. **Only the CY flag is affected.**



Cycles: 1
States: 4
Flags: CY

RRC (Rotate right)

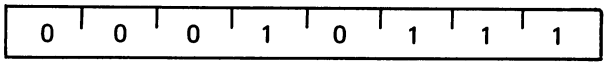
$(A_n) \leftarrow (A_{n-1})$; $(A_7) \leftarrow (A_0)$
$(CY) \leftarrow (A_0)$
The content of the accumulator is rotated right one position. The high order bit and the CY flag are both set to the value shifted out of the low order bit position. **Only the CY flag is affected.**



Cycles: 1
States: 4
Flags: CY

RAL (Rotate left through carry)

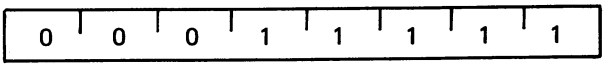
$(A_{n+1}) \leftarrow (A_n)$; $(CY) \leftarrow (A_7)$
$(A_0) \leftarrow (CY)$
The content of the accumulator is rotated left one position through the CY flag. The low order bit is set equal to the CY flag and the CY flag is set to the value shifted out of the high order bit. **Only the CY flag is affected.**



Cycles: 1
States: 4
Flags: CY

RAR (Rotate right through carry)

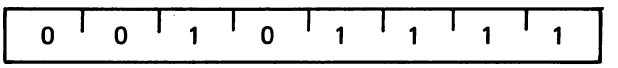
$(A_n) \leftarrow (A_{n+1})$; $(CY) \leftarrow (A_0)$
$(A_7) \leftarrow (CY)$
The content of the accumulator is rotated right one position through the CY flag. The high order bit is set to the CY flag and the CY flag is set to the value shifted out of the low order bit. **Only the CY flag is affected.**



Cycles: 1
States: 4
Flags: CY

CMA (Complement accumulator)

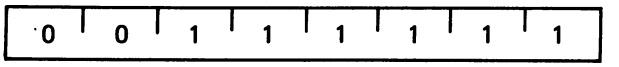
(A) ← (A)
The contents of the accumulator are complemented (zero bits become 1, one bits become (0). **No flags are affected.**



Cycles: 1
States: 4
Flags: none

CMC (Complement carry)

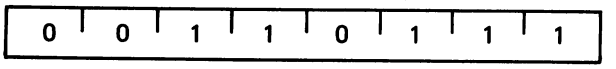
(CY) ← ($\overline{CY}$)
The CY flag is complemented. **No other flags are affected.**



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.

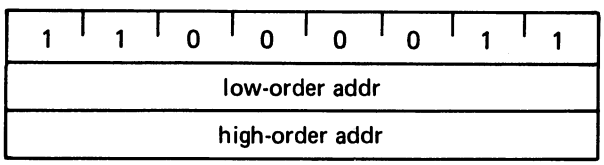
The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register PC (the

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)

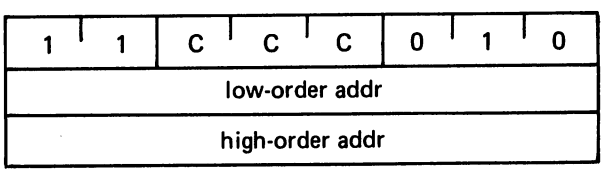
(PC) ← (byte 3) (byte 2)
Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.



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.

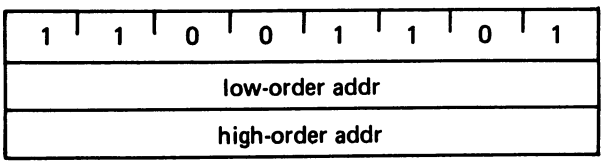


Cycles: 3 Addressing: immediate
States: 10 Flags: none

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.



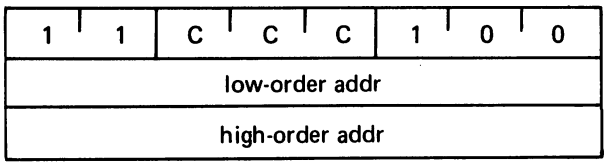
Cycles: 5
States: 17

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.



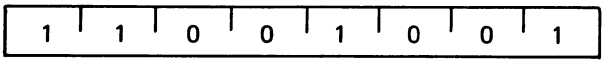
Cycles: 3/5
States: 11/17

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.

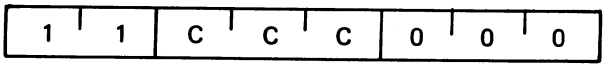


Cycles: 3
States: 10
Addressing: reg. indirect
Flags: none

RNE RNC COP CP
RE RC CPE CM (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.



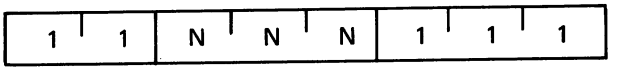
Cycles: 1/3
States: 5/11
Addressing: reg. indirect
Flags: none

RST n (Restart)

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

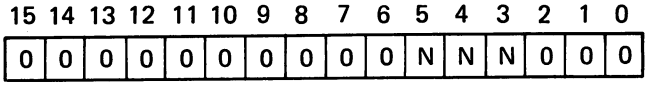
The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP.

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

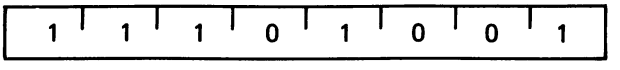


Program Counter After Restart

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

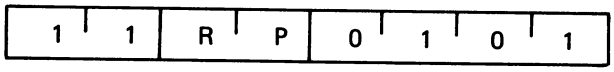
Stack, I/O, and Machine Control Group

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.**



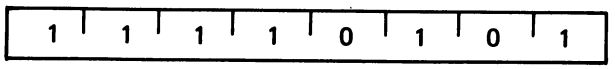
Cycles: 3
States: 11

Addressing: reg. indirect
Flags: none

PUSH PSW (Push processor status word)

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

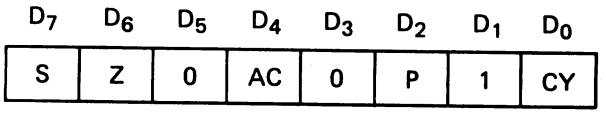
The content of register A is moved to the memory location whose address is one less than register SP. The contents of the condition flags are assembled into a processor status word and the word is moved to the memory location whose address is two less than the content of register SP. The content of register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



Cycles: 3
States: 11

Addressing: reg. indirect
Flags: none

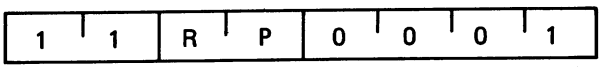
FLAG WORD



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.**



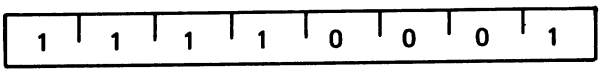
Cycles: 3
States: 10

Addressing: reg. indirect
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.



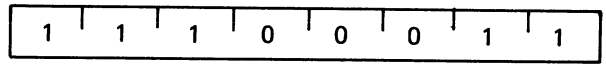
Cycles: 3
States: 10

Addressing: reg. indirect
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.



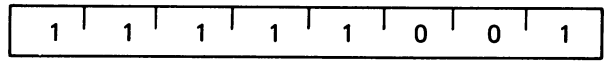
Cycles: 5
States: 18

Addressing: reg. indirect
Flags: none

SPHL (Move HL to SP)

(SP) ← (H) (L)

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



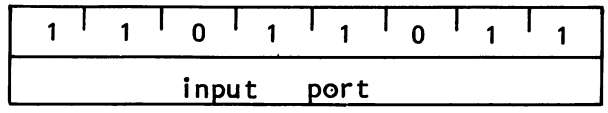
Cycles: 1
States: 5

Addressing: register
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.



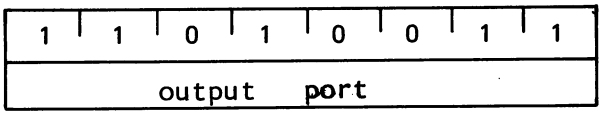
Cycles: 3
States: 10

Addressing: direct
Flags: none

OUT port (Output)

(data) ← (A)

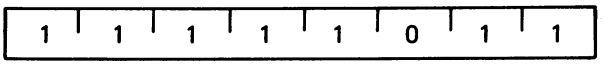
The content of register A is placed on the eight bit bidirectional data bus for transmission to the specified port.



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

EI (Enable interrupt)

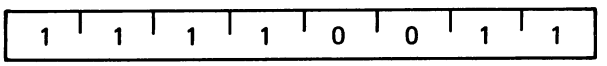
The interrupt system is enabled **following the execution of the next instruction.**



Cycles: 1
States: 4
Flags: none

DI (Disable interrupt)

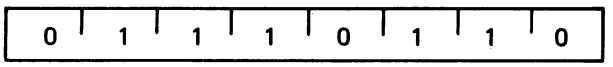
The interrupt system is disabled **immediately following the execution of the DI instruction.**



Cycles: 1
States: 4
Flags: none

HLT (Halt)

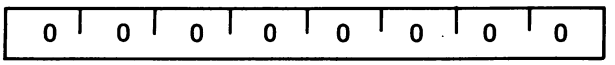
The processor is stopped. The registers and flags are unaffected.



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 20 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 HASL-8 and are not translated as instructions for the H8. Some pseudo ops conditionally 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. HASL-8 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

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, the H8 does not receive a starting value for the program counter where the binary tape is loaded.

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.

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	40 100A	The program is started at location 040 100 (offset octal) and the label BEGIN is assigned the offset octal value 040 100.
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

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.

Listing Control

HASL-8 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 HASL-8 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 speudo was executed.

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: Define byte 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.

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

HASL-8 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. So any change which causes 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.

USING THE ASSEMBLER

Before the Heath Assembly Language is used, the source program must be prepared using a Text Editor such as TED-8. Once the source program is prepared and stored on tape, Heath Assembly Language can be loaded in the H8. The loading procedure is outlined in "Appendix A" (Page 4-57). Once a configured version of HASL-8 is loaded and started, a series of questions must be answered. First the assembler asks

```
PAGE SIZE?
```

The normal page (8-1/2 × 11) has 66 lines. You can specify longer or shorter page sizes by counting the total number of lines required to fill the page top to bottom. The assembler then asks

```
INTER-PAGE GAP SIZE?
```

The normal page allows the last six lines for the inter-page gap. However, you may specify any desired gap. The PAGE SIZE and INTER-PAGE GAP inputs are used when the title, subtitle, space and eject pseudo-ops are executed. The assembler then asks

```
LISTING PORT:
```

The normal reply to listing port is a carriage-return. A carriage-return with no port number specified indicates the listing port is to be the console terminal. If you specify, a port number, HASL-8 returns control to PAM-8 so you may configure the port as desired. Once the port is configured, HASL-8 is restarted at location 040 100. Once listing port is specified, the assembler asks

```
BINARY (Y/N)?
```

If you do not want a binary (N), the output tape transport is not used and no binary image of the assembled program is placed in memory. Often, no binary is specified until you are sure the program will assemble. If a yes (Y) is given in reply to the question, the assembler then asks

```
BINARY TAPE (Y/N)?
```

A no (N) reply to this question directs HASL-8 to place the binary generated from the assembly into memory at the proper location. If NO BINARY TAPE is specified, you should set the HIGH MEMORY limit below the point used by the object program. NOTE: To do this may require reconfiguration of the assembler. See "Appendix A," (Page 4-57).

A yes (Y) reply to this question directs HASL-8 to place the binary generated from the assembly of the source code onto tape at the dump port. This tape is in the memory image format and contains the starting and ending addresses, and the entry point address of the desired program.

Once the assembler determines whether a binary is to be generated or not, and if it is to be placed into memory or dumped onto tape, it then asks

```
INPUT
```

The response to this is the character string used to identify the source file when it was created by the Text Editor. Do not include any string delimiters to specify the file name when outputting from Text Editor. For example:

```
NEWOUT "TEST"
FLUSH
SURE?
```

dumps a file named TEST using TED-8. It is loaded by the assembler by

```
INPUT?TEST
FOUND TEST
```

The file name does not have to be complete and can be a null, which allows HASL-8 to load the next file on the tape. (Enter a null by typing a range return.)

Once the file is found, HASL-8 begins the assembly process. The entire file is read for the first pass. Once the first pass is complete, HASL-8 issues the instruction

```
REWIND SOURCE TAPE TYPE CR WHEN DONE.
```

The tape drive is not turned off, so the source tape may be easily returned at its starting point. Once the tape is at its starting point, type a carriage return (CR) and start the tape transport. HASL-8 then issues the instruction

```
FOUND TEST
POSITION PAPER. TYPE CR:
```

The paper in any printer on the H8 system should be in place and the dump tape transport should be made ready at this time. Position the paper at the bottom of a form. HASL-8 starts by spacing an interpage gap. It then prints the title and subtitles.

Once you type the carriage return, HASL-8 begins the second pass, generating the listing and creating the binary tape. The listing may require reading several records of the input tape and the output binary dump may come in a number of records. Once the listing and the binary dump are complete, HASL-8 terminates its operation by outputting

```
STATEMENTS = xxxxx
FREE BYTES - xxxxx
NO ERRORS DETECTED.     or

STATEMENTS = xxxxx
FREE BYTES - xxxxx

ERRORS - xxxxx
```

This first version of this terminating statement indicates that you have successfully completed an assembly of your source program, and if a binary output is specified it is generated. The second version of this terminating statement indicates that you have completed assembly of your source program, but there are errors which the assembler is able to detect. These errors exist in any binary output which may have been specified. Up to three errors per statement line will be shown on the listing output. The errors are shown as single letters in the left hand three columns of the listing. A typical output listing format is shown below.

```
HEATH/WINTEK H8 ASSEMBLER
ISSUE # 4.01.00.
COPYRIGHT WINTEK CORP., 01/77

.PAD = 4/0
.CONSOLE LENGTH = 00080/72
.HIGH MEMORY = 24575/
.LOWER CASE (Y/N)?
.
HEATH H8ASM ISSUE #4.01.00.

PAGE SIZE? 60
INTER-PAGE GAP SIZE? 6
LISTING PORT:
BINARY (Y/N)?Y
BINARY TAPE (Y/N)?Y
INPUT?USR PROGRAM FOR BASIC #1.0
FOUND USR PROGRAM FOR BASIC #1.0
REWIND SOURCE TAPE. TYPE CR WHEN DONE.
FOUND USR PROGRAM FOR BASIC #1.0
POSITION PAPER. TYPE CR:
                                                        HASL  #04.01.00
                                                           PAGE  1
Errors   Addresses Object          Labeler  Opcodes  Operands      Comments
                   Code
------   --------- ------          -------  -------  ----------    --------
          117.220                           ORG      120000A-160Q
          063.207                  FPNRM    EQU      063207A
          117.220   003             START    INX      B             INC UP
          117.221   003                      INX      B              TO
          117.222   003                      INX      B                EXPONENT
          117.223   012                      LDAX     B             (A) = ACCX EXP
          117.224   247                      ANA      A             SET CONDX CODE
          117.225   310                      RZ
          117.226   075                      DCR      A             /2
          117.227   312 233 077              JZ       USR1          IF UNDER FLOW
          117.232   075                      DCR      A             /2 AGAIN (/4)
          117.233   002             USR1     STAX     B             RET TO ACCX
          117.234   315 207 063              CALL     FPNRM         NORMALIZE
          117.237   311                      RET                    IN CASE 0

          117.240                           END      START
STATEMENTS = 00016
FREE BYTES - 10331
NO ERRORS DETECTED.
```

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 nonexistant.
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 instruciton.
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 HASL-8 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 BUG-8 or PAM-8. However, this **does not** result in a correct listing.

During an Input, one of two error messages may be generated. They are:

```
SEQ ERR    and
CHKSUM ERR.
```

A sequence error (SEQ ERR) is generated if the file records are not in the proper sequence. For example, if two consecutive label records are read, an error is generated, as a TED-8 Source file consists of a label record followed by text records. The form of the sequence error is

```
SEQ ERR
```

Typing a CNTRL-C after the SEQ ERR message generates a tape error message

```
TRY AGAIN?
```

Reply Y to TRY AGAIN? if you wish to try once more to read the tape. Rewind the tape until you are sure it is before the bad/missing record. HASL-8 will discard all records until the bad/missing one is located. Watch the record numbers on the H8 front panel LED's to make sure you don't miss the record again.

Reply N to TRY AGAIN? if you wish to restart the assembly completely.

A checksum error (CHKSUM ERR) is generated if the actual computed CRC for the record in question does not match the CRC recorded at the start of the record. The form of the checksum error message is

```
CHKSUM ERR IGNORE?
```

A Y in response to the question ignore aborts the error message and the next consecutive record is read. NOTE: Do not ignore the checksum error unless there is no other way to recover the data. If a checksum error is flagged, the chances are very good that the data in the designated record is faulty.

Control Characters

CONTROL-C

CONTROL-C is a general-purpose cancel key. Typing CNTRL-C causes HASL-8 to start over at the beginning.

RESTARTS

The H8 front panel keyboard can be used to restart the assembler if control has been returned to PAM-8. HASL-8 can be restarted in two places. They are:

NEW PASS #1 040 100 and;
NEW PASS #2 040 103.

OUTPUT SUSPENSION and RESTORATION, CNTRL-S and CNTRL-Q

Typing CONTROL-S during an output suspends the output to the terminal and suspends program execution. This command is particularly useful when you use a video terminal, since you can use the CONTROL-S or suspend feature each time a screen is nearly filled and information at the top is about to be lost due to scrolling.

Typing a CONTROL-Q permits HASL-8 to resume execution and outputting information to the terminal. The CONTROL-Q cancels the CONTROL-S function.

The DISCARD FLAG, CNTRL-O and CNTRL-B

Typing the CONTROL-O toggles the DISCARD FLAG. This stops output on the terminal but does not halt program execution until the program terminates. Typing a CONTROL-P (or retyping CONTROL-O) clears the discard flag. CONTROL-O is often used to discard the remainder of long listings and other similar outputs.

APPENDIX A

Loading Procedures

Loading From the Software Distribution Tape

1. Load the tape in the reader.
2. Ready the tape transport.
3. Press LOAD on the H8 front panel.
4. A single beep indicates a successful load.
5. Press GO on the H8 front panel.
6. The console terminal will respond with:

```
HEATH/WINTEK H8 ASSEMBLER
ISSUE #4.01.00
COPYRIGHT WINTEK CORP., 01/76
```

7. Configure HASL-8 as desired, answering the following questions. Prompt each question by typing its first character on the console terminal keyboard.

```
•AUTO NEW-LINE (Y/N)?
•BKSP = 00008/
•CONSOLE LENGTH = 00080/
•HIGH MEMORY = 16383/
•LOWER CASE (Y/N)?
•PAD = 4/
•RUBOUT = 00127/
•SAVE?
•
```

8. Before executing SAVE, have the tape transport ready at the DUMP port.
9. To use HASL-8 directly from the distribution tape, type the return key at any time rather than a question prompt key. HASL-8 responds

```
HEATH HASL-8 ISSUE #4.01.00.
*
```

Loading From a Configured Tape

1. Load the tape in the tape transport.
2. Ready the tape transport.
3. Press LOAD on the H8 front panel.
4. A medium beep indicates a successful load.
5. Press GO key on the H8 front panel.
6. The console terminal responds with:

```
HEATH HASL-8 ISSUE #4.01.00
BINARY (Y/N)←?Y
```

HASL-8 is ready to use in the configured form. Proceed to answer the question directing the desired assembly procedure.

INDEX

REQUIRED PATCHES FOR

HEATH H8 HASL - 8

Upgrading 04.01.00.
To 04.01.01.

THESE PATCHES MUST BE INSTALLED IN THE
PRODUCT EVERY TIME A LOAD IS MADE FROM
THE CONFIGURATION TAPE

NOTES: None.

USE: None.

- -

041221 041
041366 315 072 067
051114 127
053133 303 101 067
053227 112 067
054010 127
055104 064
055112 061
055150 127
055213 325
055231 315 120 067 000
067072 062 122 063 062 126 063 311 302
067102 043 053 315 042 056 303 136 053
067112 315 144 040 303 042 056 062 050
067122 047 376 325 311
073002 064
073010 061