21798

6809 MICROCOMPUTER PROGRAMMING & INTERFACING

WITH EXPERIMENTS

BY ANDREW C. STAUGAARD, JR.

BLACKSBURG CONTINUING EDUCATION SERIES™
edited by Titus, Titus & Larsen

The Blacksburg Continuing Education™ Series

The Blacksburg Continuing Education Series™ of books provide a Laboratory—or experiment-oriented approach to electronic topics. Present and forthcoming titles in this series include:

- Basic Business Software
- Circuit Design Programs for the TRS-80
- DBUG: An 8080 Interpretive Debugger
- Design of Active Filters, With Experiments
- Design of Op-Amp Circuits, With Experiments
- Design of Phase-Locked Loop Circuits, With Experiments
- Design of Transistor Circuits, With Experiments
- Design of VMOS Circuits, With Experiments
- 8080/8085 Software Design (2 Volumes)
- 8085A Cookbook
- 555 Timer Applications Sourcebook, With Experiments
- Guide to CMOS Basics, Circuits, & Experiments
- How to Program and Interface the 6800
- Microcomputer—Analog Converter Software and Hardware Interfacing
- Microcomputer Interfacing With the 8255 PPI Chip
- NCR Basic Electronics Course, With Experiments
- NCR Data Communications Concepts
- NCR Data Processing Concepts Course
- NCR EDP Concepts Course
- PET Interfacing
- Programming and Interfacing the 6502, With Experiments
- 6502 Software Design
- 6801, 68701, and 6803 Microcomputer Programming and Interfacing
- 6809 Microcomputer Programming & Interfacing, With Experiments
- TEA: An 8080/8085 Co-Resident Editor/Assembler
- TRS-80 Interfacing (2 Volumes)

In most cases, these books provide both text material and experiments, which permit one to demonstrate and explore the concepts that are covered in the book. These books remain among the very few that provide step-by-step instructions concerning how to learn basic electronic concepts, wire actual circuits, test microcomputer interfaces, and program computers based on popular microprocessor chips. We have found that the books are very useful to the electronic novice who desires to join the "electronics revolution," with minimum time and effort.

Additional information about the "Blacksburg Group" is presented inside the rear cover.

Jonathan A. Titus, Christopher A. Titus, and David G. Larsen
"The Blacksburg Group"

Bug symbol trademark Nanotran, Inc., Blacksburg, VA 24060

6809 Microcomputer Programming & Interfacing, With Experiments

by

Andrew C. Staugaard, Jr.

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

FIRST EDITION
FIRST PRINTING—1981

International Standard Book Number: 0-672-21798-6
Library of Congress Catalog Card Number: 81-50567

Printed in the United States of America.

Preface

Welcome to the world of *advanced* microprocessors! In the early seventies we witnessed the dawn of a second industrial revolution with the introduction of first-generation programmable logic devices. These "smart" devices on a single piece of sand (silicon chip) were appropriately called *microprocessors*. They revolutionized the engineering of many everyday products, from toys and appliances to the automobile and large computer systems.

In the past few years the microprocessor chip industry has *exploded* into a multibillion dollar business. As stated in the June 30, 1980, issue of *Newsweek* magazine, "The explosion is just beginning. In 1979, the world market for microelectronics topped $11 billion. Over the next five years, chip sales are expected to grow by at least 20 percent annually, and the market for microprocessors 'computers on a chip' will expand by *50* percent each year—even though the chips themselves and the computing power they represent are diving in price."

Two large microprocessor application markets have emerged—the *dedicated* market and the *systems* market. The resources of the first- and second-generation devices, such as the 6800, 8080, and Z-80, were made to satisfy both market applications. Each market, however, requires separate microprocessor features for efficient utilization of the device. For example, the dedicated market requires a device which incorporates many functions such as CPU, R/W memory, ROM/EPROM, timer, serial i/o, etc., onto *one* chip to minimize the chip count for such applications as the automobile, appliances, machine tool control, toys, etc. The systems market, on the other hand, requires a very powerful software device such that high-level language programming can be efficiently implemented. Most microprocessor chip manufacturers have taken these two directions in their newer-generation chip designs. In the Motorola family the 6801 and 6805 satisfy the dedicated market applications, with the 6809 and 68000 having been designed particularly for the systems market.

The 6809 is a *high-performance* 8-bit microprocessor. It has many very powerful software features which are particularly useful for high-level language (Pascal, FORTRAN, BASIC, COBOL, etc.) implementation. In fact, as you are about to discover, the 6809 approaches the performance of many 16-bit devices, such as the 8086, Z-8000, and 68000, without the inherent overhead costs required to engineer such a 16-bit system. Flexible 8-bit devices, like the 6809, will be around for many years to come despite the onslaught of the 16-bitters, since many applications do not require such high performance. In addition, when 16-bit systems do become common, they

This book is meant to be a tutorial type of text for a first exposure
will rely on 8-bit devices to perform many dedicated tasks such as peripheral control, data acquisition, etc.
to the 6809 or to high-performance microprocessors in general. I am confident that you will also find it extremely valuable as a "cookbook" type aid when you are working with the 6809. Since the 6809 is a "souped up" 6800, a basic understanding of the 6800 will be assumed throughout the text. If a review is needed, you may wish to consult my *How to Program and Interface the 6800,* published by Howard W. Sams & Co., Inc.

A set of objectives is provided in the first part of each chapter, with review questions and answers provided at the end of each chapter. There are also numerous examples that illustrate the text. I encourage you to study the examples in detail, since many of the important software concepts are demonstrated within the example programs. I also encourage you to pay particular attention to Chapter 2, "6809 Addressing Modes." You will find that the *secret* to understanding the 6809 software concepts is understanding its 19 addressing modes.

Finally, I would like to express my appreciation to Motorola Semiconductor Products at Austin, Texas, and Phoenix, Arizona, for their technical assistance and permission to use their 6809 documentation in this text.

Andrew C. Staugaard, Jr.

To my mother and father, who provided me the good home life and education required to be successful in today's world. And to one of my best friends, my father in law, Zane Benefiel, whose encouragement in my early professional days led to the completion of this and two previous manuscripts.

Contents

CHAPTER 6

CHAPTER 7

APPENDIX A

APPENDIX B

APPENDIX C

APPENDIX D

CHAPTER 1

Fundamental 6809 Concepts and Chip Structure

INTRODUCTION

As you will discover in this and subsequent chapters, the 6809 is one of the most powerful 8-bit processors to come on the market to date. In conceiving the 6809 the Motorola design engineers wanted to maintain compatibility with the popular 6800 at some level, yet "soup up" the 6800 architecture and instruction set to approach the performance of a 16-bit processor, such as the 68000 or Z-8000. From user surveys Motorola concluded that many customers desired such performance from an 8-bit, 40-pin device. It was found that many users did not want to pay the price for conversion to a 16-bit device with up to 64 pins if a high-performance 8-bit device were available. Therefore, the 6809 was designed to approach 16-bit performance at minimum cost to the user.

In this chapter we begin by discussing the 6809 evolution and design philosophy. You will see that the 6809 has been designed primarily for the systems market, where high-level language implementation is common. Throughout the discussion, comparisons will be made between the 6809, 6800, and other competitive processors. In addition, as an introduction to subsequent chapters, this chapter will summarize the 6809's architecture, software, and hardware. It is important that you understand what improvements have been made in the 6809 over the 6800 since this will enable you to better understand the material to follow.

OBJECTIVES

At the end of this chapter you will be able to do the following:

- Explain the evolution of the 6800 family.
- Understand the design philosophy that created the 6809.
- State the differences between the 6809, 68A09, 68B09, and 6809E.
- List the additional registers present in the 6809 that do not exist in the 6800.
- Describe the two additional condition code flags available in the 6809.
- Define indirect addressing.
- Compare 6809 branching to 6800 branching.
- Explain what is meant by "memory paging" and how this is accomplished with the 6809.
- Describe the fundamental hardware differences between the 6809 and 6800.
- Explain the difference between a standard interrupt request and a fast interrupt request.

6809 EVOLUTION AND DESIGN PHILOSOPHY

As stated in the introduction to this chapter the 6809 was designed to upgrade or "soup up" the 6800 to be superior to any 8-bit microprocessor. Also, Motorola wanted to capitalize on their customers' familiarity with the 6800 so that exposure to the 6809 would not create severe learning problems for those 6800 users. Therefore this design philosophy dictates that the fundamental 6800 architecture be used as a basis for the 6809 architecture and that software compatibility be available at some level. As you will discover in Chapter 3 this compatibility exists at the source code (mnemonic) level and *not* the object code (op-code) level. You will also discover that the 6809 *does not* contain dozens of new instructions. However, it uses over three times as many addressing modes as the 6800 to provide more efficient utilization of the existing instructions. The power of a processor is not a function of the number of unique instructions available in its instruction set. The *real* power of a processor lies in how many different ways a given instruction can operate on the same data and also how the given instruction set can operate on different data in the same manner. This flexible instruction power is provided by the different addressing modes available to the fundamental instruction set. The 6809 uses its 19 addressing modes in conjunction with 59 fundamental instructions to provide a total of *1464* unique operations. Motorola believes that the 6809 contains the most powerful addressing modes available in any microprocessor to date.

Since the 6809 is primarily designed for the systems market, program position independence, program re-entrancy, and easy implementation of block-structured high-level languages (such as Pascal) were also prime design considerations. These terms have the following meanings:

- *program position independence*—that quality of a program to execute properly when placed anywhere within the memory address map. Thus the program is *independent* of its position within the memory map.
- *program re-entrancy*—that quality of a program which allows a subroutine to be shared by several tasks concurrently, without destroying the return addresses by nesting routines.

With position independence, programs can be loaded from a mass storage disc and located anywhere within R/W memory without requiring the use of a relocating loader routine. In addition, position independence will allow ROM programs to be written for general distribution. The user can assign any arbitrary set of addresses to the ROM since the program execution is "independent" of its position within the memory map. This will eliminate the necessity for full ROM address decoding and will also allow the user to locate the ROM such that it will not interfere with other software. As you will see shortly, the advanced 6809 architecture also facilitates the use of modular programming. Such programming will allow the system software designer to divide a project up into modular programs which can be designed and tested independently before being incorporated into the final system design. This same architecture allows programs to be structured and interrupted in any part of the address map and still execute properly on return, thus satisfying the program re-entrancy design goal. In addition, high-level block-structured languages, such as Pascal, BASIC, FORTRAN, and COBOL, can be compiled into more efficient and faster-running machine code than was possible with earlier processors.

The 6800 family evolution scheme is shown in Fig. 1-1. Note the position of the 6809. As far as central processing unit (CPU) performance is concerned the 6809 is the most advanced *processor* in the Motorola 8-bit family. However, the 6809, by itself, is a microprocessor and *not* a microcomputer. It requires external R/W memory and ROM to function as a microcomputer. Therefore a minimum 6809 system would consist of three chips. Recall from our discussion in the Preface that microprocessor/microcomputer applications take two directions: small, dedicated applications and systems applications. In the 8-bit Motorola family the 6809 satisfies the needs of the latter, while the 6801, 6802, 6803, and 6805 were designed for the dedicated applications. These four devices all contain various amounts of R/W memory, ROM, and parallel/serial i/o capabilities. They are therefore more advanced as far as chip integration is concerned since some or all of these capabilities are integrated into the

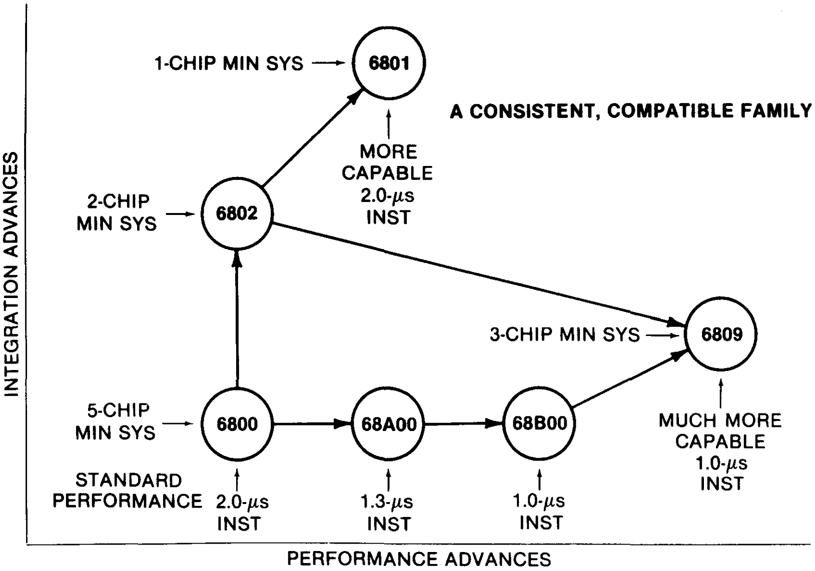


Fig. 1-1. M6800 family evolution.

same chip, along with the CPU. For example, the 6801 contains an enhanced 6800 CPU, R/W memory, ROM, timer, parallel interfacing ports, and a serial port, all on one 40-pin chip. Hence, we say that the 6801 is a "single-chip" microcomputer.* The 6809, however, is a far more advanced processor, approaching 16-bit performance.

As you will see in Chapter 6, the standard 6809 does contain an on-chip clock/oscillator and therefore only requires an external crystal to provide the clock signal. The standard 6809 operates at 1 MHz. However, it is also available in 1.5-MHz and 2.0-MHz versions: the 68A09 and 68B09, respectively. In addition, there is an off-chip clock version of the 6809 available, the 6809E.

Now let's compare the performance of the 6809 to the 6800 and some other well-known processors. Compared to the 6800, the 6809 boasts the following performance:

- 72 percent decrease in the number of instructions required for a program compared to similar 6800 programs.
- 58 percent decrease in the required program memory compared to 6800 programs.
- 167 percent increase in 6809 processor throughput compared to 6800 throughput.

* For a detailed discussion of the 6801/68701 and 6803 consult Staugaard, A.C. *6801/68701 and 6803 Microcomputer Programming and Interfacing,* published by Howard W. Sams & Co., Inc., Indianapolis.

Motorola claims that these statistics allow the user to achieve 2.5 to 5 times the performance from a 6809 system compared with a similar 6800-based system. Of course, the exact amount of increased performance depends on how efficiently the increased capabilities of the 6809 are utilized and on the specific application.

Comparisons of the 6809 to the 6800 and other processors are summarized in Fig. 1-2 and in Tables 1-1 through 1-3. Note, especially from Fig. 1-2, that the 6809 approaches 16-bit performance. The comparison values in these tables and Fig. 1-2 were supplied by Motorola and thus tend to illustrate the better aspects of the 6809 over the other processors.

6809 IMPROVEMENTS

The increased performance of the 6809 over other 8-bit processors is made possible by specific improvements in architecture, software, and hardware. Each of the following improvements over the 6800 is discussed in detail in subsequent chapters; however, we will summarize them here.

Architectural Improvements

Compared with the 6800 architecture the 6809 adds an 8-bit register and three 16-bit registers as shown in Fig. 1-3. The additional 8-bit register is the direct page register, which will allow you to use the direct addressing mode anywhere within the 6809 memory map. (Recall that direct addressing is limited to the first 256 bytes of memory with the 6800.) The three additional 16-bit registers include a 16-bit accumulator, index register, and stack pointer. The additional accumulator is referred to as accumulator D and is simply the concatenation of the two 8-bit accumulators, A and B. The additional index register is called the Y index register (Y) and the additional stack pointer is referred to as the user stack pointer (U). The two index registers, X and Y, will also function as pointers and the two stack pointers (S and U) can be used for indexing.

In addition, you will note from Fig. 1-3 that all 8 bits of the condition code register (CC) are being used in the 6809. Recall that only the first 6 bits are utilized in the 6800. The functions of the first 6 CC bits (C, V, Z, N, I, H) in the 6809 are identical with those of the 6800. The two additional bits (F and E) are used in conjunction with the 6809's interrupts. The use of these two additional bits will be discussed in detail in subsequent chapters.

These architectural improvements along with the 6809's powerful addressing modes speed processor throughput, since less data movement is required between the internal registers and memory, and

Table 1-1. Relative Processor Execution-Time Comparisons for Eight Software Operations

		I/O Handler	Character Search	Computed Go To	Double Shift Right 5 Bits	Vector Addition 16-Bit Elements	Vector Addition 8-Bit Elements	16 × 16-Bit Multiplication	Move Block (64 Bytes)	Average Execution Time
6809	2.0 MHz	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	1.5 MHz	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
	1.0 MHz	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
Z-80	4.0 MHz	1.4	0.8	2.1	2.7	1.6	1.8	3.3	1.0	1.8
	2.5 MHz	2.2	1.2	3.4	4.4	2.6	2.9	5.2	1.6	2.9
9900	3.0 MHz	2.6	2.3	2.8	1.5	1.7	3.0	0.5	1.6	2.0
6800	2.0 MHz	0.9	1.4	1.9	1.3	3.1	2.8	5.0	3.3	2.4
	1.5 MHz	1.2	1.9	2.5	1.7	4.1	3.7	6.7	4.3	3.3
	1.0 MHz	1.8	2.8	3.7	2.5	6.1	5.5	10	6.5	4.9
8080	3.0 MHz	1.9	1.8	2.8	6.1	2.3	2.7	9.6	2.4	3.7
8085	2.0 MHz	2.8	2.6	4.2	9.1	3.4	4.1	14.3	3.7	5.5

Table 1-2. Actual Processor Execution-Time Comparisons for Eight Software Operations

		I/O Handler	Character Search	Computed Go To	Double Shift Right 5 Bits	Vector Addition 16-Bit Elements	Vector Addition 8-Bit Elements	16 × 16-Bit Multiplication	Move Block (64 Bytes)
6809	2.0 MHz	28	287.5	34.5	15	325	180	82	344.5
	1.5 MHz	37.3	383	46	20	433	240	109.3	459.3
	1.0 MHz	56	575	69	30	650	360	164	689
Z-80	4.0 MHz	38.3	220.5	73.3	41	518	323	267	342
	2.5 MHz	61.3	352.8	117.2	65.6	828.8	516.8	427.2	547.6
9900	3.0 MHz	72	661	98	22	537	537	42	537
6800	2.0 MHz	24.5	404	64.5	19	993.5	498.5	409.5	1123.5
	1.5 MHz	32.7	539	86	25.3	1325	665	546	1498
	1.0 MHz	49	808	129	38	1987	997	819	2247
8080	3.0 MHz	52.7	506.7	96.7	91.3	732	492	784	841
8085	2.0 MHz	79	760	145	137	1098	738	1176	1262

Table 1-3. Summarized Processor Performance Comparisons

Performance Criteria	MC6809	Z-80A	MC6800	8085
Number of Instructions	1.0*	1.56	1.72	2.30
Number of Bytes	1.0	1.31	1.58	1.80
Number of Microseconds	1.0	1.80	2.40	2.20
	(2 MHz)	(4 MHz)	(2 MHz)	(5 MHz)

*Normalized to 1.00 for the MC6809—poorer performance has higher numbers.

they also aid software development since many of the internal registers can be made to perform different functions at different times.

Software Improvements

Improvements in the 6809's architecture have allowed the Motorola designers to make many significant software improvements over

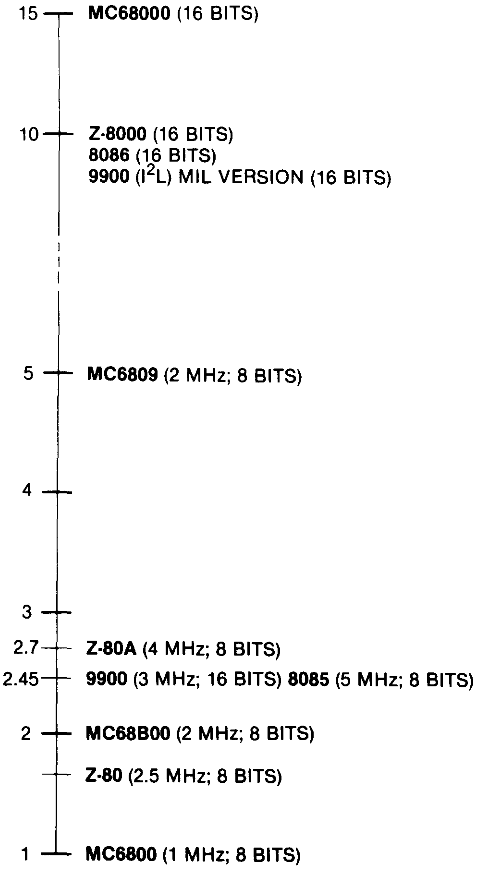


Fig. 1-2. Relative processor execution-time comparisons.

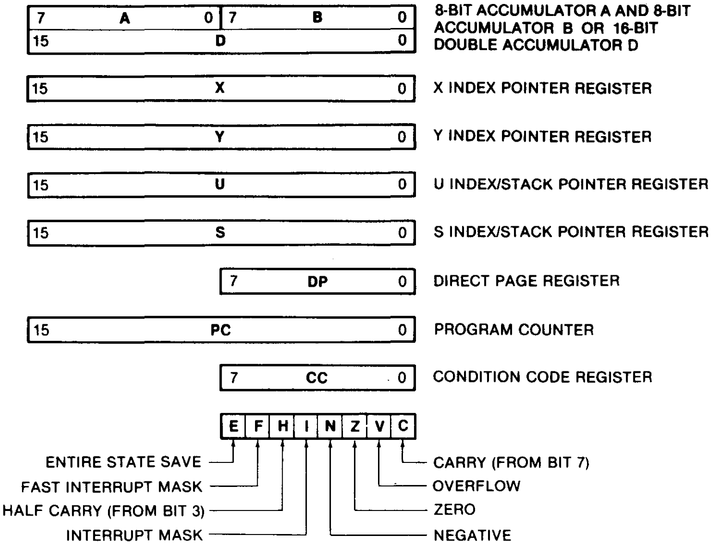


Fig. 1-3. 6809 internal registers.

the 6800. In fact, the 6809 has been termed "the programmer's dream machine." Prior to firm definition of the 6809 instruction set Motorola conducted a survey of 6800 users to determine: (1) if a 16-bit architecture was more desirable than an 8-bit architecture and (2) should 6809 compatibility with the 6800 occur at the object code level or source code level? The survey results indicated that most of the 6800 users felt that an 8-bit architecture was adequate for their future applications. They did not want to pay the price for a new 16-bit device if an 8-bit device could be designed to perform common 16-bit operations, such as load, store, add, subtract, compare, and multiply. As you will discover in Chapter 3, the inclusion of the 16-bit accumulator and associated instructions will allow such 16-bit operations to be performed internally. In answer to the second survey question, almost all of the responses indicated that source code compatibility would be adequate, mainly because they did not foresee using 6800 ROMs in future 6809 systems. Source code compatibility meant that users could take full advantage of their familiarity with the 6800 mnemonic code for assembly language programming and it also allowed the 6809 designers to completely remap or reassign the 6800 op codes to produce more efficient and faster running 6809 programs. Also, source code compatibility would allow any 6800 programs to be processed through a 6809 assembler to produce 6809 code. Therefore, you will see a familiar mnemonic code in Chapters 3, 4, and 5; however, most of the corresponding op codes are different from those of the 6800.

As mentioned earlier, the 6809 adds many new and powerful addressing modes. For example, the indexed mode of addressing can use the four 16-bit indexible registers (X, Y, U, and S) to point to the address of an operand or to the address of the address of an operand. The latter is referred to as *indirect addressing*. These registers can also be incremented and decremented automatically or under program control. The 6809 can even use the program counter to access operands or operand addresses. Furthermore, with the use of the direct page register, you can access any part of the 6809 memory map using direct addressing. You can also *branch* to any part of the program using relative addressing. Recall that with the 6800, you were limited to a plus 127_{10} and minus 128_{10} branch range within the program, since the relative address offset is only 1 byte. However, the 6809 allows for a 2-byte relative address offset and thus permits branching anywhere within the 64K memory map (long branch). This allows position-independent programs to be written for the 6809. We will discuss the idea of position independence in more detail when discussing relative addressing.

Other software features of the 6809 include:

- An 8 × 8 unsigned multiply instruction which generates a 16-bit result.
- 2-byte instructions which will push or pull any or all registers onto or from either stack (U or S).
- 16-bit add, subtract, load, store, and compare instructions which utilize the 16-bit accumulator (D).
- Instructions which permit you to add any of the accumulators (A, B, or D) to any of the index registers and stack pointers (X, Y, S, or U).
- Instructions which permit you to perform exchanges and transfers between any two like-size CPU registers.

Hardware Improvements

Besides the architectural and software improvements of the 6809 over the 6800, many hardware improvements were also made. Most of these improvements involve the 6809's interrupts, control signals, and associated control lines. We will summarize these new hardware features here; however, a complete discussion of the 6809 pin-outs and associated chip operation is provided in Chapter 6.

Like the 6800, the 6809 is a 40-pin device available in both a plastic (P-suffix) and ceramic (L-suffix) package. A bus and control signal diagram for the 6809 is shown in Fig. 1-4. The most obvious change to the chip hardware is the inclusion of an on-chip clock/oscillator for the standard 6809 package. (Recall that the 6809E is an off-chip clock version of the 6809.) The 6809 on-chip clock re-

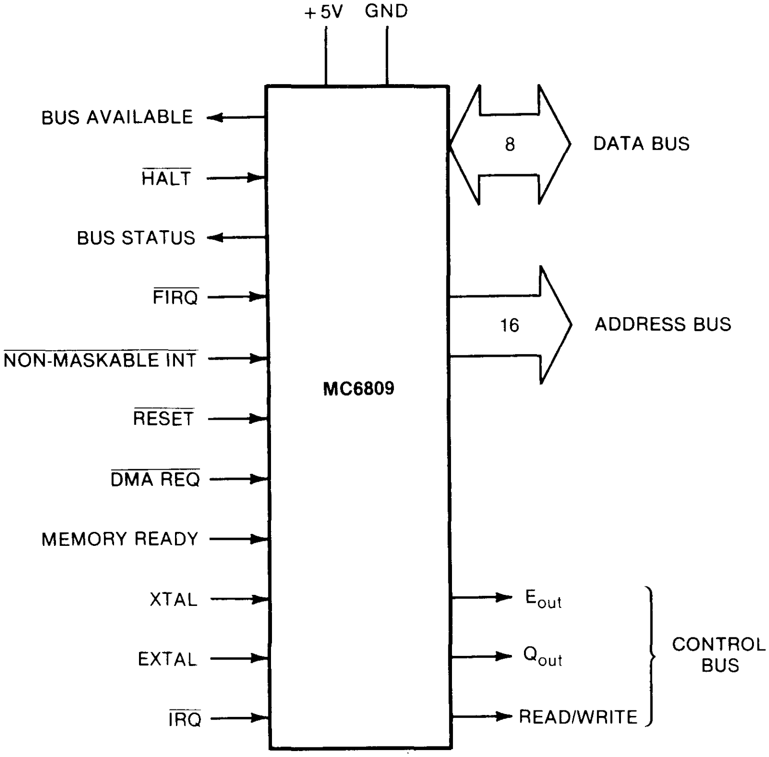


Fig. 1-4. 6809 bus and control signals.

quires only the addition of an external crystal (EXTAL and XTAL) to establish the internal clock frequency. The crystal frequency must be four times (4×) the desired internal clock frequency. Therefore, to achieve a 1-MHz operation, a 4-MHz crystal must be used. This has been done to create a more cost-effective system since an inexpensive 3.58-MHz tv color-burst crystal can be used, resulting in a 0.895-MHz clock frequency, without seriously affecting the chip performance.

The new control features of the 6809 include a fast interrupt request (FIRQ), bus status (BS), quadrature clock (Q), memory ready (MRDY), and direct memory access request (DMA REQ) control signals. You will use the fast interrupt request (FIRQ) line when it is known that the interrupt routine will use existing register data. Thus, with the fast interrupt, all the registers are not stored unnecessarily requiring extra time; *only* the program counter and condition code register are saved. However, as you will discover in Chapter 6, you can also use FIRQ to initiate a standard interrupt request where the contents of all the internal registers are saved.

The new bus status (BS) line is used in conjunction with the bus available (BA) line to indicate bus status and provide an interrupt acknowledge. The quadrature clock (Q) is a clock signal which leads the enable clock signal (E) by one-quarter cycle (90°). Enable (E) is the same as the 6800's $\phi 2$ clock signal. Valid addresses are available from the 6809 on the leading edge of Q, with data being latched on the trailing edge of E. These two external clock signals provide four effective system timing edges for interfacing purposes. The memory ready line (MRDY) is for interfacing with slow memories. This input control signal effectively *stretches* the E pulse to extend data-access time. Finally, the direct memory access request line ($\overline{\text{DMA REQ}}$) is an input control line which provides a method for suspending processor execution and freeing the external buses for other purposes, such as direct memory access by a peripheral device or dynamic memory refresh.

A bus and control signal diagram for the 6809E is shown in Fig. 1-5. Since the 6809E does not require an external crystal connection, two additional control signals are added. They are: Busy, and Last Instruction Cycle (LIC). Since these two control signals are avail-

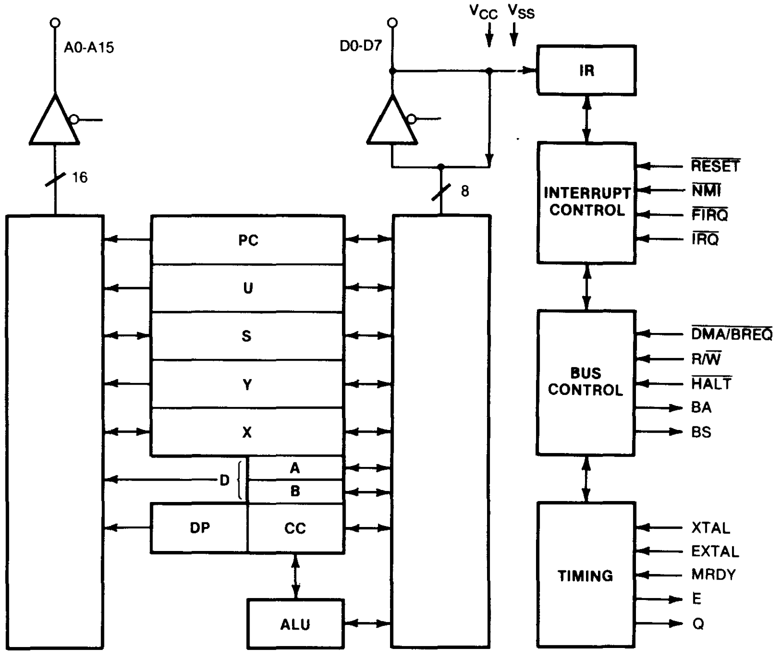


Fig. 1-6. 6809 chip structure.

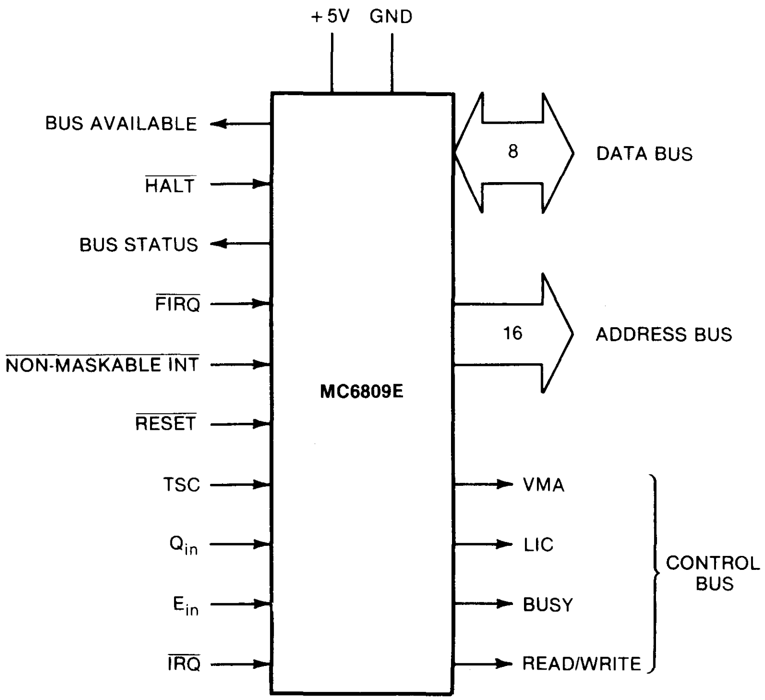


Fig. 1-5. 6809E bus and control signals.

able, the 6809E is ideally suited for multiprocessor applications. Busy is an output control line which indicates that the 6809E is accessing memory and the last instruction cycle (LIC) signal alerts external devices to the fact that the 6809E is executing the last cycle of an instruction.

If you are familiar with the 6800, note that the standard 6809 does not include two control lines that are available on the 6800. They are the valid memory address (VMA) and three-state control (TSC) signals. Recall that VMA was required in external device interfacing to provide proper device selection. Therefore, the elimination of this signal eliminates the gating required to use it for external device selection. Instead of using VMA to indicate a valid address exists on the address bus, the 6809 places FFFF on the address bus during any clock cycle when it is not using the bus structure. The three-state control line (TSC) is simply replaced by the $\overline{\text{DMA REQ}}$ line on the 6809. The 6809E, however, uses both VMA and TSC (Fig. 1-5).

Finally, there is one other subtle hardware difference between the 6800 and 6809. Both the 6809 and 6809E contain a Schmitt trigger input on the $\overline{\text{RESET}}$ interrupt such that a simple *RC* circuit is all that is required to reset the processor during "power up." This is not included in the 6800.

6809 CHIP STRUCTURE

The 6809 chip structure is shown in Fig. 1-6 in summary to the discussion just completed. You should now have a basic understanding of each functional region and the associated control features of the 6809 and 6809E. In addition, you should now be aware of the architectural, software, and hardware improvements of the 6809 and 6809E over the 6800. In the chapters that follow we will discuss each of these new features in detail, beginning with the 6809's addressing modes and its instruction set.

REVIEW QUESTIONS

1. **Compatibility between the 6800 and 6809 instruction set exists at what level?**

2. **How many basic instructions are contained in the 6809 instruction set?**

3. **How many different addressing modes does the 6809 utilize?**

4. **The 6809 was designed primarily for the ____________ market.**

5. **Two prime design considerations for the 6809 are: __________ and __________.**

6. **The most advanced processor in the Motorola 8-bit family is the __________.**

7. **The most advanced microcomputer in the Motorola 8-bit family is the ____________.**

8. **What is the fundamental difference between the 6809 and 6809E?**

9. **List the additional registers present in the 6809 architecture which do not exist in the 6800.**

10. **The two new condition code flags and their bit positions in the CC register are: ____________ and ____________.**

11. **Which of the internal 6809 registers are indexible and can therefore be used with the index mode of addressing?**

12. **When an instruction points to an address of the address of an operand,**

this is called __________ addressing.

13. What is the branching range of the 6809 and how does this compare to the 6800?

14. What is the difference in the direct addressing mode for the 6809 versus the 6800?

15. A 3.2-MHz crystal would produce a ________-MHz 6809 clock frequency.

16. With the 6809's fast interrupt request ($\overline{\text{FIRQ}}$) only the __________ and __________ registers are stacked.

17. The quadrature clock signal (Q) is the same frequency as the enable clock signal (E); however, Q leads E by what amount?

18. When a valid memory address is not present from the 6809, what is the status of the address bus?

19. The 6809 replaces the 6800 three-state control line (TSC) with ________.

20. The two extra control lines available on the 6809E which are not available on the 6809 are: __________ and __________.

ANSWERS

1. Source code (mnemonic)
2. 59
3. 19
4. Systems
5. Position-independent code and 6800 compatibility
6. 6809
7. 6801
8. The 6809E is an off-chip clock version of the 6809.
9. Accumulator D (D)
   Direct page register (DP)
   Y index register (Y)
   User stack pointer (U)
10. F (bit 6)
    E (bit 7)
11. All the index registers and stack pointers (X, Y, S, and U)
12. Indirect
13. The 6809 can branch anywhere within the 64K memory map using long relative addressing where the 6800 is limited to a plus 127_{10} and minus 128_{10} byte branching range.

14. With the direct addressing mode the 6809 can access any part of the 64K memory map using the direct page register (DP), where the 6800 is limited to the first 256 bytes of memory (page 0).

15. 0.8 MHz

16. Program counter (PC) and condition code (CC)

17. One-quarter cycle (90°)

18. The 6809 address is FFFF_{16} during any clock signal when it is not using the bus structure.

19. Direct memory access request ($\overline{\text{DMA REQ}}$)

20. Busy and last instruction cycle (LIC)

CHAPTER 2

6809 Addressing Modes

INTRODUCTION

You are about to see what makes the 6809 a *super* microprocessor. As was mentioned earlier, it is not the mere number of instructions in an instruction set that makes one processor more powerful than another but a more important consideration: how many different ways the processor can utilize the fundamental instructions available to it. Instructions can access and operate on data in different ways by using different *addressing modes.* The 6809 has 59 instructions which utilize ten *fundamental* addressing modes, bringing the total number of unique operations to 1464. The ten fundamental addressing modes available to the 6809 are: *inherent* (*implied*), *immediate, direct, extended, branch relative, indexed, extended indirect, program counter relative, indexed indirect,* and *register.*

In addition, variations of these ten fundamental modes actually bring the total number of unique addressing modes to *19.* For example, one of the most powerful addressing modes, indexed addressing, has five options: zero offset, constant offset (5-, 8-, or 16-bit), accumulator offset (A, B, or D) and auto-increment or auto-decrement (by 1 or 2). All of these options can access data indirectly using *indexed indirect* addressing. Indirect addressing means to address a memory location that contains the address of the operand rather than the operand itself. Thus, the instruction accesses the address of the operand, which in turn accesses the operand. Indirect operations are also available with the extended and program counter relative addressing modes.

As with the 6800, relative addressing is used with branch operations. With the 6809, however, relative addressing can also be used to access and operate on memory data. It is also available in two versions: short relative (8-bit offset) and long relative (16-bit offset). Thus 6809 programs can be written with complete *position independence.*

Finally, *register addressing* will allow you to transfer to, or exchange, data between any two like-size registers within the 6809's architecture.

You might be wondering how an 8-bit machine can perform so many unique operations and, given an instruction, how do you specify which addressing mode is to be used? The answer is found in a *post byte.* A post byte is used with indexed, indexed indirect, extended indirect, program counter relative, and register addressing. The post byte follows the instruction op code in the instruction statement. It is used to specify the addressing mode and which internal register is to be used in the operation. With register addressing it is used to specify which two like-size registers are to be used in the transfer or exchange of data.

It is very important that you understand how to use the post byte and, for that matter, *all* the concepts presented in this chapter such that you can take full advantage of all of the 6809's capabilities. Once a full understanding of the addressing modes is achieved, you will be ready to apply these modes to the 6809 instruction set presented in Chapters 3, 4, and 5.

OBJECTIVES

At the end of this chapter you will be able to do the following:

- State the difference between direct addressing for the 6809 versus direct addressing for the 6800.
- Explain the role of the direct page register in direct addressing for the 6809.
- List the ten fundamental 6809 addressing modes.
- Explain how to use program counter relative addressing.
- Describe the four basic forms of indexed addressing.
- Understand how to use a post byte for indexed, program counter relative, extended, and register addressing.
- Interpret the meaning of a given post byte.
- Determine the required post byte for a given addressing situation.
- Define indirect addressing.
- Determine the effective operand address for instructions using indirect addressing.
- Understand assembly language symbols used for the various 6809 addressing modes.

INHERENT, IMMEDIATE, AND EXTENDED ADDRESSING

Inherent Addressing

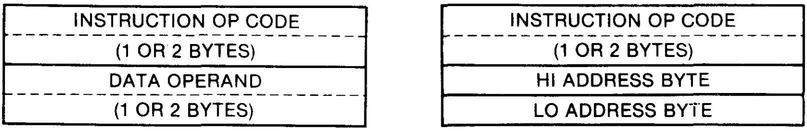
Inherent addressing, also referred to as *implied addressing,* is the simplest type of addressing since it only involves 1-byte instructions. Instructions involving the accumulators, such as increment, decrement, clear, shift (left or right), complement, etc., fall into this category. This type of addressing is also sometimes referred to as *accumulator addressing* when the instruction involves an operation on one of the accumulators. The format for inherent addressing is the same as that for the 6800 and most of the same instructions are involved.

Immediate Addressing (#)

As with the 6800 the 6809 uses immediate addressing when the operand *immediately* follows the instruction op code. Therefore this type of addressing includes the data operand within the instruction statement. Operations involving the accumulators and index registers, such as load, add, subtract, AND, OR, compare, etc., can all utilize immediate addressing. When the immediate operation involves accumulator A or B, the instruction will be 2 bytes: a 1-byte instruction op code followed by the 1-byte operand. When the operation involves a 16-bit register, accumulator D or one of the index registers (X, Y, S, or U), the instruction will be 3 or 4 bytes: a 1- or 2- byte instruction op code (depending upon the particular instruction, refer to Chapter 3) followed by a 2-byte operand. With the 6800, op codes were never more than 1 byte in length; however, with the 6809 you will frequently see 2-byte instruction op codes, especially where the 16-bit registers are involved. This is necessitated since, with the 6809, there are more than 256 unique executable instructions. Also, a 2-byte operand is required when a 16-bit register is involved. The immediate addressing format is shown in Fig. 2-1A.

Extended Addressing ($$)

Extended addressing is used to access memory. Here, the memory *address* of the data operand follows the instruction op code. Extended addressing is used to access the full 64K memory address



(A) Immediate addressing.

(B) Extended addressing.

Fig. 2-1. Immediate and extended addressing formats.

map, and therefore the address is 2 bytes in length: a HI address byte followed by a LO address byte. The instruction op code will be 1 or 2 bytes, depending on the particular instruction involved. Therefore 6809 instructions using extended addressing will be a total of 3 or 4 bytes in length. The instruction format for extended addressing is shown in Fig. 2-1B. Another form of extended addressing, extended indirect, will be discussed later in this chapter.

DIRECT ADDRESSING AND THE DIRECT PAGE REGISTER ($)

In the 6800, direct addressing was used to access operands that were stored in the first 256 bytes of memory (addresses 0000–00FF). Recall that when accessing this area of memory it was advantageous to use direct addressing over extended addressing, since fewer instruction bytes are used. For example, consider the following instruction codes:

```
1. LDA $        2. LDA $$
     C7              00
                     C7
```

These instructions accomplish the same function, i.e., load accumulator A with the contents of memory address C7. However, instruction No. 1 uses direct addressing while instruction No. 2 uses extended addressing. The obvious advantage is that direct addressing requires only 2 instruction bytes while extended addressing requires 3. In addition, more MPU cycles would be required to execute the instruction using extended addressing. Therefore, direct addressing is used whenever operands are located in *low* memory. The drawback to the use of direct addressing in the 6800 is that you are limited to addressing the first 256 bytes of memory (00–FF) in this way. An operand residing in higher memory would *require* the use of extended addressing. However, the 6809 design engineers have eliminated this problem by including a direct page register (DPR) as part of the 6809's architecture. The direct page register essentially forms the most significant byte of the effective address. The least significant address byte is part of the instruction. When in the direct addressing mode the 6809 simply looks to the DPR for the most significant address byte and relies on the instruction to supply the least significant address byte. Therefore, to access the data located at address 00C7, as in the previous example, the direct page register would contain 00 (all zeros) and the 2-byte instruction

```
LDA $
 C7
```

would be used. If, however, the DPR contained 01, the same instruction would access address 01C7. This eliminates the need to use extended addressing when you are accessing addresses above 00FF.

Some terminology is useful here. When the DPR contains 00, we say it is accessing page zero; when the DPR contains 01, it is accessing page 1, and so on, up to page 255 (FF). Note that the DPR can be used to access 256 pages (including page zero). Each page contains 256 address bytes. In other words, the DPR can be used to "*page*" through the entire 6809 address map, since 256 pages with 256 address bytes per page equals 64K address bytes, or the total 6809 map. A common application of the DPR is in high-level languages where *global variables* are accessed frequently. A global variable is a variable whose value is accessible throughout an entire program, in contrast to a *local variable,* whose value is accessible only in the program block (or subroutine) in which it is defined. In a subroutine the DPR can be used to "point" to a page containing the global variables, with the stack containing local variables. No problems arise as long as the language compiler keeps track of the DPR value. The DPR can also be used for *multitasking* operations. Multitasking is when several separate but interrelated *tasks* operate within a single program. In this application each task will be allocated a different page by the main program and accessed via the DPR. When writing programs you must be very careful in using the DPR since it is very easy to lose track of its value. But, when the DPR is used properly, you can generate very efficient, byte-saving programs.

After the 6809 is reset, the DPR contains all zeros (page zero) and the 6809 direct addressing mode will perform just as it would in the 6800. As you will see in Chapter 3 the DPR contents can then be easily modified to point to any page within the 6809's memory map.

RELATIVE ADDRESSING

There are two types of relative addressing available to the 6809. They are *branch relative* and *program counter relative.* By using these relative addressing modes you will be able to write complete position-independent programs for the 6809. A discussion of branch relative addressing will be provided first since it is very similar to the branching operation used in the 6800. Then a complete discussion of program counter relative addressing will be provided.

Branch Relative Addressing

Recall that the 6800 uses relative addressing for its branch instructions. The branch instruction op code is followed by a 1-byte twos complement (signed) relative address offset. Upon execution of the branch the offset is added to the program counter's contents to form the branch destination address.

Example 2-1: Determining Forward Branch Destinations

Given a branch instruction located at address 002B with a relative address offset of 6F, determine the branch destination.

Since the branch instruction is 2 bytes (op code followed by the relative address offset), the program counter contains 002B+2 = 002D. The destination equals the program counter contents plus the offset, or 002D+006F = 009C.

Example 2-2: Determining Backward Branch Destinations

Consider a branch instruction located at address 002B as in the previous example. The relative address offset, however, is now F7. Determine the branch destination.

The program counter contents are 002D (refer to Example 2-1). Since the most significant bit of the twos complement offset is a logic 1, it is negative and the 6809 will branch backward. To obtain the destination *the twos-complement offset* is added to the program counter's contents. Therefore the destination address is 002D+FFF7 = 0024. Here the two most significant hex digits (FF) of the offset are implied since the offset is negative.

Refer to Table 2-1 for determining the 8-bit relative address offset required to branch to a given destination.

A serious limitation arises with this type of branching: since the signed offset is only 1 byte, with the most significant bit of that byte used to determine branching direction (forward or backward), you are limited to a range of -128_{10} to $+127_{10}$ branching addresses, with respect to the branch operation. This is called *short branching.* To branch beyond this range with the 6800 would require branching to a branch or the use of a jump instruction. But, by using a jump, position independence is lost, which is a serious consequence, especially for high-level language programming, since an absolute address location is specified. However, if the relative address offset were 2 bytes, branches within a $-32{,}768_{10}$ to $32{,}767_{10}$ range could be accomplished, thus allowing branching anywhere within the entire memory map while maintaining position independence throughout the program. This has been done with the 6809 while maintaining the short-branch instructions for byte efficiency when only short branches are required.

The 6809 adds a series of *long-branch* instructions which use a 2-byte signed relative address offset. The 6809 long-branch instructions contain a *2-byte op code* followed by a *2-byte signed offset.* Therefore, long-branch instructions are 4 bytes in length as compared with the 2-byte short-branch instructions.

Table 2-1. 6809 Short-Branch Calculator Table

MSH-B	F	E	D	C	B	A	9	8	
LSH-B									LSH-F
-	-	16	32	48	64	80	96	112	0
F	1	17	33	49	65	81	97	113	1
E	2	18	34	50	66	82	98	114	2
D	3	19	35	51	67	83	99	115	3
C	4	20	36	52	68	84	100	116	4
B	5	21	37	53	69	85	101	117	5
A	6	22	38	54	70	86	102	118	6
9	7	23	39	55	71	87	103	119	7
8	8	24	40	56	72	88	104	120	8
7	9	25	41	57	73	89	105	121	9
6	10	26	42	58	74	90	106	122	A
5	11	27	43	59	75	91	107	123	B
4	12	28	44	60	76	92	108	124	C
3	13	29	45	61	77	93	109	125	D
2	14	30	46	62	78	94	110	126	E
1	15	31	47	63	79	95	111	127	F
0	16	32	48	64	80	96	112	-	-
	0	1	2	3	4	5	6	7	MSH-F

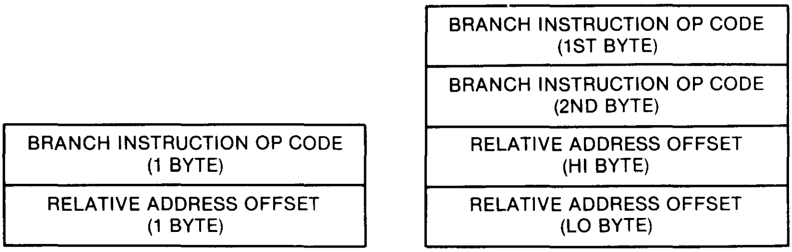
1. Count the number of bytes (in decimal) from the instruction following the branch to the branch target instruction.
2. Find this number inside the table.
3. Read this hexadecimal equivalent.
   a. Top and left for branching backward.
   b. Bottom and right for branching forward.

Examples: Back 15_{10} bytes $F1_{16}$, Forward 77_{10} bytes = $4D_{16}$, Back 107_{10} bytes = 95_{16}.

4. Key:
   MSH-B = Most significant hex-backward
   LSH-B = Least significant hex-backward
   MSH-F = Most significant hex-forward
   LSH-F = Least significant hex-forward

Courtesy Mr. Ray Boaz, 1516 Jarvis Pl., San Jose, Calif. 95118

Both the 6809 long- and short-branch instruction formats are shown in Fig. 2-2. The short-branch instruction format shown in Fig. 2-2A is the same as that required for the 6800. The long-branch instruction format shown in Fig. 2-2B requires a 2-byte op code followed by the 2-byte signed offset. The high offset byte is first, followed by the low offset byte.


BRANCH INSTRUCTION OP CODE (1 BYTE)
RELATIVE ADDRESS OFFSET (1 BYTE)
BRANCH INSTRUCTION OP CODE (1ST BYTE)
BRANCH INSTRUCTION OP CODE (2ND BYTE)
RELATIVE ADDRESS OFFSET (HI BYTE)
RELATIVE ADDRESS OFFSET (LO BYTE)

(A) Short-branch instruction format.

(B) Long-branch instruction format (except for long branch always and subroutine).

Fig. 2-2. 6809 branch instruction formats.

Example 2-3: Determining Long-Branch Destinations

Given a branch instruction located at address 5000 with a relative address offset of 02F0, determine the branch destination.

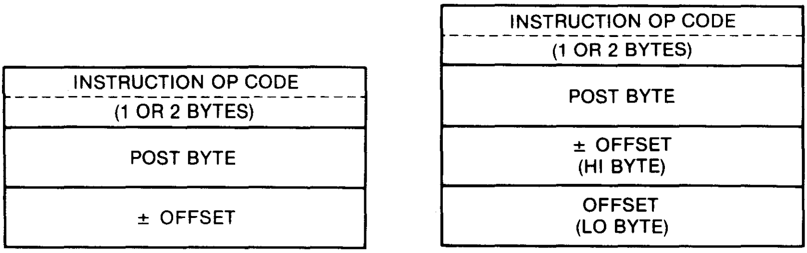
Since the long branch instruction is 4 bytes (a 2-byte op code followed by a 2-byte signed offset), the program counter contains 5000+4 = 5004. The destination equals the program counter's contents plus the offset, or 5004+02F0 = 52F4.

Now, suppose the offset were F2F0. Then, since the most significant bit of the offset is a logic 1, the 6809 will branch backward. As with short branches, to obtain the destination, add the twos complement offset to the program counter's contents. Therefore the destination address is 5004+F2F0 = 42F4.

Program Counter Relative (PC Relative) Addressing

To allow complete position-independent programs to be written without a lot of software overhead, the 6809 includes program counter relative addressing. With this type of addressing the address of an operand located in memory can be determined *relative* to the program counter's contents. With PC relative addressing an 8- or 16-bit signed offset is added to the program counter's contents to create the effective address of the operand, or the effective address of the address of the operand. (The latter case is referred to as indirect addressing, and will be discussed later.) Therefore data can be accessed anywhere within the 6809's memory map, relative to the program counter contents, thus allowing for complete position-independent programs.

The instruction format required for program counter relative addressing is shown in Fig. 2-3. Note that the instruction can be 3 or 4 bytes in length, depending on the offset desired. Both 8- and 16-bit offsets are allowed. In either case a *post byte* follows the instruction op code. The offset byte(s) then follows the *post byte*. When this type of addressing is used, the program counter is used as a pointer register with an 8- or 16-bit offset; thus program counter relative



(A) 8-bit signed offset. **(B) 16-bit signed offset.**

Fig. 2-3. Program counter relative addressing.

addressing can be thought of as a type of indexed addressing. As you will see in Chapter 3 any instructions which use indexed addressing can also use program counter relative addressing. Since program counter addressing is considered a type of indexed addressing, a *post byte* is required to designate that the program counter register is being used as the pointer register rather than one of the indexible registers (X,Y,S,U). The use of the post byte will be discussed in more detail in the next section, on indexed addressing.

The following examples illustrate program counter relative addressing.

Example 2-4: Using Program Counter Relative Addressing With an 8-Bit Signed Offset

Consider the following instruction mnemonic sequence:

```
  .           .
  .           .
00FF          .
0100        LDA (instruction op code)
0101        post byte
0102        10 (8-bit offset)
0103          .
  .           .
  .           .
```

The instruction is to load accumulator A with the contents of the memory address determined by adding the signed offset to the program counter's contents. The signed offset is 10_{16} ($0001\ 0000_2$) and the post byte specifies that the program counter is to be the pointer register. Recall that the program counter always points to the *next* instruction to be executed. Therefore, the program counter's contents are 0103 at this point in the program. Thus the operand to be loaded into accumulator A is located at address 0103+0010 = 0113. In this example the offset was positive since its most significant bit is 0. However, if the most significant bit of the offset is 1, the *twos complement* offset is negative. For example, if the offset were F0 ($1111\ 0000_2$), the effective operand address would be 0103+FFF0 = 00F3.

Example 2-5: Using Program Counter Relative Addressing With a 16-Bit Signed Offset

Consider the following instruction mnemonic sequence:

```
  .
  .           .
00FF          .
0100        LDB (instruction op code)
0101        post byte
0102        01 (offset HI byte)
```

```
0103        FF (offset LO byte)
0104         .
  .          .
  .          .
```

This instruction will load accumulator B with the contents of the memory location formed by adding the 16-bit offset (01FF) to the program counter's contents. The program counter contains 0104 at this point in the program. Therefore the effective operand address is 0104+01FF = 0303.

INDEXED ADDRESSING

You can probably recall that indexed addressing is the most powerful form of addressing available to the 6800. It became extremely valuable when operating on data located in consecutive memory locations. In the 6800 the indexed addressing instruction is a 2-byte instruction consisting of the instruction command followed by a signed indexed offset. To form the effective operand address the 6800 adds the offset to the contents of the 16-bit index register. Several 6800 instructions are associated with the index register. Recall also that you can increment, decrement, load, store and transfer the index register contents to and from the stack pointer. A substantial amount of programming power was gained by including the index register within the 6800's architecture.

Now, with the 6809, the indexed addressing mode has been enhanced by the inclusion of four 16-bit registers which may be used as *pointer registers* in the indexed mode of addressing. As mentioned in Chapter 1 these registers are the X, Y, S, and U registers (refer to Fig. 1-3). With the 6809 there are *four* basic forms of indexed addressing that can be used with these registers. They are: *zero-offset* indexed, *constant-offset* indexed, *accumulator-offset* indexed, and *auto-increment/decrement* indexed.

The instruction format for indexed addressing is shown in Fig. 2-4. You see that the instruction op code is *always* followed by a *post byte,* which in turn may or may not be followed by an *offset.* The *post byte* of an indexed instruction specifies the basic form of in-

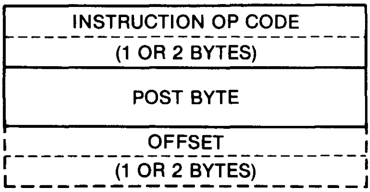


Fig. 2-4. General indexed addressing instruction format.

dexed addressing to be used as well as the specific pointer register that will be used in determining the effective operand address. Indexed addressing can be used with 31 of the 59 instructions of the 6809.

Now let us discuss each basic type of indexed addressing; then, we will learn how to use the post byte. Examples of each indexed mode are provided at the end of this chapter.

Zero-Offset Indexed Addressing

This type of indexed addressing allows the pointer register to point directly to the effective operand address, since the offset is zero. In other words, the specified pointer register contains the *address* of the operand to be used in the operation. These instructions will be 2 or 3 bytes: the instruction op-code byte(s)* followed by the post byte. The post byte will specify the zero-offset mode and the pointer register to be used. This mode of indexed addressing is the fastest since a minimum number of bytes is required and no offset calculation is required.

Constant-Offset Indexed Addressing

This form of indexed addressing is very similar to the 6800's indexed addressing. However, any of the pointer registers (X, Y, U, or S) can be used and the signed offset can be 5, 8, or 16 bits. The post byte will follow the instruction op code and will specify the pointer register *and* the offset size. When a 5-bit signed offset is used, the *offset is included as part of the post byte* and therefore the 5-bit offset is most efficient in the use of bytes and MPU cycles as compared with the other constant-offset versions. Also, the offset is a twos complement (signed) value with the most significant bit (msb) of the offset used to determine its sign. If the most significant bit is a logic 0, the offset will be positive, and if the most significant bit is a logic 1, the offset will be negative. Therefore, the 5-bit offset reduces to a ±4-bit offset with a corresponding offset range of from -16_{10} to $+15_{10}$.

If an 8-bit offset is desired, the instruction will be 3 or 4 bytes in length. Here the instruction op-code byte(s)* is followed by the post byte, which in turn is followed by the 8-bit offset byte. The post byte will specify constant-offset indexed addressing and the pointer register to be used. The offset byte is then added to the pointer register contents to determine the effective address. Again, the offset is a twos complement (signed) value with the msb used to determine the sign. Therefore the 8-bit offset reduces to a ±7-bit offset with a corresponding range of from -128_{10} to $+127_{10}$.

* With the 6809 some instructions require 2 op-code bytes.

If a 16-bit offset is desired, the post byte will be followed by 2 offset bytes. The most significant offset byte will be first, then the least significant byte. To determine the effective address, the 16-bit signed offset will be added to the pointer register specified by the post byte. Since the offset is signed, it reduces to ±15 bits with a corresponding range of from $-32{,}768_{10}$ to $+32{,}767_{10}$.

Accumulator-Offset Indexed Addressing

This type of indexed addressing is similar to the constant-offset mode, except that the contents of one of the accumulators (ACCA, ACCB, or ACCD) are added to the specified index register (X, Y, S, or U) to obtain the effective address. The obvious advantage here is that the offset can be calculated just prior to the indexed operation. The instruction will be 2 or 3 bytes in length: the instruction op-code byte(s) followed by the post byte. The post byte will specify the accumulator-offset mode, pointer register, and which accumulator is to be used. The 6809 uses the twos complement (signed) value of the specified accumulator to determine the effective address. Neither the specified accumulator or index register contents are affected by the offset calculation.

Auto-Increment/Decrement Indexed Addressing

This mode of indexed addressing is a blessing, since it eliminates the need to increment/decrement the index register with a separate instruction when stepping through memory tables and moving blocks of data within memory. In the auto-increment mode the specified pointer register contains the address of the first operand. Then, after the first operand is used in the operation specified by the op code, the pointer register is automatically incremented to point to the next consecutive operand address and so on, as many times as the instruction is executed. Therefore, memory data are fetched consecutively from a low to a higher memory address. In the auto-decrement mode, memory data are fetched from a high to a lower memory address since the specified pointer register is automatically decremented just *prior to* the operand's being fetched. The auto-increment is therefore a *post-increment* operation and the auto-decrement a *pre-decrement* operation. Thus, if you are using the auto-decrement option, the starting address must be $n+1$ to fetch information from address n. The increment/decrement can be either by 1 *or* 2 to allow for 8- or 16-bit data.

The auto-increment/decrement instruction will be 2 or 3 bytes in length: the instruction op-code byte(s) followed by the post byte. The post byte will specify auto incrementing or decrementing, the pointer register to be incremented or decremented, and the amount of increment or decrement (1 or 2).

POST BYTE

From the previous discussion it is obvious that the post byte plays an important role in the indexed addressing mode. In addition, you saw that a post byte was also required when using program counter relative (PC relative) addressing. In summary, you should have discovered from previous discussions that:

1. The post byte is used to specify one of the following modes of addressing:
   program counter relative
   zero-offset indexed
   constant-offset indexed
   accumulator-offset indexed
   auto-increment/decrement indexed
2. The post byte specifies either the program counter, X, Y, S, or U register to be used as the pointer register.
3. The post byte specifies the offset size to be used in program counter relative and constant-offset indexed addressing.
4. The post byte specifies the accumulator (A, B, or D) to be used during accumulator-offset indexed addressing.
5. When using a 5-bit signed constant-offset, the offset value is included as part of the post byte.

How is all of this accomplished with 1 byte of information? The answer is found in the post-byte format shown in Fig. 2-5. You see that the post byte is divided into the following bit fields:

addressing mode field (bits 0–3)
indirect field (bit 4)
pointer register field (bits 5 and 6)
5-bit offset field (bit 7)

Each of the above post-byte fields will now be discussed.

Addressing Mode Field (Bits 0–3)

This 4-bit field is used to select the type of addressing mode that is to be used. The addressing modes and their respective bit pattern

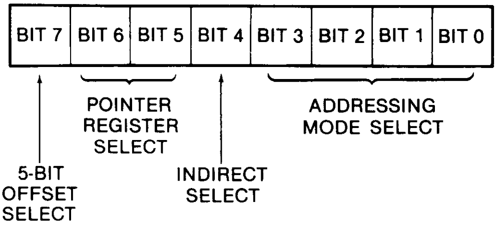


Fig. 2-5. Indexed addressing post-byte format.

Table 2-2. Bit Pattern Definitions of Addressing Mode Fields

Post-Byte Addressing Mode Field					
Bit 3	Bit 2	Bit 1	Bit 0	Addressing Mode	Symbol
0	0	0	0	Auto Increment (+1)	R^{+}
0	0	0	1	Auto Increment (+2)	R^{++}
0	0	1	0	Auto Decrement (−1)	−R
0	0	1	1	Auto Decrement (−2)	−−R
0	1	0	0	Zero Offset	R ±0
0	1	0	1	ACCB Offset	R ±ACCB
0	1	1	0	ACCA Offset	R ±ACCA
1	0	0	0	8-Bit Signed Offset	R ±7 Bit
1	0	0	1	16-Bit Signed Offset	R ±15 Bit
1	0	1	1	ACCD Offset	R ±ACCD
1	1	0	0	PC Relative—8-Bit Signed	PC ±7 Bit
1	1	0	1	PC Relative—16-Bit Signed	PC ±15 Bit
1	1	1	1	Extended Indirect	[n]

definitions are shown in Table 2-2, where R denotes the specified pointer register (X, Y, S, or U). Note that all of the previously discussed modes of indexed addressing and program counter relative addressing can be defined with the first 4 post-byte bits. Also, an additional mode of addressing, *extended indirect,* can also be specified. This mode will be discussed shortly.

Indirect Field (Bit 4)

Bit 4 of the post byte selects *indirect* addressing. A logic 0 in this bit position indicates that *direct* addressing is to be used. Thus the operand is actually located at the address specified in the operation. A logic 1 indicates that *indirect* addressing is to be used. Here the address points to a location that contains the actual address of the operand. More will be said about indirect addressing shortly.

Pointer Register Field (Bits 5 and 6)

This field selects the pointer register that is to be used in the address determination. The pointer registers and their respective bit pattern definitions are shown in Table 2-3. Note that any one of the four indexible registers (X, Y, S, or U) may be specified as the pointer register.

Five-Bit Offset Field (Bit 7)

You will always set this bit to a logic 1 state *except* when you desire a 5-bit signed offset. In this case, bit 7 will be cleared and the 5-bit, twos complement offset is placed in bits 0 through 4 of the

Table 2-3. Bit Pattern Definitions of Pointer Register Fields

Post-Byte Pointer Register Field		
Bit 6	Bit 5	Pointer Register
0	0	R = X
0	1	R = Y
1	0	R = U
1	1	R = S

post byte. Bit 4 becomes the sign bit, with bits 0 through 3 representing the offset weight.

All of the post-byte register bit assignments are summarized in Table 2-4. Now, let us look at some examples of instructions which use the post byte. Before you can write mnemonic code for these instructions, however, you need a code which can be used to represent the various post-byte designations. Such a code is shown in Table 2-5. In this table, n denotes offset value, R the specified pointer register, and PC the program counter. You will follow the instruction mnemonic with the respective post-byte designation shown in Table 2-5. For example, LDA 23,X means to load accumulator A with the contents of the memory location specified by adding the constant offset 23_{16} to pointer register X; LDX D, Y means to load index register X with the contents of the memory location specified by adding the contents of accumulator D to pointer register Y.

Example 2-6: Assembler Code and Post-Byte Determination

Suppose you wish to store the contents of accumulator D in memory at the address specified by the S pointer register using a constant offset of −5. What would be the assembler code to represent this operation?

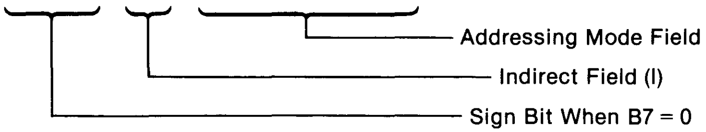
From Table 2-5, the proper assembler code would be: STD −5, S. What post byte is required to perform this operation? The post byte would be:

$$7B_{16} = 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1_2$$

Let's analyze the above post byte. Since the constant offset is −5, a 5-bit signed offset can be used. Therefore the offset can be included as part of the post byte. Bit 7 of the post byte is cleared to indicate a 5-bit constant offset. Bits 5 and 6 are set to specify the S register as the pointer register. The constant offset is then represented by bits 0 through 4: bit 4 is set to indicate a negative offset, while the *twos complement* offset value (1011) is specified in bits 0 through 3.

Table 2-4. Indexed Addressing Post-Byte 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 = ,R ± 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	0	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



Register Field
00:R = X
01:R = Y
10:R = U
11:R = S
X = Don't Care

Example 2-7: Post-Byte Interpretation

Given the post byte A0, interpret its meaning and determine the assembler code that would be used with an instruction which used this post byte.

Refer to Table 2-4:

$$\text{post byte} = A0_{16} = 10100000_2$$

Bit 7 is set, meaning that a 5-bit constant offset is *not* specified. The specified pointer register is the Y register since the pointer register field (bits 5 and 6) is 01. The address mode field (bits 0 through 4) is all zeros, indicating that auto increment by one is the specified addressing mode. The required assembler code for this post byte would be: ,Y+ (refer to Table 2-5). The effective address for an operation using this post byte would be the contents of the Y index

Table 2-5. Post-Byte Assembler Code

Addressing Mode Symbol	Assembler Code
R ±0	,R
R ±4 Bit	n,R
R ±7 Bit	n,R
R ±15 Bit	n,R
R ±ACCA	A,R
R ±ACCB	B,R
R ±ACCD	D,R
PC ±7 Bit	n,PCR
PC ±15 Bit	n,PCR
R^+	$,R^+$
R^{++}	$,R^{++}$
−R	,−R
−−R	,−−R

register. After the operation has been executed, the Y index register contents would be automatically incremented by 1.

Example 2-8: Post-Byte Interpretation

Repeat the steps of Example 2-7 for a post byte of ED.

Here the post byte $ED_{16} = 11101101_2$. Again, refer to Table 2-4 to follow the following interpretation. The addressing mode field contains 1101 which, from Table 2-4, specifies program counter relative addressing with a 16-bit signed offset. The indirect field is cleared, meaning that indirect addressing is *not* specified. The pointer register field bits are set; however, since PC relative addressing is specified, we "don't care" what the contents of this field are since the program counter is the pointer register. Finally, the 5-bit constant offset field is also set meaning that a 5-bit offset is *not* specified. Note that if this field were cleared the post byte would take on a whole new meaning.

The proper assembler code would be: n, PCR where n would equal the 16-bit signed offset value. The effective address for an operation using this post byte would be obtained by adding the constant offset n to the present program counter contents.

Example 2-9: Assembler Code and Post-Byte Determination

Suppose you wish to load the X index register with the contents of the memory location specified by the X pointer register using the accumulator D contents as a constant offset. What would be the proper assembler code to represent this operation and what post byte would be required?

From Table 2-5, the proper assembler code would be: LDX D, X. Since accumulator D offset is required, the addressing mode field of the post byte must be 1011 (refer to Table 2-4). Also, since the X register must be the specified pointer register, the pointer register field must be 00. The indirect addressing field (bit 5) must be cleared and the 5-bit constant offset field (bit 7) must be set. Therefore the required post byte would be 10001011_2 or $8B_{16}$.

INDIRECT ADDRESSING

As mentioned in previous discussions the 6809 is capable of a very powerful mode of addressing known as *indirect addressing*. With indirect addressing the *operand's address* (effective address) is contained at the location specified by the operation. Therefore you obtain the operand "indirectly" via an intermediate address. For example, suppose that an indirect addressing operation is to locate an operand at address 02F0. The operation would first point to an intermediate address, say 01F0, which would contain the actual address of the operand, 02F0. Actually, address 01F0 would contain 02 and address 01F1 would contain F0. This operation is summarized in Fig. 2-6.

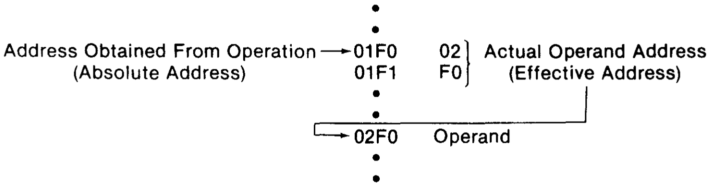


Fig. 2-6. Indirect addressing

Indirect addressing is a very valuable asset of the 6809 since it can be used with *any* of the indexed modes of addressing, *except* for auto-increment/decrement by 1. In addition, indirect addressing can be used with program counter relative addressing and extended addressing. To specify indirect addressing for any of the above addressing modes, you will simply set the indirect field, bit 4, of the post byte. The assembler codes to indicate indirect addressing will be the same as those listed in Table 2-5, except that the code is bracketed: [ ]. For example, LDA [10, X] would mean to load accumulator A with the contents of the address located at the memory location specified by the X pointer register plus the offset 10_{16}.

Example 2-10: Indirect Addressing (Indexed)

The following program segment is given:

```
  .
  .
00FF
0100        LDA [10,X]
0101        post byte
0102        10
  .          .
  .          .
020E        F0
020F        95
0210        05
0211        00
0212        B7
  .
  .
04FE         C6
04FF        E9
0500        AA
0501        F5
  .
  .
```

If, prior to executing the LDA instruction, the X index register contains 0200, what will be loaded into accumulator A? The operand's *address* will be found at the memory location formed by adding the constant offset 0010 to the contents of the index register. Therefore the *memory location which contains the address of the operand* is 0010 + 0200, or 0210. From the program listing you find 05 at this address, which is the high byte of the operand's address. The low byte is found in the next consecutive memory location. Therefore the operand's address is 0500. Thus the operand is located at address 0500. At this address you find the value AA, which will be loaded into accumulator A.

What post byte would be required for this operation? Since an 8-bit signed offset is required, the addressing mode field of the post byte must be 1000 (refer to Table 2-4). Since the pointer register is the X index register, the pointer register field must be 00. The indirect addressing field (bit 5) must be *set* to indicate indirect addressing and the 5-bit offset field (bit 7) must be set. Therefore the proper post byte would be 10011000_2 or 98_{16}. This value must appear at address 0101 in the above program listing. The constant offset, 10_{16}, follows the post byte at address 0102.

Example 2-11: Indirect Addressing (PC Relative)

Consider the following program segment:

```
  .
  .
00FF        STD [0E, PCR]
```

```
0100        post byte
0101        0E
0102        next instruction
  .           .
  .           .
010F        7D
0110        F1
0111        50
0112        AB
  .           .
  .           .
```

Where will the contents of accumulator D be stored? The instruction specifies program counter relative, indirect addressing with an 8-bit signed offset of 0E. The program counter always points to the *next* instruction; therefore the program counter contains 0102. The high byte of the address where the accumulator D contents are to be stored is found at the memory location formed by adding the constant offset, OE, to the program counter contents of 0102, or 0E+ 0102 = 0110. Note from the program listing that F1 is at this address. The next consecutive address contains 50. Therefore, the address where the contents of accumulator D will be stored is F150. Since accumulator D is a 16-bit accumulator, the high byte will be stored at address F150, with the low byte stored at address F151.

What post byte would be required for this operation? Since program counter relative addressing with an 8-bit signed offset is required, the addressing mode field of the post byte must be 1100 (refer to Table 2-4). The pointer register is the program counter; therefore, it doesn't matter what the pointer register field contains (don't care–refer to Table 2-4). The indirect addressing field (bit 5) must be set to indicate indirect addressing and the 5-bit offset field (bit 7) must be set. Therefore the proper post byte would be $1XX11100_2$ or $9C_{16}$ if the don't cares were zeros. This value must appear at address 0100 in the above program. The constant offset, $0E_{16}$, follows the post byte at address 0101.

Example 2-12: Indirect Addressing (Extended)

Consider the following program segment:

```
  .
  .
0500        STX [F150]
0501        post byte
0502        F1
0503        50
  .           .
  .           .
F14F        AF
F150        C5
```

```
F151      50
F152      9C
  .        .
  .        .
```

Where will the contents of the X index register be stored? The assembly language code for extended indirect addressing is [n], where n is the address of the address of the operand (or where the operand is to be stored). Therefore the above instruction specifies extended indirect addressing. Note that this instruction consists of 4 bytes: the instruction op code followed by the post byte, which is in turn followed by the *address* at which the effective address will be found. Thus the high byte of the effective address is located at F150 and the low byte is located at F151. Address F150 contains C5 and F151 contains 50. Therefore, the address where the X index register contents will be stored is C550. Since the X index register is a 16-bit register, the high byte will be stored at address C550 and the low byte at address C551.

What post byte would be required for this operation? Since extended indirect addressing is required, the addressing mode field of the post byte must be 1111. Neither of the pointer registers is used to determine the effective address and the pointer register field *must* be cleared (refer to Table 2-6). The indirect addressing field (bit 5) must be set to indicate indirect addressing and the 5-bit offset field (bit 7) must be set. Therefore, the proper post byte would be 10011111_2 or $9F_{16}$. This value must appear at address 0501 in the above program.

Obviously, many more possibilities exist than those provided in these few examples. The reader is encouraged to work the problems given in the review questions at the end of this chapter, and then to make up examples such that a full understanding of the various addressing modes is obtained. Finally, Table 2-6 summarizes the various indexed addressing modes, along with their respective assembly language codes and post-byte formats.

REGISTER ADDRESSING

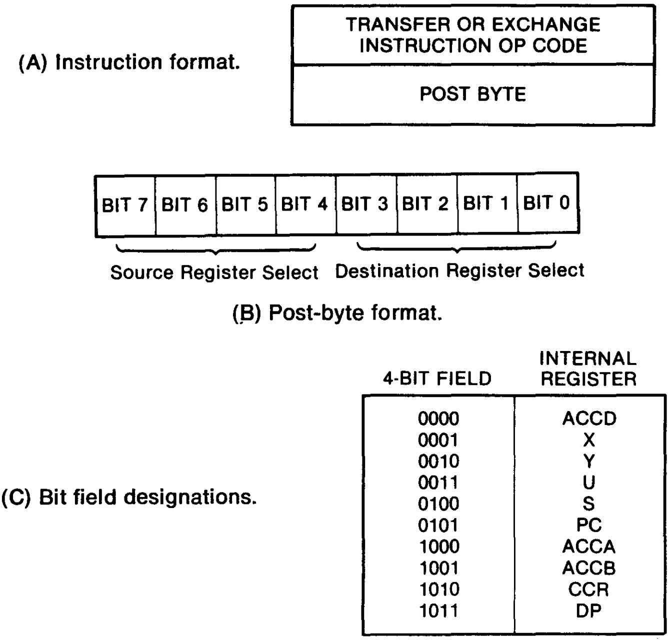
There is one more major mode of addressing available to the 6809 that should be discussed. Within the 6809, any user register may be transferred to, or exchanged with, any other register of like size. Inherent instructions are used to accomplish this task; however, a post byte is required to define the registers involved. Therefore any instructions which involve the transfer or exchange of data between internal registers will be referred to as *register addressing* instructions. As you will see in Chapter 3, these instructions will be 2 bytes in length: the instruction op code followed by a post byte. The in-

Table 2-6. Indexed Addressing Modes Summary

		Nonindirect				Indirect			
		Assembler	Post-Byte	×	+	Assembler	Post-Byte	+	+
Type	Forms	Form	Op Code	~	#	Form	Op Code	~	#
Constant Offset From R (Signed Offsets)	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 (Signed Offsets)	A – Register offset	A, R	1RR00110	1	0	[A, R]	1RR10110	4	0
	B – Register offset	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 = 00	Y = 01
X = Don't care	U = 10	S = 11

+ and + Indicate the number of additional cycles and bytes for the particular variation.
~ #



(A) Instruction format.

(B) Post-byte format.

(C) Bit field designations.

4-BIT FIELD	INTERNAL REGISTER
0000	ACCD
0001	X
0010	Y
0011	U
0100	S
0101	PC
1000	ACCA
1001	ACCB
1010	CCR
1011	DP

Fig. 2-7. Register addressing.

struction and post-byte formats are shown in Fig. 2-7. The post byte is divided into two fields: the *source register* field and *destination register* field. With the instructions provided in Chapter 3, data can be moved from the source register to the destination register or exchanged between the source and destination registers. The bit-pattern definitions for each internal register are also given in Fig. 2-7. The only requirement is that both registers defined by the post byte *must* be of like size; that is, 8-bit to 8-bit, or 16-bit to 16-bit.

REVIEW QUESTIONS

1. **Inherent addressing is also referred to as ____________ addressing.**
2. **Instructions which involve transfers or exchanges of data between internal registers use ____________ addressing.**
3. **State the difference between direct addressing with the 6809 *vs.* direct addressing with the 6800.**

4. The DPR forms the ____________ byte of the direct address and the ____________ byte is supplied as part of the instruction.

5. Two types of relative addressing that are available to the 6809 are __________ and __________.

6. Two types of branches available to the 6809 are _________ and _________.

7. Describe the 6809 long branch instruction format.

8. Given a branch instruction located at address 2000 with a relative address offset of F150, determine the branch destination.

9. State the two types of PC relative addressing.

10. List the registers available for use with indexed addressing.

11. The four basic forms of indexed addressing are: __________, __________, __________, and __________.

12. With indexed addressing the instruction op code is always followed by __________.

13. The three constant offsets available with constant-offset indexed addressing are: __________, __________, and __________.

14. Why is the 5-bit signed offset mode the most efficient constant-offset indexed addressing mode?

15. The range of the 16-bit signed offset is from __________ to __________.

16. State an advantage of accumulator-offset over constant-offset indexed addressing.

17. Another name for auto-increment/decrement could be ____________.

18. Using auto-increment/decrement, the specified pointer register can be automatically incremented or decremented by ____________ or ____________.

19. List the bit fields, and their corresponding bit positions, that make up the indexed addressing post byte.

20. When a 5-bit signed offset is *not* included in the post byte, the 5-bit offset field (bit 7) must be ____________.

21. You want to load accumulator A with the contents of a memory location specified by the Y index register using a constant offset of 10_{16}. What would be the assembler code used to represent this operation?

22. What post byte would be required to perform the operation in Question 21?

23. Given an indexed addressing post byte of 83_{16}, interpret its meaning and determine the proper assembler code.

24. Define indirect addressing.

25. Which of the 6809 addressing modes can use indirect addressing?

26. The assembly language symbol used to represent indirect addressing is:

27. Given the following program segment,

```
  .
  .
04FB          LDA #
04FC          10
04FD          LDU #
04FE          FC
04FF          50
0500          LDX [A, U]
0501          post byte
  .           .
  .           .
06FE          C7
06FF          F5
0700          1B
0701          AA
  .           .
  .           .
FC5E          D5
FC5F          C7
FC60          06
FC61          FF
FC62          00
  .
  .
```

what will be loaded into the X index register?

28. What post byte would be required for this operation?

29. How many bytes are required for an instruction using extended indirect addressing?

30. As you will discover in Chapter 3, the EXG instruction exchanges the contents of two like-size registers. Which two registers would be exchanged using the following instruction?

```
EXG
8B
```

ANSWERS

1. Implied or accumulator if the operation involves one of the accumulators
2. Register
3. With the 6800, direct addressing is limited to the first 256 bytes of memory. With the 6809, direct addressing can be used with the entire 64K memory map by using the direct page register (DPR).
4. Most significant
   Least significant
5. Branch relative and program counter relative
6. Short and long branches
7. A 2-byte instruction op code followed by a 2-byte relative address offset (except for LBRA and LBSR, where the op code is only 1 byte; see Chap. 5)
8. Since the offset is 2 bytes (F150), a long branch instruction is required. Long branches use 4 bytes; therefore the program counter contains 2000 + 4 = 2004. Now, the most significant bit of the offset is a logic 1; therefore the 6809 will branch backward. To obtain the destination, you add the twos complement offset to the program counter contents. Thus the destination address is 2004 + F150 = 1154.
9. Program counter relative with an 8-bit signed offset and program counter relative with a 16-bit signed offset
10. X, Y, S, and U
11. Zero-offset, constant-offset, accumulator-offset, and auto-increment/decrement indexed
12. A post byte
13. 5-, 8-, and 16-bit signed offsets
14. Since the 5-bit signed offset is included as part of the post byte
15. −32,768 to +32,767 (±15 bits)
16. With accumulator-offset, the offset can be calculated just prior to the indexed operation.
17. Post-increment/pre-decrement
18. 1 or 2
19. Addressing mode field (bits 0–3)
    Indirect field (bit 4)
    Pointer register field (bits 5 and 6)
    5-Bit offset field (bit 7)
20. Set (logic 1)
21. LDA 10, Y (reference, Table 2-5)
22. $10101000_2 = A8_{16}$ (reference, Table 2-6)
23. This post byte specifies auto decrement by two on the X index register.

The X index register will be decremented by two *after* each execution of an instruction using this post byte. The assembler code will be: ,— —X.

24. With indirect addressing the addressed memory location contains the address of the operand rather than the operand itself.

25. Extended addressing, program counter relative addressing, and all of the indexed addressing modes *except* auto-increment/decrement by 1

26. The bracket, [ ]

27. Prior to the LDX instruction, accumulator A is loaded with 10 and the U pointer register is loaded with FC50. The LDX instruction is using indirect accumulator A offset addressing on the U pointer register. Therefore the memory location which contains the address of the operand is 0010 + FC50 = FC60. From the program listing you find 06 at this address, which is the high byte of the operand's address. The low byte is found at address FC61. Thus the operand is located at address 06FF. At this address you find the value F5. But, the X index register is a 16-bit register. Therefore F5 forms the high byte to be loaded and the contents of the next consecutive memory location will form the low byte to be loaded. Thus the value F51B will be loaded into the index register.

28. $11010110_2 = D6_{16}$ (reference, Table 2-6)

29. 4 or 5: the instruction op-code byte(s), followed by a post byte, followed by a 2-byte address

30. Accumulator A and the direct page register (reference Fig. 2-7)

CHAPTER 3

6809 Registers and Data Movement Instructions

INTRODUCTION

As stated previously, the main design goal for the 6809 was to make it a *super* 8-bit processor, approaching 16-bit performance, without sacrificing compatibility with the 6800. Therefore, the 6800's architecture was expanded and the instruction set "cleaned up" to provide more efficient processing. The 72 fundamental 6800 instructions have been reduced to 59 for the 6809. Combined with the various addressing modes discussed in the previous chapter, these 59 fundamental instructions, however, allow for 1464 unique operations within the 6809, as compared with 197 for the 6800. Prior to defining the 6809's instruction set, Motorola conducted a survey of 6800 users to determine which 6800 instructions were used most frequently and which were not so well used. The most frequently used instructions were the loads and stores, followed by subroutine calls, branches, compares, increments/decrements, clears, and adds/subtracts, in that order. Therefore, the major 6809 instruction set improvements have been made in the *data movement* instruction category: loads, stores, transfers, exchanges, etc. For example, data transfers and exchanges may be made between any two like-size registers (R1, R2) using the TFR R1, R2 and EXG R1, R2 instructions. There are 42 valid combinations of the TFR instruction and 21 valid combinations of the EXG instruction. You therefore have just learned 63 6809 instructions with two mnemonics rather than 63 unique mnemonics of the form: transfer A to B (TAB), transfer A to CCR (TAP), transfer X to S (TXS), etc., as was the case with the 6800. This frees a number of op codes for more efficient use elsewhere. To maintain compatibility with the 6800 the 6809 source code (mnemonic code) has not changed, except for the new instructions. Changes have been made, however, at the object code (op code) level to allow for more efficient instruction coding.

You have already been familiarized with the 6809's internal register format (architecture). We will begin this chapter by discussing each internal register in detail. Then a discussion of the 6809 instruction set will begin and continue through the next two chapters. For convenience, we have broken the 59 fundamental 6809 instructions down into six functional categories. They are: *data movement, arithmetic, logic, test, branch,* and *miscellaneous* instructions. The data movement instructions will be discussed in this chapter, with the remaining instructions being covered in Chapters 4 and 5.

If you are familiar with the 6800, you will find many of the 6809 instructions are familiar. The difference is in how they are used with the various addressing modes which were presented in Chapter 2. Therefore, a basic understanding of the 6800's instruction set will be assumed in this and subsequent chapters. If you are not familiar with the 6800 instruction set or need some "brushing up," consult a textbook on the 6800, such as *How to Program and Interface the 6800,* published by Howard W. Sams & Co., Inc.

OBJECTIVES

At the end of this chapter you will be able to do the following:

- List and describe the function of each 6809 internal register.
- Describe the function of each condition code register flag.
- Understand the relationship between the E and F condition code register flags.
- Explain the use of the load effective address (LEA) instructions.
- Explain how constant offsets or any accumulator contents may be added to any of the indexible registers(X, Y, S, U) using the LEA instructions.
- Describe how to transfer or exchange data between any two like-size registers.
- Write a program to load and store the direct page register (DPR).
- Understand how the X and Y registers may be used as stack pointers with the auto-increment/decrement mode of indexed addressing.

- Explain how to push/pull any or all of the internal 6809 registers on the S or U stacks.
- List the order of stacking with the PSHS and PSHU instructions.
- Define a "stack processor."

6809 INTERNAL REGISTER FORMAT

Before we begin a discussion of the 6809 instruction set, let's briefly review the function of each internal register available to the programmer. The programming register format is shown in Fig. 3-1. As stated earlier, this format adds one 8-bit and three 16-bit registers to the 6800 architecture. The new 8-bit register is the direct page register (DPR) whose function was discussed in Chapter 2. The three new 16-bit registers are accumulator D, the Y index register, and the U stack pointer. The remaining registers are identical with those in the 6800 and they perform similar functions. The following is a brief review of each register, as shown in Fig. 3-1.

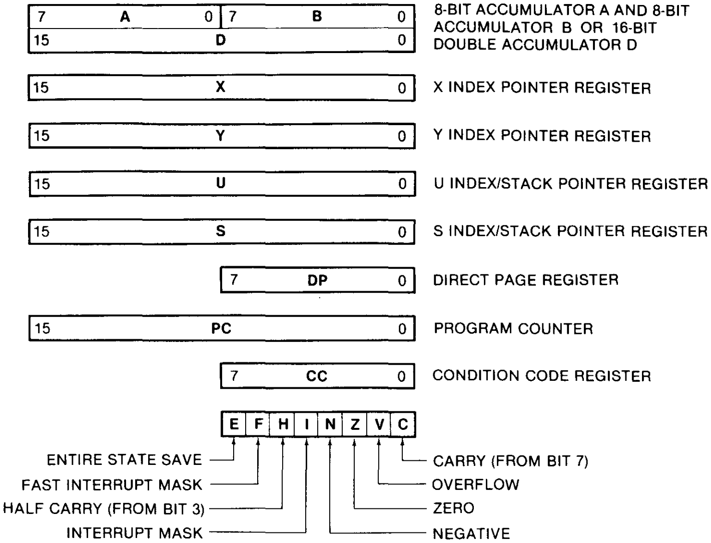


Fig. 3-1. 6809 programming register format.

Accumulators A, B, and D (A, B, D)

Accumulators A and B are 8-bit storage registers that are used to hold operands before they are used in an operation, and also to hold the results of various operations. As with the 6800, data may be loaded into, stored from, transferred to and from, exchanged with, added to, subtracted from, shifted, ANDed with, ORed with, XORed with, and compared to, the contents of either accumulator. Accumulator D is a 16-bit register which combines accumulators A and B. Accumulator A forms the high byte, with accumulator B forming the low byte, of accumulator D. Accumulator D is used in the same way as accumulators A and B, but it is used for 16-bit arithmetic operations. As you will see presently, there are several instructions associated with this accumulator. These instructions allow 16 bits of data to be loaded into, stored from, transferred to and from, exchanged with, added to, subtracted from, and compared with the contents of accumulator D.

X Register

The X register is a 16-bit register that can be used as a pointer register to store data or 16-bit addresses in conjunction with the indexed mode of memory addressing, as discussed in Chapter 2. As you discovered in Chapter 2, this register can be automatically incremented or decremented by either one or two counts. The 6809's instruction set will allow the X register to be loaded from and copied into memory. In addition, you can add to, subtract from, and compare its contents, as well as push and pull its contents from a memory stack. As you will soon see, this register may also be used as a software controlled *stack pointer* using auto-increment/decrement.

Y Register

The Y register functions are identical to those of the X register. Instructions are provided to manipulate this register in the same ways as the X register.

U Register

This 16-bit register has the same capabilities and can function in the same way as the X and Y registers, or it can be used to define a *user* or *software* memory stack, which is not used automatically by the 6809's interrupts as in the case of the *hardware* stack. As you will see, the same instructions which operate on the X and Y registers are also available to the U register. Moreover, push and pull instructions are provided such that you may stack any or all of the internal 6809 registers' contents in an R/W memory stack defined by the U register. We will refer to such a stack as the *U stack*.

S Register

The S register corresponds to the 6800's stack pointer. It is a *hardware* stack pointer and thus provides for *automatic* stacking of the internal 6809 register information during subroutine calls and interrupts. However, it may also be used as a pointer register for any of the indexed modes of addressing, and it has associated instructions which allow it to be loaded, stored, added to, subtracted from, and compared. In addition, it can be used to define a software stack since push and pull instructions are provided so that you can stack any or all of the 6809 internal register contents in the S stack.

Direct Page Register (DP)

This is an 8-bit register which is used to define the high-memory address byte (page) for the direct mode of memory addressing (see Chapter 2).

Program Counter (PC)

The program counter is a 16-bit binary counter that provides instruction sequencing. In addition, it is used in the calculation of branch destinations for branch relative addressing and effective addresses for the program counter relative mode of memory addressing (see Chapter 2).

Condition Code Register (CCR)

As with the 6800 the 6809 CCR is an 8-bit status flag register. The functions of the first 6 bits (0 through 5) are identical with those of the 6800. These first 6 CCR bits are the carry (C), twos complement overflow (V), zero (Z), negative (N), interrupt (I) and half-carry (H) flags, respectively. Let us briefly review the function of each of these flags:

Carry Flag (C—Bit 0)—The carry flag is set (=1) whenever there is a "last" carry (or borrow) generated by the eighth bit column in an arithmetic operation, or, as you will see later, whenever data are "moved" or rotated within the accumulators. The carry bit can sometimes be thought of as a "ninth bit" on the most significant end of the accumulator.

Twos Complement Overflow Flag (V—Bit 1)—The V flag is set whenever a twos complement overflow occurs. This condition is a result of twos complement arithmetic. If you add two positive numbers, you expect to obtain a positive result. If you add two negative numbers, you expect to obtain a negative result. Bit 7 in twos complement arithmetic indicates the sign of the number. Whenever two twos complement numbers are added, or subtracted, a carry or borrow generated in the D6 column would overflow into the sign bit, bit D7. If this happens, the sign of the result would be incorrectly changed. Thus, the V flag will set to indicate this error condition. Note N exclusive-OR V ($N \forall V$) will always give the correct sign, even if the result sign is wrong. Loads, stores, and logical operations will clear V.

Zero Flag (Z–Bit 2)—The Z flag is set (=1) whenever the result of an operation or data transfer is identically equal to zero. It can also be used to *reflect* the equal or unequal condition between 2 data bytes that are being compared. If the bytes are identical, the Z flag will set.

Negative Flag (N–Bit 3)—The N flag is the twos complement sign bit and can be thought of as being connected directly to the msb (bit 7) of the result. A 1 indicates a negative result, with a 0 indicating a positive result. If a twos complement overflow occurs, the sign of the result (N flag) will be incorrect. Therefore signed branches will always test the condition N $\forall$ V to obtain a valid sign result.

Interrupt Flag (I–Bit 4)—This flag is used in conjunction with external i/o device interfacing and it will be discussed in detail later. Briefly, when this flag is set, it will *not* allow the 6809 to be interrupted by the interrupt request (IRQ) line. As you will discover later, the 6809 interrupts are: NMI, FIRQ, IRQ, RESET, SWI1, SWI2, and SWI3. All of the above, *except* SWI2 and SWI3, will automatically set the I flag during their respective interrupt sequences.

Half-Carry Flag (H–Bit 5)—The half-carry flag is used to indicate a carry from bit 3 to bit 4 in the arithmetic logic unit (alu). It will be set if a carry from bit 3's column to bit 4's column took place during an addition operation only (ADD or ADC). The 6809 uses this flag to implement the decimal-adjust instruction that allows it to operate on binary-coded decimal, or bcd, values.

Fast Interrupt Mask (F–Bit 6) and Entire State Saved (E–Bit 7) Flags—The two new 6809 flags which were mentioned in Chapter 1 are the *fast interrupt mask (F)* and *entire state saved (E)* flags, bits 6 and 7 of the CCR, respectively. Recall that these 2 bits are not used and are permanently set (logic 1) in the 6800. Associated with these flags is a hardware interrupt line called the *fast interrupt request* ($\overline{\textit{FIRQ}}$) line. Like the 6800, the 6809 automatically stacks *all* the internal register data when an interrupt request ($\overline{\text{IRQ}}$) is acknowledged. There is, however, no need for the computer to waste its time stacking *all* of the internal register data if it is known ahead of time that the interrupt service routine will either use the existing internal register contents *or* it is not necessary to save these contents. For this reason the 6809 includes a fast interrupt request ($\overline{\text{FIRQ}}$) in addition to the standard interrupt request ($\overline{\text{IRQ}}$). When an $\overline{\text{FIRQ}}$ is acknowledged, the 6809 will finish executing its current instruction and then it automatically "stacks" *only* the contents of the program counter and the condition code register.

Program control is then passed to the $\overline{\text{FIRQ}}$ interrupt service routine via the FIRQ interrupt vector. (Interrupt vectors are discussed in Chapters 5 and 6.) The *F flag* of the condition code register is a mask bit for all $\overline{\text{FIRQ}}$ interrupts. When it is cleared (logic 0), fast interrupts are allowed; however, when it is set (logic 1), fast interrupts are masked out or disallowed. The *E flag* of the condition code register is used in conjunction with the fast interrupt request ($\overline{\text{FIRQ}}$) and all other 6809 interrupts. When an $\overline{\text{FIRQ}}$ is acknowledged, the E flag is *automatically* cleared (logic 0), indicating that *only* the program counter and condition code register have been stacked. However, the E flag is set (logic 1) when an $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$, or SWI is acknowledged, indicating that *all* the internal working registers have been stacked. The E flag of the *stacked* condition code register is then used by the 6809 during a return from interrupt (RTI) to determine the extent of "unstacking" that is required. All the 6809 interrupts, *except* $\overline{\textit{FIRQ}}$ will cause the E flag to set. We will discuss the 6809's interrupt structure in detail later. At this time it is only necessary that you understand the basic functions of the E and F flags.

Now that you are familiar with the 6809 addressing modes and internal register structure, you are ready to study the 59 fundamental instructions that make up the 6809 instruction set.

DATA MOVEMENT INSTRUCTIONS

The data movement instructions will perform the following operations:

- Load and store registers, A, B, D, X, Y, U, and S.
- Transfer and exchange between any two *like-size* internal registers.
- Load an effective address into the X, Y, U, and S registers.
- Push and pull to or from the S or U stacks, any or all of the internal registers.
- Add 8-bit data to X, Y, U, and S.
- Add 16-bit data to X, Y, U, and S.
- Add A, B, and D to X, Y, U, and S.
- Increment and decrement X, Y, U, and S by one or two counts.

The instruction mnemonics which will accomplish the above operations are listed in Table 3-1 with their respective operation symbols. First, note from the information in Table 3-1 that you can load or store any of the accumulators (A, B, or D) or any of the indexible registers (X, Y, S, or U). As you will see at the end of this chapter, the *load* operation may be performed using immediate, direct, extended, program counter relative, or indexed addressing.

Table 3-1. Data Movement Instructions

Mnemonic		Operation	Operation Symbol
LD	LDA	Load A immediate, or from memory	M→A
	LDB	Load B immediate, or from memory	M→B
	LDD	Load D immediate, or from memory	M:M + 1→D
	LDS	Load S immediate, or from memory	M:M + 1→S
	LDU	Load U immediate, or from memory	M:M + 1→U
	LDX	Load X immediate, or from memory	M:M + 1→X
	LDY	Load Y immediate, or from memory	M:M + 1→Y
ST	STA	Store A to memory	A→M
	STB	Store B to memory	B→M
	STD	Store D to memory	D→M:M + 1
	STS	Store S to memory	S→M:M + 1
	STU	Store U to memory	U→M:M + 1
	STX	Store X to memory	X→M:M + 1
	STY	Store Y to memory	Y→M:M + 1
TFR	R1, R2	Transfer R1 to R2	R1→R2
EXG	R1, R2	Exchange R1 with R2	R1↔R2
LEA	LEAS	Load effective address into S	EA→S
	LEAU	Load effective address into U	EA→U
	LEAX	Load effective address into X	EA→X
	LEAY	Load effective address into Y	EA→Y
PSH	PSHS	Push onto hardware stack (S)	None
	PSHU	Push onto user stack (U)	None
PUL	PULS	Pull from hardware stack (S)	None
	PULU	Pull from user stack (U)	None

The *store* operation can be performed using direct, extended, program counter relative, or indexed addressing. The op codes for these instructions will be given shortly. However, for now, let's concentrate on the meaning of the instructions.

From the load and store operation symbols, you can see that whenever a 16-bit register is involved, 2 bytes of memory or immediate data must be provided. These 2 bytes are symbolized by *M:M+1*. The instruction will determine the effective address M, with M+1 being the next consecutive memory location's address. Memory address M is associated with the high byte of the register being operated upon, and M+1 is associated with the low register byte. In the case of the immediate addressing mode, M and M+1 are the actual high and low data bytes, respectively. For example, the 3-byte instruction

```
LDX $$
 F1
 C5
```

means to load the X register with the *contents* of memory locations F1C5 and F1C6. Here, M equals F1C5, whose *contents* will be loaded into the high byte of the X register. The value to be loaded into the low X register byte is found at memory location M+1, or F1C6 in this example. In contrast, the instruction

```
LDX #
F1
C5
```

would mean to actually load the X register with F1C5.

We should briefly consider the load and store operations on accumulator D. Since accumulator D is actually the concatenation of accumulators A and B, any operations on D obviously operate on A and B. For example, a load accumulator D operation (LDD) actually loads accumulator A with the high data byte and accumulator B with the low data byte. On the other hand, a store accumulator D operation (STD) actually stores accumulator A at memory address M and accumulator B at memory address M+1.

The transfer (TFR) and exchange (EXG) instructions are 2-byte instructions which utilize *register addressing*. The TFR or EXG instruction op code is followed by a post byte to define the registers (R1 and R2) involved in the data transfer or exchange. The definition of this post byte was discussed in the previous chapter under *register addressing*. Because of the various register combinations, there are actually *42* different transfers and exchanges which can be executed using these *two* instructions. The only requirement is that the two registers involved must be of like size.

The inclusion of the *load effective address* (LEA) instructions in the 6809's instruction set provides a unique capability that is not intuitively obvious on the surface. These instructions are used to load the indicated pointer register (S, U, X, or Y) with the value which results from the indexed mode of addressing's effective address (EA) calculation. Therefore the *effective address value is loaded* in the pointer register *rather than* the data at the effective address, as would be the case with a standard load instruction. *All* the indexed modes of addressing, including program counter relative, are available to the LEA instructions.

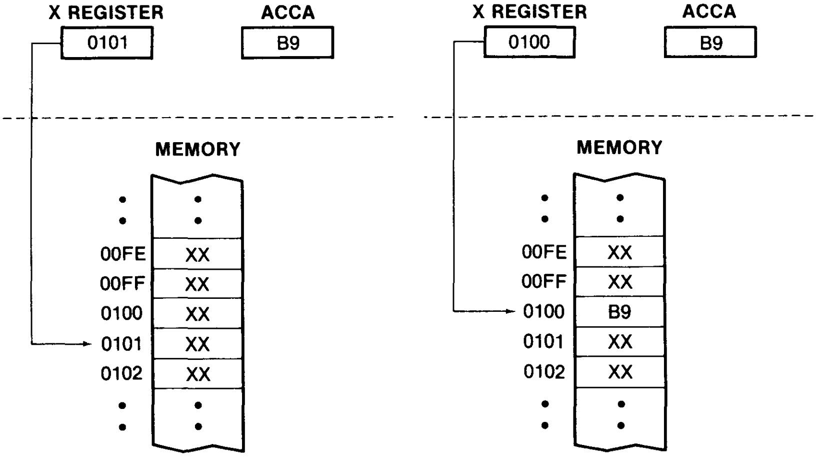
Using these instructions, the address of a data byte can be calculated by the main program using the indexed mode of addressing then passed on to a subroutine, since the pointer register contents are not destroyed when a subroutine is called. In addition, these instructions will allow you to *add* or *subtract* from any of the pointer registers. You can add or subtract a constant offset *or* any of the accumulator contents. For example, LEAX −5, X will subtract 5 from the X register; LEAS A, S will add the contents of accumulator A to the S register; and LEAY 10, U will add 10 to the U register and transfer the sum to the Y register. To increment the X register, you could use LEAX 1, X; while LEAY −1, Y would decrement the Y register. Obviously there are many more possibilities than the examples presented here. The LEA instructions used in conjunction with the various modes of indexed addressing will find many applications in your assembly-language programs.

The 6809 PSH and PUL instructions will allow you to push or pull one or any number of internal registers to and from the user (U) or hardware (S) memory stack. Before we discuss these instructions, let's notice something about the 6809 stacking. First, the user stack pointer (U) defines the beginning of a software stack which is not affected by the use of subroutine calls or interrupts. Therefore the U stack is controlled completely by the programmer. In contrast, the hardware stack pointer (S) defines the beginning of a hardware stack that is used automatically by the 6809 to stack internal register data during subroutine calls and interrupt servicing. During a subroutine call the program counter is automatically saved on the S stack, and during an interrupt all the internal registers are automatically saved on the S stack. The fast interrupt request (FIRQ) is an exception, since only the condition code register and program counter are "stacked." More will be said later about the use of the S stack with subroutines and interrupts. Recall that both the U and S stack pointers can be used as index registers in the indexed mode of addressing, but *in addition* they support the *push* and *pull* instructions. This dual function allows the 6809 to be used as a *stack processor*, and thus it can readily support such high-level languages as Pascal, FORTRAN, and so on.

- *stack processor*—a processor which allows unlimited memory stacking for easy processing of multiple-level interrupts and subroutine nesting.

In the 6800 the stack pointer "points" to the next *available* location on an R/W memory stack. However, with the 6809 the stack pointer actually points to the *last value* pushed onto the stack, sometimes referred to as the *top* of the stack. This change was made to allow the 6809's X and Y registers to function also as software stack pointers in the auto-increment/decrement mode of indexed addressing. For example, an STA, −X is equivalent to a *pull-accumulator-A*-from-the-stack instruction, using a stack defined by the address in the X register. The instruction LDA, X+ would pull the top byte of a stack defined by the X register into accumulator A. Before-and-after illustrations of the operation of these instructions are shown in Figs. 3-2 and 3-3.

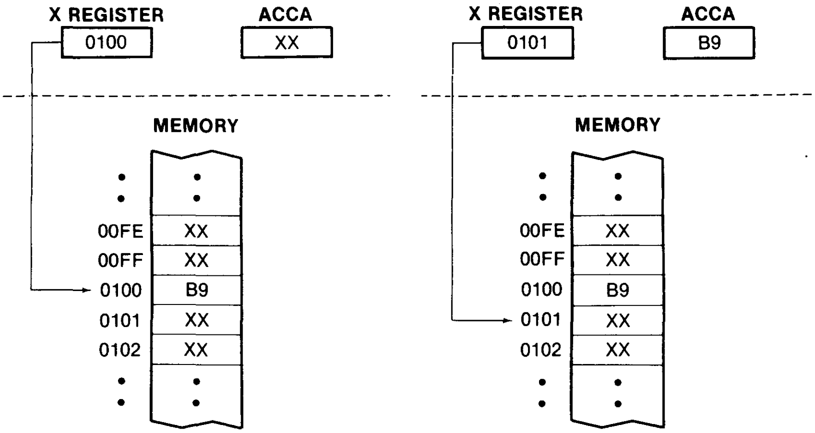
The same types of instructions can be used to push and pull ac-



(A) 6809 chip and X stack before STA, —X.

(B) 6809 chip and X stack after STA, —X.

Fig. 3-2. Before and after STA, —X.

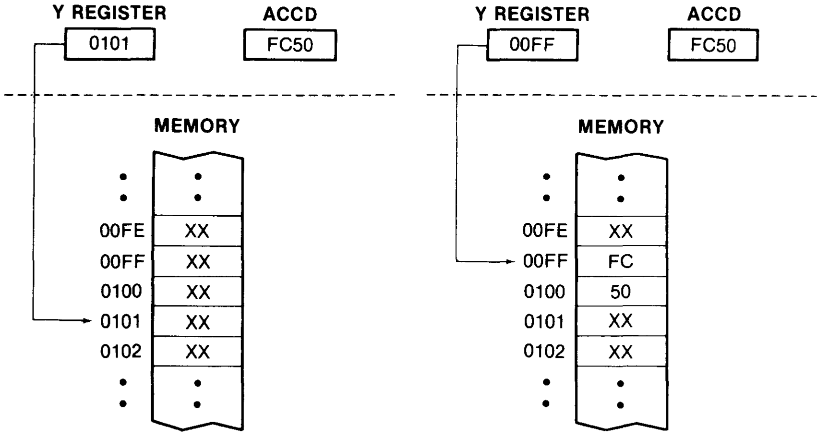


(A) 6809 chip and X stack before LDA, X+.

(B) 6809 chip and X stack after LDA, X+.

Fig. 3-3. Before and after LDA, X+.

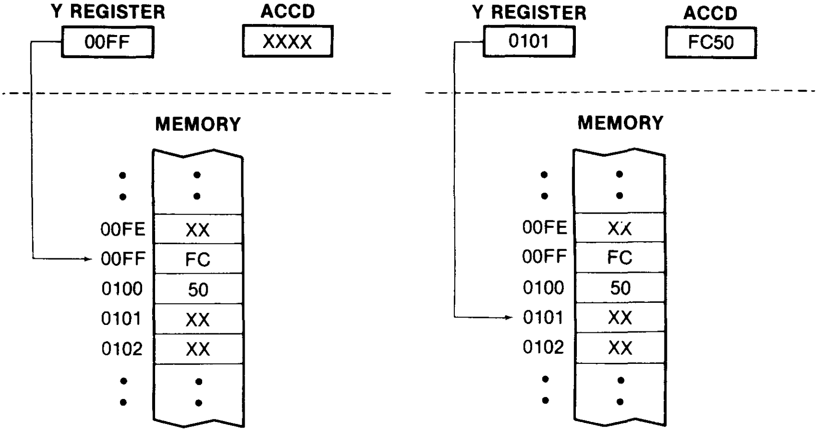
cumulator data to and from stacks defined by other pointer registers by replacing the pointer register designations in the above instructions with Y, S, or U, for example, STA, —Y; LDA, Y+; etc. In addition, accumulators B or D and registers X, Y, S, or U may be pushed or pulled using the remaining store (ST) and load (LD) instructions given in Table 3-1 (STB, STD, LDY, LDD, etc.). However, when you push or pull to or from the 16-bit register, you must use an instruction that causes an auto-increment/decrement



(A) 6809 chip and Y stack before STD, − −Y.

(B) 6809 chip and Y stack after STD, − −Y.

Fig. 3-4. Before and after STD, − −Y.



(A) 6809 chip and Y stack before LDD, Y++.

(B) 6809 chip and Y stack after LDD, Y++.

Fig. 3-5. Before and after LDD, Y+ +.

by 2 since the data will require 2 *bytes* in the stack. For example, STD, − −Y would push the contents of accumulator D onto a stack defined by the Y register, while LDD, Y++ would pull the top 2 bytes of a stack defined by the Y register into accumulator D. Before and after illustrations of these instructions are shown in Figs. 3-4 and 3-5.

So, you might be wondering *why* we need the PSH and PUL instructions listed in Table 3-1. The advantage is that, with these

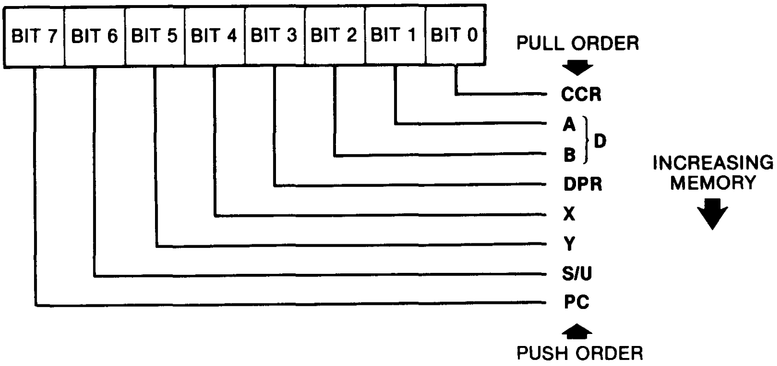


Fig. 3-6. Push/pull post-byte format.

instructions, you can push or pull any or *all* of the internal 6809 registers onto a stack defined by the S or *U* registers with *one* 2-byte instruction. The PSH and PUL instructions require 2 bytes: the instruction op code followed by a post byte. The post byte defines which registers are to be pushed or pulled. The push/pull post-byte format is shown in Fig. 3-6. A logic 1 in a bit position will cause that respective register to be pushed/pulled. The order in which the registers are pushed or pulled is also shown in Fig. 3-6. For example, if you wanted to push accumulators A, B, and the program counter (PC) onto the U stack, the proper assembly code would be: PSHU A, B, PC. The required post byte would then be 1000 0110_2 or 86_{16}.

When using PSH*U* or PUL*U*, bit 6 of the post byte will correspond to the S register being pushed or pulled. When using PSHS or PULS, bit 6 corresponds to the U register. Not *all* the registers need to be stacked, but the order given in Fig. 3-6 is the same even if only a subset is stacked. You will see shortly that this order of stacking is the same when the 6809 services interrupts. When a 16-bit register is pushed, the low byte is pushed first, followed by the high byte. And when a 16-bit register is pulled from the stack, the high byte is pulled first, then the low byte. Note from Fig. 3-6 that stacking all of the internal 6809 registers would require 12 bytes of stack memory. Also, accumulator D is not specifically stacked since it is simply the concatenation of accumulators A and B.

The data handling instruction *op codes* are listed alphabetically in Table 3-2. Note that, besides providing the instruction op codes, this table also provides the number of bytes and MPU cycles required for a given instruction. In addition, the effect that an instruction has on the condition code register flags is also provided. The notes and legend provided with this table will also apply to all subsequent op-code tables.

Table 3-2. Data Movement Instruction Op Codes

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]	#						
EXG	R1,R2	1E	7	2																R1↔R2[2]	•	•	•	•	•
LD	LDA				96	4	2	B6	5	3	86	2	2	A6	4+	2+				M→A	•	↕	↕	0	•
	LDB				D6	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 DE	6	3	10 FE	7	4	10 CE	4	4	10 EE	6+	3+				M:M + 1→S	•	↕	↕	0	•
	LDU				DE	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 9E	6	3	10 BE	7	4	10 8E	4	4	10 AE	6+	3+				M:M + 1→Y	•	↕	↕	0	•
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	•	•	↕	•	•
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	•	•	•	•	•
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 DF	6	3	10 FF	7	4				10 EF	6+	3+				S→M:M + 1	•	↕	↕	0	•
	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 9F	6	3	10 BF	7	4				10 AF	6+	3+				Y→M:M + 1	•	↕	↕	0	•
TFR	R1,R2	1F	7	2																R1→R2[2]	•	•	•	•	•

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
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. SW1 sets I&F bits. SW12 and SW13 do not affect I&F.
7. Conditions codes 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);
~	Number of MPU cycles;
#	Number of program bytes;
+	Arithmetic plus;
−	Arithmetic minus;
*	Multiply
$\overline{M}$	Complement of M;
→	Transfer Into;
H	Half-carry from bit 3;
N	Negative (sign bit)
Z	Zero (byte)
V	Overflow, twos complement
C	Carry from bit 7
↕	Test and set if true, cleared otherwise
•	Not affected
CC	Condition code register
:	Concatenation
∨	Logical OR
∧	Logical AND
⊻	Logical exclusive OR

This completes the discussion of the data handling instructions. The following examples illustrate the use of these instructions. Study the examples carefully. Then, if you have a Motorola MEK-6809D4 trainer, or an equivalent 6809-based system available to you, implement the following examples to verify their proper execution. (A discussion of the Motorola MEK6809D4 evaluation system is provided in Chapter 7.)

Example 3-1: Loading and Storing the Direct Page Register

Write an instruction sequence to load the direct page register with a new value and to store the old value.

Even though there is no LDDP or STDP, the exchange instruction (EXG) makes this a rather simple process. Suppose the new value to be loaded into the direct page register (DP) is E5, and the old DP contents are to be stored in memory location E510. Then, the following instructions should accomplish this task:

```
LDA #
E5
EXG A, DP
STA $
10
```

The above instruction sequence loads accumulator A with immediate data byte E5. The contents of accumulator A and the direct page register are then exchanged. Thus the new value (E5) is now in the DP register and the DP contents are in accumulator A. Since it is desired to store the old DP contents in memory location E510 and the DP register now contains E5, you can use direct addressing for the store operation. The store accumulator A direct operation will cause the contents of accumulator A (old DP register contents) to be stored at memory location E510 since the DP register contains E5 and forms the high memory location byte and the instruction contains 10, which forms the low memory location byte.

What post byte would be required for the EXG instruction in the above program?

To exchange accumulator A and the DP register, the required post byte would be $1000\ 1011_2$ or $8B_{16}$ (see Fig. 2-7).

Determine the number of bytes and MPU cycles required to execute this program.

Adding the number of bytes and MPU cycles required for each instruction from Table 3-2, you should find that this program requires 6 bytes of memory and 13 MPU cycles for execution.

Example 3-2: Adding to a Pointer Register

Write an instruction sequence to add the contents of accumulator A to the Y register, then transfer that result to the S register.

There are no 6809 *ADD* instructions which allow you to add to any of the pointer registers. However, the load effective address (LEA) instruction can be used for this purpose. The following *single* instruction will accomplish the above task:

```
LEAS A, Y
```

The effective address (EA) is formed by adding the contents of accumulator A to the Y register contents. The LEAS instruction then transfers this effective address to the S register.

Determine the post byte required for the above instruction.

The proper post byte would be $1010\ 0110_2$ or $A6_{16}$ (see Table 2-6).

The following instruction sequence would accomplish the same task. However, two instructions are required.

```
LEAY A, Y
TFR  Y, S
```

How many bytes and MPU cycles are required for each of the above programs?

From Table 3-2 the LEAS A, Y instruction requires 2+ bytes and 4+ MPU cycles. This means that to determine the *exact* number of bytes and MPU cycles, you must *add* the values from the 6809 indexed addressing modes table (Table 2-6) for the given operation. Referring to Table 2-6, you find that this operation requires *no* additional bytes and *one* additional MPU cycle. Therefore the LEAS A, Y instruction requires *2 bytes* and *5 MPU cycles* for execution. In the same way you find from Tables 3-2 and 2-6 that the second program requires *4 memory bytes* and *12 MPU cycles* for execution. Note the difference in the number of bytes and MPU cycles for the two different programs, both of which accomplish the same task.

Example 3-3: Stack Operations

Write an instruction sequence that will stack accumulator A, the DP register and the U stack pointer on a stack defined by the Y register.

To use the Y register as a stack pointer, you must use auto-increment/decrement by 1, or 2, depending upon the size of the register to be stacked or unstacked. The following instructions will accomplish the given task:

```
TFR DP, B
STD ,-- Y
STU ,-- Y
```

The transfer instruction transfers the DP register contents into accumulator B, since we cannot store the DP register directly. Now, since you must store both accumulators A and B, you can use an STD instruction, since accumulator D is simply the combination of A and B. The STD, – – Y instruction will push accumulator B onto the Y stack first, followed by accumulator A. The STU, – – Y instruction will then push the U register onto the Y stack. The low U register byte (U_L) will be pushed first, followed by the high U register byte (U_H). After the instructions have been executed, the Y register will point to the top of the stack or the memory location which contains U_H. Accumulator B will be at the bottom of the stack.

Determine the proper post byte for each of the above instructions.

The transfer (TFR) instruction post byte will be $1011\ 1001_2$ or $B9_{16}$ since the source register is the direct page register and the destination register is accumulator B (see Fig. 2-7). The proper post byte for the STD, – – Y instruction is $1010\ 0011_2$ or $A3_{16}$. The post byte for the STU, – – Y instruction will be the same since both instructions use the same mode of indexed addressing and operate on the same pointer register (see Table 2-6).

Write an instruction sequence to unstack the above registers.

The correct instruction sequence would be:

```
LDU, Y++
LDD, Y++
TFR B, DP
```

Note that you will unstack the register data in the *reversed order* to that in which they were originally stacked, since the *last-in, first-out* (*LIFO*) principle applies. The proper post bytes for the above instructions would be: $A1_{16}$, $A1_{16}$, and $9B_{16}$, respectively (see Table 2-6 and Fig. 2-7).

Example 3-4: Stack Operations

Write an instruction sequence that will stack accumulator A, the CCR, DPR, PC, and X register on the user stack.

This task only requires one 2-byte instruction:

```
PSHU A, CC, DP, PC, X
```

The first byte of the above instruction will be the instruction op code, which you will find to be 36_{16}, from Table 3-2. The second byte is the post byte. From Fig. 3-6 you can determine that the post byte required to stack the given register data is $1001\ 1011_2$ or $9B_{16}$.

In what order would the above registers be stacked?

As shown in Fig. 3-6 the stacking order would be: PC, X, DPR, A, CCR. The CCR would be at the top of the stack (lowest memory address) and the PC at the bottom of the stack (highest memory address).

Chart 3-1. Performing Additional Data Transfer Operations With 6809 Instructions

Exchange A and X $(X_H \rightarrow A)$ $(A{:}X_L \rightarrow X)$	=	PSHS A,B TFR X,D PULS A TFR D,X PULS B
Exchange B and A $(X_L \rightarrow B)$ $(X_H{:}B \rightarrow X)$	=	PSHS A PSHS B TFR X,D PULS B TFR D,X PULS A
Transfer A to X $(A{:}X_L \rightarrow X)$	=	PSHS X STA 0,S PULS X
Transfer B to X $(X_H{:}B \rightarrow X)$	=	PSHS X STB 1,S PULS X

Finally, Chart 3-1 shows you how to perform some additional data transfer operations using the instructions discussed in this chapter.

REVIEW QUESTIONS

1. List the 8-bit registers of the 6809.

2. List the 16-bit registers of the 6809.

3. Which of the 16-bit registers can be used as index registers?

4. **Which register is used automatically by the 6809 for stacking data during subroutines and interrupts?**

5. **What is the difference between the X and Y registers?**

6. **When is the Z flag *set*?**

7. **Which CCR flag indicates a sign error in the result of an operation?**

8. **For the 6809 to acknowledge a fast interrupt request, the F flag must be**

__________.

9. **Which registers are automatically stacked during an FIRQ?**

10. **What is the function of the E flag?**

11. **Which of the internal 6809 registers can be loaded and stored?**

12. **Explain the difference between:**

```
LDX #        and        LDX $$
 FC                      FC
 50                      50
```

13. EA→Y is the operation symbol for the __________ instruction.

14. Write an instruction that will subtract 7 from the contents of the S register.

15. Write an instruction that will add the contents of accumulator B to the Y register.

16. Write an instruction that will add the contents of accumulator A to the U register, then transfer the result to the X register.

17. Which of the 6809's registers may be used as stack pointers?

18. With the 6809 a stack pointer always points __________ (where?).

19. How are the X and Y registers used as stack pointers?

20. Write an instruction sequence which will stack the Y register, accumulator A, and the condition code register, in that order on the X stack.

21. Why wouldn't the following be a correct answer to the preceding question?

```
STY, ——X
TFR CC, B
STD, ——X
```

22. Write an instruction sequence that will unstack the stack created in Question 20.

23. What is the order of stacking when using the PSH instructions?

24. Write an instruction which will stack both accumulators (A and B), the X register, and the program counter on the U stack.

25. Write the correct op-code listing for the preceding instruction.

ANSWERS

1. Accumulator A (A)
   Accumulator B (B)
   Direct page register (DPR)
   Condition code register (CCR)

2. Accumulator D (D)
   Program counter (PC)
   X register (X)
   Y register (Y)
   U register (U)
   S register (S)

3. The X, Y, S, or U register

4. The S register

5. None. They can be used in exactly the same manner and are operated on by the same set of instructions.

6. When the result of an operation is *zero*

7. The V flag

8. Cleared (logic 0)

9. Only the program counter and condition code register

10. The E flag is used in conjunction with the F flag. The E flag is automatically cleared when a fast interrupt request is acknowledged and set for all other

interrupts. This flag is used by the 6809 for unstacking purposes to determine the amount of unstacking required.

11. Accumulators A, B, and D and the X, Y, S, and U registers.

12. LDX # loads the X register immediately with FC50, while LDX $$ loads the high X register byte (X_H) with the *contents* of FC50 and the low register byte (X_L) with the *contents* of FC51.

13. Load effective address into Y (LEAY)

14. LEAS –7, S

15. LEAY B, Y

16. LEAX A, U

17. The X, Y, S, and U registers

18. To the "top" of the stack or last value pushed onto the stack

19. By using auto-increment/decrement indexed addressing on the X and Y registers in conjunction with the various load and store instructions

20. STY, – –X
    STA, –X
    TFR CC, B
    STB, –X

21. This instruction sequence will stack the Y register, the CCR, then accumulator A which is *not* the specified order of stacking in Question 20.

22. LDB, X+
    TFR B, CC
    LDA, X+
    LDY, X++
    Note that the data must be unstacked in the reverse order of stacking.

23. CC
    A Increasing
    B memory
    DP
    X
    Y
    U/S
    PC

24. PSHU A, B, X, PC

25. From Table 3-2 and Fig. 3-6 the correct op-code listing would be:
    PSHU op code: 36_{16}
    Post byte: 96_{16}

CHAPTER 4

Arithmetic, Logic, and Test Instructions

INTRODUCTION

Most of the arithmetic, logic, and test instructions of the 6809 are very similar to those of the 6800, except in the way they can be used with the various addressing modes. There are, however, a few surprises. You will see arithmetic and test instructions which allow you to operate on accumulator D. One of the most significant of these is the multiply (MUL) instruction that allows you to multiply the contents of accumulators A and B, with a 16-bit product being generated in accumulator D. In addition, you will find an instruction that will allow you to extend the sign bit of accumulator B into accumulator D such that 8-bit signed (twos complement) values can be extended into 16-bit signed (twos complement) values.

The logic instructions of the 6809 allow you to perform the standard logic operations of AND, OR, XOR or EOR, complement, shift, etc., on the contents of accumulators A, B, and memory. However, rather than have separate, inherent instructions to set and clear the condition code register flags, the 6809 uses two logic instructions, ANDCC and ORCC. These two 6809 instructions replace six 6800 instructions and therefore release valuable op codes for more efficient use elsewhere.

The discussions and examples in this chapter will concentrate on *how* to apply the arithmetic, logic, and test instructions using the addressing modes presented in Chapter 2. Only the new instructions will be defined and discussed separately. Again, a basic understanding of the 6800 instruction set is assumed.

OBJECTIVES

At the end of this chapter you will be able to do the following:

- Write a program to multiply two values and store the result indirectly.
- Understand why the 6809 multiply (MUL) instruction is an unsigned multiply.
- Write an instruction sequence to compute the U stack pointer.
- Determine the correct op-code listing, given an assembly-language program listing.
- Determine the MPU cycles required to execute a given instruction sequence.
- Explain how to set and clear the condition code register flags.
- Describe the three main categories of test instructions.
- Understand which CCR flags are affected by the various test instructions.
- Determine CCR flag status after execution of a given test instruction.

ARITHMETIC INSTRUCTIONS

The 6809 arithmetic instructions can be used to perform the following operations:

- Add and subtract memory into A, B and D.
- Add and subtract memory with carry into A and B.
- Increment and decrement A, B, D, and memory.
- Clear A, B, and memory.
- Negate A, B, and memory.
- Operate on bcd numbers.
- Multiply A times B.
- Sign extend B into D.

The 6809 arithmetic instructions are listed in Table 4-1. If you are familiar with the 6800, you will recognize many of the instruction mnemonics. The difference between these and the similar 6800 instructions is simply in how they can be applied using the numerous 6809 addressing modes. Therefore a detailed description of each instruction will not be given here, but examples will be given presently, showing various ways in which many of these instructions may be used. However, three *new* instructions do justify explanation. They are: ABX (add B to X), MUL (multiply), and SEX (sign extend B into A).

The ABX instruction adds the unsigned (8-bit straight binary) contents of accumulator B to the X register contents, with the result

Table 4-1. 6809 Arithmetic Instructions

Mnemonic		Operation	Operation Symbol
ABX		Add B to X (unsigned)	B + X→X
ADC	ADCA	Add memory to A with carry	A + M + C→A
	ADCB	Add memory to B with carry	B + M + C→B
ADD	ADDA	Add memory to A	A + M→A
	ADDB	Add memory to B	B + M→B
	ADDD	Add memory to D	D + M:M + 1→D
CLR	CLRA	Clear A	0→A
	CLRB	Clear B	0→B
	CLR	Clear memory	0→M
DAA		Decimal adjust A	None
DEC	DECA	Decrement A	A − 1→A
	DECB	Decrement B	B − 1→B
	DEC	Decrement memory	M − 1→M
INC	INCA	Increment A	A + 1→A
	INCB	Increment B	B + 1→B
	INC	Increment memory	M + 1→M
MUL		Multiply A times B (unsigned)	A × B→D
NEG	NEGA	Negate A (twos complement)	$\overline{A}$ + 1→A
	NEGB	Negate B (twos complement)	$\overline{B}$ + 1→B
	NEG	Negate memory (twos complement)	$\overline{M}$ + 1→M
SBC	SBCA	Subtract memory from A with borrow	A − M − C→A
	SBCB	Subtract memory from B with borrow	B − M − C→B
SEX		Sign extend B into A	None
SUB	SUBA	Subtract memory from A	A − M→A
	SUBB	Subtract memory from B	B − M→B
	SUBD	Subtract memory from D	D − M:M + 1→D

being placed in the X register. ABX is a 1-byte inherent instruction. This instruction is very similar to LEAX B, X. However, ABX treats accumulator B as an *unsigned positive* offset between 0 and 255, and LEAX B, X treats accumulator B as a twos complement signed offset between −128 and +127. In addition, the LEAX B, X instruction requires 2 bytes compared to 1 for ABX. Therefore it will be advantageous to use ABX when it is known that a relatively large *positive offset* is to be placed, or generated, in accumulator B.

In high-level languages it is often necessary to perform calculations on various arrays of information. These calculations frequently require multiplication operations, and therefore a multiply instruction (MUL) has been added to the 6809 instruction set. The MUL instruction permits you to multiply directly in the 6809 without any special algorithms. The *unsigned* (8-bit binary) contents of accu-

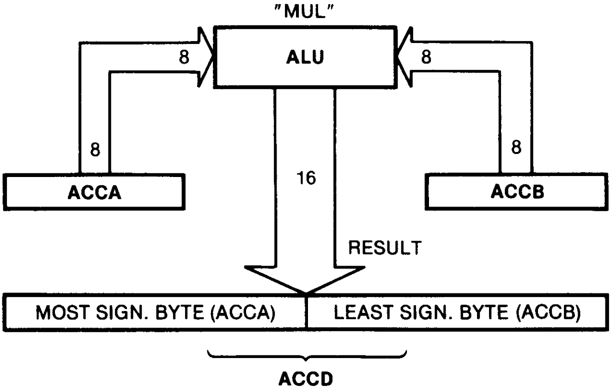


Fig. 4-1. Execution of the multiply (MUL) instruction.

mulators A and B are multiplied together. An *unsigned* 16-bit result is generated and placed in accumulator D. The internal execution of this instruction is shown in Fig. 4-1. Note from the figure that the contents of accumulators A and B are multiplied in the arithmetic logic unit (alu), with the result being placed in accumulator D. Since accumulators A and B combine to form accumulator D, the most significant byte of the result is actually placed in accumulator A and the least significant result byte is placed in accumulator B. Therefore the previous contents (multiplier and multiplicand) of these accumulators are lost. The MUL instruction is a 1-byte inherent instruction. It has been made an *unsigned* multiply to facilitate multiple-precision (multibyte) multiplications. A signed multiply would not lend itself easily to such a task.

Finally, the 6809 does have SEX appeal, by way of the sign extend instruction. This instruction will allow you to convert a signed (twos complement) 8-bit value in accumulator B to a 16-bit twos complement value in accumulator D. The SEX instruction actually extends the most significant bit (sign bit) of accumulator B into the most significant bit of accumulator A. This will allow 8-bit signed numbers to be easily converted to 16-bit numbers for subsequent internal 16-bit operations. For example, data communications with the outside world use 8-bit words, but internally the 6809 can perform 16-bit operations; therefore, such a conversion might be desirable. SEX is also a 1-byte inherent instruction.

Now, let's look at some example programs which utilize these new instructions and the other arithmetic instructions. If you have a Motorola MEK6809D4 or equivalent trainer available to you, you may wish to implement the example programs on your trainer to verify their proper execution. The arithmetic instruction op codes are provided in Table 4-2.

Table 4-2. Arithmetic Instruction Op Codes

Instruction/ Forms		6809 Addressing Modes																		Description	5	3	2	1	0
		Inherent			Direct			Extended			Immediate			Indexed			Relative								
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	Description	H	N	Z	V	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	↕	↕	↕	↕	↕
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	7	3				6F	6+	2+				0→M	•	0	1	0	0
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	6	2	7A	7	3				6A	6+	2+				M − 1→M	•	↕	↕	↕	•
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	•	↕	↕	↕	•
MUL		3D	11	1																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+	2+				$\overline{M}$ + 1→M	8	↕	↕	↕	↕
SEX		1D	2	1																Sign extend B into A	•	↕	↕	0	•
SBC	SBCA				92	4	2	B2	5	3	82	2	2	A2	4+	2+				A − M − C→A	8	↕	↕	↕	↕
	SBCB				D2	4	2	F2	5	3	C2	2	2	E2	4+	2+				B − M − C→B	8	↕	↕	↕	↕
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	•	↕	↕	↕	↕

Example 4-1: Using the Multiply (MUL) Instruction

Describe the task of the following instruction sequence:

```
LDA #
 20
LDB #
 0A
MUL
STD [, X]
```

The instruction sequence first loads accumulator A with 20, then loads accumulator B with 0A. The MUL instruction then multiplies these values and places the 16-bit result, 0140, in accumulator D. Note, since accumulator D is the combined contents of accumulators A and B, accumulator A should now contain 01 and accumulator B will contain 40. The previous accumulator contents are lost. Finally, the result is stored by the STD instruction using *indirect* addressing, with the X pointer register (no offset). Therefore, the contents of the X register specify the address where you will find the high *address* byte of the high result byte storage location. The low *address* byte of the high result byte storage location will be found at the address specified by the index register contents plus 1, or X+1. For example, suppose the index register contains 0100. Then, X=0100 and X+1=0101. Also, suppose that memory location 0100 contains FC and memory location 0101 contains 50. Then the high result byte is stored in memory location FC50 and the low result byte is stored in memory location FC51.

Write the correct op-code listing for the above instruction mnemonic sequence.

From Tables 2-6, 3-2, and 4-2 the proper op-code listing would be:

```
LDA #        86
 20          20
LDB #        C6
 0A          0A
MUL          3D
STD [, X]    ED
             94
```

Verify that the above listing is correct.

By observation, you see that the above instruction sequence requires 7 bytes. How many MPU cycles would be required to execute this instruction sequence?

From the same set of tables (Tables 2-6, 3-2, 4-2) you can determine that 9 MPU cycles would be required.

Example 4-2: Computed Stack Pointer

Write an instruction sequence that will compute an X register offset by adding the contents of two consecutive memory locations which are specified by the Y register. Once the offset is computed, add it to the X register to create an X stack pointer. Then, save the S and U registers on the X stack.

The following instruction sequence will perform the above task:

```
 LDB  , Y
ADDB 1, Y
 ABX
 STS ,--X
 STU ,--X
```

The LDB instruction will load accumulator B with the contents of the memory location specified by the Y register, since a zero offset is being used. The ADDB instruction will then add the contents of the next consecutive memory location (specified by Y+1, since a constant offset of 1 is used) to accumulator B and place the result in accumulator B. Now that the offset is computed, the ABX will add the computed offset in accumulator B to the X register and place the result in the X register, thus providing the computer X stack pointer. The STS and STU instructions are then used with auto-decrement by 2 and the X register to save the S and U registers on the X stack.

Write the correct op-code listing for this instruction sequence.

From Tables 2-6, 3-2, and 4-2 the proper op-code listing would be:

```
 LDB , Y          E6
                  A4
ADDB 1, Y         EB
                  21
 ABX              3A
 STS ,--X         10
                  EF
                  83
 STU ,--X         EF
                  83
```

You should verify the proper instruction op codes and post bytes such that a complete understanding of the 6809 instruction set is achieved. A few comments are in order:

1. The constant offset of 1 in the ADDB instruction is included as part of the post byte (see Chapter 2).
2. The STS instruction requires a 2-byte op code (see Table 3-2).

How many MPU cycles would be required to execute the above program?

From the same set of tables you can determine that 29 MPU cycles would be required.

You will see more applications of the arithmetic instructions as we develop other examples throughout the text.

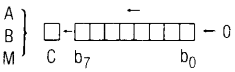
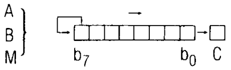
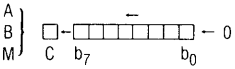
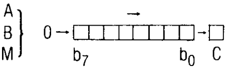
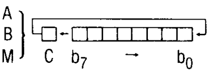
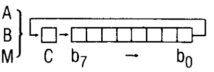
LOGIC INSTRUCTIONS

The logic instructions of the 6809 can be used to perform the following operations:

- AND, OR, and EOR memory into A or B.
- Shift and rotate memory, A, or B.
- Complement memory, A, or B.
- AND or OR immediate data into the CCR.

The 6809 logic instructions are listed in Table 4-3. As with the 6800, the standard logic operations of AND, OR, and EOR are provided along with the various arithmetic and logic shift functions. The reader is encouraged to consult a 6800 text if a more detailed discus-

Table 4-3. 6809 Logic Instructions

Mnemonic		Operation	Operation Symbol
AND	ANDA	AND memory with A (∧)	A ∧ M→A
	ANDB	AND memory with B	B ∧ M→B
	ANDCC	AND condition code register	CC IMM CC
ASL	ASLA	Arithmetic shift left A	
	ASLB	Arithmetic shift left B	
	ASL	Arithmetic shift left memory	
ASR	ASRA	Arithmetic shift right A	
	ASRB	Arithmetic shift right B	
	ASR	Arithmetic shift right memory	
COM	COMA	Ones complement A (1→0; 0→1)	$\overline{A}$→A
	COMB	Ones complement B	$\overline{B}$→B
	COM	Ones complement memory	$\overline{M}$→M
EOR	EORA	Exclusive OR A (⊻)	A ⊻ M→A
	EORB	Exclusive OR B	B ⊻ M→B
LSL	LSLA	Logic shift left A	
	LSLB	Logic shift left B	
	LSL	Logic shift left memory	
LSR	LSRA	Logic shift right A	
	LSRB	Logic shift right B	
	LSR	Logic shift right memory	
OR	ORA	OR memory with A (∨)	A ∨ M→A
	ORB	OR memory with B	B ∨ M→B
	ORCC	OR condition code register	CC ∨ IMM→CC
ROL	ROLA	Rotate left A	
	ROLB	Rotate left B	
	ROL	Rotate left memory	
ROR	RORA	Rotate right A	
	RORB	Rotate right B	
	ROR	Rotate right memory	

sion is needed on these instructions. The arithmetic shift left (ASL) and logic shift left (LSL) use the same operation symbol (Table 4-3). This is because they perform exactly the same operation and, as you will see shortly, use the same op codes. Therefore there is *no* functional difference between ASL and LSL. Motorola has simply provided some duplication here since some users think of this operation as an arithmetic shift left, while others think of it as a logic shift left operation.

Two surprises are provided in this instruction listing. They are the ANDCC and ORCC instructions. These two instructions do not appear in the 6800 instruction set. However, they are meant to replace several 6800 instructions. With the 6800 there are eight instructions that allow you to operate on the condition code register. Two of the eight (TAP and TPA) are used to transfer data between accumulator A and the CCR. These have been replaced with the two 6809 exchange (EXG) and transfer (TFR) instructions. The remaining six 6800 condition code register instructions are used to set and clear the C, I, and V flags. They are: SEC, CLC, SEI, CLI, SEV, and CLV. These *six* instructions have been replaced in the 6809 with *two* instructions: ANDCC and ORCC. Both instructions use immediate addressing to AND or OR a data byte immediately with the contents of the condition code register for the purposes of setting or clearing *any CCR* flag. To *set* any CCR flag, *you simply* OR a *logic 1* immediately with that flag's bit position. To *clear* a flag, you AND *a logic 0* with the respective flag bit position. Note, if you OR a logic 0 or AND with a logic 1, the flag status *will not* change. This can be used to preserve the state of various bits, while changing others. Some examples may help at this point.

Example 4-3: Clearing Condition Code Register Flags

Suppose that you wish to clear the C and I flags in the condition code register. Determine the required instruction and immediate data byte to be used.

To *clear* a flag you must AND *a logic 0* with that flag's bit position. Therefore, since the C and I flags are in bit positions 0 and 4, respectively, the data byte must contain a logic 0 in these bit positions. All other bit positions will contain a logic 1 so that the other flags are not affected by the ANDing operation. Therefore the correct data byte would be 1110 1110_2 or EE_{16}. The complete instruction would then be:

```
ANDCC #
EE
```

Example 4-4: Setting Condition Code Register Flags

Suppose that you wish to set the N and F flags. Determine the required instruction and immediate data byte to be used.

To *set* a flag you must OR *a logic 1* with that flag's bit position. The N and F flags are in bit positions 3 and 6, respectively. Therefore the data byte must contain a logic 1 in these bit positions. All other bit positions will contain a logic 0 so that the other flags are not affected by the ORing operation. Thus the correct data byte would be 0100 1000_2 or 48_{16}. The complete instruction would then be:

ORCC #
48

Example 4-5: Instruction Interpretation

Interpret the following:

1. ANDCC #
   00
2. ORCC #
   10
3. ANDCC #
   FF

1. The first instruction *clears* the condition code register, since a logic 0 is ANDed with all of the CCR bits.
2. The second instruction *sets* the I flag, since a logic 1 is being ORed with the I-flag bit position of the CCR. All other bits are unchanged.
3. The third instruction will *not* accomplish anything, except to waste MPU time, since logic 1s are being ANDed with all the CCR flags.

The logic instruction op codes are provided in Table 4-4.

TEST INSTRUCTIONS

The 6809 test instructions can be used to test data in the following ways:

- Arithmetic compare memory with A, B, D, X, Y, U, and S.
- Logical compare memory with A and B.
- Test for zero, positive or minus on memory, A, and B.

There are no real big surprises in the 6809 test instructions over those of the 6800. The 6809 test instructions are listed in Table 4-5. The three main categories of testing are: logic bit test (BIT), arithmetic compare test (CMP), and byte test (TST) for zero, positive, or negative.

The bit test allows you to test for single bit status (logic 1 or 0) in accumulator A or B using a data mask byte supplied (immedi-

Table 4-4. Logic Instruction Op Codes

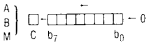
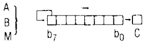
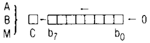
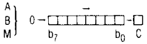
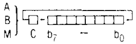
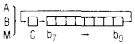
Instruction/ Forms		6809 Addressing Mode															Description	5	3	2	1	0
		Inherent			Direct			Extended			Immediate			Indexed								
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	N	Z	V	C
AND	ANDA				94	4	2	B4	5	3	84	2	2	A4	4+	2+	A ∧ M→A	•	↕	↕	0	•
	ANDB				D4	4	2	F4	5	3	C4	2	2	E4	4+	2+	B ∧ M→B	•	↕	↕	0	•
	ANDCC										1C	3	2				CC ∧ IMM→CC					1
ASL	ASLA	48	2	1														8	↕	↕	↕	↕
	ASLB	58	2	1														8	↕	↕	↕	↕
	ASL				08	6	2	78	7	3				68	6+	2+		8	↕	↕	↕	↕
ASR	ASRA	47	2	1														8	↕	↕	•	↕
	ASRB	57	2	1														8	↕	↕	•	↕
	ASR				07	6	2	77	7	3				67	6+	2+		8	↕	↕	•	↕
COM	COMA	43	2	1													A→A	•	↕	↕	0	1
	COMB	53	2	1													B→B	•	↕	↕	0	1
	COM				03	6	2	73	7	3				63	6+	2+	M→M	•	↕	↕	0	1
EOR	EORA				98	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	•
LSL	LSLA	48	2	1														•	↕	↕	↕	↕
	LSLB	58	2	1														•	↕	↕	↕	↕
	LSL				08	6	2	78	7	3				68	6+	2+		•	↕	↕	↕	↕
LSR	LSRA	44	2	1														•	0	↕	•	↕
	LSRB	54	2	1														•	0	↕	•	↕
	LSR				04	6	2	74	7	3				64	6+	2+		•	0	↕	•	↕
OR	ORA				9A	4	2	BA	5	3	8A	2	2	AA	4+	2+	A ∨ M→A	•	↕	↕	0	•
	ORB				DA	4	2	FA	5	3	CA	2	2	EA	4+	2+	B ∨ M→B	•	↕	↕	0	•
	ORCC										1A	3	2				CC ∨ IMM→CC					7
ROL	ROLA	49	2	1														•	↕	↕	↕	↕
	ROLB	59	2	1														•	↕	↕	↕	↕
	ROL				09	6	2	79	7	3				69	6+	2+		•	↕	↕	↕	↕
ROR	RORA	46	2	1														•	↕	↕	•	↕
	RORB	56	2	1														•	↕	↕	•	↕
	ROR				06	6	2	76	7	3				66	6+	2+		•	↕	↕	•	↕

Table 4-5. Test Instructions

Mnemonic		Operation	Operation Symbol
BIT	BITA	Bit test A	$A \wedge M$
	BITB	Bit test B	$B \wedge M$
CMP	CMPA	Compare with A	$A - M$
	CMPB	Compare with B	$B - M$
	CMPD	Compare with D	$D - M{:}M + 1$
	CMPS	Compare with S	$S - M{:}M + 1$
	CMPU	Compare with U	$U - M{:}M + 1$
	CMPX	Compare with X	$X - M{:}M + 1$
	CMPY	Compare with Y	$Y - M{:}M + 1$
TST	TSTA	Test A	$A - 0$
	TSTB	Test B	$B - 0$
	TST	Test Memory	$M - 0$

ately) as part of the instruction statement or obtained from memory via direct, extended, or indexed addressing (any of the indexed addressing modes may be used). Note from the operation symbol in Table 4-5 that bit test is an ANDing operation which only affects the N and Z flags of the CCR. As with all the test instructions, no "result" is generated, besides setting or clearing the appropriate flag, since the main function of any test instruction is to simply *test* data, affecting only the CCR for conditional branch purposes.

The compare instructions will allow you to compare the contents of accumulators A, B, D, and any of the indexible registers (X, Y, S, U) with data supplied as part of the instruction (immediate) or obtained from memory using direct, extended, or any of the indexed modes of addressing. Compare operations are *subtraction operations* which are used to determine whether two quantities are equal, or whether one *is* larger (smaller) than the other. These comparison operations are normally used just prior to conditional branch instructions such as branch if equal (BEQ), branch if not equal (BNE), branch if higher (BHI), branch if lower (BLO), etc. When comparing data to one of the accumulators or indexible registers, the data are subtracted from the respective accumulator or register with the N, Z, V, and C flags being set or cleared accordingly. Neither the accumulator/register contents or memory location contents are affected by this operation. With the *6800* you can compare accumulator B to accumulator A with an inherent (1-byte) instruction. There is no such instruction for the 6809; however, 16-bit data may be compared to any of the 16-bit registers (except the PC) in the 6809—an acceptable tradeoff.

The byte test (TST) instructions are used to determine if the contents of accumulator A, B, or any memory location are positive, negative, or zero, without affecting the contents of the respective register. These instructions will normally be used just prior to conditional branch instructions such as branch if plus (BPL), branch if minus (BMI), branch if equal zero (BEQ), and branch if not equal zero (BNE). Note from the operation symbol that zero is subtracted from the respective register. *Only* the N and Z flags are affected and *no* result is generated with the byte test operation.

The following examples illustrate the effect of the test instructions on the condition code register (CCR). You will observe their use in conjunction with the branch instructions in the next chapter. You may wish to execute these examples on a 6809-based computer and then check the condition code register for the correct flag status. The instruction op codes are supplied in Table 4-6.

Example 4-6: Using the Bit Test (BIT) Instruction

Suppose that you wish to check the logic status of bit 6 in accumulator A. Which test instruction is required and what would be the correct data mask byte?

To test for bit status in accumulator A you would use the BITA instruction. The mask byte must be 01000000_2 or 40_{16} to check for bit 6 status. Note that all other bit positions in the mask byte are "masked out" with zeros. Therefore the proper instruction listing would be:

```
BITA #
 40
```

If bit 6 in accumulator A is set (logic 1) when the above instruction is executed, the ANDing operation would cause the Z flag of the CCR to clear since the "result" of the ANDing operation *is not* 0. If the Z flag is set as a result of the above operation, this would indicate that bit 6 of accumulator A is cleared (logic 0), since the ANDing operation result *is* zero. Try this by loading various values into accumulator A, executing the above instruction, and then examining the Z-flag status.

Example 4-7: Using the Compare (CMP) Instruction

Write an instruction sequence that will compare the contents of the X register with data whose *address* is found in two consecutive memory locations which begin *50* decimal *positions* after the compare instruction in the program.

The only instruction required to perform the above task is the CMPX instruction. Indirect addressing is required since the *address* of the operand is being specified rather than the operand itself. In addition, since the operand's address is to be found *relative* to the position of the CMPX instruction, program counter relative addressing must be used. Therefore the CMPX instruction will use *indirect*

Table 4-6. Test Instruction Op Codes

Instruction/ Forms		6809 Addressing Mode																5	3	2	1	0
		Inherent			Direct			Extended			Immediate			Indexed								
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	Description	H	N	Z	V	C
BIT	BITA				95	4	2	B5	5	3	85	2	2	A5	4+	2+	Bit test A (M A)	•	↕	↕	0	•
	BITB				D5	4	2	F5	5	3	C5	2	2	E5	4+	2+	Bit test B (M B)	•	↕	↕	0	•
CMP	CMPA				91	4	2	B1	5	3	81	2	2	A1	4+	2+	Compare M from A	8	↕	↕	↕	↕
	CMPB				D1	4	2	F1	5	3	C1	2	2	E1	4+	2+	Compare M from B	8	↕	↕	↕	↕
	CMPD				10 93	7	3	10 B3	8	4	10 83	5	4	10 A3	7+	3+	Compare M:M + 1 from D	•	↕	↕	↕	↕
	CMPS				11 9C	7	3	11 BC	8	4	11 8C	5	4	11 AC	7+	3+	Compare M:M + 1 from S	•	↕	↕	↕	↕
	CMPU				11 93	7	3	11 B3	8	4	11 83	5	4	11 A3	7+	3+	Compare M:M + 1 from U	•	↕	↕	↕	↕
	CMPX				9C	6	2	BC	7	3	8C	4	3	AC	6+	2+	Compare M:M + 1 from X	•	↕	↕	↕	↕
	CMPY				10 9C	7	3	10 BC	8	4	10 8C	5	4	10 AC	7+	3+	Compare M:M + 1 from Y	•	↕	↕	↕	↕
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	•

program counter relative addressing. Now, the instruction will be 3 bytes: the CMPX op code followed by the post byte, which in turn is followed by the program counter relative offset. Since the CMPX instruction requires 3 bytes, the memory locations which contain the operand address will be located 47_{10} (not 50_{10}) positions *from the program counter*. This is because the program counter always points to the *next* instruction to be executed. Therefore the correct PC relative offset would be 47_{10} or $2F_{16}$. Thus the assembly language code for this operation is CMPX [2F, PCR].

Determine the correct op-code listing for this instruction.

From Tables 2-6 and 4-6 the corresponding op-code listing would be:

CMPX instruction op code:	AC
post byte:	9C
relative offset:	2F

Example 4-8: Using the Compare Instruction

Suppose accumulator D contains 0F50 at the time the following instruction is executed.

```
CMPD #
0F
49
```

Determine the CCR flag status after execution of this instruction.

The CMPD instruction will subtract 0F49 from 0F50. Since the accumulator's contents are larger than the "data" used in the compare operation, the N flag will be cleared, indicating a positive result. The Z flag will also be cleared, indicating a *nonzero* result. The C flag will also clear since no carry or borrow is generated. The remaining flags are not affected by this operation and will remain set, or cleared, from some previous operation.

Example 4-9: Using the Test (TST) Instruction

Suppose accumulator A contains FF when TSTA is encountered. Determine the CCR flag status after TSTA is executed.

Since the contents of accumulator A (FF) are nonzero and negative (twos complement), the Z flag will be cleared and the N flag set. The V flag is *always* cleared during this operation (see Table 4-6). The other flags are not affected by the operation and will remain set, or cleared, as the result of some previous operation.

Table 4-7 shows you how to perform some additional arithmetic, logic, and test operations with previously discussed 6809 instructions.

Table 4-7. Performing Additional Arithmetic, Logic, and Test Operations With 6809 Instructions

Arithmetic Operations	**Instructions**	
Add A to B	PSHS	A
(A + B→B)	ADDB	,S^{+}
Add X to D	PSHS	X
(X + D→D)	ADDD	,S^{++}
Add D to X	LEAX	D,X
(D + X→X)		
Add Y to X	EXG	D,Y
(Y + X→X)	LEAX	D,X
	EXG	D,Y
Decrement D	EXG	D,X
(D − 1→D)	LEAX	−1,X
	EXG	D,X
Increment D	EXG	D,X
(D + 1→D)	LEAX	1,X
	EXG	D,X
Negate D	COMA	
($\overline{D}$ + 1→D)	COMB	
	ADDD	#
	01	
Negate X	EXG	D,X
($\overline{X}$ + 1→X)	COMA	
	COMB	
	ADDD	#
	01	
	EXG	D,X
Subtract X from D	PSHS	X
(D − X→D)	SUBD	,S^{++}
Subtract D from X	PSHS	D
(X − D→X)	COMA	
	COMB	
	ADDD	#
	01	
	LEAX	D,X
	PULS	D
Logic Operations	**Instructions**	
AND B to A	PSHS	B
(B ∧ A→A)	ANDA	,S^{+}
AND A to B	PSHS	A
(A ∧ B→B)	ANDB	,S^{+}
Arithmetic shift left D	ASLB	
	ROLA	
Logic shift right D	LSRA	
	RORB	
Test Operations	**Instructions**	
Bit test B to A	PSHS	B
(B ∧ A)	BITA	,S^{+}
Compare B to A	PSHS	B
(A − B)	CMPA	,S^{+}
Compare A to B	PSHS	A
(B − A)	CMPB	,S^{+}
Compare Y to X	PSHS	Y
(X − Y)	CMPX	,S^{++}

REVIEW QUESTIONS

1. What is the difference between ABX and LEAX B, X?

2. Why is the MUL operation an *unsigned* multiply operation?

3. Which 6809 arithmetic instruction will allow you to convert an 8-bit signed number in accumulator B to a 16-bit signed number in accumulator D?

4. Write an instruction sequence which will compute the U stack pointer with the product of accumulators A and B. Then, stack *all* the internal register contents on the U stack defined by this pointer.

5. Write the correct op-code listing for the instruction sequence in Question 4.

6. How many MPU cycles would be required to execute the program in Questions 4 and 5?

7. What is the difference between arithmetic shift left (ASL) and logic shift left (LSL)?

8. What are the functions of the ANDCC and ORCC instructions?

9. How would you clear a CCR flag bit?

10. How would you set a CCR flag bit?

11. Interpret the following:

a. ORCC # FF	b. ANDCC # AF	c. ORCC # 00

12. What are the three main categories of test instructions?

13. What is the purpose of the test instructions?

14. The BIT instruction performs a __________ operation and only affects the __________ and __________ CCR flags.

15. The CMP instruction performs a __________ operation.

16. Which of the 6809 registers can be compared?

17. Which CCR flags are affected by the CMP instruction?

18. What can be tested with the TST instruction and what does the test indicate?

19. Which CCR flags are affected by the TST instruction?

20. Determine the CCR flag status after execution of the following instructions (assume accumulator A contains 5F):

```
a. BITA #          b. CMPA          c. TSTA
     20                 5F
```

ANSWERS

1. ABX is an inherent instruction which will add the *unsigned* contents of accumulator B to the X register. LEAX B, X is a 2-byte instruction which will add the *signed* contents of accumulator B to the X register.

2. To facilitate multiprecision (multibyte) multiplication operations.

3. Signed extend (SEX)

4. MUL
   TFR D, U
   PSHU CC, A, B, DP, X, Y, S, PC

5. From Tables 3-2 and 4-2 and Figs. 2-7 and 3-6 the correct op-code listing would be:

```
MUL          3D
TFR D, U     1F
             03
PSHU         36
             FF
```

6. From Tables 3-2 and 4-2 the above program would require 35 MPU cycles to execute. The MUL instruction requires 11, the TFR requires 7, and the PSHU requires 17. Note that 5 MPU cycles *plus 1 cycle* for each *byte* pushed are required for the PSHU instruction.

7. There is no functional difference between these two instructions.

8. The ANDCC and ORCC instructions are used to set and clear the CCR flags.

9. To clear a CCR flag bit you would AND a logic 0 with the respective flag bit position using the ANDCC instruction.

10. To set a CCR flag bit you would OR a logic 1 with the respective flag bit position using the ORCC instruction.

11. a. This instruction will set all the CCR flags since a logic 1 is being ORed with all CCR bit positions.
    b. This instruction will clear the F and I flags since logic 0s are being ANDed with CCR bit positions 4 and 6.
    c. This instruction will not accomplish anything since logic 0s are being ORed with all the CCR flags.

12. Logic bit test (BIT), arithmetic compare test (CMP), and byte test (TST) for zero, positive, or negative

13. To test data, affecting only the condition code register for conditional branch purposes. No result is generated.

14. ANDing, N, Z

15. Subtract

16. Accumulators A, B, and D and any of the indexible registers (X, Y, S, U)

17. The N, Z, V, and C flags

18. The TST instruction can be used to test the contents of accumulators A and B or any memory location for a positive, negative, or zero quantity.

19. Only the N and Z flags

20. a. The N flag will be cleared. The Z flag will be set indicating a zero result from the ANDing operation. All other flags are not affected and might be set or cleared as the result of some previous operation.
    b. The Z flag will be set, indicating a zero result from the subtract operation. The C flag will also set since a last carry (borrow) is generated as a result of the twos complement operation. The N and V flags will be cleared, and all other flags are not affected by the operation and might be set or cleared as the result of some previous operation.
    c. The N flag is cleared, indicating a positive value in accumulator A. The Z flag is also cleared, indicating a nonzero value in accumulator A. The V flag will always clear with the TST operation. All other flags are not affected by the operation and might be set or cleared as the result of some previous operation.

CHAPTER 5

Branch and Miscellaneous Instructions

INTRODUCTION

The last two instruction categories that we need to discuss are the branch and miscellaneous instructions.

The instructions that give the 6809 decision-making capability or *intelligence* are *branch* instructions. These instructions are normally used *after* arithmetic, logic, or test instructions which set or clear the condition code register (CCR) flags according to the result of the operation. The branch instruction will make a decision based on the results, as indicated by the CCR flags status. When a branch is initiated, a new address is loaded into the program counter and this causes the program to alter its flow so that program execution continues starting at the new address specified by the branch operation. The 6809 instruction set contains 18 branch instructions. Each of these instructions is available in a *short-* or *long-branch* version which utilizes *relative* addressing, as discussed in Chapter 2.

The last instruction category, miscellaneous, is made up of instructions that do not fit conveniently in any of the previously discussed categories. Such instructions as jump (JMP), jump to subroutine (JSR), software interrupt (SWI), return from interrupt (RTI), etc., are included in this category. Many of these instructions will be familiar to 6800 users; however, there are a few surprises. The 6809 includes three separate software interrupts such that there will always be one available to the end user. Many 6800 firmware packages used the *one* software interrupt available and left the end user without this capability. The 6809 CWAI instruction replaces the 6800

WAI instruction. The CWAI instruction is similar to WAI; however, it is a 2-byte instruction which allows you to clear any of the CCR flags. Finally, the 6809 instruction set includes an instruction that will allow you to synchronize the system software to an external hardware process. This instruction is called SYNC and should prove valuable in large system applications where rapid data transfers are common.

OBJECTIVES

At the end of this chapter you will be able to do the following:

- Describe the three categories of 6809 conditional branch instructions.
- List each 6809 branch instruction along with its complement (opposite) branch instruction.
- Understand how branch instructions are used in a program to make decisions based on the results of arithmetic, logic, or test operations.
- Explain what happens when CWAI is executed and how this differs from the 6800 WAI instruction.
- Understand the difference between the three 6809 software interrupts and explain what happens when any of the software interrupts are executed.
- Understand what is meant by the term "absolute indirect addressing."
- Explain how to synchronize the main program to an external hardware event by using the SYNC instruction.
- Understand what is meant by the "syncing state."
- Explain how the SYNC instruction differs from the CWAI instruction.
- Understand how to implement several 6800 instructions which are not part of the 6809 instruction set.

BRANCH INSTRUCTIONS

The 6809 branch instructions are listed in Table 5-1. All of the 6809 branch instructions use relative addressing (refer to Chapter 2). Branch instructions can be divided into two general categories: unconditional branches and conditional branches.

The *unconditional* branch is one that causes the program to branch (or not branch) regardless of any conditions. In the 6809 these branches are the branch always (BRA), branch to subroutine (BSR), and branch never (BRN) instructions. The BRA and BSR instructions will cause the 6809 to branch to the destination address

Table 5-1. 6809 Branch Instructions

Mnemonic		Operation	Branch Test
BCS	BCS LBCS	Branch if the carry (C) flag is set	$C = 1$
BEQ	BEQ LBEQ	Branch if the register contents are equal to the memory contents (Z flag set)	$Z = 1$
BGE	BGE LBGE	Branch if the *signed* register contents are *greater* than or *equal* to the *signed* memory contents	$N \veebar V = 0$
BGT	BGT LBGT	Branch if the *signed* register contents are *greater* than the *signed* memory contents	$Z \vee (N \veebar V) = 0$
BHI	BHI LBHI	Branch if the *unsigned* register contents are *higher* than the *unsigned* memory contents	$C \vee Z = 0$
BHS	BHS LBHS	Branch if the *unsigned* register contents are higher than or the *same* as the *unsigned* memory contents	$C = 0$
BLE	BLE LBLE	Branch if the *signed* register contents are *less* than or *equal* to the *signed* memory contents	$Z \vee (N \veebar V) = 1$
BLO	BLO LBLO	Branch if the *unsigned* register contents are lower than the *unsigned* memory contents	$C = 1$
BLS	BLS LBLS	Branch if the *unsigned* register contents are *lower* than or the *same* as the *unsigned* memory contents	$C \vee Z = 1$
BLT	BLT LBLT	Branch if the *signed* register contents are *less* than the signed memory contents	$N \veebar V = 1$
BMI	BMI LBMI	Branch if minus (N flag set)	$N = 1$
BNE	BNE LBNE	Branch if the register contents are *not* equal to the memory contents (Z flag cleared)	$Z = 0$
BPL	BPL LBPL	Branch if plus (N flag cleared)	$N = 0$
BRA	BRA LBRA	Branch always (unconditional)	None
BRN	BRN LBRN	Branch never	None
BSR	BSR LBSR	Branch to subroutine	None
BVC	BVC LVBC	Branch if the overflow (V) flag is cleared	$V = 0$
BVS	BVS LBVS	Branch if the overflow (V) flag is set	$V = 1$

regardless of any conditions. However, the BSR instruction is used to call subroutines with the program counter's contents being saved on the hardware (S) stack such that proper return to the main program is accomplished. This instruction is very similar to the jump to subroutine (JSR) instruction to be discussed shortly. The BRN instruction is new and was not available in the 6800. This instruction is actually a 2- or 4-byte no-operation (NOP) instruction. The 6809 will cycle through the instruction byte(s) and relative address offset without altering its execution. On the surface you might think this is a meaningless instruction and wonder why it was included in the 6809 instruction set. You can, however, *bury* or *hide* an instruction op code as the relative address offset. For example, when the 6809 first cycles through the BRN instruction, it will read the buried (hidden) op code as a relative address offset and no operation will result. Then, later in the program, you can jump back to the buried (hidden) op code for its execution. This saves programming steps and is a useful trick to remember.

The remaining 6809 branch instructions are *conditional* branches. The conditional branch is one which is dependent upon some condition as indicated by the condition code register. If the condition is met, the branch will occur. If not, the program will continue to the next sequential instruction without branching. Conditional branch instructions will normally be used after one of the test or arithmetic/logic instructions discussed in Chapter 4. However, these instructions can be used after any instruction that operates on a register, setting or clearing the condition code register flags according to the result of the operation. The branch will occur (or not occur) based on the condition code register flag(s) status at the time the conditional branch instruction is encountered. Several of the conditional branch instructions, such as the BEQ, BNE, BCS, BCC, BVS, and BVC, involve a *simple* check of *one* of the condition code register's status flags, and are obvious. For example, BEQ/BNE is a direct check of the Z-flag status. We will refer to these as *simple* branches. The remaining conditional branches can be subdivided into two categories, *signed* conditional branches and *unsigned* conditional branches. The signed branches are used when operating on twos complement data. Unsigned branches are used for operating on non-twos-complement data. The unsigned branches are not, in general, useful after instructions such as LD, ST, INC, DEC, TST, CLR, or COM.

Note from Table 5-1 that both the signed and unsigned branches cause the program to alter its execution as the result of some logical combination of the condition code register flags.

Finally, Table 5-2 lists the three categories of conditional branch instructions just discussed, that is, simple, signed and unsigned. In

Table 5-2. 6809 Conditional Branch Instructions

Simple Conditional Branches	
Condition	*Complement Condition*
BEQ	BNE
BMI	BPL
BCS	BCC
BVS	BVC
Signed Conditional Branches	
Condition	*Complement Condition*
BGT	BLE
BGE	BLT
BEQ	BNE
Unsigned Conditional Branches	
Condition	*Complement Condition*
BHI	BLS
BHS	BLO
BEQ	BNE

Table 5-2 each branch instruction has its complement (opposite) branch condition listed next to it. For example, BEQ provides the opposite test than that of BNE, BLT provides the opposite test than that of BGE, and so on. In addition, note that BEQ/BNE falls into all three categories of conditional branches. The branch instruction op codes are given in Table 5-3.

Now, let's look at some examples involving branch instructions. Verify their proper execution on your 6809-based system or trainer.

Example 5-1: Using the BEQ Instruction

Suppose the 6809 encounters the following instruction sequence in your program:

```
  LDY #
  FC
  50
┌→LEAY ,–Y
│ CMPY
│ 00
│ 00
│ BNE
└─F8
  .
  .
  .
```

What will happen?

This instruction sequence first loads the Y register immediately with FC50. The Y register is then decremented by the LEAY instruc-

Table 5-3. Branch Instruction Op Codes

Instruction/		Relative				5	3	2	1	0
Forms		OP	$\sim^5$	#	Description	H	N	Z	V	C
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 = 1	•	•	•	•	•
	LBCS	10 25	5(6)	4	Long branch C = 1	•	•	•	•	•
BEQ	BEQ	27	3	2	Branch Z = 1	•	•	•	•	•
	LBEQ	10 27	5(6)	4	Long branch Z = 1	•	•	•	•	•
BGE	BGE	2C	3	2	Branch ≥zero	•	•	•	•	•
	LBGE	10 2C	5(6)	4	Long branch ≥ zero	•	•	•	•	•
BGT	BGT	2E	3	2	Branch >zero	•	•	•	•	•
	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	•	•	•	•	•
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 always	•	•	•	•	•
BRN	BRN	21	3	2	Branch never	•	•	•	•	•
	LBRN	10 21	5	4	Long branch 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 28	5(6)	4	Long branch V = 0	•	•	•	•	•
BVS	BVS	29	3	2	Branch V = 1	•	•	•	•	•
	LBVS	10 29	5(6)	4	Long branch V = 1	•	•	•	•	•

tion. After decrementing, the Y register is compared immediately to 0000. Recall that the compare operation subtracts its operand from the specified register (Y in this case) and sets or clears the condition code register flags accordingly. In this example you are concerned with the Z flag and it will be set *only* when the Y register contents are 0000. The BNE instruction will cause the program to branch back to the LEAY instruction until the Z flag is set. Therefore, the program will loop until the Y register is decremented down to zero. This type of routine can be used to create time delays within your program. You might want to verify the correct relative address offset (F8) for the BNE instruction by counting the number of bytes required by each instruction in the loop.

Example 5-2: Using the BLT Instruction

Suppose that the 6809 encounters the following instruction sequence:

```
      .
      .
  ┌→CMPA, X+
  │ BLT
  └─FC
    LDB,—X
      .
      .
```

What will happen?

In this example the contents of the memory location *specified* by the X register are compared to the contents of accumulator A. As long as the *signed* accumulator A contents are less than the *signed* memory contents, the BLT instruction will cause the program to branch *back* to the CMPA instruction. Note that the CMPA instruction uses auto-increment by 1, on the contents of the X register. Therefore, each time the loop is executed, the X register contents are incremented. Thus consecutive memory locations are "searched" until a value is found that is *less than or equal to* the existing accumulator A contents. Once the value is found, it is loaded into accumulator B for further processing. To load the proper value the LDB instruction must use a pre-decrement on the X register since the CMPA instruction had previously post-incremented the X register.

Example 5-3: Using the BLO Instruction

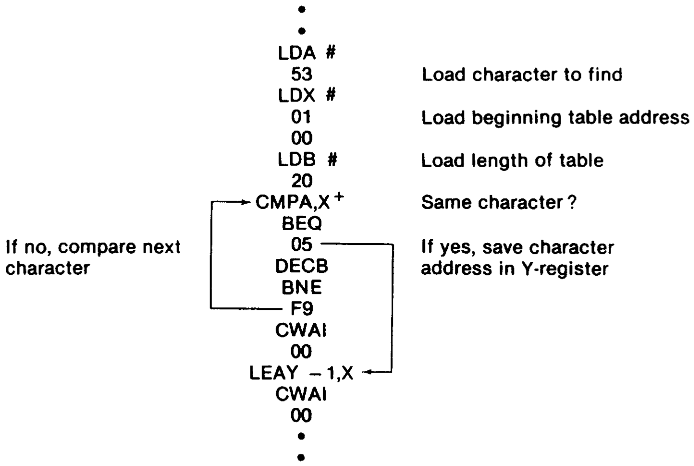
What would happen if the BLT instruction were replaced with BLO in Example 5-2?

The only difference is that the 6809 would *not* look at the accumulator A and memory contents as twos complement signed numbers. The branch would occur as long as the *unsigned* accumulator A contents are less than the *unsigned* memory contents. Thus the program would be *searching* consecutive memory locations for a value which is *less than* or *equal to* the existing *unsigned* contents of accumulator A.

Example 5-4: Character Search

Suppose you wish to search a memory table of 32_{10} (20_{16}) characters for a specific character. Once that character is found, you will store its address in the Y register. This character you wish to locate is an "S," which has an ASCII representation of 53_{16}. The character table begins at memory location 0100.

The instruction sequence in Fig. 5-1 will accomplish the character search. Note that no *instruction* addresses need to be specified. Thus the program is totally position independent. The X register is being used as the pointer register and is incremented using the auto-increment CMPA instruction to point to each successive table value until a match is made. The LEAY instruction will then save the character's address in the Y register. This instruction decrements the X register before the address value is transferred to the Y register. This is done since, at this point in the program, the X pointer is one ahead of the actual character address.

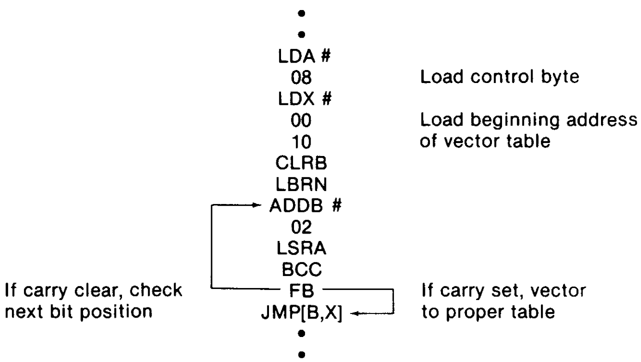


```
                         •
                         •
                       LDA #
                        53           Load character to find
                       LDX #
                        01           Load beginning table address
                        00
                       LDB #         Load length of table
                        20
                       CMPA,X+       Same character?
                        BEQ
If no, compare next     05           If yes, save character
character              DECB          address in Y-register
                        BNE
                        F9
                       CWAI
                        00
                     LEAY -1,X
                       CWAI
                        00
                         •
                         •
```

Fig. 5-1. Instruction sequence for character search.

Example 5-5: Computed Go To

Suppose you wish to *vector* to a table based on a vector control byte. The control byte has only 1 bit set (logic 1). The logic 1 bit position in the control byte determines which of eight table vectors is to be used to transfer program execution to the table. Assume, for example



```
                                 •
                                 •
                               LDA #
                                08              Load control byte
                               LDX #
                                00              Load beginning address
                                10              of vector table
                               CLRB
                               LBRN
                         ┌──→ ADDB #
                         │      02
                         │     LSRA
                         │     BCC
If carry clear, check    └──── FB ──────┐     If carry set, vector
next bit position            JMP[B,X] ←─┘     to proper table
                                 •
                                 •
```

Fig. 5-2. Instruction sequence to access vector.

purposes, that bit 3 of the control byte is set and the eight vectors are located in sixteen (2 bytes per vector) consecutive memory locations beginning at address 0100. (Therefore the vector we wish to access is located at address 0106.) Write an instruction sequence which will access the proper vector, based on the control byte, and transfer the program execution to the corresponding table.

The instruction sequence in Fig. 5-2 will accomplish the given task. The control byte is first loaded into accumulator A. Then the index register is loaded with 0100, which is the beginning address of the vector table. Accumulator B will be used as a constant offset to locate the proper vector. Each time a control byte bit is tested via the LSRA and BCC instructions, accumulator B will be incremented by 2. Therefore, since bit 3 of the control byte is set, accumulator B will contain 06_{16} when the branch loop is broken. Thus the vector at address 0100 + 06, or 0106, will be accessed by the JMP instruction. By using *indirect* addressing with the JMP instruction, program control will be transferred to the table which is "pointed to" by the vector located at address 0106. Note the significance of the long branch never instruction (LBRN). The ADDB instruction is not executed the first time through the routine, since it forms the relative address offset of the LBRN instruction. However, if the C flag is clear after the LSRA instruction, the routine will branch back to execute the *buried* (hidden) op code (ADDB). This will allow the *beginning* address of the vector table to be initially loaded into the X register, *rather than* an address which is *two less* than the beginning vector table address. Think about it! This is an excellent example of using the BRN instruction to hide or bury an instruction op code.

MISCELLANEOUS INSTRUCTIONS

The remaining instructions in the 6809 instruction set we have categorized as miscellaneous, since they do not fit into any of the

Table 5-4. 6809 Miscellaneous Instructions

Mnemonic		Operation	Operation Symbol
CWAI		AND CC, then wait for interrupt	CC ∧ IMM→CC Wait for interrupt
JMP		Jump	EA→PC
JSR		Jump to subroutine	None
NOP		No operation	None
RTI		Return from interrupt	None
RTS		Return from subroutine	None
SWI	SWI1	Software interrupt 1	None
	SWI2	Software interrupt 2	None
	SWI3	Software interrupt 3	None
SYNC		Synchronize to interrupt	None

previously discussed instruction categories. These instructions are listed in Table 5-4. Many of the instructions listed here should be familiar to 6800 users; for example, jump (JMP), jump to subroutine (JSR), no operation (NOP), return from interrupt (RTI), and return from subroutine (RTS) all perform essentially the same operations as in the 6800. The JMP and JSR instructions, however, can now be used with direct, extended, or any of the indexed modes of addressing, including indirect addressing. Consult one of the previously referenced 6800 texts for a detailed discussion of the above instructions. The miscellaneous instructions for the 6809 that we intend to present in detail here are new, or improved, over other 6800 family instructions. They are: wait (CWAI), software interrupts 1, 2, and 3 (SWI1, SWI2, SWI3), and synchronize with interrupt (SYNC).

The 6809 wait (CWAI) instruction is similar to the 6800 wait (WAI) instruction in that it puts the processor in a *wait for interrupt* state. However, the 6809 CWAI is a *2-byte* instruction: the instruction op code followed by a data byte. When the CWAI instruction is executed, its data byte is ANDed with the existing contents of the condition code register. The result of the ANDing operation is placed in the condition code register. This operation will allow you to alter the contents of the condition code register prior to its stacking. In addition, if the CWAI data byte is 00_{16}, you will clear the condition code register. Clearing the condition code register may be desirable if it is known that its contents will not be needed for any subsequent interrupt service routine. This will allow the service routine to begin with a "clean" (clear) condition code register slate. However, if the CWAI data byte is FF_{16}, the contents of the condition code register will remain unchanged.

The sequence of events associated with the CWAI instruction is

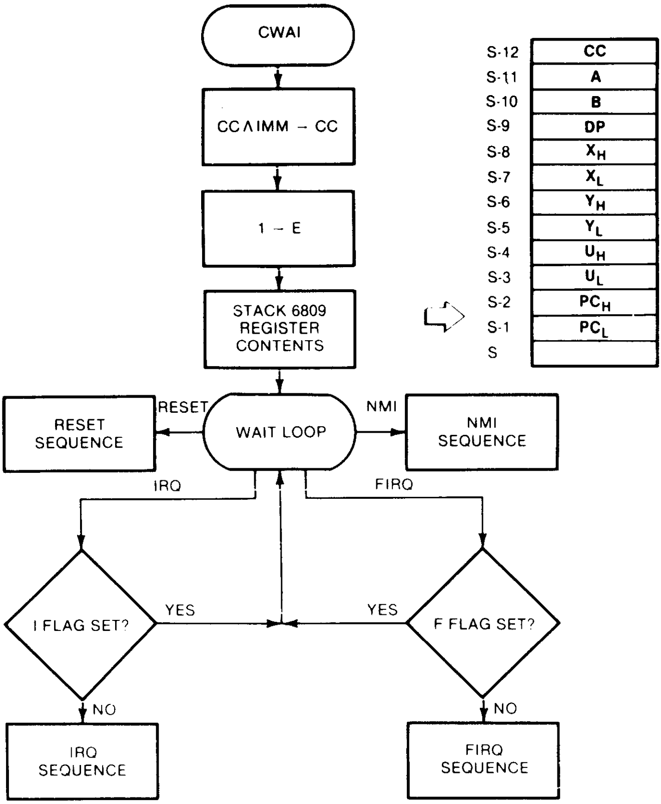


Fig. 5-3. CWAI sequence of events.

shown in Fig. 5-3. When the 6809 encounters the CWAI instruction, it ANDs the CWAI data byte with the contents of the condition code register, placing the result in the condition code register. Then the E flag of the condition code register is set. Why? The E flag is set since the next operation is to stack *all* the internal register contents in the S stack. Recall that the E flag is used for unstacking purposes to tell the 6809 that, when set (logic 1), all the registers (except S) have been stacked by a previous stacking operation. Note from Fig. 5-3 that the order of register stacking is the same as for the PSH and PUL instructions discussed in Chapter 4. This stacking order is the same for all the automatic stacking operations that the 6809 performs. After the internal registers are stacked on the S stack, the 6809 enters a *wait loop*. The wait loop may be broken only by any of the four *hardware* interrupts—$\overline{\text{RESET}}$, $\overline{\text{NMI}}$, $\overline{\text{IRQ}}$, or $\overline{\text{FIRQ}}$. Note from Fig. 5-3 that the I flag in the condition code register must be cleared to allow the $\overline{\text{IRQ}}$ interrupt to break the wait loop, and the F flag must be cleared to allow the $\overline{\text{FIRQ}}$ interrupt to break the loop. The CWAI ANDing operation will allow you to have some control over the "breaking" of the wait loop. For example, setting the I flag just prior to CWAI and using a 10_{16} CWAI data byte to preserve the I flag's status will prevent an IRQ interrupt from breaking the wait loop. The same can be done with the F flag to prevent an $\overline{\text{FIRQ}}$ from breaking the wait loop. On the other hand, you can use CWAI to assure that an $\overline{\text{IRQ}}$, $\overline{\text{FIRQ}}$, or both will be enabled. For example, a CWAI data byte of EF_{16} will enable $\overline{\text{IRQ}}$; BF_{16} will enable $\overline{\text{FIRQ}}$; and AF_{16} will enable both $\overline{\text{IRQ}}$ and $\overline{\text{FIRQ}}$. The hardware interrupts will be discussed in detail in the next chapter.

Example 5-6: Executing the CWAI Instruction

Suppose the 6809 encounters the following instruction sequence:

```
.
.
LDA #
 40
TFR A, CC
CWAI
 40
```

Determine the condition code register contents after CWAI is executed.

Since accumulator A is first loaded with 40_{16} and then transferred to the condition code register, the F flag will set and all other CCR flags will be cleared. The CWAI data byte, 40_{16}, will preserve the F-flag status during the CWAI ANDing operation. Thus an FIRQ interrupt will *not* break the wait loop. In addition, the E flag is set as a result of the CWAI operation. Therefore the condition code register contents will be $1100\ 0000_2$, or $C0_{16}$.

If $3C_{16}$ is the CWAI op code, determine the correct op-code listing for the above instruction sequence.

From the tables and figures provided in previous chapters the proper op-code listing is:

LDA instruction op code:	86
data byte:	40
TFR instruction op code:	1F
post byte:	8A
CWAI instruction op code:	3C
data byte:	40

If the S register contains $E600_{16}$ prior to executing the above sequence, at what address will the condition code register contents be found in the S stack? From Fig. 5-3 the CCR contents would be stacked at address ($E600_{16} - 12_{10}$), or $E5F4_{16}$. Verify the above results on your 6809-based system or trainer.

There are three levels of software interrupts available to the 6809. They are: SWI1, SWI2, and SWI3. Recall that a software interrupt is an *instruction* which will cause the processor to *vector* to an associated interrupt service routine. Software interrupts are normally used to insert *breakpoints* in a program for program debugging and are also useful in single-step routines, operating system calls, and software development systems. In addition, *hardware* interrupts can be simulated with software interrupts. Three software interrupt levels were provided in the 6809 since it was found that the one level which was provided in the 6800 was often used by ROM monitor packages and was therefore not available to the end user. However, it is quite unlikely that commercial software packages will use all three 6809 software interrupt levels and thus allow at least one software interrupt to be available for user systems. (Motorola promises *never* to use SWI2 and encourages other 6809 commercial software manufacturers to do the same.)

The three 6809 software interrupts have priorities in the following order: SWI1, SWI2, and SWI3. The sequence of events which is initiated with any of the three software interrupts is shown in Fig. 5-4. Note that once the present instruction is completed, the E flag of the condition code register is set, indicating that *all* of the 6809 registers (except S) are to be stacked in the S stack in the order shown in Fig. 5-4. If, after stacking, the SWI1 interrupt is being executed, the F and I flags are set to mask out any $\overline{\text{FIRQ}}$ or $\overline{\text{IRQ}}$ hardware interrupts which might occur during the SWI1 interrupt service routine. If an SWI2 or SWI3 is being executed, $\overline{\text{FIRQ}}$ and $\overline{\text{IRQ}}$ hardware interrupts will *not* be masked out unless you set the F and/or I flags during the interrupt service routine. The 6809 then acknowledges acceptance of the software interrupt to external devices by providing a logic 1 (high) level on the bus status (BS) output line. (The use of this line will be discussed shortly.) The proper interrupt vector is then loaded into the program counter, BS is brought back to a logic 0 (low) level, and the appropriate interrupt service routine is executed. Once the service routine has been completed, execution must be directed back to the main program. This is done by using a return from interrupt (RTI) instruction as the last instruction in the interrupt service routine. The RTI returns the 6809 to its previous status by automatically unstacking the old register contents from the S stack.

As mentioned previously, a software interrupt (or any interrupt for that matter) causes the 6809 to *vector* to the appropriate interrupt service routine. The *interrupt vector* is the beginning address

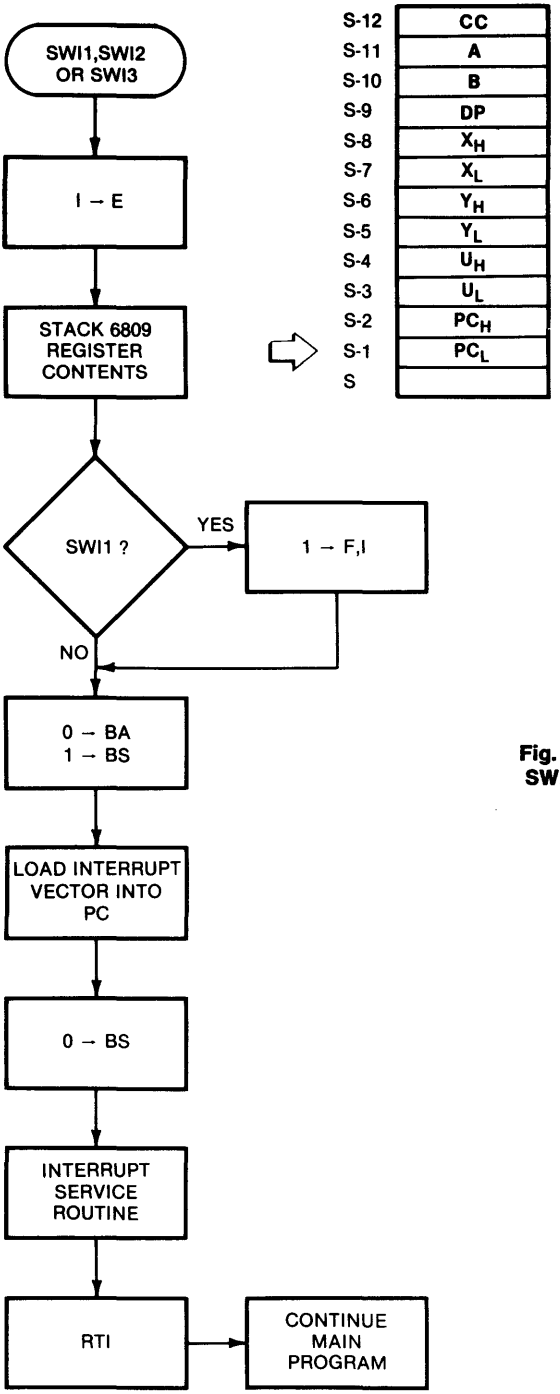


Fig. 5-4. SWI1, SWI2, and SWI3 sequence of events.

Table 5-5. Memory Map for the 6809 Interrupt Vectors

Vector Locations	Vector Assignment	Relative Priority
FFF0:FFF1	Reserved	Low
FFF2:FFF3	SWI3	
FFF4:FFF5	SWI2	
FFF6:FFF7	$\overline{\text{FIRQ}}$	↓
FFF8:FFF9	$\overline{\text{IRQ}}$	
FFFA:FFFB	SWI1	
FFFC:FFFD	$\overline{\text{NMI}}$	
FFFE:FFFF	$\overline{\text{RESET}}$	High

of the respective interrupt service routine. Each of the 6809 software and hardware interrupts has a unique vector located in the last 16 memory address locations (FFF0–FFFF).

The 6809 *vector map* is given in Table 5-5. Note that each interrupt vector is assigned a *pair* of addresses within the map. These address locations are normally in ROM and, therefore, are fixed and not alterable by the end-user. Since the 6809 "looks" to these addresses to fetch the vector, which in turn is another address, the term *absolute indirect addressing* is sometimes associated with interrupt vectoring. In addition, Table 5-5 indicates the relative priority of each interrupt, with $\overline{\text{RESET}}$ assuming the highest priority. You will also note that vector address FFF0:FFF1 has been reserved by Motorola for possibly some future application. The 6809 hardware interrupts will be covered in the next chapter.

The last instruction that we need to discuss is the synchronize to interrupt (SYNC) instruction. This instruction is used to *synchronize* the system software to external hardware events, such as rapid data transfers from i/o devices. When the SYNC instruction is encountered, all execution is halted and the 6809 goes into a wait-for-interrupt loop. This is referred to as the *syncing state*. During this syncing state the 6809 address and data lines are in a high-impedance (tri-state) state or are effectively disconnected from their external buses. Now, if a hardware interrupt occurs, two things can happen:

1. If the interrupt is not masked *and* is active for 3 MPU cycles or more, the 6809 will break the wait loop and execute the respective interrupt service routine.
2. If the interrupt is masked *or* is active for less than 3 MPU cycles, the 6809 will simply continue execution of the main program by executing the next sequential instruction.

In the first case the SYNC instruction behaves very much like the CWAI instruction, except that it does *not* cause the internal register data to be stacked. Any normal interrupt, if not masked, will probably cause the sync state to be broken in this way.

In the second case, however, the SYNC instruction can be used to synchronize the main program with an external hardware process. Recall that one of the main disadvantages to using interrupts is that they are not *synchronous* with program execution, meaning that they are not scheduled and can happen at any time within the main program. The SYNC instruction provides you with a means of using interrupts and at the same time allows you to schedule the interruption for optimum system efficiency. For example, an i/o device, such as a high-speed disk, can use a 6809 interrupt request line ($\overline{\text{IRQ}}$ or $\overline{\text{FIRQ}}$) and activate the line to indicate it is ready for data transfer. Prior to SYNC you will set the respective interrupt mask bit (I or F) and thus mask out a *normal* interrupt request. But, the active i/o device strobe will cause the next instruction to be executed instead of the normal service routine for that interrupt. The next instruction would begin the data transfer process and the main program would continue. In this way the hardware and software activities are synchronized and time is saved since no vectoring or stacking is involved. The program in Example 5-7 illustrates this process.

Example 5-7: Using the SYNC Instruction

Suppose that you wish to load 100_{10} bytes of data into memory from a disk system at a particular point in the program. The program in Fig. 5-5 will accomplish this task.

```
                                      •
                                      •
                                   ORCC #        Set I flag
                                     10
                                    SYNC         Sync state
                                   LDB #
                                     64          Load table length
                           ┌──→ LDA $($$)
                           │  (Disc Address)     Transfer disk data
Get next data byte if      │     STA,X+          into 100$_{10}$ consecutive
transfer not complete      │      DECB           memory locations specified
                           │      BNE            by the X register
                           └───── F9(8)
                                 ANDCC #         Clear I flag
                                    00
                                    •
                                    •
```

Fig. 5-5. Instruction sequence to load data into memory.

It is assumed that the disk system is using the $\overline{\text{IRQ}}$ (interrupt request) line to initiate the data transfer. The first instruction sets the I flag in the condition code register to mask out any normal $\overline{\text{IRQ}}$ interrupt. The SYNC instruction then halts the main program execution until the $\overline{\text{IRQ}}$ line is activated by the disk system. When activated, the main program will cause 100_{10} (64_{16}) bytes of data to be stored in 100_{10} (64_{16}) consecutive memory locations specified by the

Table 5-6. Miscellaneous Instruction Op Codes

Instruction/ Forms		Inherent			Direct			Extended			Immediate			Indexed			Relative			Description	5	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	N	Z	V	C
CWAI		3C	20	2																CC ∧ IMM→CC Wait for interrupt					1
JMP					0E	3	2	7E	4	3				6E	3+	2+				EA→PC	•	•	•	•	•
JSR					9D	7	2	BD	8	3				AD	7+	2+				Jump to subroutine	•	•	•	•	•
NOP		12	2	1																No operation	•	•	•	•	•
SWI	SWI	3F	19	1																Software interrupt 1	•	•	•	•	•
	SWI2	10 3F	20	2																Software interrupt 2	•	•	•	•	•
	SWI3	11 3F	20	2																Software interrupt 3	•	•	•	•	•
SYNC		13	≥2	1																Synchronize to interrupt	•	•	•	•	•
RTI		3B	6/15	1																Return from interrupt					7
RTS		39	5	1																Return from subroutine	•	•	•	•	•

X register using auto-increment indexed addressing. The main program then clears the I flag and continues. If, during the sync state, an emergency situation developed, any of the other hardware interrupts ($\overline{\text{NMI}}$, $\overline{\text{FIRQ}}$, $\overline{\text{RESET}}$) could be used to break the sync state.

The miscellaneous instruction op codes are listed in Table 5-6. A complete description of all the 6809 instructions is provided in Appendix A, and a 6809 instruction set summary is provided in the last appendix for quick reference.

Finally, as you are now probably aware, many of the 6800 instruction mnemonics are not included in the 6809 instruction set. For running 6800 software on 6809-based systems, the translations provided in Table 5-7 can be made to provide functionally equivalent 6809 operations. Some of these translations have already been discussed.

This completes the discussion of the 6809 software. Before going on, it is important that you understand the material that has been presented in the last five chapters, especially the addressing modes

Table 5-7. 6800 Equivalent Instructions

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
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	TFR A,B; TST A
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

(Chapter 2). If you still feel uncomfortable with programming the 6809, go over the material again. If at all possible, obtain a 6809-based system or trainer to get some "hands-on" programming experience. Run the examples presented here on the system and make up your own short routines. The more programming practice you get, the better you'll understand how to use the 6809.

REVIEW QUESTIONS

1. **The three 6809 *unconditional* branch instructions are ________, ________,**

   and ________.

2. **In what ways are BSR and JSR similar? Different?**

3. **Conditional branches can be divided into three categories. They are**

   __________, __________, and __________ conditional branches.

4. **BLO can be replaced with __________ since the branch test is the same for both branches.**

5. **The complement branch condition for BMI is __________; for BGE is**

   __________; and for BHI is __________.

6. **The prime advantage to relative addressing is __________.**

7. **What is the functional difference between the following two instructions:**
   BRA and JMP 3,PC
   05

8. Why is the E flag of the condition code register set during the execution of CWAI?

9. You want to clear the I flag such that an interrupt request (IRQ) will be acknowledged during CWAI. All other flags should remain unchanged. What would be the proper CWAI assembly code?

10. What *6800* code listing does the above CWAI code replace?

11. If *bit* 7 of the control byte in Example 5-5 were set, what vector address would be accessed?

12. Which software interrupt automatically sets the F and I flags to mask out FIRQ and IRQ interrupts?

13. Acceptance of an interrupt is indicated on the 6809's ____________ line.

14. The SWI2 interrupt vector is located at address ____________.

15. An addressing mode which is sometimes associated with vectoring is ____________.

16. What instruction could be replaced by the following sequence:

```
PSHS    ALL
JMP     [FFF4]
```

17. Why is the above instruction sequence not *exactly* like SWI2?

18. What instruction could be replaced by PULS PC?

19. What instruction could be used in lieu of RTI?

20. What is the main purpose of the SYNC instruction?

21. When in the syncing state, what will happen if an FIRQ is received and the F flag of the condition code register is set?

22. What would happen if the F flag is cleared in the previous question?

23. How does SYNC differ from CWAI?

24. What is the 6809 equivalent of the 6800 ABA instruction?

25. What is the 6809 equivalent of the 6800 DEX instruction?

ANSWERS

1. Branch always (BRA), branch to subroutine (BSR) and branch never (BRN).
2. They are similar since they are both used to call subroutines *and* cause the program counter contents to be saved on the S stack. In addition, they both require the use of RTS as the last subroutine instruction for proper return to the main program. They are different because BSR uses relative addressing and JSR can use direct, extended or indexed addressing.
3. Simple, signed, and unsigned
4. BCS (reference Table 5-1)

5. BPL, BLT, BLS

6. Position independence

7. There is *no* functional difference. The JMP instruction will cause the program to branch to the same location as the BRA instruction. Note that the JMP is using program counter relative addressing and therefore is also position independent.

8. The E flag of the condition code register is set during the execution of CWAI since *all* of the internal registers (except S) are stacked during this operation.

9. CWAI #
   EF

10. CLI
    WAI

11. 001E

12. SWI1

13. Bus status (BS) output line.

14. FFF4:FFF5

15. Absolute indirect addressing

16. SWI2

17. The E flag is not necessarily set for stacking. To set the E flag, the instruction ORCC #80 should be added prior to the PSHS instruction.

18. RTS

19. PULS ALL

20. To synchronize the main program with external hardware events such as data transfers

21. The syncing state will be cleared and the next sequential instruction will be executed.

22. The syncing state will be cleared and the FIRQ interrupt service routine will be executed.

23. SYNC does *not* cause the 6809 registers to be stacked as does CWAI. In addition, CWAI does not allow for external hardware synchronization as does SYNC.

24. PSHS B
    ADDA ,S+

25. LEAX –1, X

CHAPTER 6

6809/6809E Input and Output Signals

INTRODUCTION

Now that you have completed the chapters about the 6809 software, you are ready to begin learning about the hardware aspects of both the 6809 and 6809E. It should be pointed out at this time that everything which has been discussed to this point will apply to both the 6809 and 6809E, except where noted otherwise. Recall that the 6809E is the off-chip clock version of the 6809. Thus the differences between the two devices are in the available i/o signals and *not* the instruction set. Because the 6809E does not require a crystal connection, two extra status lines are available. These two lines will facilitate *multiprocessor* system applications.

In this chapter we will discuss the 6809 and 6809E i/o signals in detail. For convenience, these signals have been broken down into five functional categories. They are: *power/clock, data/address, bus status, bus timing,* and *control.* You will find that both the 6809 and 6809E provide much better capabilities in all of these categories than did the 6800. For example, by decoding only five lines you can determine exactly which interrupt vector is being fetched and thus provide a complete interrupt acknowledge to an i/o device. In addition, the direct memory access (DMA) capabilities of the 6809 are greatly enhanced over the 6800 by the addition of the $\overline{\text{DMAREQ}}$ control line. In this chapter we will discuss three types of DMA which are available for the 6809 and 6809E. They are: *halt mode, cycle stealing,* and *bus multiplexing* DMA. Now, let's begin our discussion.

OBJECTIVES

At the end of this chapter you will be able to do the following:

- Describe the crystal and clock requirements of both the 6809 and 6809E.
- Explain the four possible *MPU states* of the 6809/6809E.
- Decode five 6809/6809E i/o signals to determine which of the interrupt vectors is being fetched.
- Design a decoding circuit to provide the above interrupt vector acknowledgment signal.
- Explain the relationship between the two bus timing signals, E and Q.
- Understand how the 6809 can be interfaced to *slow* peripheral devices, such as slow memories.
- Describe three different methods by which the 6809/6809E can be used to provide direct memory access (DMA) to peripheral devices.
- Explain the sequence of events associated with each of the 6809/6809E hardware interrupts.
- Understand the differences between the 6809 and 6809E i/o signals.

6809 PIN-OUTS

The 6809 "footprint," or pin configuration, is shown in Fig. 6-1. As stated earlier, the 6809 is a 40-pin HMOS (high-density NMOS) device available in either a standard plastic package (P suffix) or ceramic package (L suffix). In discussing the pin functions we will divide the pin-outs into the following five functional categories: *power/clock, data/address, bus status, bus timing,* and *control.* A detailed discussion of each follows:

Power/Clock

As with all 6800 family devices, the 6809 requires a single +5-Vdc ±5-percent supply voltage. Maximum current drain is 200 mA, for a maximum power dissipation of 1 watt. The two power connections are V_{SS} at pin 1 (ground) and V_{DD} at pin 7 (+5 Vdc).

The 6809 has an internal clock oscillator/driver and therefore does not *require* an external clock source as does the 6800. However, you must supply an external parallel-resonant crystal between pins 38 and 39 (EXTAL and XTAL) to provide crystal control of the internal oscillator's frequency. The standard 6809 operates with a maximum internal frequency of 1 MHz. There is, however, an internal circuit that divides the external crystal frequency by *4* to obtain the inter-

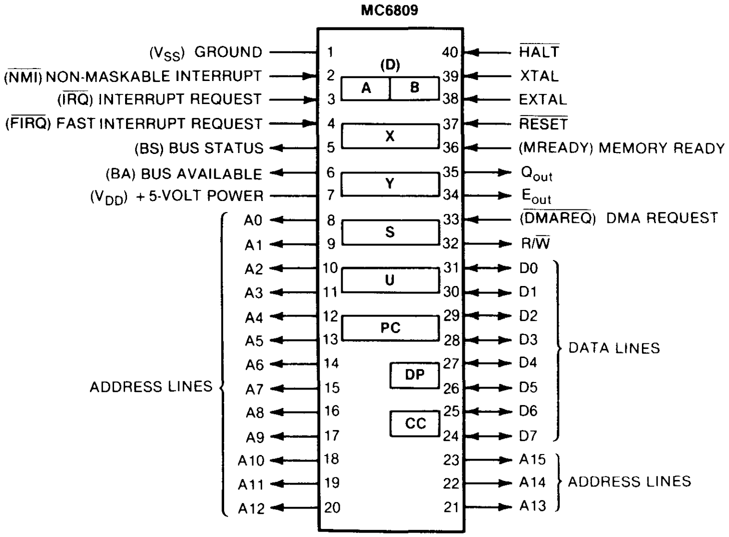
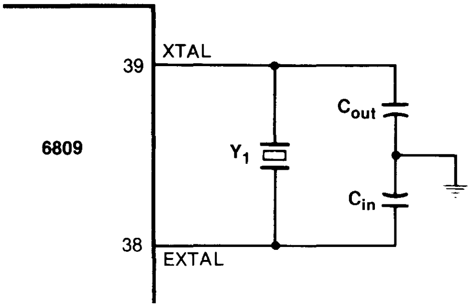


Fig. 6-1. 6809 pin configuration.

nal operating frequency. Therefore a 4-MHz crystal must be used to obtain a 1-MHz internal operating frequency. Actually, you can even use an inexpensive 3.58-MHz tv color-burst crystal and thus provide an internal operating frequency of 0.895 MHz, which is fast enough for many applications. The 6809 is also available in a 1.5-MHz version, the 68A09, and a 2.0-MHz version, the 68B09. For a maximum operating frequency these two versions would require crystals of 6 MHz and 8 MHz, respectively.

Now a word about speed. Many microprocessors boast very high clock frequencies as a selling point with the idea being that MPU cycle time, and therefore overall processing speed, is greatly enhanced. This is true to some extent; however, they are not telling the whole story. Clock frequency (MPU cycle time) alone is *not* the *only* factor which controls the speed of a processor. A better gauge of overall speed is *processor throughput.* This term takes into account not only MPU cycle time, but other important factors such as architecture, instruction set efficiency, addressing capabilities, bus timing, etc. Because of these factors the processing throughput of the 6809 is very high when compared with competitive devices. Therefore the 6809's internal clock frequency need not be above 2 MHz to achieve an equal or better performance than that of its competitors. At the same time, inherent high-frequency problems are avoided. In any event, if an application *does* require superhigh processing speeds, the solution is in a multiprocessor system and *not* one processor with a 100-MHz clock. Furthermore, the "bottom line" is what the *application* requires. Once a processor(s) can do the job in the time required, processor speed ceases to be an important design consideration.



(A) Circuit.

Y_1	C_{in}	C_{out}
4 MHz	24 pF	24 pF
6 MHz	20 pF	20 pF
8 MHz	18 pF	18 pF

(B) Component values.

Fig. 6-2. 6809 crystal connections.

The required 6809 crystal connections are shown in Fig. 6-2. Note that two capacitors (C_{in} and C_{out}) need to be connected from the crystal's pins to ground. This connection will ensure optimum chip performance. In addition, to minimize distortion the crystal should be mounted as close as possible to the EXTAL and XTAL pins. The crystal vendor specifications are given in the 6809 specification sheet (refer to Appendix C). Motorola recommends that LC networks *not* be used in lieu of a crystal and also warns that the internal oscillator may take as long as 20 milliseconds to become operational after power-up.

You may also drive the 6809 with an external TTL or CMOS clock signal. If this is desired, the signal must be applied to pin 38 (EXTAL) with pin 39 (XTAL) grounded. The external clock signal will be divided by 4 to establish the internal operating frequency.

Data/Address

As you are now probably aware, there are eight data and sixteen address pins located as shown in Fig. 6-1. Each pin will drive one standard low-power Schottky (LS) TTL load *plus* eight 6800 family devices at the rated bus speed. More than eight 6800-series devices can be driven if the clock speed is reduced to lower the capacitive loading effect of the bus. The data lines will typically drive 130 pF and the address lines 90 pF. Both the data and address lines are internally three-state buffered. The particular on/off state of any given line is a function of the various 6809 control signals. The associated control signals and data/address bus timing will be discussed shortly.

Bus Status

Two of the 6809's pins are used to communicate the MPU status to peripheral devices. They are: *bus status* (BS) at pin 5 and *bus available* (BA) at pin 6. *Bus available* (BA) provides an indication of the three-state logic status of the data, address and R/$\overline{W}$ lines. If BA=0, the processor is running and the data, address and R/$\overline{W}$ three-state logic is *on*. If BA=1, the processor is in a halt condition and the data, address, and R/$\overline{W}$ three-state logic are off or in the high-impedance state, thereby effectively disconnecting the processor from the external data and address buses and making those *buses available* to external devices. However, BA=1 does *not* imply that the data/address bus structure will be free for more than one MPU cycle. *Bus status* (BS) is used together with BA to indicate the *MPU state*. There are four possible logic combinations of these two lines and thus four unique MPU states. They are: *normal* (or running), *interrupt acknowledge* (IACK), *SYNC acknowledge*, and *halt* (or *bus grant*).

The above MPU states are shown with their respective BA/BS logic in Table 6-1. The normal or running state is obvious and needs no further discussion. The interrupt acknowledge (IACK) state will provide an indication (via BA/BS) to external devices that one of the 6809 interrupts ($\overline{\text{RESET}}$, $\overline{\text{NMI}}$, $\overline{\text{IRQ}}$, $\overline{\text{FIRQ}}$, SWI1, SWI2, or SWI3) has been accepted. The obvious use for the IACK state is to provide an external i/o device with an indication that its interrupt has been accepted. This can be done with external decoding logic to detect the BA=0, BS=1 (IACK) state. In addition, if address lines A1, A2, and A3 are also decoded with BA/BS, the decoding logic can indicate *exactly* which interrupt has been accepted. Recall that when an interrupt is accepted, its interrupt vector is fetched.

Table 6-1. MPU States

MPU State	Bus Available (BA)	Bus Status (BS)
Normal	0	0
Interrupt acknowledge	0	1
Sync acknowledge	1	0
Halt/Bus grant	1	1

Since each interrupt has a unique vector address assignment, one of fourteen addresses (FFF2–FFFF) will be placed on the address bus when the vector is fetched. Thus the BA/BS logic will indicate that an interrupt has been accepted and address lines A1–A3 will indicate exactly which interrupt vector is being fetched. The simple 74154 decoding circuit shown in Fig. 6-3 could be used for this purpose. Here an active (low) output on line 2 would indicate that an SWI3 interrupt has been accepted since SWI3 has an interrupt vector assignment of FFF2:FFF3; an active output on line 8 would indicate that an $\overline{\text{IRQ}}$ has been accepted since $\overline{\text{IRQ}}$ has a vector assignment of FFF8:FFF9, and so on. Address line A0 does not need to be decoded since all the interrupt vector addresses begin with an *even* address and therefore A1 is the least significant line which needs to be decoded. You could even use the active output of this circuit to *turn off* the ROM containing the interrupt vectors. Then, load *your* own vector on the data bus and thereby redefine the interrupt service routine locations. Thus an external device can define its own interrupt vector.

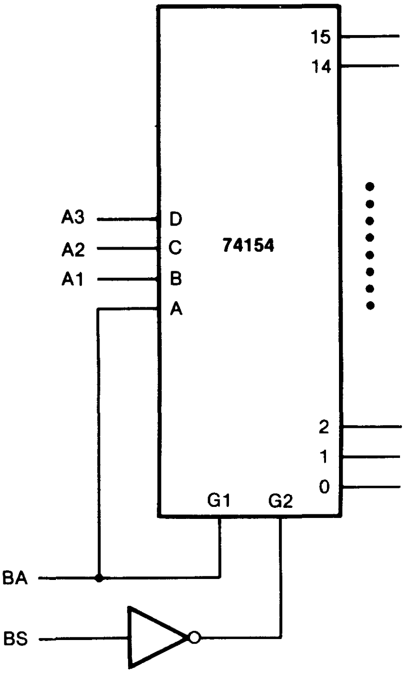


Fig. 6-3. IACK decoding circuit.

The SYNC acknowledge state indicates that the 6809 is in the *syncing state* as the result of a SYNC instruction. The syncing state was discussed in the previous chapter. In the next chapter you will see how this state is decoded to provide external hardware synchronization.

The *halt* (or *bus grant*) MPU state means exactly what it says. The MPU has halted all execution and the data, address and $R/\overline{W}$ lines are in their high-impedance state, thus making the data and address buses available for external use such as direct memory access (DMA). This state will normally exist as the result of a low level on the $\overline{DMAREQ}$ or $\overline{HALT}$ pins (to be discussed presently).

Another line which could be considered a status line is the read/write ($R/\overline{W}$) line at pin 32. This line serves the same purpose as the 6800 $R/\overline{W}$ line, that is, to indicate the direction of data transfer on the data bus. If $R/\overline{W} = 1$, the 6809 is performing a *read* operation, while $R/\overline{W} = 0$ indicates a *write* operation is being performed. This line is three-state and is placed in its high-impedance state whenever BA = 1, indicating that the data bus is available for external use. Another use for the R/W line is to determine *valid memory addresses*. Recall that the 6800 has a valid memory address line (VMA) which is used to indicate the validity of addresses placed on the address bus. You had to use this line in your external decoding circuits such that all external devices were disabled if a nonvalid address (VMA=0) existed on the address bus. The 6809 does *not* have a VMA line. Instead, when the 6809 is not using the data bus for data transfer, it will place $FFFF_{16}$ on the address bus and output $R/\overline{W} = 1$ and BS = 0. This will replace the 6800 VMA function since external decoding circuits can be designed to recognize the above conditions as a nonvalid memory address state. Note that the nonvalid memory address condition is distinguished from a $\overline{RESET}$ interrupt vector fetch by the bus state (BS) line, since BS = 1 for a vector fetch.

Bus Timing

The E_{out} (pin 34) and Q_{out} (pin 35) lines provide external clock signals for timing and synchronization. E_{out} is the standard 6800 family timing signal similar to the $\phi 2$ clock signal on the 6800. When E_{out} goes high, this indicates to i/o devices that the address information has been placed on the address bus to provide a sufficient setup time and that data is being placed on the data bus. This line defines the MPU cycle and should be part of any external decoding scheme. The Q_{out} line is a *quadrature* clock signal which *leads* E_{out} by 90°. The relationship between E_{out} and Q_{out} is shown in Fig. 6-4. The frequency of each equals the internal operating frequency, but they are displaced by 90° and thus provide four separate clock edges for interfacing purposes. The 6809 read/write timing diagrams are shown in Fig. 6-5. Note that address information

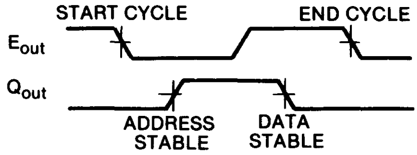
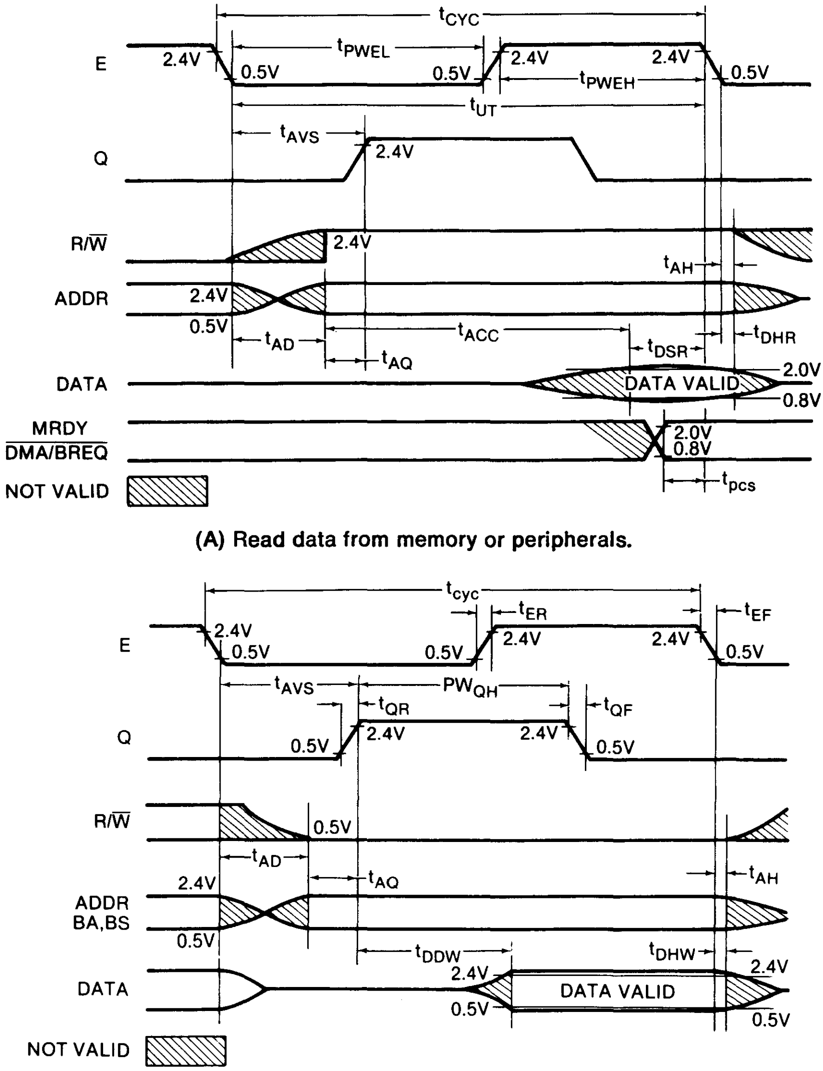


Fig. 6-4. Relationship between E_{out} and Q_{out}.



(A) Read data from memory or peripherals.

(B) Write data to memory or peripherals.

Fig. 6-5. Read and write timing diagrams.

is valid at the leading edge of Q_{out} with data being valid during E_{out}. Data, in or out, is latched with the falling edge of E_{out}.

The memory ready (MREADY) line (pin 36) is used in conjunction with the E_{out} and Q_{out} timing signals. Recall that data is valid on the data bus as long as E_{out} is high. For some i/o devices, especially slow memories, this is not long enough to satisfy the data setup, access, and hold time requirements of the device. Therefore some means must be provided to *stretch* the E_{out} and Q_{out} pulses such that slow memories will have time to respond to the 6809's signals. This is the function of the MREADY line. By holding MREADY low, you can stretch the E_{out} and Q_{out} pulses to extend data access time. As long as MREADY is held low, the data valid period will be extended until the MREADY line is returned high. However, the maximum stretch period is 10 microseconds. (A 100-microsecond stretch period can be obtained from a special-order 6809, at a higher price.) The effect of MREADY on E_{out} and Q_{out} is shown in Fig. 6-6. Some of the 6809s produced early in the product cycle (mask numbers G7F, T5A, P6F, and T6M) require that the MREADY signal be synchronized with the crystal frequency. Consult the 6809 specification in Appendix C for a synchronization circuit, if you have one of these earlier devices.

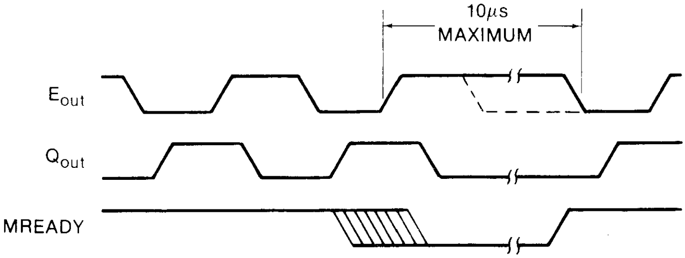


Fig. 6-6. Effects of MREADY on E_{out} and Q_{out}.

Control

The remaining 6809 pins are used for control purposes. Our discussion will begin with $\overline{\text{HALT}}$ and $\overline{\text{DMAREQ}}$ and end with the 6809 hardware interrupts. The $\overline{\text{HALT}}$ pin provides for a hardware salt of the 6809 operation. When you supply a logic 0 state to pin 40, the 6809 will stop running at the end of the current instruction. When this happens the three-state buffers on the data, address, and R/$\overline{\text{W}}$ lines will go into their high-impedance state and effectively disconnect the 6809 from the external data and address buses. As long as a low level exists at pin 40, the 6809 will remain halted without any loss of internal register data, the E_{out} and Q_{out} bus timing signals will continue normal operation, and the BA/BS logic will indicate the $\overline{\text{HALT}}$ MPU state. When halted, the 6809 will not respond to any external control signals except $\overline{\text{DMAREQ}}$ (to be discussed shortly). However, if $\overline{\text{RESET}}$ or $\overline{\text{NMI}}$ interrupts are received, they will be latched for execution after the halt mode. The $\overline{\text{HALT}}$ function is normally used for hardware troubleshooting and program debugging, since it allows an external device to control program execution one step at a time. It can also be used for a halt mode *direct memory access* (DMA), since an external device can gain control of the external buses. When not in use, you can connect the $\overline{\text{HALT}}$ pin to the +5-volt dc supply for uninterrupted system operation.

$\overline{\text{DMAREQ}}$ (pin 33) provides for another external method of stopping the 6809. This line will normally be used to allow fast access to the bus for direct memory access (DMA) or dynamic memory refresh. It is appropriate at this time to say a few words about direct memory access. This technique provides a means for high-speed data transfers *directly* between a peripheral device, such as a floppy disc, and memory without going through the MPU. The direct memory access function is normally controlled by a single-chip *direct memory access controller* (*DMAC*). The DMAC takes bus control away from the MPU and controls the data transfer directly between memory and a selected peripheral device. The Motorola 6800 family DMAC is the MC6844.

With the 6809, DMA can be performed by one of three methods. They are: *halt mode*, *cycle stealing*, and *bus multiplexing*. With halt-mode DMA the DMAC pulls the 6809 $\overline{\text{HALT}}$ pin low and takes control of the address and data buses. The 6809 remains halted until the data transfer is completed. The bus timing signals (E_{out} and Q_{out}) will continue to provide timing for the data transfer. Cycle-stealing DMA is accomplished when the DMAC pulls the $\overline{\text{DMAREQ}}$ line (pin 33) low. When this happens the 6809 will stop instruction execution at the end of the current cycle and acknowledge the DMA request by setting both the BA and BS bus status lines. The 6809 address, data, and R/$\overline{\text{W}}$ lines will then be placed in the high-impedance state and the DMAC can then take control of the bus structure. The DMAC will have up to fifteen MPU cycles for data transfer before the 6809 will automatically *steal* the bus structure for one MPU cycle to refresh the internal MPU registers. When this happens the BA line will be cleared to signal the DMAC to get off the bus structure. After one cycle, control will be transferred back to the DMAC, provided that the $\overline{\text{DMAREQ}}$ line (pin 33) is still held low. This process will be repeated until $\overline{\text{DMAREQ}}$ is returned to a high (logic 1) level. The $\overline{\text{DMAREQ}}$ sequence of events is summarized in Fig. 6-7. The third method of DMA, bus multiplexing, provides that DMA peripheral devices be gated onto the bus struc-

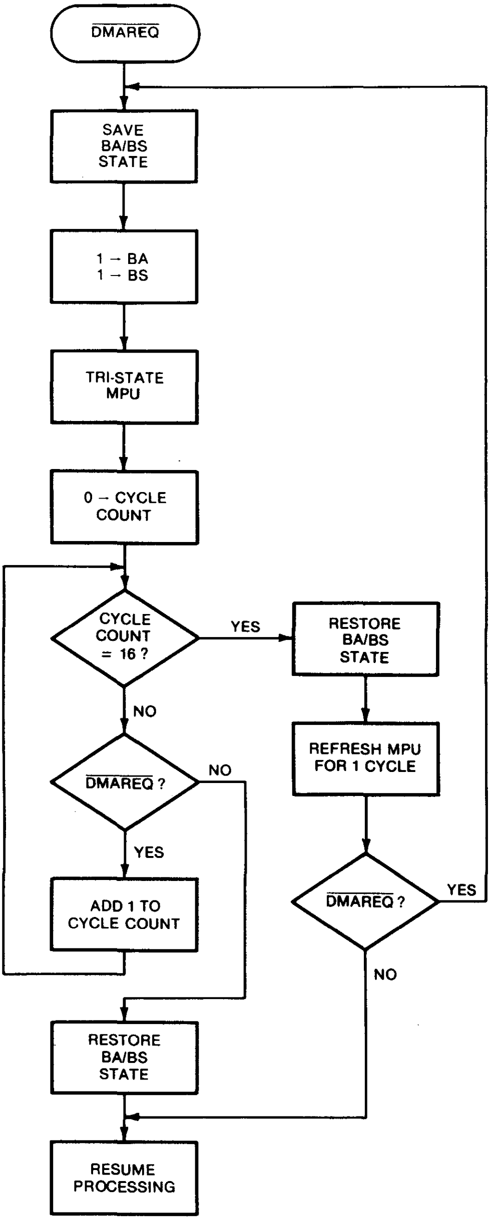


Fig. 6-7. $\overline{\text{DMAREQ}}$ sequence of events.

ture when E_{out} is low (MPU data not valid). With this method the 6809 shares the bus structure (50/50) with the DMA peripheral devices. This method is less efficient than the other two and requires more external decoding logic. Therefore it is less often used. Consult the 6809 specification in Appendix C for halt-mode ($\overline{\text{HALT}}$) and cycle-stealing ($\overline{\text{DMAREQ}}$) timing diagrams. In a typical application, DMA peripherals such as floppy discs would use the halt-mode of DMA, with the cycle-stealing DMA used to refresh dynamic memory. Dynamic memory devices require refreshing every 2 milliseconds and are very high priority. Recall that the 6809 will respond to $\overline{\text{DMAREQ}}$ when in the halt mode.

The last four 6809 pins that remain to be discussed represent the four 6809 hardware interrupts: $\overline{\text{RESET}}$, non-maskable interrupt ($\overline{\text{NMI}}$), fast interrupt request ($\overline{\text{FIRQ}}$), and interrupt request ($\overline{\text{IRQ}}$). Refer to Fig. 6-1 for their respective pin assignments. All but one of these should be familiar to 6800 users. The new kid on the block is $\overline{\text{FIRQ}}$, and our previous discussions have introduced you to this one. Now, let's discuss each interrupt in more detail. First, note from Fig. 6-1 that they are all "barred" ($\overline{\quad\quad}$), meaning that they are all active low. However, only $\overline{\text{NMI}}$ is *edge* triggered and thus activated by a high-to-low transition on pin 2. The other three ($\overline{\text{RESET}}$, $\overline{\text{FIRQ}}$, and $\overline{\text{IRQ}}$) are all *level* triggered and must remain low for at least one MPU cycle to be properly clocked or sensed by the 6809.

$\overline{\text{RESET}}$ is used to initialize or restart the system. It provides a starting point for program execution by placing the address of the first instruction to be executed in the program counter. A reset subroutine, located in ROM, will be required to perform the initialization task. Reset is always required after power is applied to the system and should be held low until the internal clock oscillator is fully operational, and then released. Recall that it may take up to 20 milliseconds for the internal clock to become operational. This is not normally a problem with a manual reset; however, it must be considered for automatic resets. The $\overline{\text{RESET}}$ pin uses a Schmitt-trigger input and therefore a simple RC network can be used to reset the system. However, this Schmitt-trigger input requires a *minimum* of 4.0 Vdc to represent a high (logic 1) level, whereas other 6809 signals may use the standard TTL-compatible 2.4 Vdc as a minimum value to represent a high (logic 1) state. An advantage of using an RC network on the $\overline{\text{RESET}}$ input is that, because of the RC time constant, other peripheral devices will have had enough time to completely reset themselves before the 6809 completes its reset routine and begins any peripheral initialization routines. For example, when the 6809 starts up and initializes a peripheral interface adapter (PIA), you can be assured that the PIA

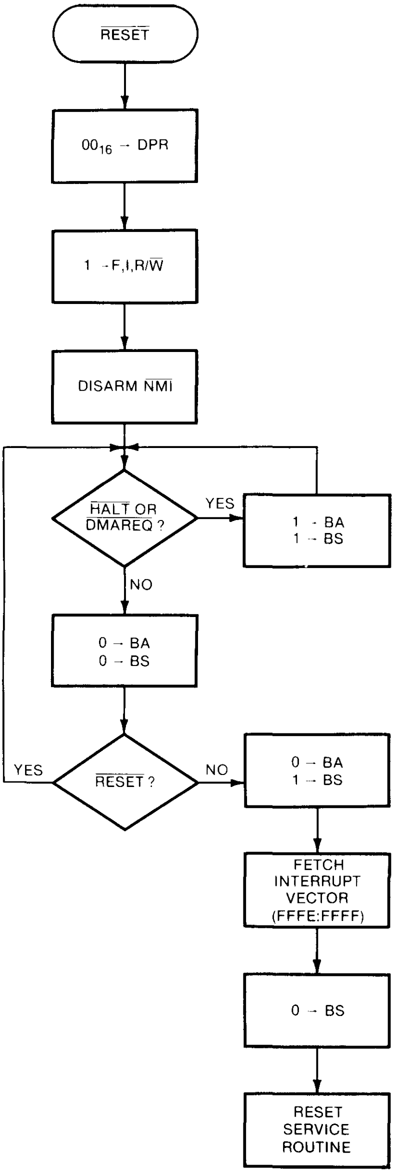


Fig. 6-8. RESET sequence of events.

is no longer in its own reset mode. The sequence of events which take place when RESET (pin 37) is active is shown in Fig. 6-8. When RESET becomes active for at least one MPU cycle the present instruction is *aborted* and the direct page register is cleared.

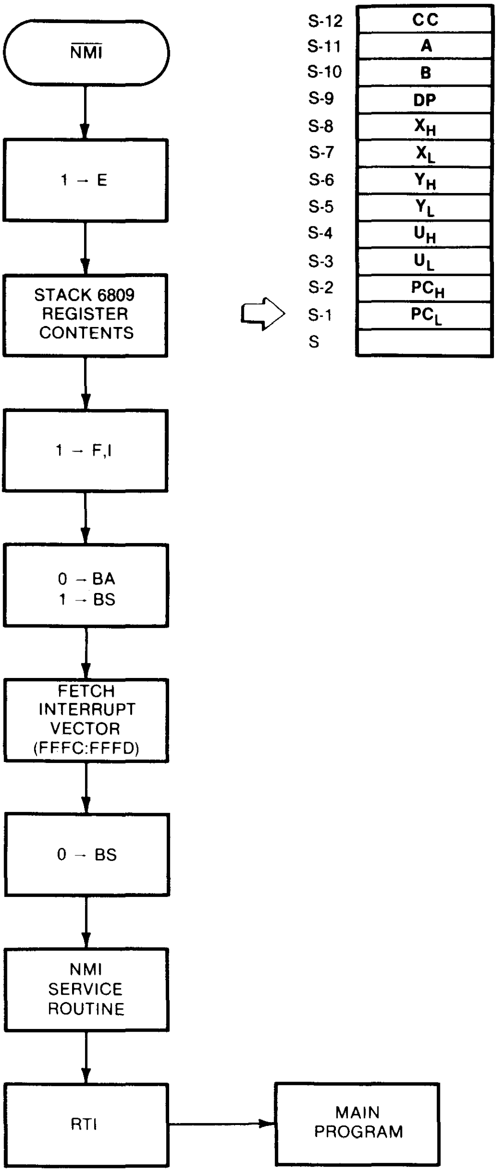


Fig. 6-9. $\overline{\text{NMI}}$ sequence of events.

All of the other hardware interrupts ($\overline{\text{NMI}}$, $\overline{\text{IRQ}}$, and $\overline{\text{FIRQ}}$) are either masked out or disarmed. However, if a non-maskable interrupt ($\overline{\text{NMI}}$) occurs during the reset sequence, its active edge will be latched (saved) and the $\overline{\text{NMI}}$ sequence of events (Fig. 6-9) will be executed immediately after any instruction which loads the

S register (LDS; EXG R, S; TFR R, S; LEAS 1, X; etc.). The *only* event which can interrupt the reset sequence is a $\overline{\text{HALT}}$ or $\overline{\text{DMAREQ}}$. If no active $\overline{\text{HALT}}$ or $\overline{\text{DMAREQ}}$ exists, the bus status signals, BA and BS, will be cleared to indicate the MPU-running state. When $\overline{\text{RESET}}$ goes back high (to 4.0 volts) the bus status signals will indicate an interrupt acknowledge (BA=0, BS=1). The $\overline{\text{RESET}}$ interrupt vector will then be fetched from address FFFE: FFFF, BS will be cleared to indicate the MPU-running state and the reset interrupt service routine will be executed. Now, if $\overline{\text{HALT}}$ or $\overline{\text{DMAREQ}}$ is active during the initial reset sequence, the halt-mode or cycle-stealing mode of DMA will take place. If $\overline{\text{RESET}}$ goes back high during a DMA operation, the 6809 will complete the reset sequence after the DMA operation is completed. Note: $\overline{\text{HALT}}$ or $\overline{\text{DMAREQ}}$ *may* interrupt the reset sequence; however, $\overline{\text{RESET}}$ will *not* interrupt $\overline{\text{HALT}}$ or $\overline{\text{DMAREQ}}$.

The 6809 non-maskable interrupt ($\overline{\text{NMI}}$) is very similar to the 6800 $\overline{\text{NMI}}$. As the name implies, this interrupt cannot be masked by the programmer. The *only* time that an $\overline{\text{NMI}}$ is not accepted is during the reset operation. Recall from our discussion of $\overline{\text{RESET}}$ that the $\overline{\text{NMI}}$ logic is disarmed during the reset sequence. However, any $\overline{\text{NMI}}$ occurring during the reset operation will be latched (saved), with the $\overline{\text{NMI}}$ sequence occurring after the first load into the S register. In addition, if a non-maskable interrupt occurs during the $\overline{\text{HALT}}$ operation, the active $\overline{\text{NMI}}$ state will be latched and the $\overline{\text{NMI}}$ sequence will not be executed until after the MPU is released from the $\overline{\text{HALT}}$ state. The $\overline{\text{NMI}}$ sequence of events is shown in Fig. 6-9. The E flag is set to indicate that *all* of the internal registers are to be stacked. The registers are then stacked in the order shown. The F and I flags are set to mask out any $\overline{\text{FIRQ}}$ or $\overline{\text{IRQ}}$ (maskable) interrupts such that the $\overline{\text{NMI}}$ sequence will not be interrupted by a maskable-type interrupt. However, another active NMI could interrupt an existing $\overline{\text{NMI}}$ sequence since $\overline{\text{NMI}}$ is not masked by the execution of the $\overline{\text{NMI}}$ sequence. If this is allowed to occur repeatedly before the sequence can be completed, the stack will definitely overflow. When the $\overline{\text{NMI}}$ interrupt vector is fetched at address FFFC:FFFD, the bus status lines (BA/BS) indicate an interrupt acknowledge (IACK). Once the vector is fetched and loaded into the program counter, the bus status (BS) line is cleared to indicate the normal MPU running state and the $\overline{\text{NMI}}$ interrupt service routine is executed. When the service routine has been completed, execution must be directed back to the main program. This is done by inserting a return from interrupt (RTI) instruction as the last instruction in the interrupt service routine. The RTI returns the 6809 to its previous status by unstacking the old register contents.

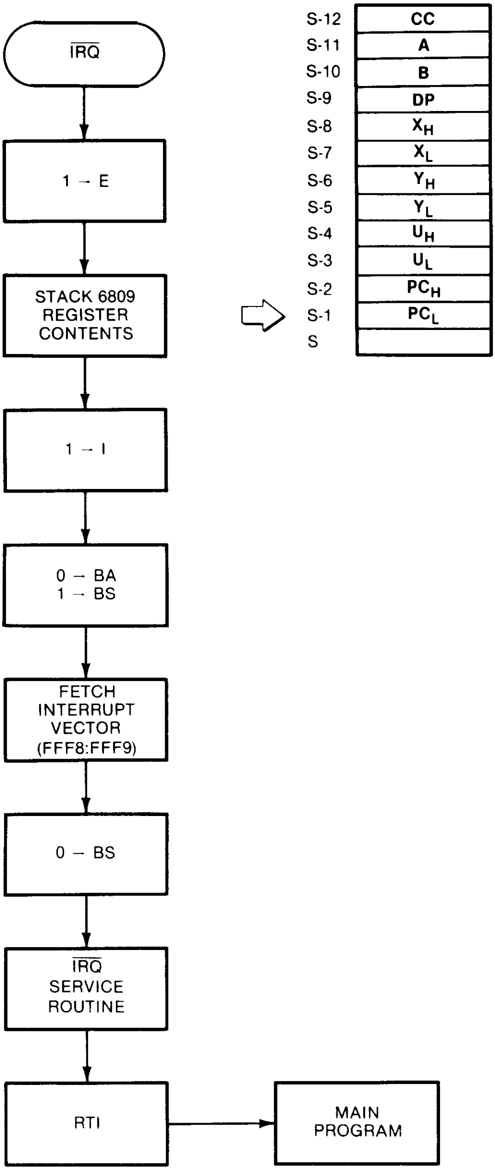


Fig. 6-10. $\overline{\text{IRQ}}$ sequence of events.

The two 6809 maskable interrupts are the normal 6800-type interrupt request ($\overline{\text{IRQ}}$) and the new fast interrupt request ($\overline{\text{FIRQ}}$). The $\overline{\text{IRQ}}$ sequence of events is shown in Fig. 6-10. You may *mask out* the $\overline{\text{IRQ}}$ by setting the I flag of the condition code register.

If the I flag is cleared (logic 0), the interrupt will be accepted. Once accepted, the sequence of events for $\overline{\mathrm{IRQ}}$ is similar to that of $\overline{\mathrm{NMI}}$. The E flag is set and *all* of the internal registers are stacked on the S stack in the order shown. Then, the I flag is set to mask out any further $\overline{\mathrm{IRQ}}$s until the present sequence is completed. Note, however, that the F flag is *not* set and therefore the $\overline{\mathrm{IRQ}}$ sequence may be interrupted by a fast interrupt request ($\overline{\mathrm{FIRQ}}$). You can therefore conclude that $\overline{\mathrm{FIRQ}}$ is of higher priority than $\overline{\mathrm{IRQ}}$. When the $\overline{\mathrm{IRQ}}$ interrupt vector is fetched at address FFF8:FFF9, the bus status lines indicate an interrupt acknowledge (IACK) MPU state. Once the vector has been fetched, the bus status (BS) line is cleared to indicate the normal running MPU state and the $\overline{\mathrm{IRQ}}$ interrupt service routine is executed. Again, you must use the RTI instruction at the end of your interrupt service routine to get back to the main program.

The sequence of events for $\overline{\mathrm{FIRQ}}$ is shown in Fig. 6-11. Note the differences from the $\overline{\mathrm{IRQ}}$ sequence (Fig. 6-10). The E flag is *cleared* to indicate that only the program counter and condition code register are to be stacked. Both the F and I flags are set such that the sequence will not be interrupted by any $\overline{\mathrm{IRQ}}$s or further $\overline{\mathrm{FIRQ}}$s. However, your interrupt service routine could clear the I flag if the automatic $\overline{\mathrm{FIRQ}}/\overline{\mathrm{IRQ}}$ priority was not desired. The remaining events are the same as those of the $\overline{\mathrm{IRQ}}$ sequence except that the $\overline{\mathrm{FIRQ}}$ interrupt vector is located at address FFF6:FFF7. As stated earlier, the advantage of $\overline{\mathrm{FIRQ}}$ over $\overline{\mathrm{IRQ}}$ is in saving time when *all* the internal register data need not be saved during the interrupt. If additional register data need to be saved, you can always use PSHS/PULS in your $\overline{\mathrm{FIRQ}}$ interrupt service routine.

6809E PIN-OUTS

The 6809E, sister to the 6809, is also a 40-pin HMOS device and is available in either a plastic (P) package or ceramic (L) package. This device is intended primarily for multiprocessor system use, since it provides extra signal lines for additional MPU status information. An example of the 6809E's use in a multiprocessor system will be given in the next chapter. The 6809E footprint is shown in Fig. 6-12. Two extra signal lines (BUSY and LIC) are provided in lieu of the standard 6809 crystal connect pins (XTAL and EXTAL). Moreover, the 6809E includes an advanced valid memory address (AVMA) line in place of the 6809 memory ready (MREADY) line and a three-state control (TSC) line replaces the 6809 $\overline{\mathrm{DMAREQ}}$ line. As stated earlier, the 6809E is an off-chip clock version of the 6809. Therefore you must supply the required E_{in} and Q_{in} clock signals at pins 34 and 35, respectively. The phase relationship be-

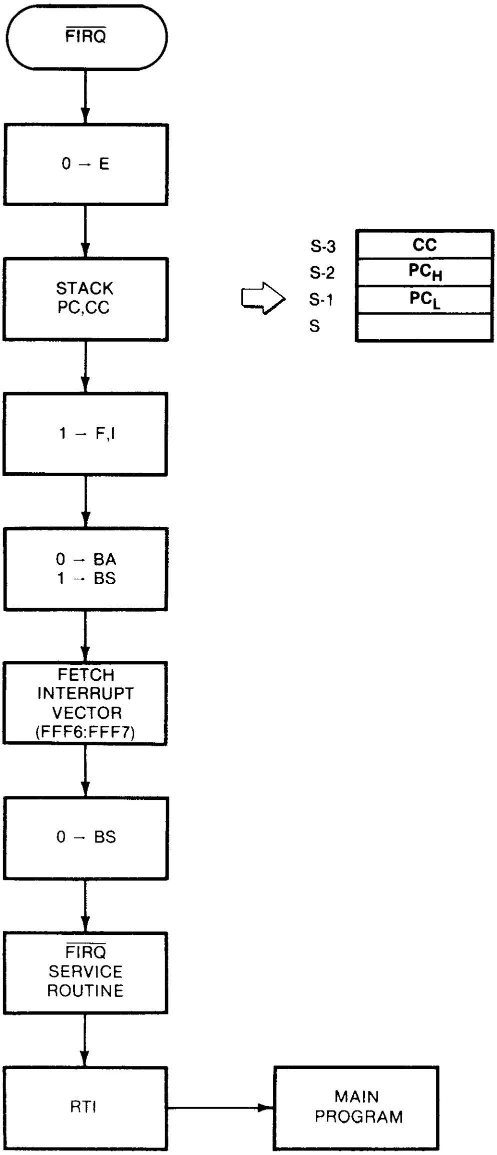


Fig. 6-11. $\overline{\text{FIRQ}}$ sequence of events.

tween E_{in} and Q_{in} is the same as that of E_{out} and Q_{out} for the standard 6809 (Refer to Fig. 6-4). The Q_{in} pin (pin 35) is fully TTL compatible. However, the E_{in} pin (pin 34) drives internal MOS circuits which require levels which are above and below normal TTL values. Fig. 6-13 shows a circuit which will provide the proper

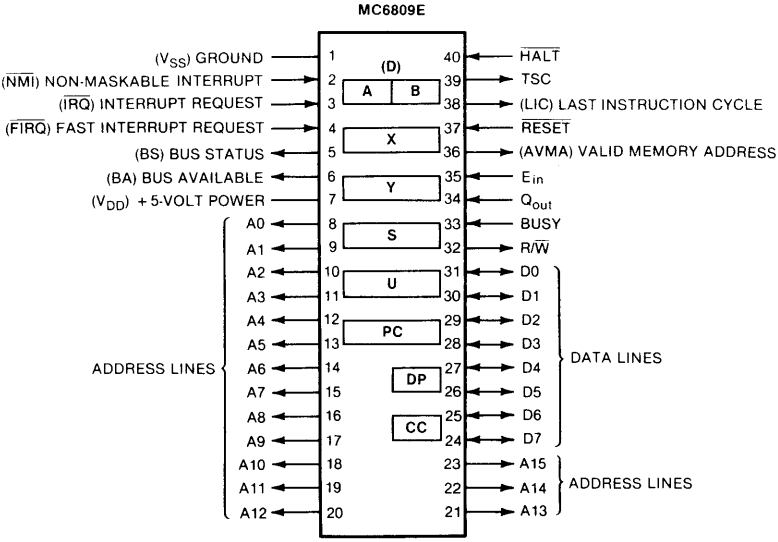

MC6809E
(V_{SS}) GROUND 1
(NMI) NON-MASKABLE INTERRUPT 2
(IRQ) INTERRUPT REQUEST 3
(FIRQ) FAST INTERRUPT REQUEST 4
(BS) BUS STATUS 5
(BA) BUS AVAILABLE 6
(V_{DD}) +5-VOLT POWER 7
ADDRESS LINES A0–A12 8–20
(D) A B X Y S U PC DP CC
40 HALT
39 TSC
38 (LIC) LAST INSTRUCTION CYCLE
37 RESET
36 (AVMA) VALID MEMORY ADDRESS
35 E_{in}
34 Q_{out}
33 BUSY
32 R/W
31–24 D0–D7 DATA LINES
23 A15
22 A14
21 A13 ADDRESS LINES

Fig. 6-12. 6809E pin configuration.

E_{in}/Q_{in} phase relationships and levels from one oscillator. The E_{in}/Q_{in} lines from this circuit can be applied directly to pins 34 and 35, respectively, of the 6809E. The E_{out}/Q_{out} lines will provide the

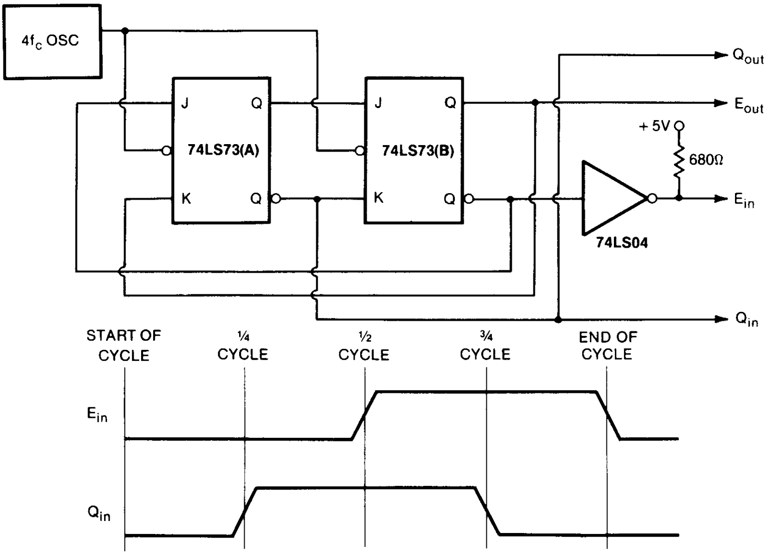


Fig. 6-13. 6809E clock generator circuit.

proper TTL signals for external timing. Note that the oscillator frequency is four times the desired clock frequency (f_c). The clock frequency obtained from this circuit will drive the 6809E directly and is *not* divided by 4 with internal 6809E logic. The standard operating frequency for the 6809E is 1 MHz ($f_c = 1$ MHz). However, the 6809E is also available in 1.5-MHz (68A09E) and 2-MHz (68B09E) versions.

Two additional status lines, BUSY (pin 33) and last instruction cycle (LIC, pin 38) are provided with the 6809E. The BUSY output line indicates that the 6809E is accessing memory. This line will be high through the entire read/write cycle, then return low when the cycle is completed. In multiprocessor systems this signal is used to supervise the system, so that only one processor has access to global memory or peripheral devices at a given time.

- *global memory/peripheral device*—that amount of memory or a peripheral device which is common to more than one, or to all the processors in a multiprocessor system. Just the opposite of a dedicated memory or peripheral device.

Of course, external *bus arbitration logic* will be required to interpret the various MPU status signals and *arbitrate* the global memory/peripherals between the processors.

The last instruction cycle (LIC) status line will also facilitate the use of the 6809E in multiprocessor systems. Pin 38 will be activated (high) during the last MPU cycle of any instruction. Therefore the next cycle will be an op-code fetch. This line may be used to signal multiprocessor bus arbitration logic that an MPU is about to complete its instruction cycle and thus provide a head start for the arbitration process. The 6809E also includes an advanced valid memory address (AVMA) output line at pin 36. It will go to a logic 1 state when the 6809E is about to use the data/address bus structure. A low-to-high transition at pin 36 indicates that the MPU will use the bus structure during the following bus timing cycle. Thus in a shared-bus system, other devices, such as DMACs and MPUs, are altered so that the bus structure must be relinquished.

Finally, the 6809E three-state control (TSC) line (pin 39) is similar to the 6809 $\overline{\text{DMAREQ}}$ line (pin 33). When TSC is active high the 6809E address, data, and R/$\overline{\text{W}}$ lines will be placed in their high-impedance state. When TSC is active (high) the MPU input clock signals (E_{in}/Q_{in}) must be stopped, then restarted when TSC is brought back low. Thus TSC can be used to provide a cycle-stealing mode of DMA or for dynamic memory refresh. The remaining 6809E pins provide the same functions as their 6809 counterparts, which have already been discussed. Furthermore, as stated earlier, there is *no* difference between the 6809E and 6809 instruction sets.

That concludes our discussion of the 6809/6809E hardware. In the next chapter some interfacing hints and ideas as well as some special applications of both the 6809 and 6809E will be given.

REVIEW QUESTIONS

1. A 3.2-MHz crystal would provide a 6809 internal clock frequency of ________.

2. How must the XTAL and EXTAL pins be connected to drive the 6809 with an external TTL or CMOS clock signal?

3. The best measure of overall MPU speed is ________.

4. What is the drive capability of each 6809/6809E data and address line?

5. The *three* 6809 bus status lines are ________, ________, and ________.

6. The bus available (BA) line is at a ________ level when the MPU address and data lines are in their high-impedance state.

7. The four MPU states which are represented by the BA and BS status lines are ________, ________, ________, and ________.

8. Suppose BA=0, BS=1, and A3 A2 A1 = 110. What is the MPU doing?

9. Which 74154 output line would be active in Fig. 6-3 for the conditions given in Question 8?

10. Which MPU state will be indicated on the bus status lines when a low level (logic 0) is applied to the $\overline{\text{DMAREQ}}$ pin?

11. How does the 6809 indicate a nonvalid memory address?

12. How is the nonvalid memory address state distinguished from a $\overline{\text{RESET}}$ interrupt vector fetch?

13. What is the relationship between E_{out} (E_{in}) and Q_{out} (Q_{in})?

14. When is data valid on the 6809/6809E data bus?

15. What is the function of the 6809 MREADY pin?

16. What will happen if a non-maskable interrupt ($\overline{\text{NMI}}$) occurs when a low level is being applied to the $\overline{\text{HALT}}$ pin?

17. What will happen if a low level is applied to the $\overline{\text{DMAREQ}}$ pin when the $\overline{\text{HALT}}$ pin is low?

18. A device which controls the DMA function is called a __________ and is represented by the MC __________ in the 6800 family.

19. The three ways that direct memory access can be accomplished with the 6809 are: __________, __________, and __________.

20. Using the DMAREQ line for cycle-stealing DMA, the MPU will steal the

    address/data bus structure every __________ MPU cycles to refresh the internal register data.

21. The only 6809/6809E hardware interrupt which is edge triggered is the

    __________ interrupt.

22. How long should $\overline{\text{RESET}}$ be held low when power is applied to the system?

23. A high (logic 1) level is represented on the $\overline{\text{RESET}}$ pin by __________ volts.

24. The only two events which will interrupt the reset sequence are ________

    and __________.

25. What events could interrupt the $\overline{\text{NMI}}$ sequence?

26. What are the differences between the $\overline{\text{IRQ}}$ and $\overline{\text{FIRQ}}$ sequences of events?

27. Which interrupt is of higher priority, $\overline{\text{IRQ}}$, or $\overline{\text{FIRQ}}$?

28. The 6809E is best applied to __________ systems.

29. The three *additional* bus status lines available to the 6809E are

    __________, __________, and __________.

30. The 6809E line which is similar to the 6809 $\overline{\text{DMAREQ}}$ line is __________.

ANSWERS

1. 0.8 MHz

2. The TTL or CMOS signal is applied to EXTAL with XTAL connected to ground. The applied frequency must be four times the desired clock frequency.

3. Processor throughput

4. One standard Schottky TTL load *plus* eight 6800 family devices at the rated bus speed

5. Bus available (BA), bus status (BS), read/write ($R/\overline{W}$)

6. High

7. Normal, interrupt acknowledge (IACK), SYNC acknowledge and halt/bus grant

8. BA=0 and BS=1 represents the interrupt acknowledge (IACK) MPU state and indicates that an interrupt vector is being fetched. Since A3 A2 A1 = 110_2, the lower *four* address lines (A3 A2 A1 A0) will represent either 1100_2 (C_{16}) or 1101_{16} (D_{16}). In either case the non-maskable interrupt ($\overline{NMI}$) vector is being fetched. Note the status of A0 does not have to be known to decode the proper interrupt vector.

9. Line 12

10. Halt/bus grant

11. Address lines A0–A15 are all high ($FFFF_{16}$), $R/\overline{W} = 1$, and BS = 0

12. BS=0 for a nonvalid memory address and BS=1 for a $\overline{RESET}$ interrupt vector fetch

13. E_{out} (E_{in}) is similar to the 6800 $\phi 2$ clock. Q_{out} (Q_{in}) is called the quadrature clock. E_{out} (E_{in}) and Q_{out} (Q_{in}) are all the same frequency; however, Q_{out} (Q_{in}) leads E_{out} (E_{in}) by 90° or ¼ clock cycle.

14. When E_{out} (E_{in}) is at a logic 1 level (high)

15. To stretch the E_{out} and Q_{out} pulses a maximum of 10 microseconds such that slow peripheral devices will have time to respond to the 6809's signals.

16. The $\overline{HALT}$ state will not be interrupted. However, the active $\overline{NMI}$ state will be latched and the $\overline{NMI}$ sequence will be executed when the $\overline{HALT}$ pin returns high.

17. The $\overline{DMAREQ}$ sequence of events (Fig. 6-7) will be executed.

18. Direct memory access controller (DMAC), MC6844

19. Halt-mode ($\overline{HALT}$ pin), cycle-stealing ($\overline{DMAREQ}$ pin), and bus multiplexing (external logic)

20. Fifteen

21. Non-maskable ($\overline{NMI}$)

22. Until the internal clock oscillator becomes fully operational (approximately 20 milliseconds)

23. A minimum of 4 volts

24. $\overline{HALT}$ and $\overline{DMAREQ}$

25. Any of the following:

    Another $\overline{\text{NMI}}$
    $\overline{\text{RESET}}$
    $\overline{\text{DMAREQ}}$
    $\overline{\text{HALT}}$

26. a. $\overline{\text{IRQ}}$ sets the E flag, $\overline{\text{FIRQ}}$ clears the E flag.
    b. $\overline{\text{IRQ}}$ stacks all internal registers; $\overline{\text{FIRQ}}$ stacks only PC and CCR.
    c. $\overline{\text{IRQ}}$ sets the I flag; $\overline{\text{FIRQ}}$ sets both the F and I flags.
    d. $\overline{\text{IRQ}}$ vector is at address FFF8:FFF9;
    $\overline{\text{FIRQ}}$ vector is at address FFF6:FFF7.

27. $\overline{\text{FIRQ}}$

28. Multiprocessor

29. BUSY, LIC, and AVMA

30. Three-state control (TSC)

CHAPTER 7

6809/6809E Interfacing and Applications

INTRODUCTION

You are now ready to apply your knowledge of the 6809 to the "real world." Interfacing the 6809 is very similar to interfacing the 6800. The major difference is in the use of the bus status (BS) and $\overline{\text{DMAREQ}}$ signals, which were discussed in the previous chapter. We will begin our discussion by developing a *minimum* 6809 system consisting of scratch-pad R/W memory, ROM, and a peripheral interface adapter (PIA). The 6820/6821 PIA will be used in just about every 6809 system; thus a thorough understanding of this device should be attained. A complete discussion of the PIA is provided in Appendix B, if a review is needed.

Since the 6809 is mainly a *systems* device, most of our discussion in this chapter will center on 6809 systems applications. After developing a minimum system we will discuss an expanded system which provides serial data communication for crt's, modems, and printers along with mass storage via a floppy disc system. Such a system can *stand alone* or can be integrated into a *multiprocessor* system. The 1980s will see the development of many multiprocessor systems with 8-bit devices, such as the 6809, handling dedicated tasks and 16-bit devices, such as the 68000, providing system control and supervision. You will see how the 6809 can be interfaced to a 16-bit data bus for such multiprocessor applications. In addition, you will see how the 6809E can be used in an 8-bit multiprocessor system. A 6809 data acquisition system will also be presented.

Finally, for any *new* processor to be marketable the manufacturer *must* develop a broad *family* of devices which support the processor. To this end Motorola has developed a complete line of support devices for the 6800. All of these devices are also directly compatible with the 6809. In addition, several new devices are being developed specifically for the 6809. For example, the 6809 has been referred to as "a 500-horsepower engine with a three-gallon gas tank" because it is capable of addressing only a 64K memory space. However, Motorola has developed a 6829 Memory Management Unit (MMU) which expands the 6809 address space to 2 megabytes. This device, along with others, will be presented in this chapter. Furthermore, in an effort to support understanding of the 6809, Motorola has marketed the MEK6809D4 evaluation system. One version of the system, the D4A, is complete with power supply, hex keypad, and LED display module. Another version, the D4B, can be interfaced directly to a crt data terminal. Both systems contain an excellent ROM monitor (D4BUG) to provide efficient engineering evaluation or student learning of the 6809. A general description of the D4A will be provided in this chapter. Now, let's open up the unending world of 6809 applications.

OBJECTIVES

At the end of this chapter you will be able to do the following:

- Develop a minimum 6809 system.
- Expand the minimum system to provide serial data communication, parallel data communication and mass storage via a floppy disc system.
- Understand how parallel 16-bit data can be communicated to a 6809-based system.
- Design a 6809-based data acquisition system.
- Explain how a 6809 can be integrated into a multiprocessor system.
- Describe the new 6809 family devices.
- Understand the major features of the MEK6809D4 evaluation system.

A MINIMUM 6809 SYSTEM

A minimum 6809 microcomputer system would consist of read/write memory (R/W), read-only memory (ROM), and parallel interface device. Such a system is shown in Fig. 7-1. In this system we are using all 6800 family devices such that device compatibility is ensured. A 6810 is being used to provide 128 bytes of scratch-pad R/W memory. For many dedicated applications this is all that is

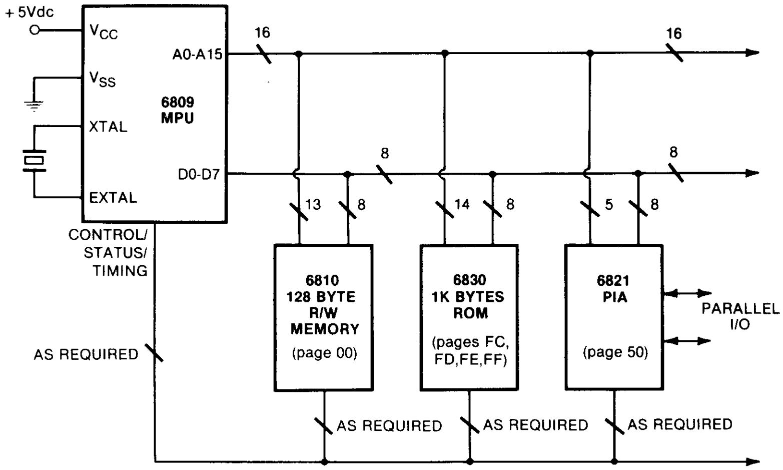


Fig. 7-1. Minimum 6809 microcomputer system.

required. The ROM function is being provided by a 1K 6830 chip. The 6830 is a mask-programmed ROM that will contain the interrupt vectors, interrupt service routines, and subprograms required by the specific application. Parallel interfacing to the 6809 is provided via a 6821 peripheral interface adapter (PIA). An associated memory map for this circuit is shown in Fig. 7-2. Note that we have assigned the 6810 R/W memory to page zero (addresses 0000–007F), the 6821 PIA to page 50 (addresses 5000–5003) and the 6830 ROM to pages FC, FD, FE, and FF (addresses FC00–FFFF). The only critical assignment here is for the ROM, since the 6809 interrupt vectors are located at addresses FFF0–FFFF.

The PIA has two 8-bit *channels* or *ports* that may be connected to peripheral devices. These ports can be programmed as either input or output ports. In fact, each bit within the port can be *separately* programmed for either input or output data transfers. Once the PIA is *initialized* by configuring the port lines, you will simply treat each port as if it were a separate memory location. Then data can be transferred between the 6809 and an i/o device (via the PIA) by using any of the 6809 load and store instructions. In addition, the two PIA ports can be configured for all input or all output and 16-bit data transferred by writing a small routine which would load or store byte-size data between the two PIA ports and one of the 6809 16-bit registers.

Now let's look at a couple of examples which involve 16-bit parallel data communication for our minimum 6809 system. A complete review of the PIA is provided in Appendix B, if needed.

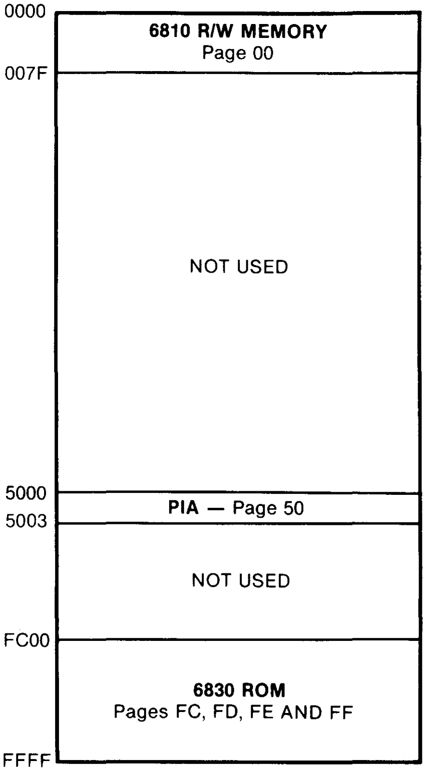


Fig. 7-2. Minimum 6809 system memory map.

Example 7-1: 16-Bit Data Input

Using the system in Fig. 7-1 and its associated memory map (Fig. 7-2), write a routine which will input data to accumulator D from a 16-bit input device.

Let's assume that the PIA has already been initialized to configure both ports (A and B) for input. Now, for 6800 systems you would *normally* connect RS0 (pin 36) of the PIA to address line A0, and RS1 (pin 35) to address line A1. This would result in the PIA memory map in Table 7-1. However, when interfacing the PIA to the 6809, if you reverse the above connections such that RS0 is connected to A1 and RS1 is connected to A0, then the PIA memory map in Table 7-2 will result. Thus DRA and DRB are assigned to consecutive memory locations, and any of the load or store instructions which are associated with the 6809 16-bit registers (LDD, STD, LDX, STX, etc.) can be used to transfer 16-bit data between the 6809 and PIA. With this in mind, the following program will accomplish the given task:

Table 7-1. PIA Memory Map for RS0 to A0 and RS1 to A1

Address	PIA Register Selected
5000	DDRA or DRA*
5001	CRA
5002	DDRB or DRB*
5003	CRB

*Depends on bit 2 of the control register.

```
LDA #
 50
TFR A,DPR
LDD $
 00
```

Note that we first store the PIA page number (50) into the direct page register. Then direct addressing can be used to input the 16-bit data directly into accumulator D. The port A data will form the most significant data byte and port B data will form the least significant byte.

Example 7-2: Data Input Synchronization

Now suppose that your PIA is connected as shown in Fig. 7-3. You want to synchronize the data input operation with the peripheral input device such that each time the peripheral device interrupts the 6809 via CA1 of the PIA, the 6809 will read a 16-bit data word and store it in a memory location specified by the X register. In addition, you want to provide complete handshaking with the input device. Therefore you must acknowledge the data read operation via CA2 of the PIA. Write a routine which will properly initialize the PIA, then read a block of 100_{10} (64_{16}) 16-bit data words using the synchronization handshake method described above.

The routine in Fig. 7-4 will accomplish the given task. As in the previous example it is assumed that the PIA is assigned to addresses 5000–5003 with RS0 connected to A1 and RS1 connected to A0. The routine first configures the PIA. The port A control register (CRA)

Table 7-2. PIA Memory Map for RS0 to A1 and RS1 to A0

Address	PIA Register Selected
5000	DDRA or DRA*
5001	DDRB or DRB*
5002	CRA
5003	CRB

*Depends on bit 2 of the control register.

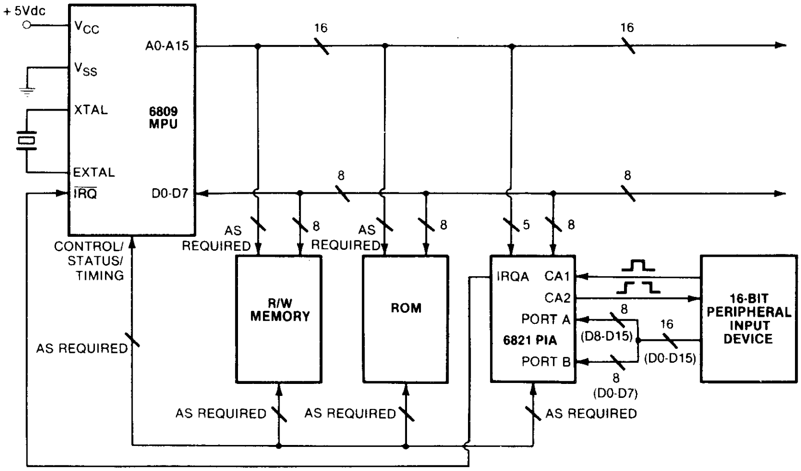


Fig. 7-3. Sixteen-bit input synchronization.

is configured such that an interrupt request ($\overline{\text{IRQ}}$) will be generated to the 6809 when CA1 is active high. Furthermore, CA2 will go high when the interrupt is received, then go back low when the data is read such that the handshake is complete. Once the PIA is initialized, the I flag of the condition code register is set and the 6809 is placed

Fig. 7-4. Routine for data input synchronization.

```
LDA #
 50          Load DPR with PIA
TFR A,DPR    page number
CLRA
CLRB
STD$         Select DDRA and DDRB
 02
STD$         Configure ports A and B
 00          for input
LDA #
 27          Configure CRA for
LDB #        input handshaking
 04          via CA1 and CA2.
STD$         Set CRB bit 2
 02
ORCC #       Set I flag
 10
SYNC         Syncing state
LDB #
 C8          Read 100₁₀ (64₁₆) 16-bit
LDY $        data words from PIA and
 00          store in 200₁₀ (C8₁₆)
STY,X + +    consecutive 8-bit
DECB         memory locations
BNE          specified by X register
 F4
ANDCC #      Clear syncing state
 00
```

in the syncing state by the SYNC instruction. Each time CA1 is active, the 6809 will read the 16-bit word into the Y register and then store the data in two consecutive memory locations specified by the X register. The routine will then branch back to the syncing state until another $\overline{\text{IRQ}}$ is received. The read cycle will repeat until 100_{10} (64_{16}) 16-bit words have been read and stored in 200_{10} ($C8_{16}$) consecutive 8-bit memory locations. Note that a complete handshake takes place between the 6809 and input device (via the PIA) each time a data word is transferred.

In the coming years you will be seeing many 6809s being used in multiprocessor *16-bit* systems. The 6809s will handle dedicated tasks, while a 16-bit microprocessor, such as the MC68000, will *supervise* the system activity. Thus 16-bit data transfers similar to the ones just shown will be quite common in such a system. The 16-bit peripheral input device in Example 7-2 could be a 16-bit processor.

AN EXPANDED 6809 SYSTEM

As you are now aware the 6809 is a very versatile, high-performance microprocessor. Because of its software and hardware efficiency it can be easily expanded in ways that first- and second-generation devices (such as the 6800, 8080, etc.) could not, without sacrificing performance. Such things as time sharing, high-level language interpretation (Pascal, BASIC, FORTRAN, COBOL) can be performed efficiently.

An expanded 6809 system is shown in Fig. 7-5. Here, most functions of a microcomputer are provided. Note the use of the various 6800 family devices. Direct memory access is provided for the 6843 floppy disc controller (FDC) via the 6844 direct memory access controller (DMAC). Serial communication for a crt, modem, and printer is provided via the 6850 asynchronous communications interface adapter (ACIA). Parallel i/o is provided via the 6821 PIAs. A system such as this can be *stand-alone* or easily interfaced with a 16-bit 68000 system via the PIA interfaces as previously discussed. In fact, several 6809 systems such as this could interface to a 68000 system—a topic for another book. The expansion and application possibilities are almost endless and only limited by the imagination.

MULTIPROCESSOR SYSTEMS

Multiprocessor systems will be the wave of the future. Many dedicated system functions can be handled by separate processors, each a complete system in its own right. The separate processor systems will contain *dedicated* R/W memory, ROM, and peripheral devices.

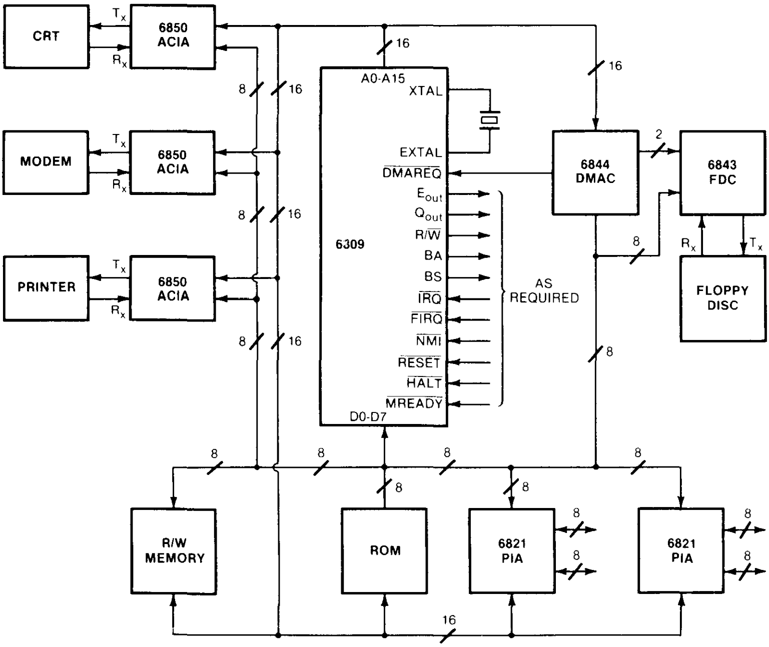


Fig. 7-5. An expanded 6809 microcomputer system.

However, each dedicated system will share a common address/data bus structure and *feed* into a large system containing *global* memory and peripherals that are shared by all the system processors. In many cases the total system operation will be *supervised* by a 16-bit processor, such as the MC68000. The *system processor* will coordinate the dedicated processor activities in much the same way that an engineering supervisor or project director coordinates the activities of others to get the job done in the most efficient way possible.

As discussed earlier, the 6809E has been specially designed for multiprocessor system applications. As you know, the 6809E has extra status lines (LIC and BUSY) which are particularly suited for multiprocessor systems. A simple multiprocessor system using two 6809Es is shown in Fig. 7-6. Obviously, many of the interfacing details are left out in this diagram and the actual system diagram would be much more complicated. Fig. 7-6, however, should give you the basic idea of a multiprocessor system.

REMOTE DATA ACQUISITION

Another common use of a processor like the 6809 is for automated data acquisition. A typical 6809-based data acquisition system is

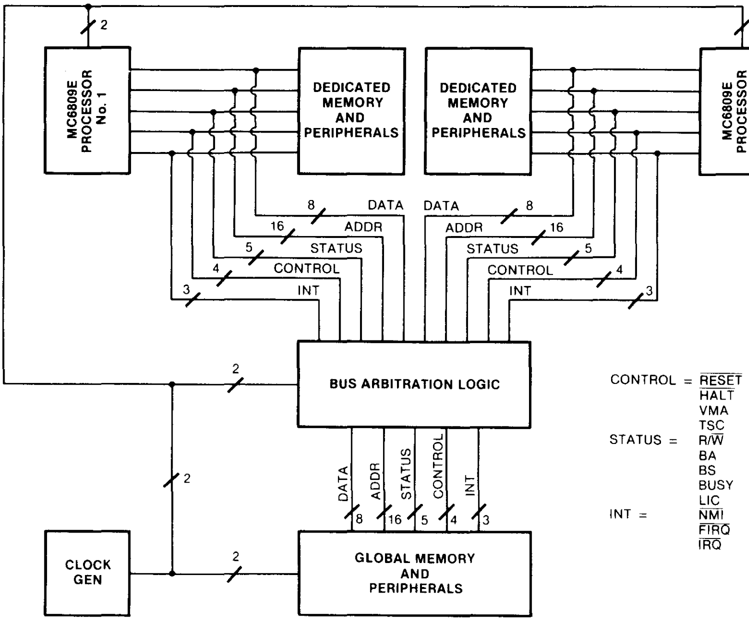


Fig. 7-6. 6809E multiprocessor system.

shown in Fig. 7-7. A system such as this could be stand-alone or part of a larger data acquisition and analysis system. Many manufacturing processes use some form of automated data acquisition to obtain information about a product and its associated production process. Individual product data are analyzed by a *local* system such as the one in Fig. 7-7 for immediate product disposition. In addition, a simple ongoing statistical analysis can be performed at the local system for a given production run to provide the engineering staff with up-to-the-minute product and process-related information. Current information such as this can keep a process from getting out of control, especially in high-volume production runs. A local system can also be made to control process variables to automatically compensate for changing product characteristics. In many cases, several local systems will "feed" into a larger multiprocessor system which will provide further engineering analysis, reporting, and permanent data storage. For example, several of these 6809-based systems could feed a larger 68000-based system.

Now let's take a closer look at the system shown in Fig. 7-7. First, sufficient R/W memory is supplied for scratch-pad calculations and temporary data storage. A ROM monitor will provide the routine system operating software required for data analysis and communi-

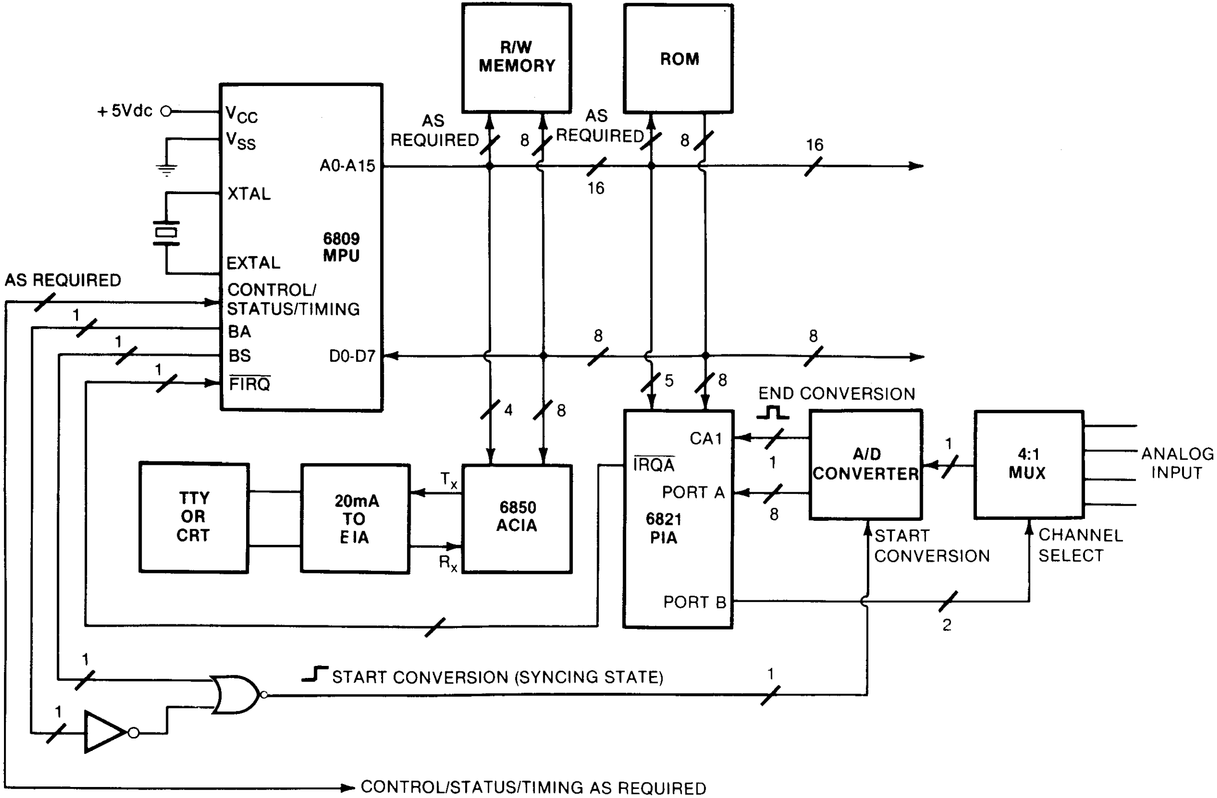


Fig. 7-7. Remote data acquisition system.

cations. A high-level language interpreter could also be provided in the local system ROM to enhance local engineering data analysis. Operator communication is provided via a teletypewriter (tty) and/or crt data terminal. Interfacing these devices to the 6809 would require a 6850 ACIA for serial/parallel translations and external logic to establish the required RS-232C (EIA) or 20-mA current-loop serial communication formats. Four analog inputs are multiplexed with a 4:1 multiplexer circuit. Channel selection can be accomplished by using any two of the PIA port B data lines as output lines. Any one of the four analog inputs may then be selected by writing the proper 1s and 0s to the respective port B data register (DRB) bits. The A/D converter will convert the selected analog signal to digital. The digital information is then fed to port A of the PIA. A syncing routine will be used to control the data transfer. After the PIA is properly initialized, you will insert a SYNC instruction in your program. When the 6809 executes the SYNC, BA=1 and BS=0 to indicate the *syncing state*. The external digital logic will decode BA/BS to provide a low-to-high transition to the A/D converter. This transition will signal the conversion process to begin. Once the analog signal has been converted to a digital byte of information, the A/D converter will activate CA1 on the PIA to provide an active $\overline{\text{FIRQ}}$ for the 6809. If the F flag has been previously set, the 6809 will clear the syncing state and execute the next sequential program instruction, which will read the port A data. Then, a branch back to the SYNC instruction will cause the process to be repeated for the next data byte. Thus a complete handshake is provided between the 6809 and A/D converter for each data byte transferred. All the software details should now be familiar to you and you should now have no trouble writing a routine which will properly configure the PIA and acquire the data in the manner just described.

THE 6809 FAMILY

In addition to the peripheral devices that have already been discussed, there are many more devices which support the 6809. As stated earlier, the 6809 is directly compatible with the existing 6800 family. A listing of existing 6800 family peripheral control devices which will support the 6809 is provided in Table 7-3. Any of these devices can be used with the 6809 in much the same way that they are used with the 6800. Consult *The Complete Motorola Microcomputer Data Library,* Motorola Semiconductor Products, Inc., Box 20912, Phoenix, Arizona 85036, for specifications and interfacing details of the devices listed in Table 7-3. Also, the *M6800 Applications Manual,* available from Motorola, might be helpful along with other 6800 family application notes and texts.

Table 7-3. 6800 Family Peripherals That Are Directly Compatible With the 6809

Part Number	Device
6821	Peripheral interface adapter
6828	Priority interrupt controller
6840	Programmable timer module
6843	Floppy disc controller
6844	DMA controller
6845	Crt controller
6846	ROM i/o timer
6847	Video display generator
6850	Asynchronous communication interface adapter
6852	Synchronous serial data adapter
6854	Advanced data link controller
6859	Data security device
6860	0- to 600-bps digital modem
6862	2400-bps modulator
68488	General-purpose interface adapter
68120	Intelligent peripheral controller
68540	Error detection and correction circuit

At the time of this printing, Motorola is in the process of developing additional peripheral devices to support the 6809. These devices include a memory management unit (MC6829), floating-point ROM (MC6839) and serial DMA processor (MC6842). The 6829 memory management unit (MMU) permits you to expand the 64K address space of the 6809 to a maximum of 2 megabytes as shown in Fig. 7-8. The 6829 MMU combines the 11 lower 6809 address lines (A0–A10) with ten address lines (PA11–PA20) of its own to form the 2-megabyte address space. This space consists of 1024 pages of 2K bytes each. The 6829 address lines (PA11–PA20) specify the page number, and the lower 11 6809 address lines (A0–A10) specify the particular byte within the page. An internal mapping R/W memory uses the upper five 6809 address lines (A11–A15) along with the contents of an internal 5-bit task register to form the ten most signifi-

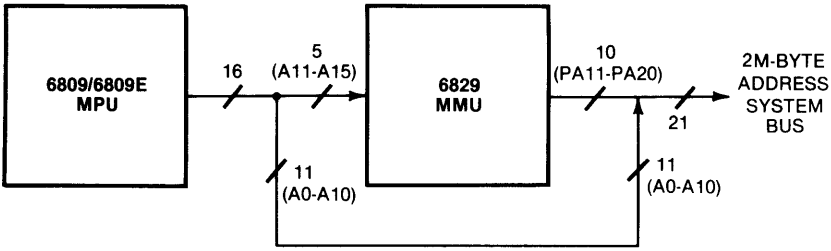


Fig. 7-8. Expanding from 65K to 2M bytes using the 6829 MMU.

cant address lines (PA11–PA20) of the 2-megabyte address space. Up to eight 6829s can be used in a system.

The 6839 floating-point ROM will facilitate the use of high-level languages, such as Pascal, BASIC, and FORTRAN, on 6809-based systems. The 6839 includes such floating-point operations as add, subtract, multiply, divide, square root, absolute value, negate, and routines which provide binary/decimal conversions and allow you to convert between integer and floating-point values.

Finally, the 6842 serial direct memory access (SDMA) processor provides direct memory access for high-speed *serial* data links. Serial data transfer rates of up to 4 megabits per second are possible. Many future multiprocessor systems will be relying on a serial data link for communication between processors. The SDMA can be used for this purpose. The 6842 SDMA is a serial device, while the 6844 DMAC, which was previously discussed, is a parallel DMA device.

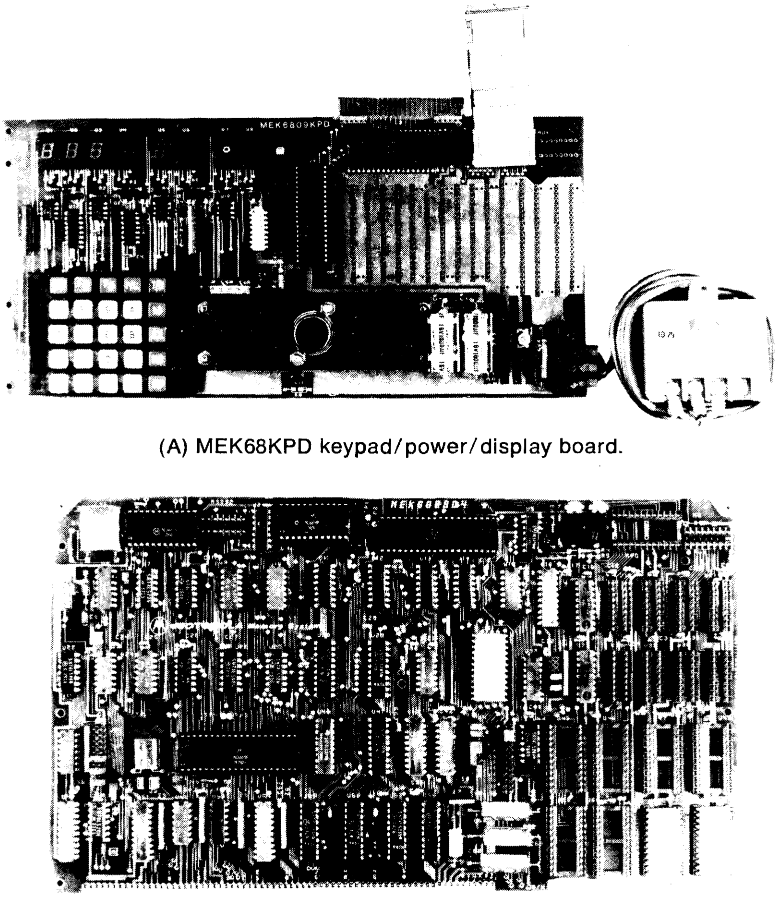
A brief product description of each of these new 6809 family devices is provided in Appendix C. Consult the respective device data sheets, obtainable from Motorola Semiconductor Products, Inc., for additional device information and interfacing details.

THE MEK6809D4 MICROCOMPUTER EVALUATION SYSTEM

In an effort to support understanding of the 6809 and provide an engineering evaluation tool, Motorola has marketed a relatively-low-cost evaluation system built around the 6809. The MEK6809D4 system is pictured in Fig. 7-9. In this section we intend on providing a general description of the *D4* system. In addition, some helpful hints and examples will be provided to enhance your understanding of the system operation. The following discussion will complement the system instruction manual. Refer to the manual for additional information and system details.

There are two different MEK6809D4 models available: the D4A and D4B. The D4A evaluation system consists of two boards: an MEK6809D4 microcomputer board and an MEK68KPD keypad/power/display board. In addition, a wall-outlet type transformer is supplied with the system such that no external power supply is required for the *minimum* system configuration. The D4B model is a single-board system and is intended for use with an RS-232C serial terminal. All the associated RS-232C interface circuitry is provided with the system. The user, however, must supply +12 V, +5 V and −12 V power from an external source. We will confine our discussion here to the self-contained D4A system.

A functional layout of each D4A board, with numbered references, is provided in Figs. 7-10 and 7-11. An explanation of each numbered reference follows.



(A) MEK68KPD keypad/power/display board.

(B) MEK6809D4 microcomputer board.

Courtesy Motorola Semiconductor Products, Inc.

Fig. 7-9. MEK6809D4A microcomputer evaluation system.

MEK68KPD Keyboard/Power/Display Board (Fig. 7-10)

1. *Hex keyboard*—This allows the operator to enter data in hexadecimal (white keys) and system commands (blue keys). When a key is depressed, a non-maskable interrupt ($\overline{NMI}$) is generated to the 6809. The $\overline{NMI}$ service routine then searches for (decodes) the closed key. All of the keys are single function. A discussion of each key function will be given shortly.
2. *Seven-segment LED display*—This is an output display which consists of 8 seven-segment LED displays. The system will dis-

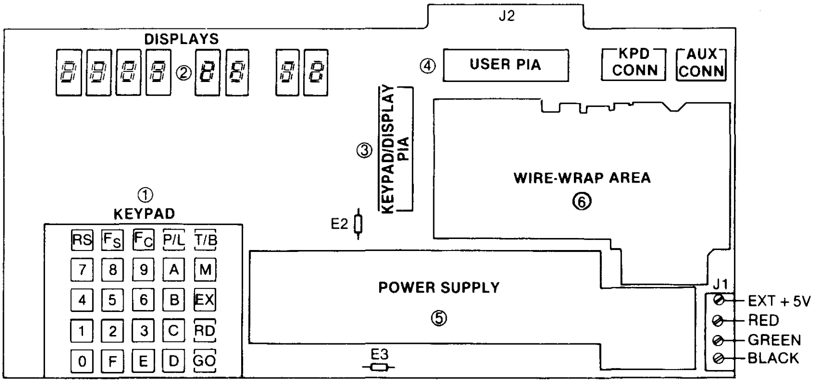


Fig. 7-10. MEK68KPD keyboard /power/display board.

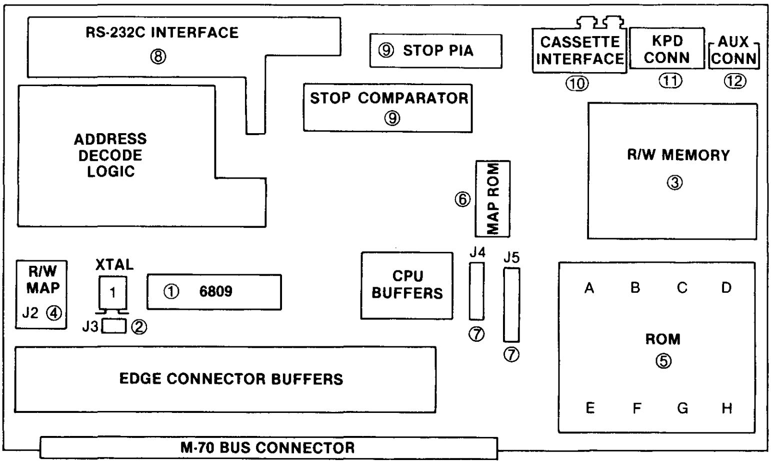


Fig. 7-11. MEK6809D4 microcomputer board.

play internal register, external memory, and address information in addition to various user prompts to facilitate the use of the system. The 8 displays use a multiplexing scheme to display messages and multiple characters.

3. *Keypad/display PIA*—This PIA is dedicated to the system and serves two functions: keypad decoding and character display. When a key is depressed, an interrupt is generated to the 6809. Port B of the PIA then scans the keypad to determine in which column the depressed key is located. Keypad data is input via port A. In addition, for the display function the LED character designations are output via port A and the displays are sequentially enabled (multiplexed) by port B at a rapid rate to pre-

vent display flickering. The keypad/display PIA is assigned to addresses E0F8–E0FB as follows:

DRA/DDRA	E0F8
CRA	E0F9
DRB/DDRB	E0FA
CRB	E0FB

4. *User PIA*—This PIA is entirely available for user configuration and use. All of the port A and port B data lines, along with their associated control lines (CA1, CA2, CB1, CB2), are brought out to the adjacent J2 edge connector. See the system user's manual for the proper connector pin assignments. The user PIA is assigned to addresses E0FC–E0FF as follows:

DRA/DDRA	E0FC
CRA	E0FD
DRB/DDRB	E0FE
CRB	E0FF

5. *Power supply*—The MEK68KPD contains two +5-Vdc power supplies. One supply is used to drive the 8 LED displays via regulator VR1 and the other used to supply power for the D4A logic via regulator VR2. An external 18-V center-tap transformer is supplied with the system and is to be connected as shown in Fig. 7-10.
   CAUTION: The on-board supply is designed to provide only enough power for the *minimum* system configuration. If additional devices are added, an external supply is required. To connect an external supply you must first remove the E2 and E3 connecting links. This will prevent damage to the on-board supply regulators. Then, connect the external TTL quality +5-Vdc source to the J1 connector as shown in Fig. 7-10.
6. *Wire-wrap area*—This area is provided for you to wire-wrap your own external devices to the system. Both ground and +5-Vdc supply buses are provided in this region for convenience. The 6809 data, address, timing, status, and control signals are available at the KPD and AUX connectors on the microcomputer module board. See the system user's manual for the proper connector pin assignments.

MEK6809D4 Microcomputer Board (Fig. 7-11)

1. *6809 MPU and crystal*—This is the microprocessor "heart" of the system together with its 4× crystal, which establishes the internal clock frequency.
2. *Clock select*—The jumper connector (J3) provides for selection of either internal or external clock control. See the system user's manual for the proper jumper connections.

3. *R/W memory*—Sockets are provided for 5K bytes of R/W memory via five pairs of 2114 (1K×4) or equivalent R/W memory chips. The minimum system includes two 2114s (1K bytes) to provide for storing variables and to support the system stack. These two chips are assigned to addresses E400–E7FF. Any additional user R/W memory address assignments are determined by the R/W memory map connector.
4. *R/W memory map*—This is a jumper connector (J2) which allows mapping of the additional user R/W memory in any one of 16 possible 4K-byte blocks within the 64K-byte memory space. See the system user's manual for the proper jumper connections.
5. *ROM*—Eight ROM/EPROM sockets, A through H, are provided in this area for up to 48K bytes of ROM or EPROM. The minimum system includes one MCM68332 ROM (4K×8), which provides the system monitor called D4BUG. Any additional ROM address assignments are determined by the mapping *ROM* and *ROM type* connector. ROM sockets A through D will normally be used for larger ROMs (4K–8K) containing editor/assembler type programs. ROM sockets E and F can be used for smaller (1K, 2K, or 4K) user ROMs/EPROMs of the single *or* triple supply variety. ROM socket G is reserved for a special 2K ROM (R2-RS-232) required for the RS-232C interface which is available on the D4B version of the MEK6809D4 system. ROM socket H is reserved for the 4K D4BUG ROM on both the D4A and D4B system versions.
6. *Mapping ROM*—The mapping ROM is provided with both the D4A and D4B systems. This ROM provides you with a unique and flexible means of mapping the eight ROM devices into the 64K memory space. See the system user's manual for the ROM mapping details.
7. *ROM type connectors (J4 and J5)*—These are jumper connectors which allow you to select any one of the following ROM type options for ROM sockets A through G:

   ROM/EPROM Sockets A through D
   2K×8 single supply (MCM2716, TMS2616, MCM68A316E)
   4K×8 single supply (MCM25A32, TMS2532, MCM68A332)
   8K×8 single supply (MCM68A764, MCM68A364)

   ROM/EPROM Sockets E through H
   1K×8 triple supply (MCM2708, TMS2708)
   2K×8 triple supply (TMS2716)
   2K×8 single supply (MCM2716, TMS2516, MCM68A316E)
   4K×8 single supply (MCM25A32, TMS2532, MCM68A332)

Connector J5 configures sockets A through F, and the J4 connector configures sockets G and H. Note the extreme flexibility that you have in the selection of the type of ROM to be used in your system. See the system user's manual for the proper jumper connections for your ROM configuration.

8. *RS-232C interface*—With the D4B system this area provides the required RS-232C interfacing devices (ACIA, baud rate generator, etc.). The sockets in this area will be omitted or empty with the D4A system.
9. *Stop PIA and comparator*—The D4 system provides you with a *stop-on-address* capability via a PIA and associated comparator logic. The purpose of a *stop address* is to enable a user program to be executed until a certain predetermined address is reached. The stop address is stored in the stop address PIA ports A and B. When the address on the address bus is the same as the stop address, the comparator generates a nonmaskable ($\overline{\text{NMI}}$) interrupt via CA1 of the PIA to stop the program execution. The stop PIA is automatically initialized as part of the system reset routine.
10. *Cassette interface*—This will provide you with a means to interface an inexpensive cassette recorder to the system. The D4BUG monitor provides for the proper data formatting and recovery functions via the P/L and FS keys on the keypad. Data is stored and recovered by D4BUG software in a Kansas City Standard, 300- or 1200-bps data format.
11. *KPD connector*—This 24-pin connector interfaces the microcomputer board to the MEK68KPD board. The 6809 data and selected address/control lines are provided via this connector. All the signals supplied to this connector are buffered. See the system user's manual for the respective pin assignments.
12. *AUX connector*—This 16-pin connector is provided to allow you to extend *all* of the 6809 address and control signals to the MEK68KPD, for use in the wire-wrap area. All the signals supplied to this connector are buffered. See the system user's manual for the respective pin assignments.

Finally, the key functions of the MEK68KPD keypad are summarized in Fig. 7-12. The control keys will allow you to do the following:

- Reset the system.
- Insert and delete breakpoints in your program.
- Display and alter the internal 6809 register contents.
- Examine and alter the contents of any R/W memory location.
- Single-step a program.
- Calculate both 8- and 16-bit relative address offsets.

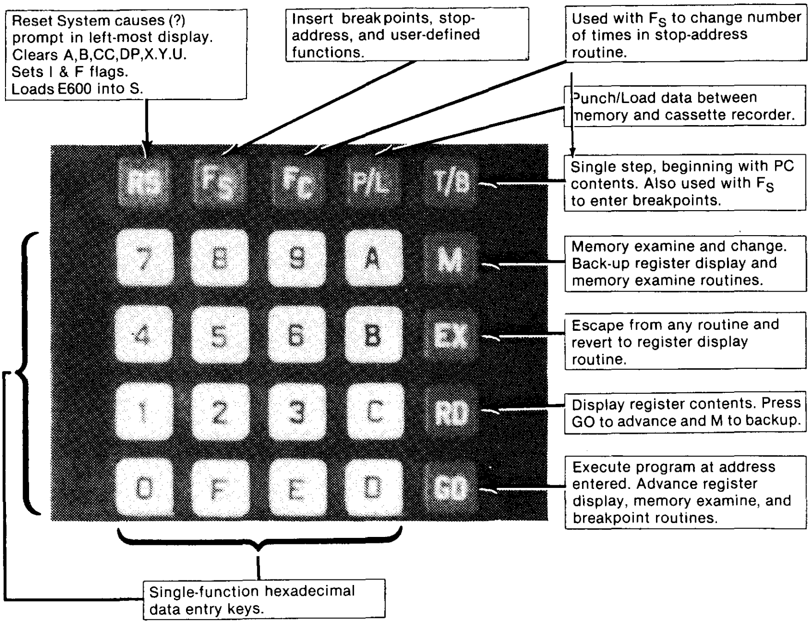

Reset System causes (?) prompt in left-most display. Clears A,B,CC,DP,X.Y.U. Sets I & F flags. Loads E600 into S.
Insert breakpoints, stop-address, and user-defined functions.
Used with F_S to change number of times in stop-address routine.
Punch/Load data between memory and cassette recorder.
Single step, beginning with PC contents. Also used with F_S to enter breakpoints.
Memory examine and change. Back-up register display and memory examine routines.
Escape from any routine and revert to register display routine.
Display register contents. Press GO to advance and M to backup.
Execute program at address entered. Advance register display, memory examine, and breakpoint routines.
Single-function hexadecimal data entry keys.
RS
Fs
Fc
P/L
T/B
7
8
9
A
M
4
5
6
B
EX
1
2
3
C
RD
0
F
E
D
GO

Fig. 7-12. D4A key functions.

- Punch (record) a cassette tape with a program in memory.
- Load (read) a cassette tape into memory.
- Provide for R/W memory and ROM hardware paging.
- Escape from a system routine without executing a reset operation.
- Select a program *stop address* and the number of times that this address is executed before the program stops.
- Define up to 16 special user functions.

Refer to the system user's manual for a detailed explanation of how to accomplish the above tasks. As you can see, the D4A is a very versatile and powerful system. The following hints might be helpful when you first begin to work with your D4A:

1. User memory for the minimum system is located at addresses E400–E7FF.
2. The reset operation will *always clear* the program counter (PC), accumulator A, accumulator B, X register, Y register, U register, and direct page register (DPR). In addition, the F and I flags are automatically set with all other flags cleared (CCR= 50_{16}) and the hardware stack pointer (S register) is loaded with the value E600. The reset operation will also erase any breakpoints which have inserted into the user program.

3. After "GO," depressing "EX" will automatically enter the register display routine without altering any of the internal register contents. Then, pressing "GO" will advance the register display and pressing "M" will back up the display routine.
4. The system stack begins at address E600. If the S register is loaded with a non-R/W memory address value *or* there is not enough R/W memory available to stack the internal register contents, the display will indicate "Bad SP??"
5. The system uses SWI1 for the breakpoint routine.
6. To use SWI2, store the SWI2 interrupt service routine vector at address E777:E778.
7. To use SWI3, store its interrupt service routine vector at address E775:E776.
8. The single-step function (T/B) sets the E flag.

The following will help familiarize you with your D4A and illustrate the versatility of both the 6809 and D4A system.

Example 7-3: Using the SWI2 Interrupt

Step 1: Load the program in Fig. 7-13 into the D4 system beginning at address E400. Refer to the D4 user's manual for the program loading procedure.

NOTE: *Do not* press RS (reset) after entering the program, use the escape (EX) key.

Step 2: Depress RD and set the program counter to the beginning address of the above program (E400).

Step 3: Depress T/B. The first instruction was executed and the D4

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	OPERATION REMARKS
E400	C6	LDB #	Main Program
E401	BB	BB	
E402	86	LDA #	
E403	AA	AA	
E404	1F	TFR A,CC	
E405	8B	8B	
E406	10	SWI2	Software Interrupt (SWI2)
E407	3F		
E408	3C	CWAI #	
E409	FF	FF	
•			
•			
E500	CE	LDU #	SWI2 Interrupt Service Routine
E501	CC	CC	
E502	CC	CC	
E503	3B	RTI	
•			
•			
E777	E5	E5	Our SWI2 Interrupt Vector
E778	00	00	

Fig. 7-13. Program for Example 7-13.

is now in the register display routine. The program counter contents should be the address of the next instruction to be executed (E402).

Step 4: Verify that accumulator B was loaded with the value BB by depressing the GO key twice. Also, at this time, you might want to note the contents of the other registers by successively pushing the GO key.

Step 5: Depress T/B a second time. Verify the proper PC contents (E404) and that accumulator A was loaded with the value AA.

Step 6: Single step the program a third time and verify that the accumulator A value of AA was transferred to the direct page register (DP).

Step 7: Single step the program again and observe the PC contents of FDBC. Why? Because the software interrupt (SWI2) was just executed and the processor was vectored to this address by the SWI2 software interrupt vector located at address FFF4:FFF5. Escape (EX) from the register display routine and verify that the SWI2 vector, located at address FFF4:FFF5, is FDBC.

Step 8: The SWI2 instruction also caused the internal register data to be stacked. The S stack pointer was originally set to E600 by the reset operation. Thus, examine memory locations E600 through E5F4 and verify the correct register stacking and the proper stacking order. Note also that the S register now contains E5F4. You should find the registers stacked in the following manner:

S-12 →	E5F4	CCR
S-11 →	E5F5	A
S-10 →	E5F6	B
S-9 →	E5F7	DPR
S-8 →	E5F8	X_H
S-7 →	E5F9	X_L
S-6 →	E5FA	Y_H
S-5 →	E5FB	Y_L
S-4 →	E5FC	U_H
S-3 →	E5FD	U_L
S-2 →	E5FE	PC_H
S-1 →	E5FF	PC_L
S →	E600	

Step 9: Now, return to the register display routine by depressing EX, then RD. The PC should still contain FDBC, which is in ROM. Continue to single step through this ROM routine until the PC contains address E500. Note that E500 is *our*

SWI2 interrupt vector. You might say that the ROM routine converted the *system* SWI2 interrupt vector (FDBC) located at FFF4:FFF5 to *our* SWI2 interrupt vector (E500) located at address E777:E778. If you want a real challenge and learning experience, interpret the ROM routine instructions which begin at address FDBC to understand how the above vector conversion is accomplished.

Step 10: With the PC set at E500, single step the program and verify that the U register was loaded with the value CCCC by *our* SWI2 interrupt service routine.

Step 11: Single step the program again. This time, the RTI instruction was executed and therefore control was returned back to the main program. Verify that the original register data has been restored as a result of the unstacking operation caused by the RTI instruction.

Step 12: You might want to rewrite the program to use the SWI3 interrupt. To do this, you must insert the SWI3 op code at address E406:E407 and an interrupt vector of your choosing at address E775:E776. Remember to also include an interrupt service routine.

Example 7-4: Buried Op Code and Computed Vector Fetch

Step 1: Load the program in Fig. 7-14 into the D4 system beginning at address E400.

Step 2: The program is from Example 5-5. Turn back to this example and review the program explanation.

Step 3: The vector table in the program of Fig. 7-14 begins at address FFF0, which is the beginning of the 6809 vector table. Thus, we will be using a control byte to determine which of the 6809 vectors will be accessed by the program. Because

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	OPERATION REMARKS
E400	86	LDA #	Load control byte
E401	Control Byte	Control Byte	
E402	8E	LDX #	Load beginning address of vector table
E403	FF	FF	
E404	F0	F0	
E405	5F	CLRB	Vector to table determined by control byte
E406	10	LBRN	
E407	21		
E408	CB	ADDB #	
E409	02	02	
R40A	44	LSRA	
R40B	24	BCC	
R40C	FB	FB	
R40D	6E	JMP[B,X]	
E40E	95	95	

Fig. 7-14. Program for Example 7-14.

of the fixed 6809 vector table mapping, the control byte will access the 6809 vectors in the following manner:

Control Byte	*Vector Accessed*
00	Reserved
02	SWI3
04	SWI2
08	$\overline{\text{FIRQ}}$
10	$\overline{\text{IRQ}}$
20	SWI1
40	$\overline{\text{NMI}}$
80	$\overline{\text{RESET}}$

Step 4: Insert 80 at address E401 for the control byte. This should access the $\overline{\text{RESET}}$ interrupt vector.

Step 5: Execute the program. A question mark(?) should appear in the left-most display as the result of the accessed reset operation.

Step 6: You might want to access the other interrupt vectors by changing the control byte. In all cases you will end up in the respective interrupt service routine located in ROM. The display will either remain blank or revert to the register display routine, depending on the particular interrupt which is being accessed .

Step 7: Now, create your own vector table in R/W memory as in Example 5-5. Remember to also include some type of vector service routine such that you can verify proper vectoring. Try it! In addition, go back and execute the text examples on your D4. In this way you will gain a better understanding of both the 6809 and the D4 system.

APPENDIX A

6809/6809E Instruction Set

The following pages contain detailed definitions of the 59 executable instructions. These pages are provided through the courtesy of Motorola Semiconductor Products, Inc., Austin, Texas.

A-1. Nomenclature

Operation Notation

← =	is transferred to
∧ =	Boolean AND
∨ =	Boolean OR
⊕ =	Boolean exclusive OR
‾ =	(overline) = Boolean NOT
: =	concatenation

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 register
CCR = CC =	condition code register
PC =	program counter
R =	a register before the operation: A, B, 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 registers, i.e., A, B, D, X, Y, U, S, PC, DP, and CC
ZZ =	a pointer register, i.e., 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

A-2. Definitions of Executable Instructions

Add ACCB into IX **ABX**

Source Form: ABX

Operation: IX′ ← IX + ACCB

Condition Codes: Not affected.

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

Addressing Mode: Inherent

Add with Carry Memory into Register **ADC**

Source Forms: ADCA P; ADCB P

Operation: R′ ← R + M + C

Condition Codes: H: Set IFF the operation caused a carry from bit 3 in the alu.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF the operation caused an 8-bit twos complement arithmetic overflow.
C: Set IFF 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

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Add Memory into Register—8 Bits **ADD**

Source Forms: ADDA P; ADDB P

Operation: R′ ← R + M

Condition Codes: H: Set IFF the operation caused a carry from bit 3 in the alu.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF the operation caused an 8-bit twos complement arithmetic overflow.
C: Set IFF 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

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Add Memory into Register—16 Bits **ADD**

Source Form: ADDD P

Operation: R′ ← R + M:M+1

Condition Codes: H: Not affected.
N: Set IFF bit 15 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF there was a 16-bit twos complement arithmetic overflow.
C: Set IFF the operation on the MS byte caused a carry from bit 7 in the alu.

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

Register Addressing Mode: Double accumulator

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Logical AND Memory into Register **AND**

Source Forms: ANDA P; ANDB P

Operation: R′ ← R ∧ M

Condition Codes: H: Not affected.
N: Set IFF bit 7 of result is set.
Z: Set IFF 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

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Logical AND Immediate Memory into CCR **AND**

Source Form: ANDCC #XX

Operation: R′ ← R ∧ MI

Condition Codes: CCR′ ← CCR ∧ MI

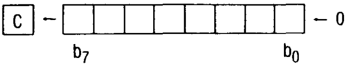
Description: Performs a logical AND between the CCR and the MI byte and places the result in the CCR.

Register Addressing Modes: CCR

Memory Addressing Mode: Memory immediate

Arithmetic Shift Left **ASL**

Source Form: ASL Q



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

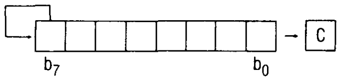
Condition Codes:
H: Undefined.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Loaded with the result of $(b_7 \oplus b_6)$ of the original operand.
C: Loaded with bit 7 of the original operand.

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.

Addressing Modes:
Accumulator
Direct
Indexed
Extended

Arithmetic Shift Right **ASR**

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 IFF bit 7 of the result is set.
Z: Set IFF 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.

Addressing Modes:
Accumulator
Direct
Indexed
Extended

Branch on Carry Clear **BCC**

Source Forms: BCC dd; LBCC DDDD

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

Condition Codes: Not affected.

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

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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

Branch on Carry Set **BCS**

Source Forms: BCS dd; LBCS DDDD

Operation: TEMP ← MI
IFF 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 Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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."

Branch on Equal **BEQ**

Source Forms: BEQ dd; LBEQ DDDD

Operation: TEMP ← MI
IFF 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.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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

Branch on Greater Than or Equal to Zero **BGE**

Source Forms: BGE dd; LBGE DDDD

Operation: TEMP ← MI
IFF $[N \oplus V] = 0$ then PC′ ← PC + TEMP

Condition Codes: Not affected.

Description: Causes a branch if N and V are either both set or both clear, i.e., branch if the sign of a *valid* twos complement result is—or would be—positive.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Comments: Used after a subtract or compare operation on twos complement values, this instruction will "branch if the register was greater than or equal to the memory operand."

Branch on Greater **BGT**

Source Forms: BGT dd; LBGT DDDD

Operation: TEMP ← MI
IFF $Z \vee [N \oplus V] = 0$ 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* twos complement result is—or would be—positive and nonzero.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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

Branch if Higher **BHI**

Source Forms: BHI dd; LBHI DDDD

Operation: TEMP ← MI
IFF $[C \vee Z] = 0$ then PC′ ← PC + TEMP

Condition Codes: Not affected.

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

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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.

Branch if Higher or Same

BHS

Source Form: BHS dd; LBHS DDDD

Operation: $\text{TEMP} \leftarrow \text{MI}$
$\text{IFF } C = 0 \text{ then } \text{PC}' \leftarrow \text{PC} + \text{MI}$

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

Effective Addressing Modes: Relative
Long relative

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.

Bit Test

BIT

Source Form: BIT P

Operation: $\text{TEMP} \leftarrow R \land M$

Condition Codes: H: Not affected.
N: Set IFF bit 7 of the result is set.
Z: Set IFF 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

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Branch on Less Than or Equal to Zero

BLE

Source Form: BLE dd; LBLE DDDD

Operation: $\text{TFMP} \leftarrow \text{MI}$
$\text{IFF } Z \lor [N \oplus V] = 1 \text{ then } \text{PC}' \leftarrow \text{PC} + \text{TEMP}$

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* twos complement result is—or would be—negative.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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

Branch on Lower **BLO**

Source Form: BLO dd; LBLO DDDD

Operation: TEMP ← MI
IFF 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 Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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.

Branch on Lower or Same **BLS**

Source Form: BLS dd; LBLS DDDD

Operation: TEMP ← MI
IFF (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.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

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.

Branch on Less Than Zero **BLT**

Source Forms: BLT dd; LBLT DDDD

Operation: TEMP ← MI
IFF [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* twos complement result is—or would be—negative.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Comments: Used after a subtract or compare operation on twos complement binary values, this instruction will "branch if the register was less than the memory operand."

Branch on Minus **BMI**

Source Form: BMI dd; LBMI DDDD

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

Condition Codes: Not affected.

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

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Comments: Used after an operation on twos complement binary values, this instruction will "branch if the (possibly invalid) result is minus."

Branch if Not Equal **BNE**

Source Forms: BNE dd; LBNE DDDD

Operation: TEMP ← MI
IFF 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

Effective Addressing Modes: Relative
Long relative

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."

Branch on Plus **BPL**

Source Form: BPL dd; LBPL DDDD

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

Condition Codes: Not affected.

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

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Comments: Used after an operation on twos complement binary values, this instruction will "branch if the possibly invalid result is positive."

Branch Always **BRA**

Source Forms: BRA dd; LBRA DDDD

Operation: TEMP ← MI
PC′ ← PC + TEMP

Condition Codes: Not affected.

Description: Causes an unconditional branch.

Memory Assessing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Branch Never **BRN**

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.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Branch to Subroutine **BSR**

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.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Branch on Overflow Clear **BVC**

Source Form: BVC dd; LBVC DDDD

Operation: TEMP ← MI
IFF 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 twos complement result was valid.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Comments: Used after an operation on twos complement binary values, this instruction will "branch if there was no overflow."

Branch on Overflow Set **BVS**

Source Form: BVS dd; LBVS DDDD

Operation: TEMP ← MI
IFF 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 twos complement result was invalid.

Memory Addressing Mode: Memory immediate

Effective Addressing Modes: Relative
Long relative

Comments: Used after an operation on twos 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.

Clear **CLR**

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.

Addressing Modes: Accumulator
Direct
Indexed
Extended

Compare Memory from a Register—8 Bits **CMP**

Source Form: CMPA P; CMPB P

Operation: TEMP ← R − M [i.e., TEMP ← R + $\overline{\text{M}}$ + 1]

Condition Codes: H: Undefined.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF the operation caused an 8-bit twos complement overflow.
C: Set IFF 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

Memory Addressing: Immediate
Direct
Indexed
Extended

Flag Results: (N ⊕ V) = 1 R .LT. M (twos complement)
C = 1 R .LO. M (unsigned)
Z = 1 R .EQ. M

Compare Memory From a Register—16 Bits **CMP**

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

Operation: TEMP ← R − M:M+1 [i.e., TEMP ← R + $\overline{\text{M:M+1}}$ +1]

Condition Codes: H: Unaffected.
N: Set IFF bit 15 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF the operation caused a 16-bit twos complement overflow.
C: Set IFF 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)

Memory Addressing: Immediate
Direct
Indexed
Extended

Flag Results: (N ⊕ V) = 1 R .LT. M (twos complement)
C = 1 R .LO. M (unsigned)
Z = 1 R .EQ. M

Complement **COM**

Source Form: COM Q

Operation: $M' \leftarrow 0 + \overline{M}$

Condition Codes: H: Not affected.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Cleared.
C: Set.

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

Memory Addressing Modes: Accumulator
Direct
Indexed
Extended

Comments: When operating on unsigned values, only BEQ and BNE branches can be expected to behave properly. When operating on twos complement values, all signed branches are available.

Clear and Wait for Interrupt **CWAI**

Source Form: CWAI #$XX

E	F	H	I	N	Z	V	C

Operation: $CCR \leftarrow CCR \wedge MI$ (Possibly clear masks)
Set E (Entire state saved)
$SP' \leftarrow SP - 1$, $(SP) \leftarrow PCL$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow PCH$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow USL$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow USH$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow IYL$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow IYH$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow IXL$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow IXH$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow DPR$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow ACCB$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow ACCA$
$SP' \leftarrow SP - 1$, $(SP) \leftarrow CCR$

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 mask bit(s), stacks the entire machine state on the hardware stack, and then looks for an interrupt. When a (non-masked) interrupt occurs, no further machine state 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.

Decimal Addition Adjust **DAA**

Source Form: DAA

Operation: ACCA′ ← ACCA + CF(MSN):CF(LSN)
where CF is a correction factor, as follows:
The CF for each nibble (bcd digit) is determined separately, and is either 6 or 0.

Least Significant Nibble:
CF(LSN) = 6 IFF 1)H = 1
or 2)LSN $>$ 9
Most Significant Nibble:
CF(MSN) = 6 IFF 1)C = 1
or 2)MSN $>$ 9
or 3)MSN $>$ 8 *and* LSN $>$ 9

Condition Codes: H: Not affected.
N: Set IFF MSB of result is set.
Z: Set IFF 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 nibble such that: $0 \leq$ nibble ≤ 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

Decrement **DEC**

Source Form: DEC Q

Operation: M′ ← M − 1 [i.e., M′ ← M + FF_{16}]

Condition Codes: H: Not affected.
N: Set IFF bit 7 of result is set.
Z: Set IFF all bits of result are clear.
V: Set IFF 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 loop counter in multiple-precision computations.

Memory Addressing Modes: Accumulator
Direct
Indexed
Extended

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

Exclusive OR **EOR**

Source Forms: EORA P; EORB P

Operation: R′ ← R ⊕ M

Condition Codes: H: Not affected.
N: Set IFF bit 7 of result is set.
Z: Set IFF 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

Memory Addressing Modes: Direct
Extended
Immediate
Indexed

Exchange Registers **EXG**

Source Form: EXG R1, R2

Operation: R1 ↔ 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-bit with 16-bit.

Addressing Modes: Inherent

Increment **INC**

Source Form: INC Q

Operation: M′ ← M + 1

Condition Codes: H: Not affected.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF 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.

Memory Addressing Modes: Accumulator
Direct
Indexed
Extended

Comments: When operating on unsigned values, only the BEQ and BNE branches can be expected to behave consistently. When operating on twos complement values, all signed branches are correctly available.

Jump to Effective Address **JMP**

Source Form: JMP

Operation: PC′ ← EA

Condition Codes: Not affected.

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

Addressing Modes: Direct
Indexed
Extended

Jump to Subroutine at Effective Address **JSR**

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.

Addressing Modes: Direct
Indexed
Extended

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

Source Forms: LDA P; LDB P

Operation: R′ ← M

Condition Codes: H: Not affected.
N: Set IFF bit 7 of loaded data is set.
Z: Set IFF 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

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

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

Source Form: LDD P; LDX P; LDY P; LDS P; LDU P

Operation: R′ ← M:M+1

Condition Codes: H: Not affected.
N: Set IFF bit 15 of loaded data is set.
Z: Set IFF 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)

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Load Effective Address **LEA**

Source Form: LEAX, LEAY, LEAS, LEAU

Operation: R′ ← EA

Condition Codes: H: Not affected.
N: Not affected.
Z: LEAX, LEAY: Set IFF 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.

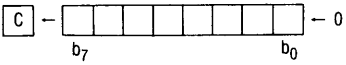
LEAX and LEAY affect Z to allow use as counters and for 6800 INX/DEX compatibility. LEAU and LEAS do not affect Z to allow for cleaning up the stack while returning Z as a parameter to a calling routine, and for 6800 INS/DES compatibility.

Register Addressing Mode: Pointer (X, Y, S, or U)

Memory Addressing Mode: Indexed

Logical Shift Left **LSL**

Source Form: LSL Q



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

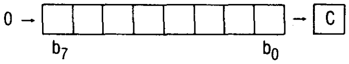
Condition Codes: H: Undefined.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Loaded with the result of $(b_7 \oplus b_6)$ of the original operand.
C: Loaded with bit 7 of the original operand.

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.

Addressing Modes: Accumulator
Direct
Indexed
Extended

Logical Shift Right LSR

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 IFF 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.

Addressing Modes: Accumulator
Direct
Indexed
Extended

Multiply Accumulators MUL

Source Form: MUL

Operation: ACCA':ACCB' ← ACCA × ACCB

Condition Codes: H: Not affected.
N: Not affected.
Z: Set IFF all bits of the result are clear.
V: Not affected.
C: Set IFF 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.

Addressing Modes: Inherent

Negate **NEG**

Source Form: NEG Q

Operation: $M' \leftarrow 0 - M$ i.e., $M' \leftarrow \overline{M} + 1$

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

Description: Replaces the operand with its twos 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.

Addressing Modes: Accumulator
Direct
Indexed
Extended

Flag Results: $(N \oplus V) = 1$ if 0 .LT. M (twos complement)
C = 1 if 0 .LO. M (Unsigned)
Z = 1 if 0 .EQ. M

No Operation **NOP**

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.

Addressing Modes: Inherent

Inclusive OR Memory into Register **OR**

Source Forms: ORA P; ORB P

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

Condition Codes: H: Not affected.
N: Set IFF high-order bit of result is set.
Z: Set IFF 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

Memory Address Modes: Immediate
Direct

Indexed
Extended

Inclusive OR Memory-Immediate into CCR **OR**

Source Form: ORCC #XX

Operation: $R \leftarrow R \vee MI$

Condition Codes: $CCR' \leftarrow CCR \vee MI$

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 Addressing Mode: CCR

Memory Addressing Mode: Memory immediate

Push Registers on the Hardware Stack **PSHS**

Source Form: PSHS register list
PSHS #Label

PC	U	Y	X	DP	B	A	CC

push order ⟶

```
Operation:  IFF B7 of MI set, then: SP' ← SP − 1, (SP) ← PCL
                                    SP' ← SP − 1, (SP) ← PCH
            IFF B6 of MI set, then: SP' ← SP − 1, (SP) ← USL
                                    SP' ← SP − 1, (SP) ← USH
            IFF B5 of MI set, then: SP' ← SP − 1, (SP) ← IYL
                                    SP' ← SP − 1, (SP) ← IYH
            IFF B4 of MI set, then: SP' ← SP − 1, (SP) ← IXL
                                    SP' ← SP − 1, (SP) ← IXH
            IFF B3 of MI set, then: SP' ← SP − 1, (SP) ← DPR
            IFF B2 of MI set, then: SP' ← SP − 1, (SP) ← ACCB
            IFF B1 of MI set, then: SP' ← SP − 1, (SP) ← ACCA
            IFF 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

Push Registers on the User Stack **PSHU**

Source Form: PSHU register list
PSHU #Label

PC	S	Y	X	DP	B	A	CC

push order ⟶

Operation: IFF B7 of MI set, then: US′ ← US − 1, (US) ← PCL
US′ ← US − 1, (US) ← PCH
IFF B6 of MI set, then: US′ ← US − 1, (US) ← SPL
US′ ← US − 1, (US) ← SPH
IFF B5 of MI set, then: US′ ← US − 1, (US) ← IYL
US′ ← US − 1, (US) ← IYH
IFF B4 of MI set, then: US′ ← US − 1, (US) ← IXL
US′ ← US − 1, (US) ← IXH
IFF B3 of MI set, then: US′ ← US − 1, (US) ← DPR
IFF B2 of MI set, then: US′ ← US − 1, (US) ← ACCB
IFF B1 of MI set, then: US′ ← US − 1, (US) ← ACCA
IFF 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

Pull Registers from the Hardware Stack **PULS**

Source Form: PULS register list
PULS #LABEL

PC	U	Y	X	DP	B	A	CC

←—— pull order

Operation: IFF B0 of MI set, then: CCR′ ← (SP), SP′ ← SP + 1
IFF B1 of MI set, then: ACCA′ ← (SP), SP′ ← SP + 1
IFF B2 of MI set, then: ACCB′ ← (SP), SP′ ← SP + 1
IFF B3 of MI set, then: DPR′ ← (SP), SP′ ← SP + 1
IFF B4 of MI set, then: IXH′ ← (SP), SP′ ← SP + 1
IXL′ ← (SP), SP′ ← SP + 1
IFF B5 of MI set, then: IYH′ ← (SP), SP′ ← SP + 1
IYL′ ← (SP), SP′ ← SP + 1
IFF B6 of MI set, then: USH′ ← (SP), SP′ ← SP + 1
USL′ ← (SP), SP′ ← SP + 1
IFF 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

Pull Registers From the User Stack **PULU**

Source Form: PULU register list
PULU #LABEL

PC	S	Y	X	DP	B	A	CC

←—— pull order

Operation: IFF B0 of MI set, then: $CCR' \leftarrow (US), US' \leftarrow US + 1$
IFF B1 of MI set, then: $ACCA' \leftarrow (US), US' \leftarrow US + 1$
IFF B2 of MI set, then: $ACCB' \leftarrow (US), US' \leftarrow US + 1$
IFF B3 of MI set, then: $DPR' \leftarrow (US), US' \leftarrow US + 1$
IFF B4 of MI set, then: $IXH' \leftarrow (US), US' \leftarrow US + 1$
$IXL' \leftarrow (US), US' \leftarrow US + 1$
IFF B5 of MI set, then: $IYH' \leftarrow (US), US' \leftarrow US + 1$
$IYL' \leftarrow (US), US' \leftarrow US + 1$
IFF B6 of MI set, then: $SPH' \leftarrow (US), US' \leftarrow US + 1$
$SPL' \leftarrow (US), US' \leftarrow US + 1$
IFF B7 of MI set, then: $PCH' \leftarrow (US), US' \leftarrow US + 1$
$PCL' \leftarrow (US), US' \leftarrow US + 1$

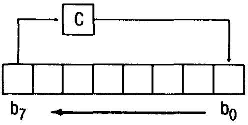
Condition Codes: May be pulled from stack; otherwise unaffected.

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

Rotate Left **ROL**

Source Form: ROL Q



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

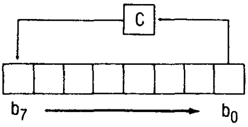
Condition Codes: H: Not affected.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Loaded with the result of $(b_7 \oplus b_6)$ of the original operand.
C: Loaded with bit 7 of the original operand.

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

Addressing Modes: Accumulator
Direct
Indexed
Extended

Rotate Right **ROR**

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 IFF bit 7 of result is set.
Z: Set IFF 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 9-bit rotation. The 6800 processor also affects the V flag.

Addressing Modes: Accumulator
Direct
Indexed
Extended

Return from Interrupt **RTI**

Source Form: RTI

Operation: CCR′ ← (SP), SP′ ← SP + 1

IFF CCR bit E is CLEAR then:		← (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
IFF 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.

Addressing Mode: Inherent

Return from Subroutine **RTS**

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.

Addressing Mode: Inherent

Subtract with Borrow **SBC**

Source Forms: SBCA P; SBCB P

Operation: $R' \leftarrow R - M - C$ [i.e., $R' \leftarrow R + \overline{M} + \overline{C}$]

Condition Codes: H: Undefined.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF the operation causes an 8-bit twos complement overflow.
C: Set IFF 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

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Sign Extended **SEX**

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 IFF the MSB of the result is set.
Z: Set IFF all bits of ACCD are clear.
V: Not affected.
C: Not affected.

Description: This instruction transforms a twos complement 8-bit value in ACCB into a twos complement 16-bit value in the double accumulator.

Addressing: Inherent

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

Source Form: STA P; STB P

Operation: M' ← R

Condition Codes: H: Not affected.
N: Set IFF bit 7 of stored data was set.
Z: Set IFF 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

Memory Addressing Modes: Direct
Indexed
Extended

Store Register into Memory—16 Bits **ST**

Source Form: STD P; STX P; STY P; STS P; STU P

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

Condition Codes: H: Not affected.
N: Set IFF bit 15 of stored data was set.
Z: Set IFF all bits of stored data are clear.
V: Cleared.
C: Not affected.

Description: Writes the 16-bit register into consecutive memory locations.

Register Addressing Modes: Double accumulator
Pointer (X, Y, S, or U)

Memory Addressing Modes: Direct
Indexed
Extended

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

Source Forms: SUBA P; SUBB P

Operation: $R' \leftarrow R - M$ [i.e., $R' \leftarrow R + \overline{M} + 1$]

Condition Codes: H: Undefined.
N: Set IFF bit 7 of the result is set.
Z: Set IFF all bits of the result are clear.
V: Set IFF the operation caused an 8-bit twos complement overflow.
C: Set IFF 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 binary carry.

Register Addressing Mode: Accumulator

Flag Results: $(N \oplus V) = 1$ if R .LT. M (twos complement)
$C = 1$ if R .LO. M (unsigned)
$Z = 1$ if R .EQ. M

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Subtract Memory from Register—16 Bits **SUB**

Source Form: SUBD P

Operation: $R' \leftarrow R - M:M+1$ [i.e.,, $R' \leftarrow R + \overline{M:M+1} + 1$]

Condition Codes: H: Unaffected.
N: Set IFF bit 15 of result is set.
Z: Set IFF all bits of result are clear.
V: Set IFF the operation caused a 16-bit twos complement overflow.
C: Set IFF 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

Memory Addressing Modes: Immediate
Direct
Indexed
Extended

Subtract Sets: (N⊕V) = 1 if R .LT. M (twos complement)
C = 1 if R .LO. M (unsigned)
Z = 1 if R .EQ. M

Software Interrupt SWI

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.

Addressing Mode: Absolute indirect

Software Interrupt 2 SWI2

Source Form: SWI2

Operation: Set E (Entire state saved)
SP′ ← SP − 1, (SP) ← PCL
SP′ ← SP − 1, (SP) ← PCH

SP′ ← SP − 1, (SP) ← USL
SP′ ← SP − 1, (SP) ← USH
SP′ ← SP − 1, (SP) ← IYL
SP′ ← SP − 1, (SP) ← IYH
SP′ ← SP − 1, (SP) ← IXL
SP′ ← SP − 1, (SP) ← IXH
SP′ ← SP − 1, (SP) ← DPR
SP′ ← SP − 1, (SP) ← ACCB
SP′ ← SP − 1, (SP) ← ACCA
SP′ ← SP − 1, (SP) ← CCR
PC′ ← (FFF4):(FFF5)

Condition Codes: Not affected.

Description: All of the 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.

Addressing Mode: Absolute indirect

Software Interrupt 3 **SWI3**

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.

Addressing Mode: Absolute indirect

Synchronize to External Event **SYNC**

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 three cycles or more, the processor will perform the interrupt routine. If the interrupt is masked or is shorter than three cycles, the processor simply continues to the next instruction (without stacking registers). While SYNCING, the address and data buses are tri-state.

Addressing Modes: Inherent

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

```
FAST  SYNC              WAIT FOR DATA              interrupt
      LDA     DISC      DATA FROM DISC AND CLEAR INTER-
                        RUPT
      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.

Transfer Register to Register **TFR**

Source Form: TFR R_1,R_2

Operation: $R_2 \leftarrow R_1$

Condition Codes: Not affected (unless $R_2 = \text{CCR}$).

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-bit to 16-bit.

Addressing Modes: Inherent

Test **TST**

Source Form: TST Q

Operation: TEMP ← M − 0

Condition Codes: H: Not affected.
N: Set IFF bit 7 of the result is set.
Z: Set IFF 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.

Memory Addressing Modes: Accumulator
Direct
Indexed
Extended

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.

APPENDIX B

The 6820/6821 Peripheral Interface Adapter (PIA)

6821 FUNCTIONAL DESCRIPTION

Before entering into a detailed discussion of the PIA pin assignments and interfacing requirements, let us take a look at the PIA from a functional viewpoint. A functional diagram of the 6821 is shown in Fig. B-1. First, note the PIA can be looked at functionally as having two sides—a 6809 side and a peripheral side. The 6809 side includes the data, address, and control lines which interface to the 6809 data, address, and control buses. The peripheral side contains two i/o *ports* (A and B) which will interface to peripheral devices. Each port contains eight data lines which may be configured independently as input or output. This allows for a high degree of interfacing flexibility. The procedure for port configuration will be discussed shortly. Internally, the PIA contains six 8-bit registers. Three registers apply to port A and three apply to port B. Each group of three registers has the same function with respect to its respective port. Each group of three registers contains a data register (DRA or DRB, data direction register (DDRA or DDRB), and control register (CRA or CRB). A discussion of each follows.

Data Registers (DRA and DRB)

Each data register acts as a temporary 8-bit storage register for data being transferred between the 6809 and the i/o device connected to the PIA chip. Each of the 8 bits in the data registers is connected to one of the i/o port data lines. Recall that each port contains eight i/o data lines. For example, port A contains eight data lines, PA0 through PA7. Therefore, bit 0 of DRA is tied to PA0, bit 1 of DRA is tied to PA1, and so on. The same arrangement is used for the DRB bits and lines PB0 through PB7. The register bits are latched when used for output operations and they are unlatched (simple gates) when they are used for input.

Data Direction Registers (DDRA and DDRB)

The data direction registers are 8-bit registers which define the port lines as being used for either input or output operations. Each bit within the DDR configures its corresponding port data line. A 1 in a DDR bit will cause its corresponding port line to be configured as an output line, while a 0 will cause it to be configured as an input line. For example, if DDRA bit 3 contains a 1, the PA3 line will be configured as an output data line. If DDRA bit 4 contained a 0, the PA4 line would be configured as an input data line. The ports are configured by storing a data byte into each data direction register. To configure port A as an input port and port B as an output port, you would store 00 in DDRA and FF in DDRB. Remember that since the data direction register and the data register are considered to be memory locations, they are loaded with memory-reference instructions.

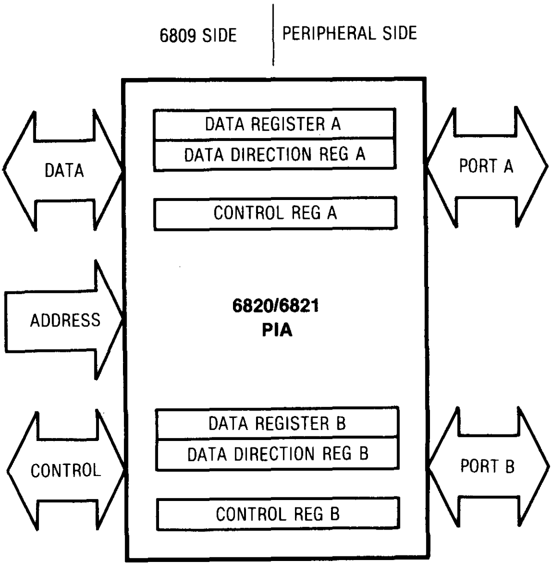


Fig. B-1. Functional diagrams of the 6820/6821 PIA.

Control Registers (CRA and CRB)

The control registers are 8-bit registers which are used for a variety of control functions. Each bit within the register controls a particular function. The control register will allow four of the peripheral control lines of the PIA to be used for interrupt servicing and polling routines. The control register is also used to select either the DR or DDR for use in a data transfer operation. A more detailed discussion of the control register is included later in this appendix.

6820/6821 PIN ASSIGNMENTS

The PIA is a 40-pin integrated circuit. The various pin assignments are shown in Fig. B-2. A functional description of each pin follows.

V_{SS} (Ground: Pin 1)

Pin 1 should be connected to the system ground.

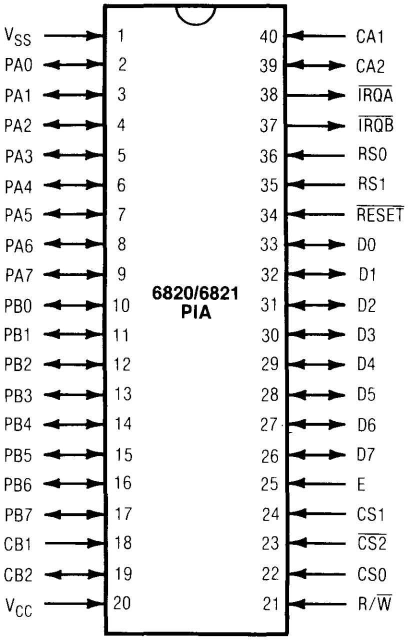


Fig. B-2. Pin assignments of the 6820/6821 PIA.

Port A Data Lines (PA0–PA7: Pins 2–9)

As stated earlier, each of these lines may be used as an input or an output line. The use of a particular line is determined through proper selection of individual bits in the data direction register (DDRA).

Data will be transferred into the 6809 through the lines that have been configured as input. This will be accomplished when a load instruction is executed, transferring the information from the PIA port input lines to the 6809 internal register that is being loaded. In the input mode, each input data line represents a maximum of one TTL load.

Data will be output to the i/o devices through the data lines that have been configured as output lines. This output transfer will be accomplished with a store instruction which, when executed, will transfer data from the desired 6809 register to data register A (DRA). The data that has been stored in DRA will then appear on the port A data lines which have been configured as output lines. With the 6820 the port A lines only have CMOS drive capabilities and must be buffered to provide drive for TTL devices. With the 6821, they are directly compatible.

Port B Data Lines (PB0–PB7: Pins 10–17)

These lines are used very similar to the port A data lines. You may configure each of the lines as being either an input or an output line through the use of data direction register B (DDRB). With the 6820 one major difference between ports A and B is that when the port B lines have been configured as output, they are TTL compatible and each may be used as a source of up to 1 milliampere at 1.5 volts to directly drive the base of a transistor switch. With the 6821 *both* ports have TTL drive capabilities.

Interrupt Input—Port B (CB1: Pin 18)

This is an input-only line used to set bit 7 of control register B which is used as a flag to indicate that a peripheral wishes to interrupt the 6809. This will be discussed in more detail later in this appendix (see *PIA Control Registers*).

Peripheral Control—Port B (CB2: Pin 19)

This line can be programmed through the use of control register B to act as an interrupt input or as a peripheral control output to provide handshaking. When in the output mode it is TTL compatible, and when configured as an interrupt input it represents one TTL load. This pin will be discussed in more detail later in this appendix (see *PIA Control Registers*).

V_{cc} (Pin 20)

This pin is connected to the system +5-volt dc power supply.

Read/Write ($R/\overline{W}$: Pin 21)

This pin is connected directly to the $R/\overline{W}$ line on the 6809. A low state on this line allows data to be transferred from the 6809 to the PIA. A high state allows for data transfer from the PIA to the 6809. Data will only be transferred when the proper address and enabling pulse are present at the PIA. PIA addressing and enabling will be discussed shortly.

Chip Selects (CS0, $\overline{\text{CS2}}$, CS1: Pins 22–24)

These pins are used in the same way that the chip-select signals were used on the 6810 R/W memory and 6830 ROM chips. They will partially decode the address bus to select the PIA. To select the PIA, CS0 and CS1 must be high and $\overline{\text{CS2}}$ must be low.

Enable (E: Pin 25)

This pin is used to supply a timing signal to the PIA and, therefore, is normally connected directly to the E clock. To completely enable the PIA, the chip selects must be held in their active state for the duration of the E pulse.

Data (D0–D7: Pins 33–26)

These pins are connected directly to the eight data bus lines D0 through D7. The data bus lines are bidirectional and allow data transfer between the 6809 and PIA. The output drivers are three-state buffered and remain in their high-impedance state except when a PIA read operation is being performed.

Reset ($\overline{\text{Reset}}$: Pin 34)

A high-to-low transition at this pin will cause all register bits in the PIA to be reset to a logical 0 state. It can be used with a power-on reset signal or tied to the 6809 reset interrupt pin to reset the PIA when the entire system is reset.

Register Selects (RS0, RS1: Pins 36, 35)

These two lines are used to select the various registers within the PIA. They are normally tied to address lines A0 and A1 and are used in conjunction with the control registers to select the specific register that is desired. This selection process is discussed in the next section of this appendix.

Interrupt Requests ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$: Pins 38, 37)

These are output lines that can be *wire*-ORed together to be connected directly to the 6809 $\overline{\text{IRQ}}$ line. When an i/o device generates an interrupt, an interrupt flag bit of the respective PIA control register will be set which, in turn, causes the respective $\overline{\text{IRQ}}$ line to go low. This generates an interrupt request that is sent to the 6809. Each of these lines has two interrupt flag bits in its respective control register that can cause the $\overline{\text{IRQ}}$ line to go low. Each of these internal flags corresponds to a particular peripheral interrupt line; CA1 and CA2 correspond to port A and CB1 and CB2 correspond to port B. Once an internal interrupt flag has been set, the $\overline{\text{IRQ}}$ line will remain low until the flag is cleared by servicing the interrupt with a PIA read or write operation. Therefore, an interrupt request is not lost if the I flag in the 6809 condition code register is set, which disables it from recognizing interrupts. These lines could also connect to the 6809 $\overline{\text{FIRQ}}$ or $\overline{\text{NMI}}$ lines.

Peripheral Control (CA2: Pin 39)

This line is used in essentially the same way as the CB2 line (pin 19). It can be programmed through the use of control register A to act as an interrupt input line or it may be used as a peripheral control output line. When in the output mode it is TTL compatible, and when configured as an interrupt input it represents one TTL load. A more detailed discussion of this pin function will follow in this appendix (see *PIA Control Registers*).

Interrupt Input (CA1: Pin 40)

This line is similar to the CB1 line (pin 18). It is an input-only line used to set bit 7 of control register A which is used as a flag to indicate a peripheral interrupt. This, in turn, will cause the $\overline{\text{IRQA}}$ line to go low, generating an interrupt request.

PIA INTERFACING AND ADDRESSING

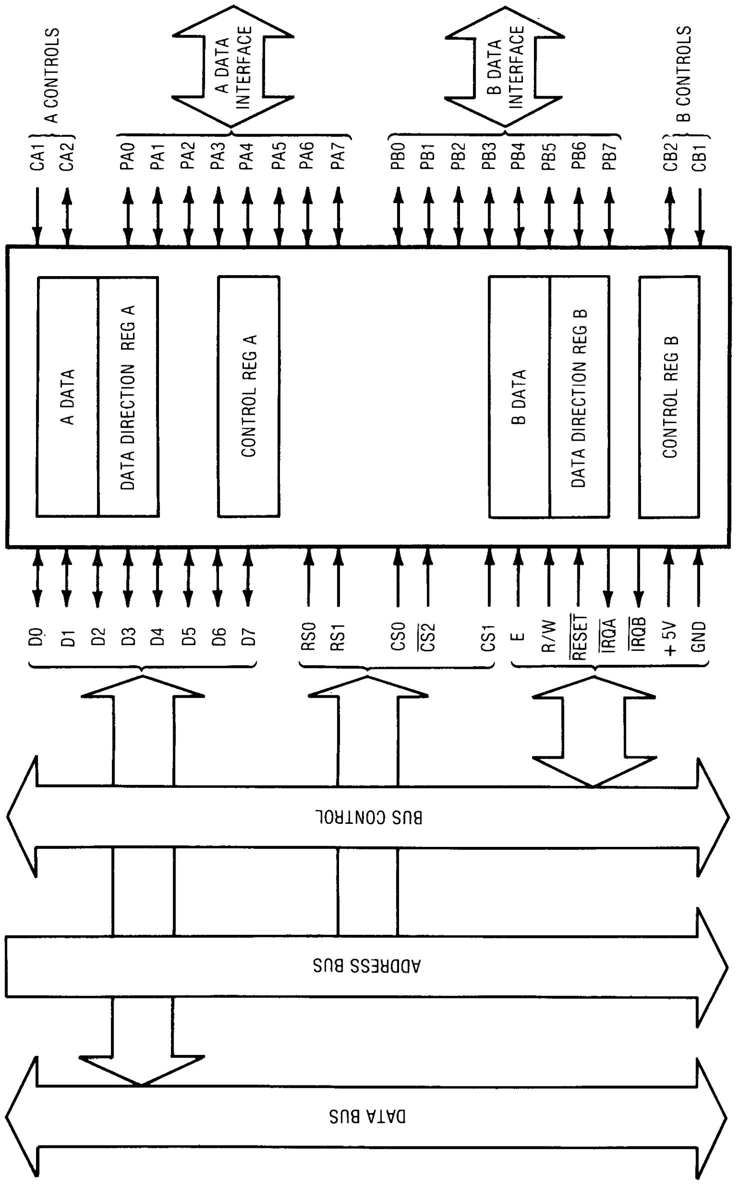
Fig. B-3 shows how the various pins would be utilized to interface the PIA to the 6809. The PIA data lines would be connected directly to the 6809 data lines D0 through D7. For control, the following PIA lines would be connected directly to the corresponding signals on the 6809 control bus: $\overline{\text{R/W}}$, $\overline{\text{RESET}}$, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, +5V, and GND.

Chip Selection

To access the PIA, you will use the PIA register-select, chip-select, and enable lines. Recall that the chip-select pins along with the enable pin will select the PIA. To provide timing between the 6809 and the PIA, you will connect the E clock to the PIA enable pin (E). The remaining chip selects (CS0, CS1, and $\overline{\text{CS2}}$) are tied to the 6809 address bus to provide partial decoding for the chip.

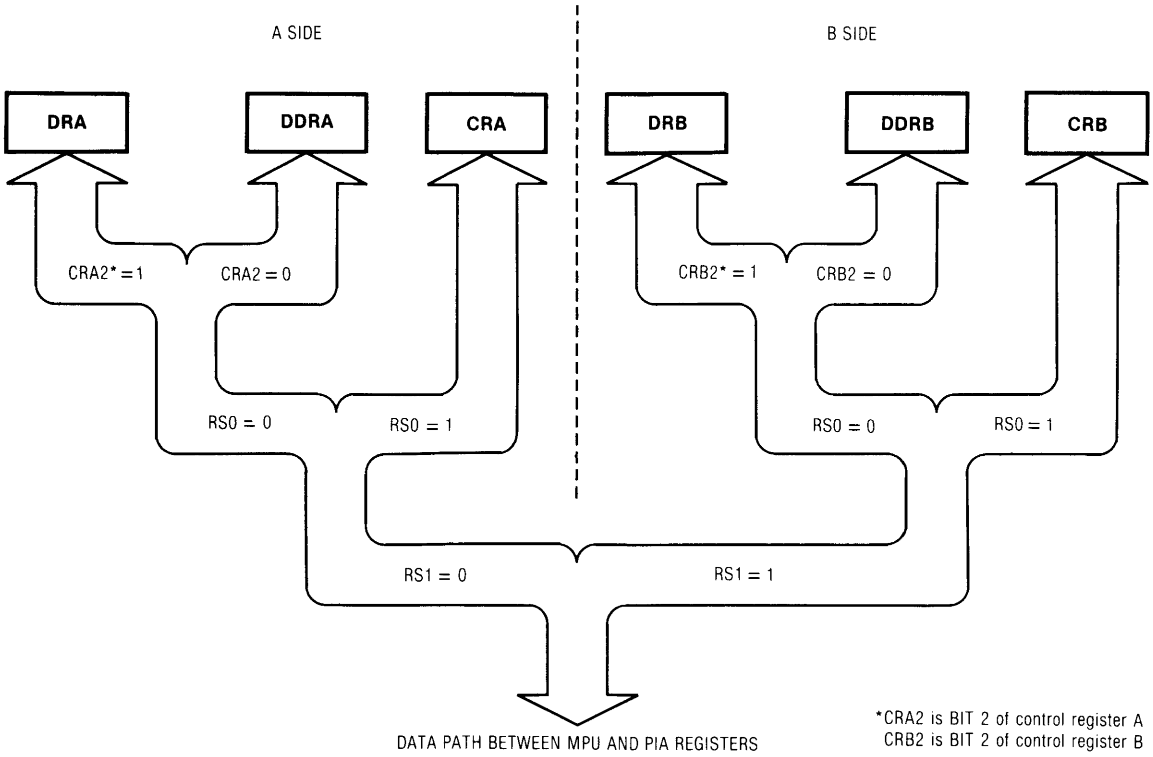
Register Selection

The register select pins, RS0 and RS1, are connected to address lines A1 and A0, respectively. Recall that these pins are used to select one of six registers within the PIA. This creates a problem since there are only four possible logic combinations for these two pins. However, we wish to select one of *six* registers. The solution to the problem is the PIA control register. The register selection process is shown in Fig. B-4. The RS1 bit is used to access either port A or port B. If RS1 is low, port A will be selected, but if RS1 is high, port B will be selected. The RS0 bit narrows the selection still further by selecting either the control register or the data register/data direction register of the selected port. If RS0 is high, the control register will be selected, but if RS0 is low, either the data register *or* the data direction register will be selected, depending upon the status of *bit* 2 of the respective control register. If bit 2 of the control register is low and RS0 is low, the data direction register



Courtesy Motorola Semiconductor Products, Inc.

Fig B-3. PIA interfacing.



*CRA2 is BIT 2 of control register A
CRB2 is BIT 2 of control register B

Fig. B-4. PIA register selection.

Courtesy Heath Co.

will be selected. However, if bit 2 of the control register is high, the data register is selected. For example, suppose RS1 = 1, RS0 = 0, and bit 2 of control register B is low. With these conditions, the data direction register of port B (DDRB) will be selected.

Fig. B-5 shows how a PIA might be connected to the 6809 system. It is necessary only to allocate four addresses to select any register in the PIA. In this example we have used addresses 5000 through 5003. Naturally we are only partially decoding the address bus since from the decoding chart you can see that the PIA will be enabled for addresses 5000 through 7FFF. This does not create a problem as long as no other chips are assigned to any of these addresses. Using this decoding scheme the information in Fig. B-6 shows how the PIA would respond to addresses 5000 through 5003 and which register would be selected for each address. Note that with addresses 5000 and 5001, the data direction *or* data register can be selected. The specific register that is

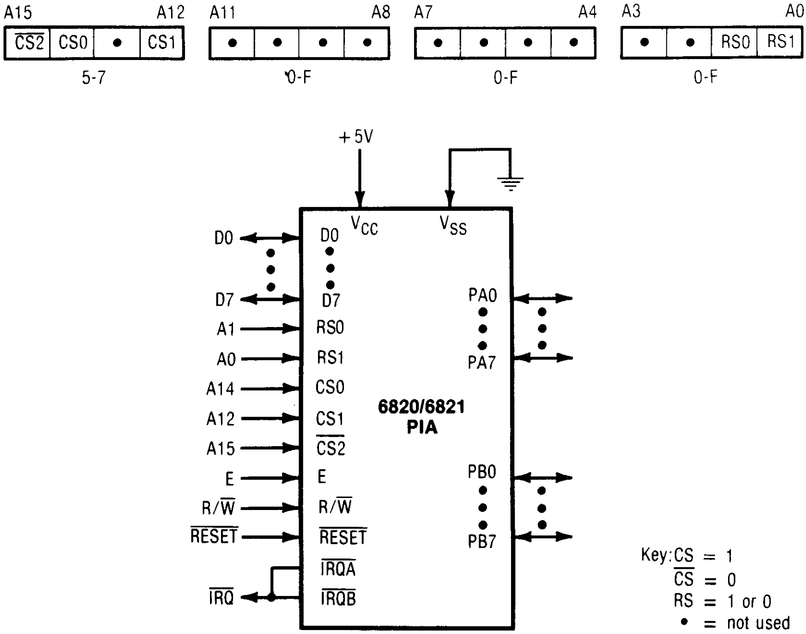


Fig. B-5. PIA pin connections and decoding chart.

ADDRESS	CS0	CS1	$\overline{CS2}$	RS1	RS0	PIA REGISTER SELECTED
5000	1	1	0	0	0	DDRA or DRA*
5001	1	1	0	1	0	DDRB or DRB*
5002	1	1	0	0	1	CRA
5003	1	1	0	1	1	CRB

*Depends on bit 2 of the control register.

Fig. B-6. Example of PIA register selection.

selected will depend on the status of bit 2 in the respective control register. The control registers are selected with addresses 5002 and 5003.

PIA INITIALIZATION AND SERVICING

Prior to using the PIA for data transfer you must initialize it by defining the port lines as either input or output lines. As you saw in the first part of this appendix this is accomplished by the 1s and 0s placed in the bits in each data direction register (DDR). Recall that if a 1 existed in a DDR bit, its corresponding port line would be configured as an output data line, while if a 0 existed its port line would be an input data line. To initialize the PIA you will have to execute an initialization program that will store a binary number in each data direction register and, thus, configure each port. The initialization procedure will be as follows:

1. Clear bit 2 of both control registers.
2. Store a number in DDRA to configure port A.
3. Store a number in DDRB to configure port B.
4. Set bit 2 of control register A (CRA).
5. Set bit 2 of control register B (CRB).

In the above procedure, Step 1 clears bit 2 of both control registers so that the data *direction* register will be selected rather than the data register. The ports will then be configured by storing a binary number in each data direction register (Steps 2 and 3). After each port has been configured, bit 2 of the control register is set such that the data register will be selected for subsequent data transfer.

Example B-1: PIA Initialization

The following program will configure port A as an 8-bit input port and port B as an output port. We will assume that the PIA has been assigned to addresses 5000 through 5003 and that the system have been reset prior to executing the initialization routine.

```
LDA #
 50
TFR A,DPR
LDD #
 00
 FF
STD $
 00
LDD #
 04
 04
STD $
 02
```

First, resetting the system will reset the PIA if the $\overline{\text{RESET}}$ pin on the PIA is connected to the system reset, as is usually the case. This will cause all of the registers in the PIA to be cleared, and therefore the first step of the initialization procedure is accomplished. The initialization routine first stores the PIA page number (50) into the direct page register such that direct addressing can be used for data transfers. Port A is then configured as an input port and port B as an output port by storing the accumulator D contents (00FF) to address 5000. Note that since DDRA and DDRB are assigned to consecutive memory locations, the STD instruction will store 00 to address 5000 (DDRA) and FF to address 5001 (DDRB). Bit 2 of both control registers is then set by storing 0404 to address 5002:5003.

Example B-2: PIA Data I/O

Assuming that the PIA has been configured as in Example B-1, the following routine will input data from port A, then output the same data to port B.

```
LDA $
 00
STA $
 01
```

To input data from port A, you can use any load instruction which addresses the port A data register (DRA at address 5000). The data register will be selected rather than the data direction register since you have already set bit 2 of the control register in your initialization program. To output the data to port B, you will use a store instruction that addresses the port B data register (DRB at address 5001). This is a very simple program since data is just being transferred from an input to an output port. Once the data is in the 6809, however, you have the full power of the 6809 instruction set available to analyze that data to determine the output conditions.

PIA Control Registers

Now we will discuss the control registers of the PIA in more detail. Besides the bit-2 function of the control register which has already been discussed, the control register is used mainly for control of interrupts. The bit format of each control register is shown in Fig. B-7. Actually, each control register is identical in format and function. Therefore we will confine our discussion to control register A, keeping in mind that the function of control register B is the same. Our discussion will begin with bit 0 (CRA-0) and bit 1 (CRA-1).

	7	6	5	4	3	2	1	0
CRA	IRQA1	IRQA2	CA2 CONTROL			DDRA ACCESS	CA1 CONTROL	

	7	6	5	4	3	2	1	0
CRB	IRQB1	IRQB2	CB2 CONTROL			DDRB ACCESS	CB1 CONTROL	

Courtesy Motorola Semiconductor Products, Inc.

Fig. B-7. Control register format.

Bits 0 and 1 (CRA-0 and CRA-1) of the control register are labeled CA1 Control since they are used to define the effect and active state of the CA1 pin on the PIA. Recall that CA1 is an input-only pin that can be used by an i/o device to generate interrupt requests. When the pin is activated, the interrupt flag bit (CRA-7) of the control register will be set, indicating an interrupt request has been generated. Bit 0 (CRA-0) of the control register will determine the effect of setting this flag. If CRA-0 is set (1), the flag will cause the $\overline{\text{IRQA}}$ output pin to go low, thus generating an interrupt request to the 6809. If CRA-0 is cleared (0), the $\overline{\text{IRQA}}$ pin will remain high, *masking* out any interrupt request. Thus CRA-0 determines whether the interrupt mode for the port is active or inactive. Bit 1 of control register A (CRA-1) defines the *active state* of pin CA1. If CRA-1 is set (1), a low-to-high transition on the CA1 pin will cause the interrupt flag bit (CRA-7) of the control register to be set. If CRA-1 is cleared (0), a high-to-low transition will cause the interrupt flag to be set. Thus, either a positive pulse or a negative pulse may be used to generate an interrupt signal. Fig. B-8 summarizes the function of these two control register bits.

CRA-1 (CRB-1)	CRA-0 (CRB-0)	INTERRUPT INPUT CA1(CB1)	INTERRUPT FLAG CRA-7(CRB-7)	MPU INTERRUPT REQUEST $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$)
0	0	↓ ACTIVE	SET HIGH ON ↓ OF CA1 (CB1)	DISABLED — $\overline{\text{IRQ}}$ REMAINS HIGH
0	1	↓ ACTIVE	SET HIGH ON ↓ OF CA1 (CB1)	GOES LOW WHEN THE INTERRUPT FLAG BIT CRA-7 (CRB-7) GOES HIGH
1	0	↑ ACTIVE	SET HIGH ON ↑ OF CA1 (CB1)	DISABLED — $\overline{\text{IRQ}}$ REMAINS HIGH
1	1	↑ ACTIVE	SET HIGH ON ↑ OF CA1 (CB1)	GOES LOW WHEN THE INTERRUPT FLAG BIT CRA-7 (CRB-7) GOES HIGH

Notes: 1. ↑ indicates positive transition (low to high).
2. ↓ indicates negative transition (high to low).
3. The interrupt flag bit CRA-7 is cleared by the MPU Read of the A Data Register and CRB-7 is cleared by the MPU Read of the B Data Register.
4. If CRA-0 (CRB-0) is low when an interrupt occurs (interrupt disabled) and is later brought high, $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$) occurs after CRA-0 (CRB-0) is written to a "one."

Courtesy Motorola Semiconductor Products, Inc.

Fig. B-8. Function of control register bits 0 and 1.

Bit 2 of the control register (CRA-2) has already been discussed and is used entirely for register selection. Bits 3, 4, and 5 of the control register (CRA-3, CRA-4, and CRA-5) are labeled CA2 Control since they are used to define the function, effect, and active states of the CA2 pin on the PIA. Recall that the CA2 pin can be designated as either input or output. This designation is accomplished by CRA-5 of the control register. When CRA-5 is cleared, the CA2 pin is configured as an input line. When CRA-5 is set, CA2 is designated as an output line.

CA2 Input

When configured as an input line, the CA2 pin is used as an interrupt line similar to CA1. In this mode an active level on CA2 will cause bit 6 (CRA-6) of the control register to be set. CRA-6 is the interrupt flag used in conjunction with the CA2 pin in the same way that CRA-7 is used in conjunction with CA1. When being used as an interrupt input, the CA2 active state and effect are defined by bits 3 and 4 of the control register (CRA-3 and CRA-4). CRA-3 is used to determine the effect of setting the CRA-6 flag similar to the way CRA-0 was used in conjunction with the CRA-7 flag. If CRA-3 is set, the CRA-6 flag will cause the $\overline{\text{IRQA}}$ pin to go low, thus generating an interrupt request to the 6809. If CRA-3 is cleared, the $\overline{\text{IRQA}}$ pin will remain high, thus masking out the interrupt request. Bit 4 of control register A (CRA-4) will define the active state of pin CA2 similar to the way CRA-1 defines the active state of CA1. If CRA-4 is set (1), a low-to-high transition on the CA2 pin will cause the interrupt flag bit (CRA-6) to be set. If CRA-4 is cleared (0), a high-to-low transition will cause the interrupt flag bit to be set. Fig. B-9 summarizes the function of these control register bits when CA2 is used as an *input line.*

CA2 Output

Recall that CA2 will be configured as an output line when bit 5 of the control register is set. When CA2 is designated as an output line, the interrupt

CONTROL OF CA2 AND CB2 AS INTERRUPT INPUTS CRA5(CRB5) IS LOW

CRA-5 (CRB-5)	CRA-4 (CRB-4)	CRA-3 (CRB-3)	INTERRUPT INPUT CA2(CB2)	INTERRUPT FLAG CRA-6(CRB-6)	MPU INTERRUPT REQUEST $\overline{\text{IRQA}}$($\overline{\text{IRQB}}$)
0	0	0	↓ ACTIVE	SET HIGH ON ↓ OF CA2 (CB2)	DISABLED — $\overline{\text{IRQ}}$ REMAINS HIGH
0	0	1	↓ ACTIVE	SET HIGH ON ↓ OF CA2 (CB2)	GOES LOW WHEN THE INTERRUPT FLAG BIT CRA-6 (CRB-6) GOES HIGH
0	1	0	↑ ACTIVE	SET HIGH ON ↑ OF CA2 (CB2)	DISABLED — $\overline{\text{IRQ}}$ REMAINS HIGH
0	1	1	↑ ACTIVE	SET HIGH ON ↑ OF CA2 (CB2)	GOES LOW WHEN THE INTERRUPT FLAG CRA-6 (CRB-6) GOES HIGH

Notes: 1. ↑ indicates positive transition (low to high).
2. ↓ indicates negative transition (high to low).
3. The interrupt flag bit CRA-6 is cleared by the MPU Read of the A Data Register and CRB-6 is cleared by the MPU Read of the,B Data Register.
4. If CRA-3 (CRB-3) is low when an interrupt occurs (interrupt disabled) and is later brought high, $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$) occurs after CRA-3 (CRB-3) is written to a "one."

Courtesy Motorola Semiconductor Products, Inc.

Fig. B-9. Function of control register bits when CA2 is used as an input line.

flag (CRA-6) will be cleared and remain in that state as long as bit 5 is set.

You will use CA2 as an output for polling and handshaking routines. Handshaking or polling requires the use of status bits that would indicate when a peripheral device has requested service and when the 6800 has completed the service. The use of CA1 as an input and CA2 as an output as shown in Fig. B-10 will provide the proper status levels to permit handshaking between the 6809 and a peripheral device. The procedure will be as follows.

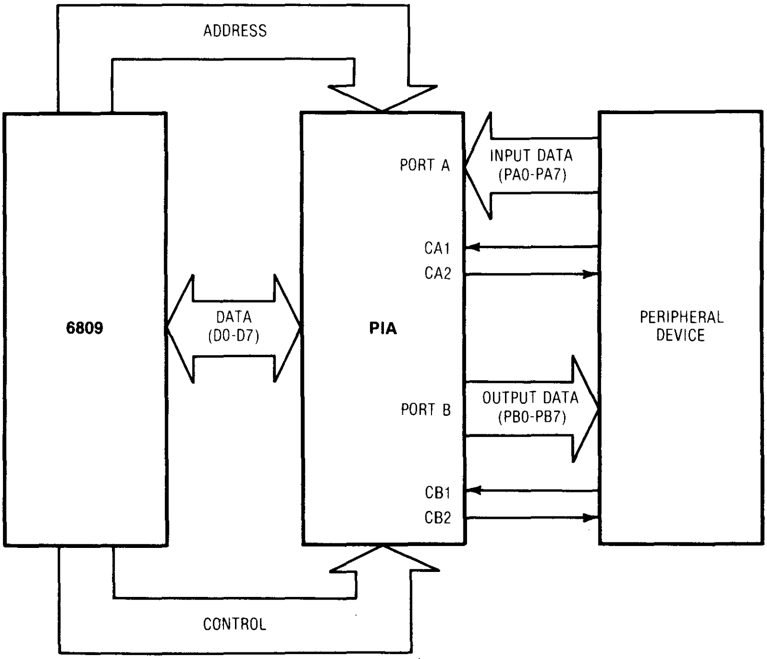


Fig. B-10. Complete input and output handshaking using the PIA.

1. The peripheral device will generate an interrupt by activating the CA1 line on the PIA, signaling that it has data to give the 6809.
2. The CA1 interrupt causes the interrupt flag bit CRA-7 to set.
3. The interrupt flag causes an interrupt request to be generated to the 6809 and *also* causes CA2 to go high.
4. When the interrupt request is acknowledged, the 6809 will read the data from the port A data register (DRA).
5. After the read operation takes place, the CA2 line will go low and CRA-7 is cleared, signaling the peripheral that the interrupt has been serviced and that the 6809 is ready for more data. Thus, the handshake is complete.

To achieve this complete handshake, CRA-5 must be set with CRA-3 and CRA-4 cleared. Therefore bits 5, 4, and 3 of control register A would be 100.

If you do not desire to use interrupts and decide to use programmed i/o, CA1 could be eliminated from Fig. B-10 and CA2 would be used as an output to signal the peripheral device. Here, the peripheral device would make data available on a continuing basis to the PIA port, but it needs to know when the 6809 has read the data from the data register so that new data can be supplied. This is a partial handshake. In this mode, CA2 will normally be high, then go low after a read-port A operation is executed. It will remain low for one enable signal (E) cycle. To achieve this mode, CRA-5 and CRA-3 must be set with CRA-4 cleared. Therefore, bits 5, 4, and 3 of control register A would be 101.

There are two other possibilities for the output condition of CA2. They are:

1. CRA-5, CRA-4, CRA-3 = 110
2. CRA-5, CRA-4, CRA-3 = 111

In the first case the CA2 output line will be held in a low state and in the second case CA2 will be held high.

When CA2 and CB2 are used as output lines, they have slightly different functions. When handshaking, port A will be used completely as an input port and port B will be used as an output port. Therefore, CA2 will indicate when the 6809 has read (loaded) data from the port A data register (DRA) and CB2 will indicate when the 6809 has written (stored) data into the port B data register (DRB). Figs. B-11 and B-12 summarize the functions of CA2

CRA-5 IS HIGH

CRA-5	CRA-4	CRA-3	CA2	
			CLEARED	SET
1	0	0	LOW ON NEGATIVE TRANSITION OF E AFTER AN MPU READ "A" DATA OPERATION.	HIGH WHEN THE INTERRUPT FLAG BIT CRA-7 IS SET BY AN ACTIVE TRANSITION OF THE CA1 SIGNAL.
1	0	1	LOW ON NEGATIVE TRANSITION OF E AFTER AN MPU READ "A" DATA OPERATION.	HIGH ON THE NEGATIVE EDGE OF THE FIRST "E" PULSE WHICH OCCURS DURING A DESELECT.
1	1	0	LOW WHEN CRA-3 GOES LOW AS A RESULT OF AN MPU WRITE TO CONTROL REGISTER "A".	ALWAYS LOW AS LONG AS CRA-3 IS LOW. WILL GO HIGH ON AN MPU WRITE TO CONTROL REGISTER "A" THAT CHANGES CRA-3 TO "ONE".
1	1	1	ALWAYS HIGH AS LONG AS CRA-3 IS HIGH. WILL BE CLEARED AS AN MPU WRITE TO CONTROL REGISTER "A" THAT CLEARS CRA-3 TO A "ZERO".	HIGH WHEN CRA-3 GOES HIGH AS A RESULT OF AN MPU WRITE TO CONTROL REGISTER "A".

Courtesy Motorola Semiconductor Products, Inc.

Fig. B-11. Control of CA-2 as an output.

CRB-5 IS HIGH

CRB-5	CRB-4	CRB-3	CB2	
			CLEARED	SET
1	0	0	LOW ON THE POSITIVE TRANSITION OF THE FIRST E PULSE FOLLOWING AN MPU WRITE "B" DATA REGISTER OPERATION.	HIGH WHEN THE INTERRUPT FLAG BIT CRB-7 IS SET BY AN ACTIVE TRANSITION OF THE CB1 SIGNAL.
1	0	1	LOW ON THE POSITIVE TRANSITION OF THE FIRST E PULSE AFTER AN MPU WRITE "B" DATA REGISTER OPERATION.	HIGH ON THE POSITIVE EDGE OF THE FIRST "E" PULSE FOLLOWING AN "E" PULSE WHICH OCCURED WHILE THE PART WAS DESELECTED.
1	1	0	LOW WHEN CRB-3 GOES LOW AS A RESULT OF AN MPU WRITE IN CONTROL REGISTER "B".	ALWAYS LOW AS LONG AS CRB-3 IS LOW. WILL GO HIGH ON AN MPU WRITE IN CONTROL REGISTER "B" THAT CHANGES CRB-3 TO "ONE".
1	1	1	ALWAYS HIGH AS LONG AS CRB-3 IS HIGH. WILL BE CLEARED WHEN AN MPU WRITE CONTROL REGISTER "B" RESULTS IN CLEARING CRB-3 TO "ZERO"	HIGH WHEN CRB-3 GOES HIGH AS A RESULT OF AN MPU WRITE INTO CONTROL REGISTER "B".

Courtesy Motorola Semiconductor Products, Inc.

Fig. B-12. Control of C-2 as an output.

and CB2, respectively, when used as an output line. The following examples should help to clarify the preceding discussion.

Example B-3

Suppose you store the hex number 27 into control register B. How will this control port B? The control register would be configured as shown below:

CRB-7	CRB-6	CRB-5	CRB-4	CRB-3	CRB-2	CRB-1	CRB-0
0	0	1	0	0	1	1	1

This configuration will provide complete *data output* handshaking through port B. The following is a description of each bit function:

CRB-0 set will cause an interrupt to be generated when the interrupt flag (CRB-7) is set.
CRB-1 set will cause the CRB-7 interrupt flag to set on a low-to-high transition of the CB1 pin.
CRB-2 set selects the data register of port B.
CRB-3 and CRB-4 cleared permits CB2 to go high when the interrupt flag bit is set by an active transition of CB1 and to go low after the 6800 *stores* data to the port B data register.
CRB-5 set designates CB2 as an output line.
CRB-6 cleared as a result of CRB-5 being set.
CRB-7 cleared to be used as an interrupt flag for CA1.

Example B-4

Suppose you store the hex number 27 into control register A. How will this control port A? The control register bit structure would be the same as control register B was in Example B-3. However, here CA2 would go high after an interrupt is generated on CA1 and go low after a *read* (load) operation has been performed on the port A data register. Therefore this would provide for complete *data input* handshaking through port A.

Example B-5

Suppose you store the hex number 0F into control register A. How will this control the port? The control register bit structure would be as shown below:

CRA-7	CRA-6	CRA-5	CRA-4	CRA-3	CRA-2	CRA-1	CRA-0
0	0	0	0	1	1	1	1

CRA-0 set will cause an interrupt to be generated when the interrupt flag bit (CRA-7) is set.
CRA-1 set will cause the CRA-7 interrupt flag to set on a low-to-high transition of the CA1 pin.
CRA-2 set selects the data register of port A.
CRA-3 set will cause an interrupt to be generated when the interrupt flag bit (CRA-6) is set.
CRA-4 cleared will cause the CRA-6 interrupt flag bit to set on a high-to-low transition of CA2.
CRA-5 cleared designates CA2 as an input pin.
CRA-6 cleared to be used as an interrupt flag for CA2.
CRA-7 cleared to be used as an interrupt flag for CA1.

Fig. B-13 summarizes the control register bit functions.

REVIEW QUESTIONS

1. List the six internal registers of the PIA.

________, ________, ________, ________, ________, ________.

2. The PIA has ________ programmable data lines. (How many?)

3. Draw the flowchart of the register selection process.

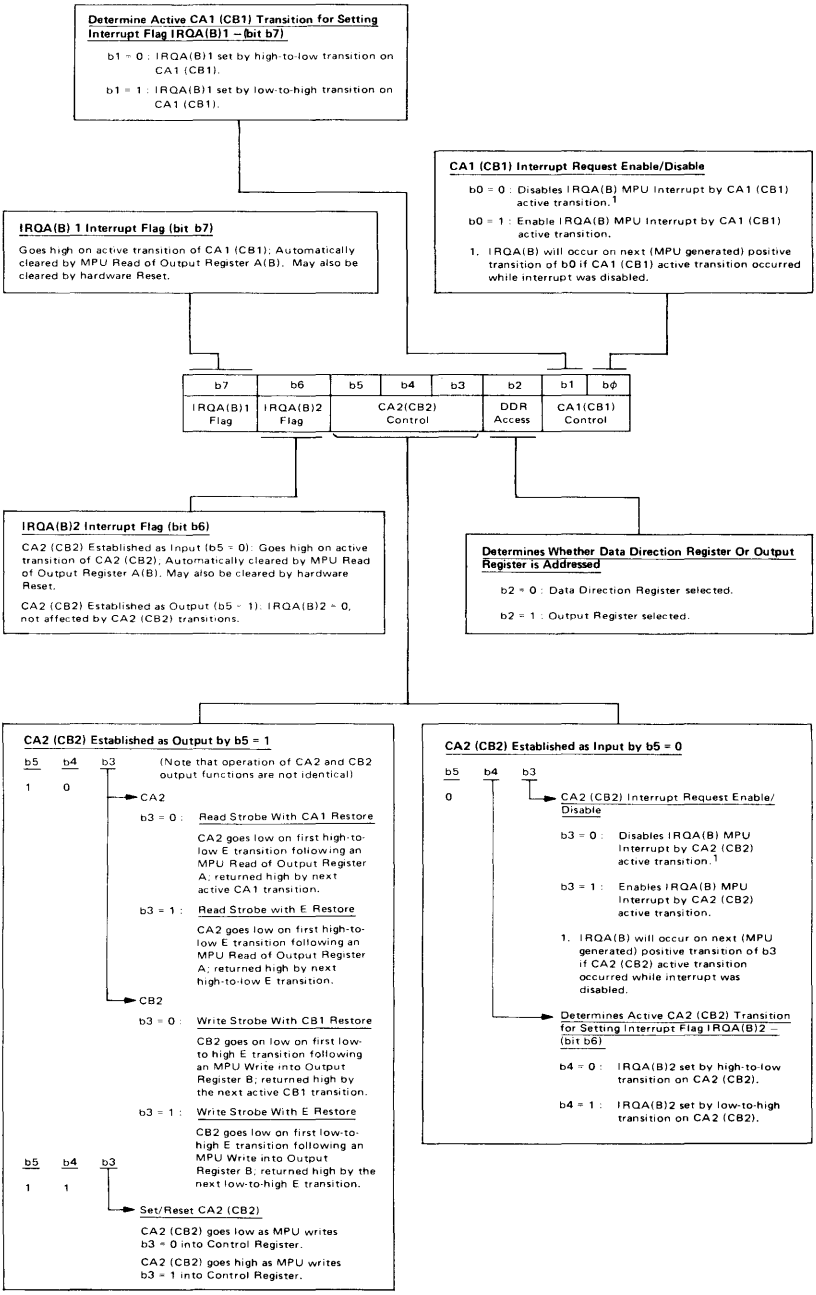
4. What type of integrated-circuit technology is used to manufacture the PIA? ________

5. To configure PA0 through PA3 as input and PA4 through PA7 as output, DDRA must contain ________$_{(16)}$.

6. Port ________ of the PIA can always be used to drive the base of a transistor directly.

7. Two pins on the PIA that *are always* used as interrupt inputs are ________ and ________.

8. Bit ________ of the control register designates CA2 (CB2) as input or output.



Determine Active CA1 (CB1) Transition for Setting Interrupt Flag IRQA(B)1 – (bit b7)

b1 = 0 : IRQA(B)1 set by high-to-low transition on CA1 (CB1).

b1 = 1 : IRQA(B)1 set by low-to-high transition on CA1 (CB1).

CA1 (CB1) Interrupt Request Enable/Disable

b0 = 0 : Disables IRQA(B) MPU Interrupt by CA1 (CB1) active transition.[1]

b0 = 1 : Enable IRQA(B) MPU Interrupt by CA1 (CB1) active transition.

1. IRQA(B) will occur on next (MPU generated) positive transition of b0 if CA1 (CB1) active transition occurred while interrupt was disabled.

IRQA(B) 1 Interrupt Flag (bit b7)

Goes high on active transition of CA1 (CB1); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

b7	b6	b5	b4	b3	b2	b1	bϕ
IRQA(B)1 Flag	IRQA(B)2 Flag	CA2(CB2) Control			DDR Access	CA1(CB1) Control	

IRQA(B)2 Interrupt Flag (bit b6)

CA2 (CB2) Established as Input (b5 = 0): Goes high on active transition of CA2 (CB2); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

CA2 (CB2) Established as Output (b5 = 1): IRQA(B)2 = 0, not affected by CA2 (CB2) transitions.

Determines Whether Data Direction Register Or Output Register is Addressed

b2 = 0 : Data Direction Register selected.

b2 = 1 : Output Register selected.

CA2 (CB2) Established as Output by b5 = 1

b5 b4 b3 (Note that operation of CA2 and CB2 output functions are not identical)

1 0

CA2

b3 = 0 : Read Strobe With CA1 Restore

CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next active CA1 transition.

b3 = 1 : Read Strobe with E Restore

CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next high-to-low E transition.

CB2

b3 = 0 : Write Strobe With CB1 Restore

CB2 goes on low on first low-to high E transition following an MPU Write into Output Register B; returned high by the next active CB1 transition.

b3 = 1 : Write Strobe With E Restore

CB2 goes low on first low-to-high E transition following an MPU Write into Output Register B; returned high by the next low-to-high E transition.

b5 b4 b3

1 1

Set/Reset CA2 (CB2)

CA2 (CB2) goes low as MPU writes b3 = 0 into Control Register.

CA2 (CB2) goes high as MPU writes b3 = 1 into Control Register.

CA2 (CB2) Established as Input by b5 = 0

b5 b4 b3

0

CA2 (CB2) Interrupt Request Enable/Disable

b3 = 0 : Disables IRQA(B) MPU Interrupt by CA2 (CB2) active transition.[1]

b3 = 1 : Enables IRQA(B) MPU Interrupt by CA2 (CB2) active transition.

1. IRQA(B) will occur on next (MPU generated) positive transition of b3 if CA2 (CB2) active transition occurred while interrupt was disabled.

Determines Active CA2 (CB2) Transition for Setting Interrupt Flag IRQA(B)2 – (bit b6)

b4 = 0 : IRQA(B)2 set by high-to-low transition on CA2 (CB2).

b4 = 1 : IRQA(B)2 set by low-to-high transition on CA2 (CB2).

Courtesy Motorola Semiconductor Products, Inc.

Fig. B-13. Control register bit functions.

9. When CA2 (CB2) is used as an interrupt input, bit ________ of the control register is used as the CA2 (CB2) interrupt flag bit.

10. The E (enable) pin of the PIA is usually connected to the ________.

11. A high-to-low transition on the $\overline{\text{Reset}}$ pin of the PIA will cause what to happen?

12. How are the interrupt request pins ($\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$) usually connected?

13. Write an initialization program to configure port A as an output port and and port B as an input port. Assume the PIA is assigned to addresses 8000 through 8003 and a reset has occurred prior to the program execution.

14. Assuming the PIA has been initialized as in problem 12, write a program to input data from port B, and output the *complement* of that data to port A.

15. When would CA2 (CB2) be used as an output pin?

16. Describe what is meant by *complete handshaking.*

17. How would the PIA ports be configured to provide *complete* input and output handshaking?

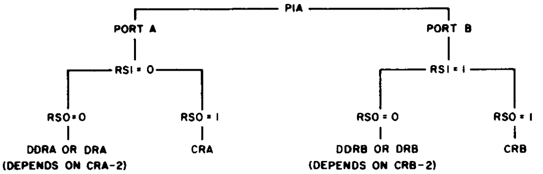
18. Bit 7 of control register A is labeled __________ and what is its function?

19. When CA1 (CB1) and CA2 (CB2) are used as interrupt inputs, bits ________ and ________ of the control register are used to define the active levels of these pins.

20. To provide complete input handshaking, bits 5, 4, and 3 of CRA must be ________ ________ ________.

ANSWERS

1. Data Register A (DRA)
   Data Direction Register A (DDRA)
   Control Register A (CRA)
   Data Register B (DRB)
   Data Direction Register B (DDRB)
   Control Register B (CRB)
2. 16
3.



4. NMOS
5. $F0_{16}$
6. B
7. CA1 and CB1

8. Five
9. Six
10. E clock
11. All internal PIA registers will be cleared.
12. Wire-ORed together, then connected to the 6809 IRQ/FIRQ lines.
13.

```
LDA #
  80
TFR A,DPR
LDD #
  FF
  00
STD $
  00
LDD #
  04
  04
STD $
  02
```

14.

```
LDA $
  01
COMA
STA $
  00
```

15. To provide complete or partial handshaking between the 6809 and a peripheral device.
16. A peripheral device requests service from the 6809; the 6809 acknowledges the request and signals the peripheral device when the service is completed.
17. Port A as an input port with CA1 as an interrupt input line and CA2 as an output peripheral control line. Port B as an output port with CB1 as an interrupt input line and CB2 as an output peripheral control line.
18. Bit 7 of control register A is the CA1 interrupt request flag for port A (IRQA1). It is used as an interrupt flag for interrupts generated on pin CA1.
19. One and four
20. 100

APPENDIX C

Specification Sheets

The following specification sheets are provided through the courtesy of Motorola Semiconductor Products, Inc., Austin, Texas.

MC6809/MC68A09/MC68B09
MC6809E/MC68A09E/MC68B09E
MC6829
MC6839
MC6842
MEK6809EAC
MEK6809D4/MEK68KPD

MC6809/MC68A09/MC68B09

HMOS

(HIGH DENSITY N-CHANNEL, SILICON-GATE)

8-BIT MICROPROCESSING UNIT

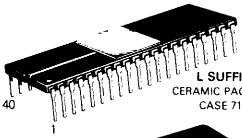
8-BIT MICROPROCESSING UNIT

The MC6809 is a revolutionary high performance 8-bit microprocessor which supports modern programming techniques such as position independence, reentrancy, and modular programming.

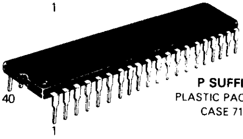
This third-generation addition to the M6800 family has major architectural improvements which include additional registers, instructions and addressing modes.

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809 has the most complete set of addressing modes available on any 8-bit microprocessor today.

The MC6809 has hardware and software features which make it an ideal processor for higher level language execution or standard controller applications.



L SUFFIX
CERAMIC PACKAGE
CASE 715



P SUFFIX
PLASTIC PACKAGE
CASE 711

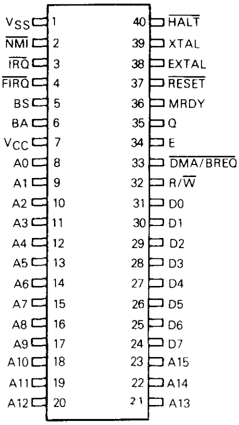
MC6800 COMPATIBLE
- Hardware — Interfaces with All M6800 Peripherals
- Software — Upward Source Code Compatible Instruction Set and Addressing Modes

ARCHITECTURAL FEATURES
- Two 16-bit Index Registers
- Two 16-bit Indexable Stack Pointers
- Two 8-bit Accumulators can be Concatenated to Form One 16-Bit Accumulator
- Direct Page Register Allows Direct Addressing Throughout Memory

HARDWARE FEATURES
- On Chip Oscillator (4 × fo XTAL)
- $\overline{\text{DMA/BREQ}}$ Allows DMA Operation or Memory Refresh
- Fast Interrupt Request Input Stacks Only Condition Code Register and Program Counter
- MRDY Input Extends Data Access Times for Use With Slow Memory
- Interrupt Acknowledge Output Allows Vectoring By Devices
- SYNC Acknowledge Output Allows for Synchronization to External Event
- Single Bus-Cycle RESET
- Single 5-Volt Supply Operation
- NMI Blocked After RESET Until After First Load of Stack Pointer
- Early Address Valid Allows Use With Slower Memories
- Early Write-Data for Dynamic Memories

SOFTWARE FEATURES
- 10 Addressing Modes
  - M6800 Upward Compatible Addressing Modes
  - Direct Addressing Anywhere in Memory Map
  - Long Relative Branches
  - Program Counter Relative
  - True Indirect Addressing
  - Expanded Indexed Addressing:
    0, 5, 8, or 16-bit Constant Offsets
    8, or 16-bit Accumulator Offsets
    Auto-Increment/Decrement by 1 or 2
- Improved Stack Manipulation
- 1464 Instructions with Unique Addressing Modes
- 8 × 8 Unsigned Multiply
- 16-bit Arithmetic
- Transfer/Exchange All Registers
- Push/Pull Any Registers or Any Set of Registers
- Load Effective Address

PIN ASSIGNMENT


V_{SS} 1
$\overline{\text{NMI}}$ 2
$\overline{\text{IRQ}}$ 3
$\overline{\text{FIRQ}}$ 4
BS 5
BA 6
V_{CC} 7
A0 8
A1 9
A2 10
A3 11
A4 12
A5 13
A6 14
A7 15
A8 16
A9 17
A10 18
A11 19
A12 20
40 $\overline{\text{HALT}}$
39 XTAL
38 EXTAL
37 $\overline{\text{RESET}}$
36 MRDY
35 Q
34 E
33 $\overline{\text{DMA/BREQ}}$
32 R/$\overline{\text{W}}$
31 D0
30 D1
29 D2
28 D3
27 D4
26 D5
25 D6
24 D7
23 A15
22 A14
21 A13

MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	Vdc
Input Voltage	V_{in}	−0.3 to +7.0	Vdc
Operating Temperature Range	T_A	0 to +70	°C
Storage Temperature Range	T_{stg}	−55 to +150	°C
Thermal Resistance Ceramic Plastic	θ_{JA}	50 100	°C/W °C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

ELECTRICAL CHARACTERISTICS (V_{CC} = 5.0 V ±5%, V_{SS} = 0, T_A = 0 to 70°C unless otherwise noted)

Characteristic		Symbol	Min	Typ	Max	Unit
Input High Voltage	Logic, EXtal, $\overline{RESET}$	V_{IH}	V_{SS} + 2.0 V_{SS} + 4.0	– –	V_{CC} V_{CC}	Vdc
Input Low Voltage	Logic, EXtal, $\overline{RESET}$	V_{IL}	V_{SS} − 0.3	–	V_{SS} + 0.8	Vdc
Input Leakage Current (V_{in} = 0 to 5.25 V, V_{CC} = max)	Logic	I_{in}	–	1.0	2.5	μAdc
Output High Voltage (I_{Load} = −205 μAdc, V_{CC} = min) (I_{Load} = −145 μAdc, V_{CC} = min) (I_{Load} = −100 μAdc, V_{CC} = min)	D0-D7 A0-A15, $R/\overline{W}$, Q, E BA, BS	V_{OH}	V_{SS} + 2.4 V_{SS} + 2.4 V_{SS} + 2.4	– – –	– – –	Vdc
Output Low Voltage (I_{Load} = 2.0 mAdc, V_{CC} = min)		V_{OL}	–	–	V_{SS} + 0.5	Vdc
Power Dissipation		P_D	–	–	1.0	W
Capacitance # (V_{in} = 0, T_A = 25°C, f = 1.0 MHz)	D0-D7 Logic Inputs, EXtal	C_{in}	– –	10 7	15 10	pF
	A0-A15, $R/\overline{W}$, BA, BS	C_{out}	–	–	12	pF
Frequency of Operation (Crystal or External Input)	MC6809 MC68A09 MC68B09	f_{XTAL}	– – –	– – –	4 6 8	MHz
Three-State (Off State) Input Current (V_{in} = 0.4 to 2.4 V, V_{CC} = max)	D0-D7 A0-A15, $R/\overline{W}$	I_{TSI}	– –	2.0 –	10 100	μAdc

READ/WRITE TIMING (Reference Figures 1, 2, 7, 8, 9, 10, 11, 12, 13 and 14)

Characteristic	Symbol	MC6809			MC68A09			MC68B09			Unit
		Min	Typ	Max	Min	Typ	Max	Min	Typ	Max	
Cycle Time	t_{CYC}	1000	–	–	667	–	–	500	–	–	ns
Total Up Time	t_{UT}	975	–	–	640	–	–	480	–	–	ns
Peripheral Read Access Time $t_{UT} - t_{AD} - t_{DSR} = t_{ACC}$	t_{ACC}	695	–	–	440	–	–	320	–	–	ns
Data Setup Time (Read)	t_{DSR}	80	–	–	60	–	–	40	–	–	ns
Input Data Hold Time	t_{DHR}	10	–	–	10	–	–	40	–	–	ns
Output Data Hold Time	t_{DHW}	30	–	–	30	–	–	30	–	–	ns
Address Hold Time (Address, $R/\overline{W}$)	t_{AH}	20	–	–	20	–	–	20	–	–	ns
Address Delay	t_{AD}	–	–	200	–	–	140	–	–	110	ns
Data Delay Time (Write)	t_{DDW}	–	–	225	–	–	180	–	–	145	ns
E_{low} to Q_{high} Time	t_{AVS}	–	–	250	–	–	165	–	–	125	ns
Address Valid to Q_{high}	t_{AQ}	50	–	–	25	–	–	15	–	–	ns
Processor Clock Low	t_{PWEL}	450	–	–	295	–	–	210	–	–	ns
Processor Clock High	t_{PWEH}	450	–	–	280	–	–	220	–	–	ns
MRDY Set Up Time	t_{PCSM}	125	–	–	125	–	–	125	–	–	ns
Interrupts Set Up Time	t_{PCS}	200	–	–	140	–	–	110	–	–	ns
$\overline{HALT}$ Set Up Time	t_{PCSH}	200	–	–	140	–	–	110	–	–	ns
$\overline{RESET}$ Set Up Time	t_{PCSR}	200	–	–	140	–	–	110	–	–	ns
$\overline{DMA/BREQ}$ Set Up Time	t_{PCSD}	125	–	–	125	–	–	125	–	–	ns
Crystal Osc Start Time	t_{RC}	100	–	–	100	–	–	100	–	–	ms
E Rise and Fall Time	t_{Er}, t_{Ef}	5	–	25	5	–	25	5	–	20	ns
Processor Control Rise/Fall	t_{PCr}, t_{PCf}	–	–	100	–	–	100	–	–	100	ns
Q Rise and Fall Time	t_{Qr}, t_{Qf}	5	–	25	5	–	25	5	–	20	ns
Q Clock High	t_{PWQH}	450	–	–	280	–	–	220	–	–	ns

FIGURE 1 — READ DATA FROM MEMORY OR PERIPHERALS

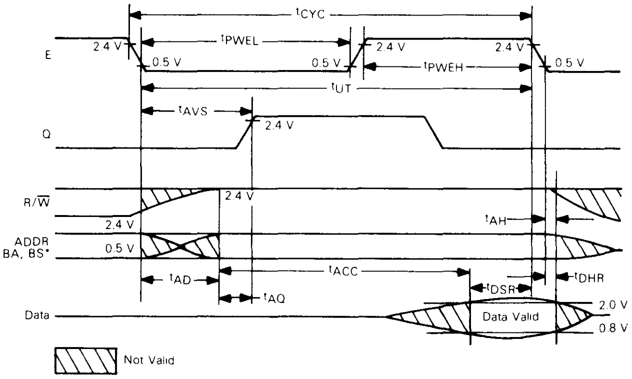
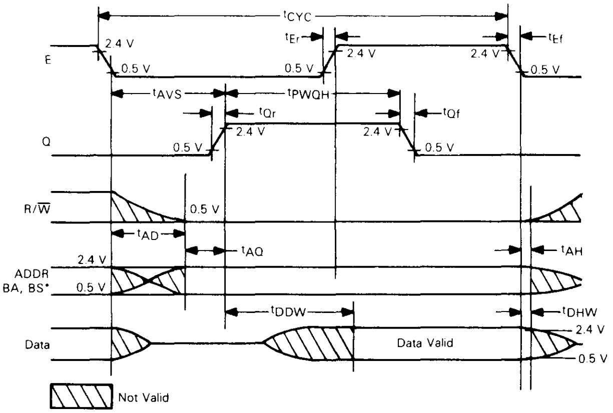


FIGURE 2 — WRITE DATA TO MEMORY OR PERIPHERALS



*Hold time for BA, BS not specified

FIGURE 3 — MC6809 EXPANDED BLOCK DIAGRAM

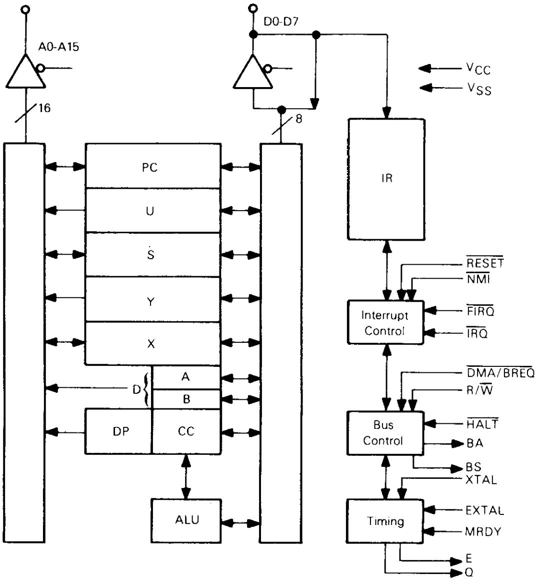
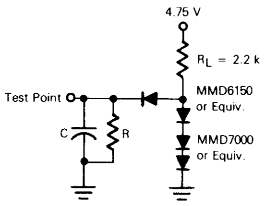


FIGURE 4 — BUS TIMING TEST LOAD



C = 30 pF for BA, BS
130 pF for D0-D7, E, Q
90 pF for A0-A15, R/W

R = 11.7 kΩ for D0-D7
16.5 kΩ for A0-A15, E, Q, R/W
24 Ω for BA, BS

PROGRAMMING MODEL

As shown in Figure 5, the MC6809 adds three registers to the set available in the MC6800. The added registers include a Direct Page Register, the User Stack pointer and a second Index Register.

ACCUMULATORS (A, B, D)

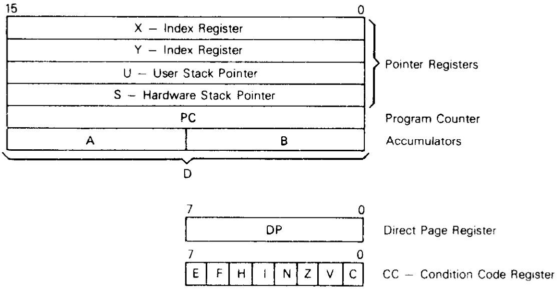
The A and B registers are general purpose accumulators which are used for arithmetic calculations and manipulation of data.

Certain instructions concatenate the A and B registers to form a single 16-bit accumulator. This is referred to as the D Register, and is formed with the A Register as the most significant byte.

DIRECT PAGE REGISTER (DP)

The Direct Page Register of the MC6809 serves to enhance the Direct Addressing Mode. The content of this register appears at the higher address outputs (A8-A15) during direct Addressing Instruction execution. This allows the direct mode to be used at any place in memory, under program control. To ensure 6800 compatibility, all bits of this register are cleared during Processor Reset.

FIGURE 5 — PROGRAMMING MODEL OF THE MICROPROCESSING UNIT



INDEX REGISTERS (X, Y)

The Index Registers are used in indexed mode of addressing. The 16-bit address in this register takes part in the calculation of effective addresses. This address may be used to point to data directly or may be modifed by an optional constant or register offset. During some indexed modes, the contents of the index register are incremented and decremented to point to the next item of tabular type data. All four pointer registers (X, Y, U, S) may be used as index registers.

STACK POINTER (U, S)

The Hardware Stack Pointer (S) is used automatically by the processor during subroutine calls and interrupts. The stack pointers of the MC6809 point to the top of the stack, in contrast to the MC6800 stack pointer, which pointed to the next free location on the stack. The User Stack Pointer (U) is controlled exclusively by the programmer thus allowing arguments to be passed to and from subroutines with ease. Both Stack Pointers have the same indexed mode addressing capabilities as the X and Y registers, but also support **Push** and **Pull** instructions. This allows the MC6809 to be used efficiently as a stack processor, greatly enhancing its ability to support higher level languages and modular programming.

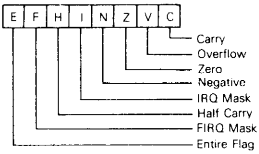
PROGRAM COUNTER

The Program Counter 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 Program Counter to be used like an index register in some situations.

CONDITION CODE REGISTER

The Condition Code Register defines the State of the Processor at any given time. See Figure 6.

FIGURE 6 — CONDITION CODE REGISTER FORMAT



CONDITION CODE REGISTER DESCRIPTION

BIT 0 (C)

Bit 0 is the carry flag, and is usually the carry from the binary ALU. C is also used to represent a 'borrow' from subtract like instructions (CMP, NEG, SUB, SBC) and is the complement of the carry from the binary ALU.

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.

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.

BIT 3 (N)

Bit 3 is the negative flag, which contains exactly the value of the MSB of the result of the preceding operation. Thus, a negative two's-complement result will leave N set to a one.

BIT 4 (I)

Bit 4 is the $\overline{\mathrm{IRQ}}$ mask bit. The processor will not recognize interrupts from the $\overline{\mathrm{IRQ}}$ line if this bit is set to a one. $\overline{\mathrm{NMI}}$, $\overline{\mathrm{FIRQ}}$, $\overline{\mathrm{IRQ}}$, $\overline{\mathrm{RESET}}$, and SWI all are set I to a one; SWI2 and SWI3 do not affect I.

BIT 5 (H)

Bit 5 is the half-carry bit, and 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 operation. The state of this flag is undefined in all subtract-like instructions.

BIT 6 (F)

Bit 6 is the $\overline{\mathrm{FIRQ}}$ mask bit. The processor will not recognize interrupts from the $\overline{\mathrm{FIRQ}}$ line if this bit is a one. $\overline{\mathrm{NMI}}$, $\overline{\mathrm{FIRQ}}$, SWI, and $\overline{\mathrm{RESET}}$ all set F to a one. $\overline{\mathrm{IRQ}}$, SWI2 and SWI3 do not affect F.

BIT 7 (E)

Bit 7 is the entire flag, and when set to a one indicates that the complete machine state (all the registers) was stacked, as opposed to the subset state (PC and CC). The E bit of the stacked CC is used on a return from interrupt (RTI) to determine the extent of the unstacking. Therefore, the current E left in the Condition Code Register represents past action.

MC6809 MPU SIGNAL DESCRIPTION

POWER (V_{SS}, V_{CC})

Two pins are used to supply power to the part: V_{SS} is ground or 0 volts, while V_{CC} is +5.0 V ±5%.

ADDRESS BUS (A0-A15)

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/$\overline{\mathrm{W}}$ = 1, and BS = 0; this is a "dummy access" or $\overline{\mathrm{VMA}}$ cycle. Addresses are valid on the rising edge of Q (see Figures 1 and 2). All address bus drivers are made high-impedance when output Bus Available (BA) is high. Each pin will drive one Schottky TTL load or four LS TTL loads, and typically 90 pF.

DATA BUS (D0-D7)

These eight pins provide communication with the system bi-directional data bus. Each pin will drive one Schottky TTL load or four LS TTL loads, and typically 130 pF.

READ/WRITE (R/$\overline{\mathrm{W}}$)

This signal indicates the direction of data transfer on the data bus. A low indicates that the MPU is writing data onto the data bus. R/$\overline{\mathrm{W}}$ is made high impedance when BA is high. R/$\overline{\mathrm{W}}$ is valid on the rising edge of Q. Refer to Figures 1 and 2.

$\overline{\mathrm{RESET}}$

A low level on this Schmitt-trigger input for greater than one bus cycle will reset the MPU, as shown in Figure 7. The Reset vectors are fetched from locations $FFFE_{16}$ and $FFFF_{16}$ (Table 1) when Interrupt Acknowledge is true, ($\overline{\mathrm{BA}}$ • BS = 1). During initial power-on, the Reset line should be held low until the clock oscillator is fully operational. See Figure 8.

Because the MC6809 Reset pin has a Schmitt-trigger input with a threshold voltage higher than that of standard peripherals, a simple R/C network may be used to reset the entire system. This higher threshold voltage ensures that all peripherals are out of the reset state before the Processor.

$\overline{\mathrm{HALT}}$

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 ($\overline{\mathrm{FIRQ}}$, $\overline{\mathrm{IRQ}}$) although $\overline{\mathrm{DMA/BREQ}}$ will always be accepted, and $\overline{\mathrm{NMI}}$ or $\overline{\mathrm{RESET}}$ will be latched for later response. During the Halt state Q and E continue to run normally. If the MPU is not running ($\overline{\mathrm{RESET}}$, $\overline{\mathrm{DMA/BREQ}}$), a halted state (BA•BS = 1) can be achieved by pulling $\overline{\mathrm{HALT}}$ low while $\overline{\mathrm{RESET}}$ is still low. If $\overline{\mathrm{DMA/BREQ}}$ and $\overline{\mathrm{HALT}}$ 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. See Figure 9.

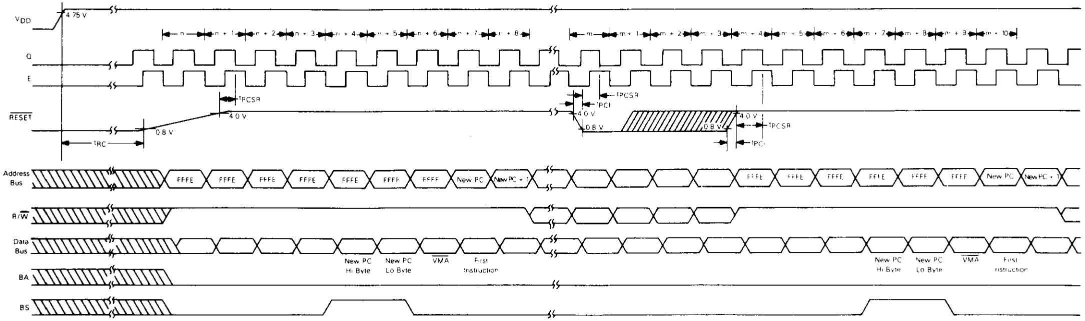
BUS AVAILABLE, BUS STATUS (BA, BS)

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).

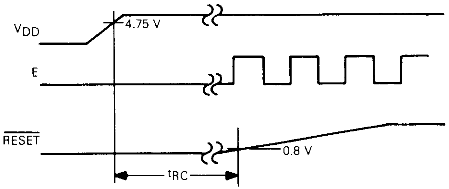
MPU State		MPU State Definition
BA	BS	
0	0	Normal (Running)
0	1	Interrupt or RESET Acknowledge
1	0	SYNC Acknowledge
1	1	HALT or Bus Grant

FIGURE 7 — $\overline{\text{RESET}}$ TIMING



*Note: Parts with data codes prefixed by 7F will come out of RESET one cycle sooner than shown.

FIGURE 8 — CRYSTAL CONNECTIONS AND OSCILLATOR START UP



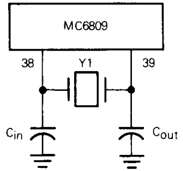
MC6809 Nominal Crystal Parameters*

	3.58 MHz	4.00 MHz	6.0 MHz	8.0 MHz
RS	60 Ω	50 Ω	30-50 Ω	20-40 Ω
C_0	3.5 pF	6.5 pF	4-6 pF	4-6 pF
C_1	0.015 pF	0.025 pF	0.01-0.02 pF	0.01-0.02 pF
Q	>40K	>30 K	>20 K	>20 K

All parameters are 10%

*NOTE: These are representative AT-cut crystal parameters only. Crystals of other types of cut may also be used.

Y1	C_{in}	C_{out}
8 MHz	18 pF	18 pF
6 MHz	20 pF	20 pF
4 MHz	24 pF	24 pF



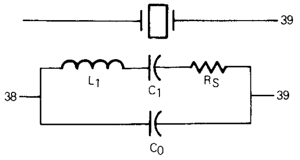
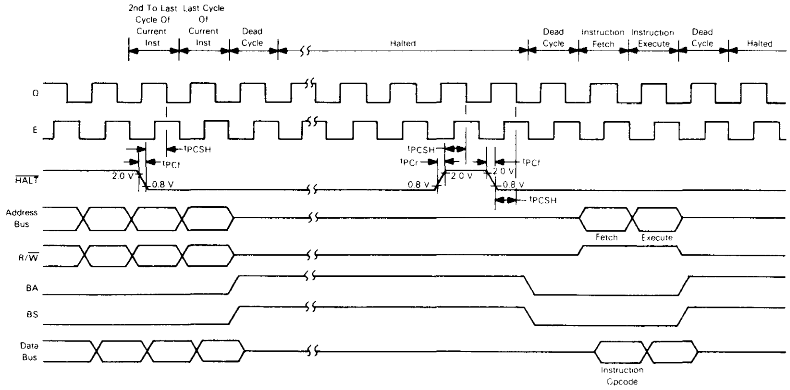


FIGURE 9 — HALT AND SINGLE INSTRUCTION EXECUTION FOR SYSTEM DEBUG



Interrupt Acknowledge is indicated during both cycles of a hardware-vector-fetch (RESET, NMI, FIRQ, IRQ, SWI, SWI2, SWI3). This signal, plus decoding of the lower four address lines, can provide the user with an indication of which interrupt level is being serviced and allow vectoring by device. See Table 1.

Sync Acknowledge is indicated while the MPU is waiting for external synchronization on an interrupt line.

Halt/Bus Grant is true when the MC6809 is in a Halt or Bus Grant condition.

TABLE 1: MEMORY MAP FOR INTERRUPT VECTORS

Memory Map For Vector Locations		Interrupt Vector Description
MS	LS	
FFFE	FFFF	RESET
FFFC	FFFD	NMI
FFFA	FFFB	SWI
FFF8	FFF9	IRQ
FFF6	FFF7	FIRQ
FFF4	FFF5	SWI2
FFF2	FFF3	SWI3
FFF0	FFF1	Reserved

NON MASKABLE INTERRUPT (NMI)*

A negative edge on this input requests that a non-maskable interrupt sequence be generated. A non-maskable interrupt cannot be inhibited by the program, and also has a higher priority than FIRQ, IRQ or software interrupts. During recognition of an NMI, the entire machine state is saved on the hardware stack. After reset, an NMI will not be recognized until the first program load of the Hardware Stack Pointer (S). The pulse width of NMI low must be at least one E cycle. If the NMI input does not meet the minimum set up with respect to Q, the interrupt will not be recognized until the next cycle. See Figure 10.

FAST-INTERRUPT REQUEST (FIRQ)*

A low level on this input pin will initiate a fast interrupt sequence, provided its mask bit (F) in the CC is clear. This sequence has priority over the standard Interrupt Request (IRQ), and is fast in the sense that it stacks only the contents of the condition code register and the program counter. The interrupt service routine should clear the source of the interrupt before doing an RTI. See Figure 11.

INTERRUPT REQUEST (IRQ)*

A low level input on this pin will initiate an Interrupt Request sequence provided the mask bit (I) in the CC is clear. Since IRQ stacks the entire machine state it provides a slower response to interrupts than FIRQ. IRQ also has a lower priority than FIRQ. Again, the interrupt service routine should clear the source of the interrupt before doing an RTI. See Figure 10.

***NOTE:** NMI, FIRQ and IRQ requests are latched by the falling edge of every Q, except during cycle stealing operations (e.g., DMA) where only NMI is latched. From this point, a delay of at least one bus cycle will occur before the interrupt is serviced by MPU.

FIGURE 10 — $\overline{\text{IRQ}}$ AND $\overline{\text{NMI}}$ INTERRUPT TIMING

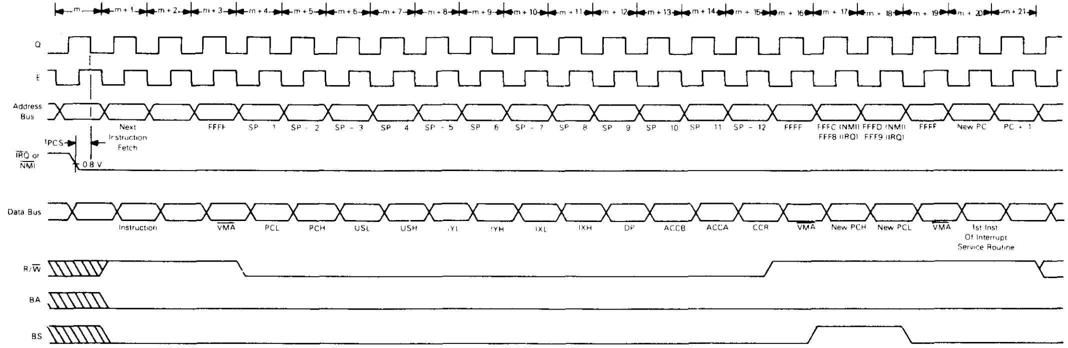
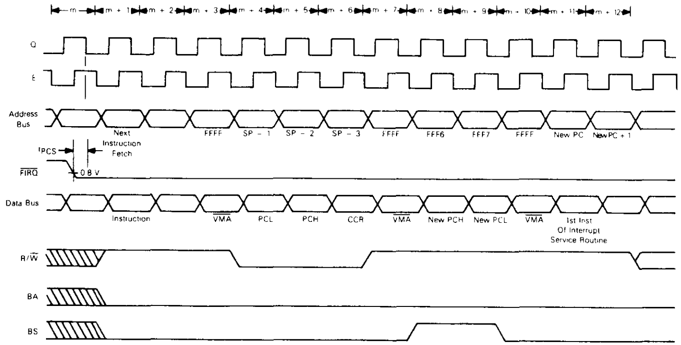


FIGURE 11 — $\overline{\text{FIRQ}}$ INTERRUPT TIMING



XTAL, EXTAL

These inputs are used to connect the on-chip oscillator to an external parallel-resonant crystal. Alternately, the pin EXTAL may be used as a TTL level input for external timing by grounding XTAL. The crystal or external frequency is four times the bus frequency. See Figure 8. Proper RF layout techniques should be observed in the layout of printed circuit boards.

E, Q

E is similar to the MC6800 bus timing signal $\phi 2$; Q is a quadrature clock signal which leads E. Q has no parallel on the MC6800. Addresses from the MPU will be valid with the leading edge of Q. Data is latched on the falling edge of E. Timing for E and Q is shown in Figure 12.

MRDY

This input control signal allows stretching of E and Q to extend data-access time. E and Q operate normally while MRDY is high. When MRDY is low, E and Q may be stretched in integral multiples of quarter (1/4) bus cycles, thus allowing interface to slow memories, as shown in Figure 13(A). A maximum stretch is 10 microseconds. During non-valid memory access (VMA cycles) MRDY has no effect on stretching E and Q; this inhibits slowing the processor during "don't care" bus accesses. MRDY may also be used to stretch clocks (for slow memory) when bus control has been transferred to an external device (through the use of HALT and DMA/BREQ).

NOTE: Four of the early production mask sets (G7F, T5A, P6F, T6M) require synchronization of the MRDY input with the 4f clock. The synchronization necessitates an external oscillator as shown in Figure 13 (B). The negative transition of the MRDY signal, normally derived from the chip select decoding, must meet the t_{PCSM} timing. With these four mask sets, MRDY's positive transition must occur with the rising edge of 4f.

In addition, on these same mask sets, MRDY will not stretch the E and Q signals if the machine is executing either a TFR or EXG instruction during the HALT high to low transition. If the MPU executes a CWAI instruction, the machine pushes the internal registers onto the stack and then awaits an interrupt. During this waiting period, it is possible to place the MPU into a Halt mode to three-state the machine, but MRDY will not stretch the clocks.

DMA/BREQ

The DMA/BREQ input provides a method of suspending execution and acquiring the MPU bus for another use, as shown in Figure 14. Typical uses include DMA and dynamic memory refresh.

Transition of 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 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 for self-refresh. Self-refresh requires one bus cycle with a leading and trailing dead cycle. See Figure 15.

Typically, the DMA controller will request to use the bus by asserting DMA/BREQ pin low on the leading edge of E. When the MPU replies by setting BA and BS to a one, that cycle will be a dead cycle used to transfer bus mastership to the DMA controller.

False memory accesses may be prevented during any dead cycles by developing a system DMAVMA signal which is LOW in any cycle when BA has changed.

When BA goes low (either as a result of DMA/BREQ = HIGH or MPU self-refresh), the DMA device should be taken off the bus. Another dead cycle will elapse before the MPU accesses memory, to allow transfer of bus mastership without contention.

MPU OPERATION

During normal operation, the MPU fetches an instruction from memory and then executes the requested function. This sequence begins at RESET and is repeated indefinitely unless altered by a special instruction or hardware occurrence. Software instructions that alter normal MPU operation are: SWI, SWI2, SWI3, CWAI, RTI and SYNC. An interrupt, HALT or DMA/BREQ can also alter the normal execution of instructions. Figure 16 illustrates the flow chart for the MC6809.

FIGURE 12 — E/Q RELATIONSHIP

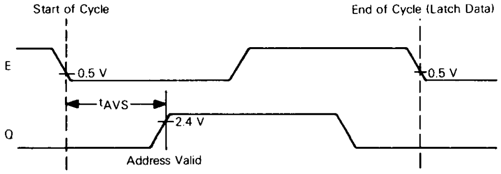


FIGURE 13(A) — MRDY TIMING

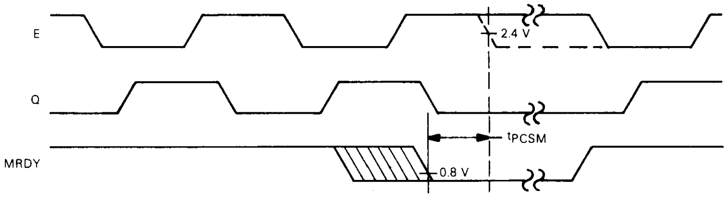


FIGURE 13(B) — MC6809 MRDY SYNCHRONIZATION

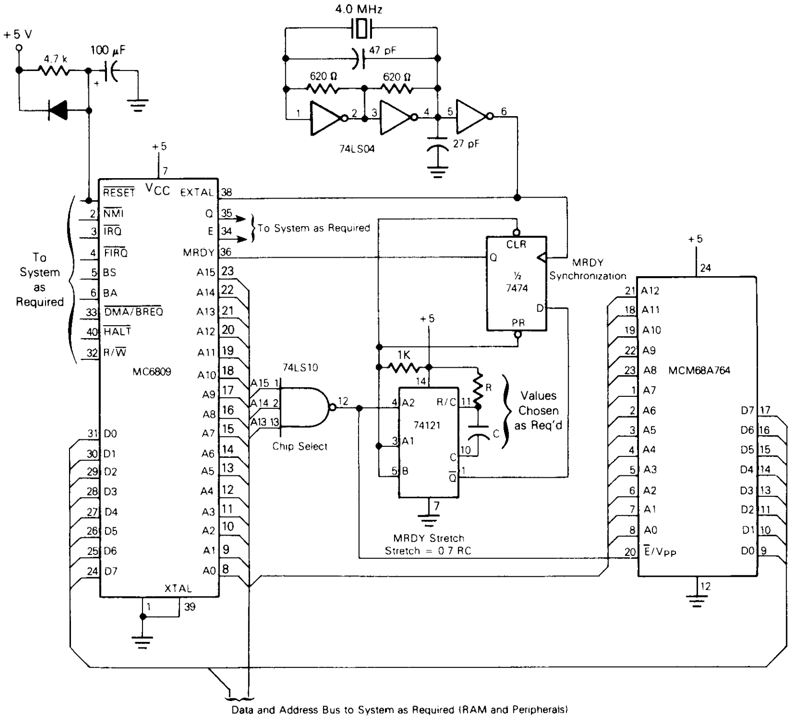


FIGURE 14 — TYPICAL DMA TIMING (< 14 CYCLES)

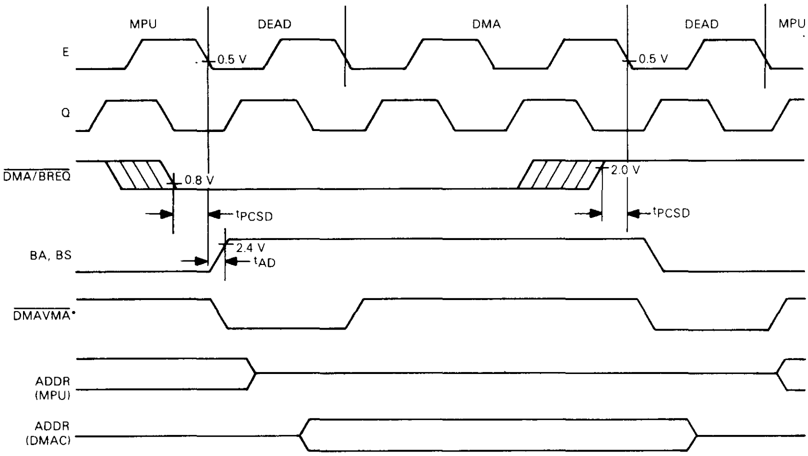
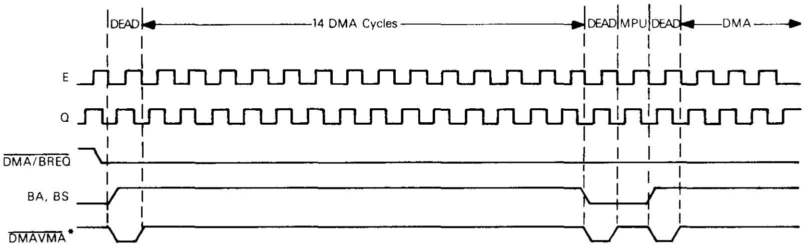
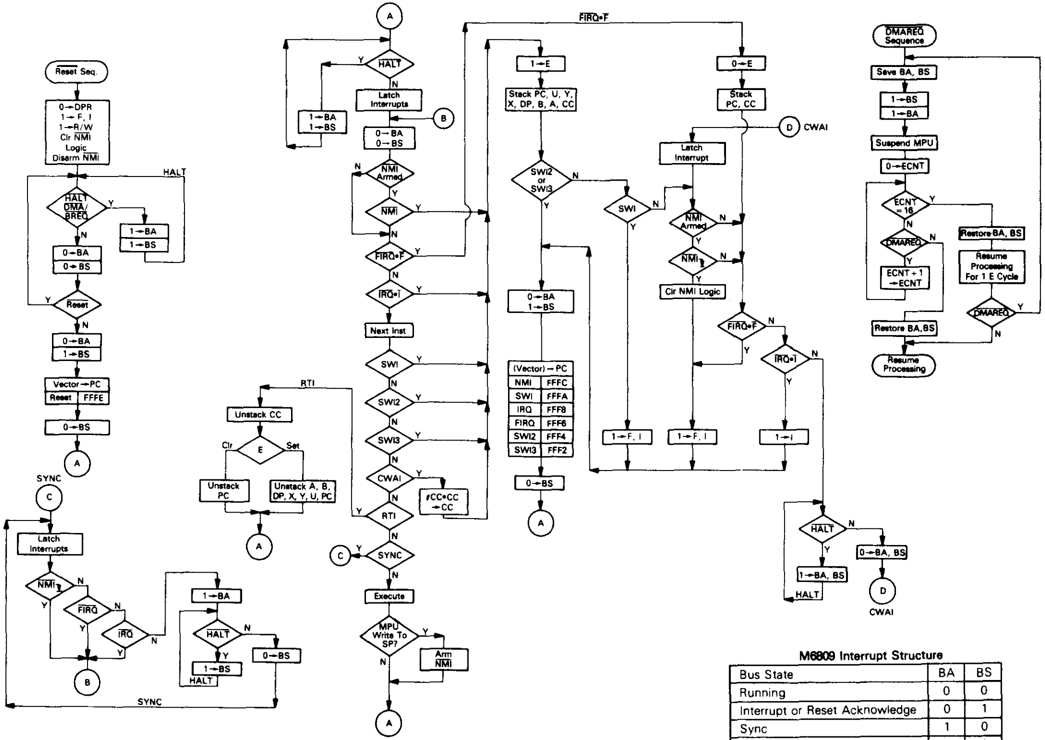


FIGURE 15 — AUTO-REFRESH DMA TIMING (> 14 CYCLES)
(REVERSE CYCLE STEALING)



*DMAVMA is a signal which is developed externally, but is a system requirement for DMA.

FIGURE 16 — FLOWCHART FOR 6809 INSTRUCTIONS



NOTE: Asserting RESET will result in entering the reset sequence from any point in the flow chart.

M6809 Interrupt Structure

Bus State	BA	BS
Running	0	0
Interrupt or Reset Acknowledge	0	1
Sync	1	0
Halt/Bus Grant	1	1

ADDRESSING MODES

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809 has the most complete set of addressing modes available on any microcomputer today. For example, the MC6809 has 59 basic instructions; however, it recognizes 1464 different variations of instructions and addressing modes. The addressing modes support modern programming techniques. The following addressing modes are available on the MC6809:

- Inherent (Includes Accumulator)
- Immediate
- Extended
  - Extended Indirect
- Direct
- Register
- Indexed
  - Zero-Offset
  - Constant Offset
  - Accumulator Offset
  - Auto Increment/Decrement
  - Indexed Indirect
- Relative
  - Short/Long Relative Branching
  - Program Counter Relative Addressing

INHERENT (INCLUDES ACCUMULATOR)

In this addressing mode, the opcode of the instruction contains all the address information necessary. Examples of Inherent Addressing are: ABX, DAA, SWI, ASRA, and CLRB.

IMMEDIATE ADDRESSING

In Immediate Addressing, the effective address of the data is the location immediately following the opcode (i.e., the data to be used in the instruction immediately follows the opcode of the instruction). The MC6809 uses both 8 and 16-bit immediate values depending on the size of argument specified by the opcode. Examples of instructions with Immediate Addressing are:

```
LDA  #$20
LDX  #$F000
LDY  #CAT
```

NOTE: # signifies Immediate addressing, $ signifies hexadecimal value.

EXTENDED ADDRESSING

In Extended Addressing, the contents of the two bytes immediately following the opcode fully specify the 16-bit effective address used by the instruction. Note that the address generated by an extended instruction defines an absolute address and is not position independent. Examples of Extended Addressing include:

```
LDA  CAT
STX  MOUSE
LDD  $2000
```

EXTENDED INDIRECT

As a special case of indexed addressing (discussed below), 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.

```
LDA  [CAT]
LDX  [$FFFE]
STU  [DOG]
```

DIRECT ADDRESSING

Direct addressing is similar to extended addressing except that only one byte of address follows the opcode. This byte specifies the lower 8 bits of the address to be used. The upper 8 bits of the address are supplied by the direct page register. Since only one byte of address is required in direct addressing, this mode requires less memory and executes faster than extended addressing. Of course, only 256 locations (one page) can be accessed without redefining the contents of the DP register. Since the DP register is set to $00 on Reset, direct addressing on the MC6809 is compatible with direct addressing on the M6800. Indirection is not allowed in direct addressing. Some examples of direct addressing are:

```
LDA    $30
SETDP $10 (Assembler directive)
LDB    $1030
LDD    < CAT
```

NOTE: < is an assembler directive which forces direct addressing.

REGISTER ADDRESSING

Some opcodes are followed by a byte that defines a register or set of registers to be used by the instruction. This is called a postbyte. Some examples of register addressing are:

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

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 of the operand to be used by the instruction. Five basic types of indexing are available and are discussed below. 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. Figure 17 lists the legal formats for the postbyte. Table 2 gives the assembler form and the number of cycles and bytes added to the basic values for indexed addressing for each variation.

FIGURE 17 — INDEXED ADDRESSING POSTBYTE REGISTER 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 + 5 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 = ,R +8 Bit Offset
1	R	R	I	1	0	0	1	EA = ,R +16 Bit Offset
1	R	R	I	1	0	1	1	EA = ,R +D Offset
1	x	x	I	1	1	0	0	EA = ,PC +8 Bit Offset
1	x	x	I	1	1	0	1	EA = ,PC +16 Bit Offset
1	R	R	1	1	1	1	1	EA = [,Address]

Addressing Mode Field

Indirect Field
(Sign bit when b_7 = 0)

Register Field: RR
00 = X
01 = Y
10 = U
11 = S

x = Don't Care

Zero-Offset Indexed — In this mode, the selected pointer register contains the effective address of the data to be used by the instruction. This is the fastest indexing mode.

Examples are:

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

Constant Offset Indexed — In this mode, a two's-complement offset and the contents of one of the pointer registers are added to form the effective address of the operand. The pointer register's initial content is unchanged by the addition.

Three sizes of offsets are available:

5 -bit (– 16 to + 15)
8 -bit (– 128 to + 127)
16-bit (– 32768 to + 32767)

The two's complement 5-bit offset is included in the postbyte and, therefore, is most efficient in use of bytes and cycles. The two's complement 8-bit offset is contained in a single byte following the postbyte. The two's complement 16-bit offset is in the two bytes following the postbyte. In most cases the programmer need not be concerned with the size of this offset since the assembler will select the optimal size automatically.

Examples of constant-offset indexing are:

```
LDA    23,X
LDX    -2,S
LDY    300,X
LDU    CAT,Y
```

TABLE 2 — INDEXED ADDRESSING MODE

Type	Forms	Non Indirect				Indirect			
		Assembler Form	Postbyte OP Code	+ ~	+ #	Assembler Form	Postbyte OP Code	+ ~	+ #
Constant Offset From R (2's Complement Offsets)	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 (2's Complement Offsets)	A Register Offset	A, R	1RR00110	1	0	[A, R]	1RR10110	4	0
	B Register Offset	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 (2's Complement Offsets)	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

RR:
00 = X
01 = Y
10 = U
11 = S

$\overset{+}{\sim}$ and $\overset{+}{\#}$ indicate the number of additional cycles and bytes for the particular variation.

Accumulator-Offset Indexed — 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 contents of one of the pointer registers are added to form the effective address of the operand. The contents of both the accumulator and the pointer register are unchanged by the addition. The postbyte specifies which accumulator to use as an offset and no additional bytes are required. The advantage of an accumulator offset is that the value of the offset can be calculated by a program at run-time.

Some examples are:

```
LDA    B,Y
LDX    D,Y
LEAX   B,X
```

Auto Increment/Decrement Indexed — In the auto increment addressing mode, the pointer register contains the address of the operand. Then, after the pointer register is used it is incremented by one or two. This addressing mode is useful in stepping through tables, moving data, or for the creation of software stacks. In auto decrement, the pointer register is decremented prior to use as the address of the data. The use of auto decrement is similar to that of auto increment; but the tables, etc., are scanned from the high to low addresses. The size of the increment/decrement can be either one or two to allow for tables of either 8 or 16-bit data to be accessed and is selectable by the programmer. The pre-decrement, post-increment nature of these modes allow them to be used to create additional software stacks that behave identically to the U and S stacks.

Some examples of the auto increment/decrement addressing modes are:

```
LDA    ,X+
STD    ,Y++
LDB    ,-Y
LDX    ,--S
```

INDEXED INDIRECT

All of the indexing modes with the exception of auto increment/decrement by one, or a ±4-bit offset may have an additional level of indirection specified. In indirect adddressing, the effective address is contained at the location specified by the contents of the Index register plus any offset. In the example below, the A accumulator is loaded indirectly using an effective address calculated from the Index register and an offset.

```
       Before Execution
       A=XX (don't care)
       X=$F000
$0100  LDA   [$10,X]      EA is now $F010

$F010  $F1                $F150 is now the
$F011  $50                new EA

$F150  $AA
       After Execution
       A=$AA Actual Data Loaded
       X=$F000
```

All modes of indexed indirect are included except those which are meaningless (e.g., auto increment/decrement by 1 indirect). Some examples of indexed indirect are:

```
LDA    [,X]
LDD    [10,S]
LDA    [B,Y]
LDD    [,X++]
```

RELATIVE ADDRESSING

The byte(s) following the branch opcode is (are) treated as a signed offset which may be added to the program counter. If the branch condition is true then the calculated address (PC + signed offset) is loaded into the program counter. Program execution continues at the new location as indicated by the PC; short (1 byte offset) and long (2 bytes offset) relative addressing modes are available. All of memory can be reached in long relative addressing as an effective address is interpreted modulo 2^{16}. Some examples of relative addressing are:

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

PROGRAM COUNTER RELATIVE

The PC can be used as the pointer register with 8 or 16-bit signed offsets. As in relative addressing, the offset is added to the current PC to create the effective address. The effective address is then used as the address of the operand or data. Program Counter Relative Addressing is used for writing position independent programs. Tables related to a particular routine will maintain the same relationship after the routine is moved, if referenced relative to the Program Counter. Examples are:

```
LDA    CAT, PCR
LEAX   TABLE, PCR
```

Since program counter relative is a type of indexing, an additional level of indirection is available.

```
LDA    [CAT, PCR]
LDU    [DOG, PCR]
```

MC6809 INSTRUCTION SET

The instruction set of the MC6809 is similar to that of the MC6800 and is upward compatible at the source code level. The number of opcodes has been reduced from 72 to 59, but because of the expanded architecture and additional addressing modes, the number of available opcodes (with different addressing modes) has risen from 197 to 1464.

Some of the new instructions and addressing modes are described in detail below:

PSHU/PSHS

The push instructions have the capability of pushing onto either the hardware stack (S) or user stack (U) any single register, or set of registers with a single instruction.

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, as shown in below.

```
        ← Pull Order     Push Order→
PC     U     Y     X     DP     B     A     CC
FFFF.....← increasing memory address.....0000
PC     S     Y     X     DP     B     A     CC
```

TFR/EXG

Within the MC6809, any register may be transferred to or exchanged with another of like-size; i.e., 8-bit to 8-bit or 16-bit to 16-bit. Bits 4-7 of postbyte define the source register, while bits 0-3 represent the destination register. These are denoted as follows:

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

NOTE: All other combinations are undefined and INVALID.

LEAX/LEAY/LEAU/LEAS

The LEA (Load Effective Address) works by calculating the effective address used in an indexed instruction and stores that address value, rather than the data at that address, in a pointer register. This makes all the features of the internal addressing hardware available to the programmer. Some of the implications of this instruction are illustrated in Table 3.

The LEA instruction also allows the user to access data in a position independent manner. For example:

```
        LEAX    MSG1, PCR
        LBSR    PDATA (Print message routine)
          •
          •
MSG1 FCC    'MESSAGE'
```

This sample program prints: 'MESSAGE'. By writing MSG1, PCR, the assembler computes the distance between the present address and MSG1. This result is placed as a constant into the LEAX instruction which will be indexed from the PC value at the time of execution. No matter where the code is located, when it is executed, the computed offset from the PC will put the absolute address of MSG1 into the X pointer register. This code is totally position independent.

MUL

Multiplies the unsigned binary numbers in the A and B accumulator and places the unsigned result into the 16-bit D accumulator. This unsigned multiply also allows multiple-precision multiplications.

Long And Short Relative Branches

The MC6809 has the capability of program counter relative branching throughout the entire memory map. In this mode, if the branch is to be taken, the 8 or 16-bit signed offset is added to the value of the program counter to be used as the effective address. This allows the program to branch anywhere in the 64K memory map. Position independent code can be easily generated through the use of relative branching. Both short (8-bit) and long (16-bit) branches are available.

TABLE 3 — LEA EXAMPLES

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

SYNC

After encountering a Sync instruction, the MPU enters a Sync state, stops processing instructions and waits for an interrupt. If the pending interrupt is non-maskable (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 service routine. Since FIRQ and IRQ are not edge-triggered, a low level with a minimum duration of three bus cycles is required to assure that the interrupt will be taken. If the pending interrupt is maskable (FIRQ, IRQ) with its mask bit (F or I) set, the processor will clear the Sync state and continue processing by executing the next inline instruction. Figure 18 depicts Sync timing.

Software Interrupts

A Software Interrupt is an instruction which will cause an interrupt, and its associated vector fetch. These Software Interrupts are useful in operating system calls, software debugging, trace operations, memory mapping, and software development systems. Three levels of SWI are available on this MC6809, and are prioritized in the following order: SWI, SWI2, SWI3.

16-Bit Operation

The MC6809 has the capability of processing 16-bit data. These instructions include loads, stores, compares, adds, subtracts, transfers, exchanges, pushes and pulls.

CYCLE-BY-CYCLE OPERATION

The address bus cycle-by-cycle performance chart illustrates the memory-access sequence corresponding to each possible instruction and addressing mode in the MC6809. Each instruction begins with an opcode fetch. While that opcode is being internally decoded, the next program byte is always fetched. (Most instructions will use the next byte, so this technique considerably speeds throughput.) Next, the operation of each opcode will follow the flow chart. $\overline{\text{VMA}}$ is an indication of FFFF$_{16}$ on the address bus, $R/\overline{W}$ = 1 and BS = 0. The following examples illustrate the use of the chart; see Figure 19.

LBSR	(Branch taken)
Cycle # 1	opcode Fetch
2	opcode +
3	opcode +
4	$\overline{\text{VMA}}$
5	$\overline{\text{VMA}}$
6	ADDR
7	$\overline{\text{VMA}}$
8	STACK (write)
9	STACK (write)

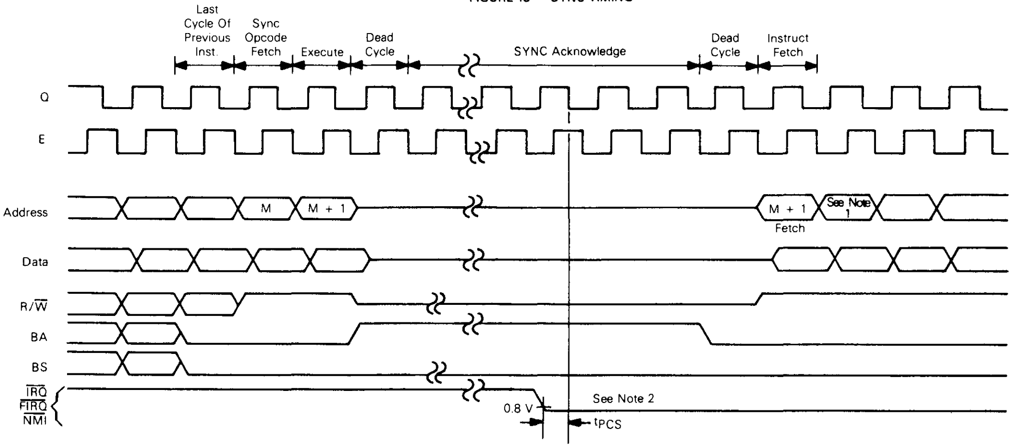
DEC	(Extended)
Cycle # 1	opcode Fetch
2	opcode +
3	opcode +
4	$\overline{\text{VMA}}$
5	ADDR (read)
6	$\overline{\text{VMA}}$
7	ADDR (write)

MC6809 INSTRUCTION SET TABLES

The instructions of the MC6809 have been broken down into five different categories. They are as follows:

8-Bit operation (Table 4)
16-Bit operation (Table 5)
Index register/stack pointer instructions (Table 6)
Relative branches (long or short) (Table 7)
Miscellaneous instructions (Table 8)
Hexadecimal Values of instructions (Table 9)

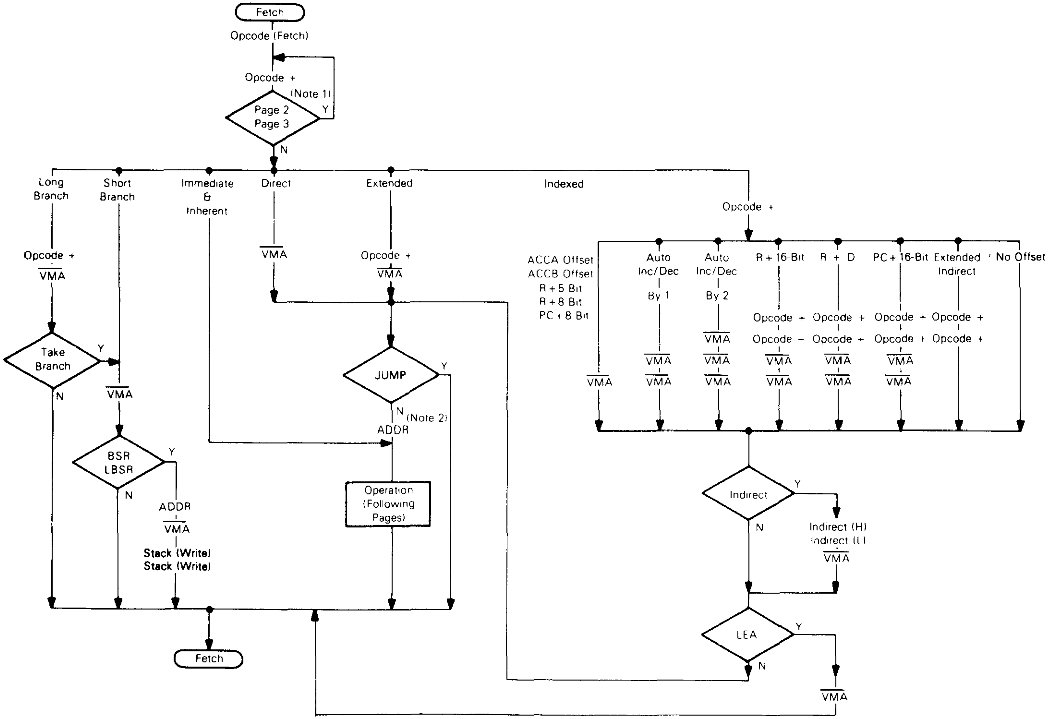
FIGURE 18 — SYNC TIMING



NOTE: 1. If the associated mask bit is set when the interrupt is requested, this cycle will continue the instruction fetched from the previous cycle. However, if the interrupt is accepted ($\overline{NMI}$ or an unmasked $\overline{FIRQ}$ or $\overline{IRQ}$) the opcode address (placed on the bus from cycle m + 1) remains on the bus and interrupt processing continues with this cycle as (m + 2) on Figures 10 and 11 (Interrupt Timing).

2. If mask bits are clear, $\overline{IRQ}$ and $\overline{FIRQ}$ must be held low for three cycles to guarantee interrupt to be taken, although only one cycle is necessary to bring the processor out of SYNC.

FIGURE 19 — ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE



NOTES: 1. All subsequent PAGE 2 and PAGE 3 opcodes will be ignored after initial opcode fetch.
2. Write operation during store instruction.

FIGURE 19 — ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE (CONTINUED)

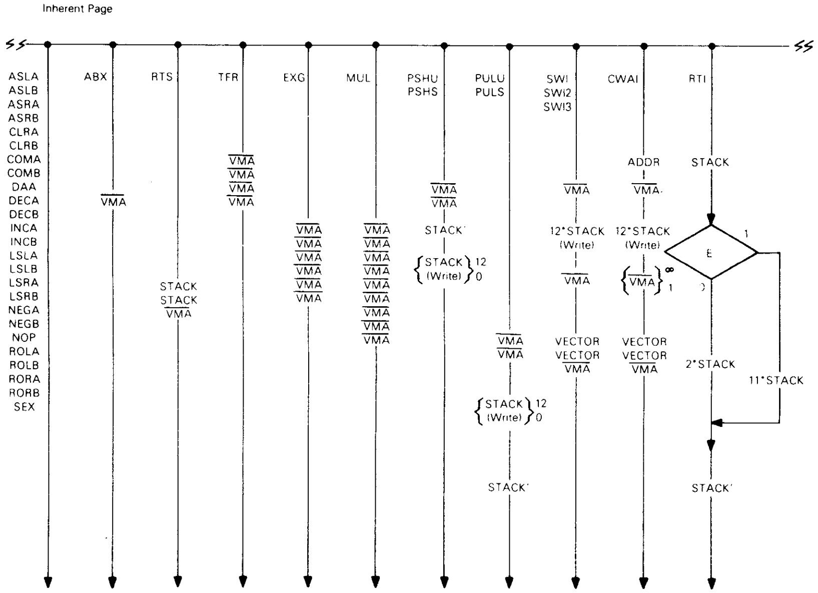


FIGURE 19 — ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE (CONTINUED)

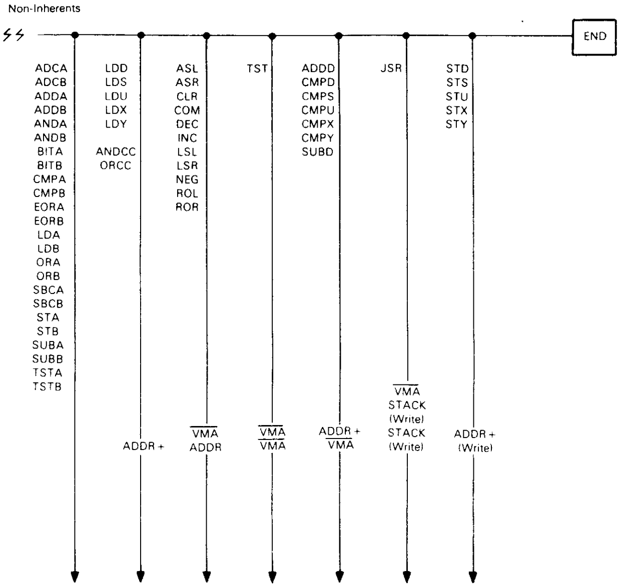


TABLE 4 — 8-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS

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

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

TABLE 5 — 16-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS

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

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

TABLE 6 — INDEX REGISTER/STACK POINTER INSTRUCTIONS

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

TABLE 7 — BRANCH INSTRUCTIONS

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

TABLE 8 — MISCELLANEOUS INSTRUCTIONS

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

TABLE 9 — HEXADECIMAL VALUES OF MACHINE CODES

OP	Mnem	Mode	~	#	OP	Mnem	Mode	~	#	OP	Mnem	Mode	~	#
00	NEG	Direct	6	2	30	LEAX	Indexed	4+	2+	60	NEG	Indexed	6+	2+
01	•	↑			31	LEAY	↑	4+	2+	61	•	↑		
02	•				32	LEAS	↓	4+	2+	62	•			
03	COM		6	2	33	LEAU	Indexed	4+	2+	63	COM		6+	2+
04	LSR		6	2	34	PSHS	Inherent	5+	2	64	LSR		6+	2+
05	•				35	PULS	↑	5+	2	65	•			
06	ROR		6	2	36	PSHU		5+	2	66	ROR		6+	2+
07	ASR		6	2	37	PULU		5+	2	67	ASR		6+	2+
08	ASL, LSL		6	2	38	•				68	ASL, LSL		6+	2+
09	ROL		6	2	39	RTS		5	1	69	ROL		6+	2+
0A	DEC		6	2	3A	ABX		3	1	6A	DEC		6+	2+
0B	•				3B	RTI		6, 15	1	6B	•			
0C	INC		6	2	3C	CWAI		20	2	6C	INC		6+	2+
0D	TST		6	2	3D	MUL		11	1	6D	TST		6+	2+
0E	JMP	↓	3	2	3E	•	↓			6E	JMP	↓	3+	2+
0F	CLR	Direct	6	2	3F	SWI	Inherent	19	1	6F	CLR	Indexed	6+	2+
10	Page 2	—	—	—	40	NEGA	Inherent	2	1	70	NEG	Extended	7	3
11	Page 3	—	—	—	41	•	↑			71	•	↑		
12	NOP	Inherent	2	1	42	•				72	•			
13	SYNC	Inherent	2	1	43	COMA		2	1	73	COM		7	3
14	•				44	LSRA		2	1	74	LSR		7	3
15	•				45	•				75	•			
16	LBRA	Relative	5	3	46	RORA		2	1	76	ROR		7	3
17	LBSR	Relative	9	3	47	ASRA		2	1	77	ASR		7	3
18	•				48	ASLA, LSLA		2	1	78	ASL, LSL		7	3
19	DAA	Inherent	2	1	49	ROLA		2	1	79	ROL		7	3
1A	ORCC	Immed	3	2	4A	DECA		2	1	7A	DEC		7	3
1B	•	–			4B	•				7B	•			
1C	ANDCC	Immed	3	2	4C	INCA		2	1	7C	INC		7	3
1D	SEX	Inherent	2	1	4D	TSTA		2	1	7D	TST		7	3
1E	EXG	↕	8	2	4E	•	↓			7E	JMP	↓	4	3
1F	TFR	Inherent	6	2	4F	CLRA	Inherent	2	1	7F	CLR	Extended	7	3
20	BRA	Relative	3	2	50	NEGB	Inherent	2	1	80	SUBA	Indexed	2	2
21	BRN	↑	3	2	51	•	↑			81	CMPA	↑	2	2
22	BHI		3	2	52	•				82	SBCA		2	2
23	BLS		3	2	53	COMB		2	1	83	SUBD		4	3
24	BHS, BCC		3	2	54	LSRB		2	1	84	ANDA		2	2
25	BLO, BCS		3	2	55	•				85	BITA		2	2
26	BNE		3	2	56	RORB		2	1	86	LDA		2	2
27	BEQ		3	2	57	ASRA		2	1	87	•			
28	BVC		3	2	58	ASLB, LSLB		2	1	88	EORA		2	2
29	BVS		3	2	59	ROLB		2	1	89	ADCA		2	2
2A	BPL		3	2	5A	DECB		2	1	8A	ORA		2	2
2B	BMI		3	2	5B	•				8B	ADDA	↓	2	2
2C	BGE		3	2	5C	INCB		2	1	8C	CMPX	Immed	4	3
2D	BLT		3	2	5D	TSTB		2	1	8D	BSR	Relative	7	2
2E	BGT	↓	3	2	5E	•	↓			8E	LDX	Immed	3	3
2F	BLE	Relative	3	2	5F	CLRB	Inherent	2	1	8F	•			

LEGEND:

~ Number of MPU cycles (less possible push pull or indexed-mode cycles)

Number of program bytes

• Denotes unused opcode

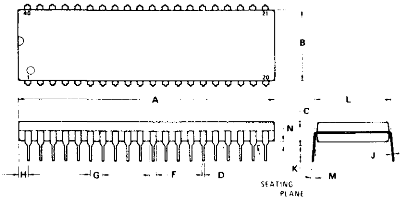
TABLE 9 — HEXADECIMAL VALUES OF MACHINE CODES (CONTINUED)

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+
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
BD	JSR		8	3
BE	LDX		6	3
BF	STX	Extended	6	3
C0	SUBB	Immed	2	2
C1	CMPB		2	2
C2	SBCB		2	2
C3	ADDD		4	3
C4	ANDB		2	2
C5	BITB	Immed	2	2
C6	LDB	Immed	2	2
C7	*			
C8	EORB		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
DD	STD		5	2
DE	LDU		5	2
DF	STU	Direct	5	2
E0	SUBB	Indexed	4+	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	LDU		5+	2+
EF	STU	Indexed	5+	2+
F0	SUBB	Extended	5	3
F1	CMPB		5	3
F2	SBCB		5	3
F3	ADDD		7	3
F4	ANDB		5	3
F5	BITB		5	3
F6	LDB		5	3
F7	STB		5	3
F8	EORB		5	3
F9	ADCB		5	3
FA	ORB		5	3
FB	ADDB	Extended	5	3
FC	LDD	Extended	6	3
FD	STD		6	3
FE	LDU		6	3
FF	STU	Extended	6	3

Page 2 and 3 Machine Codes

OP	Mnem	Mode	~	#
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		5(6)	4
102E	LBGT		5(6)	4
102F	LBLE	Relative	5(6)	4
103F	SWI2	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		8	4
10BE	LDY		7	4
10BF	STY	Extended	7	4
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	SWI3	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

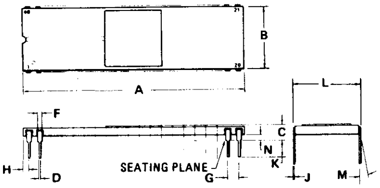
NOTE: All unused opcodes are both undefined and illegal



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	51.69	52.45	2.035	2.065
B	13.72	14.22	0.540	0.560
C	3.94	5.08	0.155	0.200
D	0.36	0.56	0.014	0.022
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.16	0.065	0.085
J	0.20	0.38	0.008	0.015
K	2.92	3.43	0.115	0.135
L	15.24 BSC		0.600 BSC	
M	0°	15°	0°	15°
N	0.51	1.02	0.020	0.040

NOTES:
1. POSITIONAL TOLERANCE OF LEADS (D), SHALL BE WITHIN 0.25 mm (0.010) AT MAXIMUM MATERIAL CONDITION, IN RELATION TO SEATING PLANE AND EACH OTHER.
2. DIMENSION L TO CENTER OF LEADS WHEN FORMED PARALLEL.
3. DIMENSION B DOES NOT INCLUDE MOLD FLASH.

P SUFFIX
PLASTIC PACKAGE
CASE 711



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.94	15.34	0.588	0.604
C	3.05	4.06	0.120	0.160
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.99	15.49	0.590	0.610
M	–	10°	–	10°
N	1.02	1.52	0.040	0.060

NOTES:
1. LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX MAT'L CONDITION.
2. DIMENSION "L" TO CENTER OF LEADS WHEN FORMED PARALLEL.

L SUFFIX
CERAMIC PACKAGE
CASE 715

MC6809E/MC68A09E/MC68B09E

HMOS

(HIGH DENSITY N-CHANNEL, SILICON-GATE)

8-BIT MICROPROCESSING UNIT

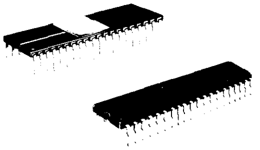
8-BIT MICROPROCESSING UNIT

The MC6809E is a revolutionary high performance 8-bit microprocessor which supports modern programming techniques such as position independence, reentrancy, and modular programming.

This third-generation addition to the M6800 family has major architectural improvements which include additional registers, instructions and addressing modes.

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809E has the most complete set of addressing modes available on any 8-bit microprocessor today.

The MC6809E has hardware and software features which make it an ideal processor for higher level language execution or standard controller applications.



MC6800 COMPATIBLE

- Hardware – Interfaces with All M6800 Peripherals
- Software – Upward Source Code Compatible Instruction Set and Addressing Modes

ARCHITECTURAL FEATURES

- Two 16-bit Index Registers
- Two 16-bit Indexable Stack Pointers
- Two 8-bit Accumulators can be Concatenated to Form One 16-Bit Accumulator
- Direct Page Register Allows Direct Addressing Throughout Memory

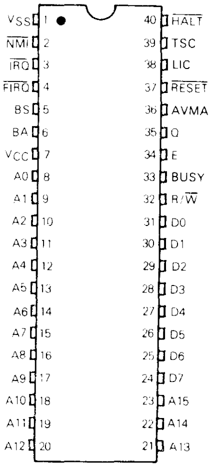
HARDWARE FEATURES

- External Clocking (÷ 1)
- Fast Interrupt Request Input Stacks Only Condition Code Register and Program Counter
- Interrupt Acknowledge Output Allows Vectoring By Devices
- SYNC Acknowledge Output Allows for Synchronization to External Event
- Single Bus-Cycle RESET
- Single 5-Volt Supply Operation
- NMI Blocked After RESET Until After First Load of Stack Pointer
- Early Address Valid Allows Use With Slower Memories
- Early Write-Data for Dynamic Memories
- BUSY Signal Supports Tightly Coupled Multiprocessor Systems

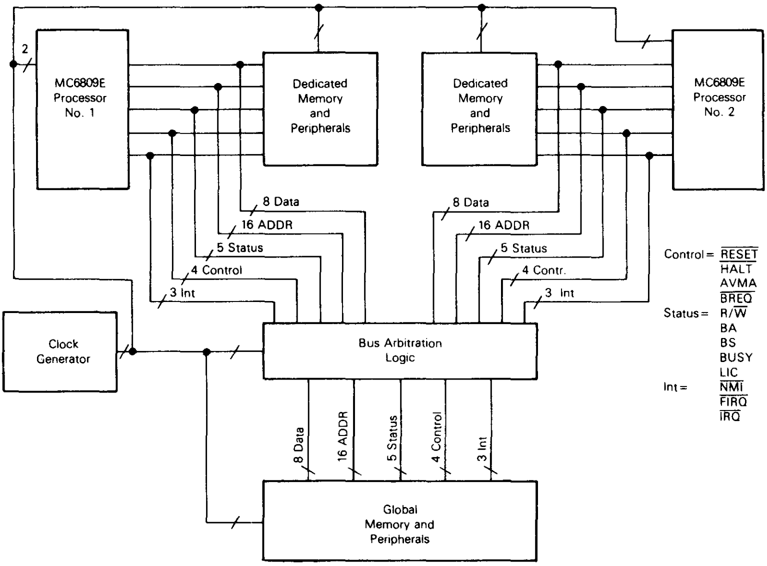
SOFTWARE FEATURES

- 10 Addressing Modes
  - M6800 Upward Compatible Addressing Modes
  - Direct Addressing Anywhere in Memory Map
  - Long Relative Branches
  - Program Counter Relative
  - True Indirect Addressing
  - Expanded Indexed Addressing:
    0, 5, 8, or 16-bit Constant Offsets
    8, or 16-bit Accumulator Offsets
    Auto-Increment/Decrement by 1 or 2
- Improved Stack Manipulation
- 59 Instruction Mnemonics
- 8 × 8 Unsigned Multiply
- 16-bit Arithmetic
- Transfer/Exchange All Registers
- Push/Pull Any Registers or Any Set of Registers
- Load Effective Address

PIN ASSIGNMENT



TYPICAL MULTIPROCESSOR SYSTEM



MODERN SOFTWARE TECHNIQUES

Available to the MC6809/MC6809E User

- Position Independent Programs – Program Executes Properly Anywhere in the Address Space
- Module Programs – Programs Easily Divided Into Small Manageable Modules
- Re-Entrant Programs – A Routine Usable By Interrupt and Non-Interrupt Programs Without Losing Data
- Structured High Level Languages – More Efficient Compilers and Compiler Generated Code
- Recursive Routines

MC6829

HMOS

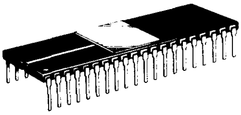
(HIGH DENSITY N-CHANNEL, SILICON-GATE)

MEMORY MANAGEMENT UNIT (MMU)

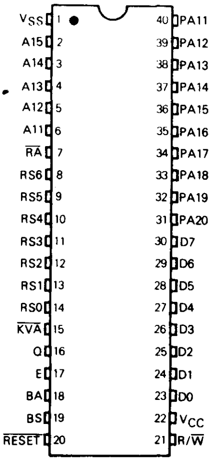
MEMORY MANAGEMENT UNIT

The principle function of the MC6829 Memory Management Unit (MMU) is to expand the address space of the MC6809 from 64K bytes to a possible maximum of 2 Megabytes. Each MMU is capable of handling four different concurrent tasks including DMA. The MMU can also protect the address space of one task from modification by another task. Memory address space expansion is accomplished by applying the upper five address lines of the processor (A11-A15) along with the contents of a five-bit task register to an internal high-speed mapping RAM. The MMU output consists of ten physical address lines (PA20-PA11) which, when combined with the eleven lower address lines of the processor (A10-A0), forms a physical address space of 2 Mbytes. Each task is assigned memory in increments of 2 k bytes up to a total of 64 k bytes. In this manner, the address spaces of different tasks can be kept separate from one another. The resulting simplification of the address space programming model will increase the reliability of a complex multi-process system.

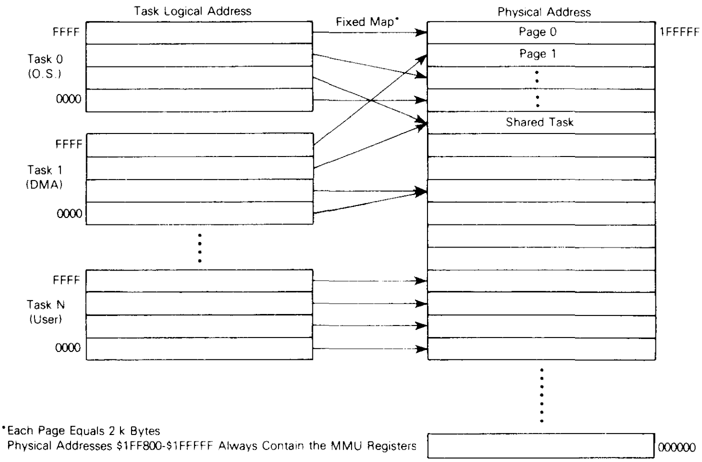
- Expands Memory Address Space from 64K to 2 Megabytes
- Each MMU is Capable of Handling 4 Separate Tasks
- Up to 8 MMUs can be Used in a System
- Task Isolation and Write Protection
- Provides Efficient Memory Allocation; 1024 Pages of 2 k Bytes Each
- Designed to be Used with DMA
- Fast, Automatic on Chip Task Switching
- Allows Interprocess Communication Through Shared Resources
- Simplifies Programming Model of Address Space
- Increases System Reliability
- 6809/6800 Bus Compatible
- Single 5-Volt Power Supply



PROPOSED PIN ASSIGNMENT

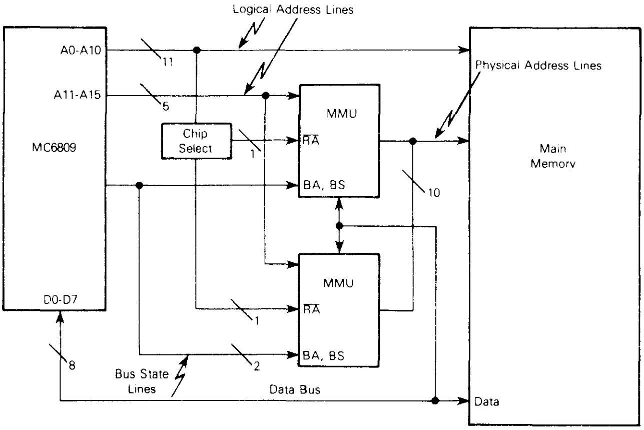


LOGICAL TO PHYSICAL ADDRESS MAPPING EXAMPLES



*Each Page Equals 2 k Bytes
Physical Addresses $1FF800-$1FFFFF Always Contain the MMU Registers

TYPICAL SYSTEM CONFIGURATION



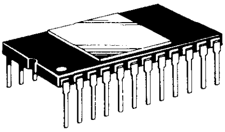
MC6839

HMOS

(HIGH DENSITY N-CHANNEL, SILICON-GATE)

M6809 FLOATING
POINT ROM

- Totally Position Independent
- No Absolute RAM Used (Re-Entrant)
- Operands in Registers or on the Stack (PASCAL)
- Compatible with Proposed IEEE Standard
- Single, Double, and Double Extended Formats
- Includes the Following Operations
  Add
  Subtract
  Multiply
  Divide
  Remainder
  Square Root
  Integer Part
  Absolute Value
  Negate
  Compare
  Convert Integer ↔ Floating Point
  Convert Binary ↔ Decimal

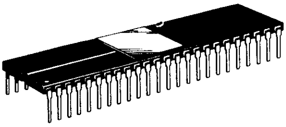


MC6842

HMOS

SERIAL DIRECT MEMORY ACCESS PROCESSOR

SERIAL DIRECT MEMORY ACCESS PROCESSOR

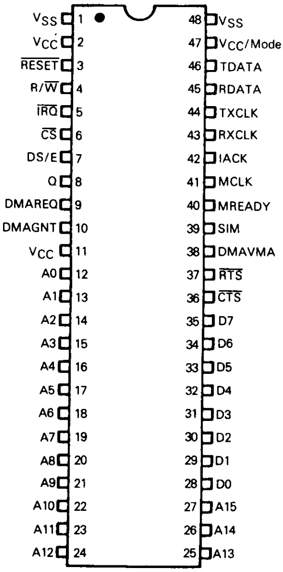


The MC6842 Serial Direct Memory Access (SDMA) processor provides a high speed serial link between microprocessors or intelligent controllers in distributed processing systems. Using IBM's Synchronous Data Link Control (SDLC) protocol, the MC6842 is capable of handling multidrop, point-to-point, or loop configurations. Many HDLC protocol features are also supported.

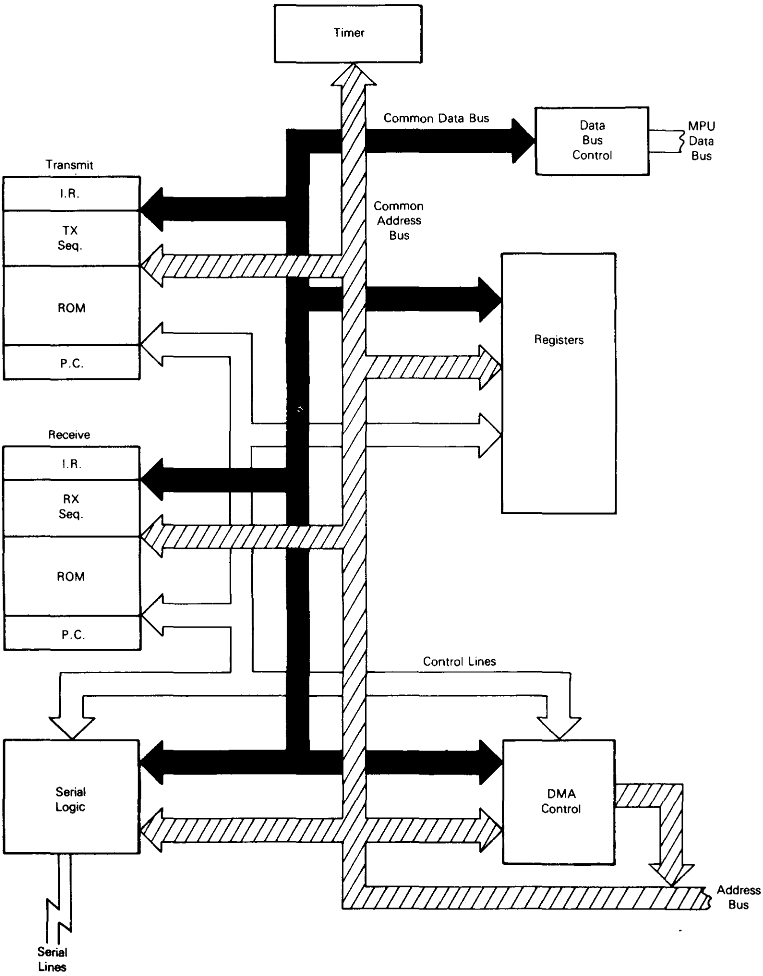
The SDMA accepts commands from the local microprocessor to either transfer data or issue link-level commands. The SDMA issues and responds to most link-level commands, ensures data integrity and validation, and handles some error recovery.

- Up to 4M Bit/Sec Rate
- External Data Recovery
- External Clocks
- DMA Command and Data Chaining
- SDLC Protocol
- HDLC
- Full/Half Duplex Operation
- Separate DMA Channel for Transmit and Receive
- Normal or Systems Address Detection
- MC6809, MC6800 Bus Compatible
- NRZ/NRZI Data
- Internal Bit and Byte Synchronization
- Point-to-Point, Multidrop, and Loop Modes
- CRC Generator and Detection
- Primary/Secondary Configurations
- External Power for Loop Mode
- Initialization Control Pin (SIM)
- Vectored Interrupt Capability (IACK)

PROPOSED PIN ASSIGNMENT



BLOCK DIAGRAM



MEK6809AC

EDITOR ASSEMBLER FOR THE MEK6809D4B

The MEK6809EAC is a software package designed to operate in conjunction with systems based on the MEK6809D4 Microcomputer Module. It is available in audio cassette form recorded at 300 Baud. It is compatible with the tape format of the MEK6809D4.

General Features

- ROMable
- Assembles both 6800 and 6809 mnemonics to 6809 object
- Able to Operate with D4 or Stand-Alone 6809
- Self-Sizing of Memory
- Can Read Tapes Produced by MEK6802EA Using MEK68I/O
- Serial or Parallel Printer Supported
- Interactive Editor
- Uses MEK6809D4 Cassette for Program Storage
- Uses MEK68R2D for Video Display

Editor Commands and Features

- Append Text Files from Audio Cassette
- Punch Text Files to Audio Cassette
- Print Text Files on Optional Printer
- Find Beginning of Text File
- Move Through Text File One or More Characters at a Time
- Find End of Text File
- Move Through Text File One or More Lines at a Time
- Insert/Delete Character Strings
- Search for a Character String
- Change Character String
- Insert/Delete Lines
- Chain Editor Commands

Assembler Commands and Features

- Speed Control of Assembly Listing Allows Examination on CRT Screen
- Paging and Space Control of Listing
- Pseudo Opcodes for Assembler Control
- Produce Assembly Listing on Screen or Printer
- List Symbols
- Output Object Code to Audio Cassette
- Output Object Code to Memory

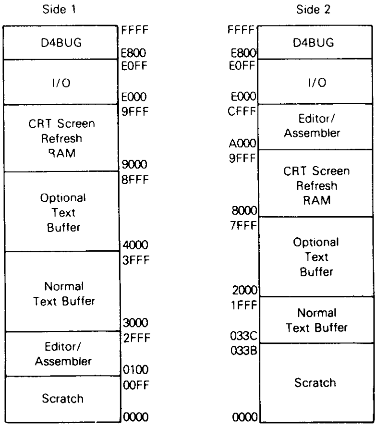
ORDERING INFORMATION

Media	Part Number
300 Baud Audio Cassette	MEK6809EAC

MEMORY MAP

The illustration below depicts the optional memory maps for typical systems using the MEK6809EAC.

MEK6809EAC MEMORY MAP



TYPICAL CONFIGURATIONS

The MEK6809EA can be used in a variety of system configurations, but each must contain basic items. Included in these are an MC6809 type microprocessor, a monitor ROM (D4BUG), I/O devices, and sufficient Read/Write memory (RAM) for the task to be accomplished.

The MEK6809D4B furnishes both the MC6809 MPU and the monitor ROM required. If an RS-232C compatible terminal is available, the MEK6809D4B also provides the I/O interface required. An alternative to the terminal consists of an MEK68R2D together with a CRT monitor (or modified TV set) and an ASCII encoded keyboard.

The RAM requirements can be met with an MEK68MM16 (16K byte) or MEK68MM32 (32K byte) dynamic memory module. The latter provides significantly more buffer space, needed for larger programs.

Side two of the MEK6809EAC contains a version of the Editor/Assembler which is suitable for operation from ROM or EPROM. This offers the user the opportunity to have this code programmed into EPROMs. These can then be inserted into the user ROM sockets provided on the MEK6809D4B. (The EPROMs required for this would be three 4K x 8 or six 2K x 8 single supply types. Multiple supply versions can be used if slight modifications are made to the MEK6809D4B). Population of the user RAM sockets on the MEK6809D4B provide text buffer space for small programs. Assuming an RS-232C terminal is available, this combination results in a single-board program development station.

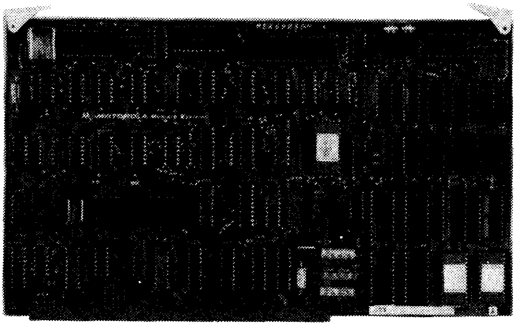
The use of any multi-board configuration will require a backplane. Alternatives available include the MEK68CMB Card Cage/Motherboard combination, and the MEK68MB5 Motherboard with stand-alone card guides. If the less expensive MEK68MB5 is chosen, it can be upgraded later with the MEK68CC.

MEK6809D4
MEK68KPD

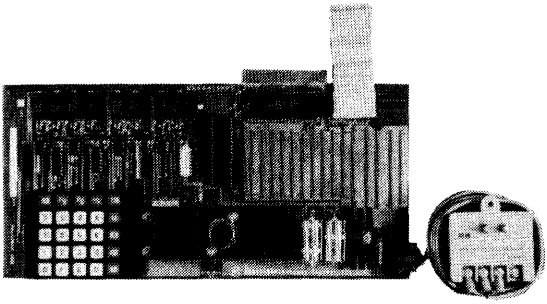
The MEK6809D4 Advanced Microcomputer Evaluation Board and MEK68KPD Keypad/Display Unit provide the necessary hardware and firmware for a computer system based on the Motorola MC6809 High Performance Microprocessor. The system forms an evaluation tool to facilitate the application of Motorola microprocessors and associated components.

The MEK6809D4A is used with an MEK68KPD and is complete with a power supply. 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.

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.



MEK6809D4



MEK68KPD

MEK6809D4/MEK68KPD

PRODUCT FEATURES

MEK6809D4

- MC6809 High Performance Microprocessor
- D4BUG Monitor Firmware (4K) Expandable to 6K
- Direct Memory Access
- Interrupt by Device
- Audio Cassette Interface, 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 Page 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 for Bidirectional Address Bus
- Control and Status Lines
- System Buffers

MEK68KPD

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

DIMENSIONS

- MEK6809D4:
  Two-Sided PC Board
  309.8 mm (12 in) Wide by
  177.8 mm (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

SUPPLY 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.

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 × 8 single or triple supply EPROMs or ROMs, 2K × 8 single or triple supply EPROMs or ROMs, 4K × 8 ROMs or EPROMs, or 8K × 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 × 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:

1. RS-232 compatible serial port including buffered handshake signals.
2. Baud rate generator providing baud rate clocks for 110, 300, 600, 1200, 4800, and 9600 baud rates.
3. Address, data, and control lines fully buffered at bus interface.

MODEL TYPES

MEK6809D4

MEK6909D4A – This variation 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.

MEK6809D4B – This variation 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 +12 V, +5 V, and −12 V power. A 4K +2K monitor is provided. No RAMs are provided in the "User RAM" array.

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 on-board power supply disconnected) may be used with the MEK6809D4B.

MEK68KPD

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.

EXPANSION

With the rapid advancements in the microprocessor industry, there is a 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 ten card slots on 5/8" centers, with alternate slots populated with 70-Pin connectors.

MEK68CMB Card Cage/Motherboard

The MEK68CMB can accomodate 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 8-1/4" high by 7-1/4" wide by 13-1/4" deep.

MEK68R2/R2D/R2M Programmable CRT Interface Modules

The MEK68R2/R2D/R2M Programmable CRT Interface 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 5×7 matrix display, semigraphics, and up to 4K of screen display memory. All modules provide an interface for an ASCII keyboard.

MEK68IO Input/Output Module

The MEK68IO is supplied with a 300/1200 Baud cassette interface, two MC6850 ACIAs, 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.

MEK6809EA 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 MEK6800AB Adapter Motherboard, and interfaces directly with the 60-Pin bus of the AB. The MEK68WW1 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.

MEK6809D4 AC Operating Conditions and Characteristics

AC OPERATING CHARACTERISTICS (Bus)

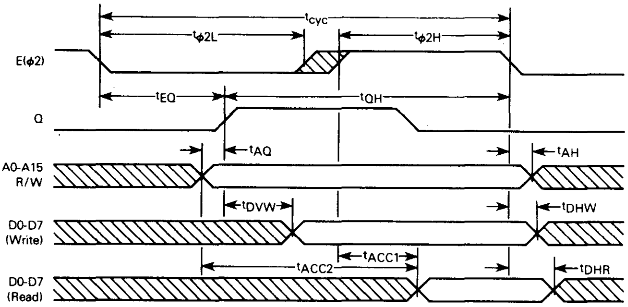
Parameter	Symbol	Min	Nom	Max	Unit
Cycle Time	t_{cyc}	1100	–	1130	ns
Address Setup	t_{AQ}	25	–	–	ns
Address Hold[3]	t_{AH}	10	30	–	ns
Write Data Valid	t_{DVW}	–	–	250	ns
Write Data Hold	t_{DHW}	10	30	–	ns
E (ϕ2) Low Time	$t_{\phi 2L}$	500	–	–	ns
E (ϕ2) High Time	$t_{\phi 2H}$	500	–	–	ns
E Low to Q High	t_{EQ}	275	–	–	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 $T_A = 25°C$
2) Timing measured at edge connector (50% points)
3) Measured from falling edge of E (ϕ2)
4) Measured from rising edge of (ϕ2)

MEK6809D4 TIMING DIAGRAM



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 D4B
  - Memory Dump to Examine Blocks of Data
  - Memory Fill
  - Memory Search
  - Memory Move
  - 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 MC6809.

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 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-1 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

Memory Dump

The memory dump command allows the user to display blocks of memory with ASCII equivalents. The display formats 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 satisfactory 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 algorithm 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

CPU

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

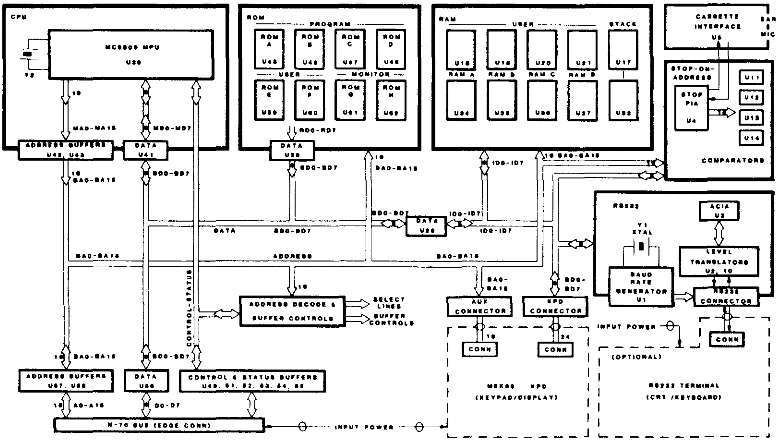
The MC6809 supports programming techniques such as position independence, re-entrancy, and modular programming. The MC6809 has hardware and software features which make it a suitable processor for higher level language execution or standard controller application.

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 × 8 ROMs or EPROMs.

The ROM type configurations are controlled by mini-jumpers which may be easily moved without the need for any tools.

MEK6809D4 BLOCK DIAGRAM



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.

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.

The 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 microprocessor 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.

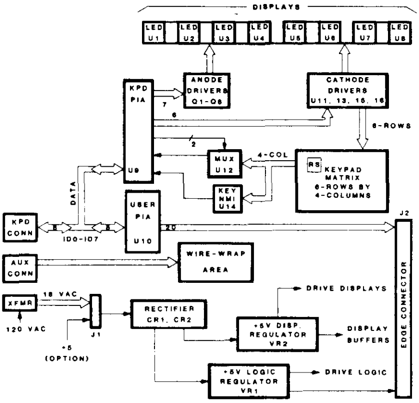
The RS-232 Circuit

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.

MEK68KPD BLOCK DIAGRAM



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.

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.

ORDERING INFORMATION

Part Number	Features	Prerequisites
MEK6809D4A	No ADDR/Data Buffers/No RS-232	MEK68KPD
MEK6809D4B	ADDR and Data Buffers/RS-232 INTFC	RS-232 Terminal and Power Supply or MEK68R2D, CRT, ASCII Keyboard and Power Supplies

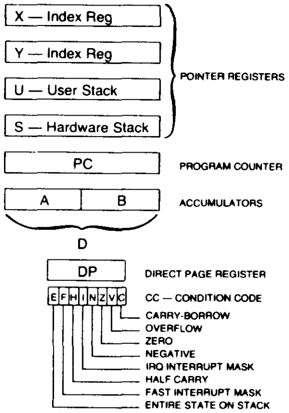
APPENDIX D

MC6809
Instruction Set Summary

The following pages are provided through the courtesy of Motorola, Inc., Austin, Texas.

MC6809 MICROPROCESSOR INSTRUCTION SET SUMMARY

6809 PROGRAMMING MODEL



6809 STACKING ORDER

INCREASING MEMORY ↓

PULL ORDER
↓
CC
A
B
DP
X Hi
X Lo
Y Hi
Y Lo
U/S Hi
U/S Lo
PC Hi
PC Lo
↑
PUSH ORDER

6809 VECTORS

FFFE	Restart
FFFC	NMI
FFFA	SWI
FFF8	IRQ
FFF6	FIRQ
FFF4	SWI2
FFF2	SWI3
FFF0	Reserved

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

MC6809 MICROPROCESSOR INSTRUCTION SET SUMMARY

HEXADECIMAL AND DECIMAL CONVERSION

HOW TO USE THE TABLES

CONVERSION TO DECIMAL Find the decimal weights 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

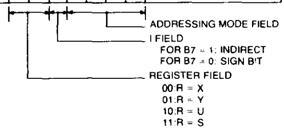
HEXADECIMAL AND DECIMAL CONVERSION							
15 BYTE 8				7 BYTE 0			
15 CHAR 12		11 CHAR 8		7 CHAR 4		3 CHAR 0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4 096	1	256	1	16	1	1
2	8 192	2	512	2	32	2	2
3	12 288	3	768	3	48	3	3
4	16 384	4	1 024	4	64	4	4
5	20 480	5	1 280	5	80	5	5
6	24 576	6	1 536	6	96	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	D	208	D	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	0
2	1
4	2
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	ı	y
A 1010	LF	SUB	•		J	Z	j	z
B 1011	VT	ESC	+	:	K	|	k	
C 1100	FF	FS	,	·	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 ± D OFFSET)
1	X	X	X	1	1	0	0	EA = ,PC ± 7 BIT OFFSET
1	X	X	X	1	1	0	1	EA = ,PC ± 15 BIT OFFSET
1	X	X	1	1	1	1	1	EA = ,ADDRESS



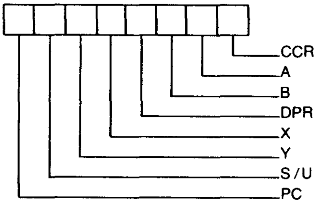
ADDRESSING MODE FIELD

I FIELD
FOR B7 = 1: INDIRECT
FOR B7 = 0: SIGN BIT

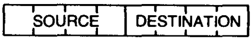
REGISTER FIELD
00:R = X
01:R = Y
10:R = U
11:R = S

MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

PUSH/PULL POST BYTE



TRANSFER/EXCHANGE POST BYTE



REGISTER FIELD

0000 = D (A:B)	1000 = A
0001 = X	1001 = B
0010 = Y	1010 = CCR
0011 = U	1011 = DPR
0100 = S	
0101 = PC	

OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
ØØ	NEG	DIRECT	6	2	1C	ANDCC	IMMED	3	2	2E	BGT	RELATIVE	3	2
Ø3	COM	↑	6	2	1D	SEX	INHERENT	2	1	2F	BLE	RELATIVE	3	2
Ø4	LSR		6	2	1E	EXG	↕	8	2	3Ø	LEAX	INDEXED	4	2
Ø6	ROR		6	2	1F	TFR	INHERENT	7	2	31	LEAY	↑	4	2
Ø7	ASR		6	2	2Ø	BRA	RELATIVE	3	2	32	LEAS	↓	4	2
Ø8	ASL/LSL		6	2	21	BRN	↑	3	2	33	LEAU	INDEXED	4	2
Ø9	ROL		6	2	22	BHI		3	2	34	PSHS	INHERENT	5	2
ØA	DEC		6	2	23	BLS		3	2	35	PULS	↑	5	2
ØC	INC		6	2	24	BHS/BCC		3	2	36	PSHU		5	2
ØD	TST		6	2	25	BLO/BCS		3	2	37	PULU		5	2
ØE	JMP	↓	3	2	26	BNE		3	2	39	RTS		5	1
ØF	CLR	DIRECT	6	2	27	BEQ		3	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	BMI		3	2	3F	SWI		19	1
19	DAA	INHERENT	2	1	2C	BGE	↓	3	2	4Ø	NEGA	↓	2	1
1A	ORCC	IMMED	3	2	2D	BLT	RELATIVE	3	2	43	COMA	INHERENT	2	1

MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
44	LSRA	INHERENT	2	1	5D	TSTB	INHERENT	2	1	77	ASR	EXTENDED	7	3
46	RORA	↑	2	1	5F	CLRB	INHERENT	2	1	78	ASL/LSL	↑	7	3
47	ASRA		2	1	6Ø	NEG	INDEXED	6	2	79	ROL		7	3
48	ASLA/LSLA		2	1	63	COM	↑	6	2	7A	DEC		7	3
49	ROLA		2	1	64	LSR		6	2	7C	INC		7	3
4A	DECA		2	1	66	ROR		6	2	7D	TST		7	3
4C	INCA		2	1	67	ASR		6	2	7E	JMP	↓	4	3
4D	TSTA		2	1	68	ASL/LSL		6	2	7F	CLR	EXTENDED	7	3
4F	CLRA		2	1	69	ROL		6	2	8Ø	SUBA	IMMED	2	2
5Ø	NEGB		2	1	6A	DEC		6	2	81	CMPA	↑	2	2
53	COMB		2	1	6C	INC		6	2	82	SBCA		2	2
54	LSRB		2	1	6D	TST		6	2	83	SUBD		4	3
56	RORB		2	1	6E	JMP	↓	3	2	84	ANDA		2	2
57	ASRA		2	1	6F	CLR	INDEXED	6	2	85	BITA		2	2
58	ASLB/LSLB		2	1	7Ø	NEG	EXTENDED	7	3	86	LDA		2	2
59	ROLB		2	1	73	COM	↑	7	3	88	EORA		2	2
5A	DECB	↓	2	1	74	LSR	↓	7	3	89	ADCA	↓	2	2
5C	INCB	INHERENT	2	1	76	ROR	EXTENDED	7	3	8A	ORA	IMMED	2	2

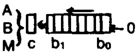
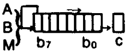
OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
8B	ADDA	IMMED	2	2	9E	LDX	DIRECT	5	2	BØ	SUBA	EXTENDED	5	3
8C	CMPX	IMMED	4	3	9F	STX	DIRECT	5	2	B1	CMPA	↑	5	3
8D	BSR	RELATIVE	7	2	AØ	SUBA	INDEXED	4	2	B2	SBCA		5	3
8E	LDX	IMMED	3	3	A1	CMPA	↑	4	2	B3	SUBD		7	3
9Ø	SUBA	DIRECT	4	2	A2	SBCA		4	2	B4	ANDA		5	3
91	CMPA	↑	4	2	A3	SUBD		6	2	B5	BITA		5	3
92	SBCA		4	2	A4	ANDA		4	2	B6	LDA		5	3
93	SUBD		6	2	A5	BITA		4	2	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		4	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	CØ	SUBB	IMMED	2	2
9D	JSR	DIRECT	7	2	AF	STX	INDEXED	5	2	C1	CMPB	IMMED	2	2

MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

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	ADDD	↑	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	DE	LDU	↓	5	2	FØ	SUBB	EXTENDED	5	3
CB	ADDB		2	2	DF	STU	DIRECT	5	2	F1	CMPB	↑	5	3
CC	LDD	↓	3	3	EØ	SUBB	INDEXED	4	2	F2	SBCB		5	3
CE	LDU	IMMED	3	3	E1	CMPB	↑	4	2	F3	ADDD		7	3
DØ	SUBB	DIRECT	4	2	E2	SBCB		4	2	F4	ANDB		5	3
D1	CMPB	↑	4	2	E3	ADDD		6	2	F5	BITB		5	3
D2	SBCB		4	2	E4	ANDB		4	2	F6	LDB		5	3
D3	ADDD		6	2	E5	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	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
FB	ADDB	EXTENDED	5	3	1Ø2E	LBGT	RELATIVE	5(6)	4	1ØCE	LDS	IMMED	4	4
FC	LDD	↑	6	3	1Ø2F	LBLE	RELATIVE	5(6)	4	1ØDE	LDS	DIRECT	6	3
FD	STD		6	3	1Ø3F	SWI/2	INHERENT	20	2	1ØDF	STS	DIRECT	6	3
FE	LDU	↓	6	3	1Ø83	CMPD	IMMED	5	4	1ØEE	LDS	INDEXED	6	3
FF	STU	EXTENDED	6	3	1Ø8C	CMPY	↕	5	4	1ØEF	STS	INDEXED	6	3
1Ø21	LBRN	RELATIVE	5	4	1Ø8E	LDY	IMMED	4	4	1ØFE	LDS	EXTENDED	7	4
1Ø22	LBHI	↑	5(6)	4	1Ø93	CMPD	DIRECT	7	3	1ØFF	STS	EXTENDED	7	4
1Ø23	LBLS		5(6)	4	1Ø9C	CMPY	↑	7	3	113F	SWI/3	INHERENT	2Ø	2
1Ø24	LBHS/LBCC		5(6)	4	1Ø9E	LDY	↓	6	3	1183	CMPU	IMMED	5	4
1Ø25	LBCS/LBLO		5(6)	4	1Ø9F	STY	DIRECT	6	3	118C	CMPS	IMMED	5	4
1Ø26	LBNE		5(6)	4	1ØA3	CMPD	INDEXED	7	3	1193	CMPU	DIRECT	7	3
1Ø27	LBEQ		5(6)	4	1ØAC	CMPY	↑	7	3	119C	CMPS	DIRECT	7	3
1Ø28	LBVC		5(6)	4	1ØAE	LDY	↓	6	3	11A3	CMPU	INDEXED	7	3
1Ø29	LBVS		5(6)	4	1ØAF	STY	INDEXED	6	3	11AC	CMPS	INDEXED	7	3
1Ø2A	LBPL		5(6)	4	1ØB3	CMPD	EXTENDED	8	4	11B3	CMPU	EXTENDED	8	4
1Ø2B	LBMI		5(6)	4	1ØBC	CMPY	↑	8	4	11BC	CMPS	EXTENDED	8	4
1Ø2C	LBGE	↓	5(6)	4	1ØBE	LDY	↓	7	4					
1Ø2D	LBLT	RELATIVE	5(6)	4	1ØBF	STY	EXTENDED	7	4					

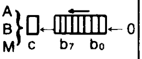
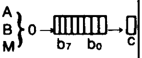
MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

		6809 ADDRESSING MODES																							
INSTRUCTION/		INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED[1]			RELATIVE				5	3	2	1	0
	FORMS	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~[5]	#	DESCRIPTION	H	N	Z	V	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					1
ASL	ASLA	48	2	1																	8	↕	↕	↕	↕
	ASLB	58	2	1																	8	↕	↕	↕	↕
	ASL				08	6	2	78	7	3				68	6+	2+					8	↕	↕	↕	↕
ASR	ASRA	47	2	1																	8	↕	↕	•	↕
	ASR	57	2	1																	8	↕	↕	•	↕
	ASR				07	6	2	77	7	3				67	6+	2+					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 = 1	•	•	•	•	•
	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 > Zero	•	•	•	•	•
	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	•	•	•	•	•
BIT	BITA				95	4	2	B5	5	3	85	2	2	A5	4+	2+				Bit Test A (M ∧ A)	•	↕	↕	0	•
	BITB				D5	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	•	•	•	•	•

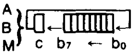
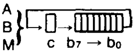
MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

INSTRUCTION/		INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED[1]			RELATIVE				5	3	2	1	0
	FORMS	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~[5]	#	DESCRIPTION	H	N	Z	V	C
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 Always	•	•	•	•	•
BRN	BRN																21	3	2	Branch Never	•	•	•	•	•
	LBRN																10 21	5	4	Long Branch 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 28	5(6)	4	Long Branch V = 0	•	•	•	•	•
BVS	BVS																29	3	2	Branch V = 1	•	•	•	•	•
	LBVS																10 29	5(6)	4	Long Branch 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	7	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 from A	8	↕	↕	↕	↕
	CMPB				D1	4	2	F1	5	3	C1	2	2	E1	4+	2+				Compare M from B	8	↕	↕	↕	↕
	CMPD				10 93	7	3	10 B3	8	4	10 83	5	4	10 A3	7+	3+				Compare M: M + 1 from D	•	↕	↕	↕	↕
	CMPS				11 9C	7	3	11 BC	8	4	11 8C	5	4	11 AC	7+	3+				Compare M: M + 1 from S	•	↕	↕	↕	↕
	CMPU				11 93	7	3	11 B3	8	4	11 83	5	4	11 A3	7+	3+				Compare M: M + 1 from U	•	↕	↕	↕	↕
	CMPX				9C	6	2	BC	7	3	8C	4	3	AC	6+	2+				Compare M: M + 1 from X	•	↕	↕	↕	↕
	CMPY				10 9C	7	3	10 BC	8	4	10 8C	5	4	10 AC	7+	3+				Compare M: M + 1 from Y	•	↕	↕	↕	↕

MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

INSTRUCTION/		INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED[1]			RELATIVE				5	3	2	1	0
	FORMS	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~[5]	#	DESCRIPTION	H	N	Z	V	C
COM	COMA	43	2	1																$\overline{A}$ → A	•	↕	↕	0	1
	COMB	53	2	1																$\overline{B}$ → B	•	↕	↕	0	1
	COM				03	6	2	73	7	3				63	6+	2+				$\overline{M}$ → M	•	↕	↕	0	1
CWAI		3C	20	2																CC ∧ IMM → CC; Wait for Interrupt					1
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	6	2	7A	7	3				6A	6+	2+				M − 1 → M	•	↕	↕	↕	•
EOR	EORA				98	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	7	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				D6	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 DE	6	3	10 FE	7	4	10 CE	4	4	10 EE	6+	3+				M: M + 1 → S	•	↕	↕	0	•
	LDU				DE	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 9E	6	3	10 BE	7	4	10 8E	4	4	10 AE	6+	3+				M: M + 1 → Y	•	↕	↕	0	•
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																	•	↕	↕	↕	↕
	LSLB	58	2	1																	•	↕	↕	↕	↕
	LSL				08	6	2	78	7	3				68	6+	2+					•	↕	↕	↕	↕
LSR	LSRA	44	2	1																	•	0	↕	•	↕
	LSRB	54	2	1																	•	0	↕	•	↕
	LSR				04	6	2	74	7	3				64	6+	2+					•	0	↕	•	↕
MUL		3D	11	1																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+	2+				$\overline{M}$ + 1 → M	8	↕	↕	↕	↕
NOP		12	2	1																No Operation	•	•	•	•	•
OR	ORA				9A	4	2	BA	5	3	8A	2	2	AA	4+	2+				A ∨ M → A	•	↕	↕	0	•
	ORB				DA	4	2	FA	5	3	CA	2	2	EA	4+	2+				B ∨ M → B	•	↕	↕	0	•
	ORCC										1A	3	2							CC ∨ IMM → CC					7
PSH	PSHS	34	5+[4]	2																Push Registers on S Stack	•	•	•	•	•
	PSHU	36	5+[4]	2																Push Registers on U Stack	•	•	•	•	•

MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

INSTRUCTION/ FORMS		INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED[1]			RELATIVE			DESCRIPTION	5	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~[5]	#		H	N	Z	V	C
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																	•	↕	↕	↕	↕
	ROLB	59	2	1																	•	↕	↕	↕	↕
	ROL				09	6	2	79	7	3				69	6+	2+					•	↕	↕	↕	↕
ROR	RORA	46	2	1																	•	↕	↕	•	↕
	RORB	56	2	1																	•	↕	↕	•	↕
	ROR				06	6	2	76	7	3				66	6+	2+					•	↕	↕	•	↕
RTI		3B	6/15	1																Return From Interrupt					7
RTS		39	5	1																Return From Subroutine	•	•	•	•	•
SBC	SBCA				92	4	2	B2	5	3	82	2	2	A2	4+	2+				A − M − C → A	8	↕	↕	↕	↕
	SBCB				D2	4	2	F2	5	3	C2	2	2	E2	4+	2+				B − M − C → B	8	↕	↕	↕	↕
SEX		1D	2	1																Sign Extend B into A	•	↕	↕	0	•
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 DF	6	3	10 FF	7	4				10 EF	6+	3+				S → M: M + 1	•	↕	↕	0	•
	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 9F	6	3	10 BF	7	4				10 AF	6+	3+				Y → M: M + 1	•	↕	↕	0	•
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 3F	20	2																Software Interrupt 2	•	•	•	•	•
	SWI3[6]	11 3F	20	2																Software Interrupt 3	•	•	•	•	•
SYNC		13	≥2	1																Synchronize to Interrupt	•	•	•	•	•
TFR	R1, R2	1F	7	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	•

MC6809 MICROPROCESSOR
INSTRUCTION SET SUMMARY

INDEXED ADDRESSING MODES

		NON INDIRECT				INDIRECT			
TYPE	FORMS	Assembler Form	Post-Byte OP Code	+ ~	+ #	Assembler Form	Post-Byte OP Code	+ ~	+ #
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 OFFSET	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

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
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. SW1 sets I&F bits. SW12 and SW13 do not affect I&F.
7. Conditions Codes 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);	Z	Zero (byte)
~	Number of MPU Cycles;	V	Overflow, 2's complement
#	Number of Program Bytes;	C	Carry from bit 7
+	Arithmetic Plus;	↕	Test and set if true, cleared otherwise
−	Arithmetic Minus;	•	Not Affected
·	Multiply	CC	Condition Code Register
$\overline{M}$	Complement of M;	:	Concatenation
→	Transfer Into;	∨	Logical or
H	Half-carry from bit 3;	∧	Logical and
N	Negative (sign bit)	⊻	Logical Exclusive or

Index

READER SERVICE CARD

To better serve you, the reader, please take a moment to fill out this card, or a copy of it, for us. Not only will you be kept up to date on the Blacksburg Series books, but as an extra bonus, **we will randomly select five cards every month, from all of the cards sent to us during the previous month. The names that are drawn will win, absolutely free, a book from the Blacksburg Continuing Education Series.** Therefore, make sure to indicate your choice in the space provided below. For a complete listing of all the books to choose from, refer to the inside front cover of this book. Please, one card per person. Give everyone a chance.

In order to find out who has won a book in your area, call (703) 953-1861 anytime during the night or weekend. When you do call, an answering machine will let you know the monthly winners. Too good to be true? Just give us a call. Good luck.

If I win, please send me a copy of:

____

I understand that this book will be sent to me absolutely free, if my card is selected.

For our information, how about telling us a little about yourself. We are interested in your occupation, how and where you normally purchase books and the books that you would like to see in the Blacksburg Series. We are also interested in finding authors for the series, so if you have a book idea, write to The Blacksburg Group, Inc., P.O. Box 242, Blacksburg, VA 24060 and ask for an Author Packet. We are also interested in TRS-80, APPLE, OSI and PET BASIC programs.

My occupation is ______________________________
I buy books through/from ______________________
Would you buy books through the mail? __________
I'd like to see a book about ____________________
Name ____
Address ______________________________________
City ____
State ___________________________ Zip _________

MAIL TO: BOOKS, BOX 715, BLACKSBURG, VA 24060
!!!!!PLEASE PRINT!!!!!

The Blacksburg Group

According to Business Week magazine (Technology July 6, 1976) large scale integrated circuits or LSI "chips" are creating a second industrial revolution that will quickly involve us all. The speed of the developments in this area is breathtaking and it becomes more and more difficult to keep up with the rapid advances that are being made. It is also becoming difficult for newcomers to "get on board."

It has been our objective, as The Blacksburg Group, to develop timely and effective educational materials and aids that will permit students, engineers, scientists and others to quickly learn how to apply new technologies to their particular needs. We are doing this through a number of means, textbooks, short courses, and through the development of educational "hardware" or training aids.

Our group members make their home in Blacksburg, found in the Appalachian Mountains of southwestern Virginia. While we didn't actively start our group collaboration until the Spring of 1974, members of our group have been involved in digital electronics, minicomputers and microcomputers for some time.

Some of our past experiences and on-going efforts include the following:

—The development of the Mark-8 computer, an 8008-based device that was featured in Radio-Electronics magazine in 1974, and generally recognized as the first widely available hobby computer. We have also designed several 8080-based computers, including the Mini-Micro Designer (MMD-1). More recently we have been working with 8085-based computers and the TRS-80.

—The Blacksburg Continuing Education Series^{TM} covers subjects ranging from basic electronics through microcomputers, operational amplifiers, and active filters. Test experiments and examples have been provided in each book. We are strong believers in the use of detailed experiments and examples to reinforce basic concepts. This series originally started as our Bugbook series and many titles are now being translated into Chinese, Japanese, German and Italian.

—We have pioneered the use of small self-contained computers in hands-on courses aimed at microcomputer users. The solderless breadboarding modules developed for use in circuit design and development make it easy for people to set up and test digital circuits and computer interfaces. Some of our technical products are marketed by Group Technology, Ltd., Check, VA 24072, USA. (703) 651-3153.

—Our short course programs have been presented throughout the world, covering digital electronics through TRS-80 computer interfacing. Programs are offered through the Blacksburg Group and the Virginia Tech Extension Division. Each course offers a mix of lectures and hands-on laboratory sessions. Courses are presented on a regular basis in Blacksburg, and at various times to open groups, companies, schools, and other sponsors.

For additional information about course offerings, we encourage you to write or call Dr. Chris Titus at The Blacksburg Group, Box 242, Blacksburg, VA 24060, (703) 951-9030, or Dr. Linda Leffer at the Center for Continuing Education, Virginia Tech, Blacksburg, VA 24061, (703) 961-5241.

Mr. David Larsen is on the faculty of the Department of Chemistry at Virginia Polytechnic Institute and State University. Dr. Jonathan Titus and Dr. Christopher Titus are with The Blacksburg Group, Inc., all of Blacksburg, Virginia.

6809
MICROCOMPUTER PROGRAMMING & INTERFACING
WITH EXPERIMENTS

The book was written to provide you with a sound understanding of how to program and interface the 6809 microprocessor.

Chapter 1 presents the chip structure and basic concepts of the 6809. Chapter 2 discusses the highly important subject of addressing modes. Registers and data movement instructions are covered in Chapter 4, while Chapter 5 treats the arithmetic, logic, and test instructions for the 6809. The next chapter deals with branching and other instructions. Input and output signals are then detailed, and the last chapter is concerned with interfacing and applications of the 6809. Review questions and answers are given at the end of Chapters 1 through 6 to test the reader on the material detailed in these chapters.

Four appendixes have been included in the book. They cover the 6809 instructions, the 6820/2821 peripheral interface adapter, and specification sheets for the 6809 and related 6800-series equipment.

Andrew C. Staugaard, Jr. is an experienced engineer/educator in microelectronics. Currently he is Assistant Professor of Electrical Engineering at Jamestown Community College, where he received the Faculty Award for Excellence in Teaching in 1977. Prior to teaching, Professor Staugaard was employed as a quality control engineer in microelectronics processing by the Bendix Corporation. He is the author of the SAMS books **How to Program and Interface the 6800** and **6801, 68701, and 6803 Microcomputer Programming and Interfacing.** He is a co-author of monthly columns on the Motorola 6800 chip family that appears in several U.S. and foreign magazines.

Howard W. Sams & Co., Inc.
4300 W. 62nd Street, Indianapolis, Indiana 46268 USA

$13.95/21798

ISBN: 0-672-21798-8