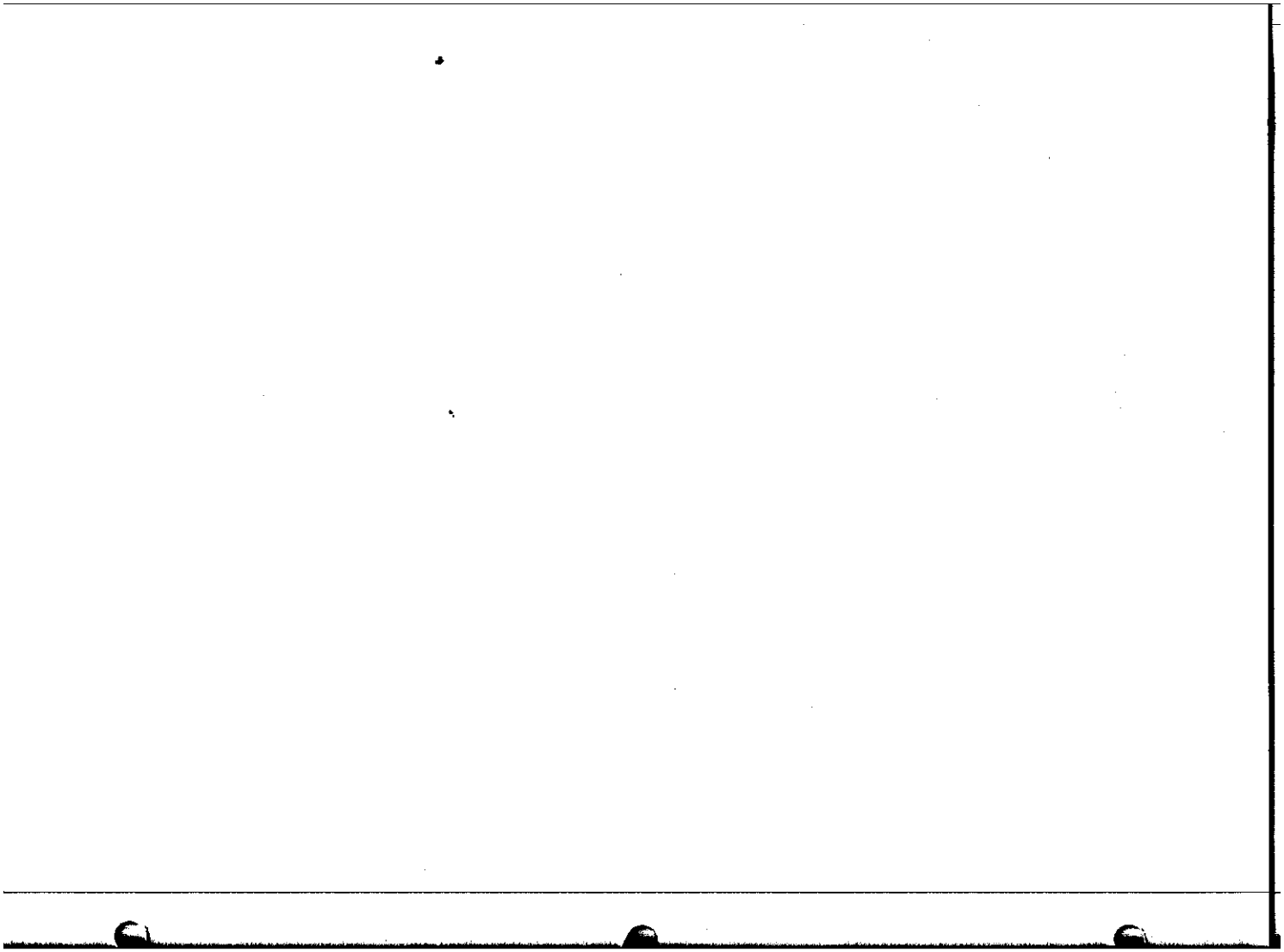
Programming Manual

altair™ 680b

Table of Contents

MITS, Inc. ©1976
2450 Alamo S.E.
Albuquerque, N.M. 87106



I INTRODUCTION

The Altair 680b Programming Manual describes the format of the 680b assembly code source language and the 6800 MPU instruction set and addressing modes.

A brief overview of arithmetic programming techniques and some general purpose sample programs are also included.

This manual is in no way intended to be a beginning course in computer programming.

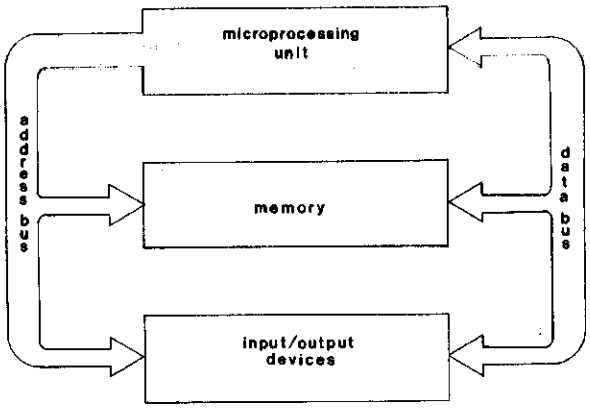


Figure 2-1

Microcomputer System Block Diagram

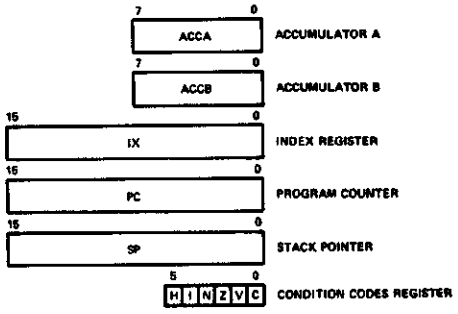


Figure 2-2. Programming Model of M6800

II SYSTEM DESCRIPTION

In order to program a computer in machine language or assembly code, it is necessary to have at least a block diagram level understanding of the computer hardware.

A general purpose microcomputer (see figure 2-1) consists of a microprocessing unit (MPU), a memory, and a number of input and output devices. These components are linked together by an address bus and a data bus.

The computer memory is used to store instructions and data for use by the MPU. In the 680b, the memory is organized into 8 bit words called bytes. Each memory byte is assigned a unique 16 bit address. This address is used by the MPU to gain access to the contents of a particular memory byte.

Input and output devices, such as Teletypes, CRT Terminals, and paper tape readers are used for communication between the computer and the external world. Each I/O device in a 680b system has one or more unique 16 bit addresses assigned to it.

The MPU is responsible for controlling the microcomputer system and performing all arithmetic and logic operations. The MPU must be told what steps to execute to perform a given task. This is accomplished by storing a program into the computer's memory. Once a program is stored in memory, a register in the MPU called the Program Counter (PC) is loaded with the address of the memory byte which contains the first instruction of the program. When the computer is put into the run mode, the MPU puts the address contained in the PC on the address bus and reads the contents of that location via the data bus. The instruction that has been read is executed after the PC is incremented to point to the next instruction.

This sequence is repeated until the processor is halted.

The 680b MPU is a Motorola M6800 which operates on 8-bit binary numbers presented to it via the data bus. A given number (byte) may represent either data or an instruction to be executed, depending on where it is encountered in a program. The M6800 has 72 unique instructions, however, it recognizes and takes action on 197 of the 256 possibilities that can occur using an 8-bit word length. This larger number of instructions results from the fact that many of the executive instructions have more than one addressing mode.

These addressing modes refer to the manner in which the program causes the MPU to obtain its instructions and data. The programmer must have a method for addressing the MPU's internal registers and all of the external memory locations. The complete executive instruction set and the applicable addressing modes are summarized in Figure 4-1, however, the addressing modes will be described in greater detail prior to introducing the instruction set in a later section. A programming model of the M6800 is shown in Figure 2-2. The programmable registers consist of: two 8-bit Accumulators; a 6-bit Condition Code Register; a Program Counter, a Stack Pointer, and an Index Register, each 16 bits long.

III SOURCE LANGUAGE & ADDRESSING MODES

While programs can be written in the MPU's language, that is, binary numbers, there is no easy way for the programmer to remember the particular word that corresponds to a given operation. For this reason, instructions are assigned a three letter mnemonic symbol that suggests the definition of the instruction. The program is written as a series of source statements using this symbolic language and then translated into machine language. The translation can be done manually using an alphabetic listing of the symbolic instruction set such as that shown in Appendix A. More often, the translation is accomplished by means of a special computer program referred to as an assembler.

The source language for the M6800 microprocessing unit is built around 72 mnemonic instructions and 12 assembler directives. Section III deals with the details of the character set and format of the source language.

CHARACTER SET

The characters used in the source language for the 680b assembler form a sub-set of ASCII (American Standard Code for Information Interchange, 1968). The ASCII Code is shown in Figure 3-1. The following characters are recognized by the assembler.

1. The alphabet A through Z
2. The integers 0 through 9
3. Four arithmetic operators:
   + - * /
4. Characters used as special prefixes:
   # (pounds sign) specifies the immediate mode of addressing
   $ (dollar sign) specifies a hexadecimal number
   @ (commercial at) specifies an octal number
   % (percent) specifies a binary number
   ' (apostrophe) specifies an ASCII literal character
5. Characters used as special suffices:
   B (letter B) specifies a binary number
   H (letter H) specifies a hexadecimal number
   O (letter O) specifies an octal number
   Q (letter Q) specifies an octal number
6. Four separating characters:
   SPACE
   Horizontal TAB
   CR (carriage return)
   , (comma)
   The use of horizontal TAB is always optional, and can be replaced by SPACE.
7. A comment in a source statement may include any characters with ASCII hexadecimal values from 20 (SP) through 5F (___).

8. In addition to the above, the assembler has the capability of reading strings of characters and of entering the corresponding 7-bit ASCII code into specified locations in the memory. This capability is provided by the assembler directive FCC (See Appendix B). Any characters corresponding to ASCII hexadecimal values 20 (SP) through 5F (___) can be processed. This kind of processing can also be done, for a single ASCII character, by using the immediate mode of addressing with an operand in the form "#C".

BITS 4 thru 6 → / BITS 0 thru 3 ↓	—	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P		p	
1	SOH	DC1	!	1	A	Q	a	q	
2	STX	DC2	"	2	B	R	b	r	
3	ETX	DC3	#	3	C	S	c	s	
4	EOT	DC4	$	4	D	T	d	t	
5	ENQ	NAK	%	5	E	U	e	u	
6	ACK	SYN	&	6	F	V	f	v	
7	BEL	ETB	'	7	G	W	g	w	
8	BS	CAN	(	8	H	X	h	x	
9	HT	EM	)	9	I	Y	i	y	
A	LF	SUB	*	:	J	Z	j	z	
B	VT	ESC	+	;	K	[	k	{	
C	FF	FS	,	<	L	/	l	/	
D	CR	GS	-	=	M	]	m	}	
E	SO	RS	.	>	N	⟨	n	≈	
F	SI	US	/	?	O	—	o	DEL	

Figure 3-1 ASCII Code

FIELDS OF THE SOURCE STATEMENT

A source statement includes from one to four fields. From left to right, the four fields are:

(1) label (2) operator (mnemonic) (3) operand (4) comment

The comment is optional, and may be used in most source statements. Comments are intended for the convenience of the programmer, and to facilitate documentation of the program. A label is required for some statements which are involved in the definition of symbols and, in some cases, at the destinations of branches and jump instructions. An operand field may or may not be present depending on the nature of the operator. The mnemonic operator must be present in any statement except when the statement consists only of a comment.

With one optional exception (explained below), the successive fields within a statement are separated by,

either: one or more SPACE characters

or: horizontal TAB

The use of the horizontal TAB is hardware-dependent in that its availability will depend on the particular type of terminal in use. The SPACE bar may always be used rather than the TAB key.

CAUTION

A SPACE in the first character position of a source statement is used to indicate that a label is not included in the statement. A label, if used, must begin in the first character position of the source statement. It follows from the above that, when typing a source program into a file in which the statements are identified by line numbers, there will be only one space following the line number if the statement includes a label. Two or more spaces following the line number will indicate that a label is not used.

The exception to the foregoing rule relating to SPACE or horizontal TAB between the elements of a source statement applies to operators with dual addressing in the operand field (indicated by the column headed "Dual Operand" in Figure 3-2 and to certain other operators if they are functioning in the "accumulator mode" of addressing (indicated by the column headed "ACCX" in Figure 3-2. In these cases, the first character of the operand field is either A or B (indicating accumulator A or B), and the second character is a SPACE. The programmer then has the option of omitting the SPACE between the operator and the operand field. This results in an apparent four-character format, as for example "ADCA", "ASRB", "STAA", "TSTB", and similarly.

Label Field

An asterisk (*) in the first character position of a statement causes the entire statement to become a comment. Otherwise, the comment will be preceded in the statement by one or more fields of the other three types, and the comment will occupy the last field in the statement.

Except in some cases when it is used with the mnemonic operator EQU (see below) a label always corresponds to a numerical address in the programmable system. It provides a means of referring to that address by using a symbol identical with the label. The address represented by the label (or symbol) may be that of an instruction in the machine code or of a location in the memory where data is stored.

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Inherent	Relative
ABA		•	•	•	•	•	2	•
ADC	x	•	2	3	4	5	•	•
ADD	x	•	2	3	4	5	•	•
AND	x	•	2	3	4	5	•	•
ASL		2	•	•	6	7	•	•
ASR		2	•	•	6	7	•	•
BCC		•	•	•	•	•	•	4
BCS		•	•	•	•	•	•	4
BEA		•	•	•	•	•	•	4
BGE		•	•	•	•	•	•	4
BGT		•	•	•	•	•	•	4
BHI		•	•	•	•	•	•	4
BIT	x	•	2	3	4	5	•	•
BLE		•	•	•	•	•	•	4
BLS		•	•	•	•	•	•	4
BLT		•	•	•	•	•	•	4
BMI		•	•	•	•	•	•	4
BNE		•	•	•	•	•	•	4
BPL		•	•	•	•	•	•	4
BRA		•	•	•	•	•	•	4
BSR		•	•	•	•	•	•	8
BVC		•	•	•	•	•	•	4
BVS		•	•	•	•	•	•	4
CBA		•	•	•	•	•	2	•
CLC		•	•	•	•	•	2	•
CLI		•	•	•	•	•	2	•
CLR		2	•	•	6	7	•	•
CLV		•	•	•	•	•	2	•
CMP	x	•	2	3	4	5	•	•
COM		2	•	•	6	7	•	•
CPX		•	3	4	5	6	•	•
DAA		•	•	•	•	•	2	•
DEC		2	•	•	6	7	•	•
DES		•	•	•	•	•	4	•
DEX		•	•	•	•	•	4	•
EOR	x	•	2	3	4	5	•	•

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Inherent
INC		2	•	•	6	7	•
INS		•	•	•	•	•	4
INX		•	•	•	•	•	4
JMP		•	•	•	3	4	•
JSR		•	•	•	9	8	•
LDA	x	•	2	3	4	5	•
LDS		•	3	4	5	6	•
LDX		•	3	4	5	6	•
LSR		2	•	•	6	7	•
NEG		2	•	•	6	7	•
NOP		•	•	•	•	•	2
ORA	x	•	2	3	4	5	•
PSH		4	•	•	•	•	•
PUL		4	•	•	•	•	•
ROL		2	•	•	6	7	•
ROR		2	•	•	6	7	•
RTI		•	•	•	•	•	10
RTS		•	•	•	•	•	5
SBA		•	•	•	•	•	2
SBC	x	•	2	3	4	5	•
SEC		•	•	•	•	•	2
SEI		•	•	•	•	•	2
SEV		•	•	•	•	•	2
STA	x	•	•	4	5	6	•
STS		•	•	5	6	7	•
STX		•	•	5	6	7	•
SUB	x	•	2	3	4	5	•
SWI		•	•	•	•	•	12
TAB		•	•	•	•	•	2
TAP		•	•	•	•	•	2
TBA		•	•	•	•	•	2
TPA		•	•	•	•	•	2
TST		2	•	•	6	7	•
TSX		•	•	•	•	•	4
TSX		•	•	•	•	•	4
WAI		•	•	•	•	•	9

NOTE: Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.

Figure 3-2 Instruction Addressing Modes and Execution Times (Times in Machine Cycles)

The following rules apply to labels:

1. A label consists of from 1 to 6 alphanumeric characters.
2. The first character of a label must be alphabetic.
3. A label must begin in the first character position of a statement.
4. All labels within a program must be unique.
5. A label must not consist of any one of the single characters A, B, or X. (These characters are reserved for special syntax, and refer to "accumulator A", "accumulator B", and "index register", respectively.)

Labels are used in source programs in the following cases:

1. A label may be included in any statement which is the destination of:
   a. Any of the conditional branch instructions: BCC BCS BEQ BGE BGT BHI BLE BLS BLT BMI BNE BPL BVC BVS
   b. The unconditional branch instruction: BRA, or
   c. The branch to a subroutine: BSR

   Correspondingly, the operand field of the branch instruction would consist only of a single symbol which would be identical with the label at the destination. In the case of the instruction BSR, the symbol in the operand field, identical with the label at the destination, could be regarded as, in effect, the name of the subroutine.

2. A label may be included in any statement which is the destination of either of the instructions:

   JMP (unconditional jump)
   or JSR (jump to subroutine)

   when the instruction is being used in the extended mode of addressing.

   Correspondingly, when used in the extended mode of addressing, the operand field of either of the instructions JMP and JSR would consist only of a symbol which would be identical with the label at the destination. For JSR, this could be regarded as, in effect, the name of the subroutine.

3. A label would be included in an assembler directive which specifies the location in memory corresponding to a symbol. This applies only to the directives:

   FCB FCC FDB RMB

   When used for this purpose, the label in the assembler directive would be identical with the corresponding symbol.

4. A label must be used in a statement which includes the assembler directive EQU. The label will be identical with the symbol which the EQU statement is defining.

5. In other cases, a label may be used in any executable instruction at the option of the programmer. Among the assembler directives EQU must always be written with a label; each of FCB, FCC, FDB, and RMB, may have a label; any other of the assembler directives must not be written with a label.

   (For further details on the assembler directives see Appendix B.)

Operator Field

The mnemonic operators recognized by the assembler include 72 executable instructions. Each instruction is translated by the assembler into one to three bytes of machine code. The remaining mnemonic operators are assembler directives, of which four (FCB, FCC, FDB, and RMB) are translated into one or more bytes of machine code. The other assembler directives control the overall assembly process and are not translated individually into machine code.

A functional classification of the mnemonic operators is shown in Figure 3-3. An alphabetical listing of the executable instructions is given, with brief definitions, in Figure 3-4.

Executable Instructions

Each of the executable instructions recognized in the source language consists of three alphabetic characters. (However, as when the first operand in the operand field is either A or B, the programmer has the option of joining the character A or B to the operator, which results in an apparent four-character format.)

Figure 3-2 gives complete information stating which modes of addressing can be used with the different executable instructions. The table also shows the execution times in clock cycles.

The assembly of an executable instruction results in from one to three bytes of machine code, depending on the addressing mode. This information is summarized in Figure 3-5.

Detailed definitions of the executable instructions are given in Appendix A.

I. Operations on 8-Bit Registers:

- A. Two-operand arithmetic .....ABA ADC ADD SBA SBC SUB
- B. Single-operand arithmetic .....CLR DAA DEC INC NEG
- C. Comparisons and Tests .....CBA CMP TST
- D. Shifts and Rotations .....ASL ASR LSR ROL ROR
- E. Logic Functions .....AND BIT COM EOR ORA
- F. Load and Store .....LDA STA PSH PUL
- G. Transfers .....TAB TBA

II. Jump and Branch Control:

- A. Conditional Branch .....BCC BCS BEQ BGE BGT BHI BLE BLS BLT BMI BNE BPL BVC BVS
- B. Unconditional Branch and Jump .BRA NOP JMP
- C. Control of Subroutines .....BSR JSR RTS
- D. Control of Interrupts .....RTI SWI WAI

III. Control of Index Register and Stack Pointer:

- A. Index Register .....DEX INX LDX STX CPX
- B. Stack Pointer .....DES INS LDS STS
- C. Transfers .....TSX TXS

IV. Control of Condition Codes Register:

- A. Bit Control .....CLC CLI CLV SEC SEI SEV
- B. Byte Transfers .....TAP TPA

V. Assembler Directives .....END EQU FCB FCC FDB MON NAM OPT ORG PAGE RMB SPC

Figure 3-3 Functional Classification of the Mnemonic Operators

ABA	Add Accumulators
ADC	Add with Carry
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
BCC	Branch if Carry Clear
BCS	Branch if Carry Set
BEQ	Branch if Equal to Zero
BGE	Branch if Greater or Equal Zero
BGT	Branch if Greater than Zero
BHI	Branch if Higher
BIT	Bit Test
BLE	Branch if Less or Equal
BLS	Branch if Lower or Same
BLT	Branch if Less than Zero
BMI	Branch if Minus
BNE	Branch if Not Equal to Zero
BPL	Branch if Plus
BRA	Branch Always
BSR	Branch to Subroutine
BVC	Branch if Overflow Clear
BVS	Branch if Overflow Set
CBA	Compare Accumulators
CLC	Clear Carry
CLI	Clear Interrupt Mask
CLR	Clear
CLV	Clear Overflow
CMP	Compare
COM	Complement
CPX	Compare Index Register
DAA	Decimal Adjust
DEC	Decrement
DES	Decrement Stack Pointer
DEX	Decrement Index Register
EOR	Exclusive OR
INC	Increment
INS	Increment Stack Pointer
INX	Increment Index Register
JMP	Jump
JSR	Jump to Subroutine
LDA	Load Accumulator
LDS	Load Stack Pointer
LDX	Load Index Register
LSR	Logical Shift Right
NEG	Negate
NOP	No Operation
ORA	Inclusive OR Accumulator
PSH	Push Data
PUL	Pull Data
ROL	Rotate Left
ROR	Rotate Right
RTI	Return from Interrupt
RTS	Return from Subroutine
SBA	Subtract Accumulators
SBC	Subtract with Carry
SEC	Set Carry
SEI	Set Interrupt Mask
SEV	Set Overflow
STA	Store Accumulator
STS	Store Stack Register
STX	Store Index Register
SUB	Subtract
SWI	Software Interrupt
TAB	Transfer Accumulators
TAP	Transfer Accumulators to Condition Code Reg.
TBA	Transfer Accumulators
TPA	Transfer Condition Code Reg. to Accumulator
TST	Test
TSX	Transfer Stack Pointer to Index Register
TXS	Transfer Index Register to Stack Pointer
WAI	Wait for Interrupt

Figure 3-4 Executable Instructions -- Alphabetic List

ADDRESSING MODE	NUMBER OF BYTES OF MACHINE CODE
Inherent	1
Accumulator (single operand)	1
Relative	2
Direct	2
Indexed	2
Immediate:	2
1. All instructions except CPX, LDS and LDX	2
2. Instructions CPX, LDS and LDX	3
Extended	3

Figure 3-5

Operand Field

The kind of information placed in the operand field depends on the particular mnemonic operator. For the 72 executable instructions the microprocessor uses various modes of addressing for obtaining the operands and saving the results of execution. The addressing mode is determined by the mnemonic operator combined with the information in the operand field. The addressing modes are summarized in Figure 3-2.

The assembler recognizes numbers, symbols and expressions in the operand field. Dual operand instructions require either of the single characters A or B as the first operand.

Numbers

Numbers are accepted by the assembler in the following formats:

Number	(decimal)
$ Number	(hexadecimal)
Number H	(hexadecimal)
@ Number	(octal)
Number O	(octal)
Number Q	(octal)
% Number	(binary)
Number B	(binary)

where Number is a positive integer. A prefix "$", "@", or "%" instructs the assembler to interpret the number as hexadecimal, octal, or binary, respectively. A suffix of "O" or "Q" indicate octal numbers while the suffix "H" indicates hexadecimal, and the suffix "B" indicates binary. When none of these prefixes or suffixes is used, the number is assumed to be decimal.

In the case where the prefix is "$" and the last character is "B" the assembler interprets the number as hexadecimal.

Symbols

Symbols when used in the operand field follow these rules.

1. A symbol must not be any of the single characters A, B, or X.
2. Subject to rule (1), a symbol may consist of from 1 to 6 alphanumeric characters, of which the first is alphabetic.
3. The single character "*" is a symbol which represents the program counter.

The special symbol "*" represents the program counter. Its value is, therefore, equal to the numerical address of the first byte of machine code which results from the assembly of any source instruction which contains "*" in the operand field.

The single characters A, B, and X are reserved for special use in the source program, to represent accumulator A, accumulator B, and the index register. The single characters A or B must be used with dual operand instructions and may be used to indicate accumulator addressing. The single character X is indication of indexed addressing.

All other symbols must be defined in the source program. There are three ways of defining a symbol, as follows:

1. An executive instruction in the source program may include a label indentical with the symbol being defined. The value of the symbol is then the numerical address of the first byte of machine code which results from the executive instruction which includes the label.

2. One of the assembler directives FCB, FCC, FDB, or RMB may be written with a label identical with the symbol being defined. The value of the symbol is then the numerical address of the first byte of machine code which results from the assembler directive (FCB, FCC, FDB, or RMB) which includes the label.

3. The symbol may be defined by using the assembler directive EQU. The mnemonic operator "EQU" is preceded by a label identical with the symbol being defined. The value of the symbol, represented by the label, is that of the operand which follows the mnemonic operator "EQU". The operand may be a number, another symbol,or an expression.

Expressions

An expression is a combination of symbols and/or numbers being separated one from the next by one of the arithmetic operators (+, -, *, or /).

The assembler evaluates expressions algebraically from left to right without parenthetical grouping, there being no heirarchy of precedence among the arithmetic operators. A fractional result, or intermediate result, if obtained during the evaluation of an expression, will be truncated to an integer value. The use of expressions in the source language does not imply any capability of the microprocessor to evaluate those expressions, since the expressions are evaluated during assembly and not during execution of the machine language program.

Evaluation of Symbols and Expressions

The assembler must complete the numerical evaluation of symbols and expressions in two passes through a source program. Reflecting the two-pass characteristic of the assembly process, only one level of forward referencing is permitted in the use of symbols or expressions in the operand field of source statements.

Comments Field

A comment may be included in a source statement at the option of the programmer. The comment, if present, may contain any characters corresponding to ASCII hexadecimal values 20 (SP) through 5F (____). Source statement comments do not affect the machine code which results from the assembly of a program. They are ignored by the assembler except for being included in the program listing.

Comments may be used in source programs for aiding comprehension of the program, and for purposes of checkout and documentation.

ADDRESSING MODES

The assembler scans the operator and operand to determine the proper addressing mode. The addressing modes are:

Inherent Addressing
Relative Addressing
Immediate Addressing
Indexed Addressing
Accumulator Addressing
Extended Addressing
Direct Addressing

Dual Addressing

Eleven of the executable instructions require addressing of two operands in the operand field. These instructions are indicated in Figure 3-2 by the column headed "Dual Operand". For all of these operators the first operand must be either accumulator A or accumulator B. This is specified respectively by A or B as the first character in the operand field, the second character in the operand field being a SPACE (OR TAB).

For dual addressing the specification of the first operand (either A or B) is separated from that of the second operand by one or more SPACE characters (or alternatively by TAB).

The second operand is specified in the operand field in accordance with the rules for immediate, direct, extended, or indexed addressing (as defined subsequently); depending on which modes of addressing are valid for the individual operators.

(For the mnemonic operators which employ dual addressing it is permissible to omit the SPACE between the operator and the operand field.)

Accumulator Addressing (single operand)

Thirteen of the operators address a single operand from the operand field and can so address either accumulator A or accumulator B in the microprocessing unit. These operators are indicated by the column headed "ACCX" in Figure 3-2. This mode of addressing is specified by writing an operand field consisting only of the single character A or B, corresponding to accumulator A or accumulator B. (It is then permissible to omit the SPACE (or TAB) between the operator and the operand field, for this type of addressing.)

For this type of addressing the assembly of a source instruction results in one byte of instruction in the machine language.

(For operators PUL and PSH, the accumulator mode is the only valid mode of addressing. The remaining eleven operators capable of this mode of addressing can alternatively be used with extended or indexed addressing.)

Inherent Addressing

In many cases the mnemonic operator itself specifies one or more registers which contain operands or in which results are saved. For example, the operator ABA requires two operands which are located in accumulator A and accumulator B of the microprocessor. The operator also determines that the result of execution will be saved in accumulator A.

For some executable instructions, all of the information which may be required for the addressing is contained in the mnemonic operator, and no operand field is used in the source statement. There are 25 such instructions. These are indicated by the column headed "inherent" in Figure 3-2.

Assembly of this type of source instruction results in only one byte of machine language code. (Some other operators which contain addressing information inherently in the mnemonic code also require further addressing or operand information which is then placed in an operand field. Examples are the operators CPX, LDS, LDX, STS, and STX.)

Immediate Addressing

The operators with which the immediate mode of addressing is permissible are indicated by the column headed "immediate" in Figure 3-2. This mode of addressing is selected by beginning the specification of the corresponding operand (in the operand field of a source statement) with the pound character "#".

With the immediate mode of addressing, the operand field of the source statement either contains the actual value of the operand, or it includes a symbol or an expression which has an algebraic value equal to the value of the operand. The operand may be specified in accordance with any of the following formats:

```
# Number
# Symbol
# Expression
# 'C
```

In the first three of these alternate forms the assembler will find or compute a numerical value of the operand. For any executive instruction in the immediate mode of addressing except CPX, LDS, or LDX, the numeric value must be an integer from 0 to 255 (decimal). For the operators CPX, LDS, or LDX, any value from 0 to 65535 (decimal) is valid.

In the last of the alternative forms, #'C, the apostrophe instructs the Assembler to translate the next character into the corresponding 7-bit ASCII code. The ASCII code so obtained is then the value of the operand. The single character "C" can be any character of the ASCII character set with hexadecimal value from 20 (SP) through 5F (____).

For the immediate mode of addressing, the assembler inserts the actual value of the operand into the machine code. Except for the three operators CPX, LDS, and LDX an instruction in the immediate mode is assembled into two bytes of machine code, and the value of the operand is entered in the second byte. When it is a number, the operand is entered in the memory in unsigned 8-bit binary code. When it is an ASCII character, the corresponding 7-bit ASCII code applies, using bits 0-6, and bit 7 is set to zero.

For the three operators CPX, LDS, or LDX, used in the immediate mode, the source statement is assembled into three bytes of machine code. The numerical operand, which can have any value from 0 through FFFF, will be entered in the second and third bytes. The second byte will contain the most significant part of the operand, the third byte will contain the least significant part of the operand. Both parts are entered into the respective bytes of the memory in unsigned 8-bit binary code.

The operators CPX, LDS, or LDX, in the immediate mode, are not normally used with an operand in the format "#'C". However, in such a case, the assembler would place the ASCII coded character "C" in the third byte of the machine code corresponding to the source instruction.

When the immediate mode of addressing is used, the numerical address is in effect that of the second byte of machine code which results from assembly of the source instruction. Data flow for the immediate addressing mode is shown in Figure 3-6.

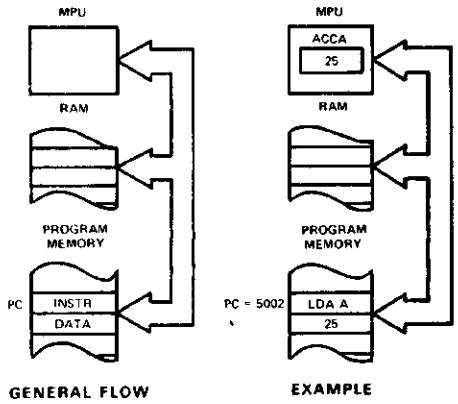


Figure 3-6 Immediate Addressing Mode Data Flow

Relative Addressing

For the relative addressing mode to be valid, there is a rule which limits the distance in the machine language program from the branch instruction to the destination of the branch. The rule which applies to the relative addressing mode is that the address of the destination of the branch must be within the range specified by:

$$(PC + 2) - 128 < D < (PC + 2) + 127$$

where:
PC = address of the first byte of the branch instruction
D = address of the destination of the branch instruction.

When it is desired to transfer control beyond the range of the branch instructions, this can be done by using JMP (unconditional jump) or JSR (jump to subroutine). These instructions do not use the relative mode of addressing.

The assembler translates a branch instruction into two bytes of the machine code. The second byte contains a relative address. This is stored as a number in 8-bit, two's complement, binary form, with decimal value in the range from -128 to +127. These numbers correspond to the limits of the range of a branch instruction, as described above.

The relationship between the relative address and the absolute address of the destination of a branch instruction is expressed by:

$$D = (PC + 2) + R$$

where:

PC = address of first byte of the branch instruction
D = address of the destination of the branch instruction
R = the 8-bit, two's complement, binary number, stored in the second byte of the branch instruction.

The relative addressing mode is available only to the conditional branch instructions, the unconditional branch instruction BRA, and the branch to subroutine BSR. None of these source instructions can use any other mode of addressing. The three-character mnemonic instruction is, therefore, sufficient to determine for the assembler when the relative mode of addressing will be used. An example of the data flow for the relative addressing mode is shown in Figure 3-7.

Indexed Addressing

A column of Figure 3-2 indicates the instructions for which indexed addressing is valid.

With this mode of addressing, the numerical address is variable and dependent on the contents of the index register. The current address is obtained whenever it is required during the execution of a program, rather than being pre-determined by the assembler as it is for the other addressing modes. The operand field of the source statement contains a numerical value which, when added to the contents of the index register during execution of the program, will provide the numerical address. Alternatively the operand field may contain a symbol or an expression which the assembler is able to replace by the value which is to be added to the contents of the index register. An example of the indexed addressing mode is shown in Figure 3-8.

For indexed addressing the data for obtaining the numerical address may be written in any of the formats:

```
X
,X
Number,X
Symbol,X
Expression,X
```

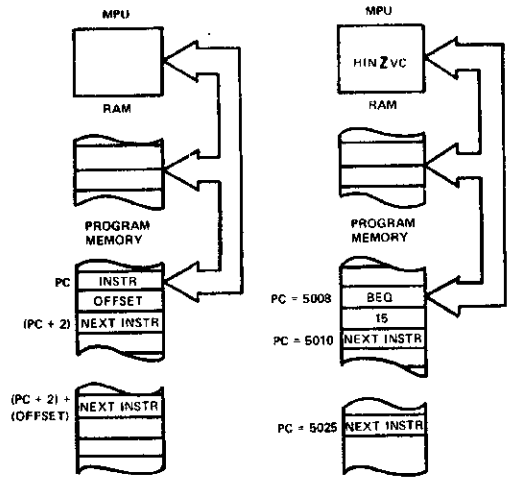


Figure 3-7 Relative Addressing Mode Data Flow

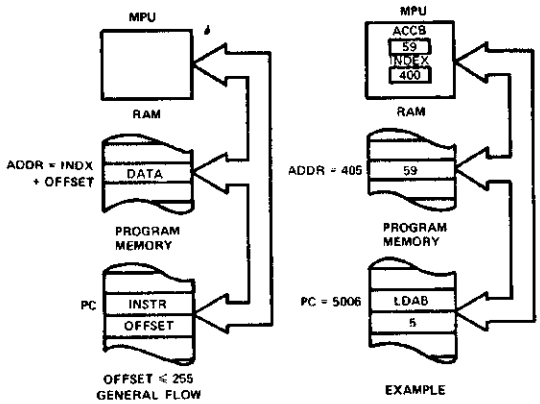


Figure 3-8 Indexed Addressing Mode

The single character "X" informs the assembler that the indexed mode is to be used, the character "X" being reserved to denote the index register.

The format "X", when used alone, instructs the assembler that the address of the operand is identical with the contents of the index register. (This format has the same effect on the assembly as if "0,X" had been written.)

If a symbol or an expression is used rather than a number, the assembler will find or compute a numerical value of that symbol or expression. The source program must then include other statements which define a numerical value for the symbol or which enable the assembler to compute a numerical value for the symbol or expression. Only values from zero to FF (hexadecimal) are valid. This value is added to the contents of the index register during execution to obtain the numerical address as shown in the following formula:

D = numerical value + X

where

X = contents of index register

D = numerical address

For indexed addressing the source instruction is translated into two bytes of the machine code. The second byte contains the number, in unsigned 8-bit binary form, which is added during execution of the instruction to the contents of the index register. The number thus obtained is the numerical address (in accordance with the foregoing formula).

Direct and Extended Addressing

For direct addressing the source instruction is translated into two bytes of machine code. The second byte will contain the address in unsigned 8-bit binary form.

For extended addressing the source instruction is translated into three bytes of machine language. The second of these bytes will contain the highest 8 bits of the address. The third byte will contain the lowest 8 bits of the address. The contents of the second and third bytes will both be coded in unsigned 8-bit binary form.

For both direct and extended addressing, the address, which is placed by the assembler into the second or the second and third bytes of the machine code, is the absolute numerical address.

As may be seen in Figure 3-2, there are several instructions for which the extended mode of addressing is valid but the direct mode is not. For these instructions, when using any of the following formats,

Number
Symbol
Expression

the assembler will select the extended mode of addressing whatever may be the value of the numerical address. The source statement will be translated into three bytes of the machine code.

For those instructions which may use the direct mode of addressing as well as the extended mode, the assembler selects the mode according to the following rule: The assembler will select direct addressing if the numerical address is in the range from zero to 255 (decimal) and will select extended addressing if the numerical address exceeds 255 (decimal). Examples of the direct and extended addressing modes are shown in Figures 3-9 and 3-10.

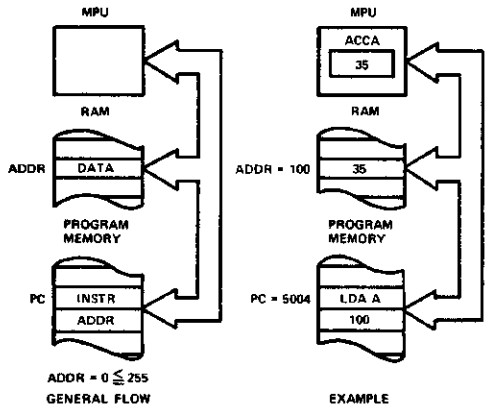


Figure 3-9 Direct Addressing Mode Data Flow

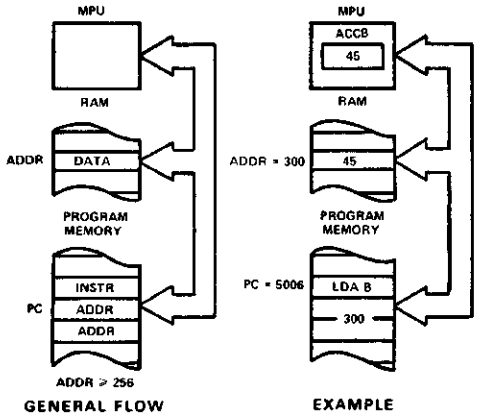


Figure 3-10 Extended Addressing Mode Data Flow

ASSEMBLER DIRECTIVES

The assembler directives allow the programmer control of the assembly of the executive instructions into machine code, including control of the allocation of memory, and assignment of values to data, when applicable. The assembler directives also provide for control of the sequencing of source programs through the assembler, and for control of the format of the assembler output.

A functional classification of the assembler directives is given below:

CODE	SUMMARY DEFINITION	FUNCTION
ORG	Assign origin of program counter	Defines the numerical address of the first byte of a subsequent segment of the coded program.
EQU	Equate a symbol to an operand	Equates a symbol to a numerical value, another symbol, or an expression.
FCB	Form constant byte	Assign values and addresses of data, and assign addresses of scratch areas of memory.
FCC	Form constant characters	
FDB	Form double constant byte	
RMB	Reserve memory bytes	
END	Define end of source program	Control the sequencing of source programs through the assembler.
MON	Return to console	
NAM	Name the program or insert text	Format control (Source program and/or assembler listing)
OPT	Assembler control options	
PAGE	Move paper to top of form	
SPC	Vertical spacing of program listing	

Assembler Directives - Operand Formating

Detailed definitions are shown in Appendix B. The formats of the assembler directives operand field are summarized below:

FCB (2) EQU (1)	FCC	END
FDB (2) ORG (1)	NAM	MON
RMB (1) SPC (1)	OPT	PAGE

Number Symbol Expression	Special Formatting Rules (see details of the assembler directives in Appendix B.)	No Operand (Operand field is left blank or will be treated by the assembler as a comment.)

Notes: (1) Only one operand.
(2) May have more than one operand, separated by commas.

Labels Used with Assembler Directives

A label must be included in any source statement which includes the assembler directive "EQU". The label must be identical with a symbol used elsewhere in the source program. The "EQU" directive is used to define the symbol, directly or indirectly.

The significance of the label, in this case depends on that of the symbol with which it is identical. It can represent a numerical address, or data, or neither of these. In the latter case the label, and the corresponding symbol, would represent an algebraic quantity which appears in one or more expressions in the source program.

A label may be included in any source statement which includes any of the assembler directives FCB, FCC, FDB, or RMB. These are the only assembler directives which are translated individually into one or more bytes of machine code. The label, if used, represents the address of the first byte of the machine code which results from the respective source statement.

Any source statement which includes any assembler directive other than EQU, FCB, FCC, FDB or RMB, must not be written with a label.

Comments Used with Assembler Directives

The assembler directive "NAM" does not distinguish between the operand field and a comment. Both are treated by the assembler as continuous text.

A comment may be used with any other assembler directive at the option of the programmer; however, comments with the SPC or PAGE assembler directives will not be printed (these two directives do not print).

IV INSTRUCTION SET

The M6800 instructions are each described in detail in Appendix A. This section will provide a brief introduction to the instructions and discuss their use in writing 680b programs.

The instruction set is shown in summary form in Figure 4-1. Each of the 72 executable instructions of the source language assembles into from 1 to 3 bytes of machine code. The number of bytes depends on the particular instruction and on the addressing mode. The addressing modes which are available for use with the various executive instructions are indicated in Figure 3-2.

The coding of the first (or only) byte, corresponding to an executable instruction, is sufficient to identify the instruction and the addressing mode. The hexadecimal equivalents of the binary codes, which result from the translation of the 72 instructions, in all valid modes of addressing, are shown in Figure 4-2. There are 197 valid machine codes, 59 of the 256 possible codes being unassigned. The octal and decimal equivalents of the machine language codes are shown similarly, in Figures 4-3 and 4-4.

Microprocessor instructions are often divided into three general classifications: (1) memory reference, so called because they operate on specific memory locations; (2) operating instructions that function without needing a memory reference; (3) I/O instructions for transferring data between the microprocessor and peripheral devices.

In many instances, the 6800 performs the same operation on both its internal accumulators and the external memory locations. In addition, the M6800 treats peripheral devices exactly like memory locations, hence, no I/O instructions as such are required. Because of these features, other classifications are more suitable for introducing the 6800 instruction set: (1) Accumulator and memory operations; (2) Program control operations; (3) Condition Code Register operations.

CONDITION CODE REGISTER OPERATIONS

The Condition Code Register (CCR), also called the Program Status Byte, will be described first since it is affected by many of the other instructions as well as the specific operations shown in Figure 4-6. The CCR is a 6-bit register within the MPU that is useful in controlling program flow during system operation. The bits are defined in Figure 4-5.

The instructions shown in Figure 4-6 are available to the user for direct manipulation of the CCR. In addition, the MPU automatically sets or clears the appropriate status bits as many of the other instructions are executed. The effect of those instructions on the condition code register will be indicated as they are introduced and is also included in the Instruction Set Summary of Figure 4-1.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	5 H	4 I	3 N	2 Z	1 V	0 C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	•	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	•	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	•	•	↕	↕	R	•
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	•	•	↕	↕	R	•
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	•	•	↕	↕	R	•
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	•	•	↕	↕	R	•
Clear	CLR							6F	7	2	7F	6	3				00 → M	•	•	R	S	R	R
	CLRA													4F	2	1	00 → A	•	•	R	S	R	R
	CLRB													5F	2	1	00 → B	•	•	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	•	•	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	•	•	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	•	•	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				M̄ → M	•	•	↕	↕	R	S
	COMA													43	2	1	Ā → A	•	•	↕	↕	R	S
	COMB													53	2	1	B̄ → B	•	•	↕	↕	R	S
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 − M → M	•	•	↕	↕	①	②
	NEGA													40	2	1	00 − A → A	•	•	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	•	•	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	•	•	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	•	•	↕	↕	④	•
	DECA													4A	2	1	A − 1 → A	•	•	↕	↕	④	•
	DECB													5A	2	1	B − 1 → B	•	•	↕	↕	④	•
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	•	•	↕	↕	R	•
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	•	•	↕	↕	R	•
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	•	•	↕	↕	⑤	•
	INCA													4C	2	1	A + 1 → A	•	•	↕	↕	⑤	•
	INCB													5C	2	1	B + 1 → B	•	•	↕	↕	⑤	•
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	•	•	↕	↕	R	•
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	•	•	↕	↕	R	•
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	•	•	↕	↕	R	•
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	•	•	↕	↕	R	•
Push Data	PSHA													36	4	1	A → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
	PSHB													37	4	1	B → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
Pull Data	PULA													32	4	1	SP + 1 → SP, M_{SP} → A	•	•	•	•	•	•
	PULB													33	4	1	SP + 1 → SP, M_{SP} → B	•	•	•	•	•	•
Rotate Left	ROL							69	7	2	79	6	3				M	•	•	↕	↕	⑤	↕
	ROLA													49	2	1	A	•	•	↕	↕	⑤	↕
	ROLB													59	2	1	B	•	•	↕	↕	⑤	↕
Rotate Right	ROR							66	7	2	76	6	3				M	•	•	↕	↕	⑤	↕
	RORA													46	2	1	A	•	•	↕	↕	⑤	↕
	RORB													56	2	1	B	•	•	↕	↕	⑤	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M	•	•	↕	↕	⑤	↕
	ASLA													48	2	1	A	•	•	↕	↕	⑤	↕
	ASLB													58	2	1	B	•	•	↕	↕	⑤	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M	•	•	↕	↕	⑤	↕
	ASRA													47	2	1	A	•	•	↕	↕	⑤	↕
	ASRB													57	2	1	B	•	•	↕	↕	⑤	↕
Shift Right, Logic	LSR							64	7	2	74	6	3				M	•	•	R	↕	⑤	↕
	LSRA													44	2	1	A	•	•	R	↕	⑤	↕
	LSRB													54	2	1	B	•	•	R	↕	⑤	↕
Store Acmltr.	STAA				97	4	2	A7	6	2	B7	5	3				A → M	•	•	↕	↕	R	•
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	•	•	↕	↕	R	•
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	•	•	↕	↕	↕	↕
Subract Acmltrs.	SBA													10	2	1	A − B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	•	•	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	•	•	↕	↕	R	•
	TBA													17	2	1	B → A	•	•	↕	↕	R	•
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	•	•	↕	↕	R	R
	TSTA													4D	2	1	A − 00	•	•	↕	↕	R	R
	TSTB													5D	2	1	B − 00	•	•	↕	↕	R	R

Figure 4-1 M6800 Instruction Set

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ARITHMETIC OPERATION	5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$(X_H/X_L) - (M/M+1)$	•	•	⑦	↕	⑤	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	↕	•	•
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	↕	•	•
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M+1) \rightarrow X_L$	•	•	⑨	↕	R	•
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	•	•	⑨	↕	R	•
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M+1)$	•	•	⑨	↕	R	•
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	•	•	⑨	↕	R	•
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

JUMP AND BRANCH OPERATIONS	MNEMONIC	RELATIVE			INDEX			EXTND			INHER			BRANCH TEST	5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	•	•	•	•	•	•
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	•	•	•	•	•	•
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•	•
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	•	•	•	•	•	•
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•	•
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	•	•	•	•	•	•
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	•	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI										3B	10	1		———			⑩	———	
Return From Subroutine	RTS										39	5	1	See special Operations	•	•	•	•	•	•
Software Interrupt	SWI										3F	12	1		•	S	•	•	•	•
Wait for Interrupt	WAI										3E	9	1		•	⑪	•	•	•	•

CONDITIONS CODE REGISTER OPERATIONS	MNEMONIC	INHER			BOOLEAN OPERATION	5	4	3	2	1	0
		OP	~	#		H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	•	•	•	•	S	•
Acmltr A → CCR	TAP	06	2	1	A → CCR	———		⑫		———	
CCR → Acmltr A	TPA	07	2	1	CCR → A	•	•	•	•	•	•

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
#	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
-	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer;	C	Carry from bit 7
		R	Reset Always
+	Boolean Inclusive OR;	S	Set Always
⊕	Boolean Exclusive OR;	↕	Test and set if true, cleared otherwise
$\overline{M}$	Complement of M;	•	Not Affected
→	Transfer Into;	CCR	Condition Code Register
0	Bit = Zero;	LS	Least Significant
		MS	Most Significant

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?

② (Bit C) Test: Result = 00000000?

③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

④ (Bit V) Test: Operand = 10000000 prior to execution?

⑤ (Bit V) Test: Operand = 01111111 prior to execution?

⑥ (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred

⑦ (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?

⑧ (Bit V) Test: 2's complement overflow from subtraction of LS bytes?

⑨ (Bit N) Test: Result less than zero? (Bit 15 = 1)

⑩ (All) Load Condition Code Register from Stack. (See Special Operations)

⑪ (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

⑫ (ALL) Set according to the contents of Accumulator A.

00	*		40	NEG	A		80	SUB	A	IMM	C0	SUB	B	IMM
01	NOP		41	*			81	CMP	A	IMM	C1	CMP	B	IMM
02	*		42	*			82	SBC	A	IMM	C2	SBC	B	IMM
03	*		43	COM	A		83	*			C3	*		
04	*		44	LSR	A		84	AND	A	IMM	C4	AND	B	IMM
05	*		45	*			85	BIT	A	IMM	C5	BIT	B	IMM
06	TAP		46	ROR	A		86	LDA	A	IMM	C6	LDA	B	IMM
07	TPA		47	ASR	A		88	*			C7	*		
08	INX		48	ASL	A		88	EOR	A	IMM	C8	EOR	B	IMM
09	DEX		49	ROL	A		89	ADC	A	IMM	C9	ADC	B	IMM
0A	CLV		4A	DEC	A		8A	ORA	A	IMM	CA	ORA	B	IMM
0B	SEV		4B	*			8B	ADD	A	IMM	CB	ADD	B	IMM
0C	CLC		4C	INC	A		8C	CPX		IMM	CC	*		
0D	SEC		4D	TST	A		8D	BSR		REL	CD	*		
0E	CLI		4E	*			8E	LDS		IMM	CE	LDX		IMM
0F	SEI		4F	CLR	A		8F	*			CF	*		
10	SBA		50	NEG	B		90	SUB	A	DIR	D0	SUB	B	DIR
11	CBA		52	*			91	CMP	A	DIR	D1	CMP	B	DIR
12	*		52	*			92	SBC	A	DIR	D2	SBC	B	DIR
13	*		53	COM	B		93	*			D3	*		
14	*		54	LSR	B		94	AND	A	DIR	D4	AND	B	DIR
15	*		55	*			95	BIT	A	DIR	D5	BIT	B	DIR
16	TAB		56	ROR	B		96	LDA	A	DIR	D6	LDA	B	DIR
17	TBA		57	ASR	B		97	STA	A	DIR	D7	STA	B	DIR
18	*		58	ASL	B		98	EOR	A	DIR	D8	EOR	B	DIR
19	DAA		59	ROL	B		99	ADC	A	DIR	D9	ADC	B	DIR
1A	*		5A	DEC	B		9A	ORA	A	DIR	DA	ORA	B	DIR
1B	ABA		5B	*			9B	ADD	A	DIR	DB	ADD	B	DIR
1C	*		5C	INC	B		9C	CPX		DIR	DC	*		
1D	*		5D	TST	B		9D	*			DD	*		
1E	*		5E	*			9E	LDS		DIR	DE	LDX		DIR
1F	*		5F	CLR	B		9F	STS		DIR	DF	STX		DIR
20	BRA	REL	60	NEG	IND		A0	SUB	A	IND	E0	SUB	B	IND
21	*		61	*			A1	CMP	A	IND	E1	CMP	B	IND
22	BHI	REL	62	*			A2	SBC	A	IND	E2	SBC	B	IND
23	BLS	REL	63	COM	IND		A3	*			E3	*		
24	BCC	REL	64	LSR	IND		A4	AND	A	IND	E4	AND	B	IND
25	BCS	REL	65	*			A5	BIT	A	IND	E5	BIT	B	IND
26	BNE	REL	66	ROR	IND		A6	LDA	A	IND	E6	LDA	B	IND
27	BEQ	REL	67	ASR	IND		A7	STA	A	IND	E7	STA	B	IND
28	BVC	REL	68	ASL	IND		A8	EOR	A	IND	E8	EOR	B	IND
29	BVS	REL	69	ROL	IND		A9	ADC	A	IND	E9	ADC	B	IND
2A	BPL	REL	6A	DEC	IND		AA	ORA	A	IND	EA	ORA	B	IND
2B	BMI	REL	6B	*			AB	ADD	A	IND	EB	ADD	B	IND
2C	BGE	REL	6C	INC	IND		AC	CPX		IND	EC	*		
2D	BLT	REL	6D	TST	IND		AD	JSR		IND	ED	*		
2E	BGT	REL	6E	JMP	IND		AE	LDS		IND	EE	LDX		IND
2F	BLE	REL	6F	CLR	IND		AF	STS		IND	EF	STX		IND
30	TSX		70	NEG	EXT		B0	SUB	A	EXT	F0	SUB	B	EXT
31	INS		71	*			B1	CMP	A	EXT	F1	CMP	B	EXT
32	PUL	A	72	*			B2	SBC	A	EXT	F2	SBC	B	EXT
33	PUL	B	73	COM	EXT		B3	*			F3	*		
34	DES		74	LSR	EXT		B4	AND	A	EXT	F4	AND	B	EXT
35	TXS		75	*			B5	BIT	A	EXT	F5	BIT	B	EXT
36	PSH	A	76	ROR	EXT		B6	LDA	A	EXT	F6	LDA	B	EXT
37	PSH	B	77	ASR	EXT		B7	STA	A	EXT	F7	STA	B	EXT
38	*		78	ASL	EXT		B8	EOR	A	EXT	F8	ADC	B	EXT
39	RTS		79	ROL	EXT		B9	ADC	A	EXT	F9	ADC	B	EXT
3A	*		7A	DEC	EXT		BA	ORA	A	EXT	FA	ORA	B	EXT
3B	RTI		7B	*			BB	ADD	A	EXT	FB	ADD	B	EXT
3C	*		7C	INC	EXT		BC	CPX		EXT	FC	*		
3D	*		7D	TST	EXT		BD	JSR		EXT	FD	*		
3E	WAI		7E	JMP	EXT		BE	LDS		EXT	FE	LDX		EXT
3F	SWI		7F	CLR	EXT		BF	STS		EXT	FF	STX		EXT

Notes: 1. Addressing Modes: A = Accumulator A; B = Accumulator B; IMM = Immediate; DIR = Direct; EXT = Extended; REL = Relative; IND = Indexed

2. Unassigned code indicated by "*".

Figure 4-2 Hexadecimal Values of Machine Codes

000	*		100	NEG	A		200	SUB	A	IMM	300	SUB	B	IMM
001	NOP		101	*			201	CMP	A	IMM	301	CMP	B	IMM
002	*		102	*			202	SBC	A	IMM	302	SBC	B	IMM
003	*		103	COM	A		203	*			303	*		
004	*		104	LSR	A		204	AND	A	IMM	304	AND	B	IMM
005	*		105	*			205	BIT	A	IMM	305	BIT	B	IMM
006	TAP		106	ROR	A		206	LDA	A	IMM	306	LDA	B	IMM
007	TPA		107	ASR	A		207	*			307	*		
010	INX		110	ASL	A		210	EOR	A	IMM	310	EOR	B	IMM
011	DEX		111	ROL	A		211	ADC	A	IMM	311	ADC	B	IMM
012	CLV		112	DEC	A		212	ORA	A	IMM	312	ORA	B	IMM
013	SEV		113	*			213	ADD	A	IMM	313	ADD	B	IMM
014	CLC		114	INC	A		214	CPX		IMM	314	*		
015	SEC		115	TST	A		215	BSR		REL	315	*		
016	CLI		116	*			216	LDS		IMM	316	LDX		IMM
017	SEI		117	CLR	A		217	*			317	*		
020	SBA		120	NEG	B		220	SUB	A	DIR	320	SUB	B	DIR
021	CBA		121	*			221	CMP	A	DIR	321	CMP	B	DIR
022	*		122	*			222	SBC	A	DIR	322	SBC	B	DIR
023	*		123	COM	B		223	*			323	*		
024	*		124	LSR	B		224	AND	A	DIR	324	AND	B	DIR
025	*		125	*			225	BIT	A	DIR	325	BIT	B	DIR
026	TAB		126	ROR	B		226	LDA	A	DIR	326	LDA	B	DIR
027	TBA		127	ASR	B		227	STA	A	DIR	327	STA	B	DIR
030	*		130	ASL	B		230	EOR	A	DIR	330	EOR	B	DIR
031	DAA		131	ROL	B		231	ADC	A	DIR	331	ADC	B	DIR
032	*		132	DEC	B		232	ORA	A	DIR	332	ORA	B	DIR
033	ABA		133	*			233	ADD	A	DIR	333	ADD	B	DIR
034	*		134	INC	B		234	CPX		DIR	334	*		
035	*		135	TST	B		235	*			335	*		
036	*		136	*			236	LDS		DIR	336	LDX		DIR
037	*		137	CLR	B		237	STS		DIR	337	STX		DIR
040	BRA	REL	140	NEG	IND		240	SUB	A	IND	340	SUB	B	IND
041	*		141	*			241	CMP	A	IND	341	CMP	B	IND
042	BHI	REL	142	*			242	SBC	A	IND	342	SBC	B	IND
043	BLS	REL	143	COM	IND		243	*			343	*		
044	BCC	REL	144	LSR	IND		244	AND	A	IND	344	AND	B	IND
045	BCS	REL	145	*			245	BIT	A	IND	345	BIT	B	IND
046	BNE	REL	146	ROR	IND		246	LDA	A	IND	346	LDA	B	IND
047	BEQ	REL	147	ASR	IND		247	STA	A	IND	347	STA	B	IND
050	BVC	REL	150	ASL	IND		250	EOR	A	IND	350	EOR	B	IND
051	BVS	REL	151	ROL	IND		251	ADC	A	IND	351	ADC	B	IND
052	BPL	REL	152	DEC	IND		252	ORA	A	IND	352	ORA	B	IND
053	BMI	REL	153	*			253	ADD	A	IND	353	ADD	B	IND
054	BGE	REL	154	INC	IND		254	CPX		IND	354	*		
055	BLT	REL	155	TST	IND		255	JSR		IND	355	*		
056	BGT	REL	156	JMP	IND		256	LDS		IND	356	LEX		IND
057	BLE	REL	157	CLR	IND		257	STS		IND	357	STX		IND
060	TSX		160	NEG	EXT		260	SUB	A	EXT	360	SUB	B	EXT
061	INS		161	*			261	CMP	A	EXT	361	CMP	B	EXT
062	PUL	A	162	*			262	SBC	A	EXT	362	SBC	B	EXT
063	PUL	B	163	COM	EXT		263	*			363	*		
064	DES		164	LSR	EXT		264	AND	A	EXT	364	AND	B	EXT
065	TXS		165	*			265	BIT	A	EXT	365	BIT	B	EXT
066	PSH	A	166	ROR	EXT		266	LDA	A	EXT	366	LDA	B	EXT
067	PSH	B	167	ASR	EXT		267	STA	A	EXT	367	STA	B	EXT
070	*		170	ASL	EXT		270	EOR	A	EXT	370	EOR	B	EXT
071	RTS		171	ROL	EXT		271	ADC	A	EXT	371	ADC	B	EXT
072	*		172	DEC	EXT		272	ORA	A	EXT	372	ORA	B	EXT
073	RTI		173	*			273	ADD	A	EXT	373	ADD	B	EXT
074	*		174	INC	EXT		274	CPX		EXT	374	*		
075	*		175	TST	EXT		275	JSR		EXT	375	*		
076	WAI		176	JMP	EXT		276	LDS		EXT	376	LDX		EXT
077	SWI		177	CLR	EXT		277	STS		EXT	377	STX		EXT

Notes: 1. Addressing Modes: A = Accumulator A, B = Accumulator B, IMM = Immediate, DIR = Direct, EXT = Extended, REL = Relative, IND = Indexed

2. Unassigned code indicated by "*".

Figure 4-3 Octal Values of Machine Codes

000	*		064	NEG	A	128	SUB	A	IMM	192	SUB	B	IMM
001	NOP		065	*		129	CMP	A	IMM	193	CMP	B	IMM
002	*		066	*		130	SBC	A	IMM	194	SBC	B	IMM
003	*		067	COM	A	131	*			195	*		
004	*		068	LSR	A	132	AND	A	IMM	196	AND	B	IMM
005	*		069	*		133	BIT	A	IMM	197	BIT	B	IMM
006	TAP		070	ROR	A	134	LDA	A	IMM	198	LDA	B	IMM
007	TPA		071	ASR	A	135	*			199	*		
008	INX		072	ASL	A	136	EOR	A	IMM	200	EOR	B	IMM
009	DEX		073	ROL	A	137	ADC	A	IMM	301	ADC	B	IMM
010	CLV		074	DEC	A	138	ORA	A	IMM	202	ORA	B	IMM
011	SEV		075	*		139	ADD	A	IMM	203	ADD	B	IMM
012	CLC		076	INC	A	140	CPX		IMM	204	*		
013	SEC		077	TST	A	141	BSR		REL	205	*		
014	CLI		078	*		142	LDS		IMM	206	LDX		IMM
015	SEI		079	CLR	A	143	*			207	*		
016	SBA		080	NEG	B	144	SUB	A	DIR	208	SUB	B	DIR
017	CBA		081	*		145	CMP	A	DIR	209	CMP	B	DIR
018	*		082	*		146	SBC	A	DIR	210	SBC	B	DIR
019	*		083	COM	B	147	*			211	*		
020	*		084	LSR	B	148	AND	A	DIR	212	AND	B	DIR
021	*		085	*		149	BIT	A	DIR	213	BIT	B	DIR
022	TAB		086	ROR	B	150	LDA	A	DIR	214	LDA	B	DIR
023	TBA		087	ASR	B	151	STA	A	DIR	215	STA	B	DIR
024	*		088	ASL	B	152	EOR	A	DIR	216	EOR	B	DIR
025	DAA		089	ROL	B	153	ADC	A	DIR	217	ADC	B	DIR
026	*		090	DEC	B	154	ORA	A	DIR	218	ORA	B	DIR
027	ABA		091	*		155	ADD	A	DIR	219	ADD	B	DIR
028	*		092	INC	B	156	CPX		DIR	220	*		
029	*		093	TST	B	157	*			221	*		
030	*		094	*		158	LDS		DIR	222	LDX		DIR
031	*		095	CLR	B	159	STS		DIR	223	STX		DIR
032	BRA	REL	096	NEG	IND	160	SUB	A	IND	224	SUB	B	IND
033	*		097	*		161	CMP	A	IND	225	CMP	B	IND
034	BHI	REL	098	*		162	SBC	A	IND	226	SBC	B	IND
035	BLS	REL	099	COM	IND	163	*			227	*		
036	BCC	REL	100	LSR	IND	164	AND	A	IND	228	AND	B	IND
037	BCS	REL	101	*		165	BIT	A	IND	229	BIT	B	IND
038	BNE	REL	102	ROR	IND	166	LDA	A	IND	230	LDA	B	IND
039	REQ	REL	103	ASR	IND	167	STA	A	IND	231	STA	B	IND
040	BVC	REL	104	ASL	IND	168	EOR	A	IND	232	EOR	B	IND
041	BVS	REL	105	ROL	IND	169	ADC	A	IND	233	ADC	B	IND
042	BPL	REL	106	DEC	IND	170	ORA	A	IND	234	ORA	B	IND
043	BMI	REL	107	*		171	ADD	A	IND	235	ADD	B	IND
044	BGE	REL	108	INC	IND	172	CPX		IND	236	*		
045	BLT	REL	109	TST	IND	173	JSR		IND	237	*		
046	BGT	REL	110	JMP	IND	174	LDS		IND	238	LDX		IND
047	BLE	REL	111	CLR	IND	175	STS		IND	239	STX		IND
048	TSX		112	NEG	EXT	176	SUB	A	EXT	240	SUB	B	EXT
049	INS		113	*		177	CMP	A	EXT	241	CMP	B	EXT
050	PUL	A	114	*		178	SBC	A	EXT	242	SBC	B	EXT
051	PUL	B	115	COM	EXT	179	*			243	*		
052	DES		115	LSR	EXT	180	AND	A	EXT	244	AND	B	EXT
053	TXS		117	*		181	BIT	A	EXT	245	BIT	B	EXT
054	PSH	A	118	ROR	EXT	182	LDA	A	EXT	246	LDA	B	EXT
055	PSH	B	119	ASR	EXT	183	STA	A	EXT	247	STA	B	EXT
056	*		120	ASL	EXT	184	EOR	A	EXT	248	EOR	B	EXT
057	RTS		121	ROL	EXT	185	ADC	A	EXT	249	ADC	B	EXT
058	*		122	DEC	EXT	186	ORA	A	EXT	250	ORA	B	EXT
059	RTI		123	*		187	ADD	A	EXT	251	ADD	B	EXT
060	*		124	INC	EXT	188	CPX		EXT	252	*		
061	*		125	TST	EXT	189	JSR		EXT	253	*		
062	WAI		126	JMP	EXT	190	LDS		EXT	254	LDX		EXT
063	SWI		127	CLR	EXT	191	STS		EXT	255	STX		EXT

Notes: 1. Addressing Modes: A = Accumulator A; B = Accumulator B; IMM = Immediate; DIR = Direct; EXT = Extended; REL = Relative; IND = Indexed

2. Unassigned code indicated by "*".

Figure 4-4 Decimal Values of Machine Codes

b_5	b_4	b_3	b_2	b_1	b_0
H	I	N	Z	V	C

H = Half-carry; set whenever a carry from b_3 to b_4 of the result is generated by ADD, ABA, ADC; cleared if no b_3 to b_4 carry; not affected by other instructions.

I = Interrupt Mask; set by hardware or software interrupt of SEI instruction; cleared by CLI instruction. (Normally not used in arithmetic operations.) Restored to a zero as a result of an RTI instruction if 1_m stored on the stack is low.

N = Negative; set if high order bit (b_7) of result is set; cleared otherwise.

Z = Zero; set if result = 0; cleared otherwise.

V = oVerflow; set if there was arithmetic overflow as a result of the operation; cleared otherwise.

C = Carry; set if there was a carry from the most significant bit (b_7) of the result; cleared otherwise.

Figure 4-5 Condition Code Register

CONDITIONS CODE REGISTER			5	4	3	2	1	0
OPERATIONS	MNEMONIC	BOOLEAN OPERATION	H	I	N	Z	V	C
Clear Carry	CLC	0→C	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0→I	•	R	•	•	•	•
Clear Overflow	CLV	0→V	•	•	•	•	R	•
Set Carry	SEC	1→C	•	•	•	•	•	S
Set Interrupt Mask	SEI	1→I	•	S	•	•	•	•
Set Overflow	SEV	1→V	•	•	•	•	S	•
Acmltr A→CCR	TAP	A→CCR	———			①		———
CCR→Acmltr A	TPA	CCR→A	•	•	•	•	•	•

R = Reset
S = Set
• = Not affected
① (ALL) Set according to the contents of Accumulator A.

Figure 4-6 Condition Code Register Instructions

NUMBER SYSTEMS

Effective use of many of the instructions depends on the interpretation given to numerical data, that is, what number system is being used? For example, the ALU always performs standard binary addition of two eight bit numbers using the 2's complement number system to represent both positive and negative numbers. However, the MPU instruction set and hardware flags permit arithmetic operation using any of four different representations for the numbers:

(1) Each byte can be interpreted as a signed 2's complement number in the range -128 to +127:

2^6 2^5 2^4 2^3 2^2 2^1 2^0

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

1 0 0 0 0 0 0 0 (-128 in 2's complement)

1 1 1 1 1 1 1 1 (-1 in 2's complement)

0 0 0 0 0 0 0 0 (0 in 2's complement)

0 0 0 0 0 0 0 1 (+1 in 2's complement)

0 1 1 1 1 1 1 1 (+127 in 2's complement)

(2) Each byte can be interpreted as a signed binary number in the range -127 to +127:

2^6 2^5 2^4 2^3 2^2 2^1 2^0

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

1 1 1 1 1 1 1 1 (-127 in signed binary)

1 0 0 0 0 0 0 1 (-1 in signed binary)

0 0 0 0 0 0 0 0 (0 in signed binary)

0 0 0 0 0 0 0 1 (+1 in signed binary)

0 1 1 1 1 1 1 1 (+127 in signed binary)

(3) Each byte can be interpreted as an unsigned binary number in the range 0 to 255:

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

0 0 0 0 0 0 0 0 (0 in unsigned binary)

1 1 1 1 1 1 1 1 (255 in unsigned binary)

(4) Each byte can be thought of as containing two 4-bit binary coded decimal (BCD) numbers. With this interpretation, each byte can represent numbers in the range 0 to 99:

$2^3\ 2^2\ 2^1\ 2^0\ 2^3\ 2^2\ 2^1\ 2^0$

$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$

0 0 0 0 0 0 0 0 (BCD 0)

0 0 1 0 0 1 1 1 (BCD 27)

1 0 0 1 1 0 0 1 (BCD 99)

The two's complement representation for positive numbers is obtained simply by adding a zero (sign bit) as the next higher significant bit position:

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$

1 1 1 1 1 1 1 (binary 127)

0 1 1 1 1 1 1 1 (+127 in 2's complement representation)

0 0 0 0 0 0 0 1 (binary 1)

0 0 0 0 0 0 0 1 (+1 in 2's complement representation)

When the negative of a number is required for an arithmetic operation, it is formed by first complementing each bit position of the positive representation and then adding one.

64 32 16 8 4 2 1

$a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$

0 1 1 1 1 1 1 1 (+127 in 2's complement representation)

1 0 0 0 0 0 0 0 (1's complement)

1 (add one)

1 0 0 0 0 0 0 1 (-127 in 2's complement representation)

0 0 0 0 0 0 0 0 (0 in 2's complement representation)

1 1 1 1 1 1 1 1 (1's complement)

1 (add one)

0 0 0 0 0 0 0 0 ("0" is same in either notation)

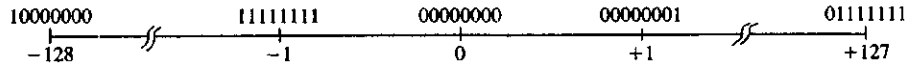
0 0 0 0 0 0 0 1 (+1 in 2's complement representation)

1 1 1 1 1 1 1 0 (1's complement)

1 (add one)

1 1 1 1 1 1 1 1 (-1 in 2's complement representation)

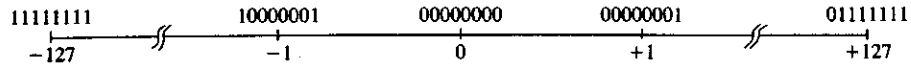
Note that while +127 is the largest positive two's complement number that can be formed with 8 digits, the largest negative two's complement number is 10000000 or -128. Hence, with this number system, an eight bit byte can represent integers on the real number line between -128 and +127 and a_7 can be regarded as a sign bit; if a_7 is zero the number is positive, if a_7 is one the number is negative:



Since much of the literature on arithmetic operations presents the information in terms of signed binary numbers, the difference between 2's complement and signed binary notation is of interest. Signed binary number notation also uses the most significant digit as a sign bit (0 for positive, 1 for negative). The remaining bits represent the magnitude as a binary number.

±	64	32	16	8	4	2	1	
a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
1	1	1	1	1	1	1	1	(-127 in signed binary)
1	0	0	0	0	0	0	1	(-1 in signed binary)
0	0	0	0	0	0	0	0	(0 in signed binary)
0	0	0	0	0	0	0	1	(+1 in signed binary)
0	1	1	1	1	1	1	1	(+127 in signed binary)

An 8-bit byte in this notation represents integers on the real number line between -127 and +127:



Comparing this to the 2's complement representation, the positive numbers are identical and the negative numbers are reversed, i. e., -127 in 2's complement is -1 in signed binary and vice versa. In normal programming of the MPU, the difference causes no particular problem since numerical data is automatically converted to the correct format during assembly of the program source statements. However, if during system operation, incoming data is in signed binary format, the program should provide for conversion. This is easily done by first complementing each bit of the signed binary number except the sign bit and then adding one:

$\pm$	64	32	16	8	4	2	1	
a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
1	1	1	1	1	1	1	1	(-127 in signed binary)
1	0	0	0	0	0	0	0	(1's complement except for sign bit)
0	0	0	0	0	0	0	1	(add 1)
1	0	0	0	0	0	0	1	(-127 in 2's complement)

The MPU instruction set provides for a simple conversion routine. For example, the following program steps can be used:

This routine assumes that the signed binary data is stored in accumulator A (ACCA). The program tests the sign bit and, if the number is negative (N=1), performs the required conversion. The contents of ACCA and the N bit of the Condition Code Register would be as follows after each step of a typical conversion:

Instr	N	a	a	a	a	a	a	a	a	
TSTA	1	1	1	1	1	0	0	0	1	(-113 is signed binary)
BPL NEXT	1	1	1	1	1	0	0	0	1	
NEGA	0	0	0	0	0	1	1	1	1	(2's complement of ACCA)
ORAA #%10000000	1	0	0	0	0	1	1	1	1	(-113 in 2's complement)

Note that the sign bit status, N, is updated as the NEG and ORA instructions are executed. This is typical for many of the instructions; the Condition Code Register is automatically updated as the instruction is executed.

ACCUMULATOR AND MEMORY OPERATIONS

For familiarization purposes, the Accumulator and Memory operations can be further subdivided into four categories: (1) Arithmetic Operations; (2) Logic Operations; (3) Data Testing; and (4) Data Handling.

Arithmetic Operations

The Arithmetic Instructions and their effect on the CCR are shown in Figure 4-7. The use of these instructions in performing arithmetic operations is discussed in section V of this manual.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG. 5	4	3	2	1	0
			H	I	N	Z	V	C
Add	ADDA	A + M → A	↕	•	↕	↕	↕	↕
	ADDB	B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	B + M + C → B	↕	•	↕	↕	↕	↕
Complement, 2's (Negate)	NEG	00 − M → M	•	•	↕	↕	①	②
	NEGA	00 − A → A	•	•	↕	↕	①	②
	NEGB	00 − B → B	•	•	↕	↕	①	②
Decimal Adjust, A	DAA	Converts Binary Add. of BCD Characters into BCD Format*	•	•	↕	↕	↕	③
Subtract	SUBA	A − M → A	•	•	↕	↕	↕	↕
	SUBB	B − M → B	•	•	↕	↕	↕	↕
Subract Acmltrs.	SBA	A − B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	A − M − C → A	•	•	↕	↕	↕	↕
	SBCB	B − M − C → B	•	•	↕	↕	↕	↕

*Used after ABA, ADC, and ADD in BCD arithmetic operation; each 8-bit byte regarded as containing two 4-bit BCD numbers. DAA adds 0110 to lower half-byte if least significant number >1001 or if preceding instruction caused a Half-carry. Adds 0110 to upper half-byte if most significant number >1001 or if preceding instruction caused a Carry. Also adds 0110 to upper half-byte if least significant number >1001 and most significant number = 9.

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?

② (Bit C) Test: Result = 00000000?

③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

Figure 4-7 Arithmetic Instructions

Logic Operations

The Logic Instructions and their effect on the CCR are shown in Figure 4-8. Note that the Complement (COM) instruction applies to memory locations as well as both accumulators.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG 5 H	4 I	3 N	2 Z	1 V	0 C
And	ANDA	$A \cdot M \rightarrow A$	•	•	↕	↕	R	•
	ANDB	$B \cdot M \rightarrow B$	•	•	↕	↕	R	•
Complement, 1's	COM	$\overline{M} \rightarrow M$	•	•	↕	↕	R	S
	COMA	$\overline{A} \rightarrow A$	•	•	↕	↕	R	S
	COMB	$\overline{B} \rightarrow B$	•	•	↕	↕	R	S
Exclusive OR	EORA	$A \oplus M \rightarrow A$	•	•	↕	↕	R	•
	EORB	$B \oplus M \rightarrow B$	•	•	↕	↕	R	•
Or, Inclusive	ORA	$A + M \rightarrow A$	•	•	↕	↕	R	•
	ORB	$B + M \rightarrow B$	•	•	↕	↕	R	•

Figure 4-8 Logic Instructions

Data Test Operations

The Data Test instructions are shown in Figure 4-9. Bit Test (BIT) is useful for updating the CCR as if the AND function were executed but does not change the contents of the accumulator. The Test (TST) instruction also operates directly on memory and updates the CCR as if a comparison (CMP) to zero had been executed.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG 5 H	4 I	3 N	2 Z	1 V	0 C
Bit Test	BITA	$A \cdot M$	•	•	↕	↕	R	•
	BITB	$B \cdot M$	•	•	↕	↕	R	•
Compare	CMPA	$A - M$	•	•	↕	↕	↕	↕
	CMPB	$B - M$	•	•	↕	↕	↕	↕
Compare Acmltrs	CBA	$A - B$	•	•	↕	↕	↕	↕
Test, Zero or Minus	TST	$M - 00$	•	•	↕	↕	R	R
	TSTA	$A - 00$	•	•	↕	↕	R	R
	TSTB	$B - 00$	•	•	↕	↕	R	R

Figure 4-9 Data Test Instructions

Data Handling Operations

The Data Handling instructions are summarized in Figure 4-10. Note that the Clear (CLR), Decrement (DEC), Increment (INC), and Shift/Rotate instructions all operate directly on memory and update the CCR accordingly.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG. 5 H	4 I	3 N	2 Z	1 V	0 C
Clear	CLR	00 → M	•	•	R	S	R	R
	CLRA	00 → A	•	•	R	S	R	R
	CLRB	00 → B	•	•	R	S	R	R
Decrement	DEC	M − 1 → M	•	•	↕	↕	④	•
	DECA	A − 1 → A	•	•	↕	↕	④	•
	DECB	B − 1 → B	•	•	↕	↕	④	•
Increment	INC	M + 1 → M	•	•	↕	↕	⑤	•
	INCA	A + 1 → A	•	•	↕	↕	⑤	•
	INCB	B + 1 → B	•	•	↕	↕	⑤	•
Load Acmltr	LDAA	M → A	•	•	↕	↕	R	•
	LDAB	M → B	•	•	↕	↕	R	•
Push Data	PSHA	A → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
	PSHB	B → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
Pull Data	PULA	SP + 1 → SP, M_{SP} → A	•	•	•	•	•	•
	PULB	SP + 1 → SP, M_{SP} → B	•	•	•	•	•	•
Rotate Left	ROL	M	•	•	↕	↕	⑥	↕
	ROLA	A	•	•	↕	↕	⑥	↕
	ROLB	B	•	•	↕	↕	⑥	↕
Rotate Right	ROR	M	•	•	↕	↕	⑥	↕
	RORA	A	•	•	↕	↕	⑥	↕
	RORB	B	•	•	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL	M	•	•	↕	↕	⑥	↕
	ASLA	A	•	•	↕	↕	⑥	↕
	ASLB	B	•	•	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR	M	•	•	↕	↕	⑥	↕
	ASRA	A	•	•	↕	↕	⑥	↕
	ASRB	B	•	•	↕	↕	⑥	↕
Shift Right, Logic.	LSR	M	•	•	R	↕	⑥	↕
	LSRA	A	•	•	R	↕	⑥	↕
	LSRB	B	•	•	R	↕	⑥	↕
Store Acmltr.	STAA	A → M	•	•	↕	↕	R	•
	STAB	B → M	•	•	↕	↕	R	•
Transfer Acmltrs	TAB	A → B	•	•	↕	↕	R	•
	TBA	B → A	•	•	↕	↕	R	•

④ (Bit V) Test: Operand = 10000000 prior to execution?

⑤ (Bit V) Test: Operand = 01111111 prior to execution?

⑥ (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred.

Figure 4-10 Data Handling Instructions

PROGRAM CONTROL OPERATIONS

Program Control operation can be subdivided into two categories: (1) Index Register/Stack Pointer instructions; (2) Jump and Branch operations.

Index Register/Stack Pointer Operations

The instructions for direct operation on the MPU's Index Register and Stack Pointer are summarized in Figure 4-11. Decrement (DEX, DES), increment (INX, INS), load (LDX, LDS) and store (STX, STS) instructions are provided for both. The Compare instruction, CPX, can be used to compare the Index Register to a 16-bit value and update the Condition Code Register accordingly.

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Compare Index Reg	CPX	$(X_H/X_L) - (M/M+1)$	•	•	①	↕	②	•
Decrement Index Reg	DEX	$X - 1 \rightarrow X$	•	•	•	↕	•	•
Decrement Stack Pntr	DES	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX	$X + 1 \rightarrow X$	•	•	•	↕	•	•
Increment Stack Pntr	INS	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	$M \rightarrow X_H, (M+1) \rightarrow X_L$	•	•	③	↕	R	•
Load Stack Pntr	LDS	$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	•	•	③	↕	R	•
Store Index Reg	STX	$X_H \rightarrow M, X_L \rightarrow (M+1)$	•	•	③	↕	R	•
Store Stack Pntr	STS	$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	•	•	③	↕	R	•
Indx Reg → Stack Pntr	TXS	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr → Indx Reg	TSX	$SP + 1 \rightarrow X$	•	•	•	•	•	•

① (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?
② (Bit V) Test: 2's complement overflow from subtraction of LS bytes?
③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

Figure 4-11 Index Register and Stack Pointer Instructions

The TSX instruction causes the Index Register to be loaded with the address of the last data byte put onto the "stack". The TXS instruction loads the Stack Pointer with a value equal to one less than the current contents of the Index Register. This causes the next byte to be pulled from the "stack" to come from the location indicated by the Index Register. The utility of these two instructions can be clarified by describing the "stack" concept relative to the M6800 system.

The "stack" can be thought of as a sequential list of data stored in the 6800's read/write memory. The Stack Pointer contains a 16-bit memory address that is used to access the list from one end on a last-in-first-out (LIFO) basis in contrast to the random access mode used by the MPU's other addressing modes.

The M6800 instruction set and interrupt structure allow extensive use of the stack concept for efficient handling of data movement, subroutines and interrupts. The instructions can be used to establish one or more "stacks" anywhere in read/write memory. Stack length is limited only by the amount of memory that is made available.

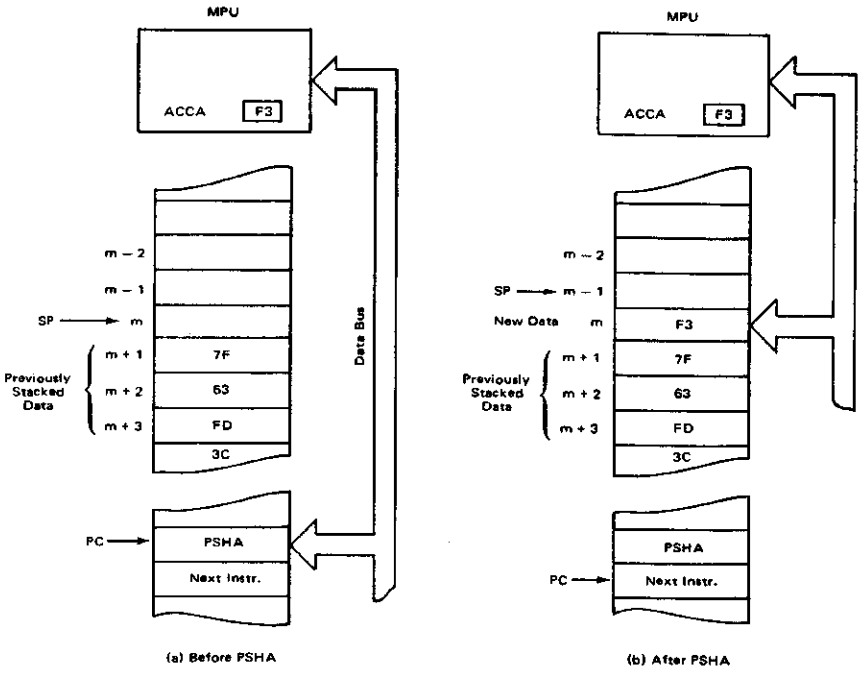


Figure 4-12 Stack Operation, Push Instruction

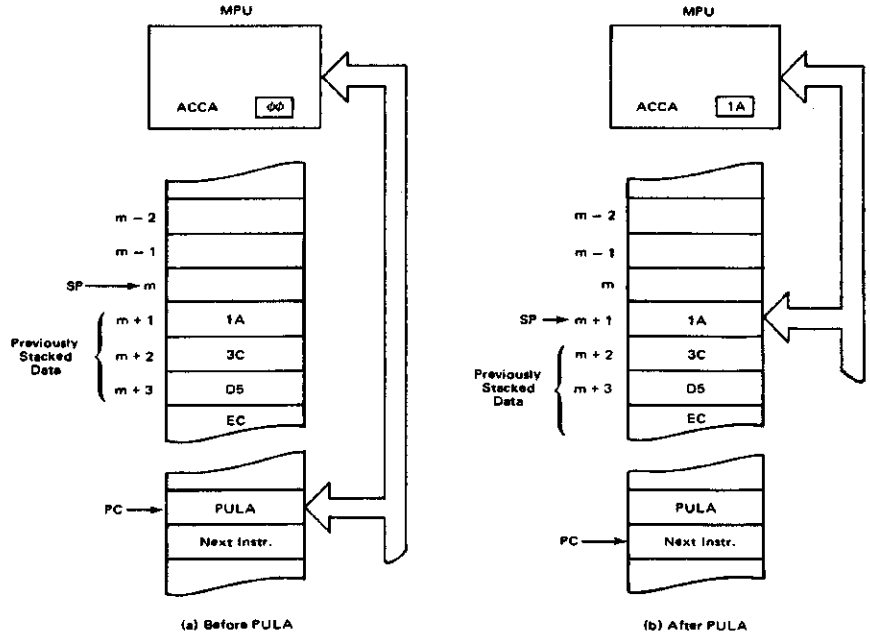


Figure 4-13 Stack Operation, Pull Instruction

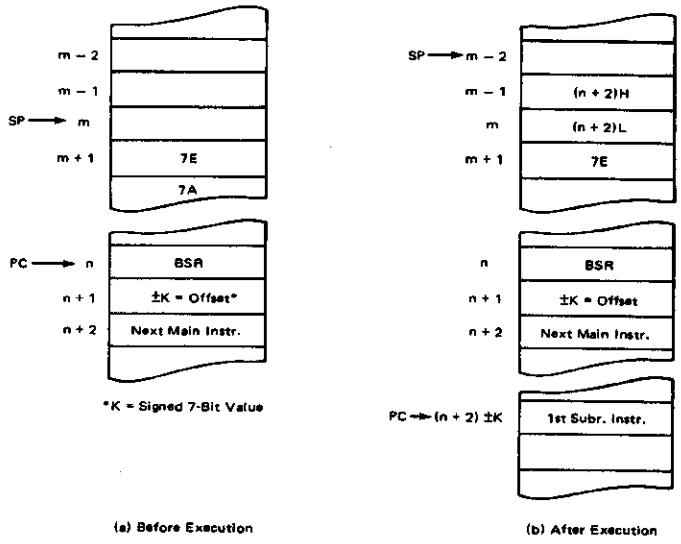


Figure 4-14 Program Flow for BSR

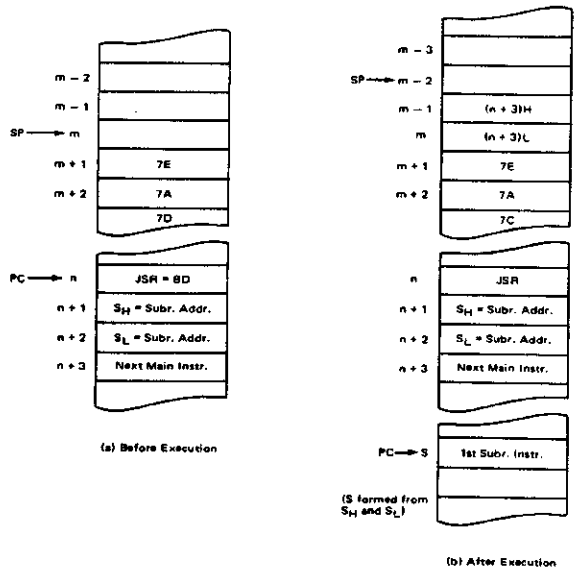


Figure 4-15 Program Flow for JSR (Extended)

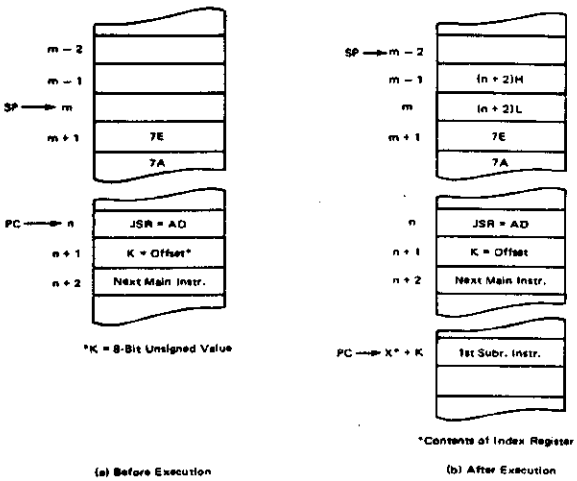


Figure 4-16 Program Flow for JSR (Indexed)

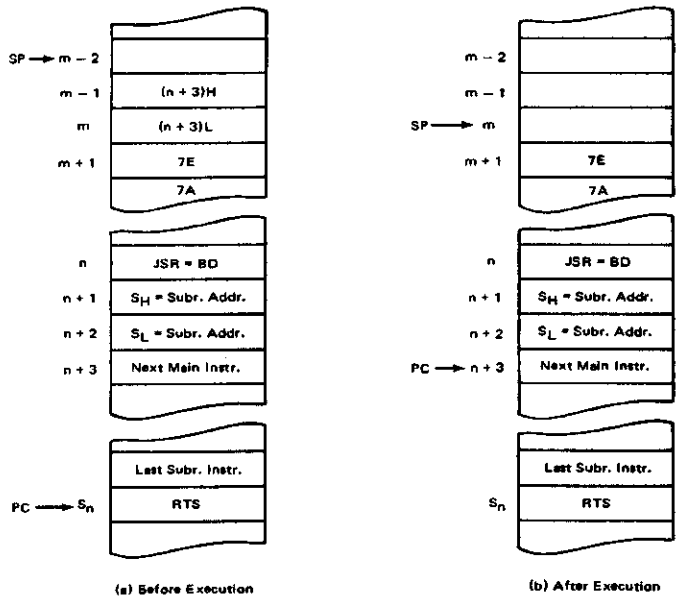


Figure 4-17 Program Flow for RTS

Operation of the Stack Pointer with the Push and Pull instructions is illustrated in Figures 4-12 and 4-13. The Push instruction (PSHA) causes the contents of the indicated accumulator (A in this example) to be stored in memory at the location indicated by the Stack Pointer. The Stack Pointer is automatically decremented by one following the storage operation and is "pointing" to the next empty stack location. The Pull instruction (PULA or PULB) causes the last byte stacked to be loaded into the appropriate accumulator. The Stack Pointer is automatically incremented by one just prior to the data transfer so that it will point to the last byte stacked rather than the next empty location. Note that the PULL instruction does not "remove" the data from memory; in the example, 1A is still in location (m+1) following execution of PULA. A subsequent PUSH instruction would overwrite that location with the new "pushed" data.

Execution of the Branch to Subroutine (BSR) and Jump to Subroutine (JSR) instructions causes a return address to be saved on the stack as shown in figures 4-14 through 4-16. The stack is decremented after each byte of the return address is pushed onto the stack. For both of these instructions, the return address is the memory location following the bytes of code that correspond to the BSR and JSR instruction. The code required for BSR or JSR may be either two or three bytes, depending on whether the JSR is in the indexed (two bytes) or the extended (three bytes) addressing mode. Before it is stacked, the Program Counter is automatically incremented the correct number of times to point at the location of the next instruction. The Return from Subroutine instruction, RTS, causes the return address to be retrieved and loaded into the Program Counter as shown in Figure 4-17.

There are several operations that cause the status of the MPU to be saved on the stack. The Software Interrupt (SWI) and Wait for Interrupt (WAI) instructions as well as the maskable (IRQ) and non-maskable (NMI) hardware interrupts all cause the MPU's internal registers (except for the Stack Pointer itself) to be stacked as shown in Figure 4-21. MPU status is restored by the Return from Interrupt, RTI, as shown in Figure 4-21.

Jump and Branch Operations

The Jump and Branch instructions are summarized in Figure 4-18. These instructions are used to control the transfer of operation from one point to another in the control program.

The No Operation instruction, NOP, while included here, is a jump operation in a very limited sense. Its only effect is to increment the Program Counter by one. It is useful during program development as a "stand-in" for some other instruction that is to be determined during debug. It is also used for equalizing the execution time through alternate paths in a control program.

Execution of the Jump Instruction, JMP, and Branch Always, BRA, effects program flow as shown in Figure 4-19. When the MPU encounters the Jump (indexed) instruction, it adds the offset to the value in the Index Register and uses the result as the address of the next instruction to be executed. In the extended addressing mode, the address of the next instruction to be executed is fetched from the two locations immediately following the JMP instruction. The Branch Always (BRA) instruction is similar to the JMP (extended) instruction except that the relative addressing mode applies. The opcode for the BRA instruction requires one less byte than JMP (extended) but takes one more cycle to execute.

JUMP AND BRANCH

OPERATIONS	MNEMONIC	BRANCH TEST	5 H	4 I	3 N	2 Z	1 V	0 C
Branch Always	BRA	None	•	•	•	•	•	•
Branch If Carry Clear	BCC	C = 0	•	•	•	•	•	•
Branch If Carry Set	BCS	C = 1	•	•	•	•	•	•
Branch If = Zero	BEQ	Z = 1	•	•	•	•	•	•
Branch If $\geq$ Zero	BGE	N ⊕ V = 0	•	•	•	•	•	•
Branch If > Zero	BGT	Z + (N ⊕ V) = 0	•	•	•	•	•	•
Branch If Higher	BHI	C + Z = 0	•	•	•	•	•	•
Branch If $\leq$ Zero	BLE	Z + (N ⊕ V) = 1	•	•	•	•	•	•
Branch If Lower Or Same	BLS	C + Z = 1	•	•	•	•	•	•
Branch If < Zero	BLT	N ⊕ V = 1	•	•	•	•	•	•
Branch If Minus	BMI	N = 1	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	Z = 0	•	•	•	•	•	•
Branch If Overflow Clear	BVC	V = 0	•	•	•	•	•	•
Branch If Overflow Set	BVS	V = 1	•	•	•	•	•	•
Branch If Plus	BPL	N = 0	•	•	•	•	•	•
Branch To Subroutine	BSR	See Special Operations	•	•	•	•	•	•
Jump	JMP		•	•	•	•	•	•
Jump To Subroutine	JSR		•	•	•	•	•	•
No Operation	NOP	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI	See special Operations	—	—	①	—	—	—
Return From Subroutine	RTS		•	•	•	•	•	•
Software Interrupt	SWI		•	S	•	•	•	•
Wait for Interrupt	WAI		•	②	•	•	•	•

① (All) Load Condition Code Register from Stack. (See Special Operations)

② (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

Figure 4-18 Jump and Branch Instructions

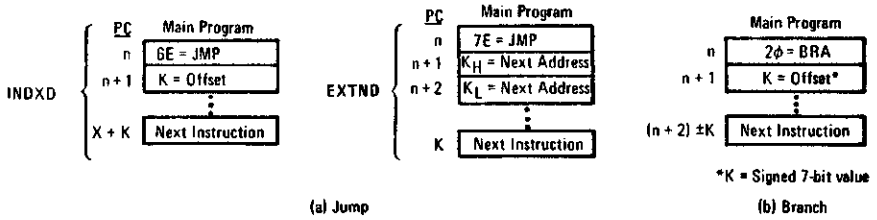


Figure 4-19 Program Flow for Jump and Branch Instructions

The effect on program flow for the Jump to Subroutine (JSR) and Branch to Subroutine (BSR) is shown in Figures 4-14 through 4-16. Note that the Program Counter is properly incremented to be pointing at the correct return address before it is stacked. Operation of the Branch to Subroutine and Jump to Subroutine (extended) instruction is similar except for the range. The BSR instruction requires less opcode than JSR (2 bytes versus 3 bytes) and also executes one cycle faster than JSR. The Return from Subroutine, RTS, is used at the end of a subroutine to return to the main program as indicated in Figure 4-17.

The effect of executing the Software Interrupt, SWI, and the Wait for Interrupt, WAI, and their relationship to the hardware interrupts is shown in Figure 4-20. SWI causes the MPU contents to be stacked and then fetches the starting address of the interrupt routine from the memory locations that respond to the addresses FFFA and FFFB. Note that as in the case of the subroutine instructions, the Program Counter is incremented to point at the correct return address before being stacked. The Return from Interrupt instruction, RTI, (figure 4-21) is used at the end of an interrupt routine to restore control to the main program. The SWI instruction is useful for inserting break points in the control program, that is, it can be used to stop operation and put the MPU registers in memory where they can be examined. The WAI instruction is used to decrease the time required to service a hardware interrupt; it stacks the MPU contents and then waits for the interrupt to occur, effectively removing the stacking time from a hardware interrupt sequence.

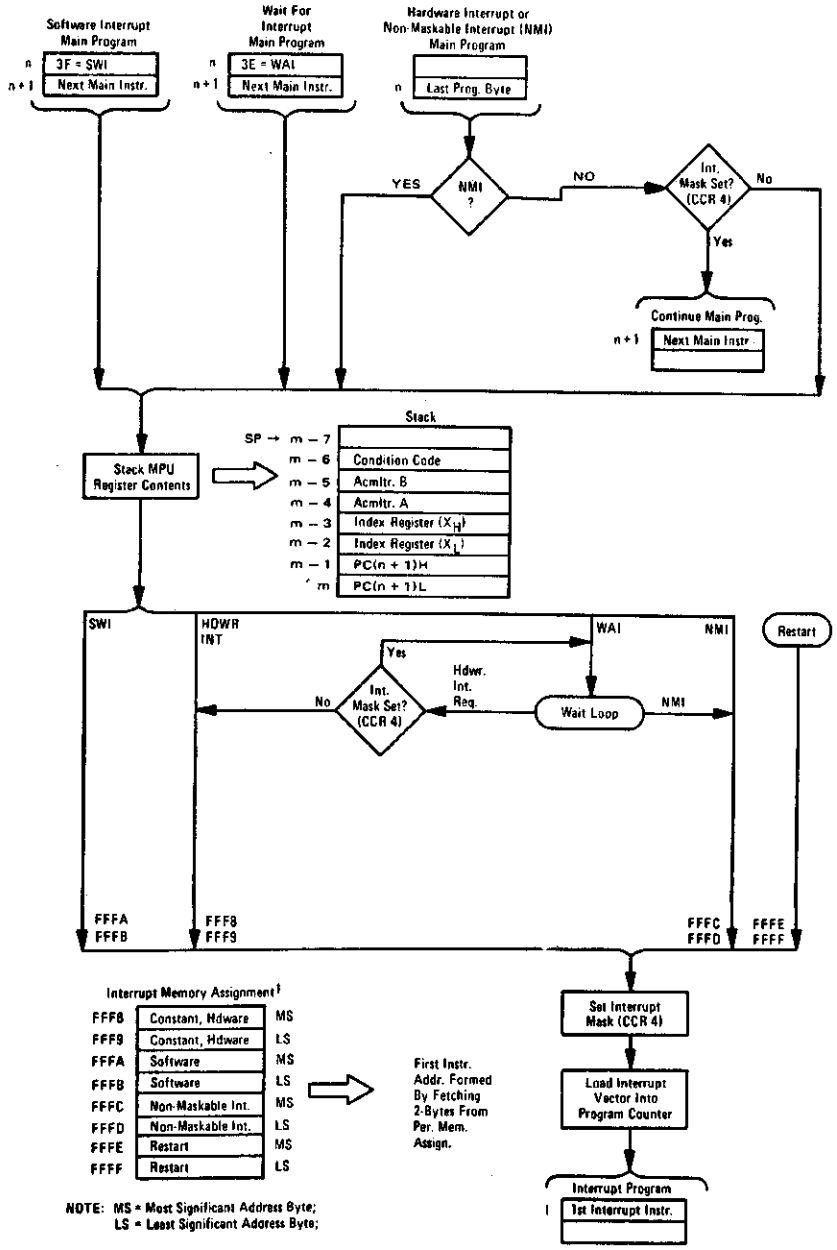


Figure 4-20 Program Flow for Interrupts

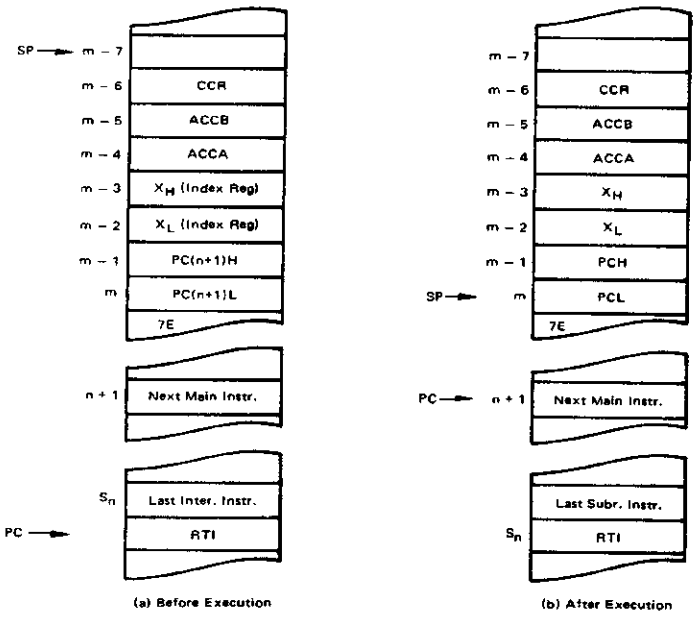


Figure 4-21 Program Flow for RTI

The conditional branch instructions, Figure 4-22, consist of seven pairs of complementary instructions. They are used to test the results of the preceding operation and either continue with the next instruction in sequence (test fails) or cause a branch to another point in the program (test succeeds).

Four of the pairs are used for simple tests of status bits N, Z, V, and C:

(1) Branch On Minus (BMI) and Branch On Plus (BPL) tests the sign bit, N, to determine if the previous result was negative or positive, respectively.

(2) Branch On Equal (BEQ) and Branch On Not Equal (BNE) are used to test the zero status bit, Z, to determine whether or not the result of the previous operation was equal to zero. These two instructions are useful following a Compare (CMP) instruction to test for equality between an accumulator and the operand. They are also used following the Bit Test (BIT) to determine whether or not the same bit positions are set in an accumulator and the operand.

(3) Branch On Overflow Clear (BVC) and Branch On Overflow Set (BVS) tests the state of the V bit to determine if the previous operation caused an arithmetic overflow.

(4) Branch On Carry Clear (BCC) and Branch On Carry Set (BCS) tests the state of the C bit to determine if the previous operation caused a carry to occur. BCC and BCS are useful for testing relative magnitude when the values being tested are regarded as unsigned binary numbers, that is, the values are in the range 00 (lowest) to FF (highest). BCC following a comparison (CMP) will cause a branch if the (unsigned) value in the accumulator is higher than or the same as the value of the operand. Conversely, BCS will cause a branch if the accumulator value is lower than the operand.

The fifth complementary pair, Branch On Higher (BHI) and Branch On Lower or Same (BLS) are in a sense complements to BCC and BCS. BHI tests for both C and Z = 0; if used following a CMP, it will cause a branch if the value in the accumulator is higher than the operand. Conversely, BLS will cause a branch if the unsigned binary value in the accumulator is lower than or the same as the operand.

The remaining two pairs are useful in testing results of operations in which the values are regarded as signed two's complement numbers. This differs from the unsigned binary case in the following sense: In unsigned, the orientation is higher or lower; in signed two's complement, the comparison is between larger or smaller where the range of values is between -128 and +127.

Branch On Less Than Zero (BLT) and Branch On Greater Than Or Equal To Zero (BGE) test the status bits for N + V = 1 and N + V = 0, respectively. BLT will always cause a branch following an operation in which two negative numbers were added. In addition, it will cause a branch following a CMP in which the value in the accumulator was negative and the operand was positive. BLT will never cause a branch following a CMP in which the accumulator value was positive and the operand negative. BGE, the complement to BLT, will cause a branch following operations in which two positive values were added or in which the result was zero.

The last pair, Branch On Less Than Or Equal Zero (BLE) and Branch On Greater Than Zero (BGT) test the status bits for $Z \oplus (N + V) = 1$ and $Z + (N \oplus V) = 0$, respectively. The action of BLE is identical to that for BLT except that a branch will also occur if the result of the previous result was zero. Conversely, BGT is similar to BGE except that no branch will occur following a zero result.

```
BMI :   N = 1 ;          BEQ :   Z = 1 ;
BPL :   N = ϕ ;          BNE :   Z = ϕ ;

BVC :   V = ϕ ;          BCC :   C = ϕ ;
BVS :   V = 1 ;          BCS :   C = 1 ;

BHI :   C + Z = ϕ ;      BLT :   N ⊕ V = 1 ;
BLS :   C + Z = 1 ;      BGE :   N ⊕ V = ϕ ;

        BLE :   Z + (N ⊕ V) = 1 ;
        BGT :   Z + (N ⊕ V) = ϕ ;
```

Figure 4-22 Conditional Branch Instructions

V. ARITHMETIC OPERATIONS

NUMBER SYSTEMS

The ALU (Arithmetic Logic Unit) always performs standard binary addition of two eight bit numbers with the numbers represented in 2's complement form. However, the MPU instruction set and hardware flags permit arithmetic operation using any of four different representations for the numbers:

(1) Each byte can be interpreted as a signed 2's complement number in the range -127 to +127:

$\pm 2^6$	2^5	2^4	2^3	2^2	2^1	2^0		
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
1	0	0	0	0	0	0	1	(-127 in 2's complement representation)
1	1	1	1	1	1	1	1	(-1 in 2's complement representation)
0	0	0	0	0	0	0	0	(0 in 2's complement representation)
0	0	0	0	0	0	0	1	(+1 in 2's complement representation)
0	1	1	1	1	1	1	1	(+127 in 2's complement representation)

(2) Each byte can be interpreted as an unsigned binary number in the range 0 to 255:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
0	0	0	0	0	0	0	0	(0 in unsigned binary)
1	1	1	1	1	1	1	1	(255 in unsigned binary)

(3) Each byte contains one 4-bit BCD number in the 4 LSBITS, the 4MS bits are zeros. This is referred to as unpacked BCD and can represent numbers in the range of 0-9:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
0	0	0	0	0	0	0	0	(BCD 0)
0	0	0	0	0	1	0	1	(BCD 5)
0	0	0	0	1	0	0	1	(BCD 9)

Must always be zero

(4) Each byte can be thought of as containing two 4-bit binary coded decimal (BCD) numbers. With this interpretation, each byte can represent numbers in the range 0 to 99:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
0	0	0	0	0	0	0	0	(BCD 00)
0	0	1	0	0	1	1	1	(BCD 27)
1	0	0	1	1	0	0	1	(BCD 99)

Each of these number systems will be illustrated with programming examples after the condition code flags and instruction set have been introduced in more detail.

THE CONDITION CODE REGISTER

During operation, the MPU sets (or clears) flags in a Condition Code Register as indicated below:

b_5	b_4	b_3	b_2	b_1	b_0	Condition Code Register
H	I	N	Z	V	C	

H = Half-carry; set whenever a carry from b_3 to b_4 of the result is generated; cleared otherwise.

I = Interrupt Mask; set by an interrupt or SEI instruction; cleared by CLI instruction. (Normally not used in arithmetic operations).

N = Negative; set if high order bit (b_7) of result is set; cleared otherwise.

Z = Zero; set if result = 0; cleared otherwise.

V - oVerflow; set if there was arithmetic overflow as a result of the operation; cleared otherwise.

C - Carry; set if there was a carry from the most significant bit (b_7) of the result; cleared otherwise.

OVERFLOW

The description of most of the condition code bits is straight forward. However, overflow requires clarification. Arithmetic overflow is an indication that the last operation resulted in a number beyond the +127 range of an 8-bit byte. Overflow can be determined by examining the sign bits of the operands and the result as indicated in Table 5-1 where the results for addition of A + B is shown.

Row	a_7	b_7	r_7	V	
1	0	0	0	0	
2	0	0	1	1	
3	0	1	0	0	
4	0	1	1	0	
5	1	0	0	0	(A + B) = R
6	1	0	1	0	
7	1	1	0	1	
8	1	1	1	0	

TABLE 5-1: Overflow for Addition

If the sign bits of the operands, a_7 and b_7, are different (rows 3 through 6 of the Table) no overflow can occur and the V flag is clear after the operation. If the operand sign bits are alike and the result exceeds the byte capacity, the sign bit of the result (r_7) will change and the overflow bit will be set. This is illustrated in the following example. The example follows actual ALU operation in that the starting number A is initially in the accumulator but is replaced by the result of the current operation.

```
V 7 6 5 4 3 2 1 0
0 0 0 1 1 0 1 1 0  A = +54;
  1 0 0 0 0 1 1 1  B = -121; (negative numbers are in 2's complement
                             notation)
-----------------
0 1 0 1 1 1 1 0 1  R0 = A + B = -67; (signs of A & B different; no
                             overflow)

0 1 0 1 1 1 1 0 1  R0 = -67;
  1 1 0 1 1 1 1 1  B = -33;
-----------------
0 1 0 0 1 1 1 0 0  R1 = R0 + B = -100; (Signs alike but byte capacity
                             not exceeded; no overflow)

V 7 6 5 4 3 2 1 0
  1 0 0 1 1 1 0 0  R1 = -100;
  1 1 1 0 0 0 0 0  B = -32;
-----------------
1 0 1 1 1 1 1 0 0  R2 = +124 (Signs of R1 & B alike and sign of result
                             occurred)
```

Here the capacity of the register has been exceeded and the result is +124 rather than -32. Overflow is said to have occurred.

In subtraction operations, the possibility of overflow exists whenever the operands differ in sign. Overflow conditions for A - B are illustrated in Table 5-2.

Row	a_7	$\overline{b_7}$	r_7	V	
1	0	0	0	0	
2	0	0	1	1	
3	0	1	0	0	
4	0	1	1	0	(A - B) = R
5	1	0	0	0	
6	1	0	1	0	
7	1	1	0	1	
8	1	1	1	0	

TABLE 5-2
Overflow for Subtraction

Note that Table 5-2 is identical to the addition table except that b_7 has been replaced by $\overline{b_7}$. This is explained by the fact that the ALU performs subtraction by adding the negative of the subtrahend B to the minuend A. Hence, the ALU first forms the 2's complement of B and then adds. The subtraction table with b_7 negated then reflects the sign bits of two numbers that are to be added. If a_7 and $\overline{b_7}$ are alike, overflow will occur if the byte capacity is exceeded.

THE ARITHMETIC INSTRUCTIONS

Table 5-3 summarizes the instructions used primarily for arithmetic operations. The effect of each operation on memory and the MPU's Accumulators is shown along with how the result of each operation effects the Condition Code Register.

The carry bit is used as a carry for addition and as a borrow for subtraction and is added to the Accumulators with the Add With Carry Instructions and subtracted from the Accumulators in the Subtract With Carry instructions.

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	5 H	4 I	3 N	2 Z	1 V	0 C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	•	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	•	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\overline{M} \rightarrow M$	•	•	↕	↕	R	S
	COMA													43	2	1	$\overline{A} \rightarrow A$	•	•	↕	↕	R	S
	COMB													53	2	1	$\overline{B} \rightarrow B$	•	•	↕	↕	R	S
Complement, 2's (Negate	NEG							60	7	2	70	6	3				00 − M → M	•	•	↕	↕	①	②
	NEGA													40	2	1	00 − A → A	•	•	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	•	•	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	•	•	↕	↕	↕	③
Rotate Left	ROL							69	7	2	79	6	3				M	•	•	↕	↕	⑥	↕
	ROLA													49	2	1	A	•	•	↕	↕	⑥	↕
	ROLB													59	2	1	B	•	•	↕	↕	⑥	↕
Rotate Right	ROR							66	7	2	76	6	3				M	•	•	↕	↕	⑥	↕
	RORA													46	2	1	A	•	•	↕	↕	⑥	↕
	RORB													56	2	1	B	•	•	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M	•	•	↕	↕	⑥	↕
	ASLA													48	2	1	A	•	•	↕	↕	⑥	↕
	ASLB													58	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M	•	•	↕	↕	⑥	↕
	ASRA													47	2	1	A	•	•	↕	↕	⑥	↕
	ASRB													57	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Logic	LSR							64	7	2	74	6	3				M	•	•	R	↕	⑥	↕
	LSRA													44	2	1	A	•	•	R	↕	⑥	↕
	LSRB													54	2	1	B	•	•	R	↕	⑥	↕
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	•	•	↕	↕	↕	↕
Subract Acmltrs	SBA													10	2	1	A − B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	•	•	↕	↕	↕	↕

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
#	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
−	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer;	C	Carry from bit 7
		R	Reset Always
		S	Set Always
+	Boolean Inclusive OR;	↕	Test and set if true, cleared otherwise
⊕	Boolean Exclusive OR;	•	Not Affected
$\overline{M}$	Complement of M;	CCR	Condition Code Register
→	Transfer into;	LS	Least Significant
0	Bit = Zero;	MS	Most Significant

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?

② (Bit C) Test: Result = 00000000?

③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

⑥ (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred.

Table 5-3 Arithmetic Instructions

The Decimal Adjust instruction, DAA, is used in BCD addition to adjust the binary results of the ALU. When used following the operations, ABA, ADD, and ADC on BCD operands, DAA will adjust the contents of the accumulator and the C bit to represent the correct BCD Sum.

Table 5-4 shows the details of the DAA instruction and how it affects and is effected by the Condition Code Register bits.

Operation: Adds hexadecimal numbers 00, 06, 60, or 66 to ACCA, and may also set the carry bit, as indicated in the following table:

State of C-Bit Before DAA (Col. 1)	Upper Half-Byte (Bits 4–7) (Col. 2)	Initial Half-Carry H-Bit (Col. 3)	Lower Half-Byte (Bits 0–3) (Col. 4)	Number Added to ACCA by DAA (Col. 5)	State of C-Bit After DAA (Col. 6)
0	0–9	0	0–9	00	0
0	0–8	0	A–F	06	0
0	0–9	1	0–3	06	0
0	A–F	0	0–9	60	1
0	9–F	0	A–F	66	1
0	A–F	1	0–3	66	1
1	0–2	0	0–9	60	1
1	0–2	0	A–F	66	1
1	0–3	1	0–3	66	1

NOTE: Columns (1) to (4) of the above table represent all possible cases which can result from any of the operations ABA, ADD, or ADC, with initial carry either set or clear, applied to two binary-coded-decimal operands. The table shows hexadecimal values.

Effect on Condition Code Register:

H Not affected.

I Not affected.

N Set if most significant bit of the result is set; cleared otherwise.

Z Set if all bits of the result are cleared; cleared otherwise.

V Not defined.

C Set or reset according to the same rule as if the DAA and an immediately preceding ABA, ADD, or ADC were replaced by a hypothetical binary-coded-decimal addition.

Table 5-4 Effect of DAA Instruction

Use of Arithmetic Instructions

Typical use of the arithmetic instructions is illustrated in the following examples:

The ABA instruction adds the contents of ACCB to the contents of ACCA:

ACCA	10101010	($AA)
ACCB	11001100	($CC)
ACCA	01110110	($76) with a carry.
CARRY	1	

The ADCA instruction adds the operand data and the carry bit to ACCA:

ACCA	1	0	1	0	1	0	1	0	$AA
OPERAND DATA	1	1	0	0	1	1	0	0	CC
CARRY									
ACCA	0	1	1	1	0	1	1	1	$77 with carry
CARRY								1	

In both of these examples, the 2's complement overflow bit, V, will be set as shown in Table 5-5.

2's complement overflow after	carry after	b_7 ACC after	b_7 ACC before	b_7 OPERAND (OR ACCB) before
0	0	0	0	0
1	0	1	0	0
0	0	1	0	1
0	1	0	0	1
0	0	1	1	0
0	1	0	1	0
1	1	0	1	1
0	1	1	1	1

TABLE 5-5
Truth Table for "Add with Carry"

The SUBA instruction subtracts the operand data from ACCA:

	b	b	b	b	b	b	b	b	
ACCA	0	1	1	0	0	1	0	1	$65
OPERAND DATA	1	0	0	0	0	1	1	1	$87
ACCA	1	1	0	1	1	1	1	0	$DE with a borrow
BORROW								1	

The SBCA instruction subtracts the operand and the borrow (carry) it from ACCA.

	b	b	b	b	b	b	b	b	
ACCA	1	0	1	1	1	1	0	0	$BC
OPERAND DATA	0	1	1	1	1	0	1	1	$7B
BORROW (carry)								1	C=1
	0	1	0	0	0	0	0	0	$40 no borrow
BORROW								0	

The 2's complement overflow and carry bits are set in accordance with Table 5-6 as a result of a subtraction operation.

2's complement overflow	carry after	b_7 ACCA after	b_7 ACCA before	b_7 OPERAND before
0	0	0	0	0
0	1	1	0	0
0	1	0	0	1
1	1	1	0	1
1	0	0	1	0
0	0	1	1	0
0	0	0	1	1
0	1	1	1	1

TABLE 5-6
Truth Table for "Subtract with Borrow"

ADDITION AND SUBTRACTION ROUTINES

Most applications will require that the arithmetic instruction set be combined into more complex routines that operate on numbers larger than one byte. If more than one number system is used, routines must be written for each, or conversion routines to some common base must be used. In many cases, however, it is more efficient to write a specialized routine for each system requirement, i.e., hexadecimal (HEX) versus unpacked BCD multiplication, etc. In this section, several algorithms will be discussed with specific examples showing their implementation with the M6800 instruction set.

The basic arithmetic operations are binary addition and subtraction:

```
ALPHA + BETA = GAMMA          ALPHA - BETA = GAMMA

    LDAA    ALPHA                 LDAA    ALPHA

    ADDA     BETA                 SUBA     BETA

    STAA    GAMMA                 STAA    GAMMA
```

These operations are so short that they are usually programmed in line with the main flow. Addition of single packed BCD bytes requires only one more instruction. The DAA instruction is used immediately after the ADD, ADC, or ABA instructions to adjust the binary generated in accumulator A (ACCA) to correct BCD value:

```
                    LDAA    ALPHA
                    ADDA     BETA
                    DAA
                    STAA    GAMMA
```

Carry	ACCA			
X	67	0110 0111	=ACCA	
X	+79	carry 0111 1001	=MEMORY	
0	146	0 1110 0000	=ACCA	binary result
	46	1 0100 0110	=ACCA	after DAA; the carry bit will also be set because of the BCD carry.

Since no similar instruction is available for BCD subtraction, 10's complement arithmetic may be used to generate the difference. The follow routine performs a BCD subtraction of two digit BCD numbers:

```
LDAA    #$99

SUBA    BETA    (99-BETA) = ACCA

SEC             carry = 1

ADCA    ALPHA   ACCA + ALPHA + C = ACCA

DAA             DECIMAL ADJUST (-100)

STAA    GAMMA   ALPHA-BETA = GAMMA
```

The routine implements the algorithm defined by the following equations.

```
ALPHA - BETA = GAMMA

ALPHA + (99-BETA) - 99 = GAMMA          9's COMPLEMENT OF BETA

ALPHA + (99-BETA + 1) - 100 = GAMMA   10's COMPLEMENT OF BETA
```

One is added to the 9's complement of the subtrahend by setting the carry bit to find the 10's complement of BETA which is then added to the minuend ALPHA and saved in ACCA. The DAA instruction adjusts the result in ACCA to the proper BCD values before storing the difference in GAMMA. Since 100 has been added (99 + 1) to the subtrahend by finding the 10's complement, 100 must also be subtracted. This is accomplished by the DAA instruction since the resulting carry is discarded.

Multiple precision operations mean that the data and results require more than one byte of memory. The simplest multiple precision routines are addition and subtraction of 16 bit binary or 2's complement numbers. This is often called double precision since 2 consecutive bytes are required to store 16 binary bits of information. The following routines illustrate these functions:

```
LDAA    ALPHA + 1
LDAB    ALPHA
ADDA    BETA + 1    ADD LS BYTES
ADCB    BETA        ADD MS BYTES WITH CARRY FROM LS BYTES
STAA    GAMMA +1
STAB    GAMMA

LDAA    ALPHA + 1
LDAB    ALPHA
SUBA    BETA + 1    SUBTRACT LS BYTES
SBCB    BETA        SUBTRACT MS BYTES WITH BORROW FROM LS BYTES
STAA    GAMMA + 1
STAB    GAMMA
```

Four digit BCD addition can be accomplished in a similar fashion with the use of the DAA instruction. The following routine has been expanded to a 2N digit addition where N is the max number of packed BCD bytes used:

```
START   CLC
        LDX     #N
LOOP    LDAA    ALPHA,X
        ADCA    BETA,X
        DAA
        STAA    GAMMA,X
        DEX
        BNE     LOOP
```

NOTE: ALPHA, BETA, and GAMMA must be in the direct addressing range and adjusted for offset for this example (See indexed addressing for further details).

This routine uses indexed address to select the bytes to be added, starting with the least significant. The carry is cleared at the start and is affected only by the DAA and ADCA instructions. This allows the carry to be included in the next byte addition.

Expanding subtraction to multiple precision is accomplished in a manner similar to the single byte case; 10's complement arithmetic is used. A suitable routine is shown in the Assembly Listing of Figure 5-7.

```
00010                    NAM   DSUB16
00030                    OPT   SYMB
00060       0000   SUBTRH EQU   0
00070       0008   MINUEN EQU   8
00080       0010   RSLT   EQU   16
00090 0100               ORG   256
00092              * DECIMAL SUBTRACT SUBROUTINE FOR 16 DECIMAL DIGIT

00094              * THIS ROUTINE SUBTRACTS THE SUBTRAHEND ("SUBTRH")
00095              * FROM THE MINUEND ("MINUEN") AND PLACES THE
00096              * DIFFERENCE IN "RSLT".

00097              * THE MEMORY ALLOCATION IS AS FOLLOWS:
00097              *                ADDRESS RANGE    LSB
00097              *    SUBTRAHEND      1-8          8
00097                  MINUEND         9-16         16
00097              *    DIFFERENCE     17-24         24
                         ADDRESS VALUES ARE DECIMAL

00100 0100 CE 0008 DSUB  LDX   #8     SET BYTE COUNTER
00110 0103 86 99   DSUB1 LDA A #$99
00120 0105 A0 00         SUB A SUBTRH,X  FIND 9'S COMPLEMENT
00130 0107 A7 10         STA A RSLT,X  USE "RSLT" AS TEMP STORE
00140 0109 09            DEX           DECREMENT BYTE COUNTER
00150 010A 26 F7         BNE   DSUB1   LOOP UNTIL LAST BYTE
00160 010C CE 0008       LDX   #8      RESTORE BYTE COUNTER
00170 010F 0D            SEC           SET CARRY TO ADD 1 TO COMPL
00180 0110 A6 08   DSUB2 LDA A MINUEN,X  LOAD MINUEND
00190 0112 A9 10         ADC A RSLT,X  ADD COMPLEMENT SUBTRAHEND
00200 0114 19            DAA           DECIMAL ADJUST
00210 0115 A7 10         STA A RSLT,X  STORE DIFFERENCE
00220 0117 09            DEX           DECREMENT BYTE COUNTER
00230 0118 26 F6         BNE   DSUB2   LOOP UNTIL LAST BYTE
00240 011A 39            RTS           RETURN TO HOST PROGRAM


00251              *  THE EXECUTION TIME OF THIS SUBROUTINE IS
00252              *  384 MPU CYCLES EXCLUDING THE RTS.

00254                    END

SYMBOL TABLE


DSUB   0100 DSUB1  0103 DSUB2  0110 MINUEN 0008 RSLT   0010
SUBTRH 0000
```

Figure 5-7 Decimal Subtract Assembly Listing

This routine first finds the 9's complement of the subtrahend and stores it in the result buffer. The carry is then set to add 1 to the 9's complement, making it the 10's complement which is then added to the minuend and stored in the result buffer. Note that this routine has 2 loops, the first to calculate the 9's complement, the second to add and decimal adjust the result. The decimal add and subtract routines operate on 10's complement numbers as well as packed BCD numbers. A number is known to be negative in 10's complement form when the most significant digit in the most significant byte is a 9. When in the 10's complement form, this digit is reserved for the sign and the actual number of magnitude digits is one less than 2 times the number of bytes. A routine similar to the above subtract program will convert the 10's complement number to decimal magnitude with sign for display or output purposes:

```
DCONV    CLR    SINFLG      CLEAR SIGN FLAG
         LDAA   RESULT+1    GET MSBYTE
         BPL    END         POSITIVE:END
         LDX    #8          NEGATIVE:
DCONV1   LDAA   #$99
         SUBA   RSLT,X      SUBTRACT RESULT FROM
         STAA   RSLT,X      ALL 9's INCLUDING
         DEX                SIGN DIGIT
         BNE    DCONV1
         LDX    #8
         CLRA
         SEC
DCONV2   ADCA   RSLT,X      ADD 1 TO RESULT
         DAA
         STAA   RSLT,X
         DEX
         BNE    DCONV2
         DEC    SINFLG      SET SIGN FLAG
END      RTS                RETURN
```

The sign flag would be used to indicate plus when clear and minus when not clear.

MULTIPLICATION

Multiplication increases programming complexity. In addition to the addition and subtraction instructions, the use of the shift and rotate instructions is required. The general algorithm for binary multiplication can be illustrated by a short example:

(1) Test the least significant multiplier bit for 1 or 0.

(a) If it is 1, add the multiplicand to the result, then 2.

(b) If it is 0, then 2.

(2) Shift the multiplicand left one bit.

(3) Test the next more significant multiplier bit; then 1a or 1b.

DECIMAL	BINARY		
13	1101		MULTIPLICAND
11	1011		MULTIPLIER LSB = 1; ADD MULTIPLICAND TO RESULT (A)
	1101	(A)	
13	1101	(B)	SHIFT MULTIPLICAND LEFT ONE BIT (B)
	100111	(C)	LSB+1 = 1; ADD MULTIPLICAND TO RESULT (C)
13	1101	(D)	SHIFT MULTIPLICAND LEFT ONE BIT (D)
	1101	(E)	LSB+2 = 0; SHIFT MULTIPLICAND LEFT 1 (E)
143	10001111	(F)	LSB+3 = 1; ADD MULTIPLICAND TO RESULT (F)

128 + 15 = 143

Signed binary numbers in 2's complement form cannot be multiplied without correcting for the cross product terms which are introduced by the 2's complement representation of negative numbers. There is an algorithm which generates the correct 2's complement product. Since positive binary numbers are correct 2's complement notations, they also may be multiplied using this procedure. It is called Booth's Algorithm. Simply stated the algorithm says:

(1) Test the transition of the multiplier bits from right to left assuming an imaginary 0 bit to the immediate right of the multiplier.

(2) If the bits in question are equal, then 5.

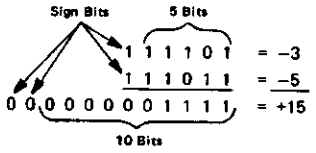
(3) If there is a 0 to 1 transition, the multiplicand is subtracted from the product, then 5.

(4) If there is a 1 to 0 transition, the multiplicand is added to the product, then 5.

(5) Shift the product right one bit with the MSBit remaining the same. (This has the same effect as shifting the multiplicand left in the previous example).

(6) Go to 1 to test the next transition of the multiplier.

The following example (Figure 5-8) shows the typical steps involved in an actual calculation. A flow chart and assembly listing for a multiplication program using the M6800 instruction set are shown in Figures 5-9 and 5-10, respectively.



	1 1 1 1 0 1	Multiplicand
	1 1 1 0 1 1 (0)	Multiplier
	0 0 0 0 0 0	0 to 1; subtract by adding the 2's
+	0 0 0 0 1 1	complement of the multiplicand
	0 0 0 0 1 1	PRODUCT
	0 0 0 0 0 1 1	Shift PRODUCT
	0 0 0 0 0 0 1 1	1 to 1 shift PRODUCT
+	1 1 1 1 0 1	1 to 0 add
	1 1 1 1 0 1 1 1	PRODUCT
	1 1 1 1 1 0 1 1 1	Shift PRODUCT
+	0 0 0 0 1 1	0 to 1 subtract
	0 0 0 0 0 1 1 1 1	PRODUCT
	0 0 0 0 0 0 1 1 1 1	Shift PRODUCT
	0 0 0 0 0 0 0 1 1 1 1	1 to 1 shift
	0 0 0 0 0 0 0 0 1 1 1 1	1 to 1 shift
	Sign 15	

Figure 5-8 Multiplication Using Booth's Algorithm

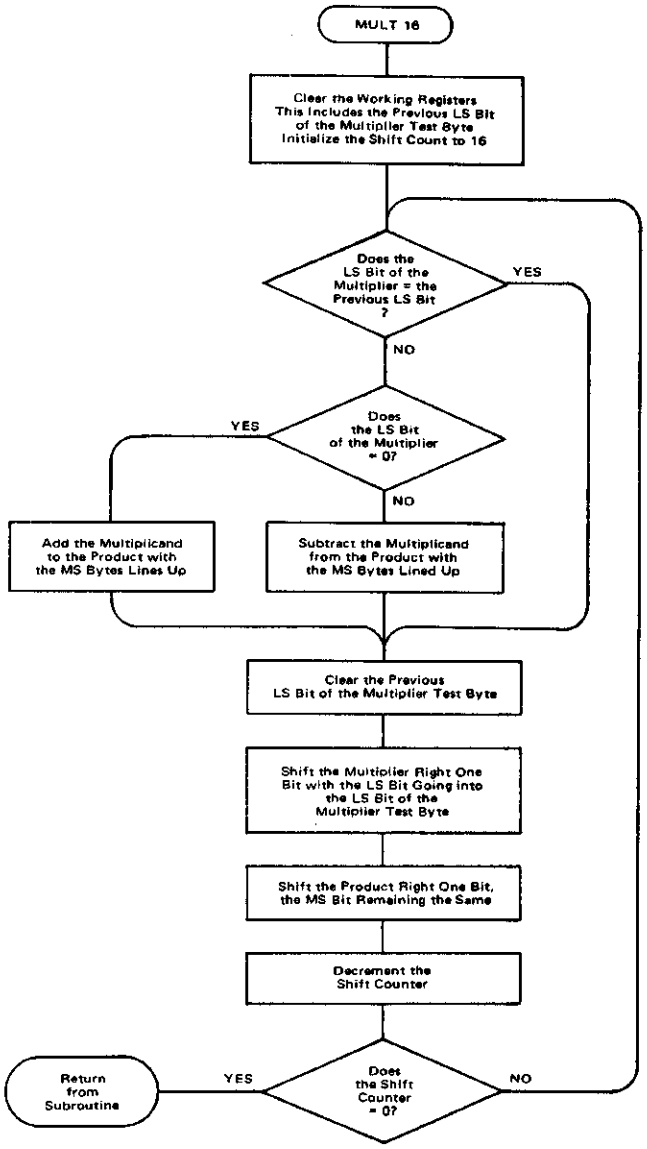


Figure 5-9 Flow Chart for Booth's Algorithm

```
00010                    NAM  MULTI6
00020                    OPT  NOPAGE
00030              *
00040              *  THIS ROUTINE MULTIPLIES TWO 16 BIT 2'S
00050              *  COMPLIMENT NUMBERS USING BOOTH'S ALGORITHM
00060              *
00070              *  THE MULTIPLIER = Y = Y(MSB),Y(LSB) = Y,Y+1
00080              *  THE MULTIPLICAND =XX=XX(MSB),XX(LSB) = XX,XX+1
00090              *  THE PRODUCT = U = U(MSB),U+1,U+2,U+3
00100              *  THE TEST BYTE FOR Y(LSB-1) = FF
00110              *
00120 0080               ORG  $80
00130 0080 0002  Y      RMB  2
00140 0082 0002  XX     RMB  2
00150 0084 0004  U      RMB  4
00160 0088 0001  FF     RMB  1
00170              *
00180              *  THE MULTIPLIER AND THE MULTIPLICAND MUST BE
00190              *  STORED IN Y AND XX RESPECTIVELY, THEN A JSR TO
00200              *  MULTI6 WILL GENERATE THE 2'S COMPLIMENT PRODUCT
00210              *  OF Y AND XX IN U.
00220              *
00230              *  THE MULTIPLICAND WILL BE UNCHANGED, THE
00240              *  MULTIPLIER WILL BE DESTROYED.
00250              *
00260 0400               ORG  $400
```

Figure 5-10 Assembly Listing for Booth's Algorithm (Sheet 1 of 2)

```
00270 0400 CE 0005 MULT16 LDX   #5      CLEAR THE WORKING REGISTERS
00280 0403 4F              CLR A
00290 0404 A7 83   LP1     STA A U-1,X
00300 0406 09              DEX
00310 0407 26 FB           BNE   LP1
00320 0409 CE 0010         LDX   #16    INIT'L SHIFT COUNTER TO 16
00330 040C 96 81   LP2     LDA A Y+1    GET Y(LSBIT)
00340 040E 84 01           AND A #1
00350 0410 16              TAB          SAVE Y(LSBIT) IN ACCB
00360 0411 98 88           EOR A FF     DOES Y(LSBIT) = Y(LSB-1) ?
00370 0413 27 1D           BEQ   SHIFT  YES: GO TO SHIFT ROUTINE
00380 0415 5D              TST B        NO:  DOES Y(LSBIT) = 0 ?
00390 0416 27 0E           BEQ   ADD    YES:  GO TO ADD ROUTINE
00400 0418 96 85           LDA A U+1    NO:  SUBTRACT MULTIPLICAND
00410 041A D6 84           LDA B U      PRODUCT WITH THE MSBYTES
00420 041C 90 83           SUB A XX+1   LINED UP
00430 041E D2 82           SBC B XX
00440 0420 97 85           STA A U+1
00450 0422 D7 84           STA B U
00460 0424 20 0C           BRA   SHIFT  THEN GO TO SHIFT ROUTINE
00470 0426 96 85   ADD     LDA A U+1    ADD THE MULTIPLICAND TO THE
00480 0428 D6 84           LDA B U      PRODUCT WITH THE MSBYTES
00490 042A 9B 83           ADD A XX+1   LINED UP
00500 042C D9 82           ADC B XX
00510 042E 97 85           STA A U+1
00520 0430 D7 84           STA B U
00530 0432 7F 0088 SHIFT   CLR   FF     CLEAR THE TEST BYTE
00540 0435 76 0080         ROR   Y      SHIFT THE MULTIPLIER RIGHT
00550 0438 76 0081         ROR   Y+1    ONE BIT WITH THE LSBIT
00560 043B 79 0088         ROL   FF     INTO THE LSBIT OF FF
00570 043E 77 0084         ASR   U      SHIFT THE PRODUCT RIGHT ONE
00580 0441 76 0085         ROR   U+1    BIT, THE MSB REMAINING THE
00590 0444 76 0086         ROR   U+2    SAME
00600 0447 76 0087         ROR   U+3
00610 044A 09              DEX          DECREMENT THE SHIFT COUNT
00620 044B 26 BF           BNE   LP2    IF NOT 0 CONTINUE
00630 044D 39              RTS
00640                      END
```

Figure 5-10 Assembly Listing for Booth's Algorithm (Sheet 2 of 2)

DIVISION

A flow chart for binary division is shown in Figure 5-11. The assembly listing of the program is given in Figure 5-12.

The algorithm used for this straight forward binary division is as follows:

(1) Left justify the divisor byte.

(2) If the MS byte of the dividend is less than the divisor byte, shift quotient left one bit with the LS bit = 0; then 4.

(3) If the MS byte of the dividend is greater than or equal to the divisor, (2)shift the quotient left one bit with the LS Bit = 1; (b) subtract the divisor from the MS byte of the dividend, the result being stored in the MS byte of the dividend; then 4.

(4) Shift the dividend left one bit with the LS Bit = 0, and the MS Bit going into the carry.

(5) If the carry is set, go to 3a.

(6) If the carry is not set, go to 2a.

The process continues until the number of quotient shifts equals 8 + number of shifts required to left justify the divisor.

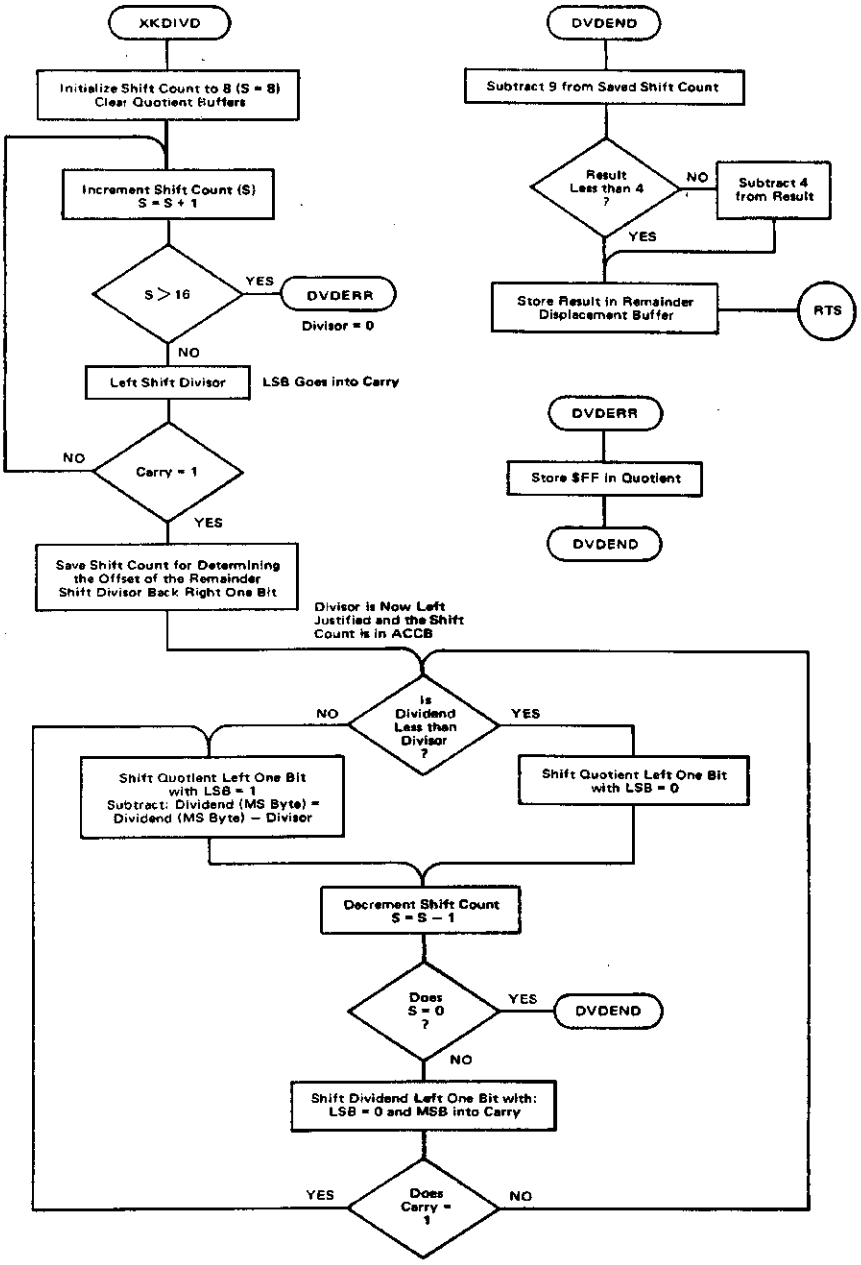


Figure 5-11 XKDIVD Flow Chart

```
00100                    OPT     L
00000                    NAM     XKDIVD
00010                    OPT     NOPAGE
00020 5900               ORG     B5900
00030              *  SUBROUTINE TO DIVIDE AN UNSIGNED 4 DIGIT
00040              *  HEX NUMBER [16 BIT BINARY] BY AN UNSIGNED
00050              *  2 DIGIT HEX NUMBER [8 BIT BINARY].
00060              *
00070              *  THE DIVISOR = X = XKDVSR = [F9]
00080              *  THE DIVIDEND = Y(M),Y(L)
00090              *               =XKDVND,XKDVND+1
00100              *               =[FA,FB]
00110              *  THE QUOTIENT = Q(M),Q(L)
00120              *               =XKQUOT,XKQUOT+1
00130              *               =[FC,FD]
00140              *  THE SHIFT COUNTER = S = ACCB
00150              *  THE LEFT DISPLACEMENT OF THE REMAINDER = XKDSPL
00160              *                                         = [FE]
00170              *
00180              *  THE DIVISOR AND THE DIVIDEND MUST BE LOADED
00190              *  INTO XKDVSR AND XKDVND,XKDVND+1 RESPECTIVELY
00200              *  THEN A JSR TO XKDIVD.
00210              *
00220              *  THE REMAINDER WILL BE IN Y(M) [XKDVND],
00230              *  SHIFTED LEFT THE # OF BITS INDICATED IN XKDSPL
00240              *  THE DIVISOR WILL BE BINARILY LEFT JUSTIFIED
```

Figure 5-12 XKDIVD Assembly Listing (Sheet 1 of 2)

```
00260 5900 C6 08     XKDIVD LDA B  #8        INIT'L S=8
00270 5902 7F 00FC          CLR    XKQUOT   ZERO QUOTIENT BUFFER
00280 5905 7F 00FD          CLR    XKQUOT+1
00290 5908 5C        DVDLOP INC B           S-S+1
00300 5909 C1 10            CMP B  #16
00310 590B 2E 34            BGT    DVDERR   IF S>16 DIVIDE ERROR
00320 590D 73 00F9          ASL    XKDVSR   IF S<16 LEFT SHIFT DIVISOR
00330 5910 24 F6            BCC    DVDLP0   IF C=0 CON'T LOOP
00340 5912 D7 FE            STA B  XKDSPL   IF C=1 XKDSPL = SHIFT COUNT
00350 5914 76 00F9          ROR    XKDVSR   SHIFT THE DIVISOR BACK 1
00360                *                       SHIFT COUNT NOW IN ACCB
00370                *                       DIVISOR LEFT JUST. IN X
00380 5917 96 FA            LDA A  XKDVND
00390 5919 91 F9     DVDLP1 CMP A  XKDVSR   IF THE DIVIDEND<DIVISOR
00400 591B 25 0D            BCS    DVNSUB   DON'T SUBTRACT
00410 591D 0D        DVDLP2 SEC             IF THE DIVIDENT >OR=DIVISOR
00420 591E 79 00FD          ROL    XKQUOT+1 SHIFT Q LEFT 1 BIT
00430 5921 79 00FC          ROL    XKQUOT   WITH LSB = 1
00440 5924 90 F9            SUB A  XKDVSR   Y(M) = Y(M)-X
00450 5926 97 FA            STA A  XKDVND
00460 5928 20 07            BRA    DVSHFT
00470 592A 0C        DVNSUB CLC             SHIFT Q LEFT WITH
00480 592B 79 00FD          ROL    XKQUOT+1 LSB = 0
00490 592E 79 00FC          ROL    XKQUOT
00500 5931 5A        DVSHFT DEC B           S = S-1
00510 5932 27 12            BEQ    DVDEND   IF S = 0 STOP
00520 5934 0C               CLC             IF S>0 SHIFT DIVIDEND
00530 5935 79 00FB          ROL    XKDVND+1 LEFT ONE BIT; LSB=0
00540 5938 79 00FA          ROL    XKDVND   MSB INTO CARRY
00550 593B 95 FA            LDA A  XKDVND
00560 593D 25 DE            BOA    DVDLP2   IF C = 1 GO TO LOOP 2
00570 593F 20 D8            BRA    DVDLP 1  GO TO LOOP 1
00580 5941 CE FFFF DVDERR   LDX    #$FFFF
00590 5944 DF FC            STX    XKQUOT
00600 5946 D6 FE   DVDEND   LDA B  XKDSPL   GET SHIFT COUNT INTO ACCB
00610 5948 C0 09            SUB B  #9       XKDSPL = XKDSPL-9
00620 594A C1 04            CMP B  #4       XKDSPL 4
00630 594C 25 02            BCS    DVDLP3   YES:  GO TO RETURN
00640 594E C0 04            SUB B  #4       NO:  XKDSPL=XKDSPL-4
00650 5950 D7 FE   DVDLP3   STA B  XKDSPL   DISPLACEMENT OF REMAINDER
00660              *                        STORED IN XKDSPL
00670 5952 39               RTS
00680                       END
```

Figure 5-12 XKDIVD Assembly Listing (Sheet 2 of 2)

NOTE

Section V is, by no means, comprehensive. It is intended to provide some examples that can be used as is or that will suggest the direction for modifying them for other specialized applications.

Sample Programs

VI

```
00100                           NAM         PUNCH
00200                   **
00300                   * PUNCH MOTOROLA HEX FORMAT TAPES
00400                   **
00500                   * USE MONITOR'S J COMMAND TO START EXECUTION
00600                   * AT 000C
00700                   **
00800                   * ENTER ADDRESS OF FIRST BYTE TO PUNCH
00900                   **
01000                   * ENTER ADDRESS OF LAST BYTE TO PUNCH
01100                   **
01200                   * MONITOR ROUTINES
01300                   * ADDRESSES ARE FOR ACIA VERSION OF MONITOR
01400                   **
01500        FF81       OUTCH   EQU         $FF81
01600        FF6D       OUT2H   EQU         $FF6D
01700        FF62       BADDR   EQU         $FF62
01800        FF82       OUTS    EQU         $FF82
01900        FFAB       CRLF    EQU         $FFAB
02000                   **
02100 0000              ORG         0
02200                   **
02300                   * DATA RECORD FORMAT
02400                   *
02500 3000 0D           FORM    FCB         $D,$A,0,0,'S,'1,$FF
      0001 0A
      0002 00
      0003 00
      0004 53
      0005 31
      0006 FF
02600 0007 0002     BEGADR  RMB         2           FIRST ADDR TO PUNCH
02700 0009 0002     LASADR  RMB         2           LAST ADDR TO PUNCH
02800 000B 0001     NUMBYT  RMB         1
02900 000C 8D 69            BSR         GETADR      GET FIRST ADDR
03000 000E DF 07            STX         BEGADR      STORE IT
03100 0010 8D 65            BSR         GETADR      GET LAST ADDR
03200 0012 DF 09            STX         LASADR      STORE IT
03300 0014 8D 4C            BSR         LEDTRL      PUNCH LEADER
03400 0016 CE FFFF  PUN     LDX         #FORM-1     POINT TO PUNCH FORMAT
03500 0019 08       PUN0    INX
03600 001A E6 00            LDA B       X
03700 001C 2B 05            BMI         PUN1        HIGH ORDER BIT SET - DONE
03800 001E BD FF81          JSR         OUTCH       PUNCH CHARACTER
03900 0021 20 F6            BRA         PUN0
04000 0023 96 0A    PUN1    LDA A       LASADR+1    SUB LOW ORDER BYTES
04100 0025 90 08            SUB A       BEGADR+1
04200 0027 D6 09            LDA B       LASADR      SUB HIGH ORDER BYTES
04300 0029 D2 07            SBC B       BEGADR
04400 002B 26 04            BNE         PUN2        LOTS MORE TO PUNCH
04500 002D 81 10            CMP A       #16         LESS THAN 16 TO PUNCH?
04600 002F 25 02            BCS         PUN3
04700 0031 86 0F    PUN2    LDA A       #15         NO, SO PUNCH 16
04800 0033 97 0B    PUN3    STA A       NUMBYT      STORE #OF BYTES TO PUNCH-1
04900 0035 8B 04            ADD A       #4
05000 0037 BD FF6D          JSR         OUT2H       PUNCH BYTE COUNT
05100 003A 08               INX                     POINT TO BEGADR
05200 003B 8D 2F            BSR         PNCH2       PUNCH ADDRESS
05300 003D 8D 2D            BSR         PNCH2
05400 003F DE 07            LDX         BEGADR      POINT TO DATA
05500 0041 8D 29    PUN4    BSR         PNCH2       PUNCH DATA
05600 0043 7A 000B          DEC         NUMBYT      MORE TO PUNCH THIS RECORD?
```

continued over

```
05700 0046 2A F9          BPL       PUN4
05800 0048 DF 07          STX       BEGADR    STORE NEW START ADDRESS
05900 004A 43             COM A               FORM 1'S COMP OF CHECKSUM
06000 004B BD FF6D        JSR       OUT2H     PUNCH CHECKSUM
06100 004E 09             DEX                 ADJUST POINTER
06200 004F 9C 09          CPX       LASADR    ARE WE DONE?
06300 0051 26 C3          BNE       PUN       NO, KEEP ON PUNCHING
06400 0053 C6 53          LDA B     #'S       YES, PUNCH EOF
06500 0055 BD FF81        JSR       OUTCH
06600 0058 C6 39          LDA B     #'9
06700 005A BD FF81        JSR       OUTCH
06800 005D 8D 03          BSR       LEDTRL    PUNCH TRAILER
06900 005F 7E FFAB        JMP       CRLF      RETURN TO PROM MONITOR
07000                  **
07100                  * SUBROUTINE TO PUNCH 50 NULLS
07200                  **
07300 0062 86 32   LEDTRL LDA A     #50
07400 0064 5F             CLR B
07500 0065 BD FF81 LED1   JSR       OUTCH     PUNCH A NULL
07600 0068 4A             DEC A
07700 0069 26 FA          BNE       LED1      KEEP PUNCHING
07800 006B 39             RTS                 RETURN TO CALLER
07900                  **
08000                  * PUNCH 2 HEX DIGITS POINTED
08100                  * TO BY X REG AND UPDATE CHECKSUM
08200                  **
08300 006C E6 00   PNCH2  LDA B     X         GET BYTE TO PUNCH
08400 006E 1B             ABA                 UPDATE CHECKSUM
08500 006F 36             PSH A               SAVE CHECKSUM
08600 0070 17             TBA                 COPY BYTE TO A
08700 0071 BD FF6D        JSR       OUT2H     PUNCH BYTE
08800 0074 32             PUL A               RESTORE CHECKSUM
08900 0075 08             INX                 BUMP BYTE POINTER
09000 0076 39             RTS                 RETURN TO CALLER
09100                  **
09200                  * READ ADDRESS FORM TTY INTO X REG
09300                  **
09400 0077 BD FF82 GETADR JSR       OUTS      SEND SPACE
09500 007A C6 3F          LDA B     #'?       SEND QUESTION MARK
09600 007C BD FF81        JSR       OUTCH
09700 007F BD FF62        JSR       BADDR     GET ADDRESS
09800 0082 39             RTS                 RETURN
09900                     END

TOTAL ERRORS 00000
```

```
00001                    NAM       MEMTEST
00002                    **
00003                    * ALTAIR 680B MEMORY TEST PROGRAM
00004                    **
00005                    * USE MONITOR'S J COMMAND TO START EXECUTION
00006                    * AT 0019
00007                    **
00008                    * ENTER ADDRESS OF FIRST LOCATION TO TEST
00009                    **
00010                    * ENTER ADDRESS OF LAST ADDRESS TO TEST
00011                    **
00012                    * MONITOR ROUTINES
00013                    * ADDRESSES ARE FOR ACIA VERSION OF MONITOR
00014                    **
00015      FF81     OUTCH   EQU       @177601
00016      FF6D     OUT2H   EQU       @177555
00017      FF62     BADDR   EQU       @177542
00018      FF82     OUTS    EQU       @177602
00019      FFAB     MONIT   EQU       @177653
00020                    **
00021 0014               ORG       $14
00022 0014 0001  STACK   RMB       1
00023 0015 0001  XHIGH   RMB       1         X REG HIGH ORDER
00024 0016 0001  XLOW    RMB       1         X REG LOW ORDER
00025 0017 0002  LSTBYT  RMB       2         LAST BYTE TO CHECK
00026                    *
00027 0019 8E 0014 GO    LDS       #STACK    INIT STACK POINTER
00028 001C BD 0053       JSR       GETADR    GET FIRST ADDR
00029 001F DF 15         STX       XHIGH     STORE IT
00030 0021 BD 0053       JSR       GETADR    GET LAST ADDR
00031 0024 08            INX                 ADJUST IT
00032 0025 DF 17         STX       LSTBYT    STORE IT
00033 0027 DE 15         LDX       XHIGH     POINT TO FIRST BYTE
00034 0029 5F      NXTBYT CLR B
00035 002A E7 00   NXTPAT STA B    X         WRITE TEST PATTERN
00036 002C E1 00         CMP B     X         CHECK WRITTEN PATTERN
00037 002E 27 18         BEQ       OKMEM     DID WE READ WHAT WE WROTE?
00038 0030 C6 0D         LDA B     #%1101    NO,SEND CR AND LF
00039 0032 BD FF81       JSR       OUTCH
00040 0035 C6 0A         LDA B     #@12      "@"
00041 0037 BD FF81       JSR       OUTCH
00042 003A DF 15         STX       XHIGH     STORE X REGISTER
00043 003C 96 15         LDA A     XHIGH
00044 003E BD FF6D       JSR       OUT2H     PRINT HIGH BYTE OF ADDRESS
00045 0041 96 16         LDA A     XLOW      PRINT LOW BYTE OF ADDRESS
00046 0043 BD FF6D       JSR       OUT2H
00047 0046 20 03         BRA       BUMP      DONE WITH THIS BYTE
00048 0048 5C      OKMEM INC B               INCREMENT TEST PATTERN
00049 0049 26 DF         BNE       NXTPAT    ALL PATTERNS TESTED?
00050 004B 08      BUMP  INX                 YES, BUMP BYTE POINTER
00051 004C 9C 17         CPX       LSTBYT
00052 004E 26 D9         BNE       NXTBYT    ALL BYTES TESTED?
00053 0050 7E FFAB       JMP       MONIT     YES, RETURN TO PROM MONITOR
00054                    *
00055                    *
00056                    * SUBROUTINE TO GET ADDRESS INTO X REG
00057                    *
00058 0053 BD FF82 GETADR JSR      OUTS      PRINT A SPACE
00059 0056 C6 3F         LDA B     #'?       PRINT A QUESTION MARK
00060 0058 BD FF81       JSR       OUTCH
00061 005B BD FF62       JSR       BADDR     GET ADDRESS
00062 005E 39            RTS                 RETURN TO CALLING PROGRAM
00063                    END

TOTAL ERRORS 00000
```

```
00100                    NAM       DMP
00200                    **
00300                    * ALTAIR 680B HEXADECIMAL MEMORY DUMP PROGRAM
00400                    **
00500                    * LOAD VIA PROM MONITOR
00600                    **
00700                    * USE MONITOR'S J COMMAND TO
00800                    * START EXECUTION AT 0005
00900                    **
01000                    * ENTER ADDRESS OF FIRST BYTE TO DUMP
01100                    **
01200                    * ENTER ADDRESS OF LAST BYTE TO DUMP
01300                    **
01400                    * TYPE ANY CHARACTER TO ABORT WHILE RUNNING
01500                    **
01600                    * CONTROL RETURNS TO THE PROM MONITOR
01700                    **
01800 00F3               ORG       $F3
01900 00F3 FF            FCB       $FF       TURN OFF TTY ECHO DURING LOAD
02000                    *
02100                    * MONITOR ROUTINES
02200                    * ADDRESSES ARE FOR ACIA VERSION OF MONITOR
02300                    *
02400      FF81  OUTCH   EQU       @177601
02500      FF6D  OUT2H   EQU       @177555
02600      FF62  BADDR   EQU       @177542
02700      FF82  OUTS    EQU       @177602
02800      FFAB  MONIT   EQU       @177653
02900      FF24  POLCAT  EQU       @177444
03000 0000               ORG       0
03100 0000 0001  XHI     RMB       1         TEMP FOR HIGH BYTE OF X
03200 0001 0001  XLO     RMB       1         TEMP FOR LOW BYTE OF X
03300 0002 0002  LSTBYT  RMB       2         ADDRESS OF LAST BYTE TO DUMP
03400 0004 0001  COUNT   RMB       1         COLUMN COUNTER
03500 0005 8D 40 GO      BSR       GETADR    GET FIRST ADDR
03600 0007 DF 00         STX       XHI       STORE IT
03700 0009 8D 3C         BSR       GETADR    GET LAST ADDR
03800 000B 08            INX                 ADJUST IT
03900 000C DF 02         STX       LSTBYT    STORE IT
04000 000E DE 00         LDX       XHI       POINT TO FIRST BYTE
04100 0010 C6 0D  CRLF   LDA B     #@15      SEND CRLF
04200 0012 BD FF81       JSR       OUTCH
04300 0015 C6 0A         LDA B     #@12
04400 0017 BD FF81       JSR       OUTCH
04500 001A C6 11         LDA B     #17
04600 001C D7 04         STA B     COUNT     INIT COUNTER
04700 001E DF 00         STX       XHI       PRINT ADDRESS
04800 0020 96 00         LDA A     XHI
04900 0022 BD FF6D       JSR       OUT2H
05000 0025 96 01         LDA A     XLO
05100 0027 BD FF6D       JSR       OUT2H
05200 002A 7A 0004 NXTBYT DEC      COUNT
05300 002D 27 E1         BEQ       CRLF
05400 002F BD FF82       JSR       OUTS      SEND A SPACE
05500 0032 A6 00         LDA A     X         BYTE TO A
05600 0034 BD FF6D       JSR       OUT2H     PRINT IT
05700 0037 08            INX                 BUMP POINTER
05800 0038 9C 02         CPX       LSTBYT    ARE WE DONE?
05900 003A 27 08         BEQ       JMONIT    YES, RETURN TO MONITOR
06000 003C BD FF24       JSR       POLCAT    NO, WANT TO QUIT?
06100 003F 24 E9         BCC       NXTBYT
06200 0041 B6 F001       LDA A     $F001     YES, READ CHAR FROM BUFFER
06300 0044 7E FFAB JMONIT JMP      MONIT     AND RETURN TO MONITOR
06400                    *
06500                    * GETADR LOADS X WITH ADDRESS
06600                    * READ FROM TTY
06700                    *
06800 0047 BD FF82 GETADR JSR      OUTS      SEND SPACE
06900 004A C6 3F         LDA B     #'?       SEND QUESTION MARK
07000 004C BD FF81       JSR       OUTCH
07100 004F BD FF62       JSR       BADDR     GET ADDDRESS
07200 0052 39            RTS                 RETURN
07300                    *
07400                    * RESTORE TTY ECHO AFTER LOAD
07500                    *
07600 00F3               ORG       $F3
07700 00F3 00            FCB       00
07800                    END

TOTAL ERRORS 00000
```

appendix A

Instruction Set

APPENDIX A
Definition of the Executable Instructions

A.1 Nomenclature

The following nomenclature is used in the subsequent definitions.

(a) *Operators*

( )	=	contents of
←	=	is transferred to
↑	=	"is pulled from stack"
↓	=	"is pushed into stack"
.	=	Boolean AND
⊙	=	Boolean (Inclusive) OR
⊕	=	Exclusive OR
≈	=	Boolean NOT

(b) *Registers in the MPU*

ACCA	=	Accumulator A
ACCB	=	Accumulator B
ACCX	=	Accumulator ACCA or ACCB
CC	=	Condition codes register
IX	=	Index register, 16 bits
IXH	=	Index register, higher order 8 bits
IXL	=	Index register, lower order 8 bits
PC	=	Program counter, 16 bits
PCH	=	Program counter, higher order 8 bits
PCL	=	Program counter, lower order 8 bits
SP	=	Stack pointer
SPH	=	Stack pointer high
SPL	=	Stack pointer low

(c) *Memory and Addressing*

M	=	A memory location (one byte)
M +1	=	The byte of memory at 0001 plus the address of the memory location indicated by "M."
Rel	=	Relative address (i.e. the two's complement number stored in the second byte of machine code corresponding to a branch instruction.

(d) *Bits 0 thru 5 of the Condition Codes Register*

C	=	Carry — borrow	bit — 0
V	=	Two's complement overflow indicator	bit — 1
Z	=	Zero indicator	bit — 2
N	=	Negative indicator	bit — 3
I	=	Interrupt mask	bit — 4
H	=	Half carry	bit — 5

(e) *Status of Individual Bits BEFORE Execution of an Instruction*

An = Bit n of ACCA (n=7,6,5,...,0)
Bn = Bit n of ACCB (n=7,6,5,...,0)
IXHn = Bit n of IXH (n=7,6,5,...,0)
IXLn = Bit n of IXL (n=7,6,5,...,0)
Mn = Bit n of M (n=7,6,5,...,0)
SPHn = Bit n of SPH (n=7,6,5,...,0)
SPLn = Bit n of SPL (n=7,6,5,...,0)
Xn = Bit n of ACCX (n=7,6,5,...,0)

(f) *Status of Individual Bits of the RESULT of Execution of an Instruction*

(i) For 8-bit Results

Rn = Bit n of the result (n =7,6,5,...,0)

This applies to instructions which provide a result contained in a single byte of memory or in an 8-bit register.

(ii) For 16-bit Results

RHn = Bit n of the more significant byte of the result (n =7,6,5,...,0)

RLn = Bit n of the less significant byte of the result (n =7,6,5,...,0)

This applies to instructions which provide a result contained in two consecutive bytes of memory or in a 16-bit register.

A.2 Executable Instructions (definition of)

Detailed definitions of the 72 executable instructions of the source language are provided on the following pages.

Add Accumulator B to Accumulator A **ABA**

Operation: ACCA ← (ACCA) + (ACCB)

Description: Adds the contents of ACCB to the contents of ACCA and places the result in ACCA.

Condition Codes:
- H: Set if there was a carry from bit 3; cleared otherwise.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = A_3.B_3 + B_3.\overline{R}_3. + \overline{R}_3.A_3$$
$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = A_7.B_7.\overline{R}_7 + \overline{A}_7.\overline{B}_7.R_7$$
$$C = A_7.B_7 + B_7.\overline{R}_7 + \overline{R}_7.A_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal:

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
Inherent	2	1	1B	033	027

ADC

Add with Carry

Operation: ACCX ← (ACCX) + (M) + (C)

Description: Adds the contents of the C bit to the sum of the contents of ACCX and M, and places the result in ACCX.

Condition Codes:

H Set if there was a carry from bit 3; cleared otherwise.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3.M_3+M_3.\overline{R}_3+\overline{R}_3.X_3$$
$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = X_7.M_7.\overline{R}_7+\overline{X}_7.\overline{M}_7.R_7$$
$$C = X_7.M_7+M_7.\overline{R}_7+\overline{R}_7.X_7$$

Addressing Formats:

See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	89	211	137
A DIR	3	2	99	231	153
A EXT	4	3	B9	271	185
A IND	5	2	A9	251	169
B IMM	2	2	C9	311	201
B DIR	3	2	D9	331	217
B EXT	4	3	F9	371	249
B IND	5	2	E9	351	233

Add Without Carry **ADD**

Operation: ACCX ← (ACCX) + (M)

Description: Adds the contents of ACCX and the contents of M and places the results in ACCX.

Condition Codes:
- H: Set if there was a carry from bit 3; cleared otherwise.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3.M_3+M_3.\overline{R}_3+\overline{R}_3.X_3$$
$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = X_7.M_7.\overline{R}_7+\overline{X}_7.\overline{M}_7.R_7$$
$$C = X_7.M_7+M_7.\overline{R}_7+\overline{R}_7.X_7$$

Addressing Formats:

See Table A-1

Addressing Modex, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8B	213	139
A DIR	3	2	9B	233	155
A EXT	4	3	BB	273	187
A IND	5	2	AB	253	171
B IMM	2	2	CB	313	203
B DIR	3	2	DB	333	219
B EXT	4	3	FB	373	251
B IND	5	2	EB	353	235

AND

Logical AND

Operation: ACCX ← (ACCX) . (M)

Description: Performs logical "AND" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "AND" of the corresponding bits of M and of ACCX before the operation.)

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = 0$

Addressing Formats:

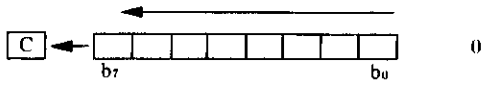
See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	84	204	132
A DIR	3	2	94	224	148
A EXT	4	3	B4	264	180
A IND	5	2	A4	244	164
B IMM	2	2	C4	304	196
B DIR	3	2	D4	324	212
B EXT	4	3	F4	364	244
B IND	5	2	E4	344	228

Arithmetic Shift Left **ASL**

Operation:



Description: Shifts all bits of the ACCX or M one place to the left. Bit 0 is loaded with a zero. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = N \oplus C = [N.\overline{C}] \odot [\overline{N}.C]$

(the foregoing formula assumes values of N and C after the shift operation)

$C = M_7$

Addressing Formats

See Table A-3

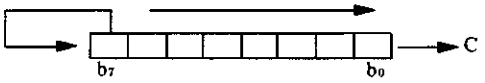
Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	48	110	072
B	2	1	58	130	088
EXT	6	3	78	170	120
IND	7	2	68	150	104

ASR

Arithmetic Shift Right

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit.

Condition Codes:

H: Not affected.
I: Not affected.
N: Set if the most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = N \oplus C = [N.\overline{C}.] \odot [\overline{N}.C]$

(the foregoing formula assumes values of N and C after the shift operation)

$C = M_0$

Addressing Formats:

See Table A-3

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	47	107	071
B	2	1	57	127	087
EXT	6	3	77	167	119
IND	7	2	67	147	103

Branch if Carry Clear **BCC**

Operation: PC ← (PC) + 0002 + Rel if (C)=0

Description: Tests the state of the C bit and causes a branch if C is clear.

See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexedecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	24	044	036

Branch if Carry Set **BCS**

Operation: PC ← (PC) + 0002 + Rel if (C)=1

Description: Tests the state of the C bit and causes a branch if C is set.

See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	25	045	037

BEQ

Branch if Equal

Operation: PC ← (PC) + 0002 + Rel if (Z)=1

Description: Tests the state of the Z bit and causes a branch if the Z bit is set.

See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	27	047	039

Branch if Greater than or Equal to Zero **BGE**

Operation: PC ← (PC) + 0002 + Rel if (N) ⊕ (V) = 0
i.e. if (ACCX) ≥ (M)
(Two's complement numbers)

Description: Causes a branch if (N is set and V is set) OR (N is clear and V is clear).

If the BGE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2C	054	044

BGT

Branch if Greater than Zero

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(Z) \odot [(N) \oplus (V)] = 0$

i.e. if (ACCX) > (M)
(two's complement numbers)

Description: Causes a branch if [Z is clear] AND [(N is set and V is set) OR (N is clear and V is clear)].

If the BGT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2E	056	046

Branch if Higher **BHI**

Operation: PC ← (PC) + 0002 + Rel if (C) . (Z) = 0

i.e. if (ACCX) > (M)
(unsigned binary numbers)

Description: Causes a branch if (C is clear) AND (Z is clear).

If the BHI instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was greater than the unsigned binary number represented by the subtrahend (i.e. M).

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	22	042	034

BIT

Bit Test

Operation: (ACCX) . (M)

Description: Performs the logical "AND" comparison of the contents of ACCX and the contents of M and modifies condition codes accordingly. Neither the contents of ACCX or M operands are affected. (Each bit of the result of the "AND" would be the logical "AND" of the corresponding bits of M and ACCX.)

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the "AND" would be set; cleared otherwise.
- Z: Set if all bits of the result of the "AND" would be cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = 0$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	85	205	133
A DIR	3	2	95	225	149
A EXT	4	3	B5	265	181
A IND	5	2	A5	245	165
B IMM	2	2	C5	305	197
B DIR	3	2	D5	325	213
B EXT	4	3	F5	365	245
B IND	5	2	E5	345	229

Branch if Less than or Equal to Zero **BLE**

Operation: PC ← (PC) + 0002 + Rel if (Z)⊙[(N)⊕(V)]=1

i.e. if (ACCX) ≤ (M)
(two's complement numbers)

Description: Causes a branch if [Z is set] OR [(N is set and V is clear) OR (N is clear and V is set)].

If the BLE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less then or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2F	057	047

BLS

Branch if Lower or Same

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(C) \oplus (Z) = 1$

i.e. if (ACCX) $\leq$ (M)
(unsigned binary numbers)

Description: Causes a branch if (C is set) OR (Z is set).

If the BLS instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was less than or equal to the unsigned binary number represented by the subtrahend (i.e. M).

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	23	043	035

Branch if Less than Zero **BLT**

Operation: PC ← (PC) + 0002 + Rel if (N) ⊕ (V) = 1

i.e. if (ACCX) < (M)
(two's complement numbers)

Description: Causes a branch if (N is set and V is clear) OR (N is clear and V is set).

If the BLT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2D	055	045

BMI

Branch if Minus

Operation: PC ← (PC) + 0002 + Rel if (N) = 1

Description: Tests the state of the N bit and causes a branch if N is set.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2B	053	043

BNE

Branch if Not Equal

Operation: PC ← (PC) + 0002 + Rel if (Z) = 0

Description: Tests the state of the Z bit and causes a branch if the Z bit is clear.

See BRA instruction for details of the execution of the Branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	26	046	038

Branch if Plus **BPL**

Operation: PC ← (PC) + 0002 + Rel if (N) =0

Description: Tests the state of the N bit and causes a branch if N is clear. See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2A	052	042

Branch Always **BRA**

Operation: PC ← (PC) + 0002 + Rel

Description: Unconditional branch to the address given by the foregoing formula, in which R is the relative offset stored as a two's complement number in the second byte of machine code corresponding to the branch instruction.

Note: The source program specifies the destination of any branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler obtains the relative address R from the absolute address and the current value of the program counter PC.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	20	040	032

BSR

Branch to Subroutine

Operation: PC ← (PC) + 0002

↓ (PCL)

SP ← (SP) − 0001

↓ (PCH)

SP ← (SP) − 0001

PC ← (PC) + Rel

Description: The program counter is incremented by 2. The less significant byte of the contents of the program counter is pushed into the stack. The stack pointer is then decremented (by 1). The more significant byte of the contents of the program counter is then pushed into the stack. The stack pointer is again decremented (by 1). A branch then occurs to the location specified by the program.

SEE BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	8	2	8D	215	141

Branch to Subroutine

EXAMPLE

		Memory Location	Machine Code (Hex)	Assembler Language Label	Operator	Operand
A. *Before*						
PC	←	$1000	8D		BSR	CHARLI
		$1001	50			
SP	←	$EFFF				
B. *After*						
PC	←	$1052	**	CHARLI	***	*****
SP	←	$EFFD				
		$EFFE	10			
		$EFFF	02			

Branch if Overflow Clear **BVC**

Operation: PC ← (PC) + 0002 + Rel if (V) = 0

Description: Tests the state of the V bit and causes a branch if the V bit is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	28	050	040

BVS

Branch if Overflow Set

Operation: PC ← (PC) + 0002 + Rel if (V) = 1

Description: Tests the state of the V bit and causes a branch if the V bit is set.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	29	051	041

Compare Accumulators **CBA**

Operation: (ACCA) − (ACCB)

Description: Compares the contents of ACCA and the contents of ACCB and sets the condition codes, which may be used for arithmetic and logical conditional branches. Both operands are unaffected.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.
- Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.
- V: Set if the subtraction would cause two's complement overflow; cleared otherwise.
- C: Set if the subtraction would require a borrow into the most significant bit of the result; clear otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = A_7.\overline{B}_7.\overline{R}_7+\overline{A}_7.B_7.R_7$$
$$C = \overline{A}_7.B_7+B_7.R_7+R_7.\overline{A}_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	11	021	017

CLC

Clear Carry

Operation: C bit ← 0

Description: Clears the carry bit in the processor condition codes register.

Condition Codes:
H: Not affected.
I: Not affected.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Cleared

Boolean Formulae for Condition Codes:
C = 0

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0C	014	012

CLI

Clear Interrupt Mask

Operation: I bit ← 0

Description: Clears the interrupt mask bit in the processor condition codes register. This enables the microprocessor to service an interrupt from a peripheral device if signalled by a high state of the "Interrupt Request" control input.

Condition Codes:
H: Not affected.
I: Cleared.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.

Boolean Formulae for Condition Codes:
I = 0

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0E	016	014

Clear **CLR**

Operation: ACCX ← (0)
or: M ← (0)

Description: The contents of ACCX or M are replaced with zeros.

Condition Codes:
H: Not affected.
I: Not affected.
N: Cleared
Z: Set
V: Cleared
C: Cleared

Boolean Formulae for Condition Codes:
N = 0
Z = 1
V = 0
C = 0

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4F	117	079
B	2	1	5F	137	095
EXT	6	3	7F	177	127
IND	7	2	6F	157	111

CLV

Clear Two's Complement Overflow Bit

Operation: V bit ← 0

Description: Clears the two's complement overflow bit in the processor condition codes register.

Condition Codes:
H: Not affected.
I: Not affected.
N: Not affected.
Z: Not affected.
V: Cleared.
C: Not affected.

Boolean Formulae for Condition Codes:
V = 0

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0A	012	010

Compare **CMP**

Operation: (ACCX) − (M)

Description: Compares the contents of ACCX and the contents of M and determines the condition codes, which may be used subsequently for controlling conditional branching. Both operands are unaffected.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.
- Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.
- V: Set if the subtraction would cause two's complement overflow; cleared otherwise.
- C: Carry is set if the absolute value of the contents of memory is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = X_7.\overline{M}_7.\overline{R}_7+\overline{X}_7.M_7.R_7$$
$$C = \overline{X}_7.M_7+M_7.R_7.\overline{X}_7$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	81	201	129
A IDR	3	2	91	221	145
A EXT	4	3	B1	261	177
A IND	5	2	A1	241	161
B IMM	2	2	C1	301	193
B DIR	3	2	D1	321	209
B EXT	4	3	F1	361	241
B IND	5	2	E1	341	225

COM

Complement

Operation: ACCX ← ≈ (ACCX) = FF − (ACCX)
or: M ← ≈ (M) = FF − (M)

Description: Replaces the contents of ACCX or M with its one's complement. (Each bit of the contents of ACCX or M is replaced with the complement of that bit.)

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Cleared.
- C: Set.

Boolean Formulae for Condition Codes:

$N = R_7$
$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$
$V = 0$
$C = 1$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	43	103	067
B	2	1	53	123	083
EXT	6	3	73	163	115
IND	7	2	63	143	099

Compare Index Register **CPX**

Operation: (IXL) − (M + 1)
(IXH) − (M)

Description: The more significant byte of the contents of the index register is compared with the contents of the byte of memory at the address specified by the program. The less significant byte of the contents of the index register is compared with the contents of the next byte of memory, at one plus the address specified by the program. The Z bit is set or reset according to the results of these comparisons, and may be used subsequently for conditional branching.

The N and V bits, though determined by this operation, are not intended for conditional branching.

The C bit is not affected by this operation.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if the most significant bit of the result of the subtraction from the more significant byte of the index register would be set; cleared otherwise.
Z: Set if all bits of the results of both subtractions would be cleared; cleared otherwise.
V: Set if the subtraction from the more significant byte of the index register would cause two's complement overflow; cleared otherwise.
C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$
$$Z = (\overline{RH_7}.\overline{RH_6}.\overline{RH_5}.\overline{RH_4}.\overline{RH_3}.\overline{RH_2}.\overline{RH_1}.\overline{RH_0}).(\overline{RL_7}.\overline{RL_6}.\overline{RL_5}.\overline{RL_4}.\overline{RL_3}.\overline{RL_2}.\overline{RL_1}.\overline{RL_0})$$
$$V = IXH_7.\overline{M_7}.\overline{RH_7} + \overline{IXH_7}.M_7.RH_7$$

Addressing Formats:

See Table A-5.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8C	214	140
DIR	4	2	9C	234	156
EXT	5	3	BC	274	188
IND	6	2	AC	254	172

DAA

Decimal Adjust ACCA

Operation: Adds hexadecimal numbers 00, 06, 60, or 66 to ACCA, and may also set the carry bit, as indicated in the following table:

State C-bit before DAA (Col. 1)	Upper Half-byte (bits 4-7) (Col. 2)	Initial Half-carry H-bit (Col. 3)	Lower Half-byte (bits 0-3) (Col. 4)	Number Added to ACCA by DAA (Col. 5)	State of C-bit after DAA (Col. 6)
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-3	1	0-3	66	1

Note: Columns (1) to (4) of the above table represent all possible cases which can result from any of the operations ABA, ADD, or ADC, with initial carry either set or clear, applied to two binary-coded-decimal operands. The table shows hexadecimal values.

Description: If the contents of ACCA and the state of the carry-borrow bit C and the half-carry bit H are all the result of applying any of the operations ABA, ADD, or ADC to binary-coded-decimal operands, with or without an initial carry, the DAA operation will function as follows.

Subject to the above condition, the DAA operation will adjust the contents of ACCA and the C bit to represent the correct binary-coded-decimal sum and the correct state of the carry.

Condition Codes: H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Not defined.
C: Set or reset according to the same rule as if the DAA and an immediately preceding ABA, ADD, or ADC were replaced by a hypothetical binary-coded-decimal addition.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

C: See table above.

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	19	031	025

Decrement **DEC**

Operation: $ACCX \leftarrow (ACCX) - 01$
or: $M \leftarrow (M) - 01$

Description: Subtract one from the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow occurs if and only if (ACCX) or (M) was 80 before the operation.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = X_7.\overline{X}_6.\overline{X}_5.\overline{X}_4.\overline{X}_3.\overline{X}_2.\overline{X}_0 + \overline{R}_7.R_6.R_5R_4.R_3.R_2.R_1.R_0$

Addressing Formats:
See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4A	112	074
B	2	1	5A	132	090
EXT	6	3	7A	172	122
IND	7	2	6A	152	106

DES

Decrement Stack Point

Operation: SP ← (SP) − 0001

Description: Subtract one from the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal);

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	34	064	052

DEX

Decrement Index Register

Operation: IX ← (IX) − 0001

Description: Subtract one from the index register

Only the Z bit is set or reset according to the result of this operation.

Condition Codes:
H: Not affected.
I: Not affected.
N: Not affected.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Not affected.
C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7}.\overline{RH_6}.\overline{RH_5}.\overline{RH_4}.\overline{RH_3}.\overline{RH_2}.\overline{RH_1}.\overline{RH_0}).(RL_7.RL_6.RL_5.RL_4.RL_3.RL_2.RL_1.RL_0)$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	09	011	009

Exclusive OR **EOR**

Operation: ACCX ← (ACCX) ⊕ (M)

Description: Perform logical "EXCLUSIVE OR" between the contents of ACCX and the contents of M, and place the result in ACCX. (Each bit of ACCX after the operation will be the logical (EXCLUSIVE OR" of the corresponding bits of M and ACCX before the operation.)

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Cleared
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = 0$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	88	210	136
A DIR	3	2	98	230	152
A EXT	4	3	B8	270	184
A IND	5	2	A8	250	168
B IMM	2	2	C8	310	200
B DIR	3	2	D8	330	216
B EXT	4	3	F8	370	248
B IND	5	2	E8	350	232

INC

Increment

Operation: ACCX ← (ACCX) + 01
or: M ← (M) + 01

Description: Add one to the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow will occur if and only if (ACCX) or (M) was 7F before the operation.
- C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = \overline{X}_7.X_6.X_5.X_4.X_3.X_2.X_1.X_0$$
$$= \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4C	114	076
B	2	1	5C	134	092
EXT	6	3	7C	174	124
IND	7	2	6C	154	108

Increment Stack Pointer **INS**

Operation: SP ← (SP) + 0001

Description: Add one to the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	31	061	049

Increment Index Register **INX**

Operation: IX ← (IX) + 0001

Description: Add one to the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Not affected.
- Z: Set if all 16 bits of the result are cleared; cleared otherwise.
- V: Not affected.
- C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH}_7.\overline{RH}_6.\overline{RH}_5.\overline{RH}_4.\overline{RH}_3.\overline{RH}_2.\overline{RH}_1.\overline{RH}_0).(\overline{RL}_7.\overline{RL}_6.\overline{RL}_5.\overline{RL}_4.\overline{RL}_3.\overline{RL}_2.\overline{RL}_1.\overline{RL}_0)$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	08	010	008

JMP

Jump

Operation: PC ← numerical address

Description: A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table A-7.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	3	3	7E	176	126
IND	4	2	6E	156	110

Jump to Subroutine **JSR**

Operation:

Either: PC ← (PC) + 0003 (for EXTended addressing)
or: PC ← (PC) + 0002 (for INDexed addressing)
Then: ↓ (PCL)
SP ← (SP) − 0001
↓ (PCH)
SP ← (SP) − 0001
PC ← numerical address

Condition Codes: Not affected.

Description: The program counter is incremented by 3 or by 2, depending on the addressing mode, and is then pushed onto the stack, eight bits at a time. The stack pointer points to the next empty location in the stack. A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Addressing Formats:

See Table A-7.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	9	3	BD	275	189
IND	8	2	AD	255	173

Jump to Subroutine

EXAMPLE (extended mode)

		Memory Location	Machine Code (Hex)	Assembler Language Label	Operator	Operand
A. *Before:*						
PC	→	$0FFF	BD		JSR	CHARLI
		$1000	20			
		$1001	77			
SP	←	$EFFF				
B. *After:*						
PC	→	$2077	**	CHARLI	***	*****
SP	→	$EFFD				
		$EFFE	10			
		$EFFF	02			

Load Accumulator **LDA**

Operation: ACCX ← (M)

Description: Loads the contents of memory into the accumulator. The condition codes are set according to the data.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \bar{R}_7.\bar{R}_6.\bar{R}_5.\bar{R}_4.\bar{R}_3.\bar{R}_2.\bar{R}_1.\bar{R}_0$

$V = 0$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	86	206	134
A DIR	3	2	96	226	150
A EXT	4	3	B6	266	182
A IND	5	2	A6	246	166
B IMM	2	2	C6	306	198
B DIR	3	2	D6	326	214
B EXT	4	3	F6	366	246
B IND	5	2	E6	346	230

LDS

Load Stack Pointer

Operation: SPH ← (M)
SPL ← (M+1)

Description: Loads the more significant byte of the stack pointer from byte of memory at the address specified by the program, and loads the less significant byte of the stack pointer from the next byte of memory, at one plus the address specified by the program.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the stack pointer is set by the operation; cleared otherwise.
- Z: Set if all bits of the stack pointer are cleared by the operation; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7}.\overline{RH_6}.\overline{RH_5}.\overline{RH_4}.\overline{RH_3}.\overline{RH_2}.\overline{RH_1}.\overline{RH_0}).(\overline{RL_7}.\overline{RL_6}.\overline{RL_5}.\overline{RL_4}.\overline{RL_3}.\overline{RL_2}.\overline{RL_1}.\overline{RL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-5.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8E	216	142
DIR	4	2	9E	236	158
EXT	5	3	BE	276	190
IND	6	2	AE	256	174

Load Index Register **LDX**

Operation: IXH ← (M)
IXL ← (M + 1)

Description: Loads the more significant byte of the index register from byte of memory at the address specified by the program, and loads the less significant byte of the index register from the next byte of memory, at one plus the address specified by the program.

Condition Codes:
N: Not affected.
I: Not affected.
N: Set if the most significant bit of the index register is set by the operation; cleared otherwise.
Z: Set if all bits of the index register are cleared by the operation; cleared otherwise.
V: Cleared.
C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$
$$Z = (\overline{RH}_7.\overline{RH}_6.\overline{RH}_5.\overline{RH}_4.\overline{RH}_3.\overline{RH}_2.\overline{RH}_1.\overline{RH}_0).(\overline{RL}_7.\overline{RL}_6.\overline{RL}_5.\overline{RL}_4.\overline{RL}_3.\overline{RL}_2.\overline{RL}_1.\overline{RL}_0)$$
$$V = 0$$

Addressing Formats:

See Table A-5.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	CE	316	206
DIR	4	2	DE	336	222
EXT	5	3	FE	376	254
IND	6	2	EE	356	238

LSR

Logical Shift Right

Operation:

Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded with a zero. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes:

H: Not affected.
I: Not affected.
N: Cleared.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = 0$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = N \oplus C = [N.\overline{C}] \bigcirc [\overline{N}.C]$

(the foregoing formula assumes values of N and C after the shift operation).

$C = M_0$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	44	104	068
B	2	1	54	124	084
EXT	6	3	74	164	116
IND	7	2	64	144	100

Negate **NEG**

Operation: $ACCX \leftarrow -(ACCX) = 00 - (ACCX)$
or: $M \leftarrow -(M) = 00 - (M)$

Description: Replaces the contents of ACCX or M with its two's complement. Note that 80 is left unchanged.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there would be two's complement overflow as a result of the implied subtraction from zero; this will occur if and only if the contents of ACCX or M is 80.
- C: Set if there would be a borrow in the implied subtraction from zero; the C bit will be set in all cases except when the contents of ACCX or M is 00.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = R_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$C = R_7+R_6+R_5+R_4+R_3+R_2+R_1+R_0$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	40	100	064
B	2	1	50	120	080
EXT	6	3	70	160	112
IND	7	2	60	140	096

NOP

No Operation

Description: This is a single-word instruction which causes only the program counter to be incremented. No other registers are affected.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	01	001	001

Inclusive OR **ORA**

Operation: $ACCX \leftarrow (ACCX)_{\oplus}(M)$

Description: Perform logical "OR" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "OR" of the corresponding bits of M and of ACCX before the operation).

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = 0$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8A	212	138
A DIR	3	2	9A	232	154
A EXT	4	3	BA	272	186
A IND	5	2	AA	252	170
B IMM	2	2	CA	312	202
B DIR	3	2	DA	332	218
B EXT	4	3	FA	372	250
B IND	5	2	EA	352	234

PSH

Push Data Onto Stack

Operation: ↓ (ACCX)
SP ← (SP) − 0001

Description: The contents of ACCX is stored in the stack at the address contained in the stack pointer. The stack pointer is then decremented.

Condition Codes: Not affected.

Addressing Formats:

See Table A-4.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	36	066	054
B	4	1	37	067	055

PUL

/Pull Data from Stack

Operation: SP ← (SP) + 0001
↑ ACCX

Description: The stack pointer is incremented. The ACCX is then loaded from the stack, from the address which is contained in the stack pointer.

Condition Codes: Not affected.

Addressing Formats:

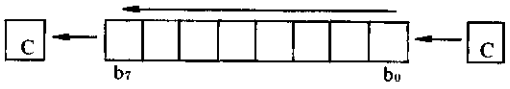
See Table A-4.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	32	062	050
B	4	1	33	063	051

Rotate Left **ROL**

Operation:



Description: Shifts all bits of ACCX or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = N \oplus C = [N.\overline{C}]_{\bigcirc}[\overline{N}.C]$

(the foregoing formula assumes values of N and C after the rotation)

$C = M_7$

Addressing Formats:

See Table A-3

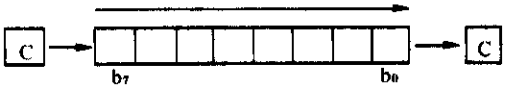
Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	49	111	073
B	2	1	59	131	089
EXT	6	3	79	171	121
IND	7	2	69	151	105

ROR

Rotate Right

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded from the C bit. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes:

H: Not affected.
I: Not affected.
N: Set if most significant bit of the result is set; cleared otherwise.
Z: Set if all bits of the result are cleared; cleared otherwise.
V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = N \oplus C = [N.\overline{C}.] \bigcirc [\overline{N}.C]$

(the foregoing formula assumes values of N and C after the rotation)

$C = M_0$

Addressing Formats:

See Table A-3

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	46	106	070
B	2	1	56	126	086
EXT	6	3	76	166	118
IND	7	2	66	146	102

RTI

Return from Interrupt

Operation:
SP ← (SP) + 0001 , ↑CC
SP ← (SP) + 0001 , ↑ACCB
SP ← (SP) + 0001 , ↑ACCA
SP ← (SP) + 0001 , ↑IXH
SP ← (SP) + 0001 , ↑IXL
SP ← (SP) + 0001 , ↑PCH
SP ← (SP) + 0001 , ↑PCL

Description: The condition codes, accumulators B and A, the index register, and the program counter, will be restored to a state pulled from the stack. Note that the interrupt mask bit will be reset if and only if the corresponding bit stored in the stack is zero.

Condition Codes: Restored to the states pulled from the stack.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	10	1	3B	073	059

Return from Interrupt

Example

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. *Before*						
PC	→	$D066	3B		RTI	
SP	→	$EFF8				
		$EFF9	11HINZVC	(binary)		
		$EFFA	12			
		$EFFB	34			
		$EFFC	56			
		$EFFD	78			
		$EFFE	55			
		$EFFF	67			
B. *After*						
PC	→	$5567	**		***	*****
		$EFF8				
		$EFF9	11HINZVC	(binary)		
		$EFFA	12			
		$EFFB	34			
		$EFFC	56			
		$EFFD	78			
		$EFFE	55			
SP	→	$EFFF	67			

CC = HINZVC (binary)
ACCB = 12 (Hex) IXH = 56 (Hex)
ACCA = 34 (Hex) IXL = 78 (Hex)

RTS

Return from Subroutine

Operation: SP ← (SP) + 0001
↑ PCH
SP ← (SP) + 0001
↑ PCL

Description: The stack pointer is incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, is loaded into the 8 bits of highest significance in the program counter. The stack pointer is again incremental (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, is loaded into the 8 bits of lowest significance in the program counter.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	5	1	39	071	057

Return from Subroutine

EXAMPLE

Example:

	Memory Location	Machine Code (Hex)	Assembler Language Label	Operator	Operand
A. *Before*					
PC	$30A2	39		RTS	
SP	$EFFD				
	$EFFE	10			
	$EFFF	02			
B. *After*					
PC	$1002	**		***	*****
	$EFFD				
	$EFFE	10			
SP	$EFFF	02			

Subtract Accumulators **SBA**

Operation: ACCA ← (ACCA) − (ACCB)

Description: Subtracts the contents of ACCB from the contents of ACCA and places the result in ACCA. The contents of ACCB are not affected.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation.
- C: Carry is set if the absolute value of accumulator B plus previous carry is larger than the absolute value of accumulator A; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = A_7.\overline{B}_7.\overline{R}_7+\overline{A}_7.B_7.R_7$$
$$C = \overline{A}_7.B_7+B_7.R_7+R_7.\overline{A}_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	10	020	016

SBC

Subtract with Carry

Operation: ACCX ← (ACCX) − (M) − (C)

Description: Subtracts the contents of M and C from the contents of ACCX and places the result in ACCX.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Carry is set if the absolute value of the contents of memory plus previous carry is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R_7}.\overline{R_6}.\overline{R_5}.\overline{R_4}.\overline{R_3}.\overline{R_2}.\overline{R_1}.\overline{R_0}$$
$$V = X_7.\overline{M_7}.\overline{R_7}+\overline{X_7}.M_7.R_7$$
$$C = \overline{X_7}.M_7+M_7.R_7+R_7.\overline{X_7}$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	82	202	130
A DIR	3	2	92	222	146
A EXT	4	3	B2	262	178
A IND	5	2	A2	242	162
B IMM	2	2	C2	302	194
B DIR	3	2	D2	322	210
B EXT	4	3	F2	362	242
B IND	5	2	E2	342	226

Set Carry **SEC**

Operation: C bit ← 1

Description: Sets the carry bit in the processor condition codes register.

Condition Codes:
H: Not affected.
I: Not affected.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Set.

Boolean Formulae for Condition Codes:
C = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0D	015	013

Set Interrupt Mask **SEI**

Operation: I bit ← 1

Description: Sets the interrupt mask bit in the processor condition codes register. The microprocessor is inhibited from servicing an interrupt from a peripheral device, and will continue with execution of the instructions of the program, until the interrupt mask bit has been cleared.

Condition Codes:
H: Not affected.
I: Set.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.

Boolean Formulae for Condition Codes:
I = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0F	017	015

SEV

Set Two's Complement Overflow Bit

Operation: V bit ← 1

Description: Sets the two's complement overflow bit in the processor condition codes register.

Condition Codes:
H: Not affected.
I: Not affected.
N: Not affected.
Z: Not affected.
V: Set.
C: Not affected.

Boolean Formulae for Condition Codes:
V = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0B	013	011

Store Accumulator **STA**

Operation: $M \leftarrow (ACCX)$

Description: Stores the contents of ACCX in memory. The contents of ACCX remains unchanged.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the contents of ACCX is set; cleared otherwise.
- Z: Set if all bits of the contents of ACCX are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = X_7$

$Z = \overline{X}_7.\overline{X}_6.\overline{X}_5.\overline{X}_4.\overline{X}_3.\overline{X}_2.\overline{X}_1.\overline{X}_0$

$V = 0$

Addressing Formats:

See Table A-2.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A DIR	4	2	97	227	151
A EXT	5	3	B7	267	183
A IND	6	2	A7	247	167
B DIR	4	2	D7	327	215
B EXT	5	3	F7	367	247
B IND	6	2	E7	347	231

STS

Store Stack Pointer

Operation: $M \leftarrow (SPH)$
$M + 1 \leftarrow (SPL)$

Description: Stores the more significant byte of the stack pointer in memory at the address specified by the program, and stores the less significant byte of the stack pointer at the next location in memory, at one plus the address specified by the program.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the stack pointer is set; cleared otherwise.
- Z: Set if all bits of the stack pointer are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$$N = SPH_7$$
$$Z = (\overline{SPH_7}.\overline{SPH_6}.\overline{SPH_5}.\overline{SPH_4}.\overline{SPH_3}.\overline{SPH_2}.\overline{SPH_1}.\overline{SPH_0})(\overline{SPL_7}.\overline{SPL_6}.\overline{SPL_5}.\overline{SPL_4}.\overline{SPL_3}.\overline{SPL_2}.\overline{SPL_1}.\overline{SPL_0})$$
$$V = 0$$

Addressing Formats:

See Table A-6.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	9F	237	159
EXT	6	3	BF	277	191
IND	7	2	AF	257	175

Store Index Register **STX**

Operation: $M \leftarrow (IXH)$
$M + 1 \leftarrow (IXL)$

Description: Stores the more significant byte of the index register in memory at the address specified by the program, and stores the less significant byte of the index register at the next location in memory, at one plus the address specified by the program.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if the most significant bite of the index register is set; cleared otherwise.
Z: Set if all bits of the index register are cleared; cleared otherwise.
V: Cleared.
C: Not affected.

Boolean Formulae for Condition Codes:

$$N = IXH_7$$
$$Z = (\overline{IXH_7}.\overline{IXH_6}.\overline{IXH_5}.\overline{IXH_4}.\overline{IXH_3}.\overline{IXH_2}.\overline{IXH_1}.\overline{IXH_0})(\overline{IXL_7}.\overline{IXL_6}.\overline{IXL_5}.\overline{IXL_4}.\overline{IXL_3}.\overline{IXL_2}.\overline{IXL_1}.\overline{IXL_0})$$
$$V = 0$$

Addressing Formats:

See Table A-6.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	DF	337	223
EXT	6	3	FF	377	255
IND	7	2	EF	357	239

SUB

Subtract

Operation: ACCX ← (ACCX) − (M)

Description: Subtracts the contents of M from the contents of ACCX and places the result in ACCX.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
- C: Carry is set if the absolute value of the contents of memory is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$
$$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$$
$$V = X_7.\overline{M}_7.\overline{R}_7+\overline{X}_7.M_7.R_7$$
$$C = \overline{X}_7.M_7+M_7.R_7+R_7.\overline{X}_7$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	80	200	128
A DIR	3	2	90	220	144
A EXT	4	3	B0	260	176
A IND	5	2	A0	240	160
B IMM	2	2	C0	300	192
B DIR	3	2	D0	320	208
B EXT	4	3	F0	360	240
B IND	5	2	E0	340	224

Software Interrupt **SWI**

Operation:

PC ← (PC) + 0001
↓ (PCL) , SP ← (SP)-0001
↓ (PCH) , SP ← (SP)-0001
↓ (IXL) , SP ← (SP)-0001
↓ (IXH) , SP ← (SP)-0001
↓ (ACCA) , SP ← (SP)-0001
↓ (ACCB) , SP ← (SP)-0001
↓ (CC) , SP ← (SP)-0001
I ← 1
PCH ← (n-0005)
PCL ← (n-0004)

Description:

The program counter is incremented (by 1). The program counter, index register, and accumulator A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

The interrupt mask bit is then set. The program counter is then loaded with the address stored in the software interrupt pointer at memory locations (n-5) and (n-4), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes:

H: Not affected.
I: Set.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.

Boolean Formula for Condition Codes:

I = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	12	1	3F	077	063

Software Interrupt

EXAMPLE

A. *Before:*

CC = HINZVC (binary)
ACCB = 12 (Hex) IXH = 56 (Hex)
ACCA = 34 (Hex) IXL = 78 (Hex)

		Memory Location	Machine Code (Hex)	Assembler Language Label	Operator	Operand
PC	→	$5566	3F		SWI	
SP	→	$EFFF				
		$FFFA	D0			
		$FFFB	55			

B. *After:*

PC	→	$D055		
SP	→	$EFF8		
		$EFF9	11HINZVC	(binary)
		$EFFA	12	
		$EFFB	34	
		$EFFC	56	
		$EFFD	78	
		$EFFE	55	
		$EFFF	67	

Note: This example assumes that FFFF is the memory location addressed when all lines of the address bus go to the high state.

Transfer from Accumulator A to Accumulator B **TAB**

Operation: ACCB ← (ACCA)

Description: Moves the contents of ACCA to ACCB. The former contents of ACCB are lost. The contents of ACCA are not affected.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the contents of the accumulator is set; cleared otherwise.
- Z: Set if all bits of the contents of the accumulator are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = 0$

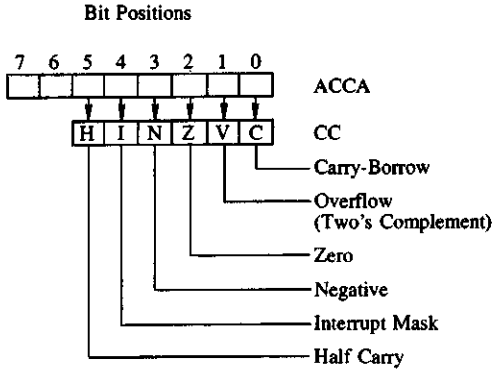
Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	16	026	022

TAP

Transfer from Accumulator A to Processor Condition Codes Register

Operation: CC ← (ACCA)



Description: Transfers the contents of bit positions 0 thru 5 of accumulator A to the corresponding bit positions of the processor condition codes register. The contents of accumulator A remain unchanged.

Condition Codes: Set or reset according to the contents of the respective bits 0 thru 5 of accumulator A.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	06	006	006

Transfer from Accumulator B to Accumulator A **TBA**

Operation: ACCA ← (ACCB)

Description: Moves the contents of ACCB to ACCA. The former contents of ACCA are lost. The contents of ACCB are not affected.

Condition Codes:
- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the contents of the accumulator is set; cleared otherwise.
- Z: Set if all bits of the contents of the accumulator are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$N = R_7$

$Z = \overline{R}_7.\overline{R}_6.\overline{R}_5.\overline{R}_4.\overline{R}_3.\overline{R}_2.\overline{R}_1.\overline{R}_0$

$V = 0$

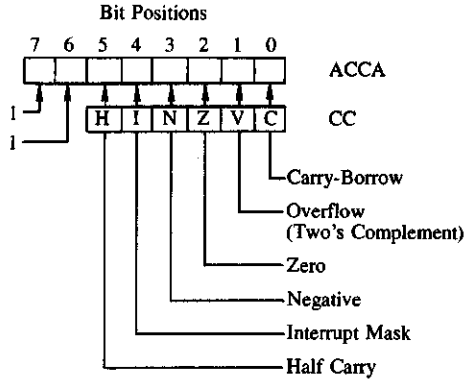
Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	17	027	023

TPA

Transfer from Processor Condition Codes Register to Accumulator A

Operation: ACCA ← (CC)



Description: Transfers the contents of the processor condition codes register to corresponding bit positions 0 thru 5 of accumulator A. Bit positions 6 and 7 of accumulator A are set (i.e. go to the "1" state). The processor condition codes register remains unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	07	007	007

Test

TST

Operation: (ACCX) − 00
(M) − 00

Description: Set condition codes N and Z according to the contents of ACCX or M.

Condition Codes:
H: Not affected.
I: Not affected.
N: Set if most significant bit of the contents of ACCX or M is set; cleared otherwise.
Z: Set if all bits of the contents of ACCX or M are cleared; cleared otherwise.
V: Cleared.
C: Cleared.

Boolean Formulae for Condition Codes:

$N = M_7$
$Z = \overline{M}_7.\overline{M}_6.\overline{M}_5.\overline{M}_4.\overline{M}_3.\overline{M}_2.\overline{M}_1.\overline{M}_0$
$V = 0$
$C = 0$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4D	115	077
B	2	1	5D	135	093
EXT	6	3	7D	175	125
IND	7	2	6D	155	109

TSX — Transfer from Stack Pointer to Index Register

Operation: IX ← (SP) + 0001

Description: Loads the index register with one plus the contents of the stack pointer. The contents of the stack pointer remains unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	30	060	048

TXS — Transfer From Index Register to Stack Pointer

Operation: SP ← (IX) − 0001

Description: Loads the stack pointer with the contents of the index register, minus one. The contents of the index register remains unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	35	065	053

Wait for Interrupt **WAI**

Operation:

PC ← (PC) + 0001
↓ (PCL) , SP ← (SP)-0001
↓ (PCH) , SP ← (SP)-0001
↓ (IXL) , SP ← (SP)-0001
↓ (IXH) , SP ← (SP)-0001
↓ (ACCA) , SP ← (SP)-0001
↓ (ACCB) , SP ← (SP)-0001
↓ (CC) , SP ← (SP)-0001

Condition Codes: Not affected.

Description: The program counter is incremented (by 1). The program counter, index register, and accumulators A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

Execution of the program is then suspended until an interrupt from a peripheral device is signalled, by the interrupt request control input going to a low state.

When an interrupt is signalled on the interrupt request line, and provided the I bit is clear, execution proceeds as follows. The interrupt mask bit is set. The program counter is then loaded with the address stored in the internal interrupt pointer at memory locations (n-7) and (n-6), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes:

H: Not affected.
I: Not affected until an interrupt request signal is detected on the interrupt request control line. When the interrupt request is received the I bit is set and further execution takes place, provided the I bit was initially clear.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/octal/decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	9	1	3E	076	062

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
IMMediate	CCC A #Number CCC A #symbol CCC A #expression CCC A #'C	CCC B #Number CCC B #symbol CCC B #expression CCC B #'C
DIRect or EXTended	CCC A Number CCC A symbol CCC A expression	CCC B Number CCC B symbol CCC B expression
INDexed	CCC A X CCC Z ,X CCC A Number,X CCC A symbol,X CCC A expression,X	CCC B X CCC B ,X CCC B Number,X CCC B symbol,X CCC B expression,X

Notes: 1. CCC = Mnemonic operator of source instruction
2. "symbol" may be the special symbol "*"
3. "expression" may contain the special symbol "*"
4. space may be omitted before A or B

Applicable to the Following Source Instructions:

ADC ADD AND BIT CMP
EOR LDA ORA SBC SUB

*Special symbol indicating program-counter

TABLE A-1. Addressing Formats (1)

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
DIRect or EXTended	STA A Number STA A symbol STA A expression	STA B Number STA B symbol STA B expression
INDexed	STA A X STA A ,X STA A Number,X STA A symbol,X STA A expression,X	STA B X STA B ,X STA B Number,X STA B symbol,X STA B expression,X

Notes: 1. "symbol" may be the special symbol "*".
2. "expression" may contain the special symbol "*".
3. space may be omitted before A or B.

Applicable to the Source Instruction:

STA

*Special symbol indicating program-counter

TABLE A-2. Addressing Formats (2)

Operand or Addressing Mode	Formats
Accumulator A	CCC A
Accumulator B	CCC B
EXTended	CCC Number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC Number,X CCC symbol,X CCC expression,X

Notes: 1. CCC + Mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".
4. space may be omitted before A or B

Applicable to the Following Source Instructions:

ASL ASR CLR COM DEC INC
LSR NEG ROL ROR TST

*Special symbol indicating program-counter

TABLE A-3. Addressing Formats (3)

Operand	Formats
Accumulator A	CCC A
Accumulator B	CCC B

Notes: 1. CCC = Mnemonic operator of source instruction
2. space may be omitted before A or B

Applicable to the Following Source Instructions:

PSH PUL

TABLE A-4. Addressing Formats (4)

Addressing Mode	Formats
IMMediate	CCC #number CCC #symbol CCC #expression CCC #'C
DIRect or EXTended	CCC Number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCCNumber,X CCC symbol,X CCC expression,X

Notes: 1. CCC = Mnemonic operator of source instruction
2. "symbol" may be the special symbol "*"
3. "expression" may contain the special symbol "*"

Applicable to the Following Source Instructions:

CPX LDS LDX

*Special symbol indicating program-counter

TABLE A-5. Addressing Formats (5)

Addressing Mode	Formats
DIRect or EXTended	CCC N CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC Number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = Mnemonic operator of source instruction
2. "symbol" may be the special symbol "*"
3. "expression" may contain the spcial symbol "*"

Applicable to the Following Source Instructions:

STS STX

*Special symbol indicating program-counter

TABLE A-6. Addressing Formats (6)

Addressing Mode	Formats
EXTended	CCC Number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC Number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = Mnemonic operator of source instruction
2. "symbol" may be the special symbol "*"
3. "expression" may contain the special symbol "*"

Applicable to the Following Source Instructions:

JMP JSR

*Special symbol indicating program-counter

TABLE A-7. Addressing Formats (7)

Addressing Mode	Formats
RELative	CCC Number CCC symbol CCC expression

Notes: 1. CCC = Mnemonic operator of source instruction
2. "symbol" may be the special symbol "*"
3. "expression" may contain the special symbol "*"

Applicable to the Following Source Instructions:

BCC BCS BEQ BGE BGT BHI BLE BLS
BLT BMI BNE BPL BRA BSR BVC BVS

*Special symbol indicating program-counter

TABLE A-8. Addressing Formats (8)

appendix B

Assembler Directives

APPENDIX B

Definition of the Assembler Directives

Alphabetic List of Assembler Directives

END	End of program
EQU	Equate symbol
FCB	Form Constant byte
FCC	Form Constant Characters
FDB	Form Double Constant Byte
MON	Return to Monitor
NAM	Name program
OPT	Option
ORG	Origin
PAGE	Advance Listing to top of page
RMB	Reserve Memory Bytes
SPC	Space n lines

END - End of Program

When the assembler directive "END", is used, it marks the end of a source program and can be followed only by a statement containing the assembler directive "MON" or another program.

The operator in the last statement of a source program must be either "END" or "MON". If the program ends with a "MON" directive, the use of "END" is optional.

The "END" directive must not be written with a label, and it does not have an operand.

The "END" directive is not translated into object code.

EQU - Equate Symbol

The "EQU" directive is used to assign a value to a symbol. The "EQU" statement must contain a label which is identical with the symbol being defined. The operand field may contain the numerical value of the symbol (decimal, hexadecimal, octal, or binary). Alternatively, the operand field may be another symbol or an expression which can be evaluated by the assembler. The EQU statement is not translated into object code.

The following are examples of valid "EQU" statements:

Location	Data	Label	Operator	Operand
	0A01	SUN	EQU	$A01
	0003	AB	EQU	3
	0A01	AA	EQU	SUN
	0A04	AC	EQU	AB+AA
	0FC1	ABC	EQU	$FC1

Relating to the use of a symbol or an expression in the operand field, only one level of forward referencing will assemble correctly. This reflects a two-pass characteristic of the assembly process. An (illegal) example of two levels of forward referencing would be:

```
E       EQU     Y
Y       EQU     C
C       EQU     5
```

This will not assemble correctly because E will not be assigned a numerical value at the end of pass 2. E and Y are both undefined throughout pass 1. E is undefined throughout pass 2 and will cause an error message.

FCB - Form Constant Byte

The "FCB" directive may have one or more operands, separated by commas. An 8-bit unsigned binary number, corresponding to the value of each operand is stored in a byte of the object program. If there is more than one operand, they are stored in successive bytes. The operand field may contain the actual value (decimal, hexadecimal, octal, or binary). Alternatively, the operand may be a symbol or an expression which can be assigned a numerical value by the assembler.

An "FCB" directive followed by one or more void operands separated by commas will store zeros for the void operands.

An "FCB" directive may be written with a label.

Examples of valid "FCB" directives follow:

```
Location    Data    Label    Operator    Operand

0000        FF      TOP      FCB         $FF
0001        00      TAB      FCB         ,$F,23,
0002        0F
0003        17
0004        00
0005        E5               FCB         *+$E0
```

FCC - Form Constant Characters

The "FCC" directive translates strings of characters into their 7-bit ASCII codes. Any of the characters which correspond to ASCII hexadecimal codes 20 (SP) thru 5F (_) can be processed by this directive.

1. Count, comma, text. Where the count specifies how many ASCII characters to generate and the text begins following the first comma of the operand. Should the count be longer than the text, spaces will be inserted to fill the count. Maximum count is 255.

2. Text enclosed between identical delimiters, each being any single character. (If the delimiters are numbers, the text must not begin with a comma.)

If the string in the operand comprises more than one character, the ASCII codes corresponding to the successive characters are entered into successive bytes of memory.

An "FCC" directive may be written with a label.

The following are examples of valid "FCC" directives:

Location	Data	Label	Operator	Operand
0A00	54	MSG1	FCC	/TEXT/
0A01	45			
0A02	58			
0A03	54			
0A04	54	MSG2	FCC	9,TEXT
0A05	45			
0A06	58			
0A07	54			
0A08	20			
0A09	20			
0A0A	20			
0A0B	20			
0A0C	20			

FDB - Form Double Constant Byte

The "FDB" directive may have one or more operands separated by commas. The 16-bit unsigned binary number, corresponding to the value of each operand is stored in two bytes of the object program. If there is more than one operand, they are stored in successive bytes. The operand field may contain the actual value (decimal, hexadecimal, octal, or binary). Alternatively, the operand may be a symbol or an expression which can be assigned a numerical value by the assembler.

An "FDB" directive followed by one or more void operands separated by commas will store zeros for the void operands.

An "FDB" directive may be written with a label.

Examples of valid "FDB" directives follow:

Location	Data	Label	Operator	Operand
0010	0002	TWO	FDB	2
0012	0000	MASK	FDB	,$F,$EF,,$AFF
0014	000F			
0016	00EF			
0018	0000			
001A	0AFF			

MON - Return to Monitor

The assembler directive "MON", if used, must be in the last statement of a source program. (See assembler directive "END" above.) The "MON" directive instructs the assembler that the source program just completed is the last to be assembled, and it returns control to the 680b PROM Monitor.

The last statement of a source program must contain either "END" or "MON".

The assembler directive "MON" must not be written with a label, and no operand is used.

The "MON" directive is not translated into object code.

NAM - Name

The "NAM" (or NAME) directive names the program, or provides the top of page heading text meaningful to users of the assembly.

The "NAM" directive must not be written with a label. The "NAM" directive cannot distinguish the operand field from the comment field. Both the operand field and the comment field are treated as continuous text.

No object code results from the "NAM" directive.

OPT - Option

The "OPT" directive is used to give the programmer optional control of the format of assembler output. The details of the "OPT" directive depend on the version of the 680b Resident Assembler being used. When the Assembler becomes available, details of the "OPT" directive will be included in the documentation.

ORG - Origin

The assembler directive "ORG" defines the numerical address of the first byte of machine code which results from the assembly of the immediately subsequent section of a source program. There may be any number of "ORG" statements in a program. The "ORG" directive sets the program counter to the value expressed in the operand field.

The operand field may contain the actual value (decimal, hexadecimal, octal, or binary) to which the program counter is to be set. Alternatively, the operand field may contain a symbol or an expression which can be assigned a numerical value by the assembler.

The location counter is initialized before each assembly. If no "ORG" statement appears at the beginning of the program, the location counter will begin as if an "ORG" zero had been entered.

An "ORG" directive must not be written with a label.

The ORG statement does not translate into object code.

The following are examples of valid ORG statements:

```
Location    Data    Label     Operator    Operand

0064                (blank)   ORG         100
AF23                (blank)   ORG         $AF23
            1100    BEGIN     EQU         $1100
1100                (blank)   ORG         BEGIN
```

PAGE - Advance Paper to Top of Next Page

The "PAGE" directive causes the Assembler to advance the paper to the top of the next page. The PAGE directive does not appear on the program listing. No label or operand is used, and no machine code results.

RMB - Reserve Memory Bytes

The "RMB" directive causes the location counter to be increased by the value of the operand field. This reserves a block of memory whose length is equal to the value of the operand field. The operand field may contain the actual number (decimal, hexadecimal, octal or binary) equal to the number of bytes to be reserved. Alternatively, the operand may be a symbol or an expression which can be assigned a numerical value by the assembler.

The block of memory which is reserved by the "RMB" directive is unchanged by that directive.

The "RMB" directive may be written with a label.

Examples of valid "RMB" directives follow (the data column indicates the number of bytes being reserved):

```
Location    Data    Label     Operator    Operand

0100        0004              RMB         4
0104        0014    TABLE 1   RMB         20
0118        0014    TABLE 2   RMB         20
```

SPC - Space N Lines

The "SPC" directive provides n vertical spaces for formatting the program listing. It does not itself appear in the listing. The number of lines to be left blank is stated by an operand in the operand field.

The operand would normally contain the actual number (decimal, hexadecimal, octal, or binary) equal to the number of lines to be left blank. A symbol or an expression is also allowed.

The "SPC" directive must not be written with a label.

When the "SPC" directive causes the listing to cross page boundries only those blank lines required to get to the top of the next page will be generated.

appendix C

Input/Output Information

APPENDIX C, INPUT/OUTPUT INFORMATION

ACIA

The 680b is supplied with an Asynchronous Communications Interface Adapter (ACIA) for the purpose of handling serial input and output operations. Initialization and control of this I/O port is usually handled by system software such as the PROM Monitor.

The following information concerning the ACIA registers is included for those who wish to do their own initialization and I/O handling.

ACIA Registers

Transmit Data Register (TDR)

Writing data into the Transmit Data Register causes the Transmit Data Register Empty bit in the Status Register to go low. Data can then be transmitted. If the transmitter is idling and no character is being transmitted, then the transfer will take place within one bit time of the trailing edge of the Write command. If a character is being transmitted, the new data character will commence as soon as the previous character is complete. The transfer of data causes the Transmit Data Register Empty (TDRE) bit to indicate empty.

Receive Data Register (RDR)

Data is automatically transferred to the empty Receive Data Register (RDR) from the receiver deserializer (a shift register) upon receiving a complete character. This event causes the Receive Data Register Full bit (RDRF) in the status buffer to go high (full). Data may then be read through the bus by addressing the ACIA and selecting the Receive Data Register with RS and R/W high when the ACIA is enabled. The nondestructive read cycle causes the RDRF bit to be cleared to empty although the data is retained in the RDR. The status is maintained by RDRF as to whether or not the data is current. When the Receive Data Register is full, the automatic transfer of data from the Receiver Shift Register to the Data Register is inhibited and the RDR contents remain valid with its current status stored in the Status Register.

Control Register

The ACIA Control Register consists of eight bits of write-only buffer that are selected when RS and R/W are low. This register controls the function of the receiver, transmitter, interrupt enables, and the Request-to-Send peripheral/modem control output.

Counter Divide Select Bits (CR0 and CR1) - The Counter Divide Select Bits (CR0 and CR1) determine the divide ratios utilized in both the transmitter and receiver sections of the ACIA. Additionally, these bits are used to provide a master reset for the ACIA which clears the Status Register (except for external conditions on $\overline{CTS}$ and $\overline{DCD}$) and initializes both the receiver and transmitter. Master reset does not affect other Control Register bits. Note that after power-on or a power fail/restart, these bits must be set high to reset the ACIA. After resetting, the clock divide ratio may be selected. These counter select bits provide for the following clock divide ratios:

CR1	CR0	Function
0	0	÷1
0	1	÷16
1	0	÷64
1	1	Master Reset

Word Select Bits (CR2, C3, and CR4) - The Word Select bits are used to select word length, parity, and the number of stop bits. The encoding format is as follows:

CR4	CR3	CR2	Function
0	0	0	7 Bits + Even Parity + 2 Stop Bits
0	0	1	7 Bits + Odd Parity + 2 Stop Bits
0	1	0	7 Bits + Even Parity + 1 Stop Bit
0	1	1	7 Bits + Odd Parity + 1 Stop Bit
1	0	0	8 Bits + 2 Stop Bits
1	0	1	8 Bits + 1 Stop Bit
1	1	0	8 Bits + Even Parity + 1 Stop Bit
1	1	1	8 Bits + Odd Parity + 1 Stop Bit

Word length, Parity Select, and Stop Bit changes are not buffered and, therefore, become effective immediately.

Transmitter Control Bits (CR5 and CR6) - Two Transmitter Control bits provide for the control of the interrupt from the Transmit Data Register Empty condition, the Request-to-Send output, and the transmission of a Break level (space). The following encoding format is used:

CR6	CR5	Function
0	0	$\overline{RTS}$ = low, Transmitting Interrupt Disabled.
0	1	$\overline{RTS}$ = low, Transmitting Interrupt Enabled.
1	0	$\overline{RTS}$ = high, Transmitting Interrupt Disabled.
1	1	$\overline{RTS}$ = low, Transmits a Break level on the Transmit Data Output, Transmitting Interrupt Disabled.

Receive Interrupt Enable Bit (CR7) - Interrupts will be enabled by a high level in bit position 7 of the Control Register (CR7). Interrupts from the receiver section, Receive Data Register Full being high or by a low to high transition on the Data Carrier Detect signal line, are enabled or disabled by the Receive Interrupt Enable Bit.

Status Register

Information on the status of the ACIA is available to the MPU by reading the ACIA Status Register. This read-only register is selected when RS is low and R/W is high. Information stored in this register indicates the status of the Transmit Data Register, the Receive Data Register and error logic, and the peripheral/modem status inputs of the ACIA.

Receive Data Register Full (RDRF), Bit 0 - Receive Data Register Full indicates that received data has been transferred to the Receive Data Register. RDRF is cleared after an MPU read of the Receive Data Register or by a master reset. The cleared or empty state indicates that the contents of the Receive Data Register are not current. Data Carrier Detect being high also causes RDRF to indicate empty.

Transmit Data Register Empty (TDRE), Bit 1 - The Transmit Data Register Empty bit being set high indicates that the Transmit Data Register contents have been transferred and that new data may be entered. The low state indicates that the register is full and that transmission of a new character has not begun since the last write data command.

Data Carrier Detect (DCD), Bit 2 - The Data Carrier Detect bit will be high when the DCD input from a modem has gone high to indicate that a carrier is not present. This bit going high causes an Interrupt Request to be generated when the Receive Interrupt Enable is set. It remains high after the DCD input is returned low until cleared by first reading the Status Register and then the Data Register or until a master reset occurs. If the DCD input remains high after read status and read data or master reset have occurred, the DCD status bit remains high and will follow the DCD input.

Clear-to-Send (CTS), Bit 3 - The Clear-to-Send bit indicates the state of the Clear-to-Send input from a modem. A low CTS indicates that there is a Clear-to-Send from the modem. In the high state, the Transmit Data Register Empty bit is inhibited and the Clear-to-Send status bit will be high. Master reset does not affect the Clear-to-Send Status bit.

Framing Error (FE), Bit 4 - Framing error indicates that the received character is improperly framed by a start and a stop bit and is detected by the absence of the 1st stop bit. This error indicates a synchronization error, faulty transmission, or a break condition. The framing error flag is set or reset during the receive data transfer time. Therefore, this error indicator is present throughout the time that the associated character is available.

Receiver Overrun (OVRN), Bit 5 - Overrun is an error flag that indicates that one or more characters in the data stream were lost. That is, a character or a number of characters were received but not read from the Receive Data Register (RDR) prior to subsequent characters being received. The overrun condition begins at the midpoint of the last bit of the second character received in succession without a read of the RDR having occurred. The Overrun does not occur in the Status Register until the valid character prior to Overrun has been read. The RDRF bit remains set until the Overrun is reset. Character synchronization is maintained during the Overrun condition. The Overrun indication is reset after the reading of data from the Receive Data Register. Overrun is also reset by the Master Reset.

Parity Error (PE), Bit 6 - The parity error flag indicates that the number of highs (ones) in the character does not agree with the preselected odd or even parity. Odd parity is defined to be when the total number of ones is odd. The parity error indication will be present as long as the data character is in the RDR. If no parity is selected, then both the transmitter parity generator output and the receiver parity check results are inhibited.

Interrupt Request (IRQ), Bit 7 - The IRQ bit indicates the state of the $\overline{IRQ}$ output. Any interrupt condition with its applicable enable will be indicated in this status bit. Anytime the $\overline{IRQ}$ output is low, the IRQ bit will be high to indicate the interrupt or service request status.

Paper Tape Reader Control

When the paper tape reader control circuit is used, the $\overline{RTS}$ output of the ACIA turns the reader on and off. When $\overline{RTS}$ is high, the reader will be on. When $\overline{RTS}$ is low, the reader will be off. Therefore, the reader is turned on when CR6 is 1 and CR5 is 0. This also turns off input interrupts. (See ACIA Control Register above.) The reader is off for the three other possible combinations of CR6 and CR5.

Interrupt Vectors

The processor interrupt vectors are located in locations FFF8 through FFFF within the 680b PROM Monitor. The contents of the interrupt vectors depends on the version of the Monitor being used. Refer to Section VI of the System Monitor Manual for further information.

2450 Alamo SE
Albuquerque, NM 87106