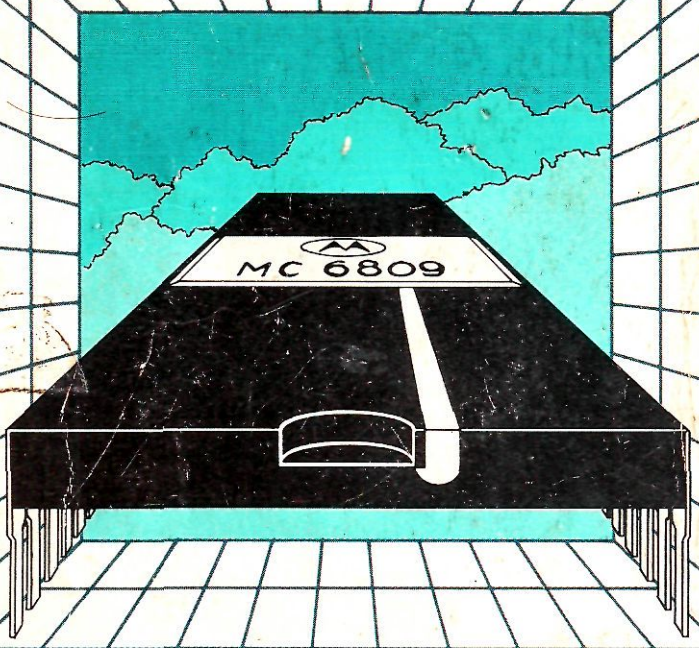
TAB 1209

$6.95

THE MC6809 COOKBOOK



THE how-to-do-it handbook for MC6809 users . . . hardware, software, architecture, and applications!

by carl d. warren

THE

MC6809 COOKBOOK

by carl d. warren

Red text is the original error correction in the book
For corrections to the PDF or owners of copyright of the book want to remove it, please contact by luis45ccs@hotmail.com.
This PDF is made only for the preservation of information of the color computer, not to affect the copyright, will always be preferable to the original purchase, the PDF also aids in searches in the book.
Or if you can not easily acquire on the market, as in my case in Venezuela.
Some pages were scan at 300 DPI (if font small) but most are 200 DPI, and ABBYY FineReader 8.0 Professional Edition

Dedication

For ACE
No questions asked

THE

by carl d. warren

TAB TAB BOOKS Inc.
BLUE RIDGE SUMMIT. PA. 17214

FIRST EDITION

SECOND PRINTING

Copyright © 1980 by TAB BOOKS Inc.

Printed in the United States of America

Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. No liability is assumed with respect to the use of the information herein.

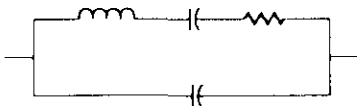
Library of Congress Cataloging in Publication Data

Warren, Carl D
The MC6809 cookbook.

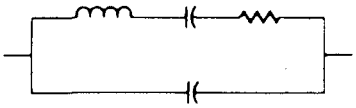
Includes index.
1. Motorola 6809 (Computer) I.Title

QA76.8.M67W37 001.64 80-23359
ISBN 0-8306-9683-0
ISBN 0-8306-1209-2 (pbk.)

Ⓜ is a trademark of Motorola Inc.



Contents



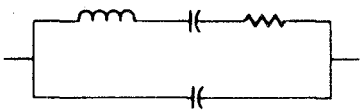
Acknowledgments

When any work such as this is embarked upon, it requires a massive amount of support from a variety of sources. It was necessary to rely upon the suggestions and resources of a number of people and companies. Among these people are; Ron Denchfield of AMI who supplied, most of the figures and the programmer's card, Tim Ahrens, Bill Clendinning and Irwin Carroll of Motorola, who provided various tables and important suggestions, along with the 6809D4 evaluation unit.

Since this book is about computer technology, it is only appropriate that it was created with aid of a computer. The manuscript was prepared on a Heath H-89 microcomputer, and printed on an Epson TX-80 dot matrix printer. The software that was used consisted of a variety of products. Among these were the editor and text formatter available from the Heath users' group (HUG), the PIE editor from the Software Toolworks, and a very special program called COPY that permits interchanging software created under HDOS to CP/M compatible files. This unique piece of magic was created by Bob Mathias, a genius of our times. Other software was supplied courtesy of Tony Gold at Lifeboat Assoc. These consisted of Organic software's Textwriter III for text formatting and Digital Research's CP/M optimized for a 4200H base.

Special thanks is reserved for the finest managing editor in the magazine industry today, Jordan Backler. It is because of his suggestions, coupled with those from fellow EDN editors Bob Peterson, and Ed Teja, that this work is as concise as it is.

My wife Anne and daughter Tami played probably the most important part in the creation of this book—putting up with the writing process and making sure the coffee was always available.



Preface

As a result of the profliferation of microprocessors (µP), since 1977 hardware and software designers have been able to extend their capabilities in terms of creating useful products for everyday life. Each day, new processor introductions are opening up even more exciting vistas. Unfortunately, there is a problem associated with the introduction of the newer devices: how to correctly use them for maximum benefit and efficiency.

This book, like many of its type, attempts to give the engineer or technician an expert command of the fundamentals of the 6809 microprocessor (µP), and the basic skills for writing 6809 assembly language level code. In systematic fashion, it proceeds from analysis of the microprocessors design to its important electrical characteristics. It continues with discussion on matters like internal logic, comparison to the 6800, interfacing to peripherals, software architecture, addressing techniques and the instruction set. It concludes with advice on how to build or make use of existing 6809 based systems. Further enhancing *The MC6809 Cookbook's* usefulness is the inclusion of a programmer's card, provided courtesy of American Microsystems Inc. (AMI).

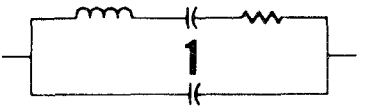
The MC6809 Cookbook may be studied as a course, proceeding from the simple to the more complex. However, it should be more appropriate viewed as a reference source to be called upon as necessary. Therefore, I have kept the idea of a compact and concise reference work utmost in my mind while creating this book. As a

result of this goal, the diversified contents are readily identifiable to facilitate the finding of specific principles or functions associated with the 6809 µP.

The MC6809 Cookbook aims to be comprehensive without being cumbersome. It seeks in all areas to be exact, clear and succinct.

Throughout this book, µP, and µC are used to mean microprocessor and microcomputer, respectively. These are stylistic nuances used at EDN magazine, and permit brevity without being imprecise.

carl d. warren



General Descriptions

The 6809 /xP, developed by Motorola and second sourced by American Microsystems Inc. (AMI), is a high performance multifaceted device. It is considered by many industry observers and Motorola to be the interim processor between 8-bit and 16-bit devices. The general design philosophy of the device seems to support this conjecture in that it permits the handling of 16-bit registers with powerful instructions.

INTRODUCTION TO THE 6809

The 6809 µP is unique because it represents an upward growth device from the ubiquitous 6800 (µP. This upward growth is in the form of software compatibility, at the source code level, and apparent similar operation of the two devices.

As a result of these similarities, it is possible to design the 6809 µP into a variety of applications. Among these applications are process control, automobile system monitoring, television sets, intelligent terminals and other devices more far reaching than this book could even begin to mention. As an example of the use of the 6809 µP was recently incorporated into what will more than likely become the small computer system of the decade, the *Radio Shack TRS-S0 Videotex* (Fig. 1-1). This unit not only uses the 6809 /xP, but the entire spectrum of Motorola support and peripheral chips. The point is that it shows the flexibility of the 6809 µP.

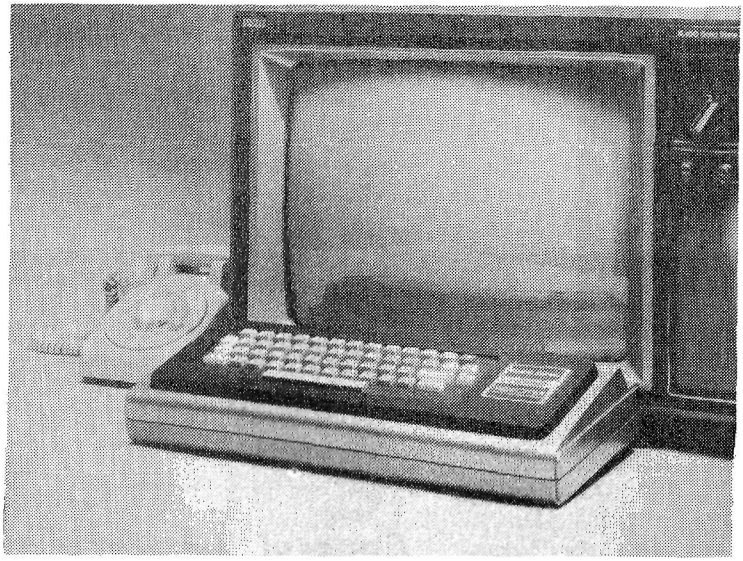


Fig. 1-1. Incorporating the 6809 µP in concert with a host of Motorola compatible peripheral device chips, the Radio Shack TRS-80 Videotex is designed to transform the home television and telephone into a high-powered communication system (courtesy of Radio Shack).

Flexibility and ease of system integration are important features of the 6809 µP, but are only representative of just a minor portion of the processor's capability. Throughout the rest of this book, you will be introduced to the specific features and functions of the processor and given sufficient information to make it work for you.

BASICS OF THE 6809 µP

The 6809 µP is an 8-bit NMOS device. With all the different mnemonics around—buzz words—it can be extremely difficult to figure out what someone is talking about in device types. For example, I say the 6809 µP is an NMOS device which it most surely is. *Metal Oxide Substrate* (MOS) technology is a method of creating integrated circuits (ICs); the N or P indicates the type of channel the device has built into it and does, in fact, refer to negative or positive. But for the purposes of this book, suffice it to say the device is NMOS. An NMOS device exhibits an electron mobility about 2.4 times that of PMOS. Consequently, the NMOS device outperforms similar PMOS devices in speed and power. This means more power to you, the designer, in a smaller and in most cases less expensive package.

The processor is designed, according to Motorola engineers, for real-time and character manipulation programming. This design philosophy implies that the device is ideal for such applications as real time, or event, data collection. Applications such as this require that the processor work in concert with data acquisition probes like thermocouples, strain gauges or flow sensors to name a few. The 6809 µP offers the important characteristics of being able to respond quickly enough to handle the influx of data from this type of device.

The character oriented capability of the 6809 µP makes it an excellent choice for word processing applications. In this type of application, ASCII data is manipulated in several ways to create useful output data. Word processing implies that the µP must work with a variety of peripheral devices, a functional plus of all 68XX type parts.

HIGH LEVEL LANGUAGE PROCESSOR

Another feature of the 6809, µP is that it is a byte-oriented device rather than operating on each bit of the 16 available on the address bus. This single characteristic enhances the processor's ability to function, with efficiency, as a high-level language computer.

The reason this processor, or any processor for that matter, works better as a high level language processor if it is *byte-oriented* is that a byte is 8 bits long. It is directly equatable to a character of some type, for example the letter A. High level language like COBOL and FORTRAN work by comparing block structures made up of characters to determine a task. Bit-oriented devices must first build a byte from individual bits, store it in a register and then permit the language instruction to perform some work or comparison. The byte-oriented device makes the assumption that registers are filled with bytes, thus speeding up execution times. This fact does not preclude bit operations that must take place in math functions.

Although the 6809 µP is not a pin-for-pin replacement of the 6800, it is not all that different in the sense of compatible functions and software. For example, the 6800 µP exhibited one stack pointer; the 6809 has two. The 6809 µP has two index registers as opposed to the single index register of the 6800. In relationship to the improved indexing capability of the device, both the stack pointers and program counter can be indexed. This feature makes it much easier for the programmer to manipulate data held in the processor's registers.

CHANGED CONFIGURATION

The 6809 µP, as mentioned, is not a pin-for-pin replacement for the 6800 µP; nor was there any thought for it being so. The idea behind the 6809 µP was to make it an optimum device, which meant that pin outs and pin definitions would change. Figure 1-2 is a representation *of both the* pin-outs *of the* 6800 and the 6809 microprocessors. You will notice that for the 6809 µP, two versions exist. The versions demonstrate the difference in the clock and at the same time represent the functional features of the 6809. To summarize these functional and electrical features:

- The 6809 µP incorporates an 8-bit data and a 16-bit address bus.
- The device is compatible to the MC6800 bus structure as defined by Motorola. (For further information, consult the M6800 microprocesor applications manual).
- The 6809, µP, housed in a 40-pin package, requires only a single +5V supply.
- The 6809 µP, exhibits the same interfacing characteristics as the 6800. This means that it is compatible with TTL logic levels, and consequently makes total system integration fairly easy.
- Addition of extra features like the *Fast Interrupt Request* (FIRQ). The FIRQ permits the 6809 µP to drop everything and handle high speed interrupts, as would be necessary in data acquisition systems.

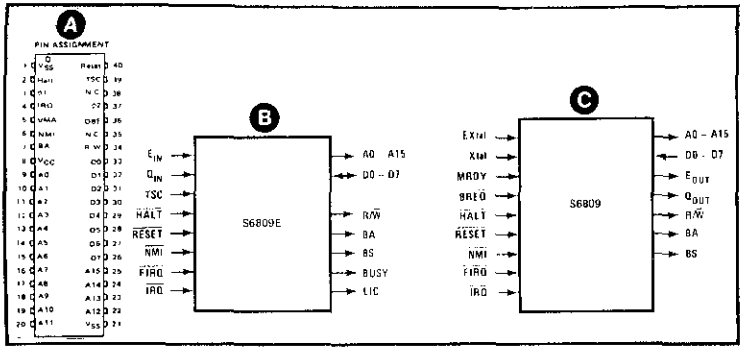


Fig. 1-2. Although compatibility exists between the 6809 and 6800 at the source code level, pin assignments differ since the 6809 offers more functions than its predecessor. (A) This represents the 6800 with its standard, pin assignments. (B) and (C) Block diagrams of the versions of the 6809 µP. Notice the pins named TSC, LIC and BUSY (courtesy of Motorola Semiconductor Products Inc. and American Microsystems, Inc.).

• Vectored interrupts allow the 6809 µP to locate an interrupt servicing routine within a minimum amount of time, and return back to the starting location without destroying the current data.

• The 6809 µP incorporates an onboard oscillator which is four times the input frequency of the crystal. The 6809E version features an external clock. See Fig. 1-3. This allows the 6809 µP to sync with an external clocking source such as that being generated by a clock source from another system.

• The 6809 MPU has two memory functions not found with the 6800: MRDY that extends data access times for use with slow memory, and DMA/BREQ that permits quick access to the bus for *Direct Memory Access* (DMA) and memory refresh.

The electrical differences in the two versions are shown in Figs. 1-3A and 1-3B. Basically, the two versions of the 6809 µP are the same, with the exception of the clocking mode. Table 1-1 defines the Read/Write for each version of the microprocessor. Table 1-2 lists the electrical characteristics of the processor.

THE RIGHT NOMENCLATURE

As you proceed through this book, you will notice reference to a part preceded by an S. This nomenclature defines the part as being from AMI. When the device being referred to is a Motorola part, the number is preceded by an M or MC. However, for the sake of clarity I have adopted the generic term—6809. There are some tables and figures in this book that do make reference to the specific manufacturers' devices.

VARIETY IN CLOCKS

The 6809 µP incorporates the choice of two clock functions. The basic 6809 processor exhibits an internal clock (oscillator). To make use of this clock, an external crystal is connected between EXTAL and XTAL pins 39 and 38. Netting or filter 0.01 disc ceramic capacitors are on either side to the system ground (Fig. 1-4). When the 6809 is in this configuration, a synchronization signal is available at the E/out terminal (pin 34). This available signal can be used as the system clock with all other devices in sync with it.

The output that is available on pin 34 is at the basic processor frequency and for most applications is connected to the *Enable* (02) *input* of 6800 peripheral devices, as shown in Fig. 1-5. This simplification of the clocking system, with 6800 family compatibility, eases system design and integration.

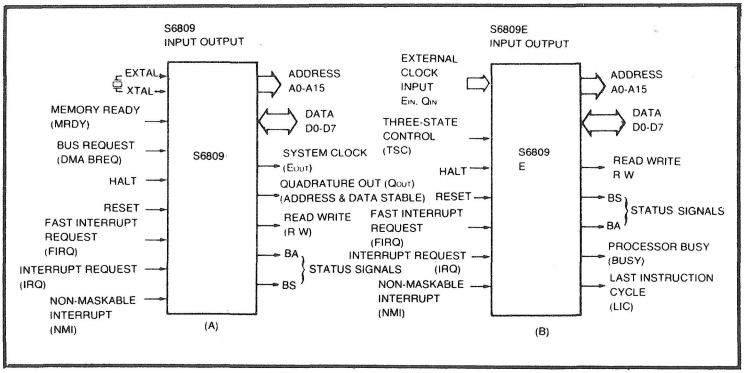


Fig. 1-3. Providing flexibility to the system designer, the 6809 *µP* is built with either an internal oscillator (A) or for use with an external clock (B) (courtesy of American Microsystems, Inc.).

Besides the timing signals discussed, another signal called the *Quadrature output* (Q/out) is available. The purpose of this signal is to signify that addresses and data are stable. This stability tells the system that operations have settled down and something else can take place.

Table 1-1. Read/Write Timing (courtesy of American Microsystems, Inc. and Motorola Semiconductor Products Inc.).

Read/Write Timing

		S6809			S68A09			S68B09				
Symbol	**Parameter**	**Min.**	**Typ.**	**Max.**	**Min.**	**Typ.**	**Max.**	**Min.**	**Typ.**	**Max.**	**Unit**	**Condition**
t_{CYC}	Cycle Time	1000			667			500			ns	
t_{UT}	Total Up Time	975			640			480			ns	$t_{acc} = t_{ut} - t_{AD} - t_{DSR}$
t_{ACC}	Peripheral Read Access Time	695			440			320			ns	$t_{ut} = t_{CYC} - t_{EF}$
t_{DSR}	Data Setup Time (Read)	80			60			40			ns	
t_{DHR}	Input Data Hold Time	10			10			10			ns	
t_{DHW}	Output Data Hold Time	30			30			30			ns	
t_{AH}	Address Hold Time (Address, R/$\overline{W}$)	30			30			30			ns	
t_{AD}	Address Delay			200			140			110	ns	
t_{DDW}	Data Delay Time (Write)			225			180			145	ns	
t_{AVS}	E_{low} to Q_{high} Time			250			165			125	ns	
t_{AQ}	Address Valid to Q_{high}	25			25			15			ns	
t_{PWEL}	Processor Clock Low	450			295			210			ns	
t_{PWEH}	Processor Clock High	450			280			220			ns	
t_{PCSR}	MRDY Set Up Time	60			60			60			ns	
t_{PCS}	Interrupts Set Up Time	200			140			110			ns	
t_{PCSH}	$\overline{HALT}$ Set Up Time	200			140			110			ns	
t_{PCSR}	$\overline{RESET}$ Set Up Time	200			140			110			ns	
t_{PCSD}	$\overline{DMA/BREQ}$ Set Up Time	125			125			125			ns	
t_{rc}	Crystal Osc Start Time	100			100			100			ms	
t_{ER}, t_{EF}	E Rise and Fall Time	5		25	5		25	5		20	ns	
t_{PCR}, t_{PLF}	Processor Control Rise/Fall			100			100			100	ns	
t_{QR}, t_{QF}	Q Rise and Fall Time	5		25	5		25	5		20	ns	
t_{PWQH}	Q Clock High	450			280			220			ns	

Table 1-2. Electrical Characteristics (courtesy American Microsystems, Inc. and Motorola Semiconductor Products Inc.).

Electrical Characteristics (V_{CC} = 5.0V ± 5%; V_{SS} = 0, T_A = 0°C to + 70°C unless otherwise noted)

Symbol	Parameter		Min.	Typ.	Max.	Unit	Condition
V_{IH}	Input High Voltage	Logic, EXtal	V_{SS} + 2.0		V_{DD}	Vdc	
		RESET	V_{SS} + 4.0		V_{DD}		
V_{IL}	Input Low Voltage	Logic EXtal, RESET	V_{SS} – 0.3		V_{SS} + 0.8	Vdc	
I_{in}	Input Leakage Current	Logic		1.0	2.5	μAdc	V_{in} = 0 to 5.25V, V_{CC} = max
V_{OH}	Output High Voltage	D0-D7	V_{SS} + 2.4			Vdc	I_{Load} = – 205μAdc, V_{CC} = min
		A0-A15, R/W, Q, E	V_{SS} + 2.4				I_{Load} = – 145μAdc, V_{CC} = min
		BA, BS	V_{SS} + 2.4				I_{Load} = – 100μAdc, V_{CC} = min
V_{OL}	Output Low Voltage				V_{SS} + 0.5	Vdc	I_{Load} = 2.0mAdc, V_{CC} = min
P_D	Power Dissipation				1.0	W	
C_{in}	Capacitance #	D_0-D_7		10	15		
		Logic Inputs, EXtal		7	10	pF	V_{in} = 0, T_A = 25°C, f = 1.0MHz
C_{out}		A_0-A_{15}, R/W			12		
f	Frequency of Operation	S6809			4		
f_{XTAL}		S68A09			6	MHz	
f_{XTAL}	(Crystal or External Input)	S68B09			8		
I_{TSI}	Three-State (Off State) Input Current	D_0-D_7		2.0	10	μAdc	V_{in} = 0.4 to 2.4V, V_{CC} = max
		A_0-A_{15}, R/W			100		

The external clock version, indicated by an E, requires that an external clock source be implemented. This external clock must generate an output at the MPU frequency. The timing signal E is similar to the 6800 bus timing signal 02; Q is a quadrature clock, signal which leads E. This quadrature signal has no parallel on the 6800. The importance of these signals are that addresses from the MPU will be valid with the leading edge of Q (Fig. 1-6). Data is latched on the falling edge of E.

You will notice from Fig. 1-3 that the external clock version of the 6809, the BREQ input, is replaced by a *tri-state* (TSC) *control*. This control serves to place the address and R/W in the high

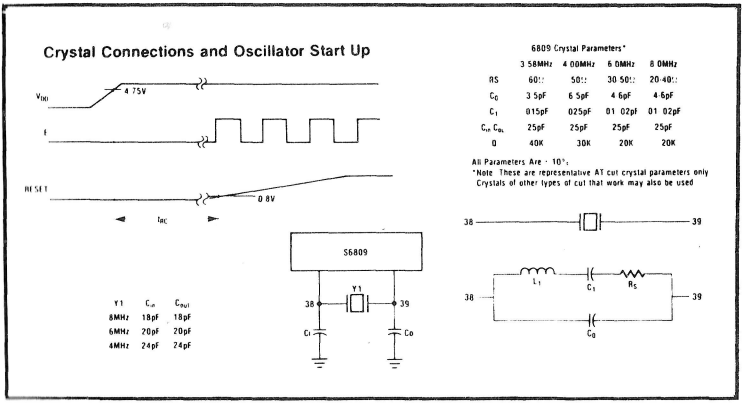


Fig. 1-4. The clock on the 6809 is invoked by tying pins 38 and 39 together via a crystal and filter capacitors (courtesy American Microsystems, inc. and Motorola Semiconductor Products Inc.).

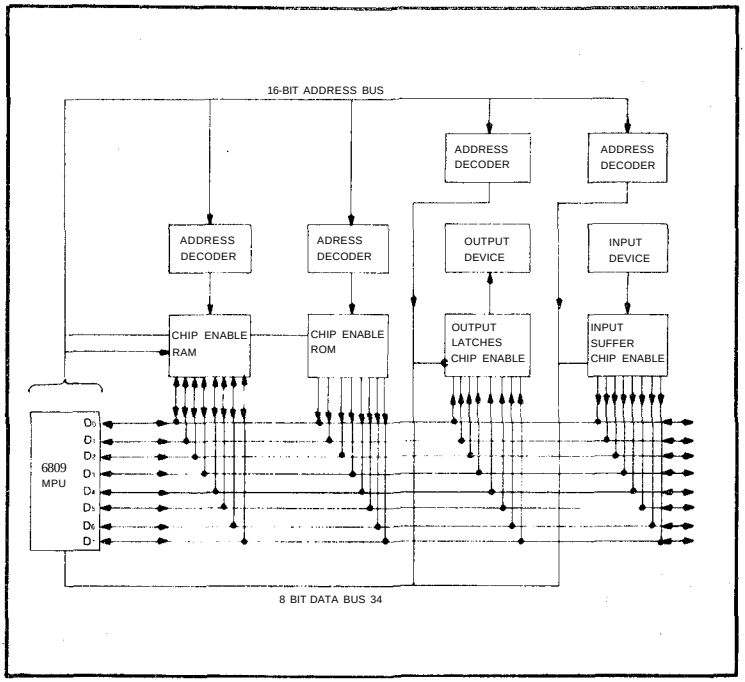


Fig. 1-5. Interfacing the 6809 µP to other devices is easy by taking the output from pin 34 and tying it to the chip enable pin of the peripheral chip. In this figure, the processor is tied to RAM, ROM and output devices. Pay particular attention to the direction of the data on the data bus. The output from 34 is tied to the chip enable of the output latch and input buffer.

cycle of any instruction. This signifies that the next instruction cycle is the opcode fetch and acts like a pipeline fetch, thus improving processing throughout. The processor BUSY signal facilitates multiprocessor applications by allowing the designer to

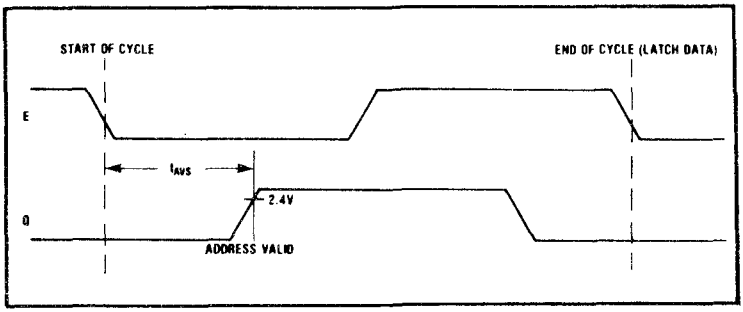


Fig. 1-6. E/Q relationship (courtesy American Microsystems, Inc. and Motorola Semiconductor Products Inc.).

impedance state for DMA or memory refresh. The E and Q pins are replaced by two status outputs: *Last Instruction Cycle (LIC)* and *processor busy signal* (BUSY). The LIC is activated during the last insure that flags being modified by one processor are not accessed by another simultaneously.

The 6809 µP, in normal operation, fetches an instruction from memory and then executes the requested function. This operational function begins when the processor is started—RESET—and repeated until forced to cease. This stopping of the operation can be from a multitude of sources including interrupts, hard and soft, or via a special instruction that permits the processor to HALT but also save the contents of the registers—that is, waiting to proceed without impacting the computing ability of the processor.

6809 MPU SIGNAL DESCRIPTION

This section describes the functional purposes of the pins available on the 6809 µP. The reason, is to create a solid foundation for the chapters on addressing and the instruction set.

The information contained in this section is, in most cases, directly from AMI literature. I have attempted, where necessary, to further clarify or amplify upon those items that seem vague.

Power (V_{ss} V_{cc}) Pins 1 And 7. Two pins are used to supply power to the part: V_{ss} is ground, or 0 volts, while V_{cc} is +5V with a 5% tolerance. This holds true whether the device is of the internal or external clock variety (Fig. 1-7).

Address Bus (A_0 - A_{15}) Pins 8-23. Sixteen pins are used to output address information from the MPU onto the address bus. When the processor does not require the bus, for a data transfer, it will output address $FFFF_{16}$, R/$\overline{W}$ = 1 and BS = 0 (Table 1-3). Addresses are valid on the rising edge of Q. All address bus drivers are made high-impedance when output *Bus Available* (BA) is high. Each pin will drive one Schottky TTL load and typically 90pF (Fig. 1-7).

Data Bus (D_0-D_7) (Pins 24-31). The eight pins, designated for data, provide communication with the system bidirectional data bus. Each pin will drive one Schottky TTL load and typically 130pF (Fig. 1-7).

Read/Write (R/W) Pin 32. This signal indicates the direction of the data transfer on the data bus. A low indicates that the

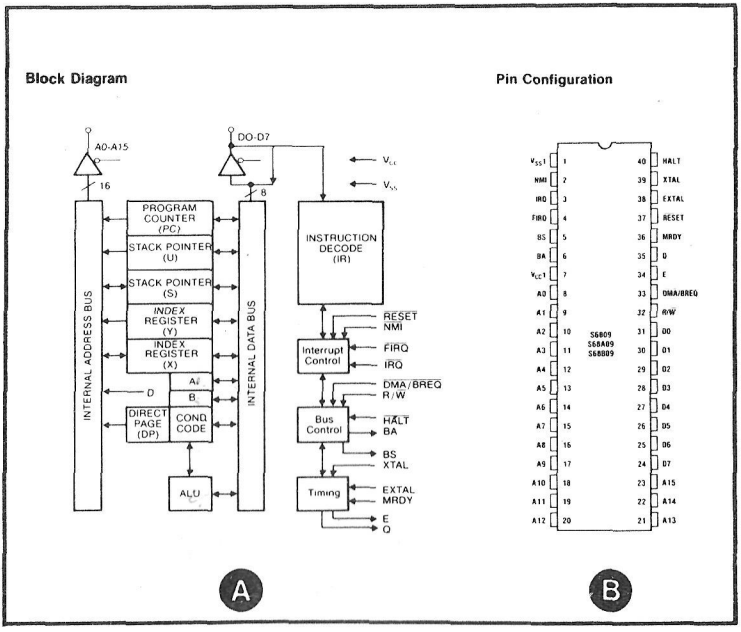


Fig. 1-7. (A) Block diagram. (B) Pin configuration (courtesy of American Microsystems, Inc. and Motorola Semiconductor Products Inc.).

MPU is writing data onto the data bus. R/$\overline{W}$ is made high impedance when BA is high. R/$\overline{W}$ is valid on the rising edge of Q.

$\overline{RESET}$. Pin 37. A low level on this Schmitt trigger* input for greater than one bus cycle will $\overline{RESET}$ the MPU. The RESET vectors are fetched from locations $FFFE_{16}$ and $FFFF_{16}$ when interrupt acknowledge is true BABS=1). During initial power on, the Reset line should be held low until the clock oscillator is fully operational.

Because the 6809 µP Reset pin has a Schmitt-trigger input with a threshold voltage higher than that of standard peripherals, a

Table 1-3. MPU State (courtesy of American Microsystems, Inc.).

MPU State		
BA	**BS**	
0	**0**	Normal (running)
0	**1**	Interrupt Acknowledge
1	**0**	SYNC Acknowledge
1	**1**	HALT or Bus grant

simple R/C network may be used to reset the entire system. This higher threshold voltage insures that all peripherals are out of the reset state before the processor.

PULLING THE SCHMITT-TRIGGER

A *Schmitt-trigger* is a special type of flip-flop circuit that permits feedback and is sometimes referred to as a regenerative switching circuit, having two stable output states. The Schmitt-trigger is frequently used in timing circuits to mark the instant when an input voltage reaches the trigger level, converting a sinusoidal input voltage into a pulse train at the output. Since it is not within scope of this book to provide complete explanations of flip-flops, I recommend *Electronic Circuits Digital and Analog,* by Charles A. Holt, John Wiley and Sons, New York.

HALT—Pin 40. A low level on this input pin will cause the MPU to stop running at the end of the present instruction and remain halted indefinitely without loss of data. When halted, the BA output is driven high indicating the buses are high-impedance. BS is also high which indicates the processor is in the Halt or Bus Grant State. While halted, the MPU will not respond to external real-time requests (FIRQ, IRQ), although DMA/BREQ will always be accepted and NMI or RESET will be latched for later response. During the HALT state Q and E continue to run normally. If the MPU is not running (RESET, DMA/BREQ), a halted state (BA and BS = 1) can be achieved by pulling HALT low while RESET is still low. If DMA/BREQ and HALT are both pulled low, the processor will reach the last cycle of the instruction (by reverse cycle stealing) where the machine will then become halted (Fig. 1-8).

Bus Available. Bus Status (BA, BS) Pins 5 and 6. The Bus Available output is an indication of an internal control signal which makes the MOS buses of the MPU high-impedance. This signal does not imply that the bus will be available for more than one cycle. When BA goes low, an additional dead cycle will elapse before the MPU acquires the bus. The bus status output signal, when decoded with BA, represents the MPU state (valid with leading edge of Q).

TRACING THE INTERRUPT

When an interrupt occurs, the processor must respond in some manner. The 6809 µP responds by going to a location in memory and executing a specific routine. In all cases, the proces-

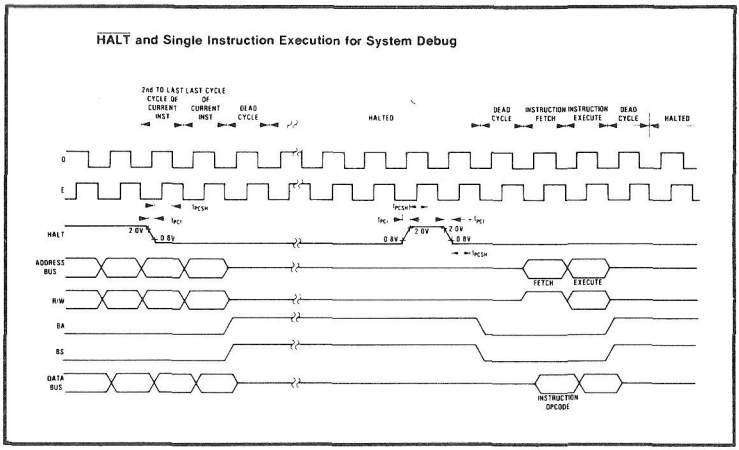


Fig. 1-8. $\overline{\text{HALT}}$ and single instruction execution for system debug (courtesy of American Microsystems, Inc. and Motorola Semiconductor Products Inc.).

sor generates a signal called an interrupt Acknowledge. It is indicated during both cycles of a hardware-vector-fetch ($\overline{\text{RESET}}$, $\overline{\text{NMI}}$, $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$, SWI, SWI2, SWI3). This signal, plus decoding of the lower 4 address lines, can provide the user with an indication of which interrupt level is being serviced and allow vectoring by device, as shown in Table 1-4.

Table 1-4. Memory Map for Interrupt Vectors (courtesy of American Microsystems, Inc.).

Memory Map for Interrupt Vectors

Memory Map for Vector Location		Interrupt Vector Description
MS	LS	
FFFE	FFFF	$\overline{\text{RESET}}$
FFFC	FFFD	$\overline{\text{NMI}}$
FFFA	FFFB	SWI
FFF8	FFF9	$\overline{\text{IRQ}}$
FFF6	FFF7	$\overline{\text{FIRQ}}$
FFF4	FFF5	SWI2
FFF2	FFF3	SWI3
FFFO	FFF1	Reserved

Other signals that play an important role either during an interrupt condition or HALT condition are *Sync acknowledge* and *Halt/Bus grant* (Fig. 1-9). The Sync acknowledge is indicated while the MPU is waiting for external synchronization on an interrupt line. Halt/Bus Grant is true when the 6809 µP is in a $\overline{\text{HALT}}$ or Bus Grant condition, as explained previously under HALT.

Nonmaskable Interrupt ($\overline{\text{NMI}}$) Pin 2. The Nonmaskable Interrupt pin is very similar to the $\overline{\text{IRQ}}$ pin 3, except that the interrupt input is nonmaskable from the MPU. This means that a program cannot inhibit the interrupt, and it has a higher priority than $\overline{\text{IRQ}}$ or $\overline{\text{FIRQ}}$, or for that matter of software interrupts (Fig. 1-10).

The $\overline{\text{NMI}}$ is invoked when a negative wedge is input on the pin. When recognized, the entire machine state is saved on the hardware stack. However, once the machine is reset, the $\overline{\text{NMI}}$ is not recognized until the first program load of the *Hardware Stack Pointer (S)*. The pulse width of $\overline{\text{NMI}}$ low must be at least one E cycle. If the $\overline{\text{NMI}}$ input does not meet the minimum set up with respect Q, the interrupt will not be recognized until the next cycle.

Fast-Interrupt Request ($\overline{\text{FIRQ}}$) Pin 4. It is unique to the 6809 µP. When a low level signal is detected at this pin, a fast interrupt sequence, provided its mask bit (F) in the CC is clear, will be initiated. The $\overline{\text{FIRQ}}$ has priority over the standard Interrupt Request $\overline{\text{IRQ}}$ and is fast in the sense that it stacks only the contents of the condition code register and the program counter. When

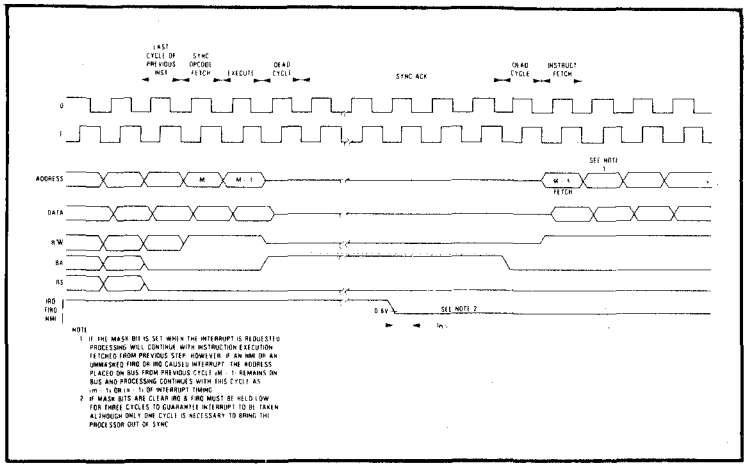


Fig. 1-9. SYNC timing (courtesy of American Microsystems, Inc., and Motorola Semiconductor Products Inc.).

used, the interrupt service routine should clear the source of the interrupt before doing a Return from Interrupt (RTI). The timing for this interrupt is shown in Fig. 1-10A.

Interrupt Request ($\overline{IRQ}$) Pin 3. When this line is forced low, from some external device, the MPU will complete the instruction it is executing and go into the interrupt sequence. This is no different than for the 6800 µP. The $\overline{IRQ}$ has a lower priority than $\overline{FIRQ}$, but the servicing routine should clear the source of the interrupt before returning to the calling routine (Fig. 1-10B).

When $\overline{IRQ}$ is invoked, the contents of the index register, the program counter, accumulators and condition code register will be stored on the stack. The I bit in the condition code register will be set to a 1 so that no further interrupts may occur, or at least until this one is serviced. As shown in Table 1-4, the MPU will nowload the contents of $FFF8_{16}$ and $FFF9_{16}$ into the program counter and vector the program to execute the interrupt routine pointed to by these locations. After an RTI is encountered, the MPU will return to its initial state.

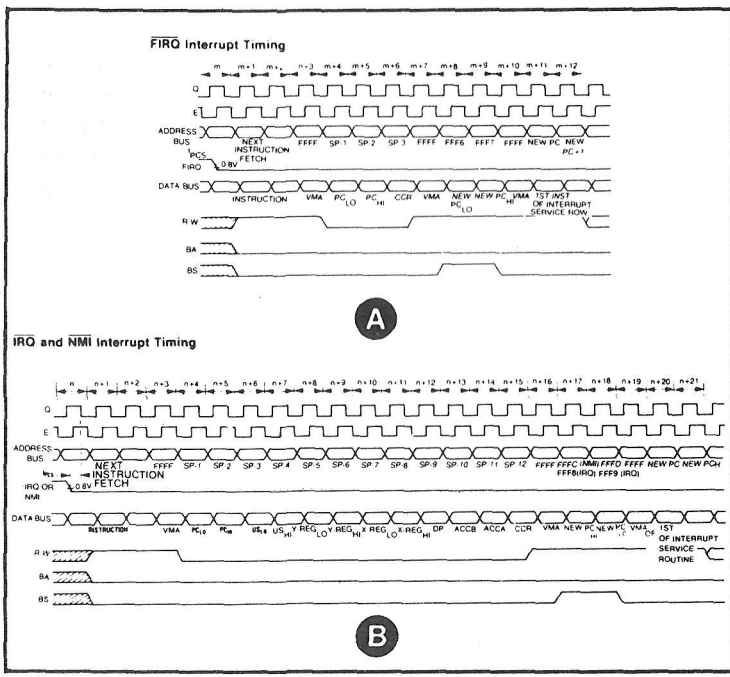


Fig. 1-10. (A) $\overline{FIRQ}$ interrupt timing. (B) $\overline{IRQ}$ and $\overline{NMI}$ interrupt timing (courtesy of American Microsystems, Inc. and Motorola Semiconductor Products Inc.).

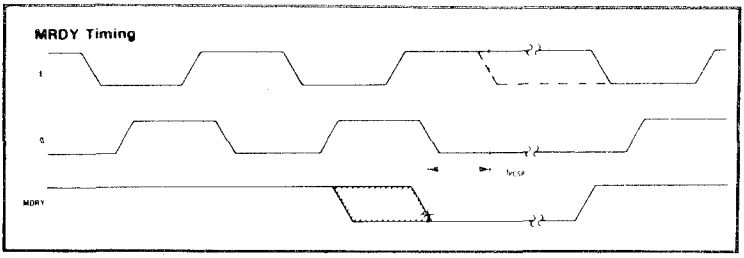


Fig. 1-11. MRDY timing (courtesy of American Microsystems, inc. and Motorola Semiconductor Products Inc.).

EXTAL, XTAL Pins 38, 39. These input pins are used to connect the on-chip oscillator to an external parallel-resonant crystal (Fig. 1-4). The pin labeled EXTAL may be used as TTL level input for external timing by grounding XTAL. The crystal or external frequency is 4 times the bus frequency which is shown in Fig. 1-4.

E, Q Pins 34, 35. Here you can see some specific similarities between the 6800 µP and the 6809. E is similar to the 6800 bus timing signal Q2; Q is a quadrature clock signal which leads E. Q has no parallel on the 6800. Addresses from the MPU will be valid with the leading edge of Q. Data is latched on the falling edge of E.

MRDY Pin 36. This input control signal allows stretching of E to extend data-access time. When MRDY is high, E will be in normal operation. When MRDY is low, E may be stretched integral multiples of quarter (¼) bus cycles, thus allowing interface to slow memories as shown in Figs. 1-11. A maximum stretch is 10 µsec. During non-valid memory accesses ($\overline{\text{VMA}}$ cycles), MRDY has no effect on stretching E. This inhibits slowing the processor speed during *don't care* bus accesses.

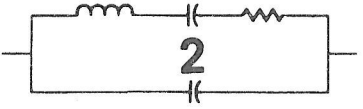
$\overline{\text{DMA/BREQ}}$ Pin 33. The DMA/BREQ input provides a method of suspending execution and acquiring the MPU bus for another use. Typical uses include DMA and dynamic memory refresh.

Transition of $\overline{\text{DMA/BREQ}}$ should occur during Q. A low level on this pin will stop instruction execution at the end of the current cycle. The MPU will acknowledge $\overline{\text{DMA/BREQ}}$ by setting BA and BS to a one. The requesting device will now have up to 15 bus cycles before the MPU retrieves the bus cycle with a leading and trailing dead cycle.

ESTABLISHING A SYSTEM

Now that you have a resonable understanding of the hardware side of the 6809 µP, you can build a working unit. All that is required is the 6809, some memory, power and form of display.

A typical system—the Motorola 6809D4 unit—is discussed in Appendix A. But the most important element behind the processor is not the hardware but rather how to program it. In the next chapter, the basic software architecture of the device will be introduced, followed by the various addressing techniques in Chapter 3 and finally the instruction set in Chapter 4.



6809 µP Software Architecture

Software development entails the understanding of several disciplines—specifically hardware logic as it relates to the hardware, mathematics and general logic flow. Interestingly enough, the software engineer doesn't really need to have an in-depth understanding of the electrical characteristics of the processor he is programming, unless of course original system software development is the goal.

However, whether the goal is system software design or developing specific utility ware, the software architecture of the device must be understood.

THE SOFTWARE TALE

The 6809 µP is, as stated in Chapter 1, an upward growth device from the 6800 µ*P*. Specifically, the 6809 adds three registers to the set available in the 6800. These include a direct page register, the user stack pointer and a second index register. These additional registers make the device extremely flexible, but the 6809 offers even other software features:

—Two 8-bit accumulators
—Two 16-bit index registers.
—Two 16-bit stack pointers with index capability.
—The previously mentioned programmable direct page register.
—59 instruction mnemonics (see Chapter 4).

—268 opcodes.
— 1464 instructions with different addressing modes.
—8x8 unsigned multiply.
— 16-bit arithmetic: load, store, add, subtract and compare.
—Powerful Push/Pull instructions.
—Powerful register transfers and exchanges.
—Powerful address-manipulation instructions.
—Extended-range long branches.

As you can see, the device is extremely flexible and offers the software designer a great deal of power in a microprocessor. Figure 2-1 is the basic programming model for the 6809 µP. You will notice that the X and Y index registers are 16-bits wide, and the U and S stack pointers are also 16-bits. The interesting register is made up of two 8-bit registers, A and B, which together make up D. It is within these three registers—*accumulators*—that most of the processor's work will be done. The direct page and condition code registers are 8-bits wide and provide programming enchancement that will be explained later.

The general architecture of the device supports software techniques such as position-independent code, structured high level-subroutined code, multi-task and multi-processor operations, development and operation of stack oriented compiler instructions, and the important facilities of re-entrancy and recursion, both important facets of software for high-level language use

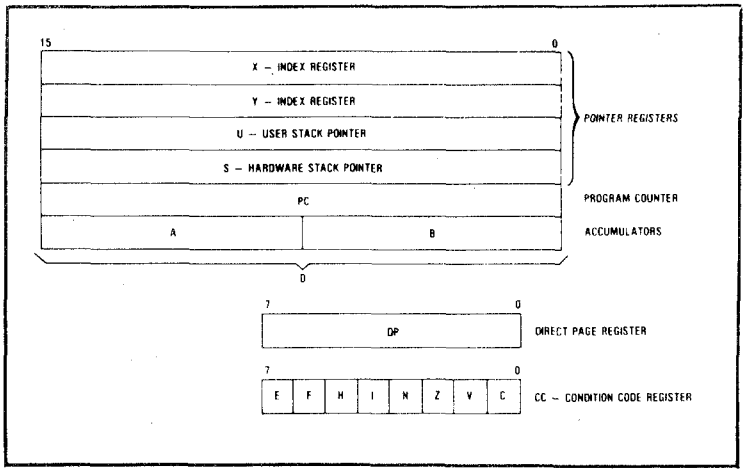


Fig. 2-1. Programming model of the microprocessing unit (courtesy of American Microsystems, Inc.).

or real-time data acquisition. Now that you know all of the good things that the software architecture behind the 6809 is supposed to provide, you are probably anxious for a more in-depth explanation of the programming model.

REGISTERS, POINTERS AND THINGS

Taking a look at Fig. 2-1, you can see that within the structure there are the X and Y 16-bit index registers. These are also referred to as the *pointer registers.*

The index registers are used in the indexed mode of addressing. The 16-bit address in either the X and Y register is used to point to data directly, or it may be modified by an optional constant or register offset. The X and Y registers are equivalent in usage and consequently support the same instructions. These registers may be used to implement software stacks, queues and buffers.

Stack pointers U and S, shown in Fig. 2-1, can also be used as index registers, but they serve very specific purposes in processing. The *Hardware Stack Pointer* (S) is used by the processor during subroutine calls and interrupts.The difference between this stack pointer and that on the 6800 µP is that it points to the top of the stack rather than the next free location (Table 2-1).

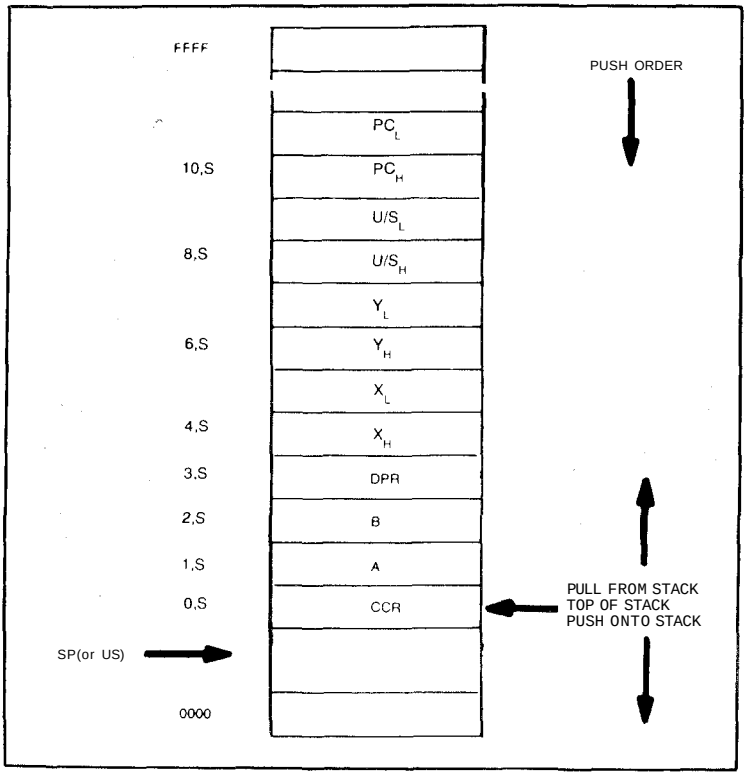
The *User Stack Pointer* (U) is for use by you, the programmer. This stack pointer permits you to pass arguments to and from subroutines with ease. This facility coupled with the hardware stack pointer makes the 6809 µP and ideal stack processor and enhances its functioning as a higher level language processor. Because the architecture of the U and S pointers are, as previously indicated, the same as the X and Y registers, they also support the same instructions plus the PUSH and PULL stack controls.

The next register is the *Program Counter* (PC). This register is 16-bits and is used by the processor to point to the address of the next instruction to be executed by the processor. Relative addressing is provided allowing the PC to be used like an index register in some situations. Limited index-mode addressing is available, but functions such as auto increment and decrement are not.

In operation, each instruction used by the processor assumes that the PC points one location past the last byte Of the op code—as it would after decoding the instruction. Consequently, as additional bytes are used by the instruction, the PC always points to the next unused byte.

The next registers, A, B and D accumulators, are made up of two 8-bit registers as shown by Fig. 2-1. The A and B registers are

Table 2-1. 6809 µP Push/Pull and Interrupt Stacking Order.



general purpose accumulators which are used for arithmetic calculations and manipulation of byte size data. What makes this pair unique is that certain instructions concatenate A and B to form the 16-bit register D, with the contents of A being the most significant byte.

The *Direct Page* register (DP) defines the most significant (MS) byte to be used in the direct mode of addressing. The DP is concatenated with the byte following the direct mode op code to form a 16-bit effective address. The contents of this register appear at the higher address output (A8-A15) during direct addressing instruction execution. This permits the use of the direct mode anywhere in memory. To maintain 6800 compatibility, all bits are initialized to $00 on Reset of the processor.

CONDITION CODES ARE SPECIAL

The final register in the programming model of Fig. 2-1 is the condition code register (CC). Figure 2-2 is the format for this 8-bit

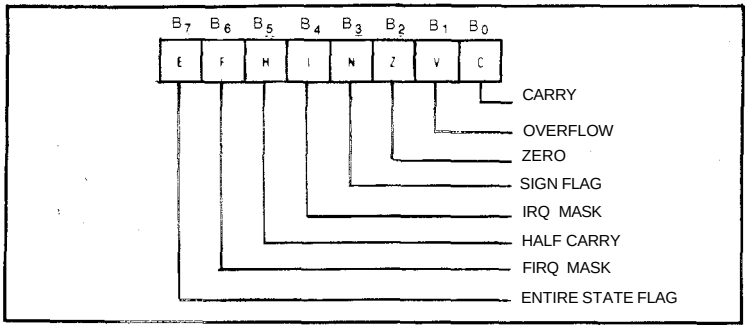


Fig. *2-2.* Condition code register format (courtesyAmerican Microsystems, Inc.).

register. Notice that each bit is defined and based on the condition—toggle 0 or 1—which defines the operation state of the processor and is always nice to know.

Each bit within the register performs a specific task. For example, bits 0-3 and 5 are set as the result of instructions that manipulate data in some way. The actual definitions of each bit follows.

Bit 0 (C)

Bit 0 is the *Carry Flag,* and is usually the carry from the binary *Arithmetic Logic Unit* (ALU). Specifically, the C flag is generated by the binary carry from the *Most Significant Bit* (MSB) of the operations (ADC, ADD). Furthermore, C is used to represent a "borrow" from subtract-like instructions (CMP, NEG, SUB, SBC). **Only arithmetic operations affect C.**

Bit 1 (V)

Bit 1 is the overflow flag and is set to a one by an operation which causes a signed *two's complement* arithmetic overflow. This overflow is detected in an operation in which the carry from the MSB in the ALU does not match the carry from the MSB-1. Loads, stores and logical operations set V.

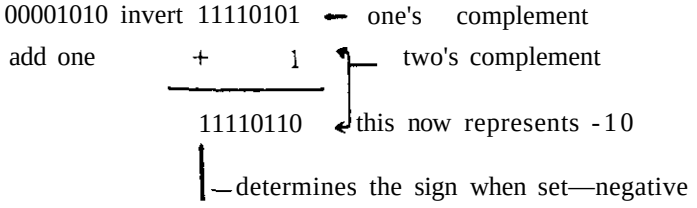
Two's Complement

If you have advanced to this point in the book and aren't sure what two's complement is, you may have a problem. I would suggest that you obtain a copy of "Basic Microprocessor and the 6800," by Ron Bishop, Hayden Book Co., 1979. You might also consider the Heath course on microprocessors. Both are excellent sources for explaining this concept, which is important if you wish

to understand what you are doing. Should you know what two's complement is all about but can't quite get a picture in your mind, this note will serve to refresh your memory—no pun intended.

The two's complement is the method used to represent signed numbers in microprocessors. Positive numbers, in this system, use the same bit pattern for all values up to decimal +127. Negative numbers are represented as the two's complement of positive numbers.

To find the two's complement of a number, you first take the one's complement and then add one. The one's complement is formed by changing all the 0s to 1s and all the 1s to 0s. Invert all the bits. For example, the decimal number 10 is 00001010 in binary. If the number is positive (°10), you follow this procedure.



Bit 2 (Z)

Bit 2 is the zero flag and is set to a one if the result of the previous operation was identically zero. Loads, stores, logical and arithmetic operations set Z.

Bit 3 (N)

Bit 3 is the negative flag, which obtains exactly the MSB value of the result of the preceding operation. Thus, a negative two's complement result will leave N set to a one. Loads, stores, logical and arithmetic operations all set N. If a two's complement overflow occurs, the sign of the result (and the N-flag) will be incorrect. Therefore, two's, compliment branches use the expression (N + V) to obtain an always valid sign result.

Bit 4 (I)

This is the *Interrupt Request* (IRQ) mask bit. The processor will noi recognize interrupts from the IRQ line if this bit is set to a one. NMI, FIRQ, IRQ, RESET and SWI all set I to a one. However, SWI2 and SWI3 do not affect I.

Bit 5 (H)

This bit is used to indicate a carry from bit 3 in the ALU as a result of an 8-bit addition only (ADC or ADD). This bit is used by the DAA instruction to perform a BCD decimal add adjust opera-

tion. The state of this flag is undefined in all subtract-like instructions.

Bit 6 (F)

This bit is associated with the Fast Interrupt Request (FIRQ). If this bit is set, the processor will not recognize interrupts from the FIRQ line. NMI, FIRQ, SWI and $\overline{\text{RESET}}$ all set F to a one. IRQ, SWI2 and SWI3 do not affect F.

Bit 7 (E)

This bit (7) is reserved for indicating the state of the ENTIRE registers. It shows when the processor is stacked or the subset state (PC or CC) is being stacked. E is used by the Return from Interrupt (RTI) instruction to determine the extent of the unstacking. This function allows some interrupt handling routines which work with both fast and slow interrupts. FIRQ will clear E while IRQ, NMI, SWI, SWI2 and SWI3 will set E before stacking. The E bit associated with the saved registers is in the E flag position in the CC of the stacked state.

Interrupts and the Condition Codes

When the 6809 accepts an $\overline{\text{IRQ}}$ interrupt, it will set the E flag bit 7 and save the entire machine state. Furthermore, the I mask bit 4 will be set to blank out the present and further IRQ interrupts. Once the interrupt is cleared, you can reset the I mask bit to permit multiple-level IRQ interrupts. When the IRQ occurs, the F mask bit 6 is not affected which means that an FIRQ interrupt can supersede the current IRQ interrupt. The machine state is recovered by the RTI instruction.

When an FIRQ interrupt is accepted, the E flag is cleared and the submachine state (return address and CC) is saved. The I and F bits are set to mask out further interrupts. Again, I and F can be reset to permit multiple interrupts.

6800/6809 SOFTWARE INCOMPATIBILITIES

The 6809 as designed is reasonably compatible with the 6800, but with the added features some inconsistencies must exist. Specifically, they are:

- The stacking order on the 6809 exchanges the order of ACCA and ACCB. This allows ACCA to stack as the MS byte of the pair and also invalidates previous 6800 code which displayed IX or PC from the stack.The 6809 stacks five more bytes for each NMI, IRQ or SWI when compared to the 6800.
- The 6809 stack pointer points directly to the last item placed on the stack rather than the location before it, as was done

on the 6800. Consequently, the stack pointer is initialized one location higher on the 6809 than the 6800. Comparison values must be one location higher.

- The 6809 uses two high-order condition code register bits and will not apear as 1s as on the 6800.
- The TST instruction does not affect the C flag in the 6809. Nor do the right shifts (ASR, LSR,ROR) affect V.
- The 6809 H flag is not defined as having any particular state after subtract-like operations (CMP, NEG, SBC, SUB). The 6800 clears this flag for these instructions.
- The CPX instruction for the 6809 functions correctly, setting all flags in the correct manner. The 6800 sets only the Z-flag.
- The 6809 instruction LEA may or may not affect the Z-flag depending upon which register is being loaded. However, LEAX and LEAY do affect the Z-flag, while LEAS and LEAU do not. See Chapter 4. As a result, the User stack (U) does not exactly emulate the index registers.

EQUIVALENCIES

Although Chapter 4 will deal with the actual instructions, the equivalent instructions between the 6800 and 6809 are important to know about for complete understanding of the architecture of the device. This is especially true if you are familiar with the 6800 instruction set.

Table 2-2 lists the 6800 instructions that are not included in the 6809. However, during assembly, the 6800 instructions are translated in to the functional equivalents as shown. I have made no attempt at this point to define each instruction, only to present the equivalent.

The interrupt structure on the 6809 µP has been extensively analyzed and improved compared to the 6800. With the 6800 µP it was useful to execute the sequence CLI, WAI. The 6809 µP logically-equivalent sequence-ANDCC #$EF, CWAI #$FF-would allow an IRQ interrupt to occur after the ANDCC instruction. If this is not desired, the 6809 instruction CWAI #$EF should be used to replace the logically-equivalent sequence.

PERFORMANCE SUMMARY

The following cycle-by-cycle performance chart (Fig. 2-3) illustrate the memory-access sequence corresponding to each possible instruction and addressing mode for the 6809 µP. Notice that

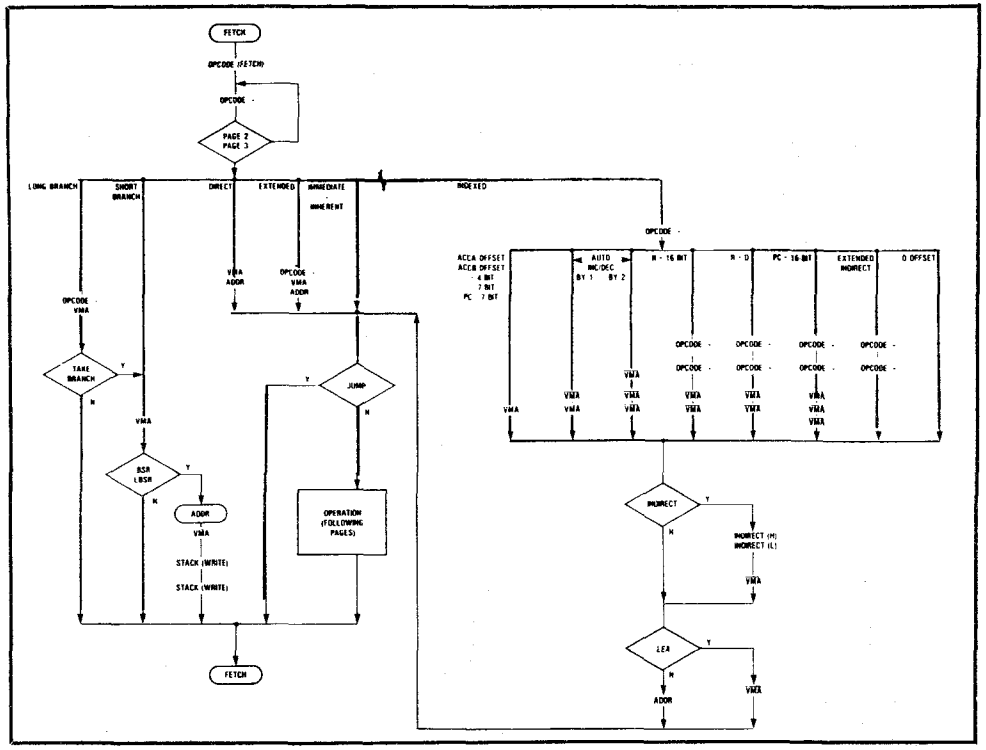


Fig. 2-3. Address bus cycle-by-cycle performance (courtesy of American Microsystems, Inc.).

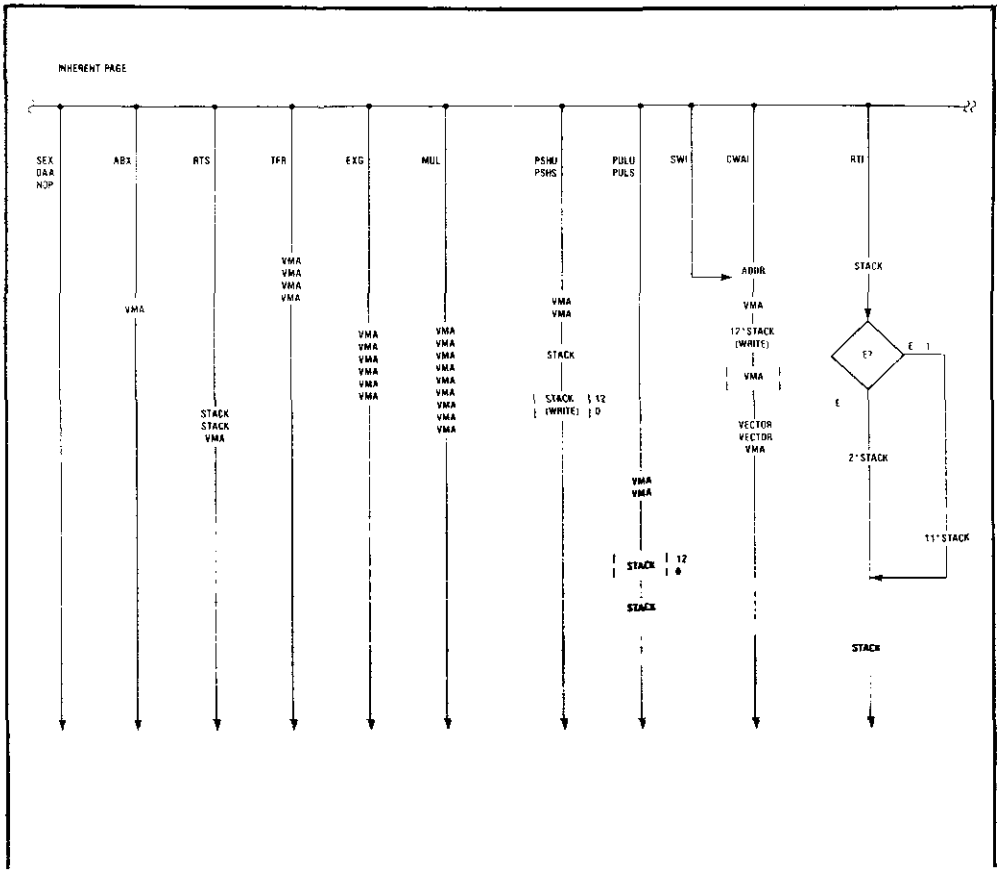


Fig. 2-3. Address bus cycle-by-cycle performance (courtesy of American Microsystems, Inc.) (continued from page 33).

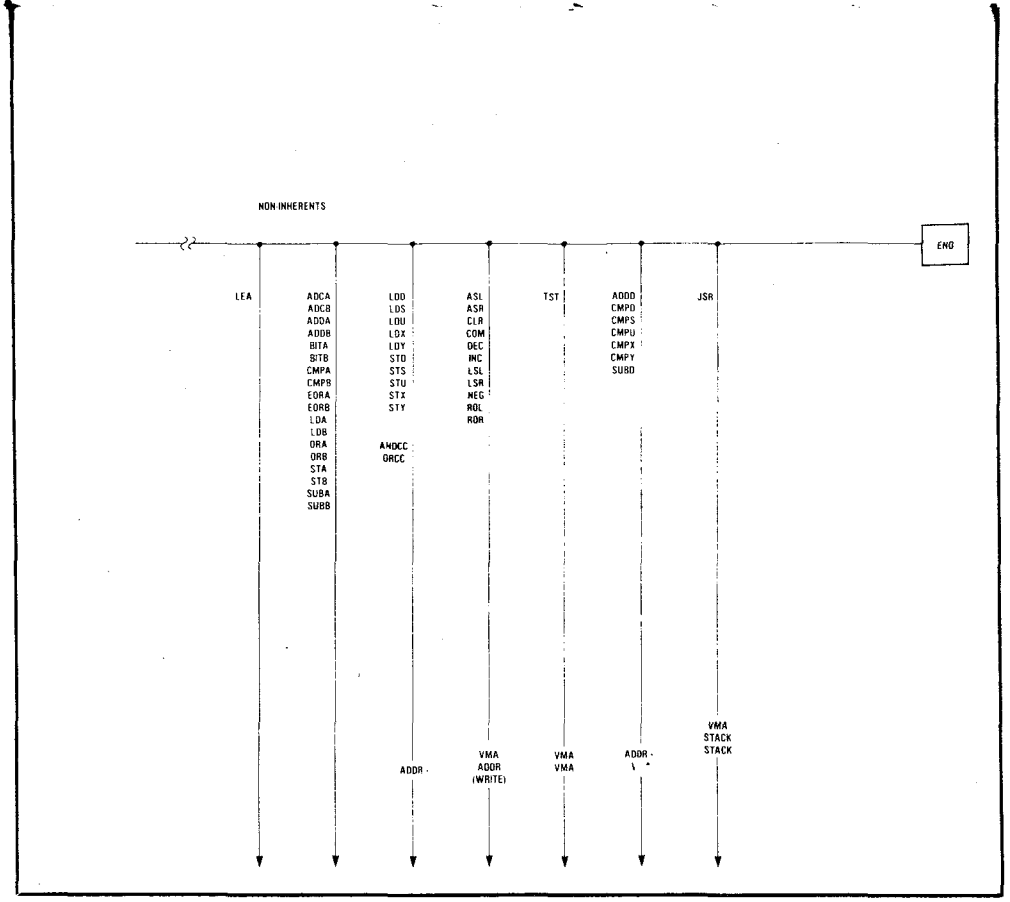
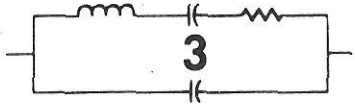


Fig. 2-3. Address bus cycle-by-cycle performance (courtesy of American Microsystems, Inc.) (continued from page 34).

Table 2-2. Equivalent Instructions
(courtesy of Motorola Semiconductor Products Inc.).

6800 Instruction	6809 Equivalent
ABA	PSHS B; ADDA ,S+
CBA	PSHS B; CMPA ,S+
CLC	ANDCC #$FE
CLI	ANDCC #$EF
CLV	ANDCC #$FD
CPX	CMPX P
DES	LEAS -1,S
DEX	LEAX -1,X
INS	LEAS 1,-S
INX	LEAX 1,X
LDAA	LDA
LDAB	LDB
ORAA	ORA
ORAB	ORB
PSHA	PSHS A
PSHB	PSHS B
PULA	PULS A
PULB	PULS B
SBA	PSHS B; SUBA ,S+
SEC	ORCC #$01
SEI	ORCC #$10
SEV	ORCC #$02
STAA	STA
STAB	STB
TAB	TFRA,B;TSTA
TAP	TFR A,CC
TBA	TFR B,A; TST A
TPA	TFR CC.A
TSX	TFR S,X
TXS	TFR X,S
WAI	CWAI #$FF

each instruction begins with an opcode fetch. While that opcode is being internally decoded, the next program byte is fetched—the so-called *pipelining effect*. Since most instructions will use the next byte, this considerably speeds processor throughput. You will find in tracing the operation that each opcode will follow the chart, and $\overline{VMA}$ is an indication of $FFFF_{16}$ on the address bus, $R/\overline{W}$ =1 and BS = 0. Although this chart may appear out of place at this time, it is my hope that it will help reinforce the architectural design of the processor and ease your understanding of addressing and the instruction set.



Addressing Modes

The first two chapters of this book were to get you into the swing of things and hopefully spark your interest in the 6809µ. This chapter is designed to build upon the power that I have hinted lies within the miniscule dot of silicon. Therefore, let's dig in.

You probably realize that the true power of any computer, regardless of size, is its ability to access memory. The *addressing modes* that the designers build in provide that capability. Within the 6809 µP, the addressing modes make it possible to extend the basic instruction set (59 instructions) to over 14 64. This statement in itself should tell you that a lot of power is possible.

BASIC CONCEPTS

This chapter is about addressing—what it is, how it works, and what modes and/or functions you have available with the 6809. In order to do this, however, it is necessary to lay down some ground rules to assist in the understanding of the subject. Consequently, rather than develop some odd-ball method. I have opted to use the same terms and definitions that Motorola prescribes.

Therefore, in the following descriptions the term *effective address* (EA) is used. The EA is the address in memory from which the argument for an instruction is fetched or stored. In two operand instructions, such as *add to accumulator* (ADD), one of the effective operands is used as a pointer. (The accumulator is inherent and not considered an addressing mode per se').

The following several pages provide descriptions and examples of the various modes of addressing the 6809 µP. To insure that understanding is achieved, I have provided examples for each mode and in some, but not all, cases the example is described in detail. Within these examples, you will see assembler instructions (described in Chapter 5) which should not be confused with an instruction set mnemonic. Specifically, I will be using the assembler instructions ORG, EQU and FCB. As does Motorola, I will use the parentheses in the examples to indicate "the contents of the location or resistor referred to. For example, (PC) indicates the contents of the location pointed to by the PC (Program Counter). The colon (:) is used to indicate a concatenation of bytes.

Furthermore, for convenience of description, it will be understood that the PC points one byte past the last byte of the instruction op code at the beginning of instruction execution. Other descriptive notation used throughout this book and Motorola and AMI documentation are shown in Tables 3-1, 3-2 and 3-3.

To fully appreciate this chapter, and to use it, I recommend that you look at the programmer's card located in Appendix C. This card will assist you in making the connection between the addressing mode and the instruction.

Before getting into the real meat of the matter, here is a run down of the types of addressing modes that will be discussed: *inherent* (includes accumulator), *immediate, extended indirect, direct, register, indexed, zero-offset, constant offset, accumulator offset, auto increment/decrement, indexed indirect, relative, short/long relative branching* and *program counter relative addressing.*

INHERENT ADDRESSING MODE

This mode of addressing has no effective address (EA). The opcode of the instruction contains all the address information

Table 3-1. Register Addressing Notation
(courtesy of Motorola Semiconductor Products Inc.).

Accumulator	ACCA or ACCB (A OR B)
Double Accumulator	ACCA:ACCB or ACCD(D)
Index Register	IX or IY (X or Y)
Stack Register	SP or US (S or U)
Program Counter	PC (PC)
Direct Page Register	DPR (DP)
Condition Code Register	CCR (CC)

The Longer-form notation (i.e, ACCA, ACCB.ACCD, IX, IY,SP,US, PC, DPR, CCR) is used to describe the MPU resisters.The short-form notation (i.e.,A,B, D, X,Y, S, U, PC, DP, CC) is used by the 6809 assembler that is discussed later.

Accumulator Double-Accumulator Inherent

Table 3-2. Register Addressing Modes (courtesy of Motorola Semiconductor Products Inc.).

necessary. Inherent addressing instructions are the only type which do not include information in the operand field. Included in inherent addressing are : ABX, DAA, SWI, ASRA and CLRB.

Assembly Example

```
0500  5F    CLRB
0501  3F    SWI
```

In Table 3-4 accumulator B is cleared (filled with 00000000) and the processor in interrupted.

IMMEDIATE ADDRESSING

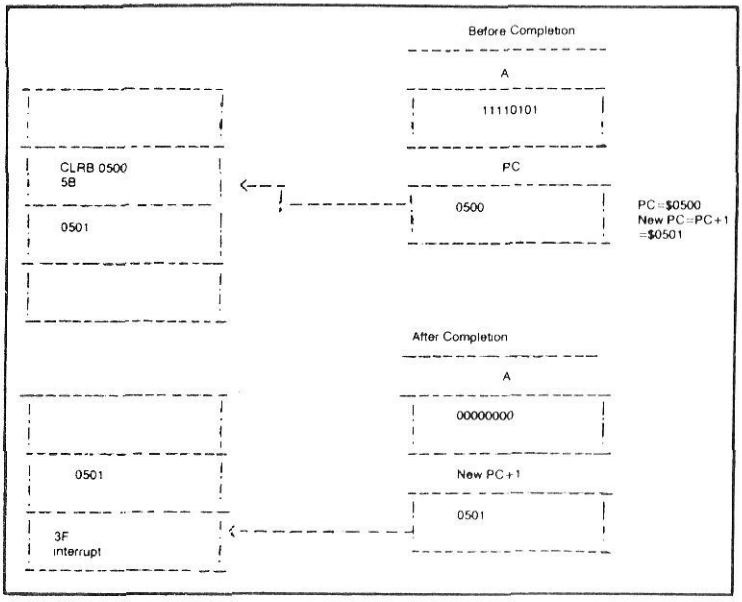
In immediate addressing, the EA of the data is the location immediately following the opcode. In other words, the data to be used in the instruction immediately follows the opcode of the instruction. The 6809 µP uses both 8 and 16-bit immediate values, depending on the size of argument specified by the opcode. Of course, immediate addressing implies that the data is a known value as the program is being created.

PC + 1 ⟶ PC
EA = PC
PC + 1 ⟶ PC

Table 3-3. Memory Addressing Notations (courtesy of Motorola Semiconductor Products Inc.).

( )	=	The (8-bit) data pointed to by the enclosed (16-bit) address.
EA	=	The Effective Address; a pointer into memory created as a result of an addressing mode.
M	=	(EA) = The data in the address space ('MEMORY') pointed to by the effective address.
MI	=	Memory Immediate Addressing; the data immediately following the last byte ot the OP code.
dd	=	8-bit. Offset, (or a relative distance to a label which evaluates to 8-bits).
DDDD	=	16-bit Offset (or a relative distance to a label).
P	=	Immediate, Direct, Indexed, Extended.
Q	=	Accumulator, Direct, indexed, Extended.
YYYY	=	Offset such that - 64K <=YYYY <= 64K.
ZZ	=	Any indexable register (IX, IY, SP, or US)
XX	=	8-bit hex value.
*	=	PC at start of present instruction
*'	=	Start of next instruction.
IN	=	Indexed Addressing only.
#	=	Immediate Addressing Bytes(s) Follow(s).
$	=	Hex Value Follows.
%	=	*Binary Value Follows.*
<	=	Before indexing: force one-byte offset form (for known forward reference, or before absolute address; force direct addressing (obtain warning If SETDP - M5 byte value
>	=	Before absolute address; force extended addressing.
,	=	Indexing symbol.
[ ]	=	Indirection.

Table 3-4. Accumulator B Is Cleared and the Processor Is Interrupted.



Assembly Examples

```
0500   86 20     LDA#$20     ;$#   signifies immediate
                                   addressing
0502   8E F000   LDX #$F000   ; $ signifies hexidecimal value
0505   10 41     LDY #$41
```

In the following example, the program says load the A accumulator with the value F8, which is the value immediately following the opcode (Fig. 3-1).

EXTENDED ADDRESSING

In extended addressing, the contents of the two bytes immediately following the opcode fully specify the 16-bit EA used by the instruction. The address generated by an extended instruction defines an absolute address and is not position independent. This addressing mode references any location available in the memory space. Extended addressing mode instructions are 3-bytes long, opcode and two-byte address.

PC + 1 → PC
EA = (PC) : (PC + 1)
PC + *2* → PC

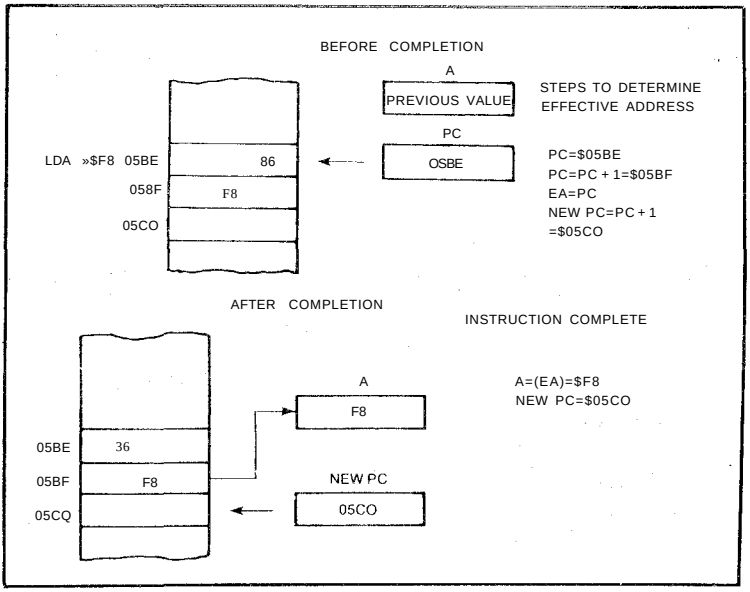


Fig. 3-1. immediate addressing mode example (courtesy of Motorola Semiconductor Products Inc.).

Assembly Example

```
          43A0        A PIG       EQU         $43A0
1000 B6 43A0                      LDA A       A PIG
```

In the following example, the program contains an instruction to load the accumulator with DOG. For this example, DOG is equal to the contents of memory location 06È5, which is the result of adding the concatenated two bytes following the opcode byte to $0000 (Fig. 3-2).

As a special case of indexed addressing, one level of indirection may be added to extended addressing. In *extended indirect,* the two bytes following the postbyte of an indexed instruction contain the address of the data.

DIRECT ADDRESSING

The EA of a direct mode instruction is the contents of the next byte of the opcode as a one-byte pointer into a single 256-byte "page" of memory. (Page is used to mean one of the 256 possible combinations of the high-order address bits). The page in use is fixed by loading the Direct Page Register with the desired high-order byte—by transferring from or exchanging with another re-

gister. As a result, the EA consists of a high-order byte, from the DP register, catenated with a low-order byte from the instruction. The direct addressing mode for the 6809 µP is directly compatible to that of the 6800 µP.

EA = DPR: (PC)

Assembly Examples

```
0500    96 30      LDA $20
0502    10         SETDP $10
0505    D6 1030    LDB $1030
```

Several things are shown here. First, this mode requires less memory and executes faster than extended addressing. Of course, only 256 location (one page) can be accessed without redefining the contents of the DP register. Indirection is not allowed with this addressing mode. The next thing demonstrated is SETDP—*Set Direct Page Pointer.*

This directive is used by the assembler. It causes the assembler's 8-bit direct page pointer to be set to the value in the operand field—in this case a hex 10. This pointer is used when the assem-

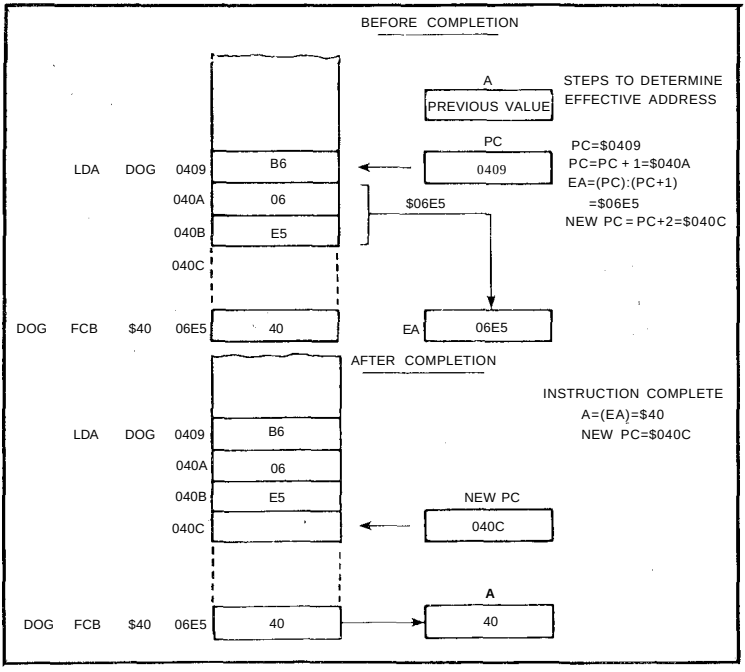


Fig. 3-2. Extended addressing mode example (courtesy of Motorola Semiconductor Products Inc.).

bler must decide whether to select the extended or direct mode of addressing. If the high or most significant (MS) byte of the EA is equal to the assembler's current direct page pointer, the direct mode is chosen. Otherwise, the extended mode is selected. The value in the operand field of the SETDP directive must be less than or equal to $FF.

In the following example of the Direct Addressing Mode the program contains an instruction to load the accumulator with CAT. For this example, CAT is equal to the contents of memory location 004B, which is the result of adding the byte following the opcode byte to $0000. Notice that this example is the same example that is used for explaining direct addressing for any of the 68XX family of processors, thus implying strict compatibility (Fig. 3-3).

REGISTER ADDRESSING

Register addressing implies no magic but merely references the selection of various on-board registers. Some of the opcodes are followed by a byte that defines a register or set of registers to be used by the instruction, which is called *a postbyte* (Table 3-5).

Examples

TFR X, Y	Transfers X into Y
EXG A, B	Exchanges A with B
PSHS A, B, X, Y	Push onto S Y,X, B then A
PULU X,Y, D	Pull from U D,X, then Y

In the following assembly example, the REG—register directive—is used to define specific registers for specific labels. See Chapter 5. The registers are then pushed and pulled from the stack in the order that is characteristic of the 6809 µP. See Table 2-1.

Assembly Example

```
                 000F   DOG    REG     A,B,XX,DP
                 0070   CAT    REG     S,X,Y
                 0070   FROG   REG     U,X,Y
0000   36        70     PSHU   #CAT
0002   35        70     PULS   #FROG
0004   34        0F     PSHS   #DOG
```

The interesting thing about this example, used courtesy of Motorola, is that a label assigned a value using the REG directive which contains the U register may not be used with the PSHU

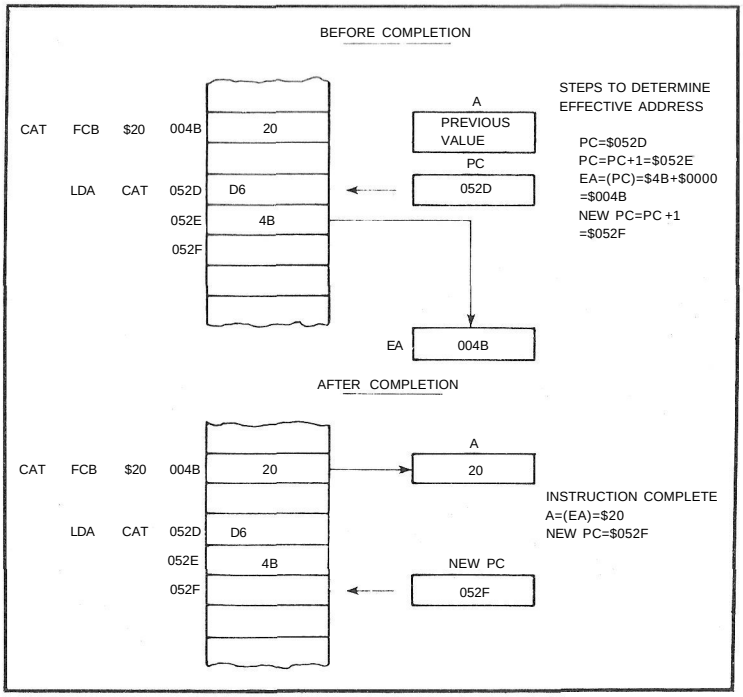


Fig. 3-3. Direct addressing mode example (courtesy of Motorola Semiconductor Products Inc.).

instruction. Similarly, a value formed using the S register may not be used with PSHS instruction. The assembler will flag either of these forms with an error message.

INDEXED ADDRESSING

In all indexed addressing one of the pointer registers (X, Y, U, S and sometimes PC) is used in a calculation of the effective address (EA) of the operand to be used by the instruction. Five

Table 3-5. Push/Pull Postbyte (courtesy of American Microsystems, Inc.).

Push/Pull Postbyte

	← PULL ORDER			PUSH ORDER →				
PC	U	Y	X	DP	B	A	CC	PSHS/PULS
FFFF.	← INCREASING MEMORY ADDRESS →						0000	
PC	S	Y	X	DP	B	A	CC	PSHU/PULU

basic types of indexing are available and are included in this discussion. The postbyte of an indexed instruction specifies the basic type and variation of the addressing mode as well as the pointer register to be used. Table 3-6 lists the legal formats for the postbyte. Table 3-7 gives the assembler form and the number of cycles and bytes added to the basic values for indexed addressing for each variation. As a result of processor compatibility, most 6800 µP index mode instructions will map into an equivalent two bytes on the 6809 µP.

Zero-Offset Indexed

This option allows selection of auto increment/decrement by one or two bits; it is a minimum two-byte instruction (opcode + postbyte). When in this mode, the selected pointer register contains the EA of the data to be used by the instruction. This is the fastest indexing mode.

Examples

```
LDD 0,X,
LDA 0,S
```

Constant Offset Indexed

When this mode of addressing is used, a two's complement, offset and the contents of one of the pointer registers are added to form the effective address (EA) of the operand. The pointer register's initial content is unchanged by the addition. Three sizes of offset are available.

±4-bit (-16 to +15)
±7-bit (-128 to +127)
±15-bit (-32768 to +32767)

Constant offset, ±4 bits, use bit 4 of the postbyte as a sign bit and bits 0 through 3 as a constant offset. It is a minimum two-byte instruction.

Constant offset, ± 7 bits, designates the byte after the postbyte as a two's complement offset. It is a minimum three-byte instruction—opcode + postbyte + offset.

Constant offset, ± 15 bits, specifies the two bytes following the postbyte to be two's complement offset. It is a minimum four-byte instruction—opcode + postbyte + two-byte offset.

Other options are the two's complement 5-bit offset that is included in the postbyte and is most efficient in use of bytes and cycles. The two's complement 8-bit offset is contained in a single

Table 3-6. Indexed Addressing Postbyte Register Bit Assignments (courtesy of American Microsystems, Inc.).

Bit Assignments

POST-BYTE REGISTER BIT								INDEXED ADDRESSING MODE
7	6	5	4	3	2	1	0	
0	R	R	X	X	X	X	X	EA = ,R ± 4-BIT OFFSET
1	R	R	0	0	0	0	0	,R+
1	R	R	I	0	0	0	1	,R+ +
1	R	R	0	0	0	1	0	,-R
1	R	R	I	0	0	1	1	, - - R
1	R	R	I	0	1	0	0	EA = ,R ± 0 OFFSET
1	R	R	I	0	1	0	1	EA = ,R±ACCB OFFSET
1	R	R	I	0	1	1	0	EA = ,R±ACCA OFFSET
1	R	R	I	1	0	0	0	EA = ,B ± 7-BIT OFFSET
1	R	R	I	1	0	0	1	EA = ,R ± 15-BIT OFFSET
1	R	R	I	1	0	1	1	EA = ,R±D OFFSET
1	X	X	I	1	1	0	1	EA = ,PC ± 7-BIT OFFSET
1	X	X	I	1	1	0	1	EA = ,PC ± 15-BIT OFFSET
1	R	R	1	1	1	1	1	EA = ,ADDRESS

ADDRESSING MODE FIELD

INDIRECT FIELD
SIGN BIT WHEN B7 = 0

REGISTER FIELD
00:R = X
01:R = Y
10:R = U
11:R = S
X = DON'T CARE

byte following the postbyte, and the two's complement 16-bit offset is in the two-bytes following the postbyte. As a programmer you will normally not worry about the offset, since the assembler should take it into account.

Examples

LDA 23,X
LDX 2,S
LDY 300,X

Example of constant-offset indexed indirect

```
LDA [, X]          (note: the brackets indicate indirection)
LDB [0,Y]
LDX [64000,S]
```

Constant offset indexed indirect addressing functions in two stages like all indirects. First, the indexed address is formed by temporarily adding the offset-value contained in the addressing byte(s) to the value from the selected pointer register (X, Y,S,U, or PC). Then this address is used to recover a two-byte absolute pointer which is used as the EA.

The following example of the indexed addressing mode with a 16-bit offset contains an instruction to load the accumulator with a tabular value containing the hexadecimal number $DB (Fig. 3-4). This value is located in memory location 0780, which is the result of adding the concatenated two bytes following the opcode byte to the contents of the index register. Take out your programmer's calculator and add up the values to see what you get. From Fig. 3-4 you can see that this mode allows the programmer to use a "table of pointer" data structures, or to do I/O through absolute values stored on the stack.

Accumulator-Offset Indexed

When this option is chosen, it designates the A, B or D register as two's complement offset. The instruction is a minimum

Table 3-7. indexed Addressing Modes (courtesy of American Microsystems, Inc.}.

Type	Forms	Non Indirect		+ ~	+ #	Indirect		+ ~	+ #
		Assembler Form	Postbyte Op Code			Assembler Form	Postbyte OP Code		
Constant, Offset Form R (Signed Offsets)	No Otlset	R	1RR00100	0	0	[R]	1RR10100	3	0
	5-Bit Offset	n,R	0RRnnnnn	1	0	default to 8 bit			
	8 Bit Offset	n,R	1RR01000	1	1	[n,R]	1RR11000	4	1
	16-Bit Offset	n,R	1RR01001	4	2	[n,R]	1RR11001	7	2
Accumulator Offset from R (Signed Offset)	A — Register Offset	A,R	1RR00110	1	0	[A,R]	1RR10110	4	0
	B — Register Offset	B,R	1RR0010I	1	0	[B,R]	1RR10101	4	0
	D — Register Offset	D,R	1RR01011	4	0	[D,R]	1RR11011	7	0
Auto Increment/Decrement R	Increment By 1	,R+	1RR00000	2	0	not allowed			
	Increment By 2	R++	1RR00001	3	0	[R++]	1RR10001	6	0
	Decrement By 1	-R	1RR00010	2	0	not allowed			
	Decrement By 2	--R	1RR00011	3	0	[--R]	1RR10011	6	0
Constant Offset From PC	8-Bit Offset	n,PCR	1XX01100	1	1	[n,PCR]	1XX11100	4	1
	16-Bit Offset	n,PCR	1XX01101	5	2	[n,PCR]	1XX11101	8	2
Extended indirect	16 Bit Address	-	-	-	-	[n]	10011111	5	2

+ and -, indicate the number or additional cycles and bytes for the particular variation

R=X, Y. U or S X = 00 Y = 01
X=Don't Care U = 10 S = 11

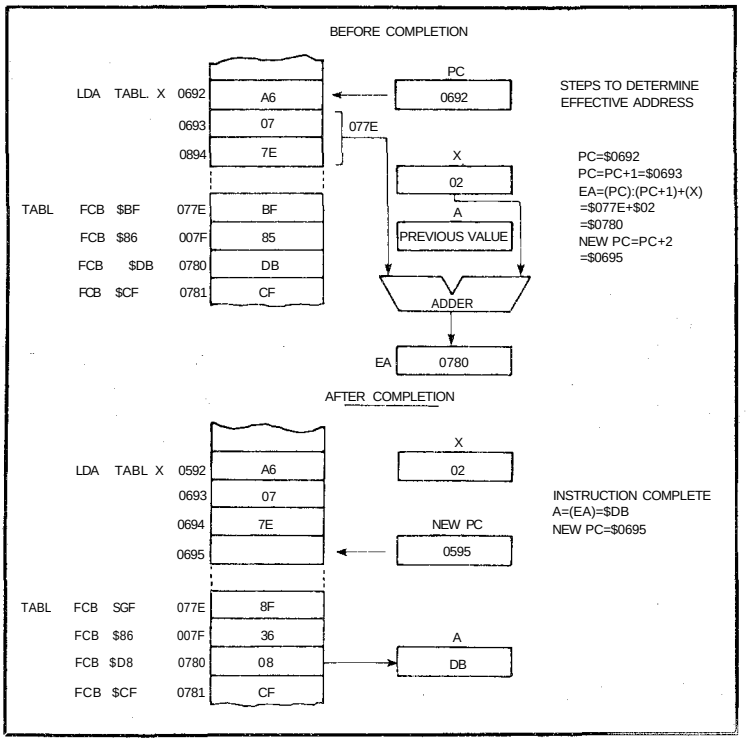


Fig. 3-4. Indexed addressing, mode, 16-bit offset example (courtesy of Motorola Semiconductor Products, Inc.).

of two-bytes. However, in all cases the offset is temporarily added to the contents of the selected pointer register to form an EA.

This mode is similar to constant offset indexed except that the two's complement value in one of the accumulators (A, B, or D), and the content of one of the pointer registers, are added for the EA as stated earlier. It is important to realize that when this process takes place, neither the contents of the accumulator or the pointer register are changed as a result of the addition. Furthermore, the postbyte specifies which accumulator to use as an offset. No additional bytes are required. The value of using an accumulator offset is that the value of the offset can be calculated by a program at run-time, thus relieving the programmer.

Examples

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

Accumulator-indexed indirect addressing uses an accumulator (A,B or D) as a two's complement offset which is temporarily added to the value from the selected pointer register (X,Y,S, or U). The resulting pointer is then used to recover another pointer from memory—the indirect notation—which is then used as the EA.

Auto Increment/Decrement Indexed

When the auto increment addressing mode is chosen, the pointer register contains the address of the operand. After the pointer register is used, it is incremented by one or two. This mode is extremely useful when you want to step through tablés, move data or create software stacks. Conversely, the auto decrement mode suggests that the pointer register be decremented prior to its use as the pointer to the address of the data. This mode is very similar in operation to the increment mode, but everything is backwards. For example, tables would be scanned from the high to low addresses.

As indicated, the increment or decrement can be one or two to all for 8 or 16-bit tables. Of course, the step is programmer selectable. Because the decisions can be made before run-time, the programmer can establish additional software stacks that are identical to the U and S stacks.

Examples

```
LDA   ,X      LDX   ,X++
LDA   ,Y+     LDX   ,Y++
LDA   ,S+     LDX   ,U++
LDA   ,U+     LDX   ,S++
```

Notice that the value in the selected pointer register addresses a one or two byte value in memory. No offset is permitted in this mode.

Example

```
LDA   [,X++]
LDB   [,Y  ++]
LDD   [,S++]
LDX   [,U++]
```

This mode references auto-increment indirect. It uses the value in the selected pointer register (X,Y,S or U) to recover an address value from memory. This value is used as the EA. The register is then incremented by two (+ +)—incrementing by one in the indirect mode is illegal and no offset is permitted.

Example

```
LDA ,-X LDX ,--X
LDA ,-Y LDX ,--Y
LDA ,-U LDX ,--U
LDA ,-S LDX ,--S
```

In the auto-decrement addressing mode, the selected pointer register (X, Y,S or U) is decremented by one (-) or two (--) and is user selectable. The resulting value then becomes the EA.

Example

```
LDA   [,--X]
LDB   [,--Y]
LDD   [,--U]
LDX    [,--S]
```

Auto-decrement indirect first decrements the selected pointer register by two (--). An auto-decrement of one is prohibited. The resulting value is used to recover a pointer value from memory and is the EA.

INDEXED INDIRECT

With the exception of the ± 4-bit constant offset and the auto-increment/decrement by one, all indexed addressing modes may be used with an additional level of indirection. The address formed by adding the offset to the selected pointer register designates a location containing the EA of the operand data. Bit 4 of the postbyte is used to select the indexed indirect mode. Interestingly enough, this same bit (bit 4) is used as a sign bit in the ± 4-bit constant offset mode. Regardless of indexing mode direct or indirect, the same number of bytes are used.

In this indirect mode, the EA is contained at the location specified by the content of the index register plus any offset. In the following example, the A accumulator is loaded indirectly using an EA calculated from the index register and an offset. It is reprinted courtesy of Motorola.

Example

```
                                        Before execution
                                  A  AA  =  XX (don't care)
                                        X =$F000

$0100   LDA      10.X                   EA is now$F010
$F010   $F1                             F150 is now the new EA
$F011 $50Q
$F150   $AA                             After Execution
                                  A  AA=$AA   Actual Data Loaded
```

RELATIVE ADDRESSING

Relative addressing involves adding a signed constant to the contents of the program counter. When this mode is used in conjunction with a branch instruction, the sum becomes the new PC content if the branch is taken; if not the PC merely advances to the next instruction. For example, the bytes following the branch opcode are treated as a signed offset which is added to the program counter. All of memory can be reached in long relative addressing as an EA is interpreted modulo 2^{16}. The following example is reprinted courtesy of American Microsystems, Inc. (AMI).

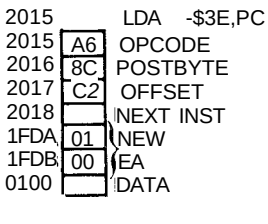
Example

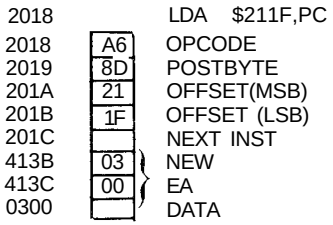
```
        BEQ     CAT     (short)
        BGT     DOG     (short)
CAT     LBEQ    RAT     (long)
DOG     LBGT    RABBIT  (long)
        .
        .
        .
RAT     NOP
RABBIT  NOP
```

According to Motorola and AMI, relative addressing differs from that contained in the 6800 µP due to two important additions. The first of these is that the offset—signed constant—can be either ± 7 bits or ± 15 bits in length. This feature permits the program to branch to any location in memory.

The second most important addition is that the relative mode is no longer limited to branch instructions. An EA which retains the position-independent nature of relative addressing may be formed by adding a ± 7-bit or ± 15-bit offset to the program counter. Doing this in-effect is an indexed addressing mode with one or two specific postbytes. The examples are reprinted courtesy of American Microsystems, Inc.

Examples





Note: the offset is added to the new value of the PC.

Table 3-8. 8-Bit Accumulator and Memory Instructions (courtesy of American Microsystems, Inc.).

8-Bit Accumulator and Memory Instructions		Addressing Modes								
Mnemonic(s)	Operation	Implied	Immediate	Direct	Extended	Extended Indirect	Indexed	Indexed Indirect	Relative	Relative Indirect
ADCA, ADCB	Add memory to accumulator with carry	—	X	X	X	X	X	X	X	X
ADDA, ADDB	Add memory to accumulator	—	X	X	X	X	X	X	X	X
ANDA, ANDB	And memory with accumulator	—	X	X	X	X	X	X	X	X
ASL	Arithmetic shift left memory location	—	—	X	X	X	X	X	X	X
ASLA, ASLB	Arithmetic shift left accumulator	X	—	—	—	—	—	—	—	—
ASR	Arithmetic shift right memory location	—	—	X	X	X	X	X	X	X
ASRA, ASRB	Arithmetic shift right accumulator	X	—	—	—	—	—	—	—	—
BITA, BITB	Bit test memory with accumulator	—	X	X	X	X	X	X	X	X
CLR	Clear memory location	—	—	X	X	X	X	X	X	X
CLRA, CLRB	Clear accumulator	X	—	—	—	—	—	—	—	—
CMPA, CMPB	Compare memory with accumulator	—	X	X	X	X	X	X	X	X
COM	Complement memory location	—	—	X	X	X	X	X	X	X
COMA, COMB	Complement accumulator	X	—	—	—	—	—	—	—	—
DAA	Decimal adjust A-accumulator	X	—	—	—	—	—	—	—	—
DEC	Decrement memory location	—	—	X	X	X	X	X	X	X
DECA, DECB	Decrement accumulator	X	—	—	—	—	—	—	—	—
EORA, EORB	Exclusive or memory with accumulator	—	X	X	X	X	X	X	X	X
EXG R1, R2	Exchange R1 with Rs (R1. R2 = A, B, CC, DP)	X	—	—	—	—	—	—	—	—
INC	Increment memory location	—	—	X	X	X	X	X	X	X
INCA, INCB	Increment accumulator	X	—	—	—	—	—	—	—	—
LDA, LDB	Load accumulator from memory	—	X	X	X	X	X	X	X	X
LSL	Logical shift left memory location	X	—	X	X	X	X	X	X	X
LSLA, LSLB	Logical shift left accumulator	X	—	—	—	—	—	—	—	—
LSR	Logical shift right memory location	—	—	X	X	X	X	X	X	X
LSRA, LSRB	Logical shift right accumulator	X	—	—	—	—	—	—	—	—
MUL	Unsigned multiply (AXB=D)	X	—	—	—	—	—	—	—	—
NEG	Negate memory *location*	—	—	X	X	X	X	X	X	X
NEGA, NEGB	Negate Accumulator	X	—	—	—	—	—	—	—	—
ORA, ORB	Or memory with accumulator	—	X	X	X	X	X	X	X	X
ROL	Rotate memory location left	—	—	X	X	X	X	X	X	X
ROLA, ROLB	Rotate accumulator left	X	—	—	—	—	—	—	—	—
ROR	Rotate memory location right	—	—	X	X	X	X	X	X	X
RORA, RORB	Rotate accumulator right	X	—	—	—	—	—	—	—	—
SBCA, SBCB	Subtract memory from accumulator with borrow	—	X	X	X	X	X	X	X	X
STA, STB	Store accumulator to memory	—	—	X	X	X	X	X	X	X
SUBA, SUBB	Subtract memory from accumulator	—	X	X	X	X	X	X	X	X
TST	Test memory location	—	—	X	X	X	X	X	X	X
TSTA, TSTB	Test accumulator	X	—	—	—	—	—	—	—	—
TFR,R1, R2	Transfer R1 to R2 (R1, R2 = A. B. CC. DP)	X	—	—	—	—	—	—	—	—

NOTE: A and 8 may be pushed to (pulled from) either stack with PSHs. PSHU (PULS. PULU) instructions.

Relative Indirect

This mode in actual use is indexed with the PC being used as the index register or in concert with the prime register. One or two bytes past the postbyte are used to provide a ± 7 bit or ± 15 bit offset. The resulting signed number is then added to the contents of the PC, which then forms a pointer to consecutive locations in memory that contain the new EA. This example is courtesy of American Microsystems, Inc.

Table 3-9. 16-Bit Accumulator and Memory Instructions (courtesy of American Microsystems, Inc.).

16-Bit Accumulator and Memory Instructions		Addressing Modes								
Mnomonic(s)	Operation	Implied	Immediate	Direct	Extended	Extended Indirect	Indexed	Indexed Indirect	Relative	Relative Indirect
ADDD	Add memory to D accumulator	—	X	X	X	X	X	X	X	X
CMPD	Compare memory with D accumulator	—	X	X	X	X	X	X	X	X
EXG D,R	Exchange D with X, Y S. U, or PC	X	—	—	—	—	—	—	—	—
LDD	Load D accumulator from memory	—	X	X	X	X	X	X	X	X
SEX	Sign Extend	X	—	—	—	—	—	—	—	—
STD	Store D accumulator to memory	—	—	X	X	X	X	X	X	X
SUBD	Subtract memory from D accumulator	—	X	X	X	X	X	X	X	X
TFR D,R	Transfer D to X. Y, S, U. or PC	X	—	—	—	—	—	—	—	—
TFR R,D	Transfer X. Y. S, U, or PC to D	X	—	—	—	—	—	—	—	—

Example

2015	LDA	-$3E,PC	2018	LDA	$2115,PC
2015	A6	OPCODE	2018	A6	OPCODE
2016	9C	POSTBYTE	2019	9D	POSTBYTE
2017	C2	OFFSET	201A	21	OFFSET (MSB)
2018		NEXT INST	201B	1F	OFFSET (LSB)
			201C		NEXT INST
1FDA	01	NEW	413B	03	NEW
1FDB	00	EA	413C	00	EA
0100		DATA	0300		DATA

Table 3-10. Index Register/Stack Pointer Instructions (courtesy of American Microsystems, Inc.).

index Register/Stack Pointer Instructions		Addressing Modes								
Mnemonic(s)	Operation	implied	Immediate	Direct	Extended	Extended indirect	Indexed	Indexed indirect	Relative	Relative Indirect
CMPS. CMPU	Compare memory with stack pointer	—	X	X	X	X	X	X	X	X
CMPX. CMPY	Compare memory with index register	—	X	X	X	X	X	X	X	X
EXG R1. R2	Exchange D, X, Y, S, U, or PC with D, X, Y, S, U, or PC	X	—	—	—	—	—	—	—	—
LEAS, LEAU	Load effectfve address into stack pointer	—	—	—	—	X	X	X	X	X
LEAX. LEAY	Load effective address into index register	—	—	—	—	X	X	X	X	X
LDS, LDU	Load slack pointer from memory	—	X	X	X	X	X	X	X	X
LDX, LDV	Load index register trom memory	—	X	X	X	X	X	X	X	X
PSHS	Push any register(s) onto hardware stack (except S)	X	—	—	—	—	—	—	—	—
PSHU	Push any register(s) onto user stack (except U)	X	—	—	—	—	—	—	—	—
PULS	Pull any register(s) trom hardware stack (except S)	X	—	—	—	—	—	—	—	—
PULU	Pull any register(s) from hardware stack (except U)	X	—	—	—	—	—	—	—	—
STS, STU	Store stack pointer to memory	—	—	X	X	X	X	X	X	X
STX, STY	Store index register to memory	—	—	X	X	X	X	X	X	X
TFR R1,R2	Transfer D, X, U. or PC to 0, X, S, U, or PC	X	—	—	—	—	—	—	—	—
ABX	Add B-accumulator to X (unsigned)	X	—	—	—	—	—	—	—	—

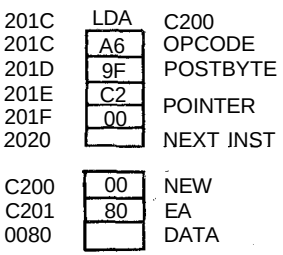
Table 3-11. Branch Instructions (courtesy of American Microsystems, Inc.).

Branch Instructions		Addressing Modes								
Mnemonic(s)	Operation	Implied	Immediate	Direct	Extended	Extended Indirect	Indexed	Indexed Indirect	Relative	Relative Indirect
BCC. LBCC	Branch if carry clear	—	—	—	—	—	—	—	X	—
BCS. LBCS	Branch if carry set	—	—	—	—	—	—	—	X	—
BEQ. LBEQ	Branch if equal	—	—	—	—	—	—	—	X	—
BGE. LBGE	Branch if greater than or equal (signed)	—	—	—	—	—	—	—	X	—
BGT. LBGT	Branch if greater (signed)	—	—	—	—	—	—	—	X	—
BHI. LBHI	Branch if higher (unsigned)	—	—	—	—	—	—	—	X	—
BHS. LBHS	Branch if higher or same (unsigned)	—	—	—	—	—	—	—	X	—
BLE. LBLE	Branch if less than or equal (signed)	—	—	—	—	—	—	—	X	—
BLO. LBLO	Branch if lower (unsigned)	—	—	—	—	—	—	—	X	—
BLS. LBLS	Branch if lower or same (unsigned]	—	—	—	—	—	—	—	X	—
BLT. LBLT	Branch if less than (signed)	—	—	—	—	—	—	—	X	—
BMI. LBMI	Branch if minus	—	—	—	—	—	—	—	X	—
BNE. LBNE	Branch if not equal	—	—	—	—	—	—	—	X	—
BPL. LBPL	Branch if plus	—	—	—	—	—	—	—	X	—
BRA. LBRA	Branch always	—	—	—	—	—	—	—	X	—
BRN. LBRN	Branch never (3.5 Cycle NOP)	—	—	—	—	—	—	—	X	—
BSR. LBSR	Branch to subroutine	—	—	—	—	—	—	—	X	—
BVC. LBVC	Branch if overflow clear	—	—	—	—	—	—	—	X	—
BVS. LBVS	Branch if overllow set	—	—	—	—	—	—	—	X	—

Extended Indirect

This is another option of indexed indirect addressing. For the extended mode, two bytes following the postbyte are used as a pointer to consecutive locations in memory which contain the new effective address. The example is courtesy of American Microsystems, Inc.

Example



Absolute Indirect

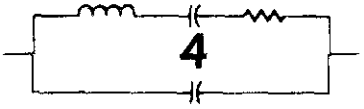
The processor must have some method of restarting and handling interrupt vectors. This addressing mode is exclusively for that purpose and no other. The conditions are serviced by fetching the contents of exact memory locations and loading it into the PC. Nothing more and nothing less happens.

Table 3-12. Miscellaneous Instructions (courtesy of American Microsystems, Inc.).

Miscellaneous Instructions		Addressing Modes								
Mnemonic(s)	Operation	Implied	Immediate	Direct	Extended	Extended Indirect	Indexed	Indexed Indirect	Relative	Relative Indirect
ANDCC	AND condition code register	—	X	—	—	—	—	—	—	—
CWA	AND condition code register, Then wait for interrupt	—	X	—	—	—	—	—	—	—
NOP	No operation	X	—	—	—	—	—	—	—	—
ORCC	OR condition code register	—	X	—	—	—	—	—	—	—
JMP	Jump	—	—	X	X	X	X	X	X	X
JSR	Jump to subroutine	—	—	X	X	X	X	X	X	X
RTI	Return trom interrupt	X	—	—	—	—	—	—	—	—
RTS	Return from subroutine	X	—	—	—	—	—	—	—	—
SWI SWI2 SWI3	Software interrupt (absolute indirect)	X	—	—	—	—	—	—	—	—
SYNC	Synchronize with interrupt line	X	—	—	—	—	—	—	—	—

SUMMARY

This chapter is a tough one to understand. I'm reasonably sure that on this first reading you haven't grasped everything that was presented. You will quite naturally have to reread this chapter and actually try the concepts explained before they really mean anything to you. However, as a quick reference I've included Tables 3-8 through 3-12 to help put the various instructions in perspective as far as addressing goes and get you ready for the next chapter on the instruction set.



Into the Instruction Set

Now that you have an understanding of how the 6809 µP works and the various methods of addressing, the next step is to become familiar with the instruction set. As discussed in Chapter 1, the 6809 is similar to that of the 6800 µP, and in most cases has the same instructions except where noted in Chapter 2. The 6809 µP as designed is upward compatible at the source level. This means that you can use 6800 instructions in a 6809 assembly and end up with a working program, which you will see in Chapter 6.

One difference that is readily discernible is the number of opcodes has been reduced from 72 to 59, primarily because of the expanded architecture and additional addressing modes. See Chapter 3. Because of the additional addressing modes, the number of available opcodes has risen from 197 to 1464—a considerable jump and indication of the type of programming power you have available to you. Before getting into a breakdown of the instruction codes, a brief overview is due to give you a better idea of what is in store.

PUSH-PULL AND ADDRESS IT

Some things you might not be aware of are the use of push (PSH) and pull (PUL), the transferring of register contents (TFR) and (EXG), the method of loading the EA (LEA), multiplying accumulators (MUL), and long and short relative branches. These and other functions of the 6809 µP are important concerns for the programmer to become familiar with and are covered here to assist in understanding.

PSHU/PSHS

The push instructions have the capability of pushing onto either hardware stack (S) or user stack (U). Any or all of the MPU register with a single instruction. In Chapter 3 I showed you how a register set could be predefined to permit pushing several defined registers on the stack at one time.

PULU/PULS

The pull instructions have the same capability of the push instruction in reverse order. The byte immediately following the push or pull opcode determines which register or registers are to be pushed or pulled. The actual PUSH/PULL sequence is fixed; each bit defines a unique register to push or pull. This push/pull postbyte was demonstrated in Table 3-5.

TFR/EXG

One of the powerful features of the 6809 µP is that any register of like size may be transferred content wise with the other, or the contents exchanged. For example, an 8-bit register can be transferred or exchanged with another 8-bit register and so on. When this feature is used, the bits 4-7 of the postbyte define the source register while bits 0-3 represent the destination. The following combinations are the valid definitions for these register transfers.

0000 - D	0101 - PC
0001 - X	1000 - A
0010 - Y	1001 - B
0011 - U	1010 - CC
0100 - S	1011 - DP

Load Effective Address (LEA)

One of the methods used by the 6809 µP to speed up processing is to use this instruction. What happens is that the LEA calculates the EA used in an indexed instruction and stores that address value, rather than the data at that address, in a pointer register. This functional addressing makes all the features of the internal addressing hardware available to the programmer, and suggests that the 6809 is a 16-bit processor in reality. Table 4-1 is an example of LEA and demonstrates its power.

Multiply (MUL)

This is a powerful instruction that multiplies unsigned binary numbers in the A and B accumulator and then places the result into

Table 4-1.LEA Examples (courtesy of American Microsystems, Inc.),

Instruction	Operation	Comment
LEAX 10,X	X+10 ->X	! Adds 5-bit constant 10 to X
LEAX 500,X	X+500 ->X	! Adds 6-bit constant 500 to X
LEAY A,Y	Y+A ->Y	! Adds 8-bit accumulator to Y
LEAY -10,U	U-10 ->U	! Subtracts 10 from 11
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.

the 16-bit D accumulator. This permits multiple-precision multiplications.

Long and Snort Relative Branches

I would imagine that the first thing that comes to mind is that this is really something difficult to master. Actually, the 6809 has the capability of PC relative branching throughout the entire memory map. When in this mode and a branch is to be taken, the 8 or 16-bit offset value is added to the PC to make the EA. Consequently, this permits the processor to branch anywhere within a 64K memory map. Position independent code can be easily generated by using relative branching. Incidentally, short refers to 8-bit and long to 16-bit.

SYNC

This is a unique instruction since it stops the MPU and makes it wait for an interrupt. If the pending interrupt is nonmaskable (NMI) or maskable (FIRQ, IRQ) with its mask bit (F or I) clear, the processor will clear the Sync state and perform the normal interrupt stacking and servicing routine. You can see that this makes it possible to handle specialized interrupts and develop programs that work well in process control or data acquisition.

Software Interrupts (SWB)

If you are familiar with the 6800 µP, then you have some ideas what a software interrupt is for. It is the instruction that will cause an interrupt in the course of program execution and will permit a goto for the associated vector fetch. Three levels of SWI are available on the 6809 and have a priority status of SWI, SWI2 and SWI3.

18-Bit Operations

These operations make the 6809 a high-powered µP and excellent precursor to 16-bit processors. The 6809 can process 16-bit datas on an 8-bit structure with almost the same power as its big brother the 68000. Included in these 16-bit instructions are: loads, stores, compares, adds, subtracts, transfers, exchanges, pushes and pulls. Refer again to Tables 3-8 through 3-12 which are summaries of the instruction set. Associated with this chapter is Appendix B which covers the hexadecimal values of machine codes, coupled with Appendix C, the programmer's quick reference card.

INDIVIDUAL INSTRUCTIONS

The next several pages will cover each instruction available on the 6809 µP. You will notice that in concert with the instruction mnemonic, I have provided the various addressing modes and the associated opcode. This sequence of presentation is coupled with the Instruction Index, located in Appendix D. The purpose is to assist you in finding the proper instruction for a particular purpose. I would suggest that as you proceed through this section of the chapter you look at the programmer's card, found in Appendix C, and use it to follow along. This will help you become familiar with each instruction and the card.

See Tables 4-2 and 4-3 for the notation that is used in the explanation of the instruction set. The notation is used by Motorola and consequently provides continuity in explanation.

ABX **ADD ACCB INTO IX**

SOURCE FORM: ABX

OPERATION: IX' ← IX + ACCB

CONDITION CODES: NOT AFFECTED

DESCRIPTION:

Add the 8-bit unsigned value in Accumulator B into the X index register.

ADDRESSING MODE:	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	3A	3	1

ADC ADD WITH CARRY MEMORY INTO REGISTER

SOURCE FORM: ADCAP ; ADCB P

OPERATION: R' ← R + M + C

CONDITION CODES:

H: Set if the operation caused a carry from bit 3 in the ALU.

N: Set if bit 7 of the result is set.

Z: Set if all bits of the result are clear.

V: Set if the operation caused an 8-bit two's complement arithmetic overflow.

C: Set if the operation caused a carry from bit 7 in the ALU.

DESCRIPTION:

Adds the contents of the carry flag and the memory byte into an 8-bit register.

REGISTER ADDRESSING MODE: Accumulator

ADCA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
IMMEDIATE	89	2	2
DIRECT	99	4	2
INDEXED	A9	4+	2+
EXTENDED	B9	5	3

ADCB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
IMMEDIATE	C9	2	2
DIRECT	D9	4	2
INDEXED	E9	4 +	2+
EXTENDED	F9	5	3

Operation Notation	
←	= is transferred
∧	= Boolean AND
∨	= Boolean OR
⊕	= Boolean EXCLUSIVE-OR
-	= (overline) = Boolean NOT
:	= Concatenation

Table 4-2. Operation Notation (courtesy of Motorola Semiconductor Products Inc.).

ADD ADD MEMORY INTO REGISTER - 8-BIT

SOURCE FORMS: ADDA P; ADDB P

OPERATION: R' ← R + M

CONDITION CODES:

H: Set if the operation caused a carry from bit 3 in the ALU.

N: Set if bit 7 of the result is set

Z: Set if all bits of the result are clear.

V: Set if the operation caused an 8-bit two's complement arithmetic overflow.

C: Set if the operation caused a carry from bit 7 in the ALU.

DESCRIPTION:

Adds the memory byte into an 8-bit register.

REGISTER ADDRESSING MODE: Accumulator

ADDA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
IMMEDIATE	8B	2	2
DIRECT	9B	4	2
INDEXED	AB	4+	2+
EXTENDED	BB	5	3

ADDB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
IMMEDIATE	CB	2	2
DIRECT	DB	4	2
INDEXED	EB	4+	2+
EXTENDED	FB	5	3

ADDD

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
IMMEDIATE	C3	4	3
DIRECT	D3	6	2
INDEXED	E3	6+	2+
EXTENDED	F3	7	3

This instruction ADDD is the 16-bit version. For this the 16-bit version. For this the operation is R' ← R + M:M+1. The condition codes are: H: not affected; N: Set if bit 15 of the result

Register Notation		
ACCA	= A	= Accumulator A
ACCB	= B	= Accumulator B
ACCX	=	Either ACCA or ACCB
ACCA:ACCB	= D	= Double Accumulator
IX	= X	= Index Register X
IY	= Y	= Index Register Y
SP	= S	= Hardware Stack Pointer
US	= U	= User Stack Pointer
DPR	= DP	= Direct Page Resister
CCR	= CC	= Condition Code Resistor
PC	=	Program Counter
R	=	A register before the operation; A,B,C,D,X,Y,U,S,PC,DP or CC (usually, only a subset of registers is legal, these are specified by 'Register Addressing Mode' in the individual instructions).
R'	=	A register after the operation
ALL	=	All Registors; A,B,D,X,Y,U,S,PC,DP and CC
ZZ	=	A pointer register; X,Y,U,S
MSB	=	Most-Significant BIT
MS BYTE	=	Most-Significant BYTE
LS BYTE	=	Least-Significant BYTE
IXH	=	MS Byte of Index X
IXL	=	LS Byte of Index X

Table 4-3. Register Notation (courtesy of Motorola Semiconductor Products Inc.).

is set; Z: Set if all bits of the result are clear; V: Set if there was a 16-bit two's complement arithmetic overflow; and C: set if the operation on the MS byte caused a carry from bit 7 in the ALU. This instruction adds the 16-bit memory value into the 16-bit accumulator (D) and has a register addressing mode of double accumulator. The memory addressing modes are shown above.

In the next group of instructions, the logical AND is implied. The logical AND is best explained by assuming that it has the property such that if X and Y are two logic variables, then the function X AND Y is defined by the following:

X	Y	X AND Y	X	Y	X AND Y
0	0	0	1	0	0
0	1	0	1	1	1

A basic operation in Boolean algebra is the AND operation which, for the two integers I and J, may be defined by saying if I and J are both 1, then the result is 1. If I is 0 and J is 1, then the result is 0 and vice versa.

AND LOGICAL AND MEMORY INTO REGISTER

SOURCE FORMS: ANDA P; ANDB P

OPERATION: R' ← R ∧ AND M

CONDITION CODES:

H: Not Affected
N: Set if bit 7 of result is set
Z: Set if all bits of result are clear
V: Cleared
C: Not affected

DESCRIPTION:

Performs the logical "AND" operation between the contents of ACCX and the contents of M and the result is stored in ACCX.

REGISTER ADDRESSING MODE: Accumulator

ANDA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYYTES
IMMEDIATE	84	2	2
DIRECT	94	4	2
INDEXED	A4	4+	2+
EXTENDED	B4	5	3

ANDB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
IMMEDIATE	C4	2	2
DIRECT	D4	4	2
INDEXED	E4	4+	2+
EXTENDED	F4	5	3

AND LOGICAL AND IMMEDIATE MEMORY INTO CCR

SOURCE FORM: ANDCC #XX
OPERATION: R' ← R ∧ MI AND
CONDITION CODE-S'. CCR' ← CCR ∧ MI
DESCRIPTION:

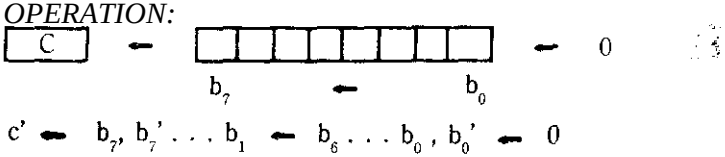
Performs a logical "AND" between the CCR and the MI byte and places the result in the CCR.

ANDCC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
MEMORY IMMEDIATE	1C	3	2

ASL **ARITHMETIC SHIFT LEFT**

SOURCE FORM: ASL Q
OPERATION:



$c' \leftarrow b_7, b_7' \ldots b_1 \leftarrow b_6 \ldots b_0, b_0' \leftarrow 0$

CONDITION CODES:

H: Undefined
N: Set if bit 7 of the result is set
Z: Set if all bits of the result are clear
V: Loaded with the result of $(b_7 \oplus b_8)$ of the original operand.
C: Loaded with bit 7 of the original operand.

DESCRIPTION:

Shifts all bits of the operand one place to the left. Bit 0 is loaded with a zero. Bit 7 of the operand is shifted into the carry flag.

ASIA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
ACCUMULATOR	48	2	1

ASLB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
ACCUMULATOR	58	2	1

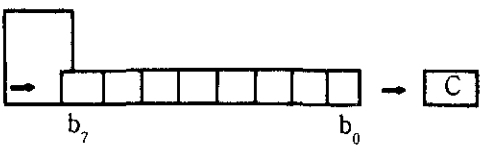
ASL

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	08	6	2
EXTENDED	78	7	3
INDEXED	68	6+	2+

ASR **ARITHMETIC SHIFT RIGHT**

SOURCE FORM: ASR Q

OPERATION:



$C' \leftarrow b_0,\ b_6' \ldots b_0' \leftarrow b_7 \ldots b_1, b_7' \leftarrow b_7$

CONDITION CODES:

H: Undefined
N: Set if bit 7 of the result is set
Z: Set if all bits of result are clear
V: Not affected
C: Loaded with bit 0 of the original operand

DESCRIPTION:

Shifts all bits of the operand right one place. Bit 7 is held constant. Bit 0 is shifted into the carry flag. The 6800/01/02/03/08 processors do affect the V flag.

ASR

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	57	2	1
DIRECT	07	6	2
EXTENDED	77	7	3
INDEXED	67	6 +	2+

ASRA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	47	2	1

BCC **BRANCH ON CARRY CLEAR**

SOURCE FORM: BCC dd; LBCC DDDD

OPERATION: TEMP ← MI
if C = 0 THEN PC' ← PC + TEMP

CONDITION CODES:
Not affected

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

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:
When used after a subtract or compare on unsigned binary values, this instruction could be called "branch" if the resister was higher or the same as the memory operand.

BCC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	24	3	2

LBCC *LONG BRANCH*

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	10	5(6)	4

BCS **BRANCH ON CARRY SET**

OPERATION: TEMP ← MI
if C = 1 THEN PC' ← PC + TEMP

CONDITION CODES:

Not affected

DESCRIPTION:

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

MEMORY ADDRESSING MODES: Memory Immediate

COMMENTS:

When used after a subtract or compare, on unsigned binary values, this instruction could be called "branch" if the register was lower than the memory operand.

BCS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	25	3	*2*

LBCS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BEQ **BRANCH ON EQUAL**

SOURCE FORMS: BEQ dd; LBEQ DDDD

OPERATION: TEMP ← MI

if Z = 1 THEN PC' ← PC + TEMP

CONDITION CODES:

Not affected.

DESCRIPTION:

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

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:

Used after a subtract or compare operation, this instruction will branch if the compared values-signed or unsigned—were exactly the same.

BEQ

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	27	3	2

LBEQ

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BGE BRANCH ON GREATER THAN OR EQUAL TO ZERO

SOURCE FORMS: BGE dd; LBGE DDDD

OPERATION: TEMP ← MI

if [N ⊕ V] = 0 THEN PC' ← PC + TEMP

CONDITION CODES:

Not affected

DESCRIPTION:

Causes a branch if N and V are either both set or both clear. For example, branch if the sign of a valid two's complement result is, or would be, positive.

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:

Used after a subtract or compare operation on two's complement values, this instruction will branch if the register was greater than or equal to the memory operand.

BGE

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	2C	3	2

LBGE

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BGT Branch on Greater

SOURCE FORMS: BGT dd; LBGT DDDD

OPERATION: TEMP ← MI

if Z V[N ⊕ V] = 0 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if (N and V are either both set or both clear) and Z is clear. In other words, branch if the sign of a *valid* two's complement result is, or would be, positive and non-zero.

MEMORY ADDRESSING MODE: Memory Immediate

COMMENTS:

Used after a subtract or compare operation on two's complement values, this instruction will "branch if the register was greater than the memory operand."

BGT

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	2E	3	2

LBGT

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BHI **Branch if Higher**

SOURCE FORMS: BHI dd; LBHI DDDD

OPERATION: TEMP ← MI

if [C ∨Z] = 0 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if the previous operation caused neither a carry nor a zero result.

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:

Used after a subtract or compare operation on unsigned binary values, this instruction will "branch if the register was higher than the memory operand." Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

BHI

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	22	3	2

LBHI

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BHS Branch if Higher or Same

SOURCE FORM: BUS dd; LBHS DDDD

OPERATION: TEMP ← MI

if C = 0 then PC ← PC' ← PC + 1 MI

CONDITION CODES: Not Affected

DESCRIPTION:

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

MEMORY ADDRESSING MODE: Memory Immediate

COMMENTS:

When used after a subtract or compare on unsigned binary values, this instruction will "branch if register was higher than or same as the memory operand." This is a duplicate assembly-language mnemonic for the single machine instruction BCC. Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

BHS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	24	3	2

LBHS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BIT **Bit Test**

SOURCE FORM: BIT P

OPERATION: TEMP ← R ∧ M

CONDITION CODES:

H: Not Affected
N: Set if bit 7 of the result is Set
Z: Set if all bits of the result are Clear
V: Cleared
C: Not Affected

DESCRIPTION:

Performs the logical "AND" of the contents of ACCX and the contents of M and modifies condition codes accordingly. The contents of ACCX or M are not affected.

REGISTER ADDRESSING MODE: Accumulator

BITA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	95	4	2
EXTENDED	B5	5	3
IMMEDIATE	85	2	2
INDEXED	A5	4+	2+

BITB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	D5	4	2
EXTENDED	F5	5	3
IMMEDIATE	C5	2	2
INDEXED	E5	4+	2+

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

SOURCE FORM: BLE dd; LBLE DDDD

OPERATION: TEMP ← MI
if Z V (N ⊕ V) = 1 then PC' = PC + 1
TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Causes a branch if the "Exclusive OR" of the N and V bits is 1 or if Z = 1. That is, branch if the sign of a *valid* two's complement result is — or would be—negative.

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:

Used after a subtract or compare operation on two's complement values, this instruction will "branch if the register was less than or equal to the memory operand."

BLE

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	2F	3	2

LBLE

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BLO **Branch on Lower**

SOURCE FORM: BLO dd; LBLO DDDD

OPERATION: TEMP ← MI
if C = 1 then PC ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

Tests the state of the C bit and causes a branch if C is Set.

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:

When used after a subtract or compare on unsigned binary values, this instruction will "branch if the register was lower" than the memory operand. Note that this is a duplicate assembly-language mnemonic for the single machine instruction BCS. Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

BLO

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	25	3	2

LBLO

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BLS **Branch on Lower or Same**

SOURCE FORM: BLS dd; LBLS DDDD

OPERATION:TEMP ← MI
if (C ∨ Z) = 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if the previous operation caused either a carry or a zero result.

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:

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." Not useful, in general, after INC/DEC, LD/ST, TST/CLR/COM.

BLS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	23	3	2

LBLS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BLT **Branch on Less than Zero**

SOURCE FORMS: BLT dd; LBLT DDDD

OPERATION: TEMP ← MI

if (N ⊕ V) = 1 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Causes a branch if either, but not both, of the N or V bits is 1. That is, branch if the sign *of a valid* two's complement result is—or would be—negative.

MEMOR Y ADDRESSING MODE: Memory Immediate

COMMENTS:

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."

BLT

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	2D	3	2

LBLT

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BMI **Branch on Minus**

SOURCE FORM: BMI dd; LBMI DDDD

OPERATION:TEMP ← MI

if N = 1 then PC ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Tests the state of the N bit and causes a branch if N is set. That is, branch if the sign of the two's complement result is negative.

MEMORY ADDRESSING MODE: Memory Immediate

COMMENTS:

Used after an operation on two's complement binary values, this instruction will "branch if the (possibly invalid result is minus."

BMI

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	2B	3	2

LBMI

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BNE **Branch Not Equal**

SOURCE FORMS: BNE dd; LBNE DDDD

OPERATION: TEMP ← MI

if Z = 0 then PC' ← PC + TEMP

CONDITION CODES: Not Affected

DESCRIPTION:

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

MEMORY ADDRESSING MODE: Memory Immediate

COMMENTS:

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."

BNE

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	26	3	2

LBNE

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BPL **Branch of Plus**

SOURCE FORM: BPL dd; LBPL DDDD

OPERATION: TEMP ← MI

if N = 0 then PC' ← PC + TEMP

CONDITION CODES: Not affected

DESCRIPTION:

Tests the state of the N bit and causes a branch if N is clear. That is, branch if the sign of the two's complement result is positive.

MEMORY ADDRESSING MODE: Memory Immediate

COMMENTS:

Used after an operation on two's complement binary values, this instruction will "branch if the possibly invalid result is positive."

BPL

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	2A	3	2

LBPL

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF CYTES
LONG RELATIVE	10	5(6)	4

BRA **Branch Always**

SOURCE FORMS: BRA dd; LBRA DDDD

OPERATION: TEMP ← MI

PC' ← PC + TEMP

CONDITION CODES: Not Affected.

DESCRIPTION:

Causes an unconditional branch.

MEMOR Y ADDRESSING MODE: Memory Immediate

BRA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	20	3	2

LBRA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	16	5	3

BRN **Branch Never**

SOURCE FORM: BRN dd; LBRN DDDD
OPERATION: TEMP ← MI
CONDITION CODES: Not Affected
DESCRIPTION:

Does not cause a branch. This instruction is essentially a NO-OP, but has a bit pattern logically related to BRA.

MEMOR Y ADDRESSING MODE: Memory Immediate

BRN

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	21	3	2

LBRN

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5	4

BSR **Branch to Subroutine**

SOURCE FORM: BSR dd; LBSR DDDD
OPERATION: TEMP ← MI
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
PC' ← 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 memory immediate offset.

MEMOR Y ADDRESSING MODE: Memory Immediate

BSR

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	8D	7	2

LBSR

ADDRESSING MODE	*OPCODE*	MPU CYCLES	NO OF BYTES
LONG RELATIVE	17	9	3

BVC **Branch on Overflow Clear**

SOURCE FORM: BVC dd; LBVC DDDD
OPERATION: TEMP ← MI
if V = 0 then PC' ← PC + TEMP
CONDITION CODES: Not Affected
DESCRIPTION:

Tests the state of the V bit and causes a branch if the V bit is clear. That is, branch if the two's complement result was valid.

MEMOR Y ADDRESSING MODE: Memory Immediate
COMMENTS:

Used after an operation on two's complement binary values, this instruction will "branch if there was no overflow."

BVC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	28	3	2

LBVC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

BVS **Branch on Overflow Set**

SOURCE FORM: BVS dd; LBVS DDDD
OPERATION: TEMP ← MI
if V = 1 then PC' ← PC + TEMP
CONDITION CODES: Not Affected
DESCRIPTION:

Tests the state of the V bit and causes a branch if the V bit is set. That is, branch if the two's complement result was invalid.

MEMORY ADDRESSING MODE: Memory Immediate
COMMENTS:

Used after an operation on two's complement binary values, this instruction will "branch if there was an overflow." This instruction is also used after ASL or LSL to detect binary floating-point normalization.

BVS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	29	3	2

LBVS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
LONG RELATIVE	10	5(6)	4

CLR **Clear**

SOURCE FORM: CLR Q

OPERATION: TEMP ← M

M ← 00_{16}

CONDITION CODES:

H: Not affected
N: Cleared
Z: Set
V: Cleared
C: Cleared

DESCRIPTION:

ACCX or M is loaded with 00000000. The C-flag is cleared for 6800 compatibility.

CLRA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	4 F	2	1

CLRB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	5F	2	1

CLR

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	0F	6	2
EXTENDED	7F	7	3
INDEXED	6F	6+	2+

CMP Compare Memory from a Register - 8 Bits

SOURCE FORM; CMPA P; CMPB P

OPERATION: TEMP ← R - M [i.e., TEMP ← R + $\overline{M}$ + 1]

CONDITION CODES:

H: Undefined

N: Set if bit 7 of the result is Set.

Z: Set if all bits of the result are Clear.

V: Set if the operation caused an 8-bit two's complement overflow

C: Set if the subtraction *did not* cause a carry from bit 7 in the ALU

DESCRIPTION:

Compares the contents of M from the contents of the specified register and sets appropriate condition codes. Neither M nor R is modified. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING: Accumulator

FLAG RESULTS:

(N ⊕ V) = 1 R XT. M(2's comp)
C = 1 R .LO. M (unsigned)
Z = 1 R .EQ. M

CMPA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	91	4	2
EXTENDED	B1	5	3
IMMEDIATE	81	2	2
INDEXED	A1	4+	2+

CMPB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	D1	4	2
EXTENDED	F1	5	3
IMMEDIATE	C1	2	2
INDEXED	E1	4+	2+

CMP Compare Memory From a Register - 16 Bits

SOURCE FORMS: CMPD P; CMPX P, CMPY P; CMPU P; CMPS P

OPERATION: TEMP ← R - M:M+ 1) (i.e., TEMP ← R + $\overline{M:M+1}$ +1)

CONDITION CODES:

H: Unaffected

N: Set if bit 15 of the result is Set

Z: Set if all bits of the result are Clear.

V: Set if the operation caused a 16-bit two's complement overflow.

C: Set if the operation on the MS byte *did not* cause a carry from bit 7 in the ALU

DESCRIPTION:

Compares the 16-bit contents of M:M+1 from the contents of the specified register and sets appropriate condition codes. Neither R nor M:M+1 is modified. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING: Double Accumulator Pointer (X, Y, S, or U)

FLAG RESULTS:

(N ⊕ V) = 1 R .LT. M (2's comp)

C = 1 R .LO. M (unsigned)

Z = 1 R .EQ. M

CMPD

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	10 93	7	3
EXTENDED	10 B3	8	4
IMMEDIATE	10 83	5	4
INDEXED	10 A3	7 +	3+

CMPS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	11 9C	7	3
EXTENDED	11 BC	8	4
IMMEDIATE	11 8C	5	4
INDEXED	11 AC	7 +	3 +

CMPU

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	11 93	7	3
EXTENDED	11 B3	8	4
IMMEDIATE	11 83	5	4
INDEXED	11 A3	7+	3+

CMPX

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	9C	6	2
EXTENDED	BC	7	3
IMMEDIATE	8C	4	3
INDEXED	AC	6+	2+

CMPY

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	10 9C	7	3
EXTENDED	10 BC	8	4
IMMEDIATE	10 8C	5	4
INDEXED	10 AC	7+	3+

COM ***Complement***

SOURCE FORM: COM Q

OPERATION: $M \leftarrow O + \overline{M}$

CONDITION CODES:

H: Not affected
N: Set if bit 7 of the result is Set
Z: Set if all bits of the result are Clear
V: Cleared
C: Set

DESCRIPTION:

Replaces the contents of M or ACCX with its one's complement (also called the logical complement). The carry flag is set for 6800 compatibility.

COMMENTS:

When operating on unsigned values, only BEQ and MBE branches can be expected to behave properly. When operating on two's complement values, all signed branches are available.

COMA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	43	2	1

COMB

ADDRESSING MODE	*OPCODE*	MPU CYCLES	NO OF BYTES
INHERENT	53	2	1

COM

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	03	6	2
EXTENDED	73	7	3
INDEXED	63	6+	2+

CWAI **Clear and Wait for Interrupt**

SOURCE FORM: CWAI #$XX

E	F	H	I	N	Z	V	C

OPERATION:

CCR → CCR ∧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

FF = enable neither
EF = enable IRQ
BF = enable FIRQ
AF = enable both

CONDITION CODES: Possibly cleared by the immediate byte.

DESCRIPTION:

The CWAI instruction ANDs an immediate byte with the condition code register which may clear interrupt maskbit(s). It stacks the entire machine state on the hardware stack and then looks for an interrupt. When a nonmasked interrupt occurs, no further machine state will be saved before vectoring to the interrupt handling routine. This instruction replaced the 6800's CLI WAI sequence, but does not tri-state the buses.

ADDRESSING MODE: Memory Immediate

COMMENTS:

An FIRQ interrupt may enter its interrupt handler with its entire machine state saved. The RTI will automatically return the entire machine state after testing the E bit of the recovered CCR.

CWAI

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	3C	20	2

DA **Decimal Addition Adjust**

SOURCE FORM: DAA

OPERATION: ACCA' ← ACCA + CF(MSN):CF(LSN) where CF is a Correction Factor, as follows: The CF. for each nybble (BCD) digit) is determined separately, and is either 6 or 0.

Least Significant Nybble

CF(LSN) = 6 if 1) H = 1
or 2) LSN > 9

Most Significant Nybble

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

CONDITION CODES:

H: Not affected
N: Set if MSB of result is Set
Z: Set if all bits of the result are Clear
V: Not defined

C: Set if the operation caused a carry from bit 7 in the ALU, or if the carry flag was Set before the operation.

DESCRIPTION:

The sequence of a single-byte add instruction on ACCA (either ADDA or ADCA) and a following DAA instruction results in a BCD addition with appropriate carry flag. Both values to be added must be in proper BCD form (each nybble such that 0 nybble 9). Multiple-precision additions must add the carry generated by this DA into the next higher digit during the add operation immediately prior to the next DA.

ADDRESSING MODE: ACCA

DAA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	19	2	1

DEC **Decrement**

SOURCE FORM: DEC Q

OPERATION: M' ← M - 1 (i.e., M' ← M + FF_{16})

CONDITION CODES:

H: Not affected
N: Set if bit 7 of result is Set
Z: Set if all bits of result are Clear
V: Set if the original operand was 10000000
C: Not affected

DESCRIPTION:

Subtract one from the operand. The carry flag is not affected, thus allowing DEC to be a loopcounter in multiple-precision computations.

COMMENTS:

When operating on unsigned values only BEQ and BNE branches can be expected to behave consistently. When operating on two's complement values, all signed branches are available.

DECA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	4A	2	1

DECB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	5A	2	1

DEC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	0A	6	2
EXTENDED	7A	7	3
INDEXED	6A	6+	2+

EOR **Exclusive Or**

SOURCE FORMS: EORA P; EORB P

OPERATION: R' ← R ⊕ M

CONDITION CODES:

H: Not affected
N: Set if bit 7 of result is Set
Z: Set if all bits of result are Clear
V: Cleared
C: Not affected

DESCRIPTION:

The contents of memory is exclusive—ORed into an 8-bit register.

REGISTER ADDRESSING MODES: Accumulator

EORA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	9B	4	2
EXTENDED	B8	5	3
IMMEDIATE	88	2	2
INDEXED	A8	4+	2+

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	D8	4	2
EXTENDED	F8	5	3
IMMEDIATE	C8	2	2
INDEXED	E8	4+	2+

EXG **Exchange Registers**

SOURCE FORM: EXG Rl, R2

OPERATION: Rl ⟷ R2

CONDITION CODES: Not affected (unless one of the registers is CCR)

DESCRIPTION:

Bits 3-0 of the immediate byte of the instruction 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

Registers may only be exchanged with registers of like size; i.e., 8-bit with 8-bit, or 16 with 16.

EXG Rl, R2

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	1E	7	2

INC **Increment**

SOURCE FORM: INC Q

OPERATION: M' ← M + 1

CONDITION CODE:

H: Not affected
N: Set if bit 7 of the result is Set
Z: Set if all bits of the result are Clear
V: Set if the original operand was 01111111.
C: Not affected

DESCRIPTION:

Add one to the operand. The carry flag is not affected, thus allowing INC to be used as a loop-counter in multiple-precision computations.

COMMENTS:

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.

INCA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	4C	2	1

INCB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	5C	2	1

INC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	0C	6	2
EXTENDED	7C	7	3
INDEXED	6C	6+	2+

JMP **Jump to Effective Address**

SOURCE FORM: JMP

OPERATION: PC' ← EA

CONDITION CODES: Not affected

DESCRIPTION:

Program control is transferred to the location equivalent to the effective address.

JMP

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	0E	3	2
EXTENDED	7E	4	3
INDEXED	6E	3+	2+

JSR **Jump to Subroutine at Effective Address**

SOURCE FORM: JSR

OPERATION: SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
PC' ← EA

CONDITION CODES: Not affected

DESCRIPTION:

Program control is transferred to the Effective Address after storing the return address on the hardware stack.

JSR

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	9D	7	2
EXTENDED	BD	8	3
INDEXED	AD	7+	2+

LD **Load Register from Memory—8 Bit**

SOURCE FORMS: LDA P; LDB P

OPERATION: R' ← M

CONDITION CODES:

H: Not affected
N: Set if bit of loaded data is Set
Z: Set if all bits of loaded data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Load the contents of the addressed memory into the register.

REGISTER ADDRESSING MODE: Accumulator

LDA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	96	4	2
EXTENDED	B6	5	3
IMMEDIATE	86,	2	2
INDEXED	A6	4+	2+

LDB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	D6	4	2
EXTENDED	F6	5	3
IMMEDIATE	*C6*	2	2
INDEXED	E6	4 +	2+

LD **Load Register from Memory—16 Bit**

SOURCE FORM: LDD P; LDX P; LDY P; LDS P; LDU P

OPERATION: R' ← M:M+1

CONDITION CODES:

H: Not affected
N: Set if bit 15 of loaded data is Set
Z: Set if all bits of loaded data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Load the contents of the addressed memory (two consecutive memory locations) into the 16-bit register.

REGISTER ADDRESSING MODES: Double Accumulator Pointer (X, Y, S, or U)

LDD

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	DC	5	2
EXTENDED	FC	6	3
IMMEDIATE	CC	3	3
INDEXED	EC	5+	2+

LDS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	10 DE	6	3
EXTENDED	10 FE	7	4
IMMEDIATE	10 CE	4	4
INDEXED	10 EE	6+	3+

LDU

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	DE	5	2
EXTENDED	FE	6	3
IMMEDIATE	CE	3	3
INDEXED	EE	5+	2+

LDX

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	9E	5	2
EXTENDED	BE	6	3
IMMEDIATE	8E	3	3
INDEXED	AE	5+	2+

LDY

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	10 9 F - 9 E	6	3
EXTENDED	10 BE	7	4
IMMEDIATE	10 8E	4	4
INDEXED	10 AE	6+	3+

LEA **Load Effective Address**

SOURCE FORM: LEAX, LEAY, LEAS, LEAU

OPERATION: R' ← EA

CONDITION CODES:

H: Not affected

N: Not affected

Z: LEAX, LEAY: Set if all bits of the result are Clear.
LEAS, LEAU: Not affected

V: Not affected

C: Not affected

DESCRIPTION:

Form the effective address to data using the memory addressing mode. Load that address, not the data itself, into the pointer register.

LEAX and LEAY affect Z to allow use as counters and for 6800 INX/DEX compatibility. LEAU and LEAS do not affect Z to allow for cleaning up the stack while returning Z as a parameter to a calling routine, and for 6800 INS/DES compatibility.

REGISTER ADDRESSING MODE: Pointer (X, Y, S, or U)

LEAS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	32	4 +	2+

LEAU

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	33	4 +	2+

LEAX

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	30	4 +	2 +

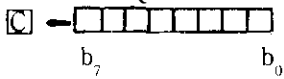
LEAY

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
RELATIVE	31	4 +	2 +

LSL **Logical Shift Left**

SOURCE FORM: LSL Q

OPERATION:



$C' \leftarrow b_7, b_7' \ldots b_1' \leftarrow b_6 \ldots b_0, b_0' \leftarrow 0$

CONDITION CODES:

H: Undefined

N: Set if bit 7 of the result are Clear

Z: Set if all bits of the results are Clear.

V: Loaded with the result of $(b_7 + b_6)$ of the original operand.

C: Loaded with bit 7 of the original operand.

DESCRIPTION:

Shifts all bits of ACCX or M one place to the left. Bit 0 is loaded with a zero. Bit 7 of ACCX or M is shifted into the carry flag. This is a duplicate assembly-language mnemonic for the single machine instruction ASL.

LSLA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	48	2	1

LSLB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	58	2	1

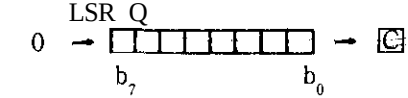
LSL

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	08	6	2
EXTENDED	78	7	3
INDEXED	68	6+	2+

LSR **Logical Shift Right**

SOURCE FORM: LSR Q

OPERATION:



$C' \leftarrow b_0, b_0' \ldots b_6' \leftarrow b_1 \ldots b_7, b_7' \leftarrow 0$

CONDITION CODES:

H: Not affected
N: Cleared
Z: Set if all bits of the result are Clear
V: Not affected
C: Loaded with bit 0 of the original operand

DESCRIPTION:

Performs a logical shift right on the operand. Shifts a zero into bit 7 and bit 0 into the carry flag. The 6800 processor also affects the V flag.

LSRA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	44	2	1

LSRB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	54	2	1

LSR

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	04	6	2
EXTENDED	74	7	3
INDEXED	64	6+	2+

MUL **Multiply Accumulators**

SOURCE FORM: MUL

OPERATION: ACCA':ACCB' ← ACCA x ACCB

CONDITION CODES:

H: Not affected
N: Not affected
Z: Set if all bits of the result are Clear
V: Not affected
C: Set if ACCB bit 7 of result is Set.

DESCRIPTION:

Multiply the unsigned binary numbers in the accumulators and place the result in both accumulators. Unsigned multiply allows multiple-precision operations. The Carry flag allows rounding the MS byte through the sequence MUL, ADCA #0.

MUL

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	3D	11	1

NEG **Negate**

SOURCE FORM: NEG Q

OPERATION: M' ← 0 - M (i.e., M' ← $\overline{M}$ + 1)

CONDITION CODES:

H: Undefined
N: Set if bit 7 of result is Set
Z: Set if all bits of result are Clear
V: Set if the original operand was 10000000
C: Set if the operation *did not* cause a carry from bit 7 in the ALU.

DESCRIPTION:

Replaces the operand with its two's complement. The C-flag represents a borrow and is set inverse to the resulting binary carry. Note that 80_{16} is replaced by itself and only in this case is V Set. The value 00_{16} is also replaced by itself, and only in this case is C cleared.

FLAG RESULTS:

(N ⊕ V) = 1 if 0 .LT. M (2's comp)
C = 1 if 0 .LO. M (unsigned)
Z = 1 if 0 .EQ. M

NEGA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	40	2	1

NEGB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	50	2	1

NEG

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	00	6	2
EXTENDED	70	7	3
INDEXED	60	6+	2+

NOP **No Operation**

SOURCE FORM: NOP

CONDITION CODES: Not affected

DESCRIPTION:

This is a single-byte instruction that causes only the program counter to be incremented. No other registers or memory contents are affected.

NOP

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	12	2	1

OR **Inclusive OR Memory into Register**

SOURCE FORMS: ORA P; ORB P

OPERATION: R' ← R v M

CONDITION CODES:

H: Not affected
N: Set if high order bit *of* result Set
Z: Set if all bits of result are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Performs an "Inclusive OR" operation between the contents of ACCX and the contents of M and the result **is** stored in ACCX.

REGISTER ADDRESS MODE: Accumulator

ORA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	9A	4	2
EXTENDED	BA	5	3
IMMEDIATE	8A	2	2
INDEXED	AA	4 +	2+

OREB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	DA	4	2
EXTENDED	FA	5	3
IMMEDIATE	CA	2	2
INDEXED	EA	4 +	2+

OR Inclusive OR Memory-Immediate into CCR

SOURCE FORM: ORCC #XX

OPERATION: R ← R v MI

CONDITION CODES: CCR' ← CCR v MI

DESCRIPTION:

Performs an "Inclusive OR" operation between the contents of CCR and the contents of MI, and the result is placed in CCR. This instruction may be used to Set interrupt masks (disable interrupts) or any other flag(s).

REGISTER ADDRESSNG MODE: CCR

MEMOR Y ADDRESSING MODÉ: Memory Immediate

ORCC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
IMMEDIATE	1A	3	2

PSHS Push Registers on the Hardware Stack

SOURCE FORM: PSHS register list

PSHS #Label | PC | U | Y | X | DP | B | A | CC |

OPERATION: push order ⟶

SP' ← SP - 1, (SP) ← PCL
if B7 of MI set, then: SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
if B6 of MI set, then; SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
if B5 of MI set, then: SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
if B4 of MI set, then: SP' ← SP - 1, (SP) ← IXH
if B3 of MI set, then: SP' ← SP - 1, (SP) ← DPR
if B2 of MI set, then: SP' ← SP - 1, (SP) ← ACCB
if Bl of MI set, then: SP' ← SP - 1, (SP) ← ACCA
if B0 of MI set, then: SP' ← SP - 1, (SP) ← CCR

CONDITION CODES: Not affected

DESCRIPTION:

Any, all, any subset or none of the MPU registers are pushed onto the hardware stack, (excepting only the hardware stack pointer itself).

MEMORY ADDRESSING MODE: Memory Immediate

PSHS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	34	5+	2

PSHU **Push Registers on the User Stack**

SOURCE FORM: PSHU register list

PSHU #LABEL | PC | S | Y | X | DP | B | A | CC |

push order ⟶

OPERATION:

if B7 of MI set, then: US' ← US - 1, (US) ← PCL
US' ← US - 1, (US) ← PCH
if B6 of MI set, then: US' ← US - 1, (US) ← SPL
US' ← US - 1, (US) ← SPH
if B5 of MI set, then: US' ← US - 1, (US) ← IYL
US' ← US - 1, (US) ← IYH
if B4 of MI set, then: US' ← US - 1, (US) ← IXL
US' ← US - 1, (US) ← IXH
if B3 of MI set, then: US' ← US - 1, (US) ← DPR
if B2 of MI set, then: US' ← US - 1, (US) ← ACCB
if Bl of MI set, then: US' ← US - 1, (US) ← ACCA
if B0 of MI set, then: US' ← US - 1, (US) ← CCR

CONDITION CODES: Not affected

DESCRIPTION:

Any, all, any subset or none of the MPU registers are pushed *onto the* user stack (excepting only the user stack pointer itself).

MEMORY ADDRESSING MODE: Memory Immediate

PSHU

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	36	5+4	2

PULS **Pull Registers from the Hardware Stack**

SOURCE FORM: PULS register list

PULS #LABEL | PC | U | Y | X | DP | B | A | CC |

OPERATION: ⟵ pull order

if B0 of MI set, then: CCR' ← (SP), SP' ← SP + 1
if Bl of MI set, then: ACCA' ← (SP), SP' ← SP + 1
if B2 of MI set, then: ACCB' ← (SP), SP' ← SP + 1
if B4 of MI set, then: IXH' ← (SP), SP' ← SP + 1
IXL' ← (SP), SP' ← SP + 1
if B5 of MI set, then: IYH' ← (SP), SP' ← SP + 1
IYL' ← (SP), SP' ← SP + 1
if B6 of MI set, then: USH' ← (SP), SP' ← SP + 1
USL' ← (SP), SP' ← SP + 1
if B7 of MI set, then: PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

CONDITION CODES:

May be pulled from stack, otherwise *unaffected*

DESCRIPTION:

Any, all, any subset or none of the MPU registers are pulled from the hardware stack, (excepting only the hardware stack pointer itself). A single register may be "PULLED" *with condition-flags set* by loading auto-increment from stack (EX:LDA, S+).

MEMORY ADDRESSING MODE: Memory Immediate

PULS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	35	5+	*2*

PULU **Pull Registers from the User Stack**

SOURCE FORM: PULU register list

PULU #LABEL | PC | S | Y | X | DP | B | A | CC |

OPERATION: ← pull order

if B0 of MI set, then: CCR' ← (US), US' ← US + 1
if Bl of MI set, then: ACCA' ← (US), US' ← US + 1
if B2 of MI set, then: ACCB' ← (US), US' ← US + 1
if B3 of MI set, then: DPR' ← (US), US' ← US + 1
if B4 of MI set, then: IXH' ← (US), US' ← US + 1
IXL' ← (US), US' ← US + 1
if B5 of MI set, then: IYH' ← (US), US' ← US + 1
IYL' ← (US), US' ← US + 1
if B6 of MI set, then: SPH' ← (US), US' ← US + 1
SPL' ← (US), US' ← US + 1
if B7 of MI set, then: PCH' ← (US), US ← US + 1
PCL' ← (US), US ← US + 1

CONDITION CODES:

May be pulled from stack, otherwise unaffected.

DESCRIPTION:

Any all, any subset or none of the MPU registers are pulled from the user stack (excepting only the user stack pointer itself). A single register may be "PULLED" *with condition-flags set* by doing an auto-increment load from the stack (EX:LDX, U++))

MEMORY ADDRESSING MODE: Memory Immediate

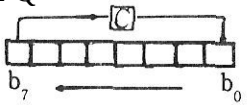
PULU

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF RYTES
INHERENT	37	5+	2

ROL **Rotate Left**

SOURCE FORM: ROL Q

OPERATION:



$C' \leftarrow b_7, b_7' \ldots b_1 \leftarrow b_6\ b_0,\ b_0' \leftarrow C$

CONDITION CODES:

H: Not affected
N: Set if bit 7 of the result is Set
Z: Set if all bits of the result are Clear
V: Loaded with the result of (b_7 b_8) of the original operand.
C: Loaded with bit 7 of the original operand

DESCRIPTION:

Rotate all bits of the operand one place left through the carry flag; this is a 9-bit rotation.

ROLA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	49	2	1

ROLB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	59	2	1

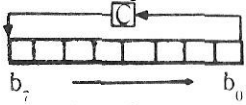
ROL

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	09	6	2
EXTENDED	79	7	3
INDEXED	69	6+	2+

ROR **Rotate Right**

SOURCE FORM: ROR Q

OPERATION:



$C' \leftarrow b_0,\ b_6' \ldots b_0 \leftarrow b_7 \ldots b_1,\ b_7' \leftarrow C$

CONDITION CODES:

H: Not affected
N: Set if bit 7 of result is Set
Z: Set if all bits of result are Clear
V: Not *affected*
C: Loaded with bit 0 of the previous operand

DESCRIPTION:

Rotates all bits of the operand right one place through the carry flag; this is a nine-bit rotation. The 6800 processor also affects the V flag.

RORA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	46	2	1

RORB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	56	2	1

ROR

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	06	6	2
EXTENDED	76	7	3
INDEXED	66	6 +	2+

RTI **Return from Interrupt**

SOURCE FORM: RTI

OPERATION: CCR' (SP), SP' ← SP + 1

if CCR bit E is SET then:

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

if CCR bit E is CLEAR then:

PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

CONDITION CODES: Recovered from stack

DESCRIPTION:

The saved machine state is recovered from the hardware stack and control is returned to the interrupted program. If the recovered E 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 to be recovered.

SBCB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	D2	4	2
EXTENDED	F2	5	3
IMMEDIATE	C2	2	2
INDEXED	E2	4 +	2 +

SEX **Sign Extended**

SOURCE FORM: SEX

OPERATION: If bit 7 of ACCB is set then ACCA' ← FF_{16}
else ACCA' ← 00_{16}

CONDITION CODES:

H: Not affected
N: Set if the MSB of the result is Set
Z: Set if all bits of ACCD are Clear
V: Not affected
C: Not affected

DESCRIPTION:

This instruction transforms a two's complement 8-bit value in ACCB into a two's complement 16-bit value in the double accumulator.

SEX

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	1D	2	1

ST **Store Register Into Memory—8 Bits**

SOURCE FORM: STA P; STB P

OPERATION: M' ← R

CONDITION CODES:

H: Not affected
N: Set if bit 7 of stored data was Set
Z: Set if all bits of stored data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Writes the contents of an MPU register into a memory location.

REGISTER ADDRESSING MODES: Accumulator

RTI

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	3B	6/15	1

RTS **Return from Subroutine**

SOURCE FORM: RTS

OPERATION: PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← 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.

RTS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	39	5	1

SBC **Subtract with Borrow**

SOURCE FORMS: SBCA P; SBCB P

OPERATION: R' ← R - M - C (i.e., R' ← R+ $\overline{M}$ + $\overline{C}$)

CONDITION CODES:

H: Undefined
N: Set if bit 7 of the result is Set
Z: Set if all bits of the result are Clear
V: Set if the operation causes an 8-bit two's complement overflow
C: Set if the operation *did not* cause a carry from bit 7 in the ALU

DESCRIPTION:

Subtracts the contents of M and the borrow (in the carry flag) from the contents of an 8-bit register, and places the result in that register. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING MODE: Accumulator

SBCA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	92	4	2
EXTENDED	B2	5	3
IMMEDIATE	82	2	2
INDEXED	A2	4 +	2+

STA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	97	4	2
EXTENDED	B7	5	3
INDEXED	A7	4 +	2+

STB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	D7	4	2
EXTENDED	F7	5	3
INDEXED	E7	4 +	2+

ST **Store Register Into Memory—16-Bi**

SOURCE FORM: STD P; STX P; STY P; STS P; STU P

OPERATION: M':M+1' ← R

CONDITION CODES:

H: Not affected
N: Set if bit 15 of stored data was Set
Z: Set if all bits of stored data are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Write the 16 bit register into consecutive memory locations

REGISTER ADDRESSING MODES: Double Accumulator
Pointer (X, Y, S, or U)

STD

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	DD	5	2
EXTENDED	FD	6	3
INDEXED	ED	5+	2+

STS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	10 DF	6	3
EXTENDED	10 FF	7	4
INDEXED	10 EF	6+	3 +

STU

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	DF	5	2
EXTENDED	FF	6	3
INDEXED	EF	5+	2+

STX

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	9F	5	2
EXTENDED	BF	6	3
INDEXED	AF	5 +	2+

STY

ADDRESSING MODE	OPCODE	MPU CYCYLES	NO OF BYTES
DIRECT	10 9F	6	3
EXTENDED	10 DF	7	4
INDEXED	10 AF	6+	3 +

SUB ***Subtract Memory from*** **Register—8 bit**

SOURCE FORMS: SUBA P; SUBB P

OPERATION: R' ← R- M; M + 1

CONDITION CODES:

H: Undefined

N: Set if but 7 of the result is Set

Z: Set if all bits of the result are Clear

V: Set if the operation caused an 8-bit two's complement overflow

C: Set if the operation *did not* cause a carry from bit 7 in the ALU

DESCRIPTION:

Subtracts the value in M from the contents of an 8-bit register. The C flag represents a borrow and is set inverse to the resulting carry.

REGISTER ADDRESSING MODE: Accumulator

FLAG RESULTS:

(N ⊕ V) = 1 if R LT. M (two's comp)

C = 1 if R .LO. M (unsigned)

Z = l if R .EQ. M

SUBA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	90	4	2
EXTENDED	BC	5	3
IMMEDIATE	80	2	2
INDEXED	A0	4 +	2 +

SUBB

ADDRESSING MODE	OPCODE	MPU CYCLES	NOOF BYTES
DIRECT	DO	4	2
EXTENDED	F0	5	3
IMMEDIATE	CO	2	2
INDEXED	EO	4 +	2+

SUB Subtract Memory from Register—16-Bit

SOURCE FORM: SUBD P

OPERATION: R' ← R - M:M+1 [i.e., R; ← R + $\overline{M:M+1}$ + l]

CONDITION CODES:

H: Unaffected

N: Set if bit 15 of result is Set

Z: Set if all bits of result are Clear

V: Set if the operation caused a 16-bit two's complement overflow.

C: Set if the operation on the MS byte *did not* cause a carry from bit 7 in the ALU

DESCRIPTION:

This information subtracts the value in M:M+1 from the 16-bit accumulator. The C flag represents a borrow and is set inverse to the resulting binary carry.

REGISTER ADDRESSING MODE: Double Accumulator

SUBTRACT SETS:

(N ⊕ V) = 1 if R .LT. M (two's comp)

C = 1 if R .LO. M (unsigned)

Z = 1 if R .EQ. M

SUBD

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	93	6	2
EXTENDED	B3	7	3
IMMEDIATE	83	4	3
INDEXED	A3	6+	2+

SWI **Software Interrupt**

SOURCE FORM: *SWI*

OPERATION: Set E (entire state will be 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
Set I, F (mask interrupts)
PC (FFFA):(FFFB)

CONDITION CODES: Not affected

DESCRIPTION:

All of the MPU registers are pushed onto the hardware stack (excepting only the hardware stack pointer itself), and control is transferred through the SWI vector.

SWI SETS I AND F BITS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	3F	19	1

SWI2 **Software Interrupt 2**

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 MPU registers are pushed onto the hardware stack (excepting only the hardware stack pointer itself), and control is transferred through the SWI2 vector. SWI2 is available to the end user and must not be used in packaged software.

SWI2 DOES NOT AFFECT I AND F BITS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	10 3F	20	2

SWI3 **Software Interrupt**

SOURCE FORM: *SWI3*
OPERATION: Set E (entire state will be 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 (FFF2):(FFF3)

CONDITION CODES: Not Affected
DESCRIPTION:

All of the MPU registers are pushed onto the hardware stack (excepting only the hardware stack pointer itself), and control is transferred through the SWI3 vector.

SWI3 DOES NOT AFFECT I AND F BITS

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	11 3F	20	2

SYNC **Synchronize to External Event**

SOURCE FORM: SYNC

OPERATION: Stop processing instructions

CONDITION CODES: Unaffected

DESCRIPTION:

When a SYNC instruction is executed, the MPU enters a SYNCING state, stops processing instructions and waits on an interrupt. When an interrupt occurs, the SYNCING state is cleared and processing continues. IF the interrupt is enabled, and the interrupt lasts 3 cycles or more, the processor will perform the interrupt routine. If the interrupt is masked or is shorter than 3 cycles long, the processor simply continues to the next instruction (without stacking registers). While SYNCING, the address and data buses are tri-state.

COMMENTS:

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

```
                FOR 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 SYNCING state is cleared by any interrupt, and any enabled interrupt will probably destroy the transfer (this may be used to provide MPU response to an emergency condition).

The same connection used for interrupt-driven I/O service may thus be used for high-speed data transfers by setting the interrupt mask and using SYNC.

SYNC

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	13	> = 2	1

TFR Transfer Register *to* Register

SOURCE FORM: TFR R_1, R_2

OPERATION: $R_2 \leftarrow R_1$

CONDITION CODES: Not affected (Unless R_2 = CCR)

DESCRIPTION:

Bits 7-4 of the immediate byte of the instruction 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

Registers may only be transferred between registers of like size; i.e., 8-bit to 8-bit, and 16 to 16.

TFR R1.R2

ADDRESSING MODE	*OPCODE*	MPU CYCLES	NO OF BYTES
INHERENT	1F	7	2

TST **Test**

SOURCE FORM: TST Q

OPERATION: TEMP $\leftarrow$ M - 0

CONDITION CODES:

H: Not affected
N: Set if bit 7 of the result is Set
Z: Set if all bits of the result are Clear
V: Cleared
C: Not affected

DESCRIPTION:

Set condition code flags N and Z according to the contents of M, and clear the V flag. The 6800 processor clears the C flag.

COMMENTS:

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.

TSTA

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	4D	2	1

TSTB

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
INHERENT	5D	2	1

TST

ADDRESSING MODE	OPCODE	MPU CYCLES	NO OF BYTES
DIRECT	OD	6	2
EXTENDED	7D	7	3
INDEXED	6D	6+	2+

HARDWARE INSTRUCTION FIRQ Fast Interrupt Request

OPERATION: if F bit CLEAR, then:
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
Clear E (subset state is saved)
SP' ← SP - 1, (SP) ← CCR
Set if, I (mask further interrupts)
PC' ← (FFF6):(FFF7)

CONDITION CODES: Not affected

DESCRIPTION:

A low level on the FIRQ input with the F 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 FIRQ vector. An RTI returns to the original task. It is possible to enter an FIRQ handler with the entire state saved if the FIRQ occurs after CWAI.

ADDRESSING MODE: Absolute Indirect

COMMENTS:

An IRQ interrupt, having lower priority than the FIRQ, is prevented from interrupting the FIRQ handling routine by automatic setting of the I flag. This mask bit could then

be reset if priority was not desired, The IRQ allows operations on memory, TST, INC, DEC, etc., without the overhead of saving the entire machine state on the stack.

HARDWARE INSTRUCTION IRQ Interrupt Request

OPERATION: IFF I bit CLEAR, then:

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
Set E (entire state saved)
SP' ← SP - 1, (SP) ← CCR
Set I (mask further IRQ interrupts)
PC' ← (FFF8):(FFF9)

CONDITION CODES: Not affected

DESCRIPTION:

If the IRQ mask bit I is clear, a low level on the IRQ input causes this interrupt sequence to occur at the end of the current instruction. Control is returned to the interrupted program via an RTI. An FIRQ may interrupt an IRQ handling routine and be recognized anytime after the IRQ vector is taken.

ADDRESSING MODE: Absolute Indirect

HARDWARE INSTRUCTION NMI Non-Maskable Interrupt

OPERATION:

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
Set E (entire state save)
SP' ← SP - 1, (SP) ← CCR
Set I, F (mask interrupts)
PC' ← (FFFC):(FFFD)

CONDITION CODES: Not affected

DESCRIPTION:

A negative edge on the NMI input causes all of the MPU registers (except the hardware stack pointer SP) 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. The NMI operation is internally blocked by RESET, any NMI-edge will be latched, and the operation will occur after the first load into the stack pointer (LDS; TFR r,s; EXG r,s; etc.).

ADDRESSING MODE: Absolute Indirect

HARDWARE INSTRUCTION **RESTART**

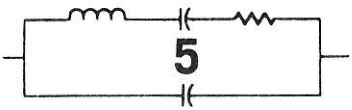
OPERATION:
CCR' ← X1X1XXXX
DPR' ← 00_{16}
PC ← (FFFE):(FFFF)

CONDITION CODES: Not affected

DESCRIPTION:

The MPU is initialized (required after power-on) to start program execution.

ADDRESSING MODE: Absolute Indirect



MEK6809EA Assembler

Motorola has not only developed the 6809D4 evaluation unit, but is supporting it with a variety of software. One piece of software available from Motorola is the MEK6809EA *assembler*. This is a specialized program that is designed to process source programs written in M6809 assembly language. This "source" is then translated into object programs that the firmware loaders on the D4 evaluation unit can understand.

The previous chapter gave you definitions to the various instruction codes that the 6809 understands and, in some cases, examples of an assembly code. I will attempt in this chapter to give you only the basics of the assembler. Should you desire to learn more, the supporting software and D4 unit should be purchased from Motorola, or the Radio Shack TRS-80 Color Computer or Videotex should be bought.

BASICS OF THE ASSEMBLER

The assembler, as stated before, performs operations on source code that contains specific operations which determine what will happen when the program executes as an object, or run-time, program. Some of these internals of the source file are operations such as instruction codes, or assembler directives, and labels—sometimes called symbolic names, special operators and special symbols. Directives, which are part of the assembler's operation, are special codes that are entered into the source file to tell the assembler to perform a specific operation.

Essentially, the role of the assembler is to translate source programs into object code in a format required by the systems' loader. As you will see in Chapter 6, this will be for the D4. The assembler is also used for archival purposes. Debugging the assembler provides a listing which contains all the information about the program in logical fashion.

TYPICAL REQUIREMENTS

The assembler takes information in source form and translates it into object form. To do this, however, certain rules are usually followed.

First, the source form of the program is a sequence of statements written in ASCII characters following conventions that the specific assembler requires. Each input source line is terminated with a carriage return. The source form usually consists of five fields:

- Sequence number. This is not always required, but is useful especially in the editing function.
- Label or an asterik (*) implying a comment.
- Operation.
- Operand.
- Optional comment.

Sequence Number

This is an option for programmer convenience for the Motorola assembler. A sequence number can consist of up to five decimal digits but less than 65,536. When used, the sequence number must be followed by a space.

Label Field

This field is right after the sequence number, or it can appear as the first field. When an asterisk (*) is used, the line is considered by the assembler to be comment and is thus ignored. A blank indicates that the field is empty and the line contains no *label.*

The symbol is a special form of a label and has the following attributes:

- Usually consists of 1 to 6 characters.
- Only the following are considered valid symbol characters: A through Z, 0 through 9, "."—period, and a dollar sign "$".
- A symbol must consist of either a period ".", or an alphanumeric character as the first character.

• Certain symbols: A, B, D, X, Y, U, S, CC, PC, PCR and DP are reserved symbols used by the assembler and are never used in the label field.

When a symbol is used, it may occur only once in the label field. If it occurs in more than one label field, a reference to that symbol will cause an error, since the assembler will have no idea to what you are referring.

A typical label can be used in an equate statement, that unique statement that sets a label equal to a specific value. Some examples of labels are:

```
INCH    EQU $FC00
.G1     LDA   #$41
```

Operating Field

The *operation field* occurs directly after the label field in an assembly language source statement. This field consists of an operation code of three or four characters. Entries in the operation code field may be one of two types. Machine mnemonic operation code entries correspond directly to M6809 machine instructions. This operation code field includes the "A" or "B" character for the "dual" or "accumulator" addressing modes. Directives are special operation codes known to the assembler which control the assembly process rather than being translated directly to machine language.

The assembler searches for operation codes in the table of machine operation codes and directives. If not found, an error message is printed.

Operand Field

Interpretation of the *operand field* is dependent on the operation field. For the M6809 machine instructions, the operand field must specify the addressing mode. The operand field formats and the corresponding addressing modes are in Table 5-1.

Comment Field

The last field of an M6809 Assembly Language source line is the *comment field.* This field is optional and is ignored by the assembler except for being included in the listing. The comment field is separated from the operand field (or the operator field if there is no operand) by one or more blanks and may consist of any

Table 5-1. Operand Field Formats and Corresponding Addressing Modes.

Operand Format	M6809 Machine Instruction Addressing Mode
no operand	inherent and accumulator
expression	direct or extended (direct will be used if possible)
# (expression)	immediate
(expression), R	indexed (where "R" is an indexable register)

ASCII character. This field is important in documenting the operation of a program.

EXPRESSIONS

An *expression* is a combination of symbols and/or numbers separated by one of the arithmetic operators (+ ,-, *, or /). The assembler evaluates expressions algebraically from left to right without parenthetical grouping. There is no precedence hierarchy among the arithmetic operators. A fractional result, or intermediate result obtained during the evaluation of an expression, will be truncated to an integer value.

Constants

Decimal: < number >
Hexadecimal: $ < number > or < number > H (first digit in latter case must be 0 - 9)
Octal: @ < number > or < number > 0 or < number > Q
Binary: % < number > or < number > B

ASCII Literals

' <*character*>*:* apostrophe followed by an ASCII character, except carriage return. The result is the numeric value for the ASCII character.

SYMBOLS

A *symbol* in an expression is similar to a symbol in the label field except that the value of the symbol is referenced instead of defined. An asterisk "*" is a special symbol recognized by the

Table 5-2. Assembly Listing.

```
Source Line  Current location  Machine operation core  Operand  Cycle Time  Label   Operation  Operand      Comment

0001                                                                     NAM    ENDST
0002                                                                     OPT    LLEN=80
0004         2000                                                        ORG
00005        2000  10CE        E0000    4             START             $2000
00006        2004  10AE        E4       6                    LDY       +$F0000
00007        2007  10AF        84       6                    STY       S
00008        200A  32          16       5                    LEAS      X
00010                          2000                          END       -10,X
TOTAL        ERRORS           00000
TOTAL        WARNINGS         00000

PAGE         001   ENDST1 *PROGRAM NOT INDICATING EXECUTION START*


00001                                                                    NAM    ENDST1
00002                                                                    OPT    LLEN
00004        2000                                                        ORG    140080
00005        2000  10CE        E000O                                     $2000
00006        2004  10AE        E4       6                    LDY       +$E000
00007        2007  10AF        84       6                    STY       S
00008        200A  32          16       5                    LEAS      X
00010                          0000                          END       -10,X
TOTAL        ERRORS           0000
TOTAL        WARNINGS         00000
```

assembler and represents the value of the current location counter (first byte of an instruction), when used in the context of the symbol.

A 16-bit integer value is associated with each symbol. This value is used in place of the symbol during expression evaluation.

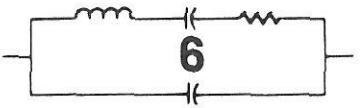
Table 5-3. Standard Format for Assembly Listings.

Column	Contents
1-5	Source line 1 - 5 digit decimal counter kept by assembler.
7-10	Current Location Counter value (in hex).
12-15	Machine Operation Code (hex).
17-23	Operand Machine Code (if any) (hex).
25-26	Cycle Count of Execution Time (decimal).
28-33	Label Field.
35-39	Operation Field.
41-48	Operand Field (longer operand extends into comment field).
50-Last Column	Comment Field.

The MEK6809EA assembler is a two-pass assembler. The symbol table is built on the first pass. Object records and listing are produced on the second pass. Certain expressions cannot be fully evaluated during the first pass because they may contain (forward) references to symbols which have not yet been defined. In some cases, a symbol may not be defined before being used in the second pass. Since the assembler cannot evaluate such symbols, these cases are treated as errors. Only one level of forward referencing is allowed.

ASSEMBLER LISTING

Assembler outputs include an assembly listing and an object program (Table 5-2). The assembly listing includes the source program as well as additional information generated by the assembler. Most lines in the listing correspond directly to a source statement. Lines which do not correspond directly to a source line include: page header line, error lines and expansion lines for the FCC, FDB and FCB directives. Most listing lines follow the standard format shown in Table 5-3.



Implementation of VTL-09

The implementation of BASIC that is to be presented in the next few pages is based on a design created by Gary Shannon and Frank McCoy for the MITs 680b microcomputer. Their implementation was named VTL-2 for Very Tiny Language, and permitted users of the 680b to have high-level language capability with only 1K bytes of working memory space. VTL-2, originally implemented, required only 768 bytes of memory and was written in such a manner to be ROMable.

As you proceed through this chapter, you will realize that the small BASIC-like language we are talking about is quite powerful. In *fact, it most* likely is still the most powerful small interpreter available today. The original copyright was 1977 for the 6800 version of VTL-2 and is still held by the Computer Store of Santa Monica, California. The following pages are designed to assist you in how to use what I call VTL-09.

DIRECT AND PROGRAM STATEMENTS

The statements that may be entered as input to the VTL-09 interpreter are of two types: the direct statement, which has no line number and is executed immediately after being entered; and the program statement, which requires line numbers used to build a program. Program statements are not executed until the program is run as opposed to the immediate execution of direct statements.

The design of VTL-09 is simple, making it ideal for the beginner and powerful enough for advanced purposes. An important feature, not found with other versions of VTL, is the inclusion of calls to permit loading and dumping of programs to tape. The implementation that is presented in this book is strictly for the Motorola 6809D4 unit which is designed to load at $2000 hex.

PRELIMINARY CONCEPTS

Line numbers must precede each program statement. The statements following the line number must be separated from the number by at least one space. As designed, each line must end with a carriage return and be less than 73 characters in length.

Typically, line numbers are incremented in steps of 10. This permits the addition of other statements if necessary. No line renumbering utility is included, so care must be taken when first beginning the program process.

Variables may be represented by any single alphabetic or special character such as !"#$%&'()=-+*:;?/.><[]. Most of these are available for the user to define as he wishes. A few of the variable names, however, have been set aside for special purposes. These so-called *system variables* will be discussed in detail later.

The value assigned to a variable may be either a numeric value in the range 0-65535, or a single ASCII character, including control characters. Numeric and string values may be freely interchanged, in which case the characters are equivalent to the decimal value of their ASCII code representation. Thus, it becomes possible to add 1 to the letter A, giving as a result the letter B.

ARITHMETIC OPERATIONS

The arithmetic operations permitted for use in expressions are:

- + addition
- - subtraction
- * multiplication
- / division
- = test for equality
- > test for greater than or equal to
- < test for less than

The test operations—equal to, greater than or equal to and less than—all return a value of zero if the test fails and a value of one if the test is successful.

Expressions in VTL-09 may contain any number of variables or numeric values—*literals*—connected by any of the above operations. Parentheses may be used to alter the order of execution of the operations. If no parentheses are included, the operations proceed in strictly right to left order.

The value resulting from the expression must be assigned *to* some variable name. This is done with the equal sign. Note that the symbol has two meanings depending on where it occurs in the expression. The expression "A=B=C" means test b and c for equality. If they are equal put a one in A; if they are unequal, put a zero in A. Some of the examples of valid arithmetic expressions would be:

Y=A*(X*X)+B*X+C with left to right execution. This is equivalent to Y=(A*X*X+B)*X+C

Y=(A*X*X)+(B*X)+C which is equivalent o AX^2+BX+C

Notice how the absence of parentheses around the quantity B*X in the first expression has completely altered its meaning. Keep the right to left order in mind, and when in doubt use parentheses to control the order of evaluation.

SYSTEM VARIABLES

In order to conserve space and to provide a more consistent syntax, VTL-09, like VTL-2 uses system variables to accomplish functions usually done with special key words in other languages. This convention is probably the single most important reason for its tiny size. These special variables are used for such functions as the BASIC PRINT, GOTO, GOSUB, RETURN, IF AND RANDOM functions.

Pound Sign

The system variable number or *pound sign* (#) represents the line number of the line being executed. Until the statement has been completed, it will contain the current line number. For example, the statement 100 A=# is equivalent to simply writing 100 A=100. After completion *of* a line, this variable will contain the number of the next line to be executed. If nothing is done to the variable, this will be the next line in the program text. If a state-

merit changes #, however, the next line executed will be the line with the number that matches the value of #. Thus, the variable # may be used to transfer control to a different part of the program. This then becomes the VTL-09 equivalent to the BASIC GOTO: #=300 means GOTO 300.

If the # variable should ever be set to zero by some statement, this value will be ignored. The program will proceed as if no change had taken place. This fact allows us to write IF statements in VTL-09. Consider the following example:

```
10 X=1                  Set X equal to 1
20 #=(X=25)*50          If X=25 goto 50
30 X=X+1                add 1 to x
40 #=20                 goto 20
50 . . . .              and so on
```

Notice that the quantity (X=25) will have the value one, if it is true that X is equal to 25, and the value zero if it is false. When this logical value is multiplied times 50, the result will either be zero or 50. If it is 50, the statement causes a goto 50 to occur. If the statement is zero, then a goto 0 occurs, which is a dummy operation causing the next statement to be executed.

Exclamation Point

Taking advantage of left-to-right evaluation, two bytes of memory could be saved by writing 20 #=x=25*50. Each time the value of # is changed by a program statement, the old value +1 is saved in the system variable—*exclamation point* (!). In other words, after executing a goto, the line number of the line that follows the goto is saved so that a subroutine will know where to return to when finished. Thus, the # variable is used for both goto and gosub operations.

```
10X=1
20 #=100
30 30 X=2
40 #=100
50 X=3
60 #=100
.....
100 x=x*x
110 #=!          goto where you came from
```

In this example, control proceeds from line 20 to line 100. After that, line 110 causes control to return to line 30. When line 40 is executed, the subroutine at 100 will return to line 50.

The actual value stored in the ! variable is old line number +1. If VTL doesn't find the exact line number, it will GO to the next higher line number.

Question Mark

The system variable *question mark* (?) represents the user's terminal. It can be either an input or an output, depending on which side of the equal sign it appears.

The statement ?=A' is interpreted as PRINT A, and the statement X=? is interpreted as INPUT X. A ? can be included anywhere within an expression.

```
10 ?="ENTER THREE VALUES"
20 A=(?+?+?)/3
30 ?="THE AVERAGE IS"
40 ?=A
```

This program will request three inputs while executing line 20.

When typing in a reply to a request for input, the user may enter any one of three different types of data: a decimal number, a variable name or any valid VTL-09 expression. Thus, for example, the user may reply with such things as "1004" or A+B*(9/X). In each case the expression is completely evaluated before the result is passed to the input statement. The only exception is that you are not allowed to respond with another question mark as this will mess up the line pointer in the interpreter, causing it to return an improper value.

When the question mark is on the left side of the first equal sign, it represents a print statement; on the right it is an input. The formatting of printing output can be controlled by either the inclusion or *omission of* leading or trailing blanks, thus giving a similar operation and PRINT USING.

Percent and Apostrophe

The system variable *percent* (%) contains the value of the remainder of the last division operation. This value will remain the same until the next division takes place.

The system variable *apostrophe* (') represents a random number. This number will have an unpredictable value in the range 0-65535. If called twice on the same line, the same value will be

returned both times. The value of the variable is scrambled each time any statement is executed. Therefore, for best results it is highly recommended that at least one computation be performed before calling for the random value again.

Dollar Sign

In addition to decimal numeric input and output, the system variable *dollar sign* ($) is used to input and output single characters. As with the question mark variable, A=$ means input a single ASCII character and place its numeric value in A. Similarly, $=X means PRINT the single ASCII character whose value is stored in X.

```
10 A=65
20 $=A
30 A=A+1
40 #=A<91*20
50 ?=" "
```

This example will print out as one continuous string all the letters of the alphabet. If you wish to find out what decimal values correspond to which characters, these can be found in any conversion chart. Simply compute by typing the direct statement ?=$ and then entering the character whose decimal value is to be found.

Asterisk

The system variable *asterisk* (*) represents the memory size of your computer. For a system with 1K, this would be 1024. Entering ?=* will give the amount of memory.

The system will accept a different definition to the amount of memory. This can be entered by typing *=1024*17, for example, for a 17K system that reserves 1K for user space.

Ampersand

The system variable *ampersand* (&) represents the next available byte of memory in the program buffer. When first initilized, VTL-09 must be set to 264. Enter &=264 to set the buffer to first byte. You will be able to find out how much remaining memory you have after entering programs by typing ?=*-&.

Greater Than

The system *variable greater than* > is used to pass a value to a machine language subroutine. When encountered on the left side of

the equal sign the expression is evaluated, the value is placed as a 16-bit integer in the A and B registers, and a software interrupt is generated. The value stored in > is pulled off the stack by the RTI instruction. If you wish to change the value placed into the variable you should first pull the condition of the stack. Then reset the registers. See Chapters 3 and 4.

Cassette In (CI) and Cassette Out (CO)

Two very special variables used by the VTL-09 are CI for cassette in, which permits loading of programs from the tape cassette; and CO for cassette out, which permits the saving of programs. Programs and data can be saved using these commands. For programs, they are entered in the direct mode, or they can be embedded in a program. For example, to load data from a program, the program must first spec space for the data using the &=xxxx. Then CI is invoked in concert with the ?.

```
10 &=2492      Some value that will allow sufficient
                space for the data.
 20?=CI        Load the data.
```

Notice no names are permitted—only very fundamental loading and saving.

SAMPLE PROGRAMS

As you can see, VTL-09 is easy as pie with no big surprises built in. The purpose of it is to show you how easy it is to program a 6809 with a very useful application. The next several pages serve as a summary to this chapter, on how to use VTL, and a roundup to put this book in proper perspective (Table 6-1).

Relocatable Program

```
64000 A=#
64010 B=&
64020 C=#
64030 &=B
64040 ?="STARTING#?";
64050 D=?
64060 ?="STEP SIZE? ";
64070 E=?
64080 &=1
64090 G=131
64100  J=0
64110  H=#+1
64120  G=&+1/2+G
64130  &=%
64140  #=:G)>A*5*(C-A)+#
64150  #=D->(A-1)+G>D)>1*C
64160  :G)=D
64170  &=&+1
64180  J=D
64190  D=D+E
```

```
64200  K=#+1                64230  #=H
64210  &=&+1                64240  &=B
64220  #=:G)>256*K         64250  ?="DONE"
```

This program is relocatable. It can be renumbered and will still run. However, the step size between program steps must remain constant or line 64140 will not work right. Also, the largest number of the program to be renumbered must be less than the first number of the renumber program.

Table 6-1. List of VTL-09 Features.

VARIABLE	
A-Z	COMMON VARIABLES
	USE FREELY FOR STORING VALUES
SYSTEM VARIABLES	
!	RETURN ADDRESS
	POINTS TO THE LINE # AFTER THE LAST #= STATEMENT
..	POINTER FOR LITERAL PRINT STATEMENTS
#	LINE NUMBER
$	SINGLE CHARACTER STRING (INPUT OR OUTPUT)
%	REMAINDER AFTER THE LAST DIVIDE OPERATION
&	POINTS TO THE LAST BYTE OF PROGRAM
'	RANDOM NUMBER
(	SETS START OF PARENTHESIZED EXPRESSION
)	END
	SETS END OF LINE
	SETS END OF PARENTHESIZED EXPRESSION
	SETS END OF ARRAY DESCRIPTION
*	USED ALSO FOR REMARK STATEMENT
	POINTS TO END OF MEMORY
>	MACHINE LANGUAGE SUBROUTINE
?	PRINT STATEMENT WHEN ON LEFT OF EQUAL SIGN
	INPUT STATEMENT WHEN ON RIGHT OF EQUAL SIGN
:	DEFINES START OF ARRAY DESCRIPTION
;	WHEN FOLLOWING A LITERAL PRINT STATEMENT,
	SAYS DO NOT PRINT CARRIAGE-RETURN LINE-FEED
. _ = : +	MAY BE USED FREELY AS STANDARD VARIABLES
, (/ ↑] [	BUT USE IS NOT RECOMMENDED FOR LEGIBILITY REASONS
	OPERATORS
+	ADD TO PREVIOUS VALUE
-	SUBTRACT FROM PREVIOUS VALUE
*	MULTIPLY TIME PREVIOUS VALUUE
/	DIVIDE PREVIOUS VALUE BY
=	IS PREVIOUS VALUE EQUAL TO (YES = 1, NO = 0)
<	IS PREVIOUS VALUE LESS THAN (YES = 1, NO = 0)
>	IS PREVIOUS VALUE EQUAL TO OR GREATER THAN (Y=1, N=0)

THE DEFAULT OPERATOR IS THE LESS THAN TEST.

Hurkle Program

```
100 ?=""
110 ?="A HURKLE IS HIDING ON A"
120 ?="10 BY 10 GRID. HOMEBASE"
130 ?="ON THE GRID IS POINT 00"
140 ?="AND A GRIDPOINT IS ANY"
150 ?="PAIR OF WHOLE NUMBERS"
160 ?="TRY TO GUESS THE HURKLE'S"
170 ?="GRIDPOINT. YOU GET 5 GUESSES'
180 ?=""
190 R=7100*0+%
200 A=R/10
210 8=%
220 K=1
230 ?="GUESS#";
240 ?=K
250 ?=" ?";
260 X=?/10
270 Y=%
280 ?=""
290 #=X*10+Y=R*540
300 K=K+1
310 #=K=6*440
320 ?="GO ";
330 #=Y=B*370+(4<B*360)
340 ?="SOUTH";
350 #=370
360 ?="NORTH";
370 #=X-A*410+(X<A*400)
380 ?="WEST";
390 #=410
400 ?="EAST";
410 ?=""
420 ?=""
430 #=230
440 ?=""
450 ?="SORRY THAT'S 5 GUESSES"
460 ?="THE HURKLE IS AT ";
470 ?=A
480 ?=B
490 ?=""
500 ?=""
510 ?="LETS PLAY AGAIN."
520 ?="HURKLE IS HIDING"
530 #=180
540 ?="YOU FOUND HIM IN ";
550 ?=K
560 ?=" GUESSES"
570 #=490
```

Time of Day Digital Clock Programs

FOR 300 BAUD TERMINALS

```
10  ?="HOUR ?";
20  H=?
30  ?="MINUTE ?";
40  M=?
50  ?="SECOND ?";
60  S=?
70  ?="READY"
80  A=$
90  S=S+1
100  M=S/60+M
110  S=%
120  H=M/60+H
130  M=%
140  H=H/24*0+%
150  ?="TIME: ";
160  ?=H/10
170  ?=%
180  ?=": ";
190  ?=M/10
200  ?=%
210  ?=": "?=": ";
220  ?=S/10
230  ?=%
240  $=13
250  A=B
260  T=31
270  T=T-1
280  #=T=0*90
290  #=270
```

FOR 110 BAUD TERMINALS

```
10  ?="HOUR ?";
20  H=?
30  ?="MINUTE ?";
40  M=?
50  ?="SECOND ?";
60  S=?
70  ?="READY"
80  A=$
90  S=S+1
100  M=S/60+M
110  S=%
120  H=M/60+H
```

```
130  M=%
140  H=H/24*0+%
150  ?=H/10
160  ?=%
170  ?=": ";
180  ?=M/10
190  ?=%
200  ?=": "?=": "
210  ?=S/10
220  ?=%
230  $=13
240  A=B
250  A=B
260  A=B
270  A=B+B
280  T=14
290  T=T-1
300  #=T=0*90
310  #=290
```

Factorials Program

This program calculates factorials until it runs out of memory. For 1K of memory, this is about 208!

```
10   A=l
20   L=2
30   :1)=1
40   I=2
50   :I)=0
60   I=I+1
70   #=L>I*50
80   ?=""
90   ?=""
ioo  ?=A
110  ?=":="
120  ?=""
130  I=L+1
140  I=I-1
150  #=: I)=0*140
160  ?=: I)
170  I=I-1
180  #=I=0*220
190  ?=:I)/10
200  ?=%
210  #=170
220  A=A+1
230  I=1
240  C=0
250  X=:I)
260  :I)=A*X
270  #=: I)<X*320
280  : I)=: I)+C
290  C=: I)/100
300  : l)=%
310  I=I+1
320  #=L>I*250
330  #=C=0*80
340  L=L+1
350  #=*-&/2<L*380
360  :I)=C
370  #=290
```

Weekday Program

```
10   #=440
20   ?="DAY OF THE WEEK"
30   ?=""
40   ?="MONTH?      ";
50   M=?
60   #=M>13*40
70   #=M=0*40
80   ?="DAY OF MONTH?";    ";
90   D=?
100  ?="YEAR?"    "
110  Y=?
120  #=Y>1800*230
130  #=Y<100*150
140  #=70
150  ?=""
160  ?="IS THAT 19";
```

```
170  ?=Y
180  ?="? ";
190  K=$
200  #=K=89=0*70
210  ?="ES"
220  Y=Y+1900
230  C=Y/100
240  Y=%
250  #=Y/4*0+%=0*280
260  : 1)=6
270  : 2)=2
280  N=Y/4+Y+D+:M)+(2*(C=18))/7*0+%
290  #=300+(20*W)
300  ?="SUN";
310  #=430
320  ?="MON";
330  #=430
340  ?="TUES";
350  #=430
360  ?="WEDNES";
370  #=340
380  ?="THURS";
390  #=430
400  ?="FRI";
410  #=430
420  ?="SATUR";
430  ?="DAY"
440  :1)=0
450  :2)=3
460  :3)=3
470  :4)=6
480  :5)=1
490  :6)=4
500  :7)=6
510  :8)=2
520  :9)=5
530  :10)=0
540  :11)=3
550  :12)=5
560  #=20
```

Starshooter Program

```
10   I=0
20   I=I+1
30   : I)=46
40   #=I<41*20
50   :25)=42
60   I=8
70   J=1
80   $=I-/7+64
90   ?" - ";
100  S=I+J
110  $=:S)
120  J=J+1
130  #=J=6*160
HO ?=" ";
150  #=100
160  I=I+7
170  ?=""
180  ?=""
190  #=I<43*70
200  ?=""
210  ?=" 1    2    3    4    5"
220  ?=""
230  ?="YOUR MOVE --";
240  I=42
250  I=I+1
260  :I)=$
270  #=: I)=13*320
280  #=: I)=3*580
290  #=: I)=95=0*250
300  I=I-1
310  #=260
320  A=:43)-64
330  ?=""
340  #=A>6*230
350  B=:44)-48
360  #=B>6*230
370  S=A*7+1+B
380  ?=""
390  #=:S)=42*420
400  ?="THAT'S NOT A STAR!"
```

```
410  #=230
420  :S)=46
430  C=S-7
440  #=520
450  C=S-1
460  #=520
470  C=S+1
480  #=520
490  C=S+7
500  #=520
510  #=60
520  =!.
530  #=: C)=42*560
540  :C)=42
550  #=
560  :C)=46
570  #=
```

The object of the game is to change this:

```
A - . . . . .          A - * * * * *
B - . . . . .          B - *. . . *
C - . . * . .    to    C - *. . . *
D - . . . . .          D - *. . . *
E - . . . . .          E - * * * * *
    1 2 3 4 5              1 2 3 4 5
```

Factors of a Number Program

This version is for the TVT:

```
10  #=200
20  D=D+2/3*0+%=0*2+(D>3)+D+l
30  Q=N/D
40  #=Q<D*300
50  #=%>1*20
60  ?=""
70  ?=D
80  N=Q
90  Q=N/D
100  #=%>1*20
110  ?=" ";
130  P=l
140  N=Q
150  Q=N/D
160  P=P+1
170  #=%=0*140
180  ?=P
190  #=20
200  ?="NUMBER? ";
210  N=?
220  X=N
230  $=22
240  ?=" "          "
250  ?=N
260  ?="     IS     ";
270  D=2
280  #=30
300  #=N=X*370
310  #=N=1*340
320  ?=""
330  ?=N
340  ?=""
350  ?="DONE"
360  #=200
370  ?="PRIME"
380  #=200
```

Primes Program

This version is for the 32 Char Terminal:

```
10  #=100          30  D=D+2/3*0+%=0*2+(D>3)+D+l
20  #=D>Q*150      40  Q=N/D
```

```
50 #=%>1*20
60 N=N+2/3*0+%=0*2+(N>3)+N+l
70 D=2
80 #=N<65533*40
90 #=N
100 $=28
101 #=102
102 ?="                PRIMES"
104 ?=""
106 ?="              ";
110 ?=1
115 ?="  "?=" ";
120 N=2
130 A=l
150 B=N
160 #=B>10000*200
170 $=32
180 #=B>1000*200
190 B*B*10
195 #=170
200 ?=N
205 ?="";
210 A=A+1
220 #A<5*60
230 ?=""
240 A=0
250 #=60
```

Craps! Program

```
10  T=100
20  $=22
30  ?="CRAPS!"
40  ?=""
50  ?="HOW MUCH DO YOU BET? - ";
60  B=?
70  #=B=0*90
80  ?="GOOD LUCK!"
90  #=B=8*480
100  #=T>B*160
no  ?="TOO MUCH!"
120 ?="YOU HAVE $";
130 ?=T
140 ?=" LEFT. "
150 #=40
160 ?=""
170 ?="ROLL-";
180 A=?
190 $=22
200 ?="FIRST ROLL: ";
210 #=500
220 #=R=7*360
230 #=R11*360
240 #=R<4*390
250 #=R=12*390
260 ?=""
270 ?=R
280 ?=" IS YOUR POINT"
290 P=R
300 ?="ROLL-":
310 A=$
320 #=500
330 #=R=7*390
340 #=R=P*360
350 #=300
360 ?="YOU WIN"
370 T=T+B
380 #=120
390 T=T-B
400 ?="YOU LOSE"
410 #=T=0*430
420 #=120
430 ?="YOU ARE BUSTED!"
440 ?="MOVE OVER AND LET THE NEXT"
450 ?="SUCKER TRY."
460 ?=""
470 #=10
480 ?="BE SERIOUS"
490 #=40
500 R='/6*0+%+1
510 ?=R
520 X=X+11213
530 ?=" AND ";
540 S='/6*0+%+1
550 X=X*56001
560 ?=5
570 ?="     (";
580 R=R+S
590 ?=R
600 ?=")"
610 #=!
```

Cipher Game Program

```
10  I=0
20  I=I+1
30  :-)=I+64
40  +=I<26*20
50  I=I
60  ?=""
```

```
70  M='/26*0+%+l
80  H=:M)
90  : M)=: I)
100  : I)=H
110  I=I+1
120  #=I<27*70
130  ?="TEXT?"
140  ?=""
150  I=27
160  : I)=$
170  #=: I)=13*220
180  #=: I)=95=0*200
190  I=I-2
200  I=I+1
210  #=160
220  ?=""
230  I=27
240  #=: I)<64*270
250  T=: I)-64
260  : I)=T)
270  I=I+1
280  #=: I)>14*240
290  ?=""
300  ?="CODE: "
310  ?=""
320  I=27
330  $=: 1)
340  #=: I)=l3*370
350  I=I+1
360  #=330
370  ?=""
380  ?="SWITCH? - ";
390  A=$
400  B=$
410  #=B=64*370
420  I=27
430  #=: 1)=A*490
440  #=: I)=B=0*460
450  :I)=A
460  I=I+1
470  #=: I)=13*290
480  #=430
490  : I)=B
500  #=460
```

Phrase Sort Program

```
10  $=22
20  1=0
30  I=I+1
40  : I)=$
50  L=: I)=95*2
60  I=I- L
70  #=: I)>14*30
80  ?=""
90  I=1
100  K=I
110  J=K
120  #=:K)=32*160
130  #=:J)=32*150
140  #=:K)>: J)*160
150  J=K
160  K=K+1
170  #=:K)>14*120
180  H=: I)
190  : I)=: J)
200  :J)=H
210  I=I+1
220  #=: I)>14*100
230  I=0
240  I=I+1
250  $=:I)
260  #=: I)>14*240
270  ?=""
```

Life (Fast Version) Program

```
10  #=370
20  S=Y<F*Y+(Y=0*E)+(Y=F)-
    1*0+(X<Q*X+(X=0*0)+(X=Q))
25  : S)=: S)+2
30  X=X+1-(J<X*3)+(J-1=X*(Y=I))
40  Y=J-1=X+Y
50  #=I+1>Y*20
70  #=90
80  #=: I-l*O+J)/2*0+%*20
90  J=J+1-(O=J*O)
100  I=J=I+I
110  ?="";
120  X=J-1
130  Y=I-1
140  #=I<F*80
150  I=1
160  J=l
180  ?=""
190  P=0
200  K=I-1*O+J
210  : K)=: K)<5+(: K)>8)=0
220  P=P+: K)
230  $=: K)*10+32
240  J=J+1-(J=0*0)
250  #=1<J*200
260  ?=""
270  I=I+1
280  #=I<F*200
290  ?="GEN = ";
300  ?=G
310  G=G+1
320  ?="        POP = ";
330  ?=P
340  I=1
350  J=l
360  #=0<P*110+(P=0*650)
370  I=1
380  G=0
390  ?="SIZE? ";
400  O=?
```

```
410  Q=O+1
420  ?="BY? ";
430  E=?
440  F=E+1
450 J=O*E+2
460  #=J*2+&>**390
470  : I)=0
480 I=I+1
490  #=J>I*470
495  #=631
500  I=1
510  ?=""
520 J=l
530  #=10*550
540  ?="";
550  ?=I
560  ?="";
570  L=$
580  : I- l*O+J)=L=32+(L=13)+(L=95)+(L=64)=0*6
590 J=J+1-(L=95*2)
600  #=L=13*620+(L=64*510)
610  #=J<Q*570
620  I=I+1
625  #=I<F*510
626  #=631
627  #=150
631  $=22
632  $=18
633  $=32
634  $=18
635  $=22
636  $=18
637  $=32
638  $=18
640  $=!
```

This program takes at least 2 K of memory to operate satisfactorily.

Motorola 6809D4

The MEK6809D4 advanced microcomputer evaluation board and MEK68KPD keypad/display unit provide the necessary hardware and firmware for a computer system based on the Motorola MC6809 high performance microprocessor (Fig. A-l). The system forms an evaluation tool to facilitate the application of Motorola microprocessors and associated components.

The MEK6809D4A is used with an MEK68KPD and is complete with a power supply (Fig. A-2). The MEK6809D4B requires an external power supply and is used with RS-232 terminal or an MEK68R2D CRT interface plus a CRT and an ASCII keyboard. See Table A-l.

The user can prototype dedicated systems plus write and evaluate software programs in machine language, using a cassette recorder/player for data storage. Provisions are made for extensive system expansion.

HIGHLIGHTS

- System buffers are used between sections of the MEK6809D4 board and between the board and its edge connectors.
- Hardware RAM and ROM page select register.
- 4K static user RAM (eight sockets) may be mapped with jumpers to appear at any 4K block in the 64K basic memory space, and in addition may be jumpered to appear on a selected "RAM page/or pages" as controlled by a 3-bit hardware RAM page register.
- Eight 24-pin ROM sockets may be configured to accept combinations of ROM/EPROM types including 1K x 8 single or

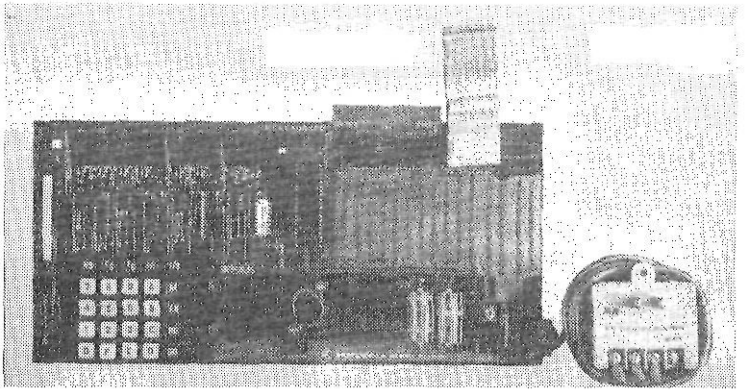


Fig. A-1. The MEK6809KPD keypad/display unit (courtesy of Motorola Semiconductor Products Inc.).

triple supply EPROMs or ROMs, 2K x 8 single or triple supply EPROMs or ROMs, 4K x 8 ROMs or EPROMs, or 8K x 8 ROMs or EPROMs.

- A ROM-based mapping technique is used to allow completely general address mapping of the eight ROM sockets anywhere in the 64K basic memory space with 1K resolution. In addition, the sockets may be mapped on any "ROM page/or pages" as controlled by a 3-bit hardware ROM page register.
- All memory and I/O on the board is fully decoded, so that address space not specifically required on the D4 is available for off-board mapping.
- A -12 volt to -5 volt regulator is provided to allow use of 3-supply EPROMs on the D4. Supply voltages of +12, -12, and +5 must be provided by the user.
- Hardware is provided which allows Monitor software to store and recover Kansas City Standard 300-baud or 1200-baud format cassette tape data.
- Interrupt driven stop-on-address comparator.
- System clock derived from 3.579 MHz on-board XTAL or from a 4 x TTL compatible external source.
- "Test" signal and logic provided to allow control of on-board memory and I/O from an external processor through the 70-pin edge connector.
- Control and status lines provided for flexible hardware control of MPU and bus decode drive logic. This allows for:

—Testing and debug
-Interrupts (RESET, NMI, IRQ, FIRQ)

—Interrupt vectoring by device (IVE, STKOP)
-Interrupt disable (IRQE, FIRQE)
—Halt and bus request (BREQ)
—Slow memory (MEMRDY)
-DMA

The following features are standard on the MEK6809D4B and may be included as options on the MEK6809D4A: RS-232 compatible serial port including buffered handshake signals; baud rate generator providing baud rate clocks for 110, 300, 600, 1200, 4800, and 9600 baud rates; and address, data and control lines fully buffered at bus interface.

MODEL TYPES

The MEK6909D4A has no RS-232 circuitry or address and data buffers to the edge connector. The D4A is intended for use with the MEK68KPD keypad/display unit which has an on-board power supply to operate the system. No RAMs are provided in the "user RAM" array. A 4K monitor program is provided (Fig. A-3 and Table A-2).

The MEK6809D4B is intended for use with an RS-232 serial terminal or an MEK68R2D CRT interface as the system terminal. The D4B has RS-232 circuitry and data and address buffers.

To operate the RS-232 interface, the user must supply +12V, +5V, and -12 V power. A4K + 2K monitor is provided. No RAMs are provided in the "user RAM" array.

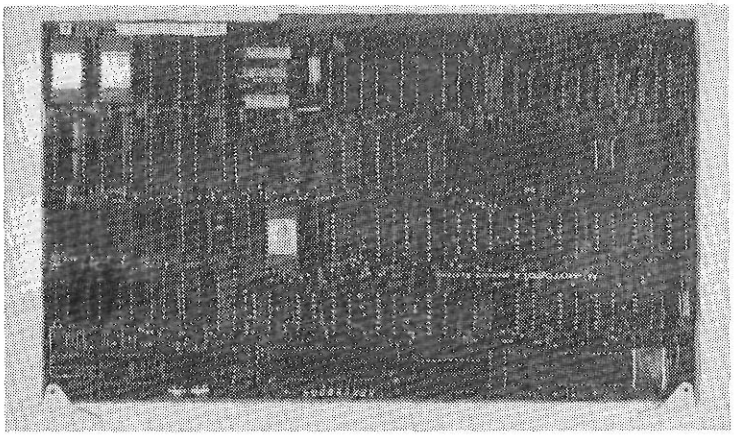


Fig. A-2. The basic MEK6809D4 advanced microcomputer evaluation board (courtesy of Motorola Semiconductor Products Inc.).

Table A-1. Product Features of the MEK6809D4 and MEK68KPD (courtesy of Motorola Semiconductor Products Inc.).

MEK6809D4
- MC6809 High Performance Microprocessor
- D4BUG Monitor Firmware (4K) Expandable to 6K
- Direct Memory Access
- Interrupt by Device
- Audio Cassette Interlace, 300 or 1200 Baud
- Optional RS-232 Port with MC6850 ACIA
- System RAM, 512 Bytes Expandable to 1K
- User RAM. 512 Bytes Expandable to 4K
- RAM/ROM Pago Select Register
- ROM Mapping Technique
- All I/O and Memory Fully Decoded
- Stop-On-Address Comparator
- System Clock Internal or External
- Test Signal and Control Logic tor Bidirectional Address Bus
- Control and Status Lines
- System Buffers

MEK68KPD- Eight 7-Segment Displays
- 25-Key Keypad
- On-Board Power Supply
- User PI A, MC6821
- Wire-Wrap Area
- 16-Pin Auxiliary Socket

DIMENSIONS
- MEK6809D4:
  Two-sided PC Board
  309.8 mm (12 in) Wide by
  177.8 (7 in) High by
  1.59 mm (0.062 in) Thick
- MEK68KPD:
  Two-sided PC Board
  304.8 mm (12 in) Wide by
  157.5 mm (6.2 in) High by
  1.59 mm (0.062 in) Thick

SUPPW VOLTAGES (£5%)
- MEK6809D4:
  5 Vdc at 2.0 A max (with all options)
  + 12 Vdc at 25 mA max
  - 12 Vdc at - 23 mA max
- MEK68KPD:
  120 Vac 60 Hz, produces 18 Vac (ct) input to board. An on-board regulator provides power required to operate the KPD plus D4 board in minimum configuration.

ENVIRONMENTAL
Operating Temperature: 0°C to 55°C (32°F to 131°F)

Relative Humidity: to 80* without condensation.

MOKEP M-60 and M-70 products are compatible with the D4B, thus allowing expansion to an ASCII keyboard interface to the microcomputer system. The MEK68KPD (with its onboard power supply disconnected) may be used with the MEK6809D4B.

The MEK68KPD is the keypad/display unit intended for use with the MEK6809D4 board and interfaces electrically with the MEK6809D4. Standard interface to the D4 is via a 24-conductor cable and plug assembly supplied with the KPD unit.

Table A-2. MEK6809D4 ac **Operating** Conditions and Characteristics (courtesy of Motorola Semiconductor Products Inc.).

AC OPERATING CHARACTERISTICS (Bus)

Parameter	Symbol	Min	Nom	Max	Unit
Cycle Time	t_{cyc}	1100	–	1130	ns
Address Setup	t_{AQ}	25	–	–	ns
Address Hold[3]	t_{AH}	10	30	–	ns
Write Data Valid	t_{DVW}	–	–	250	ns
Write Data Hold	t_{DHW}	10	30	–	ns
E (φ2) Low Time	$t_{\phi 2L}$	500	–	–	ns
E (φ2) High Time	$t_{\phi 2H}$	500		–	ns
E Low to Q High	t_{EQ}	215	–	–	ns
Q High Time	t_{QH}	500	–	–	ns

AC OPERATING CONDITIONS (Bus)

Parameter	Symbol	Min	Nom	Max	Unit
Access Time	t_{ACC1}[4]	–	–	350	ns
	t_{ACC2}	–	–	720	ns
Read Data Hold[3]	t_{DHR}	0	–	–	ns

NOTES: 1) Operating temperatures TA = 25°C
2) Timing measured at edge connector (50% points)
3) Measured from falling edge of E (02)
4) Measured from rising edge of (02)

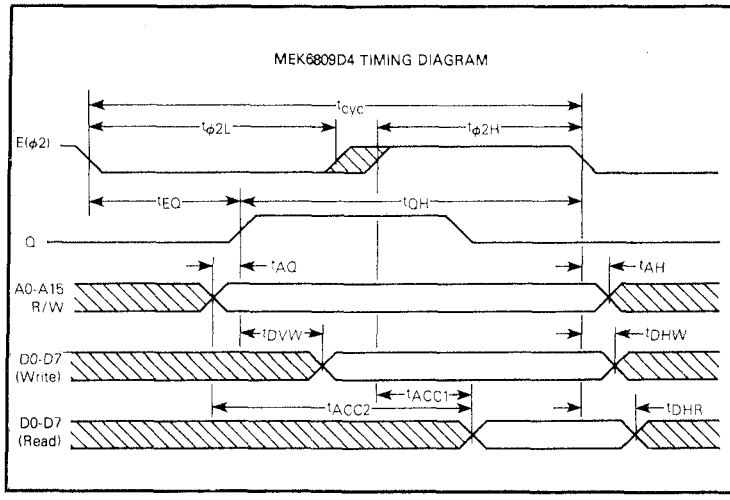


Fig. A-3. MEK6809D4 timing diagram (courtesy of Motorola Semiconductor Products Inc.).

EXPANSION

With the rapid advancements in the microprocessor industry, there is vital need to provide educational and evaluation material to help engineering/technical personnel stay abreast of this technology.

In response to this need Motorola Memory Systems has evolved a series of kit boards intended for the educational evaluation of the MC6800 family of integrated circuits. The series is called "MOKEP" (for Motorola Kit Expansion Products) and includes the following wide range of boards.

MEK68CC Card Cage. The MEK68CC is used with the MEK68MB5 motherboard.

MEK68MB5 Motherboard Module. The MEK68MB5 motherboard module has provisions for 10 card slots on " centers, with alternate slots populated with 70-Pin connectors.

MEK68CMB Card Cage/Motherboard. The MEK68CMB can accommodate ten cards of the MOKEP series. The card cage is identical to the MEK68CC. The motherboard is a fully populated version of the MEK68MB5 without the stand-alone card guides. The completed assembly measures 8-¼" high by 7-¼" wide by 13¼" deep.

MEK68R2/R2D/R2M Programmable CRT Interface Modules. The MEK68R2/R2D/R2M programmable CRT inter-

face modules are used in conjunction with other products in the MOKEP family to form a microcomputer system. The MEK68R2D is to be used with the MEK6809D4 microcomputer module and an **MEK68MB** series motherboard. All units feature software programmable line and character format, upper and lower case 5x7 matrix display, semigraphics, and up to 4K of screen display memory. All modules provide an interface for an ASCII keyboard.

MEK6810 Input/Output Module. The MEK6810 is supplied with a 300/1200 baud cassette interface, two MC6850 A CIAs, an **MC14411** baud rate generator and **one MC6821 PIA.**

MEK68EP EPROM Programmer Module. The MEK68EP has provisions for programming both single and triple power supply types of 1K, 2K and 4K EPROMs.

MEK68RR ROM/RAM Module. The MEK68RR has provisions for eight ROM sockets which may be configured to accept 1K, 2K 4K or 8K single or triple supply ROMs or EPROMs. The board also has sockets for up to 8K bytes of static RAM.

MEK68MM16/MM32 16K/32K Memory Modules. The MEK68MM16 has 16K bytes of RAM and the MEK68MM32 has 32K bytes. The MEK68MM boards employ 16K dynamic RAMs and a hidden refresh technique to achieve the low cost, low power consumption and high density of dynamic memory systems, while appearing as static memory to the system. The MEK68MM fully supports the RAM paging technique of the D4 microcomputer module, allowing up to eight boards or 256K bytes of RAM to be used in one system.

MEK6809A Editor/Assembler. The MEK6809EA editor/assembler provides the user of the MEK6809D4B with the ability to enter, assemble, edit and save assembly language programs for execution on the M6809. The editor may also be used to enter and edit text files that will not be assembled for execution. The assembler will accept both M6800 and M6809 mnemonics. The object code from the assembler can be placed in memory or saved on tape. The MEK68R2D display and stand alone terminals are supported by software.

MEK68WW/WW1 Wirewrap Modules. The MEK68WW is used with the MEK6800AD adapter motherboard, and interfaces directly with the 60-Pin bus of the AB. The MEK68 WW1 utilizes a 70-pin bus, directly interfacing with the MEK68MB series motherboards. Either product can be used as a card extender. Both are supplied with components required for buffering of address, data and control buses.

SOFTWARE FEATURES

- Memory change display
- Register change/display
- Breakpoint editor
- 4K monitor in position independent code (6K for MEK6809D4B version)
- Trace single step and user line
- Go to user program
- Calculate offset
- Cassette punch/load/verify
- Stop on address
- Escape from all functions
- 16 User special functions
- Additional features on D48 include memory dump to examine blocks of data, memory fill, memory search, memory move and ASCII entry

The MEK6809D4 operating system allows the development and operation of user-defined programs. The basic monitor program interfaces with an MEK68KPD keypad/display unit and is contained in a 4K byte ROM (MCM68332 or equivalent). This ROM is factory installed in all versions of the MEK6809D4 assembled units.

A second 2K byte ROM (MCM68316E or equivalent) is used with an MEK68R2D CRT monitor/ASCII keyboard interface or RS-232 compatible terminal. This additional 2K ROM is provided in the.MEK6809D4B version.

The monitor program source listings, complete with comments, are available from Motorola. The monitor program is written in highly subroutined, position independent code. These source listings provide a valuable starting point for many types of user programs.

The monitor program provides the following functions.

Examine/Change Memory Location

This allows the user to open any memory location and display the contents. New data may then be entered if desired, assuming read/write memory is present at the selected location. If an attempt is made to write into an invalid location, the new data will be displayed together with the fixed data at the invalid location. Only the new data is displayed when a valid change of read/write memory is accomplished. After the examine/change step, the user has

the option of automatically opening either the next or previous location—or escaping to the monitor program.

Examine—Change Registers

This function allows the user to examine/change two external registers plus those areas of the stack RAM corresponding to the storage locations of the nine internal registers of the MC6809. This has the effect of allowing the user to examine/change these registers.

This function differs from memory examine/change in that the registers are displayed in a set sequence. Register designation as well as contents are displayed to facilitate use of the function. The two external registers are incorporated on the MEK6809D4 to perform operations not inherent with the MC68Q9.

Stop on Address

In de-bugging programs, it is often advantageous to be able to halt the machine when a specific address is encountered. A typical example of the use of this function is to determine the reason for an inadvertent (or incorrect) change of a memory location during the running of a User program.

The stop on address function (SOA) function is implemented on the MEK6809D4 by circuitry which compares the MPU address outputs with user-entered data in the stop on address register. Providing the SOA function is armed, a non-maskable interrupt is generated when a comparison is achieved.

Depending on the type of instruction (more specifically, upon the timing relationship of the address assertion in the instruction cycles), the NMI may be recognized at the end of the previous instruction. Control then passes to the monitor, allowing the user to determine that one of two specific instructions has accessed the specified memory location.

In some instances it is desirable to allow the program to stop only on the N^{th} time an address is encountered. The MEK6809D4 can implement this function. It is also possible to output a trigger pulse each time the address is encountered, rather than stopping program execution.

Breakpoints

The SOA function is implemented in hardware. Software methods of program execution interruptions include the setting of

breakpoints at desired locations in the program. This effectively substitutes a software interrupt for the instruction at that location.

Up to eight breakpoints can be set in the user program (provided the program is in RAM). As with SOA, the user has the option of allowing N-l breakpoints to be bypassed if desired. (The maximum value of N for either SOA or breakpoints is 255). The User can set, clear or examine breakpoints via the breakpoint editor function.

Trace Instruction

This function allows the User to step through a program one instruction at a time. At the end of each instruction, the examine/change register routine is automatically entered, and the new program counter value is displayed.

Trace Line

It is often desirable to trace through a program while treating a subroutine as a single instruction. One obvious example of this is the situation wherein all subroutines have previously been thoroughly de-bugged. The MEK6809D4 debug routines allow this to be accomplished in either of two ways.

One of these (software method) involves a comparison of each instruction in a subroutine until the instruction following the subroutine is encountered. Thus, the portion of the program from a subroutine call to its return is treated as one instruction as far as the *trace* function is concerned. Nested subroutines are automatically handled by the monitor program.

The second trace line option uses the SOA circuitry. This has an advantage over the software method in that subroutine execution is in real time. This is particularly helpful in de-bugging time-dependent I/O routines. (It is also desirable for long subroutines, since the software method greatly increases the run time of a subroutine). Its disadvantage is that program execution often continues for one instruction after the return.

User Program Control

The MEK6809D4 de-bug routines include functions to allow the user to go to, continue, or abort user programs.

Offset Calculation

In generation or modification of programs, it is often necessary to calculate the offset from the location of a jump or branch

instruction to its destination. Some indexed mode instructions also use relative offsets. The examine/change memory location includes a subfunction to allow this to be easily accomplished.

The user opens the location of the offset, types the offset command, then enters the desired destination address. The MEK6809D4 calculates the required offset, displays it, and enters the data in the appropriate memory location(s). The offset calculation supports both short and long offsets.

Punch/Load/Verify Audio Cassette

The audio cassette interface is a modified Kansas City Standard version capable of operation at either 300 or 1200 baud. The interface allows any of the three functions (punch, load or verify with memory) to operate with or without an optional offset. This is particularly useful for user programs written in position independent code.

ADDED D4B SOFTWARE FEATURES

The *memory dump* command allows the user to display blocks of memory with ASCII equivalents. The display for mats differ slightly depending on the display device configuration. The ending address must be a larger hexadecimal number than the beginning address or a warning will be issued, followed by a new request for a begin address. The dump command cannot proceed until a satisfactorily address range has been specified.

Memory fill allows the user to fill a block of memory with a four byte pattern. The beginning and ending addresses are entered as in the memory dump command.

Memory search allows search of a specified block of memory for a 4 byte pattern subject to a corresponding 4 byte mask. For all bits in the mask which are zero, the corresponding bit in the pattern is considered to be "don't care."

After the mask has been specified, the search function will be performed over the specified address range. Each time the comparison alogrithm is successful, the address of the first location of the match is displayed. If the list of successful addresses is being displayed too quickly, the listing may be temporarily halted by typing (ESCAPE) as in the memory dump command.

Memory Move allows the user to move a block of data from one area in memory to a new area. The beginning and ending addresses are entered the same way as in memory dump. Following entry of

the end address, a message will appear requesting entry of the new beginning address where the block of data is to be moved.

ASCII Entry allows a user to store ASCII data to memory quickly and easily without having to look up each ASCII character to determine its hexadecimal equivalent. Features are incorporated to assist in setting up messages for the D4BUG callable subroutine "PDATA."

MEK6809D4 DESCRIPTION

The CPU consists of an MC6809 high performance microprocesor, a 3.579 HMz crystal, and buffers which interface the MC6809 to other circuit blocks on the D4 board.

The MC6809 supports programming techniques such as position independence, re-entrancy and modular programming. The MC6809 has hardware and software features which make it a suitable processor for higher level language execution or standard controller application (Fig. A-4).

Rom

The ROM system consists of eight ROM sockets which may be configured to accept various combinations of ROM types. These include single or triple power supply varieties of 1K, 2K, 4K, or 8K x 8 ROMs or EPROMs. The ROM type configurations are controlled by mini-jumpers which may be easily moved without the need for any tools.

Mapping of the eight ROMs in memory space is accomplished by a mapping ROM which is used as a programmed logic array. A paging technique allows up to 192K bytes of ROM to be used in the D4 system.

Ram

There are two groups of 1K x 4 RAMs. One group is *the stack RAM* and the other is the *user RAM,*

The stack RAM is used mainly for the D4 operating system stack and scratch RAM. Also, 512 bytes are available for user RAM application. This RAM is always located in the D4 system at memory location $E400 through $E7FF. The user RAM is a 4K x 8 block of memory that can be positioned anywhere in the D4 memory map, using jumper connections.

It is possible to disable the eight user RAM sockets to allow use of a MOKEP MEK68MM memory board for system expansion.

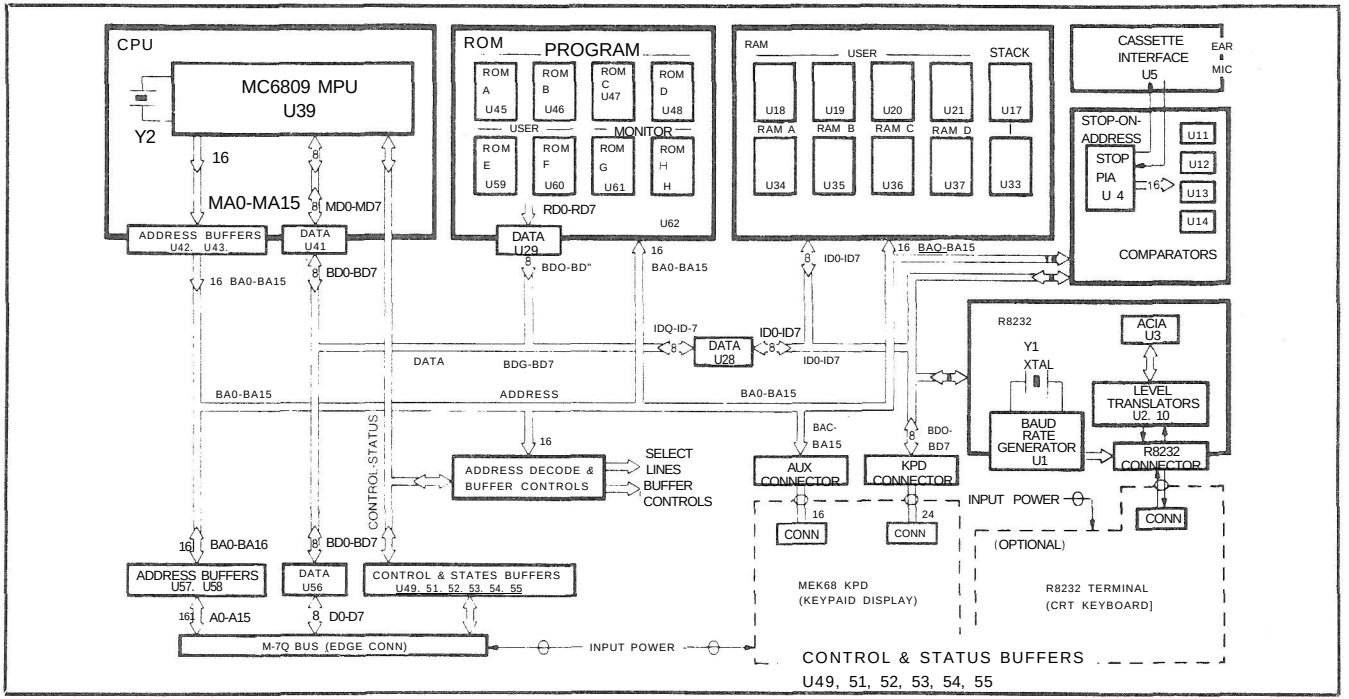


Fig. A-4, MEK6809D4 block diagram (courtesy of Motorola Semiconductor Products Inc.).

Another jumper, when removed, prevents the user RAM from being written into (write protect), but the stack RAM is not affected. With the jumper in place, read and write to the user RAM is normal.

Address Bus System

In some microprocessor systems, the address flow is from the microprocessor through buffers and to the motherboard bus. The D4 uses a more complex arrangement that permits disabling the microprocessor. This allows addresses to be fed to the D4 from an external source, to access board components and permits DMA (direct memory access) for some applications.

The Data Bus System

There are four bi-directional data buffers used in the D4; a ROM buffer, RAM/IO buffer, edge buffer, and MPU buffer. Each buffer has an enable and a direction input. Control logic configures these buffers to route data between sections of the D4 board.

The Stop-On-Address Circuit

The purpose of a stop address is to enable a user program to be executed until a certain address is reached. The address to be stopped on is stored and when the board address bus bits coincide, an output results. This causes a nonmaskable interrupt to occur which switches the microprocesor to a service routine.

When a coincidence of address occurs, an NMI is not necessarily generated. In these cases, the comparator output is available at a test point to provide a trigger signal to an oscilloscope.

The RS-232 Circuit

The RS-232 specification defines a standard for interconnecting computer terminals of different makes. The ACIA converts the parallel data on the buses to serial data. The serial data is then translated into RS-232 levels.

The Cassette Circuit Interface

The D4 uses very few components to interface a tape recorder to its operating system. Most of the cassette operation occurs in software, to create tapes and recover data from tapes. The tape information consists of a stream of 1200 and 2400 Hz serial audio data.

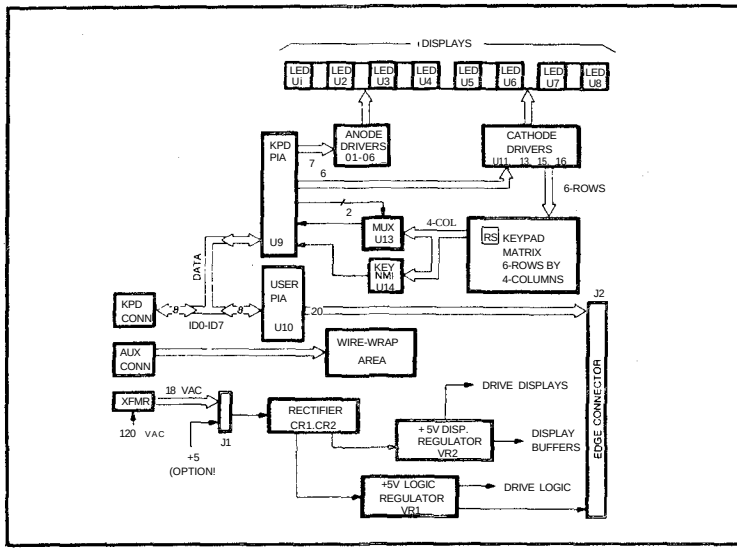


Fig. A-5. MEK68KPD block diagram (courtesy of Motorola Semiconductor Products Inc.).

MEK68KPD DESCRIPTION

The MEK68KPD includes a 25-key keypad, eight 7-segment LED displays, an on-board +5 volt power supply, and an uncommitted MC6821 PIA. Provisions are made to allow disconnection of the on-board regulators when an external +5 volt supply is used. A wire-wrap area is provided for custom circuitry and a 16-pin socket allows for additional signals to be brought to or from this wire-wrap area (Fig. A-5).

The display consists of eight seven-segment LEDs with a character height of 0.5 inches. The grouping of the displays is in a 4-2-2 linear array. A suitable anti-glare filter is provided with each MEK68KPD.

The PIAs used on the KPD are fully decoded via a peripheral chip select signal and address lines A0-A2 from the MEK6809D4. Data is furnished via the input cable.

ORIGINAL WHITE

Appendix B

Hexadecimal Values of Machine Codes

OP Mown	Mode	~	#	OP Mnem	Mode	~	#
113F SWI/3	Inherent	20	2	119C CMPS	Direct	7	3
1183 CMPU	immed	5	4	11A3 CMPU	Indexed	7+	3+
118C CMPS	immed	5	4	11AC CMPS	Indexed	7+	3+
1193 CMPU	Direct	7	3	11B3 CMPU	Extended	8	4
				11BC CMPS	Extended	8	4

NOTE All unused opcodes are both undefined and illegal

Coui'leBy of American Microsystems, Inc.

OP Mnem	Mode	~	#
00 NEG	Direct	6	2
01 •			
02 •			
03 COM		6	2
04 LSR		6	2
05 •			
06 ROR		6	2
07 ASR		6	2
08 ASL/LSL		6	2
09 ROL		6	2
OA DEC		6	2
OB •			
OC INC		6	2
OD TST		6	2
OE JMP		3	2
OF CLR	Direct	6	2
10 Page 2	—	—	—
11 Page 3	—	—	—
12 NOP	Inherent	2	1
13 SYNC	Inherent	2	1
14 •			
15 •			
16 LBRA	Relative	5	3
17 LBSR	Relative	9	3
18 •			
19 DAA	Inherent	2	1

OP Mnem	Mode	~	#
30 LEAX	Indexed	4 +	2 +
31 LEAY		4 +	2 +
32 LEAS		4 +	2 +
33 LEAU	indexed	4 +	2 +
34 PSHS	Inherent	5 +	2
35 PULS		5 +	2
36 PSHU		5 +	2
37 PULLI		5 +	2
38 •			
39 RTS		5	1
3A ABX		3	1
3B RTI		6/15	1
3C CWAI		20	2
3D MUL		11	1
3E •			
3F SWI	Inherent	19	1
40 NEGA	Inherent	2	1
41 •			
42 •			
43 COMA		2	1
44 LSRA		2	1
45 •			
46 RORA		2	1
47 ASRA		2	1
48 ASLA/LSLA		2	1
49 ROLA		2	1

OP Mnem	Mode	~	#
60 NEG	Indexed	6+	2+
61 •			
62 •			
63 COM		6+	2+
64 LSR		6+	2+
65 •			
66 ROR		6+	2+
67 ASR		6+	2+
68 ASL/LSL		6+	2+
69 ROL		6+	2+
6A DEC		6+	2+
6B •			
6C INC		6+	2+
60 TST		6+	2+
6E JMP		3 +	2+
6F CLR	Indexed	6+	2+
70 NEG	Extended	7	3
71 •			
72 •			
73 COM		7	3
74 LSR		7	3
75 •			
76 ROR		7	3
77 ASR		7	3
78 ASL/LSL		7	3
79 ROL		7	3

1A ORCC	Immed	3	2
1B •			
1C ANDCC	Immed	3	2
1D SEX	Inherent	2	1
1E EXG		8	2
1F TFR	Inherent	6	2
20 BRA	Relative	3	2
21 BRN		3	2
22 BHI		3	2
23 BLS		3	2
24 BHS/BCC		3	2
25 BLO/BCS		3	2
26 BNI		3	2
27 BEQ		3	2
28 BVC		3	2
29 BVS		3	2
2A BPL		3	2
2B BMI		3	2
2C BGE		3	2
2D BIT		3	2
2E BGT		3	2
2F BLE	Relative	3	2

4A DECA		2	1
4B •			
4C INCA		2	1
4D TSTA		2	1
4E •			
4F CLRA	Inherent	2	1
50 NEGB	Inherent	2	1
51 •			
52 •			
53 COMB		2	1
54 LSRB		2	1
55 •			
56 •			
56 RORB		2	1
57 ASRA		2	1
58 ASLB/LSLB		2	1
59 ROLB		2	1
5A DECB		2	1
58 •			
5C INCB		2	1
50 TSTB		2	1
5E •			
5F CLRB	Inherent	2	1

7A DEC		7	3
7B •			
7C INC		7	3
7D TST		7	3
7E JMP		4	3
7F CLR	Extended	7	3
80 SUBA	Immed	2	2
81 CMPA		2	2
82 SBCA		2	2
83 SUBD		4	3
84 ANDA		2	2
85 BITA		2	2
86 LDA		2	2
87 •			
88 EORA		2	2
89 ADCA		2	2
8A ORA		2	2
8B ADDA		2	2
8C CMPX	Immed	4	3
80 BSR	Relative	7	2
8E LDX	Immed	3	3
8F •			

NOTE: All unused opcodes are both undefined and ilegal

Legend

~ Number of MPU cycles (less possible push/pull or indexed-mode cycles)

Number of program bytes

* Denotes unused opcode

OP	Mnem	Mode	~	#
90	SUBA	Direct	4	2
91	CMPA		4	2
92	SBCA		4	2
93	SUBD		6	2
94	ANDA		4	2
95	BITA		4	2
96	LDA		4	2
97	STA		4	2
98	EORA		4	2
99	ADCA		4	2
9A	ORA		4	2
9B	ADDA		4	2
9C	CMPX		6	2
9D	JSR		7	2
9E	LDX		5	2
9F	STX	Direct	5	2
A0	SUBA	Indexed	4+	2+
A1	CMPA		4+	2+
A2	SBCA		4+	2+
A3	SUBD		6+	2+
A4	ANDA		4+	2+
A5	BITA		4+	2+
A6	LDA		4+	2+
A7	STA		4+	2+
A8	EORA		4+	2+
A9	ADCA		4+	2+
C3	ADDD	Immed	4	3
C4	ANDB		2	2
C5	BITB		2	2
C6	LDB		2	2
C7	•			
C8	EORQ		2	2
C9	ADCB		2	2
CA	ORB		2	2
CB	ADDB		2	2
CC	LDD		3	3
CD	•			
CE	LDU)	Immed	3	3
CF	•			
D0	SUBB	Direct	4	2
D1	CMPB		4	2
D2	SBCB		4	2
D3	ADDD		6	2
D4	ANDB		4	2
D5	BITB		4	2
D6	LDB		4	2
D7	STB		4	2
D8	EORB		4	2
D9	ADCB		4	2
DA	ORB		4	2
DB	ADDB		4	2
DC	LDD		5	2
F6	LDB	Extended	5	3
F7	STB		5	3
F8	EORB		5	3
F9	ADCB		5	3
FA	ORB		5	3
FB	ADDB		5	3
FC	LDD		6	3
FD	STD		6	3
FE	LDU		6	3
FF	STU	Extended	6	3
1021	LBRN	Relative	5	4
1022	LBHI		5(6)	4
1023	LBLS		5(6)	4
1024	LBHS/LBCC		5(6)	4
1025	LBCS/LBLO		5(6)	4
1026	LBNE		5(6)	4
1027	LBEQ		5(6)	4
1028	LBVC		5(6)	4
1029	LBVS		5(6)	4
102A	LBPL		5(6)	4
102B	LBMI		5(6!	4

AA ORA		4+	2+
AB ADDA		4+	2+
AC CMPX		6+	2+
AD JSR		7+	2+
AE LDX		5+	2+
AF STX	Indexed	5+	2+
B0 SUBA	Extended	5	3
B1 CMPA		5	3
B2 SBCA		5	3
B3 SUBD		7	3
B4 ANDA		5	3
B5 BITA		5	3
B6 LDA		5	3
B7 STA		5	3
B8 EORA		5	3
B9 ADCA		5	3
BA ORA		5	3
BB ADDA		5	3
BC CMPX		7	3
BC JSR		8	3
BE LDX		6	3
BF SFX	Extended	6	3
C0 SUBB	Immed	2	2
C1 CMPB		2	2
C2 SBCB	Immed	2	2

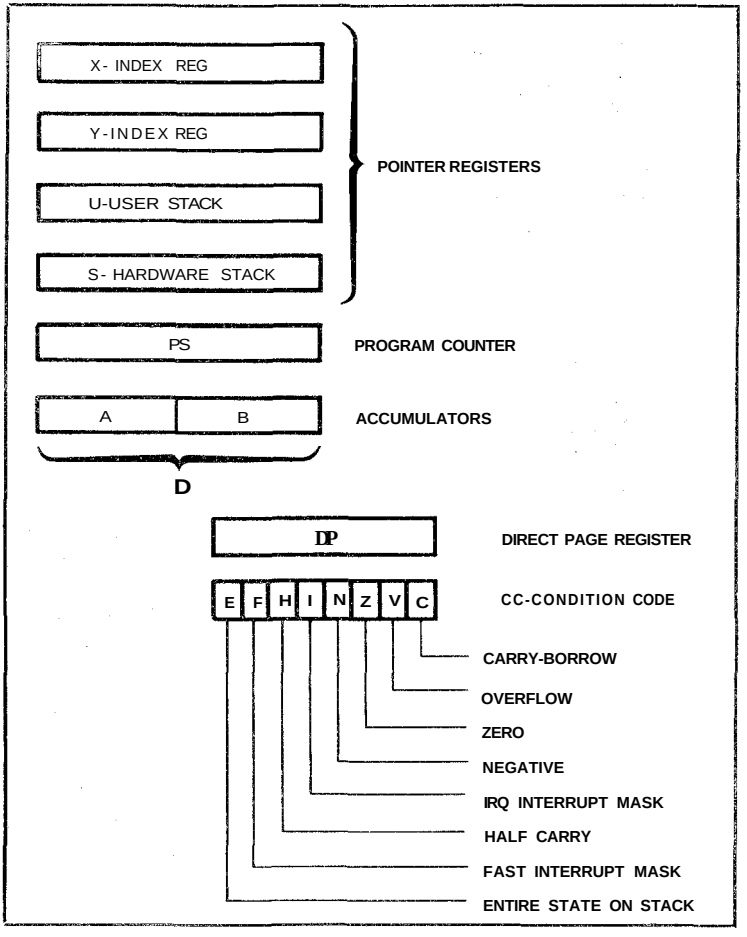
DD STD		5	2
DE LDU		5	2
DF STU	Direct	5	2
E0 SUBB	Indexed	1+	2+
E1 CMPB		4+	2+
E2 SBCB		4+	2+
E3 ADDD		6+	2+
E4 ANDB		4+	2+
E5 BITB		4+	2+
E6 LDB		4+	2+
E7 STB		4+	2+
E8 EORB		4+	2+
E9 ADCB		4+	2+
EA ORB		4+	2+
EB ADDB		4+	2+
EC LDD		5+	2+
ED STD		5+	2+
EE LOU		5+	2+
EF STU	Indexed	5+	2+
F0 SUBB	Extended	b	3
F1 CMPB		5	3
F2 SBCB		5	3
F3 ADDC		7	3
F4 ANDE		5	3
F5 BITB	Extended	5	3

102C LBGE		5(6)	4
102D LBLT	Relative	5(6)	4
102E LBGT	Relative	5(6)	4
102F LBLE	Relative	5(6)	4
103F SWI/2	Inherent	20	2
1083 CMPD	Immed	5	4
108C CMPY		5	4
108E LDY	Immed	4	4
1093 CMPD	Direct	7	3
109C CMPY		7	3
109E LDY		6	3
109F STY	Direct	6	3
10A3 CMPD	Indexed	7+	3+
10AC CMPY		7+	3+
10AE LDY		6+	3+
10AF STY	Indexed	6+	3+
1083 CMPD	Extended	8	4
10BC CMPY		8	4
10BE LDY		7	4
10BF STY	Extended	7	4
10CE LDS	immed	4	4
10DE LDS	Direct	5	3
10DF STS	Direct	6	3
10EE LDS	indexed	6+	3+
10EF STS	Indexed	6+	3+
10FE LDS	Extended	7	4
10FF STS	Extended	7	4

NOTE: All unused opcodes are both undefined and illegal

Appendix C

Programmer's Card



Courtesy of American Microsystems, Inc.

6809 STACKING ORDER

PULL ORDER ↓			
CC		6809 VECTORS	
A		FFFE	RESTART
B		FFFC	NMI
DP		FFFA	SWI
INCREASING MEMORY ↓ X Hi		FFF8	IRQ
X Lo		FFF6	FIRQ
Y Hi		FFF4	SWI2
Y Lo		FFF2	SWI3
U/S Hi		FFF0	RESERVED
U/S Lo			
PC Hi			
PC Lo			
↑ PUSH ORDER			

SIMPLE CONDITIONAL BRANCHES

Condition	Complement
BEQ	BNE
BMI	BPL
BCS	BCC
BVS	BVC

SIGNED CONDITIONAL BRANCHES

Condition	Complement
BGT	BLE
BGE	BLT
BEQ	BNE
BLE	BGT
BLT	BGE

UNSIGNED CONDITIONAL BRANCHES

Condition	Complement
BHI	BLS
BHS	BLO
BEQ	BNE
BLS	BHI
BLO	BHS

HEXADECIMAL **AND DECIMAL** CONVERSION

HOW TO USE THE TABLES

CONVERSION TO DECIMAL: Find the decimal weighls For corresponding hexadecimal characters beginning with the least significant character. The sum of the decimal weight is the decimal value of the hexadecimal number.

CONVERSION TO HEXADECIMAL: Find the highest decimal value in the table which is lower than or equal to the decimal number to be converted. The corresponding hexadecimal character is the most significant character. Subtract the decimal value found from the decimal number to be converted.. With the difference. repeat the process to find subsequent hexadecimal characters.

HEXADECIMAL AND DECIMAL CONVERSION

15 BYTE 8						7 BYTE 0			
15 CHAR 12			11 CHAR 8			7 CHAR 4		3 CHAR 0	
HEX	DEC		HEX	DEC		HEX	DEC	HEX	DEC
0	0		0	0		0	0	0	0
1	4	096	1	256		1	16	1	1
2	8	192	2	512		2	32	2	2
3	12	288	3	768		3	48	3	3
4	16	384	4	1	024	4	64	4	4
5	20	480	5	1	280	5	80	5	5
6	24	576	6	1	536	6	96	6	6
7	28	672	7	1	792	7	112	7	7
8	32	768	8	2	048	8	128	8	8
9	36	864	9	2	304	9	144	9	9
A	40	960	A	2	560	A	160	A	10
B	45	056	B	2	816	B	176	B	11
C	49	152	C	3	072	C	192	C	12
D	53	248	D	3	328	0	208	0	13
E	57	344	E	3	584	E	224	E	14
F	61	440	F	3	840	F	240	F	15

POWERS OF TWO

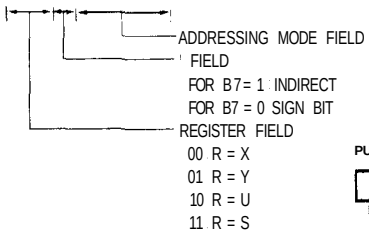
2^n	n
1	1
2	2
4	3
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1.024	10
2.048	11
4.096	12
8.192	13
16.384	14
32.768	15
65.536	16
131.072	17
262.144	18
524.288	19
1.048.576	20

ASCII CHARACTER SET (7-BIT CODE)								
M.S. CHAR / L.S. CHAR	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 0000	NUL	DLE	SP	0	@	P	\	p
1 0001	SOH	DC1	!	1	A	Q	a	q
2 0010	STX	DC2	"	2	B	R	b	r
3 0011	ETC	DC3	#	3	C	S	c	s
4 0100	EOT	DC4	$	4	D	T	d	t
5 0101	ENQ	NAK	%	5	E	U	e	u
6 0110	ACK	SYN	&	6	F	V	f	v
7 0111	BEL	ETB	'	7	G	W	g	w
8 1000	BS	CAN	(	8	H	X	h	X
9 1001	HT	EM	)	9	I	Y	i	y
A 1010	LF	SUB	*	:	J	Z	j	z
B 1011	VT	ESC	+	;	K	[	k	{
C 1100	FF	FS	,	<	L	\	l	l
D 1101	CR	GS	-	=	M	]	m	}
E 1110	SO	RS	.	>	N	↑	n	~
F 1111	SI	VS	/	?	O	←	o	DEL

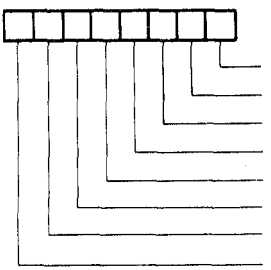
INDEXED ADDRESSING
POST BYTE REGISTER
BIT ASSIGNMENTS

POST-BYTE REGISTER BIT								INDEXED ADDRESSING MODE
7	6	5	4	3	2	1	0	
0	X	X	X	X	X	X	X	EA = R±4 BIT OFFSET
1	X	X	0	0	0	0	0	R +
1	X	X	X	0	0	0	1	R+ +
1	X	X	0	0	0	1	0	- R
1	X	X	X	0	0	1	1	- - R
1	X	X	X	0	1	0	0	EA=,R±0 OFFSET
1	X	X	X	0	1	0	1	EA=,R± ACCB OFFSET
1	X	X	X	0	1	1	0	EA=,R± ACCA OFFSET
1	X	X	X	1	0	0	0	EA=,R±7 BIT OFFSET
1	X	X	X	1	0	0	1	EA=,R±15 BIT OFFSET
1	X	X	X	1	0	1	1	EA=,R± 0 OFFSET
1	X	X	X	1	1	0	0	EA=,PC=7 BIT OFFSET
1	X	X	X	1	1	0	i	EA=,PC±15 BIT OFFSET
1	X	X	1	1	1	1	1	EA= ADDRESS

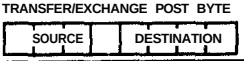
REGISTER FIELD
0000 = D (A:B)
0001 = X
0010 = Y
0011 = U
0100 = S
0101 = PC
1000 = A
1001 = B
1010 = CCR
1011 = DPR



PUSH/PULL POST BYTE



TRANSFER/EXCHANGE POST BYTE



OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
00	NEG	DIRECT	6	2	1C	ANDCC	IMMED	3	2	2E	BGT	RELATIVE	3	2
03	COM	↑	6	2	1D	SEX	INHERENT	2	1	2F	BLE	RELATIVE	3	2
04	LSR	↑	6	2	1E	EXG	↑	8	2	30	LEAX	INDEXED	4+	2+
06	ROR	↑	6	2	1F	TFR	INHERENT	6	2	31	LEAV	↑	4+	2+
07	ASR	↑	6	2	20	BRA	RELATIVE	3	2	32	LEAS	↓	4+	2+
08	ASL/LSL	↑	6	2	21	BRN	↑	3	2	33	LEAU	INDEXED	4+	2+
09	ROL	↑	6	2	22	BHI	↑	3	2	34	PSHS	INHERENT	5	3
0A	DEC	↑	6	2	23	BLS	↑	3	2	35	PULS	↑	5	2
0C	INC	↑	6	2	24	BRS/BCC	↑	3	2	36	PSHU	↑	5	2
0D	TST	↑	6	2	25	BLO/BCS	↑	3	2	37	PULU	↑	5	2
0E	JMP	↓	3	2	26	BNE	↑	3	2	39	RTS	↑	5	1
0F	CLEI	DIRECT	6	2	27	BEQ	↑	2	2	3A	ABX	↑	3	1
12	NOP	INHERENT	2	1	28	BVC	↑	3	2	3B	RTI	↑	6/15	1
13	SYNC	INHERENT	2	1	29	BVS	↑	3	2	3C	CWAI	↑	21	2
16	LBRA	RELATIVE	5	3	2A	BPL	↑	3	2	3D	MUL	↑	11	1
17	LBSR	RELATIVE	9	3	2B	SMI	↑	3	2	3F	SWI	↑	19	1
19	DAA	INHERENT	2	1	2C	BGE	↑	3	2	4Q	NEGA	↑	2	1
1A	ORCC	IMMED	3	2	2D	BLT	RELATIVE	3	2	43	COMA	INHERENT	2	1

OP	MNEM	MODE	~	#
44	LSRA	INHERENT	2	1
46	RORA		2	1
47	ASRA		2	1
48	ASLA/LSLA		2	1
49	ROLA		2	1
4A	DECA		2	1
4C	INCA		?	1
4D	TSTA		2	1
4F	CLRA		2	1
50	NEGB		2	1
53	COMB		2	1
54	LSRB		2	1
56	RORB		2	1
57	ASRA		2	1
58	ASLB/LSLB		2	1
59	ROLB		2	1
5A	DECB		2	1
5C	INCB	INHERENT	2	1

OP	MNEM	MODE	~	#
5D	TSTB	INHERENT	2	1
5F	CLRB	INHERENT	2	1
60	NEG	INDEXED	6+	2+
63	COM		6+	2+
64	LSR		6+	2+
66	ROR		6+	2+
67	ASR		6+	2+
68	ASL/LSL		6+	2+
69	ROL		6+	2+
6A	DEC		6+	2+
6C	INC		6+	2+
6D	TST		6+	2+
6E	JMP		3+	2+
6F	CLR	INDEXED	6+	2+
70	NEG	EXTENDED	7	3
73	COM		7	3
74	LSR		7	3
76	ROR	EXTENDED	7	3

OP	MNEM	MODE	~	#
77	ASR	EXTENDED	7	3
78	ASL/LSL		7	3
79	ROL		7	3
7A	DEC		7	3
7C	INC		7	3
7D	TST		7	3
7E	JMP		4	3
7F	CLR	EXTENDED	7	3
80	SUBA	IMMED	2	2
81	CMPA		2	2
82	SBCA		2	2
83	SUBD		4	3
84	ANDA		2	2
85	BITA		2	2
86	LDA		2	2
88	EORA		2	2
89	ADCA		2	2
8A	ORA	IMMED	2	2

OP	MNEM	MODE	~	#
8B	ADDA	IMMED	2	2
8C	CMPX	IMMED	4	3
8D	BSR	RELATIVE	7	2
8E	LDX	IMMED	3	3
90	SUBA	DIRECT	4	2
91	CMPA		4	2
92	SBCA		4	2
93	SUBD		6	2

OP	MNEM	MODE	~	#
9E	LDX	DIRECT	5	2
9F	STX	DIRECT	5	2
A0	SUBA	INDEXED	4 +	2 +
A1	CMPA		4 +	2 +
A2	SBCA		4 +	2 +
A3	SUBD		6 +	2 +
A4	ANDA		4 +	2 +
A5	BITA		4 +	2 +

OP	MNEM	MODE	~	#
B0	SUBA	EXTENDED	5	3
B1	CMPA		5	3
B2	SBCA		5	3
B3	SUBD		7	3
B4	ANDA		5	3
B5	BITA		5	3
B6	LDA		5	3
B7	STA		5	3

94	ANDA		4	2	A6	LDA		4 +	2 +	B8	EORA		5	3
95	BITA		4	2	A7	STA		4 +	2 +	B9	ADCA		5	3
96	LDA		1	2	A8	EORA		4 +	2 +	BA	ORA		5	3
97	STA		4	2	A9	ADCA		4 +	2 +	BB	ADDA		5	3
98	EORA		4	2	AA	ORA		4 +	2 +	BC	CMPX		7	3
99	ADCA		4	2	AB	ADDA		4 +	2 +	BD	JSR		8	3
9A	ORA		4	2	AC	CMPX		6 +	2 +	BE	LDX		6	3
9B	ADDA		4	2	AD	JSR		7 +	2 +	BF	STX	EXTENDED	6	3
9C	CMPX		6	2	AE	LDX		5 +	2 +	C0	SUBB	IMMED	2	2
9D	JSR	DIRECT	7	2	AF	STX	INDEXED	5 +	2 +	C1	CMPB	IMMED	2	2

OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
C2	SBCB	IMMED	2	2	D7	STB	DIRECT	4	2	E9	ADCB	INDEXED	4+	2+
C3	AODD		4	3	D8	EORB		4	2	EA	ORB		4+	2+
C4	ANDB		2	2	D9	ADCB		4	2	EB	ADDB		4+	2+
C5	BITB		2	2	DA	ORB		4	2	EC	LDD		5+	2+
C6	LDB		2	2	DB	ADDB		4	2	ED	STD		5+	2+
C8	EORB		2	2	DC	LDD		5	2	EE	LDU		5+	2+
C9	ADCB		2	2	DD	STD		5	2	EF	STU	INDEXED	5+	2+
CA	ORB		2	2	0E	LDU		5	2	F0	SUBB	EXTENDED	5	3
CB	ADDB		2	2	DF	STU	DIRECT	5	2	F1	CMPB		5	3
CC	LDD		3	3	E0	SUBB	INDEXED	4+	2+	F2	SBCB		5	3
CE	LDU	IMMED	3	3	E1	CMPB		4+	2+	F3	ADDD		7	3
D0	SUBB	DIRECT	4	2	E2	SBCB		4+	2+	F4	ANDB		5	3
D1	CMPB		4	2	E3	ADDD		6+	2+	F5	BITS		5	3
D2	SBCB		4	2	E4	ANDB		4+	2+	F6	LDB		5	3
D3	ADDD		6	2	Eb	BITB		4+	2+	F7	STB		5	3
D4	ANDB		4	2	E6	LDB		4+	2+	F8	EORB		5	3
D5	BITB		4	2	E7	STB		4+	2+	F9	ADCB		5	3
D6	LDB	DIRECT	4	2	E8	EORB	INDEXED	4+	2+	FA	ORB	EXTENDED	5	3

OP	MNEM	MODE	~	#
FB	ADDB	EXTENDED	5	3
FC	LDD	↑	6	3
FD	STD	↓	6	3
FE	LDU	↓	6	3
FF	STU	EXTENDED	6	3
1021	LBRN	RELATIVE	5	4
1022	LBHI	↑	5(6)	4
1023	LBLS	↓	5(6)	4
1024	LBHS/LBCC	↓	5(6)	4
1025	LBCS/LBLO	↓	5(6)	4
1026	LBNE	↓	5(6)	4
1027	LBEQ	↓	5(6)	4
1028	LBVC	↓	5(6)	4
1029	LBVS	↓	5(6)	4
102A	LBPL	↓	5(6)	4
102B	LBMI	↓	5(6)	4
102C	LBGE	↓	5(6)	4
102D	LBLT	RELATIVE	5(6)	4

OP	MNEM	MODE	~	#
102E	LBGT	RELATIVE	5(6)	4
102F	LBLE	RELATIVE	5(6)	4
103F	SWI/2	INHERENT	20	2
1083	CMPD	IMMED	5	4
108C	CMPY	↕	5	4
108E	LDY	IMMED	4	4
1093	CMPD	DIRECT	7	3
109C	CMPY	↑	7	3
109E	LDY	↓	6	3
109F	STY	DIRECT	6	3
10A3	CMPD	INDEXED	7 +	3 +
10AC	CMPY	↑	7 +	3 +
10AE	LDY	↓	6 +	3 +
10AF	STY	INDEXED	6 +	3 +
10B3	CMPD	EXTENDED	8	4
10BC	CMPY	↑	6	4
10BE	LDY	↓	7	4
10BF	STY	EXTENDED	7	4

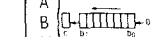
OP	MNEM	MODE	~	#
10CE	LDS	IMMED	4	4
10DE	LDS	DIRECT	6	3
10DF	STS	DIRECT	6	3
10EE	LDS	INDEXED	6 +	3 +
10EF	STS	INDEXED	6 +	3 +
10FE	LDS	EXTENDED	7	4
10FF	STS	EXTENDED	7	4
113F	SWI/3	INHERENT	20	2
1183	CMPU	IMMED	5	4
118C	CMPS	IMMED	5	4
1193	CMPU	DIRECT	7	3
119C	CMPS	DIRECT	7	3
11A3	CMPU	INDEXED	7 +	3 +
11AC	CMPS	INDEXED	7 +	3 +
11B3	CMPU	EXTENDED	8	4
11BC	CMPS	EXTENDED	8	4

INSTRUCTION/ FORMS		INHERENT OP	~	#	DIRECT OP	~	#	EXTENDED OP	~	#	IMMEDIATE OP	~	#	INDEXED[1] OP	~	#	RELATIVE OP	~[5]	#	DESCRIPTION	5 H	3 N	2 Z	1 V	0 C
ABX		3A	3	1																B + X→X (UNSIGNED)	•	•	•	•	•
ADC	ADCA				99	4	2	B9	5	3	89	2	2	A9	4+	2+				A + M + C →A	↕	↕	↕	↕	↕
	ADCB				D9	4	2	F9	5	3	C9	2	2	E9	4+	2+				B + M + C →B	↕	↕	↕	↕	↕
ADD	ADDA				9B	4	2	BB	5	3	8B	2	2	AB	4+	2+				A + M →A	↕	↕	↕	↕	↕
	ADDB				DB	4	2	FB	5	3	CB	2	2	EB	4+	2+				B + M →B	↕	↕	↕	↕	↕
	ADDD				D3	6	2	F3	7	3	C3	4	3	E3	6+	2+				D + M:M +1 →D	•	↕	↕	↕	↕

AND	ANDA				94	4	2	B4	5	3	84	2	2	A4	4 +	2 +				A∧M→A	•	‡	‡	0	•
	ANDB				D4	4	2	F4	5	3	C4	2	2	E4	4 +	2 +				B∧M→B	•	‡	‡	0	•
	ANDCC										1C	3	2							CC∧IMM→CC					7
ASL	ASLA	48	2	1																A	8	‡	‡	‡	‡
	ASLB	58	2	1																B	8	‡	‡	‡	‡
	ASL				08	6	2	78	7	3				68	6 +	2 +				M	8	‡	‡	‡	‡
ASfl	ASRA	47	2	1																A	8	‡	‡	‡	‡
	ASR	57	2	1																B	8	‡	‡	‡	‡
	ASR				07	6	2	77	7	3				67	6 +	2 +				M	8	‡	‡	‡	‡
BCC	BCC																24	3	2	BRANCH C = 0	•	•	•	•	•
	LBCC																10 24	5(6)	4	LONG BRANCH C = 0	•	•	•	•	•
BCS	BCS																25	3	2	BRANCH C = I	•	•	•	•	•
	LBCS																10 25	5(6)	4	LONG BRANCH C = 1	•	•	•	•	•
BEQ	BEQ																27	3	2	BRANCH Z#0	•	•	•	•	•
	LBEQ																10 27	5(6)	4	LONG BRANCH Z#0	•	•	•	•	•
BGE	BGE																2C	3	2	BRANCH >=ZERO	•	•	•	•	•
	LBGE																10 2C	5(6)	4	LONG BRANCH >= ZERO	•	•	•	•	•
BGT	BGT																2E	3	2	BRANCH >ZER0	•	•	•	•	•
	LBGT																10 2E	5(6)	4	LONG BRANCH > ZERO	•	•	•	•	•
BHI	BHI																22	3	2	BRANCH HIGHER	•	•	•	•	•
	LBHI																10 22	5(6)	4	LONG BRANCH HIGHER	•	•	•	•	•
BHS	BHS																24	3	2	BRANCH HIGHER OR SAME	•	•	•	•	•
	LBHS																10 24	5(6)	4	LONG BRANCH HIGHER OR SAME	•	•	•	•	•

INSTRUCTION FORMS		6809 ADDRESSING MODES																		DESCRIPTION	5	3	2	1	0
		INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED[1]			RELATIVE				H	N	Z	V	C
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~[5]	#						
BIT	BITA				95	4	2	B5	5	3	85	2	2	A5	4 +	2 +				BIT TEST A(M∧A)	•	‡	‡	0	•
	BITB				05	4	2	F5	5	3	C5	2	2	E5	4 +	2 +				BIT TEST B(M∧B)	•	‡	‡	0	•
BLE	BLE																2F	3	2	BRANCH ≤ZERO	•	•	•	•	•
	LBLE																10 2F	5(6)	4	LONG BRANCH ≤ ZERO	•	•	•	•	•
BLO	BLO																25	3	2	BRANCH LOWER	•	•	•	•	•
	LBLO																10 25	5(6)	4	LONG BRANCH LOWER	•	•	•	•	•
BLS	BLS																23	3	2	BRANCH LOWER OR SAME	•	•	•	•	•
	LBLS																10 23	5(6)	4	LONG BRANCH LOWER OR SAME	•	•	•	•	•
BLT	BLT																2D	3	2	BRANCH <ZERO	•	•	•	•	•
	LBLT																10 2D	5(6)	4	LONG BRANCH < ZERO	•	•	•	•	•
BMI	BMI																2B	3	2	BRANCH MINUS	•	•	•	•	•
	LBMI																10 2B	5(6)	4	LONG BRANCH MINUS	•	•	•	•	•
BNE	BNE																26	3	2	BRANCH Z = 0	•	•	•	•	•
	LBNE																10 26	5(6)	4	LONG BRANCH Z = 0	•	•	•	•	•
BPL	BPL																2A	3	2	BRANCH PLUS	•	•	•	•	•
	LBPL																10 2A	5(6)	4	LONG BRANCH PLUS	•	•	•	•	•
BRA	BRA																20	3	2	BRANCH ALWAYS	•	•	•	•	•
	LBRA																16	5	3	LONG BRANCH Al WAYS	•	•	•	•	•

BRN	BRN																	21	3	2	BRANCH NEVER	•	•	•	•	•
	LBRN																	10	5	4	LONG BRANCH	•	•	•	•	•
																		21			NEVER					
BSR	BSR																	8D	7	2	BRANCH TO	•	•	•	•	•
																					SUBROUTINE					
	LBSR																	17	9	3	LONG BRANCH TO	•	•	•	•	•
																					SUBROUTINE					
BVC	BVC																	28	3	2	BRANCH V = 0	•	•	•	•	•
	LBVC																	10	5(6)	4	LONG BRANCH	•	•	•	•	•
																		28			V = 0					
BVS	BVS																	29	3	2	BRANCH V = 1	•	•	•	•	•
	LBVS																	10	5(6)	4	LONG BRANCH	•	•	•	•	•
																		29			V = 1					
CLR	CLRA	4F	2	1																	0→A	•	0	1	0	0
	CLRB	5F	2	1																	0→B	•	0	1	0	0
	CLR				0F	6	2	7F	?	3				6F	6 +	2 +					0→M	•	0	1	0	0
CMP	CMPA				91	4	2	B1	5	3	81	2	2	A1	4 +	2 +					COMPARE M	(8)	↕	↕	↕	↕
																					FROM A					
	CMPB				D1	4	2	F1	5	3	C1	2	2	E1	4 +	2 +					COMPARE M	(8)	↕	↕	↕	↕
																					FROM B					
	CMPD				10	7	3	10	8	4	10	5	4	10	7 +	3 +					COMPARE M:M	•	↕	↕	↕	↕
					93			B3			83			A3							+ 1 FROM D					
	CMPS				11	7	3	11	8	4	11	5	4	11	7 +	3 +					COMPARE M:M	•	↕	↕	↕	↕
					9C			BC			8C			AC							+ 1 FROM S					
	CMPU				11	7	3	11	8	4	11	5	4	11	7 +	3 +					COMPARE M:M	•	↕	↕	↕	↕
					93			B3			83			A3							+ 1 FROM U					
	CMPX				9C	6	2	BC	7	3	8C	4	3	AC	6 +	2 +					COMPARE M:M	•	↕	↕	↕	↕
																					+ 1 FROM X					
	CMPY				10	7	3	10	8	4	10	5	4	10	7 +	3 +					COMPARE M:M	•	↕	↕	↕	↕
					9C			BC			8C			AC							+ 1 FROM Y					
COM	COMA	43	2	1																	$\overline{A} \to A$	•	↕	↕	0	1
	COMB	53	2	1																	$\overline{B} \to B$	•	↕	↕	0	1
	COM				03	6	2	73	7	3				63	6 +	2 +					$\overline{M} \to M$	•	↕	↕	0	1
CWAI		3C	20	2																	CC∧IMM→CC:					7
																					Wait for Interrupt					

INSTRUCTION/ FORMS		6809 ADDRESSING MODES																DESCRIPTION	5 H	3 N	2 Z	1 V	0 C
		INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED[1]			RELATIVE						
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~[5]	#					
DAA		19	2	1														Decimal Adjust A	•	‡	‡	0	‡
DEC	DECA	4A	2	1														A − 1 → A	•	‡	‡	‡	•
	DECB	5A	2	1														B − 1 → B	•	‡	‡	‡	•
	DEC				0A	S	2	7A	7	3				6A	6 +	2 +		M − 1 → M	•	‡	‡	‡	•
EOR	EORA				93	4	2	B8	5	3	88	2	2	A8	4 +	2 +		A ∀ M → A	•	‡	‡	0	•
	EORB				D8	4	2	F8	5	3	C8	2	2	E8	4 +	2 +		B ∀ M → B	•	‡	‡	0	•
EXG	R1, R2	1E	8	2														R1 ↔ R2[2]	•	•	•	•	•
INC	INCA	4C	2	1														A + 1 → A	•	‡	‡	‡	•
	INCB	5C	2	1														B + 1 → B	•	‡	‡	‡	•
	INC				0C	6	2	7C	7	3				6C	6 +	2 +		M + 1 → M	•	‡	‡	‡	•
JMP					0E	3	2	7E	4	3				6E	3 +	2 +		EA[3] → PC	•	•	•	•	•
JSR					9D	7	2	BD	8	3				AD	7 +	2 +		JUMP TO SUBROUTINE	•	•	•	•	•
LD	LDA				96	4	2	B6	5	3	86	2	2	A6	4 +	2 +		M → A	•	‡	‡	0	•
	LDB				C6	4	2	F6	5	3	C6	2	2	E6	4 +	2 +		M → B	•	‡	‡	0	•
	LDD				DC	5	2	FC	6	3	CC	3	3	EC	5 +	2 +		M:M + 1 → D	•	‡	‡	0	•
	LDS				10	6	3	10	7	4	10	4	4	10	6 +	3 +		M:M + 1 → S	•	‡	‡	0	•
					CE			FE			CE			EE									
	LDU				OE	5	2	FE	6	3	CE	3	3	EE	5 +	2 +		M:M + 1 → U	•	‡	‡	0	•
	LDX				9E	5	2	BE	6	3	8E	3	3	AE	5 +	2 +		M:M + 1 → X	•	‡	‡	0	•
	LDY				10	6	3	10	7	4	10	4	4	10	6 +	3 +		M:M + 1 → Y	•	‡	‡	0	•
					9E			BE			8E			AE									
LEA	LEAS													32	4 +	2 +		EA[3] → S	•	•	•	•	•
	LEAU													33	4 +	2 +		EA[3] → U	•	•	•	•	•
	LEAX													30	4 +	2 +		EA[3] → X	•	•	‡	•	•
	LEAY													31	4 +	2 +		EA[3] → Y	•	•	‡	•	•
LSL	LSLA	48	2	1														A	•	‡	‡	‡	‡
	LSLB	58	2	1														B 	•	‡	‡	‡	‡
	LSL				08	6	2	78	7	3				68	6 +	2 +		M	•	‡	‡	‡	‡

LSR	LSRA	44	2	1												A	•	0	‡	•	‡
	LSRB	54	2	1												B	•	0	‡	•	‡
	LSR				04	6	2	74	7	3				64	6 +	M	•	0	‡	•	‡
MUL		3D	11	1											2 +	A × B → D (UNSIGNED)	•	•	‡	•	9
NEG	NEGA	40	2	1												$\overline{A}$ + 1 → A	8	‡	‡	‡	‡
	NEGB	50	2	1												$\overline{B}$ + 1 → B	8	‡	‡	‡	‡
	NEG				00	6	2	70	7	3				60	6 +	$\overline{M}$ + 1 → M	8	‡	‡	‡	‡
NOP		12	2	1											2 +	NO OPERATION	•	•	•	•	•
OR	ORA				9A	4	2	BA	5	3	8A	2	2	AA	4 +	A VM → A	•	‡	‡	0	•
	ORB				DA	4	2	LA	5	3	CA	2	2	FA	4 +	B VM → B	•	‡	‡	0	•
	ORCC										1A	3	2		2 +	CC VIMM → A					7
PSH	PSHS	34	5 + [4]	2												PUSH REGISTERS ON S STACK	•	•	•	•	•
	PSHU	36	5 + [4]	2												PUSH REGISTERS ON U STACK	•	•	•	•	•
PUL	PULS	35	5 + [4]	2												PULL REGISTERS FROM S STACK	•	•	•	•	•
	PULU	37	5 + [4]	2												PULL REGISTERS FROM U STACK	•	•	•	•	•
ROL	ROLA	49	2	1												A	•	‡	‡	‡	‡
	ROLB	59	2	1												B	•	‡	‡	‡	‡
	ROL				09	6	2	79	7	3				69	6 +	M	•	‡	‡	‡	‡
ROH	RORA	46	2	1												A	•	‡	‡	•	‡
	RORB	56	2	1												B	•	‡	‡	•	‡
	ROR				06	6	2	76	7	3				66	6 +	M	•	‡	‡	•	‡
RTI		3B	6/15	1											2 +	RETURN FROM INTERRUPT					7
RTS		39	5	1												RETURN FROM SUBROUTINE	•	•	•	•	•
SBC	SBCA				92	4	2	B2	5	3	82	2	2	A2	4 +	A − M − C → A	8	‡	‡	‡	‡
	SBCB				D2	4	2	F2	5	3	C2	2	2	E2	4 +	B − M − C → B	8	‡	‡	‡	‡
SEX		10	2	1											2 +	SIGN EXTEND B INTO A	•	‡	‡	0	•

INSTRUCTION/ FORMS		6809 ADDRESSING MODES: INHERENT OP	~	#	DIRECT OP	~	#	EXTENDED OP	~	#	IMMEDIATE OP	~	#	INDEXED[1] OP	~	#	RELATIVE OP	~5	#	DESCRIPTION	5 H	3 N	2 Z	1 V	0 C
ST	STA				97	4	2	B7	5	3				A7	4 +	2 +				A→M	•	‡	‡	0	•
	STB				D7	4	2	F7	5	3				E7	4 +	2 +				B→M	•	‡	‡	0	•
	STD				DD	5	2	FD	6	3				ED	5 +	2 +				D→M:M + 1	•	‡	‡	0	•
	STS				10	6	3	10	7	4				10	6 +	3 +				S→M:M + 1	•	‡	‡	0	•
					DF			FF						EF											
	STU				DF	5	2	FF	6	3				EF	5 +	2 +				U→M:M + 1	•	‡	‡	0	•
	STX				9F	5	2	BF	6	3				AF	5 +	2 +				X→M:M + 1	•	‡	‡	0	•
	STY				10	6	3	10	7	4				10						Y→M:M + 1	•	‡	‡	0	•
					9F			BF						AF	6 +	3 +									
SUB	SUBA				90	4	2	B0	5	3	80	2	2	A0	4 +	2 +				A − M→A	8	‡	‡	‡	‡
	SUBB				D0	4	2	F0	5	3	C0	2	2	E0	4 +	2 +				B − M→B	8	‡	‡	‡	‡
	SUBD				93	6	2	B3	7	3	83	4	3	A3	6 +	2 +				D − M:M + 1→D	•	‡	‡	‡	‡
SWI	SWI[6]	3F	19	1																SOFTWARE INTERRUPT 1	•	•	•	•	•
	SWI2[6]	10	20	2																SOFTWARE INTERRUPT 2	•	•	•	•	•
		3F																							
	SWI3[6]	11	20	2																SOFTWARE INTERRUPT 3	•	•	•	•	•
		3F																							
SYNC		13	>=2	1																SYNCHRONIZE TO INTERRUPT	•	•	•	•	•
TFR	R1, R2	1F	6	2																R1→R2[2]	•	•	•	•	•
TST	TSTA	4D	2	1																TEST A	•	‡	‡	0	•
	TSTB	5D	2	1																TEST B	•	‡	‡	0	•
	TST				0D	6	2	7D	7	3				6D	6 +	2 +				TEST M	•	‡	‡	0	•

INDEXED ADDRESSING MODES

		NON INDIRECT				INDIRECT			
TYPE	FORMS	ASSEMBLER FORM	POST-BYTE OP CODE	+ ~	+ #	ASSEMBLER FORM	POST-BYTE OPCODE	+ ~	+ #
CONSTANT OFFSET FROM R	NO OFFSET	R	1RR00100	0	0	[,R]	1RR10100	3	0
	5 BIT OFFSET	n,R	0RRnnnnn	1	0	DEFAULTS TO 8-BIT			
	8 BIT OFFSET	n,R	1RR01000	1	1	[n,R]	1RR11000	4	1
	16 BIT OFFSET	n,R	1RR01001	4	2	[n,R]	1RR11001	7	2
ACCUMULATOR OFFSET FROM R	A—REGISTER OFFSET	A,R	1RR00110	1	0	[A,R]	1RR10110	4	0
	B—REGISTER OFFSEI	B,R	1RR00101	1	0	[B,R]	1RR10101	4	0
	D—REGISTER OFFSET	D,R	1RR01011	4	0	[D,R]	1RR11011	7	0
AUTO INCREMENT/DECREMENT R	INCREMENT BY 1	,R+	1RR00000	2	0	NOT ALLOWED			
	INCREMENT BY 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	DECREMENT BY 1	,-R	1RR00010	2	0	NOT ALLOWED			
	DECREMENT BY 2	,--R	1RR00011	3	0	[,--R]	1RR10011	6	0
CONSTANT OFFSET FROM PC	8 BIT OFFSET	n,PCR	1XX01100	1	1	[n,PCR]	1XX11100	4	1
	16 BIT OFFSET	n,PCR	1XX01101	5	2	[n,PCR]	1XX11101	8	2
EXTENDED INDIRECT	16 BIT ADDRESS	—	—	—	—	[n]	10011111	5	2

R = X, Y, U, or S
X = DON'T CARE

X = 00 U = 10
Y = 01 S = 11

NOTES:

1. GIVEN IN THE TABLE ARE THE BASE CYCLES AND BYTE COUNTS TO DETERMINE THE TOTAL CYCLES AND BYTE COUNTS ADD THE VALUES FROM THE '6809 INDEXING MODES' TABLE
2. R1 AND R2 MAY BE ANY PAIR OF 8 BIT OR ANY PAIR OF 16 BIT REGISTERS
   THE 8 BIT REGISTERS ARE: A, B, CC, DP
   THE 16 BIT REGISTERS ARE: X, Y, U, S, D, PC

INDEXED ADDRESSING MODES

3 EA IS THE EFFECTIVE ADDRESS.

4 THE PSH AND PUL INSTRUCTIONS REQUIRE 5 CYCLES PLUS 1 CYCLE FOR EACH **BYTE** PUSHED OR PULLED.

5 5(6) MEANS: 5 CYCLES IF BRANCH NOT TAKEN. 6 CYCLES IF TAKEN.

6 SWI SETS I & F BITS. SWI2 AND SWI3 DO NOT AFFECT I & F.

7 CONDITIONS COOES SET AS A DIRECT RESULT OF THE INSTRUCTION.

8 VALUE OF HALF-CARRY FLAG IS UNDEFINED.

9 SPECIAL CASE - CARRY SET IF b7 IS SET.

LEGEND:

OP	OPERATION CODE (HEXADECIMAL):	→	TRANSFER INTO;	•	NOT AFFECTED
~	NUMBER OF MPU CYCLES:	H	HALF-CARRY FROM BIT 3:	CC	CONDITION CODE REGISTER
#	NUMBER OF PROGRAM BYTES.	N	NEGATIVE (SIGN BIT)	:	CONCATENATION
+	ARITHMETIC PLUS:	Z	ZERO (BYTE)	V	LOGICAL OR
-	ARITHMETIC MINUS:	V	OVERFLOW 2'S COMPLEMENT	∧	LOGICAL AND
*	MULTIPLY	C	CARRY FROM BIT 7		LOGICAL EXCLUSIVE OR
$\overline{M}$	COMPLEMENT OF M:	↕	TEST AND SET IF TRUE. CLEARED OTHERWISE		

Instruction Index

MNEMONIC	ADDRESSING MODE	OP CODE	CYCLES	PAGE NUMBER
ABX	INHERENT	3A	3	59
ADCA	IMMEDIATE	89	2	60
	DIRECT	99	4	
	INDEXED	A9	4+	
	EXTENDED	B9	5	
ADCB	IMMEDIATE	C9	2	60
	DIRECT	D9	4	
	INDEXED	E9	4+	
	EXTENDED	F9	5	
ADDA	IMMEDIATE	8B	2	61
	DIRECT	9B	4	
	INDEXED	AB	4+	
	EXTENDED	BB	5	
ADDB	IMMEDAITE	CB	2	61
	DIRECT	DB	4	
	INDEXED	EB	4+	
	EXTENDED	FB	5	
ADDD	IMMEDIATE	C3	4	61
	DIRECT	D3	6	
	INDEXED	E3	6+	
	EXTENDED	F3	7	
ANDA	IMMEDIATE	84	2	63
	DIRECT	94	4	
	INDEXED	A4	4+	
	EXTENDED	B4	5	
ANDB	IMMEDIATE	C4	2	63
	DIRECT	D4	4	
	INDEXED	E4	4+	
	EXTENDED	C4	5	
ANDCC	IMMEDIATE	1C	3	63
ASLA	ACCUMULATOR	48	2	64
ASLB	ACCUMULATOR	58	2	64
ASL	DIRECT	08	6	64
	EXTENDED	78	7	
	INDEXED	68	6+	
ASR	INHERENT	57	2	65

MNEMONIC	ADDRESSING MODE	OP CODE	CYCLES	PAGE NUMBER
	DIRECT	07	6	
	EXTENDED	77	7	
	INDEXED	67	6+	
ASRA	INHERENT	47	2	66
BCC	RELATIVE	24	3	66
LBCC	LONG RELATIVE	10	5(6)	66
BCS	RELATIVE	25	3	67
LBCS	LONG RELATIVE	10	5(6)	67
BEQ	RELATIVE	27	3	67
LBEQ	LONG RELATIVE	10	5(6)	68
BGE	RELATIVE	2C	3	68
LBGE	LONG RELATIVE	10	5(6)	68
BGT	RELATIVE	2E	3	68
LBGT	LONG RELATIVE	10	5(6)	69
BHI	RELATIVE	22	3	69
LBHI	LONG RELATIVE	10	5(6)	69
BHS	RELATIVE	24	3	70
L3HS	LONG RELATIVE	10	5(6)	70
BIT A	DIRECT	95	4	71
	EXTENDED	B5	5	
	IMMEDIATE	85	2	
	INDEXED	A5	4+	
BIT B	DIRECT	D5	4	71
	EXTENDED	F5	5	
	IMMEDIATE	C5	2	
	INDEXED	E5	4+	
BLE	RELATIVE	2F	3	71
LBLE	LONG RELATIVE	10	5(6)	72
BLO	RELATIVE	25	3	72
LBLO	LONG RELATIVE	10	5(6)	72
BLS	RELATIVE	23	3	73
LBLS	LONG RELATIVE	10	5(6)	73
BLT	RELATIVE	2D	3	73
LBLT	LONG RELATIVE	10	5(6)	73
BMI	RELATIVE	2B	3	74
LBMI	LONG RELATIVE	10	5(6)	74
BNE	RELATIVE	26	3	74
LBNE	LONG RELATIVE	10	5(6)	75
BPL	RELATIVE	2A	3	75
LBPL	LONG RELATIVE	10	5(6)	75
BRA	RELATIVE	20	3	75
LBRA	LONG RELATIVE	16	5	75
BRN	RELATIVE	21	3	76
LBRN	LONG RELATIVE	10	5	76
BSR	RELATIVE	8D	7	76
LBSR	LONG RELATIVE	17	9	76
BVC	RELATIVE	28	3	77
LBVC	LONG RELATIVE	10	5(6)	77
BVS	RELATIVE	29	3	78
LBVS	LONG RELATIVE	10	5(6)	78
CLRA	INHERENT	4F	2	78
CLRB	INHERENT	5F	2	78
CLR	DIRECT	0F	6	79

MNEMONIC	ADDRESSING MODE	OP CODE	CYCLES	PAGE NUMBER
	EXTENDED	7F	7	
	INDEXED	6F	6+	
CMPA	DIRECT	91	4	79
	EXTENDED	B1	5	
	IMMEDIATE	81	2	
	INDEXED	A1	4+	
CMPB	DIRECT	D1	4	79
	EXTENDED	F1	5	
	IMMEDIATE	C1	2	
	INDEXED	E1	4+	
CMPD	DIRECT	10 93	7	80
	EXTENDED	10 B3	8	
	IMMEDIATE	10 83	5	
	INDEXED	10 A3	7+	
CMPS	DIRECT	11 9C	7	80
	EXTENDED	11 BC	8	
	IMMEDIATE	11 8C	5	
	INDEXED	11 AC	7+	
CMPU	DIRECT	11 93	7	81
	EXTENDED	11 B3	8	
	IMMEDIATE	11 83	5	
	INDEXED	11 A3	7+	
CMPX	DIRECT	9C	6	81
	EXTENDED	BC	7	
	IMMEDIATE	8C	4	
	INDEXED	AC	6+	
CMPY	DIRECT	10C 9C	7	81
	EXTENDED	10 BC	8	
	IMMEDIATE	10 8C	5	
	INDEXED	10 AC	7+	
COMA	INHERENT	43	2	82
COMB	INHERENT	53	2	82
COM	DIRECT	03	6	82
	EXTENDED	73	7	
	INDEXED	63	6+	
CWAI	INHERENT	3C	20	82

MNEMONIC	ADDRESSING MODE	OP CODE	CYCLES	PAGE NUMBER
DAA	INHERENT	19	2	84
DECA	INHERENT	4A	2	85
DECB	INHERENT	5A	2	85
DEC	DIRECT	OA	6	85
	EXTENDED	7A	7	
	INDEXED	6A	6+	
EORA	DIRECT	9B	4	85
	EXTENDED	B8	5	
	IMMEDIATE	88	2	
	INDEXED	A8	4+	
EORB	DIRECT	D8	4	86
	EXTENDED	F8	5	
	IMMEDIATE	C8	2	
	INDEXED	E8	4+	
EXG R1, R2	INHERENT	1E	7	86
INCA	INHERENT	4C	2	87
INCB	INHERENT	5C	2	87
INC	DIRECT	0C	6	87
	EXTENDED	7C	7	
	INDEXED	6C	6+	
JMP	DIRECT	OE	3	87
	EXTENDED	7E	4	
	INDEXED	6E	3+	
JSR	DIRECT	9D	7	88
	EXTENDED	BD	8	
	INDEXED	AD	7+	
LDA	DIRECT	96	4	88
	EXTENDED	B6	5	
	IMMEDIATE	86	2	
	INDEXED	A6	4+	
LDB	DIRECT	D6	4	88
	EXTENDED	F6	5	
	IMMEDIATE	C6	2	
	INDEXED	E6	4+	
LDD	DIRECT	DC	5	89
	EXTENDED	FC	6	
	IMMEDIATE	CC	3	
	INDEXED	EC	5+	
LDS	DIRECT	10 DE	6	89
	EXTENDED	10 FE	7	
	IMMEDIATE	10 CE	4	
	INDEXED	10 EE	6+	
LDU	DIRECT	DE	5	89
	EXTENDED	FE	6	
	IMMEDIATE	CE	3	
	INDEXED	EE	5+	
LDX	DIRECT	9E	5	89
	EXTENDED	BE	6	

MNEMONIC	ADDRESSING MODE	OP CODE	CYCLES	PAGE NUMBER
	IMMEDIATE	8E	3	
	INDEXED	AE	5+	
LDY	DIRECT	10 9F	6	89
	EXTENDED	10 BE	7	
	IMMEDIATE	10 8E	4	
	INDEXED	10 AE	6+	
LEAS	RELATIVE	32	4 +	90
LEAU	RELATIVE	33	4+	90
LEAX	RELATIVE	30	4 +	90
LEAY	RELATIVE	31	4+	90
LSLA	INHERENT	48	2	91
LSLB	INHERENT	58	2	91
LSL	DIRECT	08	6	91
	EXTENDED	78	7	
	INDEXED	68	6+	
LSRA	INHERENT	44	2	92
LSRB	INHERENT	54	2	92
LSR	DIRECT	04	6	92
	EXTENDED	74	7	
	INDEXED	64	6+	
MUL	INHERENT	3D	11	93
NEGA	INHERENT	40	2	93
NEGB	INHERENT	50	2	93
NEG	DIRECT	00	6	93
	EXTENDED	70	7	
	INDEXED	60	6+	
NOP	INHERENT	12	2	94
ORA	DIRECT	9A	4	94
	EXTENDED	BA	5	
	IMMEDIATE	8A	2	
	INDEXED	AA	4 +	
ORB	DIRECT	DA	4	94
	EXTENDED	FA	5	
	IMMEDIATE	CA	2	
	INDEXED	EA	4 +	
ORCC	IMMEDIATE	1A	3	95
PSHS	INHERENT	34	5+4	96
PSHU	INHERENT	36	5+4	96
PULS	INHERENT	35	5+4	96
PULU	INHERENT	37	5+4	97
ROLA	INHERENT	49	2	98
ROLB	INHERENT	59	2	98
ROL	DIRECT	09	6	98
	EXTENDED	79	7	
	INDEXED	69	6 +	
RORA	INHERENT	46	2	99
RORB	INHERENT	56	2	99
ROR	DIRECT	06	6	99

MNEMONIC	ADDRESSING MODE	OP CODE	CYCLES	PAGE NUMBER
	EXTENDED	76	7	
	INDEXED	66	6+	
RTI	INHERENT	3B	6/15	100
RTS	INHERENT	39	5	100
SBCA	DIRECT	92	4	100
	EXTENDED	B2	5	
	IMMEDIATE	82	2	
	INDEXED	A2	4+	
SBCB	DIRECT	D2	4	101
	EXTENDED	F2	5	
	IMMEDIATE	C2	2	
	INDEXED	E2	4+	
SEX	INHERENT	1D	2	101
STA	DIRECT	97	4	102
	EXTENDED	B7	5	
	INDEXED	A7	4+	
STB	DIRECT	D7	4	102
	EXTENDED	F7	5	
	INDEXED	E7	4+	
STD	DIRECT	DD	5	102
	EXTENDED	FD	6	
	INDEXED	ED	5+	
STS	DIRECT	10 DF	6	102
	EXTENDED	10 FF	7	
	INDEXED	10 EF	6+	
STU	DIRECT	DF	5	103
	EXTENDED	FF	6	
	INDEXED	EF	5+	
STX	DIRECT	9F	5	103
	EXTENDED	BF	6	
	INDEXED	AF	5+	
STY	DIRECT	10 9F	6	103
	EXTENDED	10 DF	7	
	INDEXED	10 AF	6+	
SUBA	DIRECT	90	4	104
	EXTENDED	B0	5	
	IMMEDIATE	80	2	
	INDEXED	A0	4+	
SUBB	DIRECT	D0	4	104
	EXTENDED	F0	5	
	IMMEDIATE	C0	2	
	INDEXED	E0	4+	
SUBD	DIRECT	93	6	104
	EXTENDED	B3	7	
	IMMEDIATE	83	4	
	INDEXED	A3	6+	

MNEMONIC	ADDRESSING MODE	OP CODE	CYCLES	PAGE NUMBER
SWI	INHERENT	3F	19	105
SWI2	INHERENT	10 3F	20	105
SWI3	INHERENT	11 3F	20	106
SYNC	INHERENT	13	>=2	107
TFR	INHERENT	1F	7	108
TSTA	INHERENT	4D	2	109
TSTB	INHERENT	5D	2	109
TST	DIRECT	OD	6	109
	EXTENDED	7D	7	
	INDEXED	6D	6+	

Index

Edited by Robert E. Ostrander