A 'SMAL' DEFINITION OF THE M-6800
MICROCOMPUTER

by

Edward Strasbourger,
Zary Segal,
Jeffrey Siskind,
Shuni Tooval

Technical Report #97
May 1977
A 'SMAL' DEFINITION OF THE M-6800 MICROCOMPUTER

This report defines the M-6800 microcomputer for a programmer using a high level language SMAL (Standard Microcomputer Assembly Language). The intention is to make learning the computer and programming it easier and less error prone than normal reference manual descriptions. In addition the same language will be used to define other microcomputers (such as INTEL 8080, Z1LOG Z80, and TI 9900), making machine language system software more comparable and more portable between the various microcomputers.

The M-6800 microcomputer consists of 6 registers as shown in Figure 1: 2 8-bit accumulators (register A and register B) for arithmetic and logic operations, 3 16-bit registers (index register X, program counter PC, and stack pointer SP), and a condition code register CC which contains 6 flags: CARRY, OVERFLOW, ZERO, NEGATIVE, INTERRUPT, mask, and HALF CARRY.

Stacks are stored backwards in memory, with the top-of-stack stored in a lower numbered address from the bottom-of-stack. The location filled by the next push is kept in SP. Stacks may be used explicitly by PUSH and POP instructions, but are automatically used by CALL and RETURN instructions and by the interrupts.
M-6800 REGISTERS

- 8 bit register A
- 8 bit register B
- 16 bit index register
- 16 bit program counter
- 16 bit stack pointer
- 8 bit condition codes register

Figure 1  M-6800 Registers
'SMAL' SYNTAX

A lower case italicized word is a nonterminal symbol, i.e. is explained on this page.

Brackets [ ] denote a choice of one of the included options, unless there is only one option in which case the brackets mean to either include or to omit the option.

Triple dots ... means an indefinite repetition of the preceding option.

comment is any characters in a line following a semicolon;

register is one of a subset of A,B,X,PC,SP,CC, often the subset A,B, optionally preceded by the word REGISTER.

flag is any of the condition code bits: CARRY, OVERFLOW, ZERO, NEGATIVE, INTERRUPT, or HALFCARRY.

number may be decimal such as 25; or hexadecimal such as HEX 25 = 37; or octal such as OCTAL 25 = 21, or binary such as BINARY 11001 = 25.

name is a string of continuous alphanumerics which must begin with an alphabetic and may include periods as concatenators such as GET.NEXT_SYMBOL.

label is a name immediately followed by a colon such as NEXT.WORD:
The special label START: designates the location of the first instruction to be executed.

address is a location expressed as a name such as M, or a number such as HEX 25, or a displacement from an index value such as (X-3).

character is a single, ASCII or EBCDIC character such as "T" or "I". Default is ASCII unless overridden by an EBCDIC instruction.

string is a string of characters surrounded by quotes such as "TO BE".

line of SMAL code is [label] [instruction] [comment] which may be blank or may contain just a label or just an instruction or just a comment or any two parts or all three parts.
where \textit{defined-name} is a name defined by a \texttt{DEFINE} statement.

\texttt{name} (and \textit{defined-name}) usually designates the contents of a memory location, unless preceded by \texttt{ADDRESS OF} (in which case it designates the address of a memory location) or unless preceded by \texttt{NUMBER} (or #) in which case it designates a numerical value.

\texttt{number} is interpreted as a numerical value, unless preceded by \texttt{CONTENTS OF} in which case \texttt{number} designates the address of a memory location.

\((X)\) designates the contents of the memory location whose address is in register \(X\).

\((X \pm n)\) designates the contents of the memory location whose address is the sum of \(\pm n\) plus the contents of register \(X\), (i.e. indexed addressing).
M-6800 INSTRUCTIONS

ADD [DECIMAL] source TO register [WITH CARRY]
AND source TO register
[CALL] name
CLEAR flag or register or address
COMPLEMENT flag or register or address
DECIMAL ADJUST A
DECREMENT register or address [BY number]
DISABLE INTERRUPT
DO [name] [FOREVER]
        UNDO [name]
        REDO [name]
        END [name]
ENABLE INTERRUPT
EOR source TO register
FILL number WORDS
GO TO address
IF condition THEN any instruction other than IF
    [ELSE any instruction other than IF]
INCREMENT register or address [BY number]
INPUT [FROM] address TO register
MOVE [FROM] source TO register or address
NEGATE register or address
OR source TO register
OUTPUT [FROM] register TO address
POP INTO register
PUSH register
REGARDLESS IF condition
RETURN [FROM INTERRUPT or SUBROUTINE or name or MONITOR]
ROTATE register or address LEFT or RIGHT [number BITS]
SET flag
SHIFT register or address LEFT or RIGHT [ARITHMETICALLY] [number BITS]
SUBTRACT source FROM register [WITH BORROW]
WAIT [FOR INTERRUPT]
XOR source TO register
'SMALL' PSEUDOINSTRUCTIONS

ASCII MNEMONICS
- CONDENSED LISTING
DATA string or [WORD or DOUBLEWORD [number or name] [, number or name]...]
DEFINE name AS number
EBCDIC
EXPANDED LISTING
LOCATION number
PROCEDURE name
RESERVE number WORDS
-SEQUENCE NUMBERS
SEQUENCELESS
SUBROUTINE name
?END
?SAVE

The instructions EOR and XOR are identical,
as are the instructions MOVE, INPUT, and OUTPUT,
and as are the pseudoinstructions SUBROUTINE and PROCEDURE.

CONDENSED LISTING and SEQUENCE NUMBERS are default, thus need not be
specified.

ASCII characters are default, but not ASCII MNEMONICS.
SAMPLE PROGRAM

Program BUBBLE.SORT to be called with array to be sorted starting in location ARRAY and with address of last element in doubleword location END.OF.ARRAY and END.OF.ARRAY + 1.

<table>
<thead>
<tr>
<th>MNEmonic Program</th>
<th>SMAL Program</th>
<th>Structured SMAL Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUBBLE.SORT</td>
<td>BUBBLE.SORT:</td>
<td>PROCEDURE BUBBLE.SORT</td>
</tr>
<tr>
<td>NEXT.PASS</td>
<td>NEXT.PASS:</td>
<td>DO NEXT.PASS FOREVER</td>
</tr>
<tr>
<td></td>
<td>MOVE ADDRESS OF ARRAY TO X</td>
<td>MOVE ADDRESS OF ARRAY TO X</td>
</tr>
<tr>
<td></td>
<td>MOVE (X) TO A</td>
<td>MOVE (X) TO A</td>
</tr>
<tr>
<td>CLR SWITCHED</td>
<td>CLEAR SWITCHED</td>
<td>CLEAR SWITCHED</td>
</tr>
<tr>
<td>NEXT.PAIR</td>
<td>NEXT.PAIR:</td>
<td>DECREMENT X</td>
</tr>
<tr>
<td></td>
<td>MOVE (X) TO B</td>
<td>MOVE (X) TO B</td>
</tr>
<tr>
<td></td>
<td>IF A IS NOT &gt; B</td>
<td>IF A &gt; B THEN DO SWITCH</td>
</tr>
<tr>
<td></td>
<td>INCREMENT X</td>
<td>INCREMENT SWITCHED</td>
</tr>
<tr>
<td></td>
<td>INC SWITCHED</td>
<td>DECREMENT X</td>
</tr>
<tr>
<td></td>
<td>THEN GO TO AFTER.SWITCH</td>
<td>MOVE B TO (X)</td>
</tr>
<tr>
<td></td>
<td>INCREMENT X</td>
<td>INCREMENT X</td>
</tr>
<tr>
<td></td>
<td>MOVE B TO (X)</td>
<td>MOVE B TO (X)</td>
</tr>
<tr>
<td></td>
<td>INCREMENT X</td>
<td>MOVE A TO (X)</td>
</tr>
<tr>
<td></td>
<td>MOVE A TO B</td>
<td>MOVE A TO B</td>
</tr>
<tr>
<td></td>
<td>INC SWITCHED</td>
<td>INCREMENT SWITCHED</td>
</tr>
<tr>
<td></td>
<td>STA B X</td>
<td>STA B X</td>
</tr>
<tr>
<td></td>
<td>LDA B X</td>
<td>LDA B X</td>
</tr>
<tr>
<td></td>
<td>INX</td>
<td>INX</td>
</tr>
<tr>
<td></td>
<td>STA A X</td>
<td>STA A X</td>
</tr>
<tr>
<td></td>
<td>TAB</td>
<td>TAB</td>
</tr>
<tr>
<td></td>
<td>TBA</td>
<td>TBA</td>
</tr>
<tr>
<td></td>
<td>AFT3ER.SWITCH:</td>
<td>MOVE B TO A</td>
</tr>
<tr>
<td></td>
<td>CPX END.OF.ARRAY</td>
<td>IF X IS NOT=END.OF.ARRAY</td>
</tr>
<tr>
<td></td>
<td>BNE NEXT.PAIR</td>
<td>END SWITCH</td>
</tr>
<tr>
<td></td>
<td>THEN GO TO NEXT.PAIR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TST SWITCHED</td>
<td>MOVE B TO A</td>
</tr>
<tr>
<td></td>
<td>BNE NEXT.PASS</td>
<td>IF X=END.OF.ARRAY</td>
</tr>
<tr>
<td></td>
<td>THEN UNDO NEXT.PAIR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td>RTS</td>
</tr>
<tr>
<td></td>
<td>RETURN FROM BUBBLE.SORT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SWITCHED</td>
<td>DATA WORD</td>
</tr>
<tr>
<td></td>
<td>FCB 0</td>
<td>IF SWITCHED IS ZERO</td>
</tr>
<tr>
<td></td>
<td>END.OF.ARRAY</td>
<td>DATA DOUBLEWORD</td>
</tr>
<tr>
<td></td>
<td>FDB 0</td>
<td>THEN RETURN FROM BUBBLE.SORT</td>
</tr>
<tr>
<td></td>
<td>ARRAY</td>
<td>END.NEXT.PASS</td>
</tr>
<tr>
<td></td>
<td>RMB 256</td>
<td>ARRAY: RESERVE 256 WORDS</td>
</tr>
</tbody>
</table>

SWITCHED: DATA WORD
END.OF.ARRAY: DATA DOUBLEWORD
ARRAY: RESERVE 256 WORDS.
INTERRUPTS

There are 3 external signals which may cause an interruption of the normal instruction sequence processing: Reset, Interrupt Request, and Non-maskable Interrupt.

A Reset signal causes the contents of memory locations FFFE, FFFF to be loaded into PC and used as the address for the next instruction (without remembering the old PC value). HIGH OF PC + (FFFE)
LOW OF PC + (FFFF).

An Interrupt Request signal when the interrupt is disabled (I = 1) has no effect. If the Interrupt Request Signal remains active until (I = 0) the Interrupt will take place at that time.

An Interrupt Request signal when the interrupt is enabled (I = 0) or a Nonmaskable Interrupt signal causes:

1) the registers PC, X, A, B, and CC to be pushed into the stack
   (SP) + LOW OF PC
   (SP-1) + HIGH OF PC
   (SP-2) + LOW OF X
   (SP-3) + HIGH OF X
   (SP-4) + A
   (SP-5) + B
   (SP-6) + CC
   SP + SP-7

2) the interrupt to be disabled
   I + 1

3) the next instruction to be taken from memory location whose address is contained in locations FFF8, FFF9, for interrupt Request or locations FFFC, FFFD for Nonmaskable Interrupt.

   \[
   \begin{align*}
   \text{HIGH OF PC} + (\text{FFFF8}) & \quad \text{for Interrupt Request} \\
   \text{LOW OF PC} + (\text{FFFF9}) & \quad \text{for Nonmaskable Interrupt.}
   \end{align*}
   \]
ADD [DECIMAL] source TO destination [WITH CARRY]

adds the contents of the source into the destination [optionally plus the carry from previous addition].

destination must be either registers A or B.

source may be either an actual value or a memory location.

source may also be register B but only when destination is register A.

WITH CARRY option, the carry value will also be added.

WITH CARRY option cannot be used if source is register B.

DECIMAL option causes a two digit packed decimal add in place of an 8 bit two's complement add.

DECIMAL option may only be used when destination is register A.

ACTION:

destination ← source + destination [+ carry]

H ← \{ 1 if there is a carry from bit 3 to bit 4
      0 otherwise \}

N ← most significant bit of the result (bit 7)

Z ← \{ 1 if the result is zero
       0 otherwise \}

V ← \{ 1 if two's complement overflow results
       0 otherwise \}

C ← \{ 1 if there is a carry from most significant bit (bit 7)
       0 otherwise \}

PC is incremented to the next instruction.

Examples:

ADD 5 TO B
(e.g. If before B = -2 = 11111110, then afterwards
     B + 3 = 00000011, H = 1, N = 0, Z = 0, V = 0, C = 1, PC + PC + 2)

ADD DECIMAL B TO A
(e.g. if before B = 37\textsubscript{10} = 00110111 and A = 56\textsubscript{10} = 01010110, then afterwards
     A + 93\textsubscript{10} = 100100011, H = 1, N = 1, Z = 0, V = 0, C = 0, PC + PC + 2).

ADD (X) + 3 TO B WITH CARRY
(e.g. if before B = -8 = 11111000, C = 1, X = 5, (8) = 6, then afterwards
     B + -1 = 111111111, H = 0, N = 1, Z = 0, V = 0, C = 0, PC + PC + 2).
AND source TO destination

logical bit by bit "and" of the 8 source bits into the 8 destination bits.

destination must be either register A or B.
source may be either an actual value or a memory location.

ACTION:

destination ← source AND destination (bit by bit)

N ← most significant bit of result
Z ← \[ \begin{cases} 1 & \text{if result is zero} \\ 0 & \text{otherwise} \end{cases} \]
V ← 0

PC is incremented to next instruction.

Examples:

AND HEX FO TO A
(e.g. if before A = 11001100
then after A = 11000000, N = 1, Z = 0, V = 0, PC = PC + 2)

AND (X) TO B
(e.g. if before B = 00001111, X = 4, \( \text{(4)} \) = 11000000
then after B = 0, N = 0, Z = 1, V = 0, PC = PC + 2).
CALL \text{address}

Pushes the contents of PC into the stack and loads the address into PC.

\text{address} is the memory location of the first instruction of a subroutine.

\textbf{ACTION}:

\begin{align*}
\text{(SP)} & \quad + \text{PC} \rightarrow 0 \\
\text{(SP-1)} & \quad + \text{PC} \rightarrow 15-8 \\
\text{SP} & \quad + \text{SP} \rightarrow 2 \\
\text{PC} & \quad + \text{address}
\end{align*}

\textbf{Example}:

\text{CALL SUB}

\begin{align*}
\text{e.g. if before} \quad & \text{SP} = \text{HEX FF29}, \text{PC} = \text{HEX FEE3} \quad \text{and SUB is} \quad \text{address} = \text{HEX FE02} \\
\text{then afterwards} \quad & \begin{align*}
(\text{FF29}) & \quad + \text{HEX E3} \\
(\text{FF28}) & \quad + \text{HEX FE} \\
\text{SP} & \quad + \text{HEX FF27} \\
\text{PC} & \quad + \text{HEX FE02}
\end{align*} \quad \begin{align*}
(\text{SP}) & \quad + \text{PC} \rightarrow 7-0 \\
(\text{SP}) & \quad + \text{PC} \rightarrow 15-8 \\
\text{SP} & \quad + \text{SP} \rightarrow 2 \\
\text{PC} & \quad + \text{SUB} \quad \text{address}.
\end{align*}
\end{align*}
causes: 1) the registers to be pushed onto the stack;  
2) the interrupt to be disabled; and  
3) the next instruction to be taken from memory location  
whose address is contained in locations FFFA and FFFB.

ACTION:

\[(\text{SP}) + \text{PC}_{7-0}\]  \((\text{LOW OF PC})\)
\[(\text{SP}-1) + \text{PC}_{15-8}\]  \((\text{HIGH OF PC})\)
\[(\text{SP}-2) + \text{X}_{7-0}\]  \((\text{LOW OF X})\)
\[(\text{SP}-3) + \text{X}_{15-8}\]  \((\text{HIGH OF X})\)
\[(\text{SP}-4) + A\]
\[(\text{SP}-5) + B\]
\[(\text{SP}-6) + \text{CC}\]
\[\text{SP} + \text{SP}_{-7}\]
\[l + 1\]
\[\text{PC}_{15-8} + (\text{FFFA})\]  \((\text{HIGH OF PC})\)
\[\text{PC}_{7-0} + (\text{FFFB})\]  \((\text{LOW OF PC})\)
clears all bits in the \textit{destination} to logical zero
\textit{destination} may be a memory location, register A or B, or one of the flags: CARRY or OVERFLOW

\textbf{ACTION:}

\textit{destination} = 0
If \textit{destination} is not a flag then also $N + 0, Z + 1, V + 0, C + 0$
PC is incremented to next instruction.

\textbf{Examples:}

CLEAR (X+3)
CLEAR CARRY.
COMPLEMENT

COMPLEMENT \textit{destination}

complements each bit of the \textit{destination}

\textit{destination} may be a memory location or register A or B.

\textbf{ACTION:}

destination + HEX FF = destination

N + { 1 if result is zero
         0 otherwise

Z + 0

V + 0

C + 1

PC is incremented to next instruction.

\textbf{Example:}

COMPLEMENT M

(e.g. if before \( M = 00000001 \)

then afterwards \( M = 11111110, N + 1, Z = 0, V + 0, C + 1 \).
adjusts the packed decimal carry (if $A_{3-0} > 9$ or $H = 1$) from least significant decimal digit of $A$ into most significant decimal digit ($A_{7-4}$), enabling packed decimal addition.

There are no options. Only register $A$ may be adjusted.

**ACTION:**

if $A_{3-0} > 9$ or $H = 1$ then $A + A + 6$

if $A_{7-4} > 9$ or $C = 1$ then $A + A + \text{HEX } 60$ and $C + 1$

$N + A_7$ of result (most significant bit)

$Z \left\{ \begin{array}{ll} 1 & \text{if result is zero} \\ 0 & \text{otherwise} \end{array} \right.$

$V \left\{ \begin{array}{ll} ? & \text{otherwise} \end{array} \right.$

$C \left\{ \begin{array}{ll} 1 & \text{if decimal carry occurs} \\ 0 & \text{otherwise} \end{array} \right.$

PC is incremented to next instruction

**Example:**

The instruction `ADD DECIMAL M TO A` is identical to the sequence of two instructions:

`ADD M TO A`

`DECIMAL ADJUST A`. 
DECREMENT

DECREMENT destination [BY number]

subtracts one (1) [or number] from destination.

destination may be a memory location or one of registers: A, B, X, or SP.

ACTION:

destination + destination = 1 [or number]

N + 1 if destination is X or SP then unchanged otherwise the most significant bit in result

Z + 1 if destination is SP then unchanged otherwise { 1 if result is zero 0 if result is not zero

V + 1 if destination is X or SP then unchanged otherwise { 1 if destination changes from HEX `80 to HEX 3F 0 otherwise

PC is incremented to next instruction

Examples:

DECREMENT X
DECREMENT (X)
causes the interrupt request signal (IRQ) to be ignored.

ACTION:

\[ 1 + 1 \] (1 is the interrupt mask, bit \( \text{CC}_4 \))

PC is incremented to next instruction.
DO [name] [FOREVER] one or more instructions
END [name]

causes the included sequence of instructions to be treated as a single compound instruction, called a DO block.

DO blocks serve two distinct functions:

(a) conditionally executing more than one instruction after an IF statement;

(b) looping (by using FOREVER option). FOREVER option causes the block to be repeated endlessly (unless interrupted by the execution of an UNDO instruction).

DO blocks may be nested.

name (if used) on an END instruction must match the name used on a corresponding DO instruction.

Example:

IF A > B THEN DO SWAP.A.WITH.B
    MOVE A TO TEMP
    MOVE B TO A
    MOVE TEMP TO B
END SWAP.A.WITH.B
ENABLE INTERRUPT

causes the presence of an interrupt signal (IRQ) to be recognized. (See section on interrupt for details.)

ACTION:

1 + 0 (1 is the interrupt mask, bit CC4)

PC is incremented to next instruction.
"exclusive or" bit by bit the source into the destination.

destination must be either register A or B.

source may be either an actual value or a memory location.

**ACTION:**

\[ \text{destination} \oplus \text{source} \oplus \text{destination} \]  
(bit by bit)

- \( N \) = most significant bit of result
- \( Z \) = \( \begin{cases} 1 & \text{if result is zero} \\ 0 & \text{otherwise} \end{cases} \)
- \( V = 0 \)
- \( PC \) is incremented to next instruction.

**Example:**

FOR M TO B  
(e.g. if before \( M = 11110000 \) and \( B = 11001100 \))

then afterwards \( B = 00111100, N = 0, Z = 0, V = 0, PC = PC + 3 \)
fills the next \textit{number} of bytes of memory with "dummy" instructions that do nothing but increment the PC (reserves program space to be refilled later).

\textbf{ACTION:}

\[ \text{PC} + \text{PC} + \text{number} \]

\textbf{Example:}

\texttt{FILL 5 WORDS}
loads \textit{address} into PC.

\textit{address} is the memory location of the next instruction to be executed.

\textbf{ACTION:}

\texttt{PC} + \texttt{address}

\textbf{Example:}

\texttt{GO TO START}
IF condition THEN instruction [ELSE instruction$_2$] or IF condition THEN instruction [ELSE instruction$_2$]

where:

flag is [NOT] SET [CLEAR]
RESULT is [NOT] test
source is [WHEN] INCREMENTED [DECREMENTED] [BY number] [IS] comparison
source MATCHES ANYWHERE [NOWHERE] source$_2$

flag may be CARRY, OVERFLOW, NEGATIVE, or ZERO

test is

NEGATIVE
ZERO

comparison is [UNSIGNED]

> = or = >
< = or = <

instruction [and instruction$_2$] may be any instruction (other than another IF) or may be a DO block.

condition sets up appropriate branching by setting and testing condition codes.

(Note: if only the setting of condition codes is required then the REGARDLESS instruction should be used.)
RESULT option tests (without setting) the condition codes as already set by a previous instruction.

In general the condition part changes only the condition codes (register CC).

However, the WHEN option also alters the source. 

One of source or source₂ of a comparison must be register A,B, or X and the other one must be either a numerical value of a memory location.

test of register X may only be ZERO or NONZERO 

comparison of register X may only be = or NOT=

one of source or source₂ of MATCHES must be register A or B and the other must be either a numerical value or a memory location.

UNSIGNED comparisons consider source and source₂ as absolute binary (nonnegative) numbers, whereas normally numbers are considered to be two's complement (signed).

MATCHES causes a bit by bit comparison of source and source₂ to see if any corresponding bit is set (1) in both source and source₂.

ANYWHERE option does not affect MATCHES

NOWHERE option sees if no corresponding bits are set (1) in both source and source₂.

The optimal word IS serves for readability only and does not affect the condition.

The condition CARRY IS SET for example is identical to the condition CARRY, while the condition CARRY IS CLEAR is identical to NOT CARRY.
ACTION:

CC register (H,N,Z,V,C flags) are determined by the condition

- source is incremented or decremented by WHEN option
- PC is incremented as determined by the condition and instructions in THEN part (or ELSE part)
- other registers and memory locations are affected only by the instruction in the THEN part (or ELSE part)

Examples:

IF A < B THEN ADD B TO A
(e.g. if before A = 3 and B = 2 then after N = 0, Z = 0, C = 0, PC = PC + 4) and A still contains 3.

IF X WHEN DECREMENTED IS ZERO THEN RETURN FROM INTERRUPT
(e.g) if before X = 5, then afterwards X = 4, Z = 0, PC = PC + 4).
INCREMEŃT destination [BY number]

adds one (1) [or number] to destination

destination may be a memory location or one of registers: A, B, X, or SP.

ACTION:

destination + destination + 1

N \rightarrow \begin{cases} 
\text{if } \text{destination} \text{ is X or SP then unchanged.} \\
\text{otherwise the most significant bit in result.}
\end{cases}

Z \rightarrow \begin{cases} 
\text{if } \text{destination} \text{ is SP then unchanged} \\
\text{otherwise } \begin{cases} 
1 \text{ if result is zero} \\
0 \text{ if result is not zero}
\end{cases}
\end{cases}

V \rightarrow \begin{cases} 
\text{if } \text{destination} \text{ is X or SP then unchanged} \\
\text{otherwise } \begin{cases} 
1 \text{ if } \text{destination} \text{ changes from HEX 7F to HEX 80} \\
0 \text{ otherwise}
\end{cases}
\end{cases}

PC is incremented to next instruction.

Example:

INCREMENT SP.
moves source into destination.

There is no machine distinction between MOVE and INPUT or OUTPUT. It might be useful to use INPUT if the source is a peripheral unit or to use OUTPUT if the destination is a peripheral.

The word FROM has no effect.

Moves may occur only between the following source/destination pairs (in either direction):

- A <-> B
- X <-> SP
- A <-> CC
- A <-> memory location
- B <-> memory location
- X <-> memory location (and the next location)
- SP <-> memory location (and the next location)
- value + A or B or X or SP.

**ACTION:** (Note: \( A_i \) is the \( i \)-th bit of register A.)

\[
\begin{align*}
\text{destination + source} \\
H + & \{ \quad A_5 \text{ on MOVE A TO CC} \\
& \quad \text{unchanged otherwise} \\
I + & \{ \quad A_4 \text{ on MOVE A TO CC} \\
& \quad \text{unchanged otherwise} \\
N + & \{ \quad A_3 \text{ on MOVE A TO CC} \\
& \quad \text{unchanged on MOVE CC TO A, MOVE X TO SP, or MOVE SP TO X} \\
& \quad \text{most significant bit of source otherwise} \\
Z + & \{ \quad A_2 \text{ on MOVE A TO CC} \\
& \quad \text{unchanged on MOVE CC TO A, MOVE X TO SP, or MOVE SP TO X} \\
& \quad \text{otherwise} \{ 1 \text{ if source } = 0 \} \\
& \quad \{ 0 \text{ if source } \neq 0 \} \\
V + & \{ \quad A_1 \text{ on MOVE A TO CC} \\
& \quad \text{unchanged on MOVE CC TO A, MOVE X TO SP, or MOVE SP TO X} \\
& \quad \text{0 otherwise} \\
C + & \{ \quad A_0 \text{ on MOVE A TO CC} \\
& \quad \text{unchanged otherwise} \\
\end{align*}
\]

PC is incremented to next instruction.
MOVE 6 TO B.
MOVE B TO (X+3).
MOVE ADDRESS OF TABLE TO X.

Examples:

MOVE (cont'd)
NEGATE destination

replaces the contents of the destination with its two's complement.

destination may be a memory location or register A or B.

ACTION:

destination + 0 = destination

N = most significant bit of the result

Z = \begin{cases} 1 & \text{if result is zero} \\ 0 & \text{otherwise} \end{cases}

\begin{align*}
V &= \begin{cases} 1 & \text{on two's complement overflow (i.e. if } destination = \text{HEX } 80 \\
0 & \text{otherwise} \end{cases} \\
C &= \begin{cases} 0 & \text{if } destination \text{ is zero (before and after)} \\
1 & \text{otherwise} \end{cases}
\end{align*}

PC is incremented to next instruction.

Example:

NEGATE M

(e.g. if before M = 0000000)

then afterwards M = 11111111, N = 1, Z = 0, V = 0; C = 1)
Logical bit by bit "or" of the 8 source bits into the 8 destination bits.

destination must be either register A or B.

source may be either an actual value or a memory location.

**ACTION:**

\[ \text{destination} \lor \text{source} \lor \text{destination} \ (\text{bit by bit}) \]

\[ N + \text{most significant bit of result} \]

\[ Z + \begin{cases} 1 & \text{if result is zero} \\ 0 & \text{otherwise} \end{cases} \]

\[ V + 0 \]

PC is incremented to next instruction

**Example:**

OR M TO A

(e.g. if before \( M = 11110000 \) and \( A = 11001100 \)

then afterwards \( A = 11111100, N = 1, Z = 0, V = 0, PC = PC + 3 \) \)
pops the stack, replacing the contents of the register (A or B) by the top of the stack

**ACTION:**

the register + (SP + 1)

SP ← SP + 1

**Example:**

POP INTO B
pushes the contents of the register (A or B) onto the top of the stack.

**ACTION:**

(SP) + the contents of the register

SP ← SP - 1

**Example:**

PUSH A
causes execution to proceed to the first instruction of DO block name, or of the most recently nested DO block if no name is specified.

Example:

DO LOOP FOREVER:
  ...
  IF A > B THEN REDO LOOP
  ...
  END LOOP
is equivalent to:

DO LOOP FOREVER
  DO INNER.BLOCK
  ...
  IF A > B THEN UNDO INNER.BLOCK
  ...
  END INNER.BLOCK
END LOOP
REGARDLESS IF condition

serves to set the condition codes corresponding with the given condition.

condition is defined in the IF instruction.

ACTION:
CC + values appropriate with the condition.
PC is incremented to next instruction.

Example:
REGARDLESS IF A < B
CALL SUBROUTINE USING CONDITION CODES
pops the top of the stack into the PC.

\textbf{INTERRUPT} option (and \textbf{MONITOR} option) causes the stack to be popped into all registers.

\textbf{ACTION}: without presence of \textbf{INTERRUPT} nor \textbf{MONITOR} option.

\begin{align*}
PC_{15-8} & = (SP + 1) \\
PC_{7-0} & = (SP + 2) \\
SP & = SP + 2.
\end{align*}

\textbf{ACTION}: with \textbf{INTERRUPT} or \textbf{MONITOR} option

\begin{align*}
CC & = (SP + 1) \\
B & = (SP + 2) \\
A & = (SP + 3) \\
X_{15-8} & = (SP + 4) \\
X_{7-0} & = (SP + 5) \\
PC_{15-8} & = (SP + 6) \\
PC_{7-0} & = (SP + 7) \\
SP & = SP + 7
\end{align*}

\textbf{Example:}

\texttt{RETURN FROM BUBBLE.SORT}
ROTATE destination [LEFT] [number BITS]

rotates circularly the contents of the nine (9) bits comprising the destination and the carry bit (flag C) either left or right by one (1) bit [or by number of bits]

destination may be a memory location or register A or B.

ACTION:

destination \rightarrow \text{rotated destination} \text{ (with carry)}
C \rightarrow \text{appropriate rotated bit of destination} \text{ (with carry)}
N \rightarrow \text{most significant bit of result} \text{ (bit 7)}
Z \rightarrow \begin{cases} 1 & \text{if result destination is zero} \\ 0 & \text{otherwise} \end{cases}
V \rightarrow \begin{cases} 1 & \text{if after rotation } C \neq N \\ 0 & \text{if after rotation } C = N. \end{cases}

Example:

ROTATE B LEFT
(e.g. before $B = 01111000$ and $C = 1$
then afterwards $B = 1110011$, $C = 0$, $N + 1$, $Z = 0$, $V + 1$).
sets the flag to logical 1 (one)
flag must be either CARRY or OVERFLOW.

**ACTION:**

flag + 1

PC is incremented to next instruction

**Example:**

SET CARRY.
shifts left or right the contents of the destination by one (1) bit
[or by number of bits].
Bits leaving the destination are shifted through the carry (C).
When shifting RIGHT ARITHMETICALLY the cleared bits on the left are
filled up with the leftmost (sign) bit.
On all other shifts the cleared bits are filled up with zero's.
destination may be a memory location or register A or B.

ACTION:

destination + shifted destination
C + last bit shifted out of destination
N + most significant bit of result (bit 7)
Z + \{ 1 if result is zero
        0 otherwise
V + \{ 1 if after the shift C ≠ N
        0 if after the shift C = N.

Examples:
SHIFT B LEFT
(e.g. if before B = 01111000
then afterwards B = 11110000, C = 0, N + 1, Z = 0, V + 1).

SHIFT A RIGHT ARITHMETICALLY 4 BITS
(e.g. if before A = 10010110
then afterwards A = 1111001, C = 0, N = 1, Z = 0, V + 1).
SUBTRACT source FROM destination [WITH BORROW]

subtracts the contents of the source from the destination [optionally also minus the borrow (Carry bit) from previous subtraction].

destination must be either register A or B.

source may be either an actual value or a memory location.

source may also be register B but only when destination is register A.

WITH BORROW option, the carry value will also be subtracted.

WITH BORROW option cannot be used if source is register B.

ACTION:

destination + destination - source [- carry]

N ← most significant bit of the result (bit 7)

Z ← { 1 if result is zero
       0 otherwise

V ← { 1 if two's complement overflow results
       0 otherwise

C ← { 1 if the most significant bit must borrow
       0 otherwise

PC is incremented to next instruction.

Examples:

SUBTRACT B FROM A
(e.g. if before A = 2 and B = -2
then afterwards A = 0, N = 0, Z = 1, V = 0, C = 1, PC = PC + 1)

SUBTRACT 6 FROM B WITH BORROW
(e.g. if before B = -2 and C = 1
then afterwards B = -9, N = 1, Z = 0, V = 0, C = 0, PC = PC + 2).
serves to escape from a DO FOREVER loop. UNDO can be used to escape from several levels of nesting by using the same name as one of the enclosing DO blocks. If no name is specified, then the most recently nested DO block is undone.

UNDO is normally used in the THEN part of an IF statement.

**ACTION:**

Causes a branch to the instruction following the END of the relevant DO block.

**Examples:**

```
DO LOOP FOREVER
  IF X WHEN DECREMENTED IS ZERO
  THEN UNDO LOOP
  ADD (X) + ENQ.OF.ARRAY TO A
END LOOP.
```
pushes the registers into the stack (with the PC already incremented to next instruction) and suspends execution of further instructions until an interrupt occurs.

**ACTION:**

PC is incremented to next instruction

\[
\begin{align*}
(SP) &\rightarrow PC_{7-0} & \text{(LOW OF PC)} \\
(SP-1) &\rightarrow PC_{15-8} & \text{(HIGH OF PC)} \\
(SP-2) &\rightarrow X_{7-0} & \text{(LOW OF X)} \\
(SP-3) &\rightarrow X_{15-8} & \text{(HIGH OF X)} \\
(SP-4) &\rightarrow A \\
(SP-5) &\rightarrow B \\
(SP-6) &\rightarrow CC \\
SP &\rightarrow SP-7
\end{align*}
\]
ASCII MNEMONICS causes the 32 ASCII control character mnemonics (given in the table below) to be implicitly defined as their ASCII equivalent.

ASCII MNEMONICS (if used) should occur at the beginning of the program.

<table>
<thead>
<tr>
<th>HEX</th>
<th>ASCII MNEMONIC</th>
<th>ASCII MNEMONIC</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>NUL</td>
<td>ACK</td>
<td>06</td>
</tr>
<tr>
<td>01</td>
<td>SOH</td>
<td>BEL</td>
<td>07</td>
</tr>
<tr>
<td>02</td>
<td>STX</td>
<td>BS</td>
<td>08</td>
</tr>
<tr>
<td>03</td>
<td>ETX</td>
<td>CAN</td>
<td>18</td>
</tr>
<tr>
<td>04</td>
<td>EOT</td>
<td>CR</td>
<td>0D</td>
</tr>
<tr>
<td>05</td>
<td>ENQ</td>
<td>DC1</td>
<td>11</td>
</tr>
<tr>
<td>06</td>
<td>ACK</td>
<td>DC2</td>
<td>12</td>
</tr>
<tr>
<td>07</td>
<td>BEL</td>
<td>DC3</td>
<td>13</td>
</tr>
<tr>
<td>08</td>
<td>BS</td>
<td>DC4</td>
<td>14</td>
</tr>
<tr>
<td>09</td>
<td>HT</td>
<td>DLE</td>
<td>10</td>
</tr>
<tr>
<td>0A</td>
<td>LF</td>
<td>EM</td>
<td>19</td>
</tr>
<tr>
<td>0B</td>
<td>VT</td>
<td>ENQ</td>
<td>05</td>
</tr>
<tr>
<td>0C</td>
<td>FF</td>
<td>EOT</td>
<td>04</td>
</tr>
<tr>
<td>0D</td>
<td>CR</td>
<td>ESC</td>
<td>18</td>
</tr>
<tr>
<td>0E</td>
<td>SO</td>
<td>ETB</td>
<td>17</td>
</tr>
<tr>
<td>0F</td>
<td>SI</td>
<td>ETX</td>
<td>03</td>
</tr>
<tr>
<td>10</td>
<td>DLE</td>
<td>FF</td>
<td>0C</td>
</tr>
<tr>
<td>11</td>
<td>DC1</td>
<td>FS</td>
<td>1C</td>
</tr>
<tr>
<td>12</td>
<td>DC2</td>
<td>GS</td>
<td>1D</td>
</tr>
<tr>
<td>13</td>
<td>DC3</td>
<td>HT</td>
<td>09</td>
</tr>
<tr>
<td>14</td>
<td>DC4</td>
<td>LF</td>
<td>0A</td>
</tr>
<tr>
<td>15</td>
<td>NAK</td>
<td>NAK</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>SYN</td>
<td>NUL</td>
<td>00</td>
</tr>
<tr>
<td>17</td>
<td>ETB</td>
<td>RS</td>
<td>1E</td>
</tr>
<tr>
<td>18</td>
<td>CAN</td>
<td>SI</td>
<td>0F</td>
</tr>
<tr>
<td>19</td>
<td>EM</td>
<td>SO</td>
<td>0E</td>
</tr>
<tr>
<td>1A</td>
<td>SUB</td>
<td>SOH</td>
<td>01</td>
</tr>
<tr>
<td>1B</td>
<td>ESC</td>
<td>STX</td>
<td>02</td>
</tr>
<tr>
<td>1C</td>
<td>FS</td>
<td>SUB</td>
<td>1A</td>
</tr>
<tr>
<td>1D</td>
<td>GS</td>
<td>SYN</td>
<td>16</td>
</tr>
<tr>
<td>1E</td>
<td>RS</td>
<td>US</td>
<td>1F</td>
</tr>
<tr>
<td>1F</td>
<td>US</td>
<td>VT</td>
<td>0B</td>
</tr>
</tbody>
</table>
Causes the listing to show one instruction per line including operation and operands.

*CONDENSED LISTING* is the default which may be overridden by *EXPANDED LISTING*.
enters the number(s), name(s), or string into (consecutive bytes of) memory as data for other instructions.

WORD option causes an 8-bit word to be used for each number or name truncating on the left any excess over 255 (i.e. number mod 256 = 2^8 will result).

DOUBLEWORD option causes two consecutive 8-bit words to be used for each number or name.

Examples:

DATA WORD 3,15,255
(the data sequence 00000011, 00001111, 11111111 will occur in memory)

DATA DOUBLEWORD C
(the address of C will occur in two successive memory locations).

DATA "THIS IS A STRING"
(the ASCII character string HEX 54484953204452494E47 will occur in 16 successive memory locations, unless the EBCDIC pseudoinstruction preceded, causing the EBCDIC character string HEX E3C8C9E240C9E240C140E2E3D9C9D5C7 to occur).
allows \textit{name} to be used either:

1. instead of "\textit{CONTENTS OF number}" as in Example 1;
2. or as a value (when preceded by "\textit{NUMBER}") as in Example 2.

In either case, the \texttt{DEFINE} statement provides the facility for simultaneous modification of many instructions.

\textit{defined-name} is a name already defined in a previous \texttt{DEFINE} statement.

\textbf{Examples:}

1. \texttt{DEFINE TTY AS HEX FCF5}
   \texttt{MOVE TTY TO A}
   (these two instructions are equivalent to the instruction \texttt{MOVE \textit{CONTENTS OF HEX FCF5} TO A}).

2. \texttt{DEFINE Q AS 5}
   \texttt{MOVE NUMBER Q TO B}
   (moves the number 5 into register B as does the instruction \texttt{MOVE 5 TO B}).
causes the EBCDIC character set to be used in place of ASCII for all string characters, that is for all characters surrounded by double quotes ".

EBCDIC option (if used) should occur at the beginning of the program.
EXPANDED LISTING

causes the listing to show one word of object language per line for ease of manually loading into M-6800 memory.

EXPANDED LISTING overrides CONDENSED LISTING option.
causes the succeeding instructions to be located starting from memory location number.

Example:

LOCATION HEX 500
MOVE HEX -5FF TO SP
(causes the MOVE instruction to be located in memory location HEX 500)
RESERVE \textit{number} WORDS

causes a gap of \textit{number} words in between the previous instruction and the succeeding instruction, generally with the intention of data storage.
causes columns 72 through 80 of the source file to contain the input line sequence number followed by 000. For example the first line will contain sequence number 00001000 and line 258 will have sequence number 00258000.

SEQUENCE NUMBERS is the default which may be overridden by SEQUENCELESS.
SEQUENCELESS causes each source file line to include the input characters (up to 80) without change.

SEQUENCELESS overrides SEQUENCE NUMBERS option.
is equivalent to a label.

Example:

SUBROUTINE SIN

is equivalent to:

SIN:
causes the assembly to be completed and if the program is error-free, causes the object program to be saved on disk file named M6800CODE.

?END is needed only for interactive SMAL. An end-of-file indication will suffice for batch SMAL.
causes the assembly to be completed and causes a copy of the source program to be saved on disk file named SOURCEFILE. The object program if error-free will be saved on disk file named M6800CODE.