

DUNFIELD DEVELOPMENT SYSTEMS  
P.O. BOX 31044  
NEPEAN, ONT. CANADA  
K2B 8S8

**M6809PM (AD)**

# **MC6809-MC6809E**

## **8-BIT MICROPROCESSOR**

### **PROGRAMMING MANUAL**

**Original Issue: March 1, 1981**

# TABLE OF CONTENTS

Paragraph No.

Title

Page No.

## SECTION 1 GENERAL DESCRIPTION

1.1	Introduction.....	1-1
1.2	Features.....	1-1
1.3	Software Features.....	1-2
1.4	Programming Model.....	1-3
1.5	Index Registers (X, Y) .....	1-3
1.6	Stack Pointer Registers (U, S) .....	1-3
1.7	Program Counter (PC) .....	1-4
1.8	Accumulator Registers (A, B, D) .....	1-4
1.9	Direct Page Register (DP) .....	1-4
1.10	Condition Code Register (CC) .....	1-4
1.10.1	Condition Code Bits .....	1-5
1.10.1.1	Half Carry (H), Bit 5.....	1-5
1.10.1.2	Negative (N), Bit 3.....	1-5
1.10.1.3	Zero (Z), Bit 2.....	1-5
1.10.1.4	Overflow (V), Bit 1 .....	1-5
1.10.1.5	Carry (C), Bit 0 .....	1-5
1.10.2	Interrupt Mask Bits and Stacking Indicator .....	1-5
1.10.2.1	Fast Interrupt Request Mask (F), Bit 6 .....	1-5
1.10.2.2	Interrupt Request Mask (I), Bit 4 .....	1-5
1.10.2.3	Entire Flag (E), Bit 7 .....	1-6
1.11	Pin Assignments and Signal Description.....	1-6
1.11.1	MC6809 Clocks .....	1-6
1.11.1.1	Oscillator (EXTAL, XTAL).....	1-6
1.11.1.2	Enable (E).....	1-7
1.11.1.3	Quadrature (Q) .....	1-7
1.11.2	MC6809E Clocks (E and Q) .....	1-7
1.11.3	Three State Control (TSC) (MC6809E) .....	1-7
1.11.4	Last Instruction Cycle (LIC) (MC6809E) .....	1-7
1.11.5	Address Bus (A0-A15).....	1-7
1.11.6	Data Bus (D0-D7) .....	1-7
1.11.7	Read/Write (R/W).....	1-8
1.11.8	Processor State Indicators (BA, BS) .....	1-8
1.11.8.1	Normal .....	1-8
1.11.8.2	Interrupt or Reset Acknowledge.....	1-8
1.11.8.3	Sync Acknowledge.....	1-8

## TABLE OF CONTENTS (CONTINUED)

Paragraph No.	Title	Page No.
1.11.8.4	Halt/Bus Grant .....	1-8
1.11.9	Reset (RESET) .....	1-9
1.11.10	Interrupts .....	1-9
1.11.10.1	Non-Maskable Interrupt (NMI) .....	1-9
1.11.10.2	Fast Interrupt Request (FIRQ) .....	1-9
1.11.10.3	Interrupt Request (IRQ) .....	1-9
1.11.11	Memory Ready (MRDY) (MC6809) .....	1-9
1.11.12	Advanced Valid Memory Address (AVMA) (MC6809E) .....	1-10
1.11.13	Halt (HALT) .....	1-10
1.11.14	Direct Memory Access/Bus Request (DMA/BREQ) (MC6809) .....	1-10
1.11.15	Busy (MC6809E) .....	1-10
1.11.16	Power .....	1-11

## SECTION 2 ADDRESSING MODES

2.1	Introduction .....	2-1
2.2	Addressing Modes .....	2-1
2.2.1	Inherent .....	2-1
2.2.2	Immediate .....	2-1
2.2.3	Extended .....	2-2
2.2.4	Direct .....	2-2
2.2.5	Indexed .....	2-2
2.2.5.1	Constant Offset from Register .....	2-2
2.2.5.2	Accumulator Offset from Register .....	2-3
2.2.5.3	Autoincrement/Decrement from Register .....	2-3
2.2.5.4	Indirection .....	2-4
2.2.5.5	Extended Indirect .....	2-4
2.2.5.6	Program Counter Relative .....	2-4
2.2.6	Branch Relative .....	2-4

## SECTION 3 INTERRUPT CAPABILITIES

3.1	Introduction .....	3-1
3.2	Non-Maskable Interrupt (NMI) .....	3-1
3.3	Fast Maskable Interrupt Request (FIRQ) .....	3-2
3.4	Normal Maskable Interrupt Request (IRQ) .....	3-2
3.5	Software Interrupts (SWI, SWI2, SWI3) .....	3-2

## TABLE OF CONTENTS (CONCLUDED)

Paragraph No.

Title

Page No.

### SECTION 4 PROGRAMMING

4.1	Introduction.....	4-1
4.1.1	Position-Independence.....	4-1
4.1.2	Modular Programming.....	4-1
4.1.2.1	Local Storage.....	4-1
4.1.2.2	Global Storage.....	4-2
4.1.3	Reentrancy/Recursion.....	4-2
4.2	M6809 Capabilities.....	4-2
4.2.1	Module Construction.....	4-2
4.2.1.1	Parameters.....	4-3
4.2.1.2	Local Storage.....	4-3
4.2.1.3	Global Storage.....	4-3
4.2.2	Position-Independent Code.....	4-4
4.2.3	Reentrant Programs.....	4-5
4.2.4	Recursive Programs.....	4-5
4.2.5	Loops.....	4-5
4.2.6	Stack Programming.....	4-6
4.2.6.1	M6809 Stacking Operations.....	4-6
4.2.6.2	Subroutine Linkage.....	4-7
4.2.6.3	Software Stacks.....	4-8
4.2.7	Real Time Programming.....	4-8
4.3	Program Documentation.....	4-8
4.4	Instruction Set.....	4-9

### APPENDIX A INSTRUCTION SET DETAILS

A.1	Introduction.....	A-1
A.2	Notation.....	A-1
	Instructions (listed in alphabetical order).....	A-3

### APPENDIX B ASSIST09 MONITOR PROGRAM

B.1	General Description.....	B-1
B.2	Implementation Requirements.....	B-1
B.3	Interrupt Control.....	B-2
B.4	Initialization.....	B-3

## TABLE OF CONTENTS (CONTINUED)

Paragraph No.	Title	Page No.
B.5	Input/Output Control .....	B-4
B.6	Command Format .....	B-4
B.7	Command List .....	B-5
B.8	Commands.....	B-5
	Breakpoint .....	B-6
	Call .....	B-6
	Display.....	B-7
	Encode .....	B-7
	Go .....	B-8
	Load .....	B-8
	Memory .....	B-9
	Null .....	B-10
	Offset.....	B-10
	Punch.....	B-11
	Register.....	B-11
	Stlevel.....	B-12
	Trace.....	B-12
	Verify .....	B-13
	Window .....	B-13
B.9	Services.....	B-14
	BKPT .....	B-15
	INCHP.....	B-15
	MONTR.....	B-16
	OUTCH .....	B-17
	OUT2HS.....	B-17
	OUT4HS.....	B-18
	PAUSE .....	B-18
	PCRLF .....	B-19
	PDATA .....	B-19
	PDATA1 .....	B-20
	SPACE .....	B-21
	VTRSW .....	B-21
B.10	Vector Swap Service.....	B-22
	.ACIA.....	B-23
	.AVTBL.....	B-23
	.BSDTA .....	B-24
	.BSOFF .....	B-24
	.BSON .....	B-25
	.CIDTA .....	B-25
	.CIOFF .....	B-26
	.CION .....	B-26
	.CMDL1.....	B-27
	.CMDL2.....	B-28

## TABLE OF CONTENTS (CONTINUED)

<i>Paragraph No.</i>	<i>Title</i>	<i>Page No.</i>
	.CODTA.....	B-28
	.COOFF.....	B-29
	.COON.....	B-29
	.ECHO.....	B-30
	.FIRQ.....	B-30
	.HSDATA.....	B-31
	.IRQ.....	B-31
	.NMI.....	B-32
	.PAD.....	B-32
	.PAUSE.....	B-33
	.PTM.....	B-33
	.RESET.....	B-34
	.RSVD.....	B-34
	.SWI.....	B-35
	.SWI2.....	B-35
	.SWI3.....	B-36
B.11	Monitor Listing.....	B-37

## APPENDIX C MACHINE CODE TO INSTRUCTION CROSS REFERENCE

C.1	Introduction .....	C-1
-----	--------------------	-----

## APPENDIX D PROGRAMMING AID

D.1	Introduction .....	D-1
-----	--------------------	-----

## APPENDIX E ASCII CHARACTER SET

E.1	Introduction .....	E-1
E.2	Character Representation and Code Identification .....	E-1
E.3	Control Characters .....	E-2
E.4	Graphic Characters .....	E-2

## TABLE OF CONTENTS (CONTINUED)

<i>Paragraph No.</i>	<i>Title</i>	<i>Page No.</i>
----------------------	--------------	-----------------

### APPENDIX F OPCODE MAP

F.1	Introduction .....	F-1
F.2	Opcode Map.....	F-1

### APPENDIX G PIN ASSIGNMENTS

G.1	Introduction .....	G-1
-----	--------------------	-----

### APPENDIX H CONVERSION TABLES

H.1	Introduction .....	H-1
H.2	Powers of 2; Powers of 16 .....	H-1
H.3	Hexadecimal and Decimal Conversion .....	H-2
H.3.1	Converting Hexadecimal to Decimal .....	H-2
H.3.2	Converting Decimal to Hexadecimal .....	H-2

## LIST OF ILLUSTRATIONS

<i>Figure No.</i>	<i>Title</i>	<i>Page No.</i>
1-1	Programming Model .....	1-3
1-2	Condition Code Register .....	1-4
1-3	Processor Pin Assignments .....	1-6
2-1	Postbyte Usage for EXG/TFR, PSH/PUL Instructions .....	2-2
3-1	Interrupt Processing Flowchart .....	3-5
4-1	Stacking Order .....	4-7
B-1	Memory Map .....	B-2
E-1	ASCII Character Set .....	E-1
G-1	Pin Assignments .....	G-1

## LIST OF TABLES

<i>Table No.</i>	<i>Title</i>	<i>Page No.</i>
1-1	BA/BS Signal Encoding.....	1-8
2-1	Postbyte Usage for Indexed Addressing Modes .....	2-3
3-1	Interrupt Vector Locations .....	3-1
4-1	Instruction Set .....	4-9
4-2	8-Bit Accumulator and Memory Instructions.....	4-11
4-3	16-Bit Accumulator and Memory Instructions.....	4-12
4-4	Index/Stack Pointer Instructions.....	4-12
4-5	Branch Instructions .....	4-13
4-6	Miscellaneous Instructions .....	4-13
A-1	Operation Notation.....	A-1
A-2	Register Notation.....	A-2
B-1	Command List .....	B-5
B-2	Services.....	B-14
B-3	Vector Table Entries .....	B-22
C-1	Machine Code to Instruction Cross Reference.....	C-2
D-1	Programming Aid .....	D-1
E-1	Control Characters .....	E-2
E-2	Graphic Characters .....	E-3
F-1	Opcode Map.....	F-2
F-2	Indexed Addressing Mode Data .....	F-3
H-1	Powers of 2; Powers of 16.....	H-1
H-2	Hexadecimal and Decimal Conversion Chart .....	H-2



## **SECTION 1 GENERAL DESCRIPTION**

### **1.1 INTRODUCTION**

This section contains a general description of the Motorola MC6809 and MC6809E Microprocessor Units (MPU). Pin assignments and a brief description of each input/output signal are also given. The term MPU, processor, or M6809 will be used throughout this manual to refer to both the MC6809 and MC6809E processors. When a topic relates to only one of the processors, that specific designator (MC6809 or MC6809E) will be used.

### **1.2 FEATURES**

The MC6809 and MC6809E microprocessors are greatly enhanced, upward compatible, computationally faster extensions of the MC6800 microprocessor.

Enhancements such as additional registers (a Y index register, a U stack pointer, and a direct page register) and instructions (such as MUL) simplify software design. Improved addressing modes have also been implemented.

Upward compatibility is guaranteed as MC6800 assembly language programs may be assembled using the Motorola MC6809 Macro Assembler. This code, while not as compact as native M6809 code, is, in most cases, 100% functional.

Both address and data are available from the processor earlier in an instruction cycle than from the MC6800 which simplifies hardware design. Two clock signals, E (the MC6800  $\phi 2$ ) and a new quadrature clock Q (which leads E by one-quarter cycle) also simplify hardware design.

A memory ready (MRDY) input is provided on the MC6809 for working with slow memories. This input stretches both the processor internal cycle and direct memory access bus cycle times but allows internal operations to continue at full speed. A direct memory access request ( $\overline{\text{DMA/BREQ}}$ ) input is provided for immediate memory access or dynamic memory refresh operations; this input halts the internal MC6809 clocks. Because the processor's registers are dynamic, an internal counter periodically recovers the bus from direct memory access operations and performs a true processor refresh cycle to allow unlimited length direct memory access operation. An interrupt acknowledge signal is available to allow development of vectoring by interrupt device hardware or detection of operating system calls.

Three prioritized, vectored, hardware interrupt levels are available: non-maskable, fast, and normal. The highest and lowest priority interrupts, non-maskable and interrupt request respectively, are the normal interrupts used in the M6800 family. A new interrupt on this processor is the fast interrupt request which provides faster service to its interrupt input by only stacking the program counter and condition code register and then servicing the interrupt.

Modern programming techniques such as position-independent, system independent, and reentrant programming are readily supported by these processors.

A Memory Management Unit (MMU), the MC6829, allows a M6809 based system to address a two megabyte memory space. Note: An arbitrary number of tasks may be supported — slower — with software.

This advanced family of processors is compatible with all M6800 peripheral parts.

### **1.3 SOFTWARE FEATURES**

Some of the software features of these processors are itemized in the following paragraphs. Programs developed for the MC6800 can be easily converted for use with the MC6809 or MC6809E by running the source code through a M6809 Macro Assembler or any one of the many cross assemblers that are available.

The addressing modes of any microprocessor provide it with the capability to efficiently address memory to obtain data and instructions. The MC6809 and MC6809E have a versatile set of addressing modes which allow them to function using modern programming techniques.

The addressing modes and instructions of the MC6809 and MC6809E are upward compatible with the MC6800. The old addressing modes have been retained and many new ones have been added.

A direct page register has been added which allows a 256 byte "direct" page anywhere in the 64K logical address space. The direct page register is used to hold the most-significant byte of the address used in direct addressing and decrease the time required for address calculation.

Branch relative addressing to anywhere in the memory map ( $-32768$  to  $+32767$ ) is available.

Program counter relative addressing is also available for data access as well as branch instructions.

The indexed addressing modes have been expanded to include:

- 0-, 5-, 8-, 16-bit constant offsets,
- 8- or 16-bit accumulator offsets,
- autoincrement/decrement (stack operation).

In addition, most indexed addressing modes may have an additional level of indirection added.

Any or all registers may be pushed on to or pulled from either stack with a single instruction.

A multiply instruction is included which multiplies unsigned binary numbers in accumulators A and B and places the unsigned result in the 16-bit accumulator D. This unsigned multiply instruction also allows signed or unsigned multiple precision multiplication.

## 1.4 PROGRAMMING MODEL

The programming model (Figure 1-1) for these processors contains five 16-bit and four 8-bit registers that are available to the programmer.

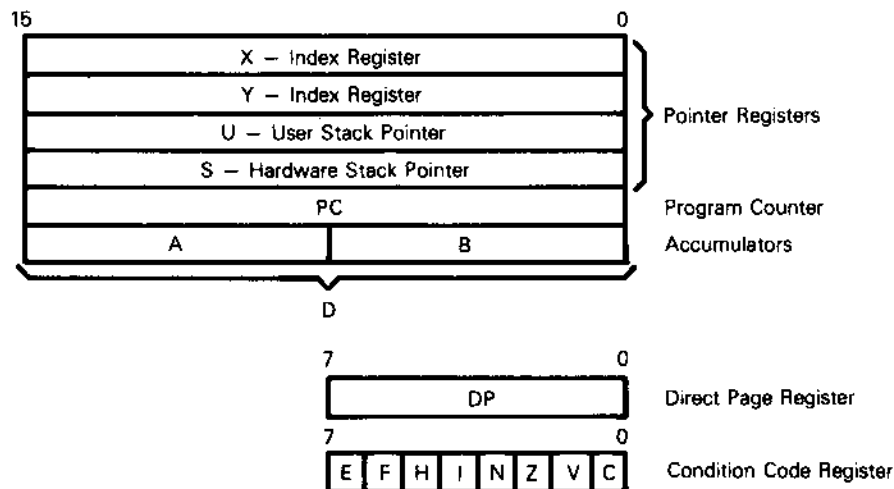


Figure 1-1. Programming Model

## 1.5 INDEX REGISTERS (X, Y)

The index registers are used during the indexed addressing modes. The address information in an index register is used in the calculation of an effective address. This address may be used to point directly to data or may be modified by an optional constant or register offset to produce the effective address.

## 1.6 STACK POINTER REGISTERS (U, S)

Two stack pointer registers are available in these processors. They are: a user stack pointer register (U) controlled exclusively by the programmer, and a hardware stack pointer register (S) which is used automatically by the processor during subroutine calls

and interrupts, but may also be used by the programmer. Both stack pointers always point to the top of the stack.

These registers have the same indexed addressing mode capabilities as the index registers, and also support push and pull instructions. All four indexable registers (X, Y, U, S) are referred to as pointer registers.

### 1.7 PROGRAM COUNTER (PC)

The program counter register is used by these processors to store the address of the next instruction to be executed. It may also be used as an index register in certain addressing modes.

### 1.8 ACCUMULATOR REGISTERS (A, B, D)

The accumulator registers (A, B) are general-purpose 8-bit registers used for arithmetic calculations and data manipulation.

Certain instructions concatenate these registers into one 16-bit accumulator with register A positioned as the most-significant byte. When concatenated, this register is referred to as accumulator D.

### 1.9 DIRECT PAGE REGISTER (DP)

This 8-bit register contains the most-significant byte of the address to be used in the direct addressing mode. The contents of this register are concatenated with the byte following the direct addressing mode operation code to form the 16-bit effective address. The direct page register contents appear as bits A15 through A8 of the address. This register is automatically cleared by a hardware reset to ensure M6800 compatibility.

### 1.10 CONDITION CODE REGISTER (CC)

The condition code register contains the condition codes and the interrupt masks as shown in Figure 1-2.

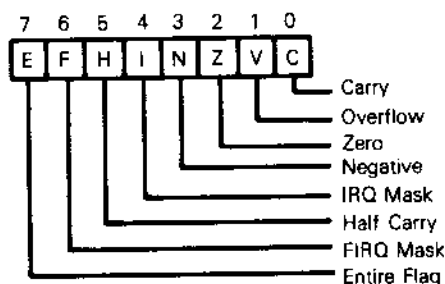


Figure 1-2. Condition Code Register

**1.10.1 CONDITION CODE BITS.** Five bits in the condition code register are used to indicate the results of instructions that manipulate data. They are: half carry (H), negative (N), zero (Z), overflow (V), and carry (C). The effect each instruction has on these bits is given in the detail information for each instruction (see Appendix A).

**1.10.1.1 Half Carry (H), Bit 5.** This bit is used to indicate that a carry was generated from bit three in the arithmetic logic unit as a result of an 8-bit addition. This bit is undefined in all subtract-like instructions. The decimal addition adjust (DAA) instruction uses the state of this bit to perform the adjust operation.

**1.10.1.2 Negative (N), Bit 3.** This bit contains the value of the most-significant bit of the result of the previous data operation.

**1.10.1.3 Zero (Z), Bit 2.** This bit is used to indicate that the result of the previous operation was zero.

**1.10.1.4 Overflow (V), Bit 1.** This bit is used to indicate that the previous operation caused a signed arithmetic overflow.

**1.10.1.5 Carry (C), Bit 0.** This bit is used to indicate that a carry or a borrow was generated from bit seven in the arithmetic logic unit as a result of an 8-bit mathematical operation.

**1.10.2 INTERRUPT MASK BITS AND STACKING INDICATOR.** Two bits (I and F) are used as mask bits for the interrupt request and the fast interrupt request inputs. When either or both of these bits are set, their associated input will not be recognized.

One bit (E) is used to indicate how many registers (all, or only the program counter and condition code) were stacked during the last interrupt.

**1.10.2.1 Fast Interrupt Request Mask (F), Bit 6.** This bit is used to mask (disable) any fast interrupt request line ( $\overline{\text{FIRQ}}$ ). This bit is set automatically by a hardware reset or after recognition of another interrupt. Execution of certain instructions such as SWI will also inhibit recognition of a  $\overline{\text{FIRQ}}$  input.

**1.10.2.2 Interrupt Request Mask (I), Bit 4.** This bit is used to mask (disable) any interrupt request input ( $\overline{\text{IRQ}}$ ). This bit is set automatically by a hardware reset or after recognition of another interrupt. Execution of certain instructions such as SWI will also inhibit recognition of an  $\overline{\text{IRQ}}$  input.

**1.10.2.3 Entire Flag (E), Bit 7.** This bit is used to indicate how many registers were stacked. When set, all the registers were stacked during the last interrupt stacking operation. When clear, only the program counter and condition code registers were stacked during the last interrupt.

The state of the E bit in the stacked condition code register is used by the return from interrupt (RTI) instruction to determine the number of registers to be unstacked.

## 1.11 PIN ASSIGNMENTS AND SIGNAL DESCRIPTION

Figure 1-3 shows the pin assignments for the processors. The following paragraphs provide a short description of each of the input and output signals.

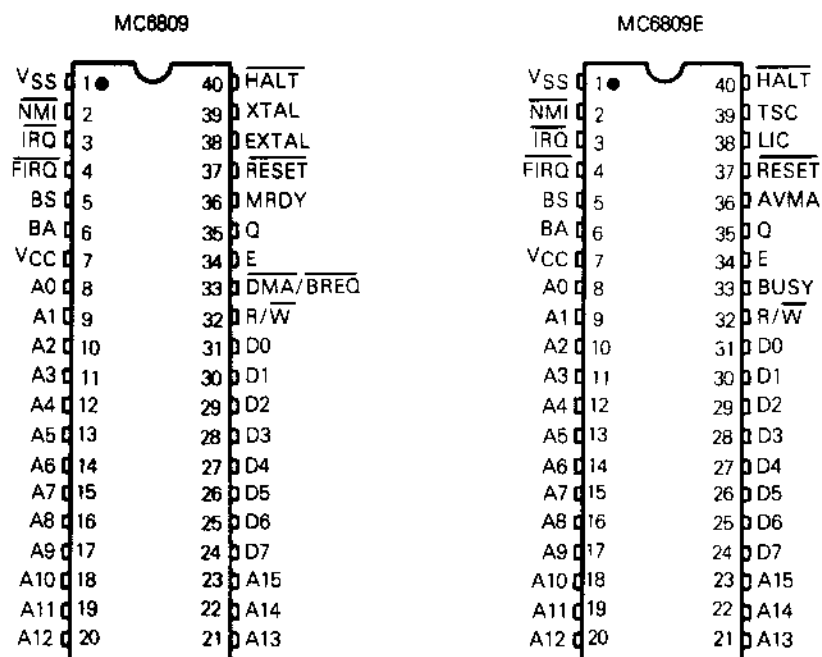


Figure 1-3. Processor Pin Assignments

**1.11.1 MC6809 CLOCKS.** The MC6809 has four pins committed to developing the clock signals needed for internal and system operation. They are: the oscillator pins EXTERNAL and XTAL; the standard M6800 enable (E) clock; and a new, quadrature (Q) clock.

**1.11.1.1 Oscillator (EXTERNAL, XTAL).** These pins are used to connect the processor's internal oscillator to an external, parallel-resonant crystal. These pins can also be used for input of an external TTL timing signal by grounding the XTAL pin and applying the input to the EXTERNAL pin. The crystal or the external timing source is four times the resulting bus frequency.

**1.11.1.2 Enable (E).** The E clock is similar to the phase 2 ( $\phi 2$ ) MC6800 bus timing clock. The leading edge indicates to memory and peripherals that the data is stable and to begin write operations. Data movement occurs after the Q clock is high and is latched on the trailing edge of E. Data is valid from the processor (during a write operation) by the rising edge of E.

**1.11.1.3 Quadrature (Q).** The Q clock leads the E clock by approximately one half of the E clock time. Address information from the processor is valid with the leading edge of the Q clock. The Q clock is a new signal in these processors and does not have an equivalent clock within the MC6800 bus timing.

**1.11.2 MC6809E CLOCKS (E and Q).** The MC6809E has two pins provided for the TTL clock signal inputs required for internal operation. They are the standard M6800 enable (E) clock and the quadrature (Q) clock. The Q input must lead the E input.

Addresses will be valid from the processor (on address delay time after the falling edge of E) and data will be latched from the bus by the falling edge of E. The Q input is fully TTL compatible. The E input is used to drive the internal MOS circuitry directly and therefore requires input levels above the normal TTL levels.

**1.11.3 THREE STATE CONTROLS (TSC) (MC6809E).** This input is used to place the address and data lines and the  $R/\overline{W}$  line in the high-impedance state and allows the address bus to be shared with other bus masters.

**1.11.4 LAST INSTRUCTION CYCLE (LIC) (MC6809E).** This output goes high during the last cycle of every instruction and its high-to-low transition indicates that the first byte of an opcode will be latched at the end of the present bus cycle.

**1.11.5 ADDRESS BUS (A0-A15).** This 16-bit, unidirectional, three-state bus is used by the processor to provide address information to the address bus. Address information is valid on the rising edge of the Q clock. All 16 outputs are in the high-impedance state when the bus available (BA) signal is high, and for one bus cycle thereafter.

When the processor does not require the address bus for a data transfer, it outputs address  $\text{FFFF}_{16}$ , and read/write ( $R/\overline{W}$ ) high. This is a "dummy access" of the least-significant byte of the reset vector which replaces the valid memory address (VMA) functions of the MC6800. For the MC6809, the memory read signal internal circuitry inhibits stretching of the clocks during non-access cycles.

**1.11.6 DATA BUS (D0-D7).** This 8-bit, bidirectional, three-state bus is the general purpose data path. All eight outputs are in the high-impedance state when the bus available (BA) output is high.

**1.11.7 READ/WRITE ( $\overline{R/W}$ ).** This output indicates the direction of data transfer on the data bus. A low indicates that the processor is writing onto the data bus; a high indicates that the processor is reading data from the data bus. The signal at the  $\overline{R/W}$  output is valid at the leading edge of the Q clock. The  $\overline{R/W}$  output is in the high-impedance state when the bus available (BA) output is high.

**1.11.8 PROCESSOR STATE INDICATORS (BA, BS).** The processor uses these two output lines to indicate the present processor state. These pins are valid with the leading edge of the Q clock.

The bus available (BA) output is used to indicate that the buses (address and data) and the read/write output are in the high-impedance state. This signal can be used to indicate to bus-sharing or direct memory access systems that the buses are available. When BA goes low, an additional dead cycle will elapse before the processor regains control of the buses.

The bus status (BS) output is used in conjunction with the BA output to indicate the present state of the processor. Table 1-1 is a listing of the BA and BS outputs and the processor states that they indicate. The following paragraphs briefly explain each processor state.

**Table 1-1. BA/BS Signal Encoding**

<u>BA</u>	<u>BS</u>	<u>Processor State</u>
0	0	Normal (Running)
0	1	Interrupt or Reset Acknowledge
1	0	Sync Acknowledge
1	1	Halt/Bus Grant Acknowledged

**1.11.8.1 Normal.** The processor is running and executing instructions.

**1.11.8.2 Interrupt or Reset Acknowledge.** This processor state is indicated during both cycles of a hardware vector fetch which occurs when any of the following interrupts have occurred:  $\overline{RESET}$ ,  $\overline{NMI}$ ,  $\overline{FIRQ}$ ,  $\overline{IRQ}$ ,  $\overline{SWI}$ ,  $\overline{SWI2}$ , and  $\overline{SWI3}$ .

This output, plus decoding of address lines A3 through A1 provides the user with an indication of which interrupt is being serviced.

**1.11.8.3 Sync Acknowledge.** The processor is waiting for an external synchronization input on an interrupt line. See SYNC instruction in Appendix A.

**1.11.8.4 Halt/Bus Grant.** The processor is halted or bus control has been granted to some other device.



**1.11.9 RESET ( $\overline{\text{RESET}}$ ).** This input is used to reset the processor. A low input lasting longer than one bus cycle will reset the processor.

The reset vector is fetched from locations \$FFFE and \$FFFF when the processor enters the reset acknowledge state as indicated by the BA output being low and the BS output being high.

During initial power-on, the reset input should be held low until the clock oscillator is fully operational.

**1.11.10 INTERRUPTS.** The processor has three separate interrupt input pins: non-maskable interrupt ( $\overline{\text{NMI}}$ ), fast interrupt request ( $\overline{\text{FIRQ}}$ ), and interrupt request ( $\overline{\text{IRQ}}$ ). These interrupt inputs are latched by the falling edge of every Q clock except during cycle stealing operations where only the  $\overline{\text{NMI}}$  input is latched. Using this point as a reference, a delay of at least one bus cycle will occur before the interrupt is recognized by the processor.

**1.11.10.1 Non-Maskable Interrupt ( $\overline{\text{NMI}}$ ).** A negative edge on this input requests that a non-maskable interrupt sequence be generated. This input, as the name indicates, cannot be masked by software and has the highest priority of the three interrupt inputs. After a reset has occurred, a  $\overline{\text{NMI}}$  input will not be recognized by the processor until the first program load of the hardware stack pointer. The entire machine state is saved on the hardware stack during the processing of a non-maskable interrupt. This interrupt is internally blocked after a hardware reset until the stack pointer is initialized.

**1.11.10.2 Fast Interrupt Request ( $\overline{\text{FIRQ}}$ ).** This input is used to initiate a fast interrupt request sequence. Initiation depends on the F (fast interrupt request mask) bit in the condition code register being clear. This bit is set during reset. During the interrupt, only the contents of the condition code register and the program counter are stacked resulting in a short amount of time required to service this interrupt. This interrupt has a higher priority than the normal interrupt request ( $\overline{\text{IRQ}}$ ).

**1.11.10.3 Interrupt Request ( $\overline{\text{IRQ}}$ ).** This input is used to initiate what might be considered the "normal" interrupt request sequence. Initiation depends on the I (interrupt mask) bit in the condition code register being clear. This bit is set during reset. The entire machine state is saved on the hardware stack during processing of an  $\overline{\text{IRQ}}$  input. This input has the lowest priority of the three hardware interrupts.

**1.11.11 MEMORY READ ( $\overline{\text{MRDY}}$ ) (MC6809).** This input allows extension of the E and Q clocks to allow a longer data access time. A low on this input allows extension of the E and Q clocks (E high and Q low) in integral multiples of quarter bus cycles (up to 10 cycles) to allow interface with slow memory devices.

Memory ready does not extend the E and Q clocks during non-valid memory access cycles and therefore the processor does not slow down for "don't care" bus accesses. Memory ready may also be used to extend the E and Q clocks when an external device is using the halt and direct memory access/bus request inputs.

**1.11.12 ADVANCED VALID MEMORY ADDRESS (AVMA) (MC6809E).** This output signal indicates that the MC6809E will use the bus in the following bus cycle. This output is low when the MC6809E is in either a halt or sync state.

**1.11.13 HALT.** This input is used to halt the processor. A low input halts the processor at the end of the present instruction execution cycle and the processor remains halted indefinitely without loss of data.

When the processor is halted, the BA output is high to indicate that the buses are in the high-impedance state and the BS output is also high to indicate that the processor is in the halt/bus grant state.

During the halt/bus grant state, the processor will not respond to external real-time requests such as  $\overline{\text{FIRQ}}$  or  $\overline{\text{IRQ}}$ . However, a direct memory access/bus request input will be accepted. A non-maskable interrupt or a reset input will be latched for processing later. The E and Q clocks continue to run during the halt/bus grant state.

**1.11.14 DIRECT MEMORY ACCESS/BUS REQUEST ( $\overline{\text{DMA/BREQ}}$ ) (MC6809).** This input is used to suspend program execution and make the buses available for another use such as a direct memory access or a dynamic memory refresh.

A low level on this input occurring during the Q clock high time suspends instruction execution at the end of the current cycle. The processor acknowledges acceptance of this input by setting the BA and BS outputs high to signify the bus grant state. The requesting device now has up to 15 bus cycles before the processor retrieves the bus for self-refresh.

Typically, a direct memory access controller will request to use the bus by setting the  $\overline{\text{DMA/BREQ}}$  input low when E goes high. When the processor acknowledges this input by setting the BA and BS outputs high, that cycle will be a dead cycle used to transfer bus mastership to the direct memory access controller. False memory access during any dead cycle should be prevented by externally developing a system DMAVMA signal which is low in any cycle when the BA output changes.

When the BA output goes low, either as a result of a direct memory access/bus request or a processor self-refresh, the direct memory access device should be removed from the bus. Another dead cycle will elapse before the processor accesses memory, to allow transfer of bus mastership without contention.

**1.11.15 BUSY (MC6809E).** This output indicates that bus re-arbitration should be deferred and provides the indivisible memory operation required for a "test-and-set" primitive.

This output will be high for the first two cycles of any Read-Modify-Write instruction, high during the first byte of a double-byte access, and high during the first byte of any indirect access or vector-fetch operation.

**1.11.16 POWER.** Two inputs are used to supply power to the processor: VCC is +5.0  $\pm$  5%, while VSS is ground or 0 volts.

## **SECTION 2 ADDRESSING MODES**

### **2.1 INTRODUCTION**

This section contains a description of each of the addressing modes available on these processors.

### **2.2 ADDRESSING MODES**

The addressing modes available on the MC6809 and MC6809E are: Inherent, Immediate, Extended, Direct, Indexed (with various offsets and autoincrementing/decrementing), and Branch Relative. Some of these addressing modes require an additional byte after the opcode to provide additional addressing interpretation. This byte is called a postbyte.

The following paragraphs provide a description of each addressing mode. In these descriptions the term effective address is used to indicate the address in memory from which the argument for an instruction is fetched or stored, or from which instruction processing is to proceed.

**2.2.1 INHERENT.** The information necessary to execute the instruction is contained in the opcode. Some operations specifying only the index registers or the accumulators, and no other arguments, are also included in this addressing mode.

Example:     **MUL**

**2.2.2 IMMEDIATE.** The operand is contained in one or two bytes immediately following the opcode. This addressing mode is used to provide constant data values that do not change during program execution. Both 8-bit and 16-bit operands are used depending on the size of the argument specified in the opcode.

Example:     **LDA #CR**  
              **LDB #7**  
              **LDA #\$F0**  
              **LDB #%1110000**  
              **LDX #\$8004**

Another form of immediate addressing uses a postbyte to determine the registers to be manipulated. The exchange (EXG) and transfer (TFR) instructions use the postbyte as shown in Figure 2-1(A). The push and pull instructions use the postbyte to designate the registers to be pushed or pulled as shown in Figure 2-1(B).

b7	b6	b5	b4	b3	b2	b1	b0
SOURCE (R1)				DESTINATION (R2)			

Code*	Register	Code*	Register
0000	D (A:B)	0101	Program Counter
0001	X Index	1000	A Accumulator
0010	Y Index	1001	B Accumulator
0011	U Stack Pointer	1010	Condition Code
0100	S Stack Pointer	1011	Direct Page

\*All other combinations of bits produce undefined results.

**(A) Exchange (EXG) or Transfer (TFR) Instruction Postbyte**

b7	b6	b5	b4	b3	b2	b1	b0
PC	S/U	Y	X	DP	B	A	CC

PC = Program Counter  
 S/U = Hardware/User Stack Pointer  
 Y = Y Index Register  
 X = U Index Register  
 DP = Direct Page Register  
 B = B Accumulator  
 A = A Accumulator  
 CC = Condition Code Register

**(B) Push (PSH) or Pull (PUL) Instruction Postbyte**

**Figure 2-1. Postbyte Usage for EXG/TFR, PSH/PUL Instructions**

**2.2.3 EXTENDED.** The effective address of the argument is contained in the two bytes following the opcode. Instructions using the extended addressing mode can reference arguments anywhere in the 64K addressing space. Extended addressing is generally not used in position independent programs because it supplies an absolute address.

Example: LDA @CAT

**2.2.4 DIRECT.** The effective address is developed by concatenation of the contents of the direct page register with the byte immediately following the opcode. The direct page register contents are the most-significant byte of the address. This allows accessing 256 locations within any one of 256 pages. Therefore, the entire addressing range is available for access using a single two-byte instruction.

Example: LDA >CAT

**2.2.5 INDEXED.** In these addressing modes, one of the pointer registers (X, Y, U, or S), and sometimes the program counter (PC) is used in the calculation of the effective address of the instruction operand. The basic types (and their variations) of indexed addressing available are shown in Table 2-1 along with the postbyte configuration used.

**2.2.5.1 Constant Offset from Register.** The contents of the register designated in the postbyte are added to a two's complement offset value to form the effective address of

the instruction operand. The contents of the designated register are not affected by this addition. The offset sizes available are:

- No offset — designated register contains the effective address
- 5-bit — 16 to + 15
- 8-bit — 128 to + 127
- 16-bit — 32768 to + 32767

**Table 2-1. Postbyte Usage for Indexed Addressing Modes**

Mode Type	Variation	Direct	Indirect
Constant Offset from Register (twos Complement Offset)	No Offset 5-Bit Offset 8-Bit Offset 16-Bit Offset	1RR00100 0RRnnnnn 1RR01000 1RR01001	1RR10100 Defaults to 8-bit 1RR11000 1RR11001
Accumulator Offset from Register (twos Complement Offset)	A Accumulator Offset B Accumulator Offset D Accumulator Offset	1RR00110 1RR00101 1RR01011	1RR10110 1RR10101 1RR11011
Auto Increment/Decrement from Register	Increment by 1 Increment by 2 Decrement by 1 Decrement by 2	1RR00000 1RR00001 1RR00010 1RR00011	Not Allowed 1RR10001 Not Allowed 1RR10011
Constant Offset from Program Counter	8-Bit Offset 16-Bit Offset	1XX01100 1XX01101	1XX11100 1XX11101
Extended Indirect	16-Bit Address	-----	10011111

The 5-bit offset value is contained in the postbyte. The 8- and 16-bit offset values are contained in the byte or bytes immediately following the postbyte. If the Motorola assembler is used, it will automatically determine the most efficient offset; thus, the programmer need not be concerned about the offset size.

Examples: LDA ,X      LDY - 64000,U  
               LDB 0,Y      LDA 17,PC  
               LDX 64,000,S    LDA There,PCR

**2.2.5.2 Accumulator Offset from Register.** The contents of the index or pointer register designed in the postbyte are temporarily added to the twos complement offset value contained in an accumulator (A, B, or D) also designated in the postbyte. Neither the designated register nor the accumulator contents are affected by this addition.

Example: LDA A,X      LDA D,U  
               LDA B,Y

**2.2.5.3 Autoincrement/Decrement from Register.** This addressing mode works in a postincrementing or predecrementing manner. The amount of increment or decrement, one or two positions, is designated in the postbyte.

In the autoincrement mode, the contents of the effective address contained in the pointer register, designated in the postbyte, and then the pointer register is automatically incremented; thus, the pointer register is postincremented.

In the autodecrement mode, the pointer register, designated in the postbyte, is automatically decremented first and then the contents of the new address are used; thus, the pointer register is predecremented.

Examples:

Autoincrement		Autodecrement	
LDA ,X+	LDY ,X++	LDA ,-X	LDY ,--X
LDA ,Y+	LDX ,Y++	LDA ,-Y	LDX ,--Y
LDA ,S+	LDX ,U++	LDA ,-S	LDX ,--U
LDA ,U+	LDX ,S++	LDA ,-U	LDX ,--S

**2.2.5.4 Indirection.** When using indirection, the effective address of the base indexed addressing mode is used to fetch two bytes which contain the final effective address of the operand. It can be used with all the Indexed addressing modes and the program counter relative addressing mode.

**2.2.5.5 Extended Indirect.** The effective address of the argument is located at the address specified by the two bytes following the postbyte. The postbyte is used to indicate indirection.

Example: LDA [\$F000]

**2.2.5.6 Program Counter Relative.** The program counter can also be used as a pointer with either an 8- or 16-bit signed constant offset. The offset value is added to the program counter to develop an effective address. Part of the postbyte is used to indicate whether the offset is 8 or 16 bits.

**2.2.6 BRANCH RELATIVE.** This addressing mode is used when branches from the current instruction location to some other location relative to the current program counter are desired. If the test condition of the branch instruction is true, then the effective address is calculated (program counter plus two's complement offset) and the branch is taken. If the test condition is false, the processor proceeds to the next in-line instruction. Note that the program counter is always pointing to the next instruction when the offset is added. Branch relative addressing is always used in position independent programs for all control transfers.

For short branches, the byte following the branch instruction opcode is treated as an 8-bit signed offset to be used to calculate the effective address of the next instruction if the branch is taken. This is called a short relative branch and the range is limited to plus 127 or minus 128 bytes from the following opcode.

For long branches, the two bytes after the opcode are used to calculate the effective address. This is called a long relative branch and the range is plus 32,767 or minus 32,768

## SECTION 3 INTERRUPT CAPABILITIES

### 3.1 INTRODUCTION

The MC6809 and MC6809E microprocessors have six vectored interrupts (three hardware and three software). The hardware interrupts are the non-maskable interrupt ( $\overline{\text{NMI}}$ ), the fast maskable interrupt request ( $\overline{\text{FIRQ}}$ ), and the normal maskable interrupt request ( $\overline{\text{IRQ}}$ ). The software interrupts consist of SWI, SWI2, and SWI3. When an interrupt request is acknowledged, all the processor registers are pushed onto the hardware stack, except in the case of  $\overline{\text{FIRQ}}$  where only the program counter and the condition code register is saved, and control is transferred to the address in the interrupt vector. The priority of these interrupts is, highest to lowest,  $\overline{\text{NMI}}$ , SWI,  $\overline{\text{FIRQ}}$ ,  $\overline{\text{IRQ}}$ , SWI2, and SWI3. Figure 3-1 is a detailed flowchart of interrupt processing in these processors. The interrupt vector locations are given in Table 3-1. The vector locations contain the address for the interrupt routine.

Additional information on the SWI, SWI2, and SWI3 interrupts is given in Appendix A. The hardware interrupts,  $\overline{\text{NMI}}$ ,  $\overline{\text{FIRQ}}$ , and  $\overline{\text{IRQ}}$  are listed alphabetically at the end of Appendix A.

**Table 3-1. Interrupt Vector Locations**

Interrupt Description	Vector Location	
	MS Byte	LS Byte
Reset (RESET)	FFFE	FFFF
Non-Maskable Interrupt ( $\overline{\text{NMI}}$ )	FFFC	FFFD
Software Interrupt (SWI)	FFFA	FFFB
Interrupt Request ( $\overline{\text{IRQ}}$ )	FFF8	FFF9
Fast Interrupt Request ( $\overline{\text{FIRQ}}$ )	FFF6	FFF7
Software Interrupt 2 (SWI2)	FFF4	FFF5
Software Interrupt 3 (SWI3)	FFF2	FFF3
Reserved	FFF0	FFF1

### 3.2 NON-MASKABLE INTERRUPT ( $\overline{\text{NMI}}$ )

The non-maskable interrupt is edge-sensitive in the sense that if it is sampled low one cycle after it has been sampled high, a non-maskable interrupt will be triggered. Because the non-maskable interrupt cannot be masked by execution of the non-maskable interrupt handler routine, it is possible to accept another non-maskable interrupt before executing the first instruction of the interrupt routine. A fatal error will exist if a non-maskable interrupt is repeatedly allowed to occur before completing the return from interrupt ( $\overline{\text{RTI}}$ ) instruction of the previous non-maskable interrupt request, since the stack



will eventually overflow. This interrupt is especially applicable to gaining immediate processor response for powerfail, software dynamic memory refresh, or other non-delayable events.

### **3.3 FAST MASKABLE INTERRUPT REQUEST ( $\overline{\text{FIRQ}}$ )**

A low level on the  $\overline{\text{FIRQ}}$  input with the F (fast interrupt request mask) bit in the condition code register clear triggers this interrupt sequence. The fast interrupt request provides fast interrupt response by stacking only the program counter and condition code register. This allows fast context switching with minimal overhead. If any registers are used by the interrupt routine then they can be saved by a single push instruction.

After accepting a fast interrupt request, the processor clears the E flag, saves the program counter and condition code register, and then sets both the I and F bits to mask any further IRQ and  $\overline{\text{FIRQ}}$  interrupts. After servicing the original interrupt, the user may selectively clear the I and F bits to allow multiple-level interrupts if so desired.

### **3.4 NORMAL MASKABLE INTERRUPT REQUEST ( $\overline{\text{IRQ}}$ )**

A low level on the  $\overline{\text{IRQ}}$  input with the I (interrupt request mask) bit in the condition code register clear triggers this interrupt sequence. The normal maskable interrupt request provides a slower hardware response to interrupts because it causes the entire machine state to be stacked. However, this means that interrupting software routines can use all processor resources without fear of damaging the interrupted routine. A normal interrupt request, having lower priority than the fast interrupt request, is prevented from interrupting the fast interrupt handler by the automatic setting of the I bit by the fast interrupt request handler.

After accepting a normal interrupt request, the processor sets the E flag, saves the entire machine state, and then sets the I bit to mask any further interrupt request inputs. After servicing the original interrupt, the user may clear the I bit to allow multiple-level normal interrupts.

All interrupt handling routines should return to the formerly executing tasks using a return from interrupt (RTI) instruction. This instruction recovers the saved machine state from the hardware stack and control is returned to the interrupted program. If the recovered E bit is clear, it indicates that a fast interrupt request occurred and only the program counter address and condition code register are to be recovered.

### **3.5 SOFTWARE INTERRUPTS (SWI, SWI2, SWI3)**

The software interrupts cause the processor to go through the normal interrupt request sequence of stacking the complete machine state even though the interrupting source is the processor itself. These interrupts are commonly used for program debugging and for calls to an operating system.

Normal processing of the SWI input sets the I and F bits to prevent either of these interrupt requests from affecting the completion of a software interrupt request. The remaining software interrupt request inputs (SWI2 and SWI3) do not have the priority of the SWI input and therefore do not mask the two hardware interrupt request inputs (FIRQ and IRQ).



## **SECTION 4 PROGRAMMING**

### **4.1 INTRODUCTION**

These processors are designed to be source-code compatible with the M6800 to make use of the substantial existing base of M6800 software and training. However, this asset should not overshadow the capabilities built into these processors that allow more modern programming techniques such as position-independence, modular programming, and reentrancy/recursion to be used on a microprocessor-based system. A brief review of these methods is given in the following paragraphs.

**4.1.1 POSITION INDEPENDENCE.** A program is said to be "position-independent" if it will run correctly when the same machine code is positioned arbitrarily in memory. Such a program is useful in many different hardware configurations, and might be copied from a disk into RAM when the operating system first sees a request to use a system utility. Position-independent programs never use absolute (extended or direct) addressing; instead, inherent immediate, register, indexed and relative modes are used. In particular, there should be no jump (absolute) or jump to subroutine instructions nor should absolute addresses be used. A position-independent program is almost always preferable to a position-dependent program (although position-independent code is usually 5 to 10% slower than normal code).

**4.1.2 MODULAR PROGRAMMING.** Modular programming is another indication of quality code. A module is a program element which can be easily disconnected from the rest of the program either for re-use in a new environment or for replacement. A module is usually a subroutine (although a subroutine is not necessarily a module); frequently, the programmer isolates register changes internal to the module by pushing these registers onto the stack upon entry, and pulling them off the stack before the return. Isolating register changes in the called module, to that module alone, allows the code in the calling program to be more easily analyzed since it can be assumed that all registers (except those specifically used for parameter transfer are unchanged by each called module. This leaves the processor's registers free at each level for loop counts, address comparisons, etc.

**4.1.2.1 Local Storage.** A clean method for allocating "local" storage is required both by position-independent programs as well as modular programs. Local or temporary storage is used to hold values only during execution of a module (or called modules) and is released upon return. One way to allocate local storage is to decrement the hardware stack

pointer(s) by the number of bytes needed. Interrupts will then leave this area intact and it can be de-allocated on exiting the module. A module will almost always need more temporary storage than just the MPU registers.

**4.1.2.2 Global Storage.** Even in a modular environment there may be a need for "global" values which are accessible by many modules within a given system. These provide a convenient means for storing values from one invocation to another invocation of the same routine. Global storage may be created as local storage at some level, and a pointer register (usually U) used to point at this area. This register is passed unchanged in all subroutines, and may be used to index into the global area.

**4.1.3 REENTRANCY/RECURSION.** Many programs will eventually involve execution in an interrupt-driven environment. If the interrupt handlers are complex, they might well call the same routine which has just been interrupted. Therefore, to protect present programs against certain obsolescence, all programs should be written to be reentrant. A reentrant routine allocates different local variable storage upon each entry. Thus, a later entry does not destroy the processing associated with an earlier entry.

The same technique which was implemented to allow reentrancy also allows recursion. A recursive routine is defined as a routine that calls itself. A recursive routine might be written to simplify the solution of certain types of problems, especially those which have a data structure whose elements may themselves be a structure. For example, a parenthetical equation represents a case where the expression in parenthesis may be considered to be a value which is operated on by the rest of the equation. A programmer might choose to write an expression evaluator passing the parenthetical expression (which might also contain parenthetical expressions) in the call, and receive back the returned value of the expression within the parenthesis.

## **4.2 M6809 CAPABILITIES**

The following paragraphs briefly explain how the MC6809 is used with the programming techniques mentioned earlier.

**4.2.1 MODULE CONSTRUCTION.** A module can be defined as a logically self-contained and discrete part of a larger program. A properly constructed module accepts well defined inputs, carries out a set of processing actions, and produces a specified output. The use of parameters, local storage, and global storage by a program module is given in the following paragraphs. Since registers will be used inside the module (essentially a form of local storage), the first thing that is usually done at entry to a module is to push (save) them on to the stack. This can be done with one instruction (e.g., PSHS Y, X, B, A). After the body of the module is executed, the saved registers are collected, and a subroutine return is performed, at one time, by pulling the program counter from the stack (e.g., PULS A,B,X,Y,PC).

**4.2.1.1 Parameters.** Parameters may be passed to or from modules either in registers, if they will provide sufficient storage for parameter passage, or on the stack. If parameters are passed on the stack, they are placed there before calling the lower level module. The called module is then written to use local storage inside the stack as needed (e.g., ADDA offset,S). Notice that the required offset consists of the number of bytes pushed (upon entry), plus two from the stacked return address, plus the data offset at the time of the call. This value may be calculated, by hand, by drawing a "stack picture" diagram representing module entry, and assigning convenient mnemonics to these offsets with the assembler. Returned parameters replace those sent to the routine. If more parameters are to be returned on the stack than would normally be sent, space for their return is allocated by the calling routine before the actual call (if four additional bytes are to be returned, the caller would execute LEAS - 4,S to acquire the additional storage).

**4.2.1.2 Local Storage.** Local storage space is acquired from the stack while the present routine is executing and then returned to the stack prior to exit. The act of pushing registers which will be used in later calculations essentially saves those registers in temporary local storage. Additional local storage can easily be acquired from the stack e.g., executing LEAS - 2048,S acquires a buffer area running from the 0,S to 2047,S inclusive. Any byte in this area may be accessed directly by any instruction which has an indexed addressing mode. At the end of the routine, the area acquired for local storage is released (e.g., LEAS 2048,S) prior to the final pull. For cleaner programs, local storage should be allocated at entry to the module and released at the exit of the module.

**4.2.1.3 Global Storage.** The area required for global storage is also most effectively acquired from the stack, probably by the highest level routine in the standard package. Although this is local storage to the highest level routine, it becomes "global" by positioning a register to point at this storage, (sometimes referred to as a stack mark) then establishing the convention that all modules pass that same pointer value when calling lower level modules. In practice, it is convenient to leave this stack mark register unchanged in all modules, especially if global accesses are common. The highest level routine in the standard package would execute the following sequence upon entry (to initialize the global area):

```

PSHS    U        higher level mark, if any
TFR     S,U      new stack mark
LEAS    - 17,U   allocate global storage

```

Note that the U register now defines 17-bytes of locally allocated (permanent) globals (which are - 1,U through - 17,U) as well as other external globals (2,U and above) which have been passed on the stack by the routine which called the standard package. Any global may be accessed by any module using exactly the same offset value at any level (e.g., ROL, RAT,U; where RAT EQU - 11 has been defined). Furthermore, the values stacked prior to invoking the standard package may include pointers to data or I/O peripherals. Any indexed operation may be performed indexed indirect through those pointers, which means, for example, that the module need know nothing about the actual hardware configuration, except that (upon entry) the pointer to an I/O register has been placed at a given location on the stack.

**4.2.2 POSITION-INDEPENDENT CODE.** Position-independent code means that the same machine language code can be placed anywhere in memory and still function correctly. The M6809 has a long relative (16-bit offset) branch mode along with the common MC6800 branches, plus program-counter relative addressing. Program-counter relative addressing uses the program counter like an indexable register, which allows all instructions that reference memory to also reference data relative to the program counter. The M6809 also has load effective address (LEA) instructions which allow the user to point to data in a ROM in a position-independent manner.

An important rule for generating position-independent code is: NEVER USE ABSOLUTE ADDRESSING.

Program-counter relative addressing on the M6809 is a form of indexed addressing that uses the program counter as the base register for a constant-offset indexing operation. However, the M6809 assembler treats the PCR address field differently from that used in other indexed instructions. In PCR addressing, the assembly time location value is subtracted from the (constant) value of the PCR offset. The resulting distance to the desired symbol is the value placed into the machine language object code. During execution, the processor adds the value of the run time PC to the distance to get a position-independent absolute address.

The PCR indexed addressing form can be used to point at any location relative to the program regardless of position in memory. The PCR form of indexed addressing allows access to tables within the program space in a position-independent manner via use of the load effective address instruction.

In a program which is completely position-independent, some absolute locations are usually required, particularly for I/O. If the locations of I/O devices are placed on the stack (as globals) by a small setup routine before the standard package is invoked, all internal modules can do their I/O through that pointer (e.g., STA [ACIAD, U]), allowing the hardware to be easily changed, if desired. Only the single, small, and obvious setup routine need be rewritten for each different hardware configuration.

Global, permanent, and temporary values need to be easily available in a position-independent manner. Use the stack for this data since the stacked data is directly accessible. Stack the absolute address of I/O devices before calling any standard software package since the package can use the stacked addresses for I/O in any system.

The LEA instructions allow access to tables, data, or immediate values in the text of the program in a position-independent manner as shown in the following example:

	LEAX	MSG1,PCR
	LBSR	PDATA
MSG1	FCC	/PRINT THIS!//

Here we wish to point at a message to be printed from the body of the program. By writing "MSG1, PCR" we signal the assembler to compute the distance between the present address (the address of the LBSR) and MSG1. This result is inserted as a constant into the LEA instruction which will be indexed from the program counter value at the time of execution. Now, no matter where the code is located, when it is executed the computer offset from the program counter will point at MSG1. This code is position-independent.

It is common to use space in the hardware stack for temporary storage. Space is made for temporary variables from 0,S through TEMP-1, S by decrementing the stack pointer equal to the length of required storage. We could use:

LEAS            - TEMP,S.

Not only does this facilitate position-independent code but it is structured and helps reentrancy and recursion.

**4.2.3 REENTRANT PROGRAMS.** A program that can be executed by several different users sharing the same copy of it in memory is called reentrant. This is important for interrupt driven systems. This method saves considerable memory space, especially with large interrupt routines. Stacks are required for reentrant programs, and the M6809 can support up to four stacks by using the X and Y index registers as stack pointers.

Stacks are simple and convenient mechanisms for generating reentrant programs. Subroutines which use stacks for passing parameters and results can be easily made to be reentrant. Stack accesses use the indexed addressing mode for fast, efficient execution. Stack addressing is quick.

Pure code, or code that is not self-modifying, is mandatory to produce reentrant code. No internal information within the code is subject to modification. Reentrant code never has internal temporary storage, is simpler to debug, can be placed in ROM, and must be interruptable.

**4.2.4 RECURSIVE PROGRAMS.** A recursive program is one that can call itself. They are quite convenient for parsing mechanisms and certain arithmetic functions such as computing factorials. As with reentrant programming, stacks are very useful for this technique.

**4.2.5 LOOPS.** The usual structured loops (i.e., REPEAT...UNTIL, WHILE...DO, FOR..., etc.) are available in assembly language in exactly the same way a high-level language compiler could translate the construct for execution on the target machine. Using a FOR...NEXT loop as an example, it is possible to push the loop count, increment value, and termination value on the stack as variables local to that loop. On each pass through the loop, the working register is saved, the loop count picked up, the increment added in, and the result compared to the termination value. Based on this comparison, the loop counter might be updated, the working register recovered and the loop resumed, or the working register recovered and the loop variables de-allocated. Reasonable macros



could make the source form for loop trivial, even in assembly language. Such macros might reduce errors resulting from the use of multiple instructions simply to implement a standard control structure.

**4.2.6 STACK PROGRAMMING.** Many microprocessor applications require data stored as contiguous pieces of information in memory. The data may be temporary, that is, subject to change or it may be permanent. Temporary data will most likely be stored in RAM. Permanent data will most likely be stored in ROM.

It is important to allow the main program as well as subroutines access to this block of data, especially if arguments are to be passed from the main program to the subroutines and vice versa.

**4.2.6.1 M6809 Stacking Operations.** Stack pointers are markers which point to the stack and its internal contents. Although all four index registers may be used as stack registers, the S (hardware stack pointer) and the U (user stack pointer) are generally preferred because the push and pull instructions apply to these registers. Both are 16-bit indexable registers. The processor uses the S register automatically during interrupts and subroutine calls. The U register is free for any purpose needed. It is not affected by interrupts or subroutine calls implemented by the hardware.

Either stack pointer can be specified as the base address in indexed addressing. One use of the indirect addressing mode uses stack pointers to allow addresses of data to be passed to a subroutine on a stack as arguments to a subroutine. The subroutine can now reference the data with one instruction. High-level language calls that pass arguments by reference are now more efficiently coded. Also, each stack push or pull operation in a program uses a postbyte which specifies any register or set of registers to be pushed or pulled from either stack. With this option, the overhead associated with subroutine calls in both assembly and high-level language programs is greatly decreased. In fact, with the large number of instructions that use autoincrement and autodecrement, the M6809 can emulate a true stack computer architecture.

Using the S or U stack pointer, the order in which the registers are pushed or pulled is shown in Figure 4-1. Notice that we push "onto" the stack towards decreasing memory locations. The program counter is pushed first. Then the stack pointer is decremented and the "other" stack pointer is pushed onto the stack. Decrementing and storing continues until all the registers requested by the postbyte are pushed onto the stack. The stack pointer points to the top of the stack after the push operation.

The stacking order is specified by the processor. The stacking order is identical to the order used for all hardware and software interrupts. The same order is used even if a subset of the registers is pushed.

Without stacks, most modern block-structured high-level languages would be cumbersome to implement. Subroutine linkage is very important in high-level language generation. Paragraph 4.2.6.2 describes how to use a stack mark pointer for this important task.

Good programming practice dictates the use of the hardware stack for temporary storage. To reserve space, decrement the stack pointer by the amount of storage required with the instruction LEAS - TEMPS, S. This instruction makes space for temporary variables from 0,S through TEMPS - 1,S.

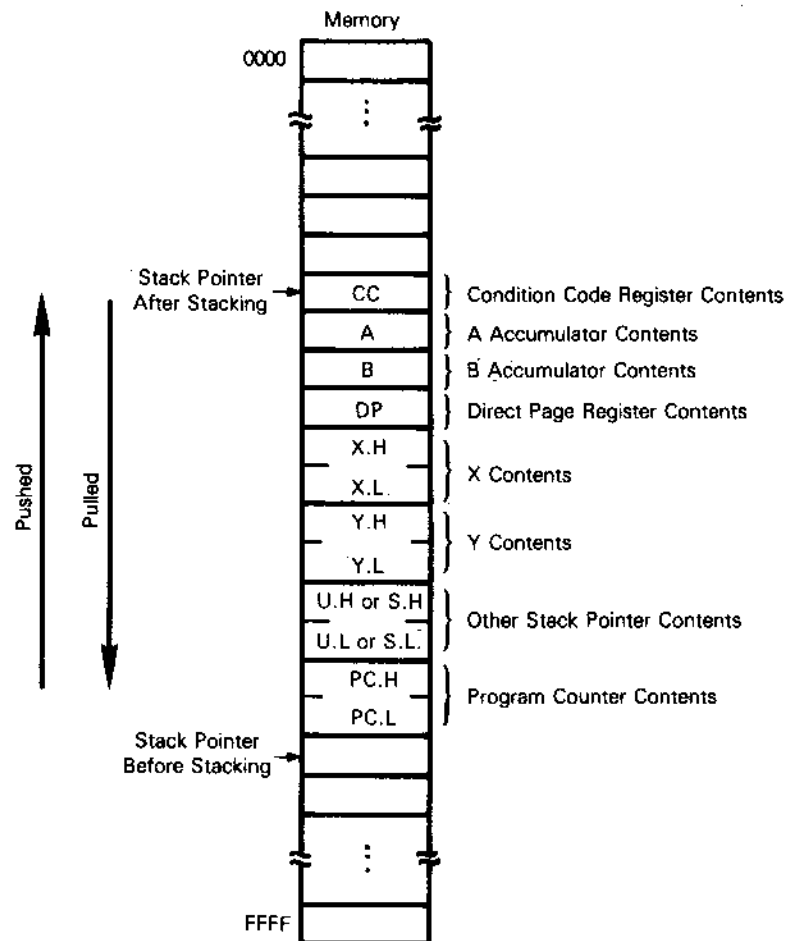


Figure 4-1. Stacking Order

**4.2.6.2 Subroutine Linkage.** In the highest level routine, global variables are sometimes considered to be local. Therefore, global storage is allocated at this point, but access to these same variables requires different offset values depending on subroutine depth. Because subroutine depth changes dynamically, the length may not be known beforehand. This problem is solved by assigning one pointer (U will be used in the following description, but X or Y could also be used) to "mark" a location on the hardware stack by using the instruction TFR S,U. If the programmer does this immediately prior to allocating global storage, then all variables will then be available at a constant negative offset location from this stack mark. If the stack is marked after the global variables are

allocated, then the global variables are available at a constant positive offset from U. Register U is then called the stack mark pointer. Recall that the hardware stack pointer may be modified by hardware interrupts. For this reason, it is fatal to use data referred to by a negative offset with respect to the hardware stack pointer, S.

**4.2.6.3 Software Stacks.** If more than two stacks are needed, autoincrement and autodecrement mode of addressing can be used to generate additional software stack pointers.

The X, Y, and U index registers are quite useful in loops for incrementing and decrementing purposes. The pointer is used for searching tables and also to move data from one area of memory to another (block moves). This autoincrement and autodecrement feature is available in the indexed addressing mode of the M6809 to facilitate such operations.

In autoincrement, the value contained in the index register (X or Y, U or S) is used as the effective address and then the register is incremented (postincremented). In autodecrement, the index register is first decremented and then used to obtain the effective address (predecremented). Postincrement or predecrement is always performed in this addressing mode. This is equivalent in operation to the push and pull from a stack. This equivalence allows the X and Y index registers to be used as software stack pointers. The indexed addressing mode can also implement an extra level of post indirection. This feature supports parameter and pointer operations.

**4.2.7 REAL TIME PROGRAMMING.** Real time programming requires special care. Sometimes a peripheral or task demands an immediate response from the processor, other times it can wait. Most real time applications are demanding in terms of processor response.

A common solution is to use the interrupt capability of the processor in solving real time problems. Interrupts mean just that; they request a break in the current sequence of events to solve an asynchronous service request. The system designer must consider all variations of the conditions to be encountered by the system including software interaction with interrupts. As a result, problems due to software design are more common in interrupt implementation code for real time programming than most other situations. Software timeouts, hardware interrupts, and program control interrupts are typically used in solving real time programming problems.

## **4.3 PROGRAM DOCUMENTATION**

Common sense dictates that a well documented program is mandatory. Comments are needed to explain each group of instructions since their use is not always obvious from looking at the code. Program boundaries and branch instructions need full clarification. Consider the following points when writing comments: up-to-date, accuracy, completeness, conciseness, and understandability.

Accurate documentation enables you and others to maintain and adapt programs for updating and/or additional use with other programs.

The following program documentation standards are suggested.

- A) Each subroutine should have an associated header block containing at least the following elements:
  - 1) A full specification for this subroutine — including associated data structures — such that replacement code could be generated from this description alone.
  - 2) All usage of memory resources must be defined, including:
    - a) All RAM needed from temporary (local) storage used during execution of this subroutine or called subroutines.
    - b) All RAM needed for permanent storage (used to transfer values from one execution of the subroutine to future executions).
    - c) All RAM accessed as global storage (used to transfer values from or to higher-level subroutines).
    - d) All possible exit-state conditions, if these are to be used by calling routines to test occurrences internal to the subroutine.
- B) Code internal to each subroutine should have sufficient associated line comments to help in understanding the code.
- C) All code must be non-self-modifying and position-independent.
- D) Each subroutine which includes a loop must be separately documented by a flowchart or pseudo high-level language algorithm.
- E) Any module or subroutine should be executable starting at the first location and exit at the last location.

#### 4.4 INSTRUCTION SET

The complete instruction set for the M6809 is given in Table 4-1.

**Table 4-1. Instruction Set**

Instruction	Description
ABX	Add Accumulator B into Index Register X
ADC	Add with Carry into Register
ADD	Add Memory into Register
AND	Logical AND Memory into Register
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Equal
BGE	Branch on Greater Than or Equal to Zero
BGT	Branch on Greater
BHI	Branch if Higher
BHS	Branch if Higher or Same
BIT	Bit Test
BLE	Branch if Less than or Equal to Zero

**Table 4-1. Instruction Set (Continued)**

Instruction	Description
BLO	Branch on Lower
BLS	Branch on Lower or Same
BLT	Branch on Less than Zero
BMI	Branch on Minus
BNE	Branch Not Equal
BPL	Branch on Plus
BRA	Branch Always
BRN	Branch Never
BSR	Branch to Subroutine
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLR	Clear
CMP	Compare Memory from a Register
COM	Complement
CWAI	Clear CC bits and Wait for Interrupt
DAA	Decimal Addition Adjust
DEC	Decrement
EOR	Exclusive OR
EXG	Exchange Registers
INC	Increment
JMP	Jump
JSR	Jump to Subroutine
LD	Load Register from Memory
LEA	Load Effective Address
LSL	Logical Shift Left
LSR	Logical Shift Right
MUL	Multiply
NEG	Negate
NOP	No Operation
OR	Inclusive OR Memory into Register
PSH	Push Registers
PUL	Pull Registers
ROL	Rotate Left
ROR	Rotate Right
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract with Borrow
SEX	Sign Extend
ST	Store Register into Memory
SUB	Subtract Memory from Register
SWI	Software Interrupt
SYNC	Synchronize to External Event
TFR	Transfer Register to Register
TST	Test

The instruction set can be functionally divided into five categories. They are:

8-Bit Accumulator and Memory Instructions

16-Bit Accumulator and Memory Instructions

Index Register/Stack Pointer Instructions

Branch Instructions

Miscellaneous Instructions

Tables 4-2 through 4-6 are listings of the M6809 instructions and their variations grouped into the five categories listed.

**Table 4-2. 8-Bit Accumulator and Memory Instructions**

Instruction	Description
ADCA, ADCB	Add memory to accumulator with carry
ADDA, ADDB	Add memory to accumulator
ANDA, ANDB	And memory with accumulator
ASL, ASLA, ASLB	Arithmetic shift of accumulator or memory left
ASR, ASRA, ASRB	Arithmetic shift of accumulator or memory right
BITA, BITB	Bit test memory with accumulator
CLR, CLRA, CLRB	Clear accumulator or memory location
CMPA, CMPB	Compare memory from accumulator
COM, COMA, COMB	Complement accumulator or memory location
DAA	Decimal adjust A accumulator
DEC, DECA, DECB	Decrement accumulator or memory location
EORA, EORB	Exclusive or memory with accumulator
EXG R1, R2	Exchange R1 with R2 (R1, R2 = A, B, CC, DP)
INC, INCA, INCB	Increment accumulator or memory location
LDA, LDB	Load accumulator from memory
LSL, LSLA, LSLB	Logical shift left accumulator or memory location
LSR, LSRA, LSRB	Logical shift right accumulator or memory location
MUL	Unsigned multiply ( $A \times B \rightarrow D$ )
NEG, NEGA, NEGB	Negate accumulator or memory
ORA, ORB	Or memory with accumulator
ROL, ROLA, ROLB	Rotate accumulator or memory left
ROR, RORA, RORB	Rotate accumulator or memory right
SBCA, SBCB	Subtract memory from accumulator with borrow
STA, STB	Store accumulator to memory
SUBA, SUBB	Subtract memory from accumulator
TST, TSTA, TSTB	Test accumulator or memory location
TFR R1, R2	Transfer R1 to R2 (R1, R2 = A, B, CC, DP)

NOTE: A, B, CC, or DP may be pushed to (pulled from) either stack with PSHS, PSHU (PULS, PULU) instructions.

**Table 4-3. 16-Bit Accumulator and Memory Instructions**

Instruction	Description
ADDD	Add memory to D accumulator
CMPD	Compare memory from D accumulator
EXG D, R	Exchange D with X, Y, S, U, or PC
LDD	Load D accumulator from memory
SEX	Sign Extend B accumulator into A accumulator
STD	Store D accumulator to memory
SUBD	Subtract memory from D accumulator
TFR D, R	Transfer D to X, Y, S, U, or PC
TFR R, D	Transfer X, Y, S, U, or PC to D

NOTE: D may be pushed (pulled) to either stack with PSHS, PSHU (PULS, PULU) instructions.

**Table 4-4. Index/Stack Pointer Instructions**

Instruction	Description
CMPS, CMPU	Compare memory from stack pointer
CMPX, CMPY	Compare memory from index register
EXG R1, R2	Exchange D, X, Y, S, U or PC with D, X, Y, S, U or PC
LEAS, LEAU	Load effective address into stack pointer
LEAX, LEAY	Load effective address into index register
LDS, LDU	Load stack pointer from memory
LDX, LDY	Load index register from memory
PSHS	Push A, B, CC, DP, D, X, Y, U, or PC onto hardware stack
PSHU	Push A, B, CC, DP, D, X, Y, X, or PC onto user stack
PULS	Pull A, B, CC, DP, D, X, Y, U, or PC from hardware stack
PULU	Pull A, B, CC, DP, D, X, Y, S, or PC from hardware stack
STS, STU	Store stack pointer to memory
STX, STY	Store index register to memory
TFR R1, R2	Transfer D, X, Y, S, U, or PC to D, X, Y, S, U, or PC
ABX	Add B accumulator to X (unsigned)

**Table 4-5. Branch Instructions**

Instruction	Description
<b>SIMPLE BRANCHES</b>	
BEQ, LBEQ	Branch if equal
BNE, LBNE	Branch if not equal
BMI, LBMI	Branch if minus
BPL, LBPL	Branch if plus
BCS, LBCS	Branch if carry set
BCC, LBCC	Branch if carry clear
BVS, LBVS	Branch if overflow set
BVC, LBVC	Branch if overflow clear
<b>SIGNED BRANCHES</b>	
BGT, LBGT	Branch if greater (signed)
BVS, LBVS	Branch if invalid twos complement result
BGE, LBGE	Branch if greater than or equal (signed)
BEQ, LBEQ	Branch if equal
BNE, LBNE	Branch if not equal
BLE, LBLE	Branch if less than or equal (signed)
BVC, LBVC	Branch if valid twos complement result
BLT, LBLT	Branch if less than (signed)
<b>UNSIGNED BRANCHES</b>	
BHI, LBHI	Branch if higher (unsigned)
BCC, LBCC	Branch if higher or same (unsigned)
BHS, LBHS	Branch if higher or same (unsigned)
BEQ, LBEQ	Branch if equal
BNE, LBNE	Branch if not equal
BLS, LBLs	Branch if lower or same (unsigned)
BCS, LBCS	Branch if lower (unsigned)
BLO, LBLO	Branch if lower (unsigned)
<b>OTHER BRANCHES</b>	
BSR, LBSR	Branch to subroutine
BRA, LBRA	Branch always
BRN, LBRN	Branch never

**Table 4-6. Miscellaneous Instructions**

Instruction	Description
ANDCC	AND condition code register
CWAI	AND condition code register, then wait for interrupt
NOP	No operation
ORCC	OR condition code register
JMP	Jump
JSR	Jump to subroutine
RTI	Return from interrupt
RTS	Return from subroutine
SWI, SWI2, SWI3	Software interrupt (absolute indirect)
SYNC	Synchronize with interrupt line



## APPENDIX A INSTRUCTION SET DETAILS

### A.1 INTRODUCTION

This appendix contains detailed information about each instruction in the MC6809 instruction set. They are arranged in an alphabetical order with the mnemonic heading set in larger type for easy reference.

### A.2 NOTATION

In the operation description for each instruction, symbols are used to indicate the operation. Table A-1 lists these symbols and their meanings. Abbreviations for the various registers, bits, and bytes are also used. Table A-2 lists these abbreviations and their meanings.

Table A-1. Operation Notation

<u>Symbol</u>	<u>Meaning</u>
←	Is transferred to
Λ	Boolean AND
V	Boolean OR
•	Boolean exclusive OR
— (Overline)	Boolean NOT
:	Concatenation
+	Arithmetic plus
—	Arithmetic minus
X	Arithmetic multiply

**Table A-2. Register Notation**

<u>Abbreviation</u>	<u>Meaning</u>
ACCA or A	Accumulator A
ACCB or B	Accumulator B
ACCA:ACCB or D	Double accumulator D
ACCX	Either accumulator A or B
CCR or CC	Condition code register
DPR or DP	Direct page register
EA	Effective address
IFF	If and only if
IX or X	Index register X
IY or Y	Index register Y
LSN	Least significant nibble
M	Memory location
MI	Memory immediate
MSN	Most significant nibble
PC	Program counter
R	A register before the operation
R'	A register after the operation
TEMP	Temporary storage location
xxH	Most significant byte of any 16-bit register
xxL	Least significant byte of any 16-bit register
Sp or S	Hardware Stack pointer
Us or U	User Stack pointer
P	A memory argument with Immediate, Direct, Extended, and Indexed addressing modes
Q	A read-modify-write argument with Direct, Indexed, and Extended addressing modes
( )	The data pointed to by the enclosed (16-bit address)
dd	8-bit branch offset
DDDD	16-bit branch offset
#	Immediate value follows
\$	Hexadecimal value follows
[ ]	Indirection
'	Indicates indexed addressing

# ABX

Add Accumulator B into Index Register X

# ABX

**Source Form:** ABX

**Operation:**  $IX' \leftarrow IX + ACCB$

**Condition Codes:** Not affected.

**Description:** Add the 8-bit unsigned value in accumulator B into index register X.

**Addressing Mode:** Inherent

# ADC

## Add with Carry into Register

# ADC

**Source Forms:** ADCA P; ADCB P

**Operation:**  $R' \leftarrow R + M + C$

**Condition Codes:** H — Set if a half-carry is generated; cleared otherwise.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Set if an overflow is generated; cleared otherwise.  
C — Set if a carry is generated; cleared otherwise.

**Description:** Adds the contents of the C (carry) bit and the memory byte into an 8-bit accumulator.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

## ADD (8-Bit)

Add Memory into Register

## ADD (8-Bit)

**Source Forms:** ADDA P; ADDB P

**Operation:**  $R' \leftarrow R + M$

**Condition Codes:**

- H — Set if a half-carry is generated; cleared otherwise.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Set if an overflow is generated; cleared otherwise.
- C — Set if a carry is generated; cleared otherwise.

**Description:** Adds the memory byte into an 8-bit accumulator.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

## **ADD (16-Bit)**

**Add Memory Into Register**

## **ADD (16-Bit)**

**Source Forms:** ADDD P

**Operation:**  $R' \leftarrow R + M:M + 1$

**Condition Codes:**

- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Set if an overflow is generated; cleared otherwise.
- C — Set if a carry is generated; cleared otherwise.

**Description:** Adds the 16-bit memory value into the 16-bit accumulator

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# AND

## Logical AND Memory into Register

# AND

**Source Forms:** ANDA P; ANDB P

**Operation:**  $R' \leftarrow R \Delta M$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** Performs the logical AND operation between the contents of an accumulator and the contents of memory location M and the result is stored in the accumulator.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# **AND** Logical AND Immediate Memory Into Condition Code Register **AND**

**Source Form:** ANDCC #xx

**Operation:**  $R' \leftarrow R \wedge MI$

**Condition Codes:** Affected according to the operation.

**Description:** Performs a logical AND between the condition code register and the immediate byte specified in the instruction and places the result in the condition code register.

**Addressing Mode:** Immediate




# ASL

## Arithmetic Shift Left

# ASL

**Source Forms:** ASL Q; ASLA; ASLB

**Operation:**



The diagram shows a horizontal row of eight squares representing bits b7, b6, b5, b4, b3, b2, b1, and b0. To the left of the first square (b7), an arrow points left towards the label 'C'. To the right of the last square (b0), an arrow points right towards the label '0'. Below the squares, a long arrow points from the position of b7 to the position of b0, indicating a left shift of all bits.

**Condition Codes:**

- H — Undefined
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Loaded with the result of the exclusive OR of bits six and seven of the original operand.
- C — Loaded with bit seven of the original operand.

**Description:** Shifts all bits of the operand one place to the left. Bit zero is loaded with a zero. Bit seven is shifted into the C (carry) bit.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

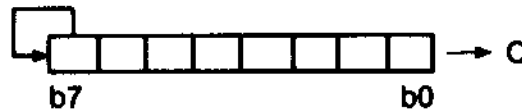
# ASR

## Arithmetic Shift Right

# ASR

**Source Forms:** ASR Q; ASRA; ASRB

**Operation:**



**Condition Codes:**

- H — Undefined.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Not affected.
- C — Loaded with bit zero of the original operand.

**Description:** Shifts all bits of the operand one place to the right. Bit seven is held constant. Bit zero is shifted into the C (carry) bit.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

# BCC

Branch on Carry Clear

# BCC

**Source Forms:** BCC dd; LBCC DDDD

**Operation:** TEMP ← MI  
IFF C = 0 then PC' ← PC + TEMP

**Condition Codes:** Not affected.

**Description:** Tests the state of the C (carry) bit and causes a branch if it is clear.

**Addressing Mode:** Relative

**Comments:** Equivalent to BHS dd; LBHS DDDD

# BCS

## Branch on Carry Set

# BCS

**Source Forms:** BCS dd; LBCS DDDD

**Operation:**  $TEMP \leftarrow MI$   
IFF  $C = 1$  then  $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Tests the state of the C (carry) bit and causes a branch if it is set.

**Addressing Mode:** Relative

**Comments:** Equivalent to BLO dd; LBLO DDDD

# BEQ

## Branch on Equal

# BEQ

**Source Forms:** BEQ dd; LBEQ DDDD

**Operation:** TEMP  $\leftarrow$  MI  
IFF Z = 1 then PC'  $\leftarrow$  PC + TEMP

**Condition Codes:** Not affected.

**Description:** Tests the state of the Z (zero) bit and causes a branch if it is set. When used after a subtract or compare operation, this instruction will branch if the compared values, signed or unsigned, were exactly the same.

**Addressing Mode:** Relative

# BGE

Branch on Greater than or Equal to Zero

# BGE

**Source Forms:** BGE dd; LBGE DDDD

**Operation:** TEMP ← MI  
IFF [N ⊕ V] = 0 then PC' ← PC + TEMP

**Condition Codes:** Not affected.

**Description:** Causes a branch if the N (negative) bit and the V (overflow) bit are either both set or both clear. That is, branch if the sign of a valid twos complement result is, or would be, positive. When used after a subtract or compare operation on twos complement values, this instruction will branch if the register was greater than or equal to the memory operand.

**Addressing Mode:** Relative

# BGT

## Branch on Greater

# BGT

**Source Forms:** BGT dd; LBGT DDDD

**Operation:**  $TEMP \leftarrow MI$   
IFF  $Z \wedge [N \oplus V] = 0$  then  $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Causes a branch if the N (negative) bit and V (overflow) bit are either both set or both clear and the Z (zero) bit is clear. In other words, branch if the sign of a valid twos complement result is, or would be, positive and not zero. When used after a subtract or compare operation on twos complement values, this instruction will branch if the register was greater than the memory operand.

**Addressing Mode:** Relative

# BHI

## Branch If Higher

# BHI

**Source Forms:** BHI dd; LBHI DDDD

**Operation:**  $TEMP \leftarrow M1$   
IFF  $[C \vee Z] = 0$  then  $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Causes a branch if the previous operation caused neither a carry nor a zero result. When used after a subtract or compare operation on unsigned binary values, this instruction will branch if the register was higher than the memory operand.

**Addressing Mode:** Relative

**Comments:** Generally not useful after INC/DEC, LD/TST, and TST/CLR/COM instructions.



# BHS

## Branch If Higher or Same

# BHS

**Source Forms:** BHS dd; LBHS DDDD

**Operation:**  $TEMP \leftarrow MI$   
IFF  $C = 0$  then  $PC' \leftarrow PC + MI$

**Condition Codes:** Not affected.

**Description:** Tests the state of the C (carry) bit and causes a branch if it is clear. When used after a subtract or compare on unsigned binary values, this instruction will branch if the register was higher than or the same as the memory operand.

**Addressing Mode:** Relative

**Comments:** This is a duplicate assembly-language mnemonic for the single machine instruction BCC. Generally not useful after INC/DEC, LD/ST, and TST/CLR/COM instructions.

# BIT

## Bit Test

# BIT

**Source Form:** Bit P

**Operation:**  $TEMP \leftarrow R \wedge M$

**Condition Codes:**

- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Always cleared.
- C — Not affected.

**Description:** Performs the logical AND of the contents of accumulator A or B and the contents of memory location M and modifies the condition codes accordingly. The contents of accumulator A or B and memory location M are not affected.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# BLE

Branch on Less than or Equal to Zero

# BLE

**Source Forms:** BLE dd; LBLE DDDD

**Operation:** TEMP ← MI  
IFF  $Z \vee [N \oplus V] = 1$  then  $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Causes a branch if the exclusive OR of the N (negative) and V (overflow) bits is 1 or if the Z (zero) bit is set. That is, branch if the sign of a valid twos complement result is, or would be, negative. When used after a subtract or compare operation on twos complement values, this instruction will branch if the register was less than or equal to the memory operand.

**Addressing Mode:** Relative

# BLO

Branch on Lower

# BLO

**Source Forms:** BLO dd; LBLO DDDD

**Operation:** TEMP  $\leftarrow$  MI  
IFF C = 1 then PC'  $\leftarrow$  PC + TEMP

**Condition Codes:** Not affected.

**Description:** Tests the state of the C (carry) bit and causes a branch if it is set. When used after a subtract or compare on unsigned binary values, this instruction will branch if the register was lower than the memory operand.

**Addressing Mode:** Relative

**Comments:** This is a duplicate assembly-language mnemonic for the single machine instruction BCS. Generally not useful after INC/DEC, LD/ST, and TST/CLR/COM instructions.

# BLS

## Branch on Lower or Same

# BLS

**Source Forms:** BLS dd; LBSL DDDD

**Operation:**  $TEMP \leftarrow MI$   
 $IFF (C \vee Z) = 1 \text{ then } PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Causes a branch if the previous operation caused either a carry or a zero result. When used after a subtract or compare operation on unsigned binary values, this instruction will branch if the register was lower than or the same as the memory operand.

**Addressing Mode:** Relative

**Comments:** Generally not useful after INC/DEC, LD/ST, and TST/CLR/COM instructions.

# BLT

## Branch on Less than Zero

# BLT

**Source Forms:** BLT dd; LBLT DDDD

**Operation:**  $TEMP \leftarrow MI$   
IFF  $[N \oplus V] = 1$  then  $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Causes a branch if either, but not both, of the N (negative) or V (overflow) bits is set. That is, branch if the sign of a valid two's complement result is, or would be, negative. When used after a subtract or compare operation on two's complement binary values, this instruction will branch if the register was less than the memory operand.

**Addressing Mode:** Relative

# BMI

## Branch on Minus

# BMI

**Source Forms:** BMI dd; LBMI DDDD

**Operation:** TEMP ← MI  
IFF N = 1 then PC' ← PC + TEMP

**Condition Codes:** Not affected.

**Description:** Tests the state of the N (negative) bit and causes a branch if set. That is, branch if the sign of the twos complement result is negative.

**Addressing Mode:** Relative

**Comments:** When used after an operation on signed binary values, this instruction will branch if the result is minus. It is generally preferred to use the LBLT instruction after signed operations.

# BNE

Branch Not Equal

# BNE

**Source Forms:** BNE dd; LBNE DDDD

**Operation:** TEMP ← Ml  
IFF Z = 0 then PC' ← PC + TEMP

**Condition Codes:** Not affected.

**Description:** Tests the state of the Z (zero) bit and causes a branch if it is clear. When used after a subtract or compare operation on any binary values, this instruction will branch if the register is, or would be, not equal to the memory operand.

**Addressing Mode:** Relative



# BPL

## Branch on Plus

# BPL

**Source Forms:** BPL dd; LBPL DDDD

**Operation:** TEMP ← MI  
IFF N = 0 then PC' ← PC + TEMP

**Condition Codes:** Not affected.

**Description:** Tests the state of the N (negative) bit and causes a branch if it is clear. That is, branch if the sign of the twos complement result is positive.

**Addressing Mode:** Relative

**Comments:** When used after an operation on signed binary values, this instruction will branch if the result (possibly invalid) is positive. It is generally preferred to use the BGE instruction after signed operations.

# BRA

Branch Always

# BRA

**Source Forms:** BRA dd; LBRA DDDD

**Operation:**  $TEMP \leftarrow MI$   
 $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Causes an unconditional branch.

**Addressing Mode:** Relative

# BRN

Branch Never

# BRN

**Source Forms:** BRN dd; LBRN DDDD

**Operation:** TEMP ← MI

**Condition Codes:** Not affected.

**Description:** Does not cause a branch. This instruction is essentially a no operation, but has a bit pattern logically related to branch always.

**Addressing Mode:** Relative

# BSR

## Branch to Subroutine

# BSR

**Source Forms:** BSR dd; LBSR DDDD

**Operation:**  $TEMP \leftarrow MI$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCH$   
 $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** The program counter is pushed onto the stack. The program counter is then loaded with the sum of the program counter and the offset.

**Addressing Mode:** Relative

**Comments:** A return from subroutine (RTS) instruction is used to reverse this process and must be the last instruction executed in a subroutine.

# BVC

## Branch on Overflow Clear

# BVC

**Source Forms:** BVC dd; LBVC DDDD

**Operation:**  $TEMP \leftarrow MI$   
IFF  $V = 0$  then  $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Tests the state of the V (overflow) bit and causes a branch if it is clear. That is, branch if the twos complement result was valid. When used after an operation on twos complement binary values, this instruction will branch if there was no overflow.

**Addressing Mode:** Relative

# BVS

## Branch on Overflow Set

# BVS

**Source Forms:** BVS dd; LBVS DDDD

**Operation:**  $TEMP \leftarrow MI$   
IFF  $V = 1$  then  $PC' \leftarrow PC + TEMP$

**Condition Codes:** Not affected.

**Description:** Tests the state of the V (overflow) bit and causes a branch if it is set. That is, branch if the twos complement result was invalid. When used after an operation on twos complement binary values, this instruction will branch if there was an overflow.

**Addressing Mode:** Relative

# CLR

Clear

# CLR

**Source Form:** CLR Q

**Operation:** TEMP ← M  
M ← 0016

**Condition Codes:** H — Not affected.  
N — Always cleared.  
Z — Always set.  
V — Always cleared.  
C — Always cleared.

**Description:** Accumulator A or B or memory location M is loaded with 00000000.  
Note that the EA is read during this operation.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

## **CMP (8-Bit)**

Compare Memory from Register

## **CMP (8-Bit)**

**Source Forms:** CMPA P; CMPB P

**Operation:**  $TEMP \leftarrow R - M$

**Condition Codes:**

- H — Undefined.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Set if an overflow is generated; cleared otherwise.
- C — Set if a borrow is generated; cleared otherwise.

**Description:** Compares the contents of memory location to the contents of the specified register and sets the appropriate condition codes. Neither memory location M nor the specified register is modified. The carry flag represents a borrow and is set to the inverse of the resulting binary carry.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed



# **CMP (16-Bit)** Compare Memory from Register **CMP (16-Bit)**

**Source Forms:** CMPD P; CMPX P; CMPY P; CMPU P; CMPS P

**Operation:**  $TEMP \leftarrow R - M:M + 1$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Set if an overflow is generated; cleared otherwise.  
C — Set if a borrow is generated; cleared otherwise.

**Description:** Compares the 16-bit contents of the concatenated memory locations  $M:M + 1$  to the contents of the specified register and sets the appropriate condition codes. Neither the memory locations nor the specified register is modified unless autoincrement or autodecrement are used. The carry flag represents a borrow and is set to the inverse of the resulting binary carry.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# COM

## Complement

# COM

**Source Forms:** COM Q; COMA; COMB

**Operation:**  $M' \leftarrow 0 + \bar{M}$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Always cleared.  
C — Always set.

**Description:** Replaces the contents of memory location M or accumulator A or B with its logical complement. When operating on unsigned values, only BEQ and BNE branches can be expected to behave properly following a COM instruction. When operating on twos complement values, all signed branches are available.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

# CWAI

Clear CC bits and Wait for Interrupt

# CWAI

Source Form:

CWAI #\$XX

E	F	H	I	N	Z	V	C
---	---	---	---	---	---	---	---

Operation:

CCR ← CCR  $\wedge$  MI (Possibly clear masks)

Set E (entire state saved)

SP' ← SP - 1, (SP) ← PCL

SP' ← SP - 1, (SP) ← PCH

SP' ← SP - 1, (SP) ← USL

SP' ← SP - 1, (SP) ← USH

SP' ← SP - 1, (SP) ← IYL

SP' ← SP - 1, (SP) ← IYH

SP' ← SP - 1, (SP) ← IXL

SP' ← SP - 1, (SP) ← IXH

SP' ← SP - 1, (SP) ← DPR

SP' ← SP - 1, (SP) ← ACCB

SP' ← SP - 1, (SP) ← ACCA

SP' ← SP - 1, (SP) ← CCR

Condition Codes:

Affected according to the operation.

Description:

This instruction ANDs an immediate byte with the condition code register which may clear the interrupt mask bits I and F, stacks the entire machine state on the hardware stack and then looks for an interrupt. When a non-masked interrupt occurs, no further machine state information need be saved before vectoring to the interrupt handling routine. This instruction replaced the MC6800 CLI WAI sequence, but does not place the buses in a high-impedance state. A  $\overline{\text{FIRQ}}$  (fast interrupt request) may enter its interrupt handler with its entire machine state saved. The RTI (return from interrupt) instruction will automatically return the entire machine state after testing the E (entire) bit of the recovered condition code register.

Addressing Mode:

Immediate

Comments:

The following immediate values will have the following results:

FF = enable neither

EF = enable  $\overline{\text{IRQ}}$

BF = enable  $\overline{\text{FIRQ}}$

AF = enable both

# DAA

## Decimal Addition Adjust

# DAA

**Source Form:** DAA

**Operation:**  $ACCA' \leftarrow ACCA + CF(MSN):CF(LSN)$   
where CF is a Correction Factor, as follows: the CF for each nibble (BCD) digit is determined separately, and is either 6 or 0.

**Least Significant Nibble**

$CF(LSN) = 6$  IFF 1)  $C = 1$   
or 2)  $LSN > 9$

**Most Significant Nibble**

$CF(MSN) = 6$  IFF 1)  $C = 1$   
or 2)  $MSN > 9$   
or 3)  $MSN > 8$  and  $LSN > 9$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Undefined.  
C — Set if a carry is generated or if the carry bit was set before the operation; cleared otherwise.

**Description:** The sequence of a single-byte add instruction on accumulator A (either ADDA or ADCA) and a following decimal addition adjust instruction results in a BCD addition with an appropriate carry bit. Both values to be added must be in proper BCD form (each nibble such that:  $0 \leq \text{nibble} \leq 9$ ). Multiple-precision addition must add the carry generated by this decimal addition adjust into the next higher digit during the add operation (ADCA) immediately prior to the next decimal addition adjust.

**Addressing Mode:** Inherent

# DEC

## Decrement

# DEC

**Source Forms:** DEC Q; DECA; DECB

**Operation:**  $M' \leftarrow M - 1$

**Condition Codes:**

- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Set if the original operand was 10000000; cleared otherwise.
- C — Not affected.

**Description:** Subtract one from the operand. The carry bit is not affected, thus allowing this instruction to be used as a loop counter in multiple-precision computations. When operating on unsigned values, only BEQ and BNE branches can be expected to behave consistently. When operating on twos complement values, all signed branches are available.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

# EOR

## Exclusive OR

# EOR

**Source Forms:** EORA P; EORB P

**Operation:**  $R' \leftarrow R \oplus M$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** The contents of memory location M is exclusive ORed into an 8-bit register.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# EXG

## Exchange Registers

# EXG

**Source Form:** EXG R1,R2

**Operation:** R1 ← R2

**Condition Codes:** Not affected (unless one of the registers is the condition code register).

**Description:** Exchanges data between two designated registers. Bits 3-0 of the postbyte define one register, while bits 7-4 define the other, as follows:

0000 = A:B	1000 = A
0001 = X	1001 = B
0010 = Y	1010 = CCR
0011 = US	1011 = DPR
0100 = SP	1100 = Undefined
0101 = PC	1101 = Undefined
0110 = Undefined	1110 = Undefined
0111 = Undefined	1111 = Undefined

Only like size registers may be exchanged. (8-bit with 8-bit or 16-bit with 16-bit.)

**Addressing Mode:** Immediate

# INC

## Increment

# INC

**Source Forms:** INC Q; INCA; INCB

**Operation:**  $M' \leftarrow M + 1$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Set if the original operand was 01111111; cleared otherwise.  
C — Not affected.

**Description:** Adds to the operand. The carry bit is not affected, thus allowing this instruction to be used as a loop counter in multiple-precision computations. When operating on unsigned values, only the BEQ and BNE branches can be expected to behave consistently. When operating on two's complement values, all signed branches are correctly available.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed



# JMP

Jump

# JMP

**Source Form:** JMP EA

**Operation:**  $PC' \leftarrow EA$

**Condition Codes:** Not affected.

**Description:** Program control is transferred to the effective address.

**Addressing Modes:** Extended  
Direct  
Indexed

# JSR

## Jump to Subroutine

# JSR

**Source Form:** JSR EA

**Operation:**  $SP' \leftarrow SP - 1, (SP) \leftarrow PCL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCH$   
 $PC' \leftarrow EA$

**Condition Codes:** Not affected.

**Description:** Program control is transferred to the effective address after storing the return address on the hardware stack. A RTS instruction should be the last executed instruction of the subroutine.

**Addressing Modes:** Extended  
Direct  
Indexed

## LD (8-Bit)

Load Register from Memory

## LD (8-Bit)

**Source Forms:** LDA P; LDB P

**Operation:**  $R' \leftarrow M$

**Condition Codes:** H — Not affected.  
N — Set if the loaded data is negative; cleared otherwise.  
Z — Set if the loaded data is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** Loads the contents of memory location M into the designated register.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

## LD (16-Bit)

Load Register from Memory

## LD (16-Bit)

**Source Forms:** LDD P; LDX P; LDY P; LDS P; LDU P

**Operation:**  $R' \leftarrow M:M + 1$

**Condition Codes:** H — Not affected.  
N — Set if the loaded data is negative; cleared otherwise.  
Z — Set if the loaded data is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** Load the contents of the memory location  $M:M + 1$  into the designated 16-bit register.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# LEA

## Load Effective Address

# LEA

**Source Forms:** LEAX, LEAY, LEAS, LEAU

**Operation:**  $R' \leftarrow EA$

**Condition Codes:** H — Not affected.  
N — Not affected.  
Z — LEAX, LEAY: Set if the result is zero; cleared otherwise.  
LEAS, LEAU: Not affected.  
V — Not affected.  
C — Not affected.

**Description:** Calculates the effective address from the indexed addressing mode and places the address in an indexable register.

LEAX and LEAY affect the Z (zero) bit to allow use of these registers as counters and for MC6800 INX/DEX compatibility.

LEAU and LEAS do not affect the Z bit to allow cleaning up the stack while returning the Z bit as a parameter to a calling routine, and also for MC6800 INS/DES compatibility.

**Addressing Mode:** Indexed

**Comments:** Due to the order in which effective addresses are calculated internally, the LEAX,  $X++$  and LEAX,  $X+$  do not add 2 and 1 (respectively) to the X register; but instead leave the X register unchanged. This also applies to the Y, U, and S registers. For the expected results, use the faster instruction LEAX 2, X and LEAX 1, X.

Some examples of LEA instruction uses are given in the following table.


Instruction		Operation	Comment
LEAX	10, X	$X + 10 - X$	Adds 5-bit constant 10 to X
LEAX	500, X	$X + 500 - X$	Adds 16-bit constant 500 to X
LEAY	A, Y	$Y + A - Y$	Adds 8-bit accumulator to Y
LEAY	D, Y	$Y + D - Y$	Adds 16-bit D accumulator to Y
LEAU	-10, U	$U - 10 - U$	Subtracts 10 from U
LEAS	-10, S	$S - 10 - S$	Used to reserve area on stack
LEAS	10, S	$S + 10 - S$	Used to 'clean up' stack
LEAX	5, S	$S + 5 - X$	Transfers as well as adds

# LSL

## Logical Shift Left

# LSL

**Source Forms:** LSL Q; LSLA; LSLB

**Operation:** C ←  0  
b7 b0

**Condition Codes:** H — Undefined.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Loaded with the result of the exclusive OR of bits six and seven of the original operand.  
C — Loaded with bit seven of the original operand.

**Description:** Shifts all bits of accumulator A or B or memory location M one place to the left. Bit zero is loaded with a zero. Bit seven of accumulator A or B or memory location M is shifted into the C (carry) bit.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

**Comments:** This is a duplicate assembly-language mnemonic for the single machine instruction ASL.

## LSR

# MUL

## Multiply

# MUL

**Source Form:** MUL

**Operation:** ACCA':ACCB' — ACCA x ACCB

**Condition Codes:** H — Not affected.  
N — Not affected.  
Z — Set if the result is zero; cleared otherwise.  
V — Not affected.  
C — Set if ACCB bit 7 of result is set; cleared otherwise.

**Description:** Multiply the unsigned binary numbers in the accumulators and place the result in both accumulators (ACCA contains the most-significant byte of the result). Unsigned multiply allows multiple-precision operations.

**Addressing Mode:** Inherent

**Comments:** The C (carry) bit allows rounding the most-significant byte through the sequence: MUL, ADCA #0.



# NEG

## Negate

# NEG

**Source Forms:** NEG Q; NEGA; NEGB

**Operation:**  $M' \leftarrow 0 - M$

**Condition Codes:** H — Undefined.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Set if the original operand was 10000000.  
C — Set if a borrow is generated; cleared otherwise.

**Description:** Replaces the operand with its twos complement. The C (carry) bit represents a borrow and is set to the inverse of the resulting binary carry. Note that 80<sub>16</sub> is replaced by itself and only in this case is the V (overflow) bit set. The value 00<sub>16</sub> is also replaced by itself, and only in this case is the C (carry) bit cleared.

**Addressing Modes:** Inherent  
Extended  
Direct

# NOP

No Operation

# NOP

**Source Form:** NOP

**Operation:** Not affected.

**Condition Codes:** This instruction causes only the program counter to be incremented.  
No other registers or memory locations are affected.

**Addressing Mode:** Inherent

# OR

## Inclusive OR Memory Into Register

# OR

**Source Forms:** ORA P; ORB P

**Operation:**  $R' \leftarrow R \vee M$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** Performs an inclusive OR operation between the contents of accumulator A or B and the contents of memory location M and the result is stored in accumulator A or B.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

**OR****Inclusive OR Memory Immediate Into Condition Code Register****OR****Source Form:** ORCC #XX**Operation:**  $R' \leftarrow R \vee MI$ **Condition Codes:** Affected according to the operation.

**Description:** Performs an inclusive OR operation between the contents of the condition code registers and the immediate value, and the result is placed in the condition code register. This instruction may be used to set interrupt masks (disable interrupts) or any other bit(s).

**Addressing Mode:** Immediate

# PSHS

## Push Registers on the Hardware Stack

# PSHS

**Source Form:** PSHS register list  
PSHS #LABEL  
Postbyte:

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

push order----->

**Operation:**

IFF b7 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow PCL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCH$

IFF b6 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow USL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow USH$

IFF b5 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow IYL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IYH$

IFF b4 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow IXL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IXH$

IFF b3 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow DPR$

IFF b2 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow ACCB$

IFF b1 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow ACCA$

IFF b0 of postbyte set, then:  $SP' \leftarrow SP - 1, (SP) \leftarrow CCR$

**Condition Codes:** Not affected.

**Description:** All, some, or none of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself).

**Addressing Mode:** Immediate

**Comments:** A single register may be placed on the stack with the condition codes set by doing an autodecrement store onto the stack (example: STX, --S).

# PSHU

## Push Registers on the User Stack

# PSHU

**Source Form:** PSHU register list

PSHU #LABEL

Postbyte:

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

push order----->

**Operation:**

IFF b7 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow PCL$   
 $US' \leftarrow US - 1, (US) \leftarrow PCH$

IFF b6 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow SPL$   
 $US' \leftarrow US - 1, (US) \leftarrow SPH$

IFF b5 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow IYL$   
 $US' \leftarrow US - 1, (US) \leftarrow IYH$

IFF b4 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow IXL$   
 $US' \leftarrow US - 1, (US) \leftarrow IXH$

IFF b3 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow DPR$

IFF b2 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow ACCB$

IFF b1 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow ACCA$

IFF b0 of postbyte set, then:  $US' \leftarrow US - 1, (US) \leftarrow CCR$

**Condition Codes:** Not affected.

**Description:** All, some, or none of the processor registers are pushed onto the user stack (with the exception of the user stack pointer itself).

**Addressing Mode:** Immediate

**Comments:** A single register may be placed on the stack with the condition codes set by doing an autodecrement store onto the stack (example: STX, - - U).

# PULS

## Pull Registers from the Hardware Stack

# PULS

**Source Form:** PULS register list

PULS #LABEL

Postbyte:

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

←-----pull order

**Operation:**

IFF b0 of postbyte set, then:  $CCR' \leftarrow (SP), SP' \leftarrow SP + 1$   
IFF b1 of postbyte set, then:  $ACCA' \leftarrow (SP), SP' \leftarrow SP + 1$   
IFF b2 of postbyte set, then:  $ACCB' \leftarrow (SP), SP' \leftarrow SP + 1$   
IFF b3 of postbyte set, then:  $DPR' \leftarrow (SP), SP' \leftarrow SP + 1$   
IFF b4 of postbyte set, then:  $IXH' \leftarrow (SP), SP' \leftarrow SP + 1$   
 $IXL' \leftarrow (SP), SP' \leftarrow SP + 1$   
IFF b5 of postbyte set, then:  $IYH' \leftarrow (SP), SP' \leftarrow SP + 1$   
 $IYL' \leftarrow (SP), SP' \leftarrow SP + 1$   
IFF b6 of postbyte set, then:  $USH' \leftarrow (SP), SP' \leftarrow SP + 1$   
 $USL' \leftarrow (SP), SP' \leftarrow SP + 1$   
IFF b7 of postbyte set, then:  $PCH' \leftarrow (SP), SP' \leftarrow SP + 1$   
 $PCL' \leftarrow (SP), SP' \leftarrow SP + 1$

**Condition Codes:** May be pulled from stack; not affected otherwise.

**Description:** All, some, or none of the processor registers are pulled from the hardware stack (with the exception of the hardware stack pointer itself).

**Addressing Mode:** Immediate

**Comments:** A single register may be pulled from the stack with condition codes set by doing an autoincrement load from the stack (example:  $LDX, S++$ ).

# PULU

## Pull Registers from the User Stack

# PULU

**Source Form:**

PULU register list

PULU #LABEL

Postbyte:

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

←----- pull order

**Operation:**IFF b0 of postbyte set, then: CCR'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IFF b1 of postbyte set, then: ACCA'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IFF b2 of postbyte set, then: ACCB'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IFF b3 of postbyte set, then: DPR'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IFF b4 of postbyte set, then: IXH'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IXL'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IFF b5 of postbyte set, then: IYH'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IYL'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IFF b6 of postbyte set, then: SPH'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1SPL'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1IFF b7 of postbyte set, then: PCH  $\leftarrow$  (US), US'  $\leftarrow$  US + 1PCL'  $\leftarrow$  (US), US'  $\leftarrow$  US + 1**Condition Codes:** May be pulled from stack; not affected otherwise.**Description:** All, some, or none of the processor registers are pulled from the user stack (with the exception of the user stack pointer itself).**Addressing Mode:** Immediate**Comments:** A single register may be pulled from the stack with condition codes set by doing an autoincrement load from the stack (example: LDX,U++).



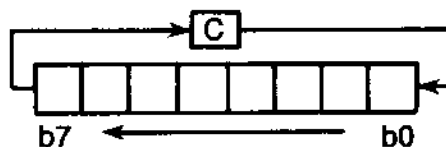
# ROL

Rotate Left

# ROL

**Source Forms:** ROL Q; ROLA; ROLB

**Operation:**



**Condition Codes:**

- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Loaded with the result of the exclusive OR of bits six and seven of the original operand.
- C — Loaded with bit seven of the original operand.

**Description:** Rotates all bits of the operand one place left through the C (carry) bit. This is a 9-bit rotation.

**Addressing Mode:** Inherent  
Extended  
Direct  
Indexed

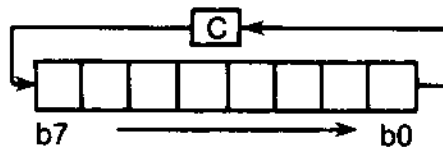
# ROR

Rotate Right

# ROR

**Source Forms:** ROR Q; RORA; RORB

**Operation:**



**Condition Codes:**

- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Not affected.
- C — Loaded with bit zero of the previous operand.

**Description:** Rotates all bits of the operand one place right through the C (carry) bit. This is a 9-bit rotation.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

# RTI

## Return from Interrupt

# RTI

**Source Form:** RTI

**Operation:**  $CCR' \leftarrow (SP), SP' \leftarrow SP + 1$ , then

IFF CCR bit E is set, then:

$ACCA'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$ACCB'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$DPR'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$IXH'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$IXL'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$IYH'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$IYL'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$USH'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$USL'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$PCH'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$PCL'$	$\leftarrow (SP), SP' \leftarrow SP + 1$

IFF CCR bit E is clear, then:

$PCH'$	$\leftarrow (SP), SP' \leftarrow SP + 1$
$PCL'$	$\leftarrow (SP), SP' \leftarrow SP + 1$

**Condition Codes:** Recovered from the stack.

**Description:** The saved machine state is recovered from the hardware stack and control is returned to the interrupted program. If the recovered E (entire) bit is clear, it indicates that only a subset of the machine state was saved (return address and condition codes) and only that subset is recovered.

**Addressing Mode:** Inherent

# RTS

## Return from Subroutine

# RTS

**Source Form:** RTS

**Operation:**  $PCH' \leftarrow (SP), SP' \leftarrow SP + 1$   
 $PCL' \leftarrow (SP), SP' \leftarrow SP + 1$

**Condition Codes:** Not affected.

**Description:** Program control is returned from the subroutine to the calling program. The return address is pulled from the stack.

**Addressing Mode:** Inherent

# SBC

## Subtract with Borrow

# SBC

**Source Forms:** SBCA P; SBCB P

**Operation:**  $R' \leftarrow R - M - C$

**Condition Codes:** H — Undefined.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Set if an overflow is generated; cleared otherwise.  
C — Set if a borrow is generated; cleared otherwise.

**Description:** Subtracts the contents of memory location M and the borrow (in the C (carry) bit) from the contents of the designated 8-bit register, and places the result in that register. The C bit represents a borrow and is set to the inverse of the resulting binary carry.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# SEX

Sign Extended

# SEX

**Source Form:** SEX

**Operation:** If bit seven of ACCB is set then  $ACCA' \leftarrow FF_{16}$   
else  $ACCA' \leftarrow 00_{16}$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Not affected.  
C — Not affected.

**Description:** This instruction transforms a two's complement 8-bit value in accumulator B into a two's complement 16-bit value in the D accumulator.

**Addressing Mode:** Inherent

## ST (8-Bit)

Store Register into Memory

## ST (8-Bit)

**Source Forms:** STA P; STB P

**Operation:**  $M' \leftarrow R$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** Writes the contents of an 8-bit register into a memory location.

**Addressing Modes:** Extended  
Direct  
Indexed

## ST (16-Bit)

Store Register Into Memory

## ST (16-Bit)

**Source Forms:** STD P; STX P; STY P; STS P; STU P

**Operation:**  $M':M + 1' \leftarrow R$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** Writes the contents of a 16-bit register into two consecutive memory locations.

**Addressing Modes:** Extended  
Direct  
Indexed



## SUB (8-Bit)

Subtract Memory from Register

## SUB (8-Bit)

**Source Forms:** SUBA P; SUBB P

**Operation:**  $R' \leftarrow R - M$

**Condition Codes:** H — Undefined.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Set if the overflow is generated; cleared otherwise.  
C — Set if a borrow is generated; cleared otherwise.

**Description:** Subtracts the value in memory location M from the contents of a designated 8-bit register. The C (carry) bit represents a borrow and is set to the inverse of the resulting binary carry.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

## **SUB (16-Bit)** Subtract Memory from Register **SUB (16-Bit)**

**Source Forms:** SUBD P

**Operation:**  $R' \leftarrow R - M:M + 1$

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Set if the overflow is generated; cleared otherwise.  
C — Set if a borrow is generated; cleared otherwise.

**Description:** Subtracts the value in memory location M:M + 1 from the contents of a designated 16-bit register. The C (carry) bit represents a borrow and is set to the inverse of the resulting binary carry.

**Addressing Modes:** Immediate  
Extended  
Direct  
Indexed

# SWI

## Software Interrupt

# SWI

**Source Form:** SWI

**Operation:** Set E (entire state will be saved)  
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow USL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow USH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IYL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IYH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IXL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IXH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow DPR$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow ACCB$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow ACCA$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow CCR$   
Set I, F (mask interrupts)  
 $PC' \leftarrow (FFFA):(FFFB)$

**Condition Codes:** Not affected.

**Description:** All of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself), and control is transferred through the software interrupt vector. Both the normal and fast interrupts are masked (disabled).

**Addressing Mode:** Inherent

# SWI2

## Software Interrupt 2

# SWI2

**Source Form:** SWI2

**Operation:** Set E (entire state saved)  
SP' ← SP - 1, (SP) ← PCL  
SP' ← SP - 1, (SP) ← PCH  
SP' ← SP - 1, (SP) ← USL  
SP' ← SP - 1, (SP) ← USH  
SP' ← SP - 1, (SP) ← IYL  
SP' ← SP - 1, (SP) ← IYH  
SP' ← SP - 1, (SP) ← IXL  
SP' ← SP - 1, (SP) ← IXH  
SP' ← SP - 1, (SP) ← DPR  
SP' ← SP - 1, (SP) ← ACCB  
SP' ← SP - 1, (SP) ← ACCA  
SP' ← SP - 1, (SP) ← CCR  
PC' ← (FFF4):(FFF5)

**Condition Codes:** Not affected.

**Description:** All of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself), and control is transferred through the software interrupt 2 vector. This interrupt is available to the end user and must not be used in packaged software. This interrupt does not mask (disable) the normal and fast interrupts.

**Addressing Mode:** Inherent

# SWI3

## Software Interrupt 3

# SWI3

**Source Form:** SWI 3

**Operation:** Set E (entire state will be saved)  
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow USL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow USH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IYL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IYH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IXL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IXH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow DPR$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow ACCB$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow ACCA$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow CCR$   
 $PC' \leftarrow (FFF2):(FFF3)$

**Condition Codes:** Not affected.

**Description:** All of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself), and control is transferred through the software interrupt 3 vector. This interrupt does not mask (disable) the normal and fast interrupts.

**Addressing Mode:** Inherent

# SYNC

## Synchronize to External Event

# SYNC

**Source Form:** SYNC

**Operation:** Stop processing instructions

**Condition Codes:** Not affected.

**Description:** When a SYNC instruction is executed, the processor enters a synchronizing state, stops processing instructions, and waits for an interrupt. When an interrupt occurs, the synchronizing state is cleared and processing continues. If the interrupt is enabled, and it lasts three cycles or more, the processor will perform the interrupt routine. If the interrupt is masked or is shorter than three cycles, the processor simply continues to the next instruction. While in the synchronizing state, the address and data buses are in the high-impedance state.

This instruction provides software synchronization with a hardware process. Consider the following example for high-speed acquisition of data:

FAST	SYNC	WAIT FOR DATA
	Interrupt!	
	LDA	DISC DATA FROM DISC AND CLEAR INTERRUPT
	STA	,X+ PUT IN BUFFER
	DECB	COUNT IT, DONE?
	BNE	FAST GO AGAIN IF NOT.

The synchronizing state is cleared by any interrupt. Of course, enabled interrupts at this point may destroy the data transfer and, as such, should represent only emergency conditions.

The same connection used for interrupt-driven I/O service may also be used for high-speed data transfers by setting the interrupt mask and using the SYNC instruction as the above example demonstrates.

**Addressing Mode:** Inherent

# TFR

## Transfer Register to Register

# TFR

**Source Form:** TFR R1, R2

**Operation:** R1  $\rightarrow$  R2

**Condition Code:** Not affected unless R2 is the condition code register.

**Description:** Transfers data between two designated registers. Bits 7-4 of the postbyte define the source register, while bits 3-0 define the destination register, as follows:

0000 = A:B	1000 = A
0001 = X	1001 = B
0010 = Y	1010 = CCR
0011 = US	1011 = DPR
0100 = SP	1100 = Undefined
0101 = PC	1101 = Undefined
0110 = Undefined	1110 = Undefined
0111 = Undefined	1111 = Undefined

Only like size registers may be transferred. (8-bit to 8-bit, or 16-bit to 16-bit.)

**Addressing Mode:** Immediate

# TST

## Test

# TST

**Source Forms:** TST Q; TSTA; TSTB

**Operation:** TEMP ← M - 0

**Condition Codes:** H — Not affected.  
N — Set if the result is negative; cleared otherwise.  
Z — Set if the result is zero; cleared otherwise.  
V — Always cleared.  
C — Not affected.

**Description:** Set the N (negative) and Z (zero) bits according to the contents of memory location M, and clear the V (overflow) bit. The TST instruction provides only minimum information when testing unsigned values; since no unsigned value is less than zero, BLO and BLS have no utility. While BHI could be used after TST, it provides exactly the same control as BNE, which is preferred. The signed branches are available.

**Addressing Modes:** Inherent  
Extended  
Direct  
Indexed

**Comments:** The MC6800 processor clears the C (carry) bit.



## FIRQ

### Fast Interrupt Request (Hardware Interrupt)

## FIRQ

**Operation:** IFF F bit clear, then:  $SP' \leftarrow SP - 1$ ,  $(SP) \leftarrow PCL$   
 $SP' \leftarrow SP - 1$ ,  $(SP) \leftarrow PCH$   
Clear E (subset state is saved)  
 $SP' \leftarrow SP - 1$ ,  $(SP) \leftarrow CCR$   
Set F, I (mask further interrupts)  
 $PC' \leftarrow (FFF6):(FFF7)$

**Condition Codes:** Not affected.

**Description:** A FIRQ (fast interrupt request) with the F (fast interrupt request mask) bit clear causes this interrupt sequence to occur at the end of the current instruction. The program counter and condition code register are pushed onto the hardware stack. Program control is transferred through the fast interrupt request vector. An RTI (return from interrupt) instruction returns the processor to the original task. It is possible to enter the fast interrupt request routine with the entire machine state saved if the fast interrupt request occurs after a clear and wait for interrupt instruction. A normal interrupt request has lower priority than the fast interrupt request and is prevented from interrupting the fast interrupt request routine by automatic setting of the I (interrupt request mask) bit. This mask bit could then be reset during the interrupt routine if priority was not desired. The fast interrupt request allows operations on memory, TST, INC, DEC, etc. instructions without the overhead of saving the entire machine state on the stack.

**Addressing Mode:** Inherent

**$\overline{\text{IRQ}}$** **Interrupt Request (Hardware Interrupt)** **$\overline{\text{IRQ}}$** 

**Operation:** IFF I bit clear, then:

- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{PCL}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{PCH}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{USL}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{USH}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{IYL}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{IYH}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{IXL}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{IXH}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{DPR}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{ACCB}$
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{ACCA}$
- Set E (entire state saved)
- $\text{SP}' \leftarrow \text{SP} - 1, (\text{SP}) \leftarrow \text{CCR}$
- Set I (mask further  $\overline{\text{IRQ}}$  interrupts)
- $\text{PC}' \leftarrow (\text{FFF8}):(\text{FFF9})$

**Condition Codes:** Not affected.

**Description:** If the I (interrupt request mask) bit is clear, a low level on the  $\overline{\text{IRQ}}$  input causes this interrupt sequence to occur at the end of the current instruction. Control is returned to the interrupted program using a RTI (return from interrupt) instruction. A  $\overline{\text{FIRQ}}$  (fast interrupt request) may interrupt a normal  $\overline{\text{IRQ}}$  (interrupt request) routine and be recognized anytime after the interrupt vector is taken.

**Addressing Mode:** Inherent

# NMI

## Non-Maskable Interrupt (Hardware Interrupt)

# NMI

### Operation:

$SP' \leftarrow SP - 1, (SP) \leftarrow PCL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow PCH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow USL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow USH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IYL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IYH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IXL$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow IXH$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow DPR$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow ACCB$   
 $SP' \leftarrow SP - 1, (SP) \leftarrow ACCA$   
Set E (entire state save)  
 $SP' \leftarrow SP - 1, (SP) \leftarrow CCR$   
Set I, F (mask interrupts)  
 $PC' \leftarrow (FFFC):(FFFD)$

**Condition Codes:** Not affected.

### Description:

A negative edge on the NMI (non-maskable interrupt) input causes all of the processor's registers (except the hardware stack pointer) to be pushed onto the hardware stack, starting at the end of the current instruction. Program control is transferred through the NMI vector. Successive negative edges on the NMI input will cause successive NMI operations. Non-maskable interrupt operation can be internally blocked by a RESET operation and any non-maskable interrupt that occurs will be latched. If this happens, the non-maskable interrupt operation will occur after the first load into the stack pointer (LDS; TFR r,s; EXG r,s; etc.) after RESET.

**Addressing Mode:** Inherent

# RESTART

Restart (Hardware Interrupt)

# RESTART

**Operation:**             $CCR' \leftarrow X1X1XXXX$   
                          $DPR' \leftarrow 00_{16}$   
                          $PC' \leftarrow (FFFE):(FFFF)$

**Condition Codes:**    Not affected.

**Description:**        The processor is initialized (required after power-on) to start program execution. The starting address is fetched from the restart vector.

**Addressing Mode:**    Extended Indirect