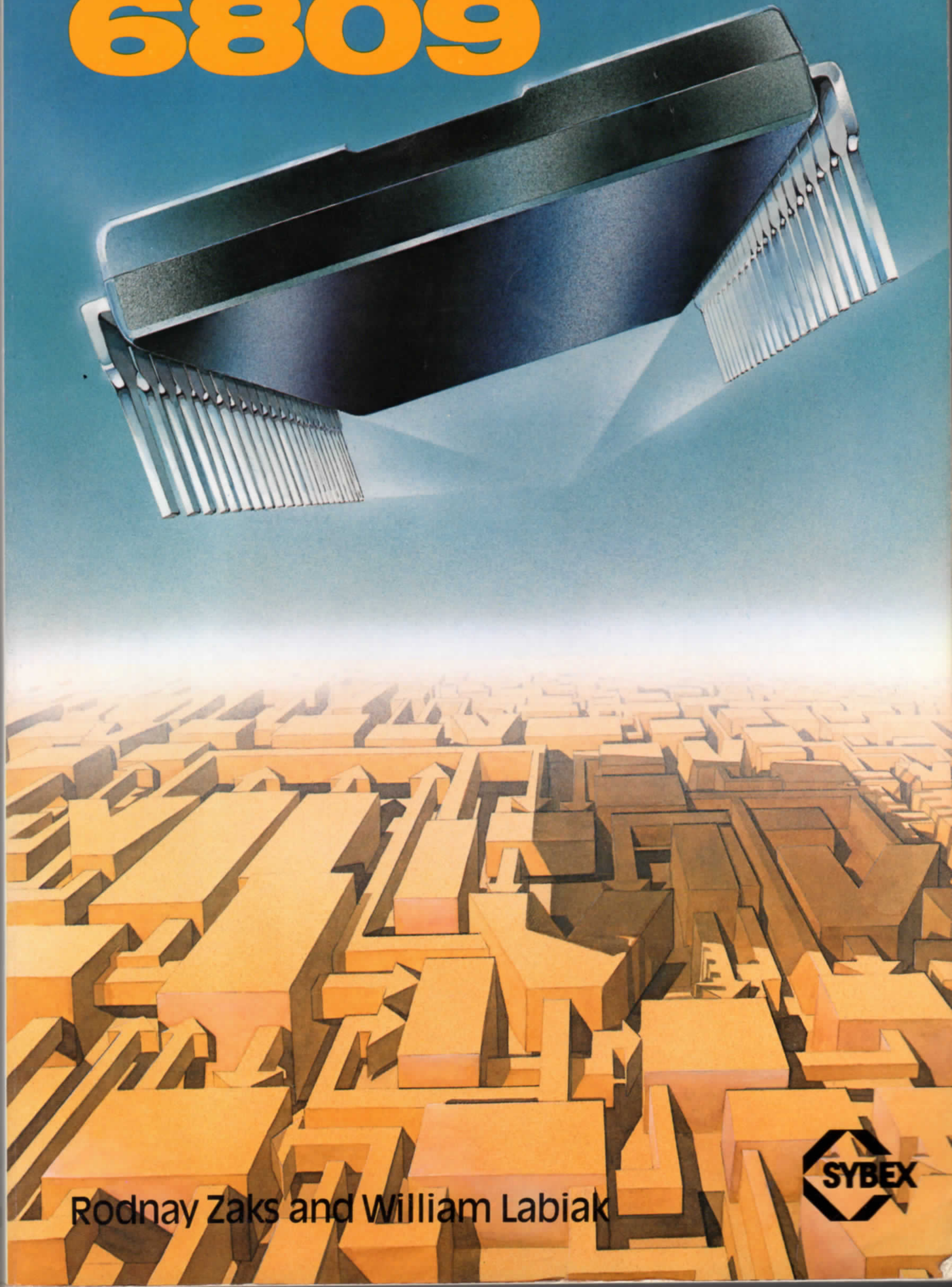
Programming the 6809



Rodnay Zaks and William Labiak

SYBEX

Programming the 6809

You can't just look at the instruction set to program a powerful chip like the 6809; you've got to know more. In this book you will find all the information you need to get the 6809 working for you.

It covers the 6809—inside and out. You'll learn how signals are handled within the chip itself and how to get them to control all essential I/O functions.

Whether you're a first-time or experienced programmer, this book will make it possible for you to use the 6809 to its fullest capacity.

ABOUT THE AUTHORS

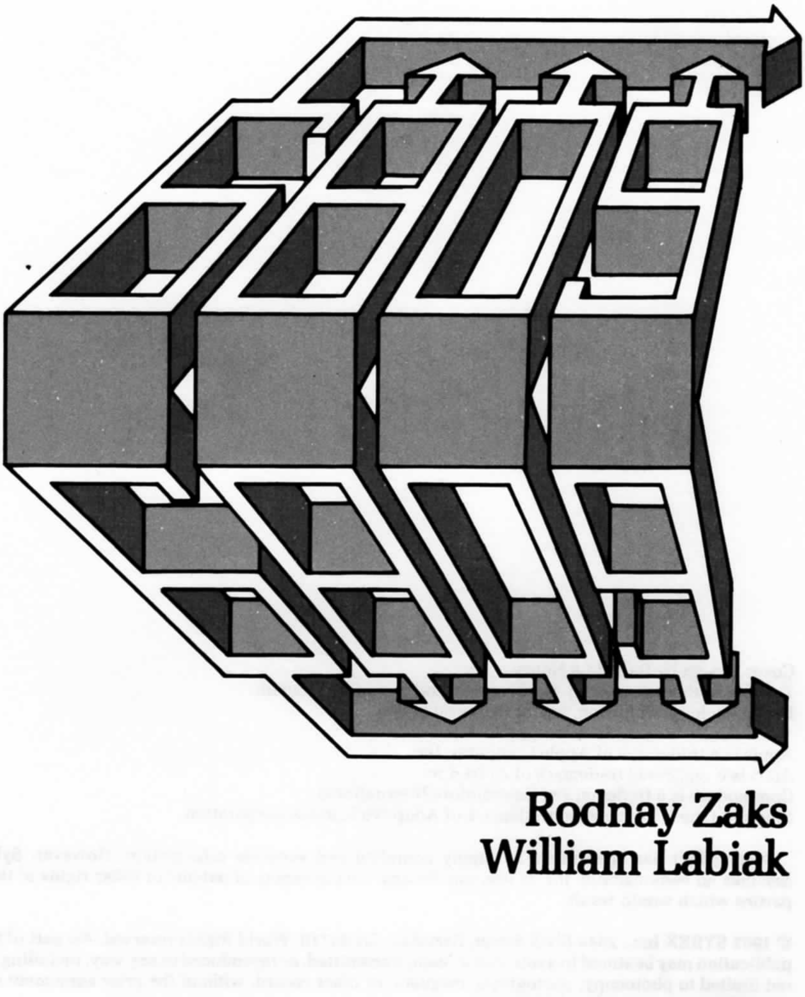
Rodnay Zaks received his Ph.D. in Computer Science from the University of California, Berkeley. A pioneer in the use of microprocessors for industrial applications, Zaks has authored many best-selling books on microprocessors.

Since 1973, **William Labiak** has been involved with mini- and micro-computer applications. He received his Master of Science degree in Electrical Engineering from the University of Illinois. Labiak is now working on the development of large computer networks for industrial control and scientific applications for alternative energy sources.

ISBN 0-89588-078-4

PROGRAMMING THE 6809

PROGRAMMING THE 6809



Rodnay Zaks
William Labiak

SYBEX

Berkeley • Paris • Düsseldorf

Cover Design by Daniel Le Noury
Technical Illustrations by j. trujillo smith and Jeanne E. Tennant.
Design by Judy Wohlfrom, layout by Ingrid Owen.

Every effort has been made to supply complete and accurate information. However, Sybex assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which would result.

Library of Congress Card Number: 82-50621
ISBN 0-89588-078-4
First Edition 1982
Printed in the United States of America
10 9 8 7 6 5 4 3 2

TABLE OF CONTENTS

APPENDICES

ACKNOWLEDGEMENTS

The authors wish to acknowledge the following people who assisted in the preparation of this book. Wayne Fischer, of Motorola, provided the authors with timely information on the 6809 processor. He also reviewed the manuscript and provided helpful suggestions. Samuel Penny, of Adaptive Science Corporation, provided the authors with a 6809 development system for testing the programs for this book. Mike Strang, also of Adaptive Science, reviewed the final manuscript and provided valuable input.

A special acknowledgement is due to Chris Williams for his contribution to the instruction set section in Chapter 4.

The staff of SYBEX have also provided expert help. In particular, Rudolph Langer, Editor-in-Chief, offered valuable suggestions and support; and Salley Oberlin, Sybex editor, provided valuable assistance in the preparation of the text.

INTRODUCTION

If you want to write assembly language programs for any system based on the 6809, this is the book for you. In it you will find:

- Everything you need to know about the organization and instruction set of this exceptionally interesting microprocessor
- A complete presentation of the elements of assembly language programming
- All the essential elementary and intermediate programming techniques that will allow you to begin programming the 6809 on your own.

When you have mastered the material in this book, you will understand how 6809 systems, when properly designed and programmed, can deliver near 16-bit performance with 8-bit economy—and you will have gained the knowledge necessary to make the 6809 do this for you.

Programming the 6809 is organized so that the chapters proceed from the simple to the complex. As you read, you will gradually encounter all the concepts and techniques required to build more and more complex programs, to do more and more advanced tasks.

Chapter 1 introduces you to the basics of programming: what it really is; how to keep track of what you are doing; and what you have to do.

Chapter 2 gives the first run-down on the 6809 processor: the registers and the buses; and how instructions are actually executed within the processor.

Chapter 3 gets you into simple programs and teaches you the kinds of arithmetic the 6809 is capable of, as well as logical operations, and the important concept of subroutines.

Chapter 4 is the big one—a complete description of the 6809 instruction set. After a discussion of the classes of 6809 instructions, we present a detailed explanation of each instruction. We discuss the instructions in alphabetical order for easy reference.

Chapter 5 details one of the most important aspects of the 6809, the addressing modes. We begin this essential treatment with a background discussion on the different kinds of addressing possible in microprocessors, and how they work. We then go on to examine the actual addressing modes in this context. Finally, we give concrete examples of the application of each of the modes, to help you completely understand what they can do.

Chapter 6 covers essential input/output techniques, 6809 style, including: the I/O instruction repertory of the 6809, simple serial and parallel I/O programs, some concrete implementations of common I/O tasks, and more advanced techniques.

Chapter 7 considers several I/O chips that are commonly used to interface the 6809 to the external world.

Chapter 8 gets into more extensive application programs. These programs do all sorts of things. But each shows how the 6809 can make simple and fast, what on other 8-bit microprocessors is cumbersome and slow.

Chapter 9·discusses data structures—another important, though more advanced, area in which the 6809 shines, including pointers, list searching, sorting, and more complex programs and techniques.

Chapter 10 concludes the book with a forward look at the world of program development that is now open to you. We compare and evaluate hexidecimal 'machine language' coding, assemblers and high-level language. We also touch on some available, and more or less desirable, program development environments.

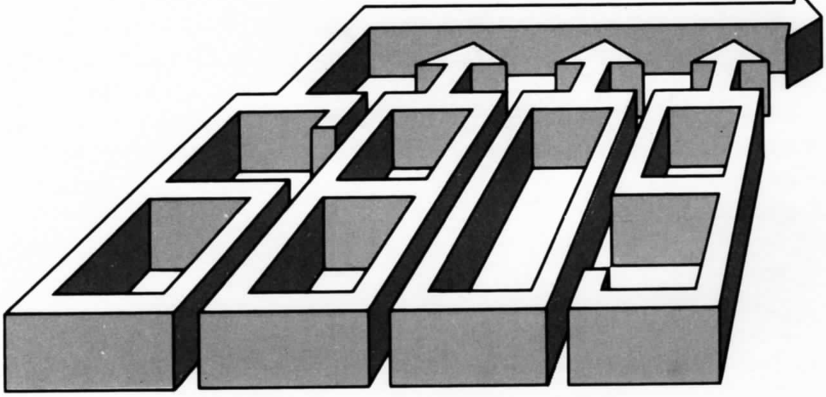
Several useful appendices and an index bring the book to an end.

Most of the programs in this book were tested on a Modulas One® single-board computer using the AS04 assembler. Those valuable resources are products of Adaptive Science Corporation of Emeryville, California, who most graciously made them available to one of the authors (William Labiak) for the development of this book. We both thank them for thus assuring the accuracy of the programs used throughout this book.

Rodnay Zaks
William Labiak

Berkeley, California
Spring 1982

CHAPTER 1



BASIC CONCEPTS

IN THIS CHAPTER, we introduce the basic concepts and definitions used in computer programming. If you are already familiar with these concepts, you may only want to glance quickly at the contents of this chapter, and then move on to Chapter 2. We suggest, however, that you read through this chapter, even if you are an experienced programmer, in order to familiarize yourself with the approach we will be using throughout this book.

WHAT IS PROGRAMMING?

Given a problem, one normally tries to devise a solution. This solution, expressed as a step-by-step procedure, is called an *algorithm*. An algorithm may be expressed in any language or symbolism, and it must terminate in a finite number of steps. Here is a simple example of an algorithm:

1. insert key in the keyhole
2. turn key one full turn to the left
3. seize doorknob
4. turn doorknob left and push the door.

At this point, if the algorithm is correct for the type of lock involved, the door will open.

Once a solution to a problem has been expressed in the form of an algorithm, the algorithm can then be executed by a computer. Unfortunately, it is now a well-established fact that computers cannot understand or execute ordinary spoken English or any other human language. The reason for this lies in the *syntactic ambiguities* of all common human languages. Only a well-defined subset of a natural language, called a *programming language,* can be "understood" by a computer. Converting an algorithm into a sequence of instructions in a programming language is called *programming.* The actual translation phase of the algorithm into the programming language is called *coding.* Programming refers not just to the coding, but also to the overall design of the programs and "data structures" that will implement the algorithm.

Effective programming requires not only an understanding of the possible implementation techniques for standard algorithms, but also the skillful use of computer hardware resources (such as internal registers, memory, and peripheral devices), and a creative use of appropriate data structures. We cover these techniques in the following chapters.

Programming also requires a strict documentation discipline. Well-documented programs are understandable to others, as well as to the author. Documentation must be both internal and external to the program. *Internal program documentation* refers to the comments used in the body of a program to explain its operation. *External documentation* refers to the design documents that are separate from the program, including, written explanations, manuals, and flowcharts.

An intermediate step is almost always taken between the designing of the *algorithm* and the *program.* It is called *flowcharting.*

FLOWCHARTING

A flowchart is simply a symbolic representation of an algorithm, expressed as a sequence of rectangles and diamonds. On the flowchart rectangles are used to represent *commands* (or executable statements) and diamonds are used for *tests* such as: If information X is true, then take action A, else B. Figure 1.1 shows an example of a flowchart. We will not present a formal definition of flowcharts at this point; we discuss flowcharting in more detail in Chapter 3.

Flowcharting is a highly recommended intermediate step between the specification of the algorithm and the actual coding of the solution. Remarkably, it has been observed that perhaps 10% of the programming population can write a program successfully, without having to flowchart. Unfortunately, it has also been observed that 90% of the population believes it belongs to this 10%! The result is that, on the average, 80% of these programs will fail the first time they are run on a computer. (These percentages are naturally not meant to be accurate.) In short, most novice programmers seldom see the necessity for drawing a flowchart. This usually results in "unclean" or erroneous programs and the programmer must then spend a long time testing and correcting his or her program. (This is called the *debugging* phase.) The discipline of flowcharting is, therefore, highly recommended in all cases. It requires a small amount of additional time prior to the coding, but it usually results in a clear program that executes correctly and quickly. Once flowcharting is well understood, a small percentage of programmers can perform this step mentally, without using paper. Unfortunately, in such cases, the programs they write are usually difficult for anyone else to

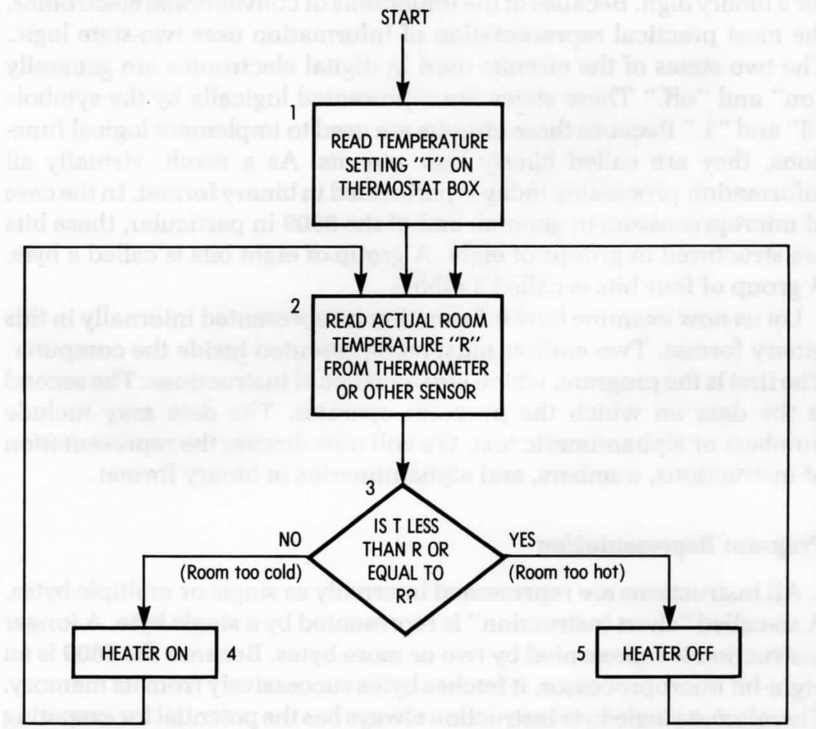

START
1
READ TEMPERATURE SETTING "T" ON THERMOSTAT BOX
2
READ ACTUAL ROOM TEMPERATURE "R" FROM THERMOMETER OR OTHER SENSOR
3
IS T LESS THAN R OR EQUAL TO R?
NO
(Room too cold)
YES
(Room too hot)
HEATER ON
4
5
HEATER OFF
(Optional delay)
(Optional delay)

Figure 1.1: A Flowchart for Keeping Room Temperature Constant

understand, since the documentation provided by the flowchart is not available. As a result, it is universally recommended that flowcharting be used as a strict discipline for any program more than ten or fifteen instructions long. Many examples of flowcharting are provided throughout this book.

INFORMATION REPRESENTATION

All computers manipulate information in the form of numbers or characters. We will now examine the external and internal representations of information on a computer.

Internal Representation of Information

All information is stored in a computer as groups of bits. A *bit* stands for a *binary digit*. Because of the limitations of conventional electronics, the most practical representation of information uses two-state logic. The two states of the circuits used in digital electronics are generally "on" and "off." These states are represented logically by the symbols "0" and "1." Because these circuits are used to implement logical functions, they are called *binary logic circuits*. As a result, virtually all information processing today is performed in binary format. In the case of microprocessors in general, and of the 6809 in particular, these bits are structured in groups of eight. A group of eight bits is called a *byte*. A group of four bits is called a *nibble*.

Let us now examine how information is represented internally in this binary format. Two entities must be represented inside the computer. The first is the program, which is a sequence of instructions. The second is the data on which the program operates. The data may include numbers or alphanumeric text. We will now discuss the representation of instructions, numbers, and alphanumerics in binary format.

Program Representation

All instructions are represented internally as single or multiple bytes. A so-called "short instruction" is represented by a single byte. A longer instruction is represented by two or more bytes. Because the 6809 is an eight-bit microprocessor, it fetches bytes successively from its memory. Therefore, a single-byte instruction always has the potential for executing faster than a two- or three-byte instruction. We will see later on that this is an important feature of the instruction set of any microprocessor, and of the 6809 in particular. However, the limitation to 8 bits in length has resulted in important restrictions, which we will outline later on in this chapter. This limitation is a classic example of a compromise that often has to be made between speed and flexibility in programming. The binary code used to represent instructions is dictated by the manufacturer. The 6809, like any other microprocessor, comes equipped with a fixed instruction set. The instructions for the 6809 are listed with their codes in Appendix D. A program is expressed as a sequence of these binary instructions.

Representing Numeric Data

Representing numbers in binary is not a straightforward task: several cases must be distinguished. We must be able to represent whole numbers, then signed numbers, i.e., positive and negative numbers or integers, and finally, numbers with a decimal point. Let us now address these requirements and possible solutions.

Integers may be represented using a *direct binary* representation. The direct binary representation is simply the representation of the decimal value of a number in the binary system. In the binary system, the right-most bit represents 2 to the power 0. The next one to the left represents 2 to the power 1, the next one represents 2 to the power 2, and the left-most bit represents 2 to the power 7 = 128. For example,

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

represents

$$b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0$$

The powers of 2 are:

$$2^7 = 128,\ 2^6 = 64,\ 2^5 = 32,\ 2^4 = 16,\ 2^3 = 8,\ 2^2 = 4,\ 2^1 = 2,\ 2^0 = 1$$

The binary representation is analogous to the decimal representation of numbers, where 123 represents:

$$\begin{array}{rrl} 1\times & 100 = & 100 \\ +2\times & 10 = & 20 \\ +3\times & 1 = & 3 \\ \hline & = & 123 \end{array}$$

Note that $100 = 10^2$, $10 = 10^1$, $1 = 10^0$. In this positional notation, each digit represents a power of 10. In the binary system, each binary digit or bit represents a power of 2, instead of a power of 10 as in the decimal system. Let's look at an example of binary. 00001001 in binary represents:

$$
\begin{array}{rrl}
1 \times & 1 = 1 & (2^0) \\
0 \times & 2 = 0 & (2^1) \\
0 \times & 4 = 0 & (2^2) \\
1 \times & 8 = 8 & (2^3) \\
0 \times & 16 = 0 & (2^4) \\
0 \times & 32 = 0 & (2^5) \\
0 \times & 64 = 0 & (2^6) \\
0 \times & 128 = 0 & (2^7) \\
\hline
\text{in decimal:} & = 9 &
\end{array}
$$

Let's look at some other examples. 10000001 represents:

$$
\begin{array}{rrr}
1 \times & 1 = & 1 \\
0 \times & 2 = & 0 \\
0 \times & 4 = & 0 \\
0 \times & 8 = & 0 \\
0 \times & 16 = & 0 \\
0 \times & 32 = & 0 \\
0 \times & 64 = & 0 \\
1 \times & 128 = & 128 \\
\hline
\text{in decimal:} & = & 129
\end{array}
$$

Therefore, 10000001 represents the decimal number 129. By examining the binary representation of numbers, it is easy to understand why bits are numbered from 0 to 7, going from right to left. Bit 0 is b_0 and corresponds to 2^0. Bit 1 is b_1 and corresponds to 2^1, and so on. The binary equivalents of the numbers from 0 to 255 are shown in Figure 1.2.

Decimal to Binary Conversely, we will now compute the binary equivalent of 11 decimal:

$$
\begin{array}{rl}
11 \div 2 = 5 \text{ remains } 1 \rightarrow 1 & \text{(lowest bit)} \\
5 \div 2 = 2 \text{ remains } 1 \rightarrow 1 & \\
2 \div 2 = 1 \text{ remains } 0 \rightarrow 0 & \\
1 \div 2 = 0 \text{ remains } 1 \rightarrow 1 & \text{(highest bit)}
\end{array}
$$

The binary equivalent is 1011 (read the right-most column from bottom to top). The binary equivalent of a decimal number may be obtained by dividing successively by 2, until a quotient of 0 is obtained.

Operating on Binary Data The arithmetic rules for binary numbers are straightforward. The rules for addition are:

$$
\begin{aligned}
0 + 0 &= \quad 0 \\
0 + 1 &= \quad 1 \\
1 + 0 &= \quad 1 \\
1 + 1 &= (1)\,0
\end{aligned}
$$

where (1) denotes a "carry" of 1 (note that 10 is the binary equivalent of 2 decimal). Binary subtraction can be performed by "adding the complement." We will discuss binary subtraction once we learn how to

Decimal	Binary	Decimal	Binary
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	•	
11	00001011	127	01111111
12	00001100	128	10000000
13	00001101	129	10000001
14	00001110		
15	00001111	•	
16	00010000	•	
17	00010001	•	
•			
•			
•		254	11111110
31	00011111	255	11111111

Figure 1.2: Decimal-Binary Table

represent negative numbers. Let's consider the following example involving addition:

```
   (2)       10
 + (1)     + 01
 -----     ----
 = (3)       11
```

Addition is performed just as in decimal, by adding the columns from right to left. First you add the right-most column:

```
                10
               +01
               ---
  (0 + 1   =    1. No carry.)
```

Then the next column:

```
           10
          +01
          ---
           11 (1 + 0 = 1. No carry.)
```

Let us now look at other examples of binary addition:

```
   0010        (2)        0011        (3)
 + 0001        (1)      + 0001        (1)
 ------------------     ------------------
 = 0011        (3)      = 0100        (4)
```

The last example illustrates the role of the carry. Looking at the right-most bits: 1 + 1 = (1) 0. A carry of 1 is generated, which must be added to the next bits:

```
   001 — column 0 has just been added
 +000 —
 +   1    (carry)
 ------
 =(1)0    (where (1) indicates a new carry into column 2)
```

The final result is 0100.

Let's consider another example:

```
   0111        (7)
 +0011       +(3)
 -----------------
   1010      =(10)
```

In this example, a carry is again generated, up to the left-most column.

With eight bits, it is, therefore, possible to directly represent the numbers 00000000 to 11111111, i.e., 0 to 255. Two limitations, however, should be immediately visible. First, we are only representing positive numbers.

Second, the magnitude of these numbers is limited to 255, if we use only eight bits. Let's now address these limitations in turn.

Signed Binary In a signed binary representation, the left-most bit is used to indicate the sign of the number. Traditionally, 0 is used to denote a *positive* number and 1 is used to denote a *negative* number. For example, 11111111 represents −127, while 01111111 represents +127. We can now represent positive and negative numbers, but we have reduced the maximum magnitude of these numbers to 127. As another example, 00000001 represents +1 (the leading 0 is "+", followed by 0000001 = 1) and 10000001 is −1 (the leading 1 is "−").

Let us now address the *magnitude* problem. In order to represent larger numbers, it is necessary to use a larger number of bits. For example, if we use sixteen bits (two bytes) to represent numbers, we will be able to represent numbers from −32K to +32K in signed binary. (1K in computer jargon represents 1,024.) Bit 15 is used for the sign, and the remaining 15 bits (bit 14 through bit 0) are used for the magnitude: $2^{15} = 32K$. If this magnitude is too small, we must use 3 bytes or more.

If we wish to represent large integers, it is necessary to use a larger number of bytes internally. This is why most simple BASIC interpreters, and other languages, provide only a limited precision for integers. This way, they can use a shorter internal format for the numbers they manipulate. Better versions of BASIC and some other languages provide a larger number of significant decimal digits at the expense of a large number of bytes for each number.

Let us now solve another problem: the one of speed efficiency. Let's perform an addition in the signed binary representation we have just introduced. We want to add +7 and −5.

+ 7 is represented by	00000111
− 5 is represented by	10000101
the binary sum is:	10001100, or −12

This is not the correct result. The correct result is +2. We have neglected the fact that in order to use this representation, special actions must be taken, depending on the sign. This results in increased complexity and reduced performance. In other words, the binary addition of signed numbers does not "work correctly." This is annoying. Clearly, the computer must not *only* represent information, but it must also perform arithmetic on it.

The solution to this problem is called the *two's complement* representation. We will use the two's complement representation, instead of *signed binary* representation. In order to introduce two's complement we will first introduce an intermediate step: *one's complement.*

One's Complement In the one's complement representation, all positive integers are represented in their correct binary format. For example +3 is represented as usual by 00000011. However, its complement, −3, is obtained by complementing every bit in the original representation. Each 0 is transformed into a 1 and each 1 is transformed into a 0. In our example, the one's complement representation of −3 is 11111100.

Let's look at another example:

+ 2 is 00000010
− 2 is 11111101

Note that, in this representation, positive numbers start with a 0 on the left, and negative numbers start with a 1 on the left. As a test, let's add −4 and +6:

− 4 is 11111011
+ 6 is 00000110

The sum is: (1) 00000001

where (1) indicates a carry. The correct result should be 2 or 00000010.

Let's try again:

− 3 is 11111100
− 2 is 11111101

The sum is: (1) 11111001

or −6, plus a carry. The correct result is −5. The representation of −5 is 11111010. It did not work.

This representation does represent positive and negative numbers, however, the result of an ordinary addition does not always come out correctly. We will now use another representation. It is evolved from the one's complement and is called the two's complement representation.

Two's Complement Representation In the two's complement representation, positive numbers are represented, as usual, in signed binary, just like in one's complement. The difference lies in the representation of *negative numbers.* A negative number represented in two's complement is obtained by first computing the one's complement and then *adding one.* Let's examine an example:

Example: +3 is represented in signed binary by 00000011. Its one's complement representation is 11111100. The two's complement is obtained by adding one. It is 11111101.

Let's try an addition:

```
    (3)       00000011
  + (5)     + 00000101
  -----------------------
  = (8)     = 00001000
```

The result is correct.

Let's try a subtraction:

```
    (3)     00000011
  (-5) +   11111011
  ---------------------
         = 11111110
```

Now, let's identify the result by computing the two's complement:

```
   (the one's complement of 11111110 is)      00000001
                                (Adding 1)  +        1
                                            -----------
     (therefore, the two's complement is)     00000010   or +2
```

The result 11111110 represents −2. It is correct.

We have now tried addition and subtraction, and the results have been correct (ignoring the carry). It seems that two's complement works!

We will now add +4 and −3 (the subtraction is performed by adding the two's complement):

```
  + 4 is              00000100
  − 3 is              11111101
  ------------------------------
  The result is: (1) 00000001
```

If we ignore the carry, the result is 00000001, i.e., 1 in decimal. This is the correct result. Without giving the complete mathematical proof, we will simply state that this representation does work. In two's complement, it is possible to add or subtract signed numbers, regardless of the sign. Using the usual rules of binary addition, the result comes out correct, including the sign. The carry is ignored. This is a very significant advantage. If this were not the case, we would have to correct the result for sign every time, causing a much slower addition or subtraction time.

For the sake of completeness, let us state that two's complement is simply the most convenient representation to use for simpler processors, such as microprocessors. On more complex processors, other representations may be used. For example, one's complement may be used, but if one's complement is used, special circuitry is required to "correct the result."

From this point on, all signed integers will be implicitly represented internally in two's complement notation. See Figure 1.3 for a table of two's complement numbers.

We will now offer examples that demonstrate the rules of two's complement. In particular, C denotes a possible carry (or borrow) condition. (It is bit 8 of the result.) V denotes a two's complement overflow, i.e., when the sign of the result is changed "accidentally," because the numbers are too large. It is an essentially internal carry from bit 6 to bit 7 (the sign bit). This will be clarified below.

Let us now demonstrate the role of the carry C and the overflow V.

The Carry C Here is an example of a carry:

```
      (128)          10000000
    + (129)        + 10000001
    ------------------------
      (257)= (1)     00000001
```

where (1) indicates a carry. The result requires a ninth bit (bit 8, since the right-most bit is 0). It is the carry bit.

If we assume that the carry is the ninth bit of the result, we recognize the result as binary 100000001 = 257. However, the carry must be recognized and handled with care. Inside the microprocessor, the registers used to hold information are generally only eight-bits wide. When storing the result, only bits 0 to 7 will be preserved.

A carry, therefore, always requires special action. It must be detected by special instructions, then processed. Processsing the carry means either storing it somewhere (with a special instruction), ignoring it, or deciding that it is an error (if the largest authorized result is 11111111).

Overflow V Here's an example of overflow:

```
 bit 6 ──────┐
 bit 7 ─────┐│
            ↓↓
            01000000        (64)
          + 01000001   +    (65)
         -------------------------
          = 10000001   =(−127)
```

An internal carry has been generated from bit 6 into bit 7. This is called an *overflow*. The result is now negative, "by accident." This situation must be detected, so that it can be corrected.

+	Two's complement code	−	Two's complement code
+ 127	01111111	− 128	10000000
+ 126	01111110	− 127	10000001
+ 125	01111101	− 126	10000010
. . .		− 125	10000011
		. . .	
+ 65	01000001	− 65	10111111
+ 64	01000000	− 64	11000000
+ 63	00111111	− 63	11000001
. . .		. . .	
+ 33	00100001	− 33	11011111
+ 32	00100000	− 32	11100000
+ 31	00011111	− 31	11100001
. . .		. . .	
+ 17	00010001	− 17	11101111
+ 16	00010000	− 16	11110000
+ 15	00001111	− 15	11110001
+ 14	00001110	− 14	11110010
+ 13	00001101	− 13	11110011
+ 12	00001100	− 12	11110100
+ 11	00001011	− 11	11110101
+ 10	00001010	− 10	11110110
+ 9	00001001	− 9	11110111
+ 8	00001000	− 8	11111000
+ 7	00000111	− 7	11111001
+ 6	00000110	− 6	11111010
+ 5	00000101	− 5	11111011
+ 4	00000100	− 4	11111100
+ 3	00000011	− 3	11111101
+ 2	00000010	− 2	11111110
+ 1	00000001	− 1	11111111
+ 0	00000000		

Figure 1.3: Two's Complement Table

Let us examine another situation:

```
        11111111        (−1)
      + 11111111       +(−1)
   ----------------------------
   =(1) 11111110       =(−2)
     ▼
    carry
```

In this case, an internal carry has been generated from bit 6 into bit 7, and also from bit 7 into C. The rules of two's complement arithmetic specify that this carry should be ignored. The result is then correct. This is because the carry from bit 6 to bit 7 did not change the sign bit.

The carry from bit 6 into bit 7 is not an *overflow* condition. When operating on negative numbers, the overflow is not simply a carry from bit 6 into bit 7. Let's examine one more example:

```
        11000000         (−64)
      + 10111111         (−65)
   ----------------------------
   = (1) 01111111       (+127)
      ▼
     carry
```

This time, there has been no internal carry from bit 6 into bit 7, but there has been an external carry. The result is incorrect, as bit 7 has been changed. An overflow condition should be indicated.

Overflow will occur in four situations, including:

1. the addition of large positive numbers
2. the addition of large negative numbers
3. the subtraction of a large positive number from a large negative number
4. the subtraction of a large negative number from a large positive number.

Let us now improve our definition of the overflow.

Technically, the overflow indicator, a special bit reserved for this purpose, and called a *condition code,* will be set when there is a carry from bit 6 into bit 7, and there is no external carry. It will also be set when there is no carry from bit 6 into bit 7, but there is an external carry. This indicates that bit 7, i.e., the sign of the result, has been accidentally changed. For the technically-minded reader, the overflow flag is set by Exclusive ORing the carry-in and carry-out of bit 7 (the sign bit). Practically every microprocessor is supplied with a special overflow flag to automatically detect this condition—a condition that requires corrective action.

Overflow indicates that the result of an addition or subtraction requires more bits than are available in the standard 8-bit register used to contain the result.

The Carry and the Overflow The carry and the overflow bits are called condition codes. They are provided in every microprocessor. We will learn to use them for effective programming in Chapter 2. These two indicators are located in a special register called the flags or "status" register. This register also contains additional indicators (as described in Chapter 4).

Examples We will now look at actual examples that illustrate the operation of the carry and the overflow. In each example, the symbol V denotes the overflow, and C denotes the carry. If there has been no overflow, V = 0. If there has been an overflow, V = 1. (The same is true for the carry C.) Remember that the rules of two's complement specify that the carry be ignored. (The mathematical proof is not supplied here.) Let's examine the following examples:

Positive-Positive

```
  00000110       (+6)
+ 00001000       (+8)
-----------------------
= 00001110       (+14)  V:0   C:0
(CORRECT)
```

Positive-Positive with Overflow

```
  01111111       (+127)
+ 00000001         (+1)
-----------------------
= 10000000       (−128)  V:1   C:0
The above is invalid because an overflow has occurred.
(ERROR)
```

Positive-Negative (result positive)

```
     00000100       (+4)
+    11111110       (−2)
-------------------------
= (1)00000010       (+2)  V:0   C:1 (disregard)
(CORRECT)
```

Positive-Negative (result negative)

```
  00000010      (+2)
+ 11111100      (−4)
= 11111110      (−2)  V:0  C:0
(CORRECT)
```

Negative-Negative

```
     11111110      (−2)
+    11111100      (−4)
= (1)11111010      (−6)   V:0   C:1 (disregard)
(CORRECT)
```

Negative-Negative with Overflow

```
     10000001      (−127)
+    11000010       (−62)
= (1)01000011       (+67)   V:1   C:1
(ERROR)
```

In the last example, an "underflow" has occurred, by adding two large negative numbers. The result is −189, which is too large to reside in eight bits.

Fixed Format Representation We now know how to represent signed integers; however, we have not yet resolved the problem of magnitude. If we want to represent larger integers, we will need several bytes. In order to perform arithmetic operations efficiently, it is necessary to use a fixed number of bytes, rather than a variable number. Therefore, once the number of bytes is chosen, the maximum magnitude of the number that can be represented is fixed.

The Magnitude Problem When adding numbers we have restricted ourselves to eight bits, because the processor we are using operates internally on eight bits at a time. However, this restricts us to the numbers in the range −128 to +127. Clearly, this is not sufficient for many applications.

Multiple precision can be used to increase the number of digits that can be represented. A two-, three-, or N-byte format can then be used.

For example, let's examine a 16-bit, "double-precision" format:

00000000	00000000	is 0
00000000	00000001	is 1
...		
01111111	11111111	is 32767
11111111	11111111	is −1
11111111	11111110	is −2

However, this method will result in disadvantages. When adding two numbers, for example, we will generally have to add them eight bits at a time, as explained in Chapter 3. This results in slower processing. Also, this representation uses 16 bits for any number, even if it could be represented with only eight bits. It is, therefore, common to use the smallest number of bytes possible.

Let us consider the following important point: the number of bits, n, chosen for the two's complement representation is usually fixed for that program. If any result or intermediate computation should generate a number that requires more than n bits, some bits will be lost. The program normally retains the n left-most bits (the most significant) and drops the low-order ones. This is called *truncating* the result.

Let's look at an example in the decimal system, using a six digit representation:

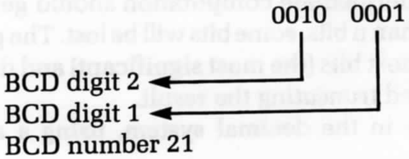
```
   123456
    × 1.2
  -------
   246912
  123456
  -------
= 148147.2
```

The result requires 7 digits. The 2 after the decimal point will be dropped, and the final result will be 148147. It has been truncated. Usually, as long as the position of the decimal point is not lost, this method is used to extend the range of the operations that can be performed, at the expense of precision. (The details of binary multiplication are given in Chapter 3.) The problem is the same in binary. This fixed-format representation may cause a loss of precision, but it may be sufficient for usual computations or mathematical operations.

Unfortunately, in the case of accounting, no loss of precision is tolerable. For example, if a customer rings up a large total on a cash register, it would not be acceptable to have a five figure total that would be approximated to the dollar. Thus, another representation must be used whenever precision in the result is essential. The solution normally used is *BCD*, or binary-coded decimal.

BCD Representation The principle used in representing numbers in BCD is to encode each decimal digit separately and to use as many bits as necessary to represent the complete number exactly. In order to encode each of the digits from 0 through 9, four bits are necessary. Three bits supply only eight combinations, and therefore, cannot encode the ten digits. Four bits allow sixteen combinations and are, therefore, sufficient to encode the digits 0 through 9. It can also be noted that six of the possible codes will not be used in the BCD representation (see Figure 1.4). This will result later on in a potential problem, when performing additions and subtractions. Since only four bits are needed to encode a BCD digit, two BCD digits may be encoded in every byte. This is called *packed BCD*. As an example, 00000000 is 00 in BCD. 10011001 is 99.

A BCD code is read as follows:

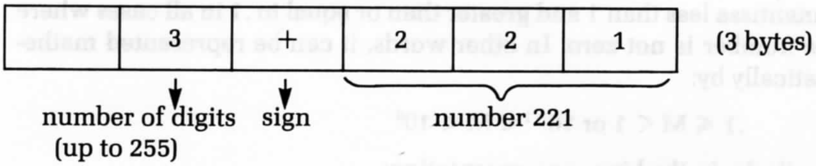


As many bytes as necessary will be used to represent all BCD digits. Typically, one or more nibbles will be used at the beginning of the representation to indicate the total number of nibbles, i.e., the total number of BCD digits used. Another nibble or byte will be used to denote the

CODE	BCD SYMBOL	CODE	BCD SYMBOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	unused
0011	3	1011	unused
0100	4	1100	unused
0101	5	1101	unused
0110	6	1110	unused
0111	7	1111	unused

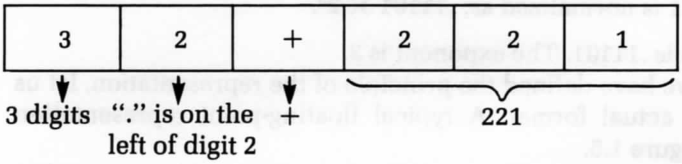
Figure 1.4: BCD Table

position of the decimal point. However, conventions may vary. Here is an example of a representation for multibyte BCD integers:



This example represents +221. (The sign may be represented by 0000 for +, and 0001 for −, for example.)

The BCD representation can easily accommodate decimal numbers. For example, +2.21 may be represented by:



The advantage of BCD is that it yields absolutely correct results. Its disadvantage is that it uses a large amount of memory and results in slow arithmetic operations. This is acceptable only in an accounting environment, but BCD is normally not used in other cases.

We have now solved the problems associated with the representation of integers, signed integers, and large integers. We have even presented one possible method of representing decimal numbers, with BCD representation. Let us now examine the problem of representing decimal numbers in fixed length format.

Floating-Point Representation The basic principle of floating point representation is that decimal numbers are represented with a fixed length format. In order not to waste bits, the representation will *normalize* all the numbers. For example, 0.000123 wastes three zeroes on the left before non-zero digits. These zeroes have no meaning except to indicate the position of the decimal point. Normalizing this number results in $.123 \times 10^{-3}$. .123 is the *normalized mantissa;* −3 is the *exponent.* We have normalized this number by eliminating all the meaningless zeroes to the left of the first non-zero digit and by adjusting the exponent. Let's consider another example.

Example: 22.1 is normalized as $.221 \times 10^{2}$. The general form of floating-point representation is M × 10^E, where M is the mantissa, and E is the exponent.

It can be readily seen that a normalized number is characterized by a mantissa less than 1 and greater than or equal to .1 in all cases where the number is not zero. In other words, it can be represented mathematically by:

$$.1 \leqslant M < 1 \text{ or } 10^{-1} \leqslant M < 10^0$$

Similarly, in the binary representation:

$$2^{-1} \leqslant M < 2^0 \text{ (or } .5 \leqslant M < 1)$$

where M is the absolute value of the mantissa (disregarding the sign). For example:

111.01 is normalized as: .11101 × 2^3.

The mantissa is .11101. The exponent is 3.

Now that we have defined the principle of the representation, let us examine the actual format. A typical floating-point representation appears in Figure 1.5.

In the representation in Figure 1.5, four bytes are used for a total of 32 bits. The first byte on the left of the illustration is used to represent the exponent. Both the exponent and the mantissa will be represented in two's complement. As a result, the maximum exponent will be −128. "S" in Figure 1.5 denotes the sign bit.

Three bytes are used to represent the mantissa. Since the first bit in the two's complement representation indicates the sign, this leaves 23 bits for the representation of the magnitude of the mantissa.

This is only one example of a floating point representation. It is possible to use only three bytes, or it is possible to use more. The four-byte representation proposed above is a common one that represents a reasonable compromise in terms of accuracy, magnitude of numbers, storage utilization, and efficiency in arithmetic operation.

We have now explored the problems associated with the representation

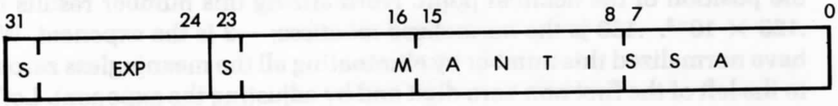


Figure 1.5: Typical Floating-Point Representation

of numbers and have learned how to represent them in integer form, with a sign, or in decimal form. Let's now go on to examine how to represent alphanumeric data internally.

Representing Alphanumeric Data

The representation of alphanumeric data, i.e., characters, is completely straightforward: all characters are encoded in an eight-bit code. Only two codes are in general use in the computer world, the ASCII Code and the EBCDIC Code. ASCII stands for "American Standard Code for Information Interchange," and is universally used in the world of microprocessors. EBCDIC is a variation of ASCII used by IBM, and is, therefore, not used in the microcomputer world unless one interfaces to an IBM terminal.

Let us briefly examine the ASCII encoding. We encode 26 letters of the alphabet for both upper and lower case, plus 10 numeric symbols, and perhaps 20 additional special symbols. This can be easily accomplished with 7 bits, which allow 128 possible codes. (See Figure 1.6.) All characters are, therefore, encoded in 7 bits. The 8th bit, when it is used, is the *parity bit.* Parity is a technique for verifying that the contents of a byte have not been accidentally changed. The number of 1's in the byte are counted and the 8th bit is set to one if the count was odd, thus making the total even. This is called *even parity. Odd parity,* i.e., writing the 8th bit (the left-most bit) so that the total number of 1's in the byte is odd, can also be used.

As an example, let us compute the parity bit for 0010011, by using even parity. The number of 1's is 3. The parity bit must, therefore, be a 1, so that the total number of bits is 4, i.e., even. The result is 10010011, where the leading 1 is the parity bit and 0010011 identifies the character.

The table of 7-bit ASCII codes is shown in Figure 1.6. In practice, it is used "as is," i.e., without parity, by adding a 0 in the left-most position, or else with parity, by adding the appropriate extra bit on the left.

In specialized situations, such as telecommunications, other codings, such as error-correcting codes, may be used. However, descriptions of these codings are beyond the scope of this book.

Now that we have examined the usual representations for both program and data inside the computer, let us examine the possible external representations.

External Representation of Information

The external representation of information refers to the way information is presented to the *user,* i.e., generally to the programmer.

Information may be presented externally in essentially three formats: binary, octal or hexadecimal, and symbolic. Let's examine these formats.

1. Binary We have seen that information is stored internally in *bytes,* which are sequences of eight *bits* (0's or 1's). It is sometimes desirable to display this internal information directly in its binary format—this is known as *binary representation.* A simple example is provided by Light Emitting Diodes (LEDs), which are essentially miniature lights on the front panel of a microcomputer. In the case of an 8-bit microprocessor, a front panel will typically be equipped with eight LEDs to display the contents of any internal register. A lighted LED indicates a 1. An unlighted LED indicates a 0. Such a binary representation may be used for the fine debugging of a complex program, especially if it involves input/output, but is naturally impractical at the human level. This is because, in most cases, it is easier to look at information in symbolic form. For example, 9 is much easier to understand and to remember than 1001. More convenient representations have been devised, that improve the interface between people and machines.

2. Octal and Hexadecimal Octal and hexadecimal encode three and four binary bits, respectively, into a unique symbol. Octal is a format using three bits, where each combination of three bits is represented by a symbol between 0 and 7. (See Figure 1.7.)

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	—	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	—	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

—Figure 1.6: ASCII Conversion Table (See Appendix B for Abbreviations.)

For example, 00 100 100 binary is represented by:

▼ ▼ ▼
0 4 4

or 044 in octal.

As another example: 11 111 111 is:

▼ ▼ ▼
3 7 7

or 377 in octal.

Conversely, the octal 211 represents

010 001 001

or 10001001 binary.

Octal has traditionally been used on older computers that employ various numbers of bits, ranging from 8 to, perhaps, 64. More recently, with the dominance of eight-bit microprocessors, the eight-bit format has become the standard, and another, more practical, representation is used—hexadecimal representation.

In the hexadecimal representation, a group of four bits is encoded as one hexadecimal digit. Hexadecimal digits are represented by the symbols from 0 to 9, and by the letters A, B, C, D, E, F. For example, 0000 is represented by 0; 0001 is represented by 1; and 1111 is represented by the letter F (see Figure 1.8).

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Figure 1.7: Octal Symbols

For example, $\underbrace{1010}_{A}\ \underbrace{0001}_{1}$ in binary is represented by

in hexadecimal.

Hexadecimal offers the advantage of encoding eight bits into only two digits. This is easier to visualize or memorize and faster to type into a computer than its binary equivalent. Therefore, on most new microcomputers, hexadecimal is the preferred method of representation for groups of bits.

Naturally, whenever the information present in the memory has a meaning, such as representing text or numbers, hexadecimal is not convenient for representing the meaning of this information for a human user.

Symbolic Representation *Symbolic representation* refers to the external representation of information in actual symbolic form. For example, decimal numbers are represented as decimal numbers, and not as sequences of hexadecimal symbols or bits. Similarly, text is represented as

DECIMAL	BINARY	HEX	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Figure 1.8: Hexadecimal Codes

such. Naturally, symbolic representation is most practical to the user. It is used whenever an appropriate display device is available, such as a CRT display or a printer. (A CRT display is a television-type screen used to display text or graphics.) Unfortunately, in smaller systems, such as one-board microcomputers, it is uneconomical to provide such displays, and the user is restricted to hexadecimal communication with the computer.

Summary of External Representations

Symbolic representation of information is the most desirable, since it is the most natural for a human user. However, it requires an expensive interface in the form of an alphanumeric keyboard, plus a printer or a CRT.display. For this reason, it may not be available on the less expensive systems. An alternative type of representation is then used, and in such a case, hexadecimal is the dominant representation. Only in rare cases, relating to fine debugging at the hardware or software level, is the binary representation used. *Binary* directly displays the contents of the registers or memory in binary format.

Now that we have seen how information is represented internally and externally, let's go on to examine the actual microprocessor that manipulates this information.

EXERCISES

1-1: *What is the decimal value of 11111100?*

1-2: *What is the binary for 257?*

1-3: *Convert 19 to binary, then back to decimal.*

1-4: *Compute 5 + 10 in binary. Verify that the result is 15.*

1-5: *Compute the result of:*

```
  1111
+ 0001
------
```

Does the result fit into four bits?

1-6: *What is the representation of −5 in signed binary?*

1-7: *The representation of +6 is 00000110. What is the representation of −6 in one's complement?*

1-8: *What is the two's complement representation of +127?*

1-9: *What is the two's complement representation of −128?*

1-10: *What are the smallest and the largest numbers that can be represented in two's complement notation, using only one byte?*

1-11: *Compute the two's complement of 20. Then compute the two's complement of your result. Do you find 20 again?*

1-12: *Complete the following additions. Indicate the result, the carry C, the overflow V, and whether the result is correct or not:*

```
  10111111    (___)             11111010    (___)
+ 11000001    (___)           + 11111001    (___)
= ________    V:__ C:__       = ________    V:__ C:__

__CORRECT     __ERROR         __CORRECT     __ERROR
```

```
   00010000   (____)            01111110   (____)
 + 01000000   (____)          + 00101010   (____)
 = ________   V:__ C:__       = ________   V:__ C:__

 __CORRECT    __ERROR         __CORRECT    __ERROR
```

1-13: *Can you show an example of overflow when adding a positive and a negative number? Why or why not?*

1-14: *What are the largest and the smallest numbers that can be represented in two bytes, using two's complement?*

1-15: *What is the largest negative integer that can be represented in a two's complement triple-precision format?*

1-16: *What is the BCD representation for 29? For 91?*

1-17: *Is 10100000 a valid BCD representation? Why or why not?*

1-18: *Using the same convention, represent −23123. Show it in BCD format, as above, then in binary.*

1-19: *Show the BCD for 222 and 111, then for the result of 222 × 111 (Compute the result by hand, then show it in the above representation.)*

1-20: *How many bits are required to encode 9999 in BCD? And in two's complement?*

1-21: *How many decimal digits can the mantissa represent with the 23 bits?*

1-22: *Compute the 8-bit representation of the digits 0 through 9, using even parity. (This code will be used in application examples of Chapter 8.)*

1-23: *Complete Exercise 1-22 for the letters A through F.*

1-24: *Using a non-parity ASCII code (where the left-most bit is 0), indicate the binary codes of the 4 characters below:*

A
?
3
b

1-25: *What is the hexadecimal representation of 10101010?*

1-26: *Conversely, what is the binary equivalent of FA hexadecimal?*

1-27: *What is the octal representation of 01000001?*

1-28: *What is the advantage of two's complement over other representations used to represent signed numbers?*

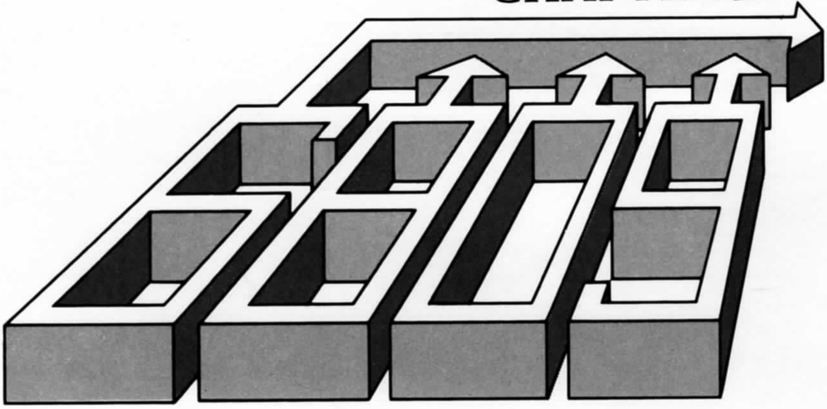
1-29: *How would you represent 1024 in direct binary? Signed binary? Two's complement?*

1-30: *What is the V-bit? Should the programmer test it after an addition or subtraction?*

1-31: *Compute the two's complement of +16, +17, +18, −16, −17, −18.*

1-32: *Show the hexadecimal representation of the following text, which has been stored internally in ASCII format, with no parity: MESSAGE.*

CHAPTER 2



6809 HARDWARE ORGANIZATION

TO PROGRAM EFFICIENTLY, you must understand the internal structure of the processor you are using. We will begin this chapter with a discussion of the basic architecture of a microcomputer system. We will then examine the internal organization of the 6809. In particular, we will study the registers of the 6809 and their combined operations. This study is particularly important, because the 6809 has an unusually large number and variety of registers.

SYSTEM ARCHITECTURE

Figure 2.1 shows the architecture of a typical microcomputer system. Appearing on the left of the illustration in Figure 2.1 is the *microprocessor unit* (the *MPU*)—in this case, the 6809—which implements the functions of the *central-processing unit* (the *CPU*) on a single chip. The CPU includes an *arithmetic-logical unit* (the *ALU*), plus its internal registers, and a *control unit* (the *CU*), which decodes and internally sequences instructions. (We will discuss the CPU in detail later in this chapter.)

The MPU has three *buses:* an 8-bit bidirectional *data bus* (shown at the top of the illustration in Figure 2.1), a 16-bit unidirectional *address bus,* and a *control bus* (both shown at the bottom of the illustration). We will now study the functions of these buses.

The *data bus* carries the data that is exchanged by the various elements of the system. Typically, it carries the data from the memory to the MPU, from the MPU to the memory, and from the MPU to an input/output chip. (An input/output chip communicates with an external device.)

The *address bus* carries an address, generated by the MPU, which specifies the source or destination of the data that transits along the data bus. The *control bus* carries the various synchronization signals required by the system. Now that we know the purpose of the buses, let's connect the additional components required for a complete system.

Every MPU requires a precise timing reference, which is supplied by a *clock* and a *crystal.* In most "older" microprocessors, the clock-oscillator is external to the MPU and requires an extra chip. In the more recent ones, the clock-oscillator is usually incorporated within the MPU. The quartz crystal, however, because of its bulk, is always external to the system. The crystal and the clock appear on the left of the MPU box in Figure 2.1.

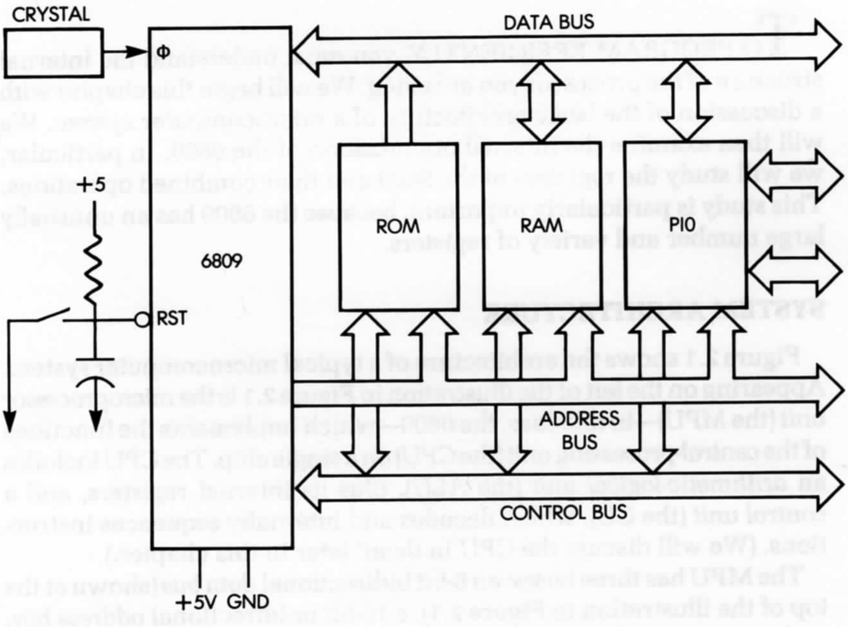


Figure 2.1: A Standard 6809 System

We will now examine the other elements of the system. Going from left to right on the illustration, we see the ROM, the RAM and the PIO.

The *ROM* or *read-only memory* stores the *program* for the system. The advantage of ROM memory is that its contents are permanent, i.e., they do not disappear when the system is turned off. The ROM, therefore, usually contains a *bootstrap* or *monitor* program to permit initial system operation. In a process-control environment, nearly all programs reside in ROM. This is because they will probably never be changed and must be protected against power failures (i.e., they must not be volatile).

RAM (random-access memory) is the read/write memory for the system. In a hobbyist or program-development environment, most of the programs reside in RAM, so that they can be easily changed. Such programs may be kept in RAM, or transferred into ROM, if desired. RAM, however, is volatile. Its contents are lost when power is turned off. In a control system, the amount of RAM is typically small (for data only); however, in a program-development environment, the amount of RAM is large, as it contains programs, plus development software. All RAM contents must be loaded, prior to use, from an external device.

Finally, a system also contains one or more interface chips, so that it can communicate with the external world. The most frequently used interface chip is the PIO or *parallel input/output* chip (shown in Figure 2.1). The PIO, like the other chips in the system, connects to all three buses and provides at least two 16-bit ports for communication with the outside world. For simplicity, the connections between the control bus and the various chips do not appear in Figure 2.1.

The functional modules just described need not necessarily reside on a single LSI chip. In fact, we could use *combination chips,* which may include both the PIO and a limited amount of ROM or RAM.

To build an actual system, we need even more components. In particular, we may need to *buffer* the buses. Also, we may need *decoding logic* for the memory RAM chips, and, finally, we may use *drivers* to amplify signals. These auxiliary circuits will not be described here, as they are not relevant to programming. For more information on specific assembly and interfacing techniques, see reference C207 in the bibliography, and for specific information regarding the 6809 system, see Chapter 7.

INSIDE A MICROPROCESSOR

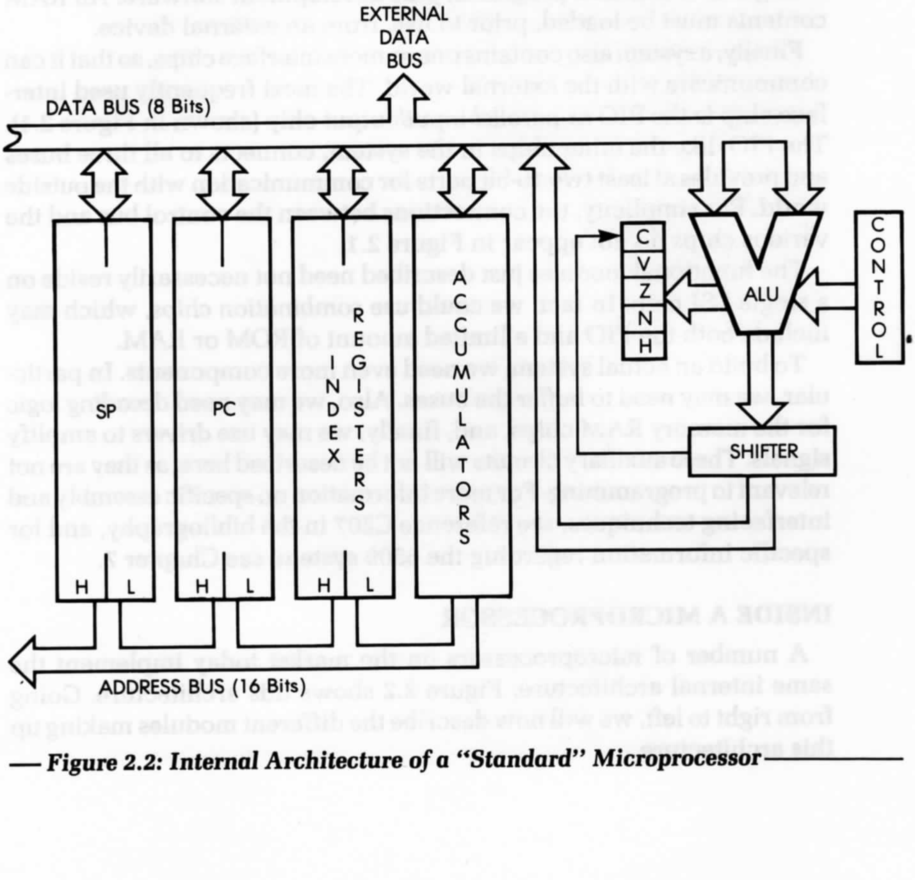
A number of microprocessors on the market today implement the same internal architecture. Figure 2.2 shows this architecture. Going from right to left, we will now describe the different modules making up this architecture.

The *control box* on the right of the illustration represents the control unit that synchronizes the entire system. We will describe the role of the control unit later in this chapter.

The *ALU* performs arithmetic and logical operations. Special registers, called *accumulators,* are usually connected to the output of the ALU. The accumulators contain the results of arithmetic operations. Each accumulator has eight bits.

The ALU also provides shift and *rotate* facilities. As illustrated in Figure 2.3, a shift moves the contents of a byte by one or more positions to the left or right. In this illustration, each bit has been moved to the left by one position. The shifter may be on the ALU output, as illustrated in Figure 2.2, or on the accumulator input. We describe shift and rotate operations in more detail in Chapter 3.

The *status* or *condition code register* appears to the left of the ALU. Its role is to store exceptional conditions within the microprocessor.



— *Figure 2.2: Internal Architecture of a "Standard" Microprocessor* —

The contents of the condition code register can be tested by specialized instructions, or read onto the internal data bus. A *conditional* instruction causes the execution of a different part of the program, depending on the value of one of the bits in the condition code register (as shown later).

Setting Condition Codes

Most of the instructions executed by the microprocessor modify some or all of the status bits. Refer to the chart provided by the manufacturer to learn which bits are modified by what instructions. This information is essential for understanding the way a program is executed. Appendix D lists this information for the 6809.

The Address Registers

Address registers are 16-bit registers used for the storage of addresses. They are also often called *data counters* or *pointers* and are double registers, i.e., two 8-bit registers. They are connected to the address bus. The address registers provide the signals for the address bus. At least two address registers are present within most microprocessors. Three address registers and an address bus appear in Figure 2.4.

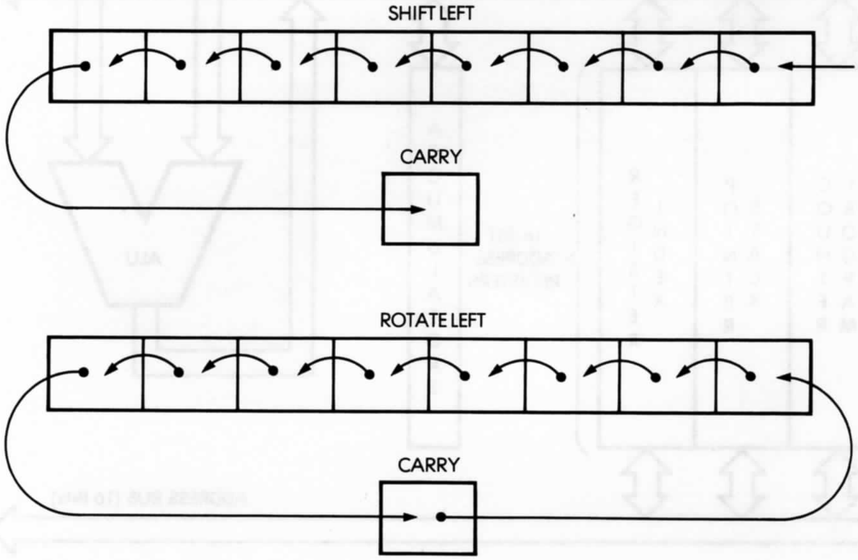


Figure 2.3: Shift and Rotate

The only way to load the contents of these 16-bit registers is via the data bus (also shown in the illustration). Two transfers are necessary along the data bus in order to transfer 16 bits. To differentiate between the lower and higher half of each register, each half is usually labeled as L (low) or H (high), denoting bits 0 through 7 or 8 through 15, respectively. Let's examine the three registers shown in the illustration.

The Program Counter (PC)

The *program counter* (or *PC*) must be present in all processors, as it is indispensable and fundamental to program execution. It contains the address of the next instruction to be executed.

Execution of a program is normally sequential. To access the next instruction, it is necessary to bring it from the memory into the microprocessor. The contents of the PC are deposited on the address bus, and transmitted towards the memory. The memory then reads the contents specified by this address and sends the corresponding word or instruction back to the MPU. In a few exceptional microprocessors, such as the

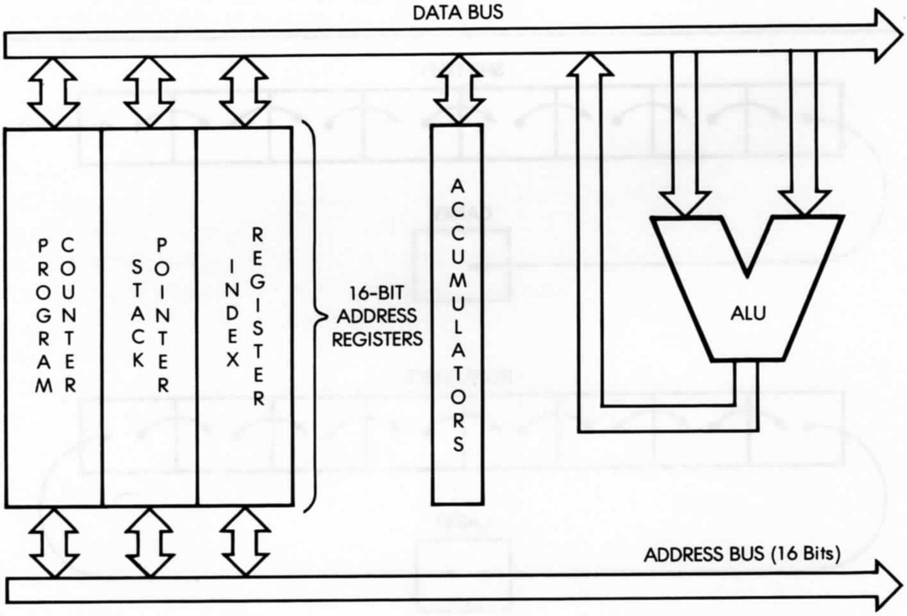


Figure 2.4: The 16-Bit Address Registers Create the Address Bus

2-chip F8, there is no PC on the microprocessor. This does not mean, however, that there is not a program counter—for reasons of efficiency, the PC is implemented directly on the memory chip.

The Stack Pointer (SP)

The *stack pointer* (the *SP*) is used to implement the stack. The stack is described in detail in the next section.

In most powerful, general-purpose microprocessors, the stack is implemented in "software," i.e., within the memory. To keep track of the top of the stack within the memory, a 16-bit register is dedicated to the stack pointer. The SP contains the address of the top of the stack within the memory. The stack is indispensable for interrupts and subroutines.

The Index Register (IX)

Indexing is a memory-addressing facility for accessing blocks of data in the memory with a single instruction. It is not always provided in microprocessors. An *index register* typically contains a displacement, which will automatically be added to a base (or, it might contain a base, which will be added to a displacement). In short, indexing is used to access any word within a block of data.

The Stack

A *stack*, formally called an LIFO (last-in, first-out) structure, is a set of registers, or memory locations, allocated to the stack data structure. The essential characteristic of the stack is that it is a *chronological* structure. The first element introduced in the stack is always at the bottom of the stack; the element most recently deposited is on the top. An analogy can be drawn with a stack of plates on a restaurant counter, if we assume there is a hole in the counter with a spring at the bottom, and plates are piled up in the hole. With this organization, it is guaranteed that the plate that has been put first in the stack is always at the bottom. The one most recently placed on the stack is the one on top. This example also illustrates another characteristic of the stack. In normal use, a stack is only accessible via two instructions: PUSH and POP (or PULL). These two instructions are illustrated in Figure 2.5. The PUSH operation deposits one element on top of the stack (possibly more in the case of the 6809); the PULL operation removes elements from the stack. In the case of a microprocessor, it is the *registers* that are deposited on top of the stack. The POP transfers the top element of the stack into the register specified in the instruction. Other specialized instructions may transfer the top of the stack between other specialized registers, such as the status register. The 6809 is more versatile than most in this respect.

A stack is required for implementing three programming facilities within the computer system: subroutines, interrupts, and temporary data storage. At this point, we will simply assume that the stack is a required facility in every computer system. The stack may be implemented in two ways:

1. as a hardware stack, where a fixed number of registers may be provided within the microprocessor itself. A hardware stack has the advantage of high speed; however, it has the disadvantage of a limited number of registers.
2. as a software stack. In order not to restrict the stack to a small number of registers, most general-purpose microprocessors, including the 6809, choose the software stack. With the software approach, a dedicated register within the microprocessor, here register SP, stores the stack pointer, i.e., the address of the top element of the stack (or, in some cases, the address of the top element of the stack, plus one). The stack is then implemented as an area of memory. The stack pointer, therefore, requires 16 bits to point anywhere in the memory.

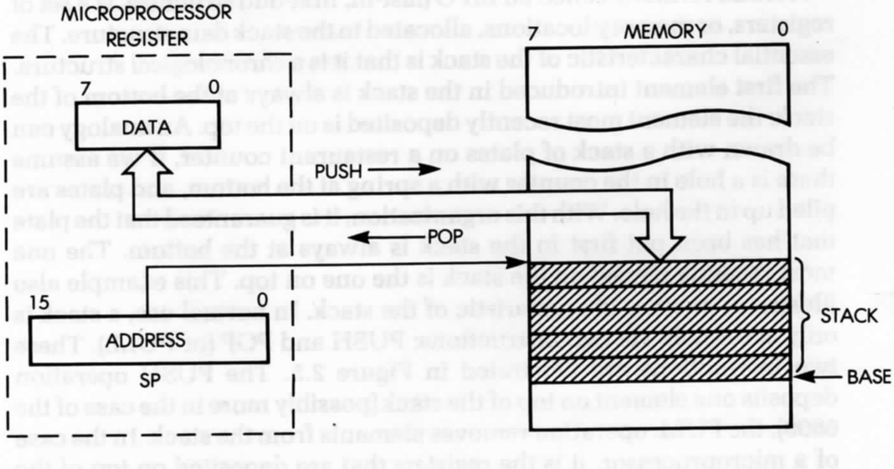


Figure 2.5: The Two-Stack Manipulation Instructions

The Instruction Execution Cycle

Let's now examine Figure 2.6, where we fetch an instruction from the memory in order to illustrate the role of the program counter. The microprocessor unit appears on the left of the illustration, and the memory appears on the right. The memory stores instructions and data. The memory chip may be a ROM or a RAM, or any other chip which happens to contain memory.

We assume that the program counter has valid contents. It now holds a 16-bit address, which is the address of the next instruction to fetch in the memory.

Every processor proceeds in three cycles, including:

1. fetching the next instruction
2. decoding the instruction
3. executing the instruction.

We will now follow this sequence.

Fetching

In the first cycle, the contents of the program counter are deposited on the address bus and gated to the memory (on the address bus). Simultaneously, a read signal may be issued on the control bus of the system, if

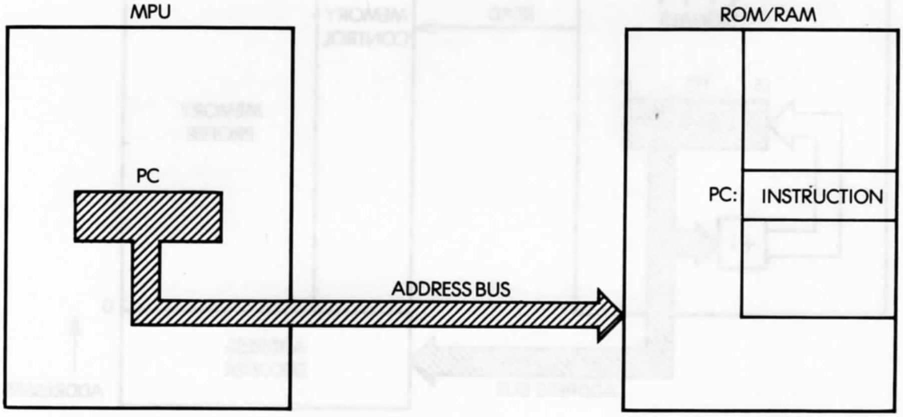


Figure 2.6: Fetching an Instruction from the Memory

required. The memory receives the address. The address is used to specify one location within the memory. Upon receiving the read signal, the memory decodes, through internal decoders, the address it has received and selects the location specified by the address. A few hundred nanoseconds later, the memory deposits the 8-bit data corresponding to the specified address on its data bus. This 8-bit word is the instruction we want to fetch. In the illustration in Figure 2.7, this instruction is deposited on the data bus.

Let us briefly summarize the sequence. The contents of the program counter are output on the address bus. A read signal is generated. The *memory* reads, and approximately 300 nanoseconds later, the instruction

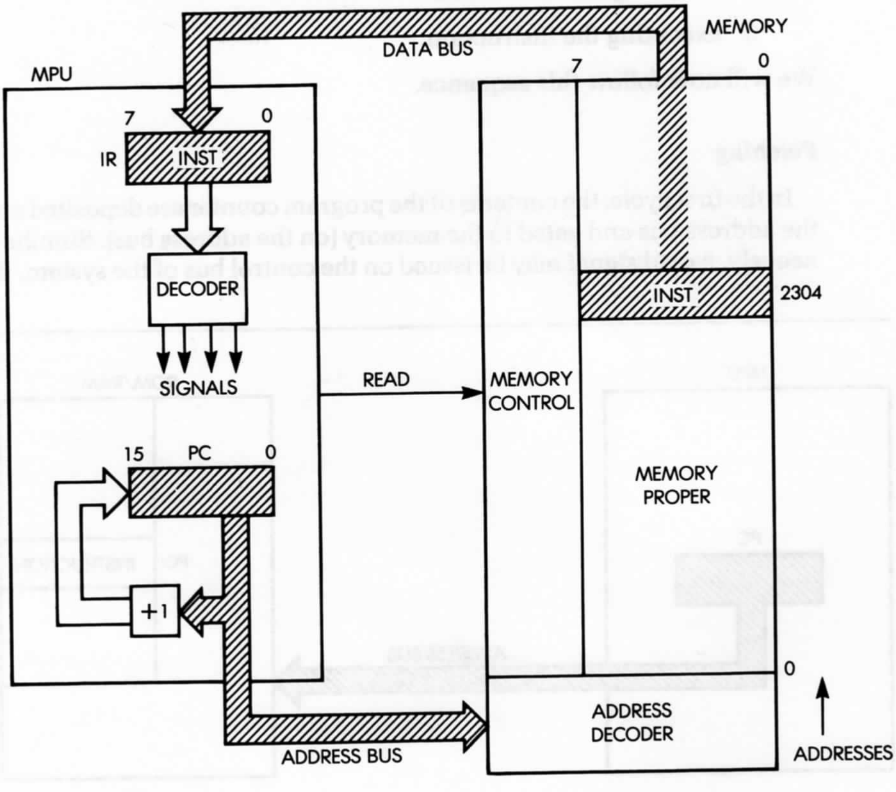


Figure 2.7: Automatic Sequencing

at the specified address is deposited on the data bus (assuming a single byte instruction). The microprocessor then reads the data bus and deposits its contents into a specialized internal register, the *IR* or *instruction register*. The IR is eight bits wide and is used to contain the instruction just fetched from the memory.

The fetch cycle is now completed. The eight bits of the instruction are now in the special internal register of the MPU, called the instruction register (the IR). The IR appears on the left of Figure 2.7. It is not accessible to the programmer.

Decoding and Executing

Once the instruction is in the IR, the control unit of the microprocessor decodes the contents and generates the correct sequence of internal and external signals for the execution of the specified instruction. There is, therefore, a short decoding delay, followed by an execution phase, the length of which depends on the nature of the instruction specified. Some instructions execute entirely within the MPU. Others fetch or deposit data from or into the memory. This is why the instructions of the MPU require various lengths of time to execute. This duration is expressed as a number of (clock) cycles. Appendix D lists the number of cycles required by each instruction. Since various clock rates may be used, speed of execution is normally expressed in number of cycles, rather than in number of nanoseconds.

Fetching The Next Instruction

We have described how an instruction can be fetched from the memory, using the program counter. During the execution of a program, instructions are fetched, in sequence, from the memory. An automatic mechanism must, therefore, be provided to fetch instructions in sequence. This task is performed by a simple incrementer attached to the program counter, as illustrated in Figure 2.7. Every time the contents of the program counter are placed on the address bus, the contents are incremented and written back into the program counter. As an example, if the program counter contains the value 0, the value 0 is output on the address bus. The contents of the program counter are then incremented, and the value 1 is written back into the program counter. In this way, the next time the program counter is used, it is the instruction at address 1 that is fetched. We have just implemented an *automatic mechanism for sequencing instructions*.

It must be stressed that the above descriptions are simplified. In reality, some instructions may be two or even three bytes long, so that successive bytes will be fetched in this manner from the memory. However, the fetch sequence is identical. The program counter is used to fetch successive bytes of an instruction, as well as successive instructions. The program counter, together with its incrementer, provides an automatic mechanism for pointing to successive memory locations.

INTERNAL ORGANIZATION OF THE 6809

Now that we understand the internal organization of a microprocessor, we will examine the 6809 in particular, and describe its capabilities. Figure 2.8 presents a logical description of the internal workings of the 6809. There may be additional interconnections that are not shown. Let's examine the diagram from right to left.

On the right side of the illustration, we see the *arithmetic-logical unit* (the ALU), recognizable by its characteristic "V" shape. The operation of the ALU will become clear in the next section, when we describe the execution of actual instructions.

The *condition code register*, called the CC in the 6809, appears to the right of the ALU. The contents of the condition code register are essentially conditioned by the ALU; however, some of its bits may also be conditioned by other modules or events (see Chapter 4).

The two registers to the left of the ALU are the accumulators, A and B. The accumulators are 8-bit registers, but for some instructions they can be used together to form the 16-bit D accumulator. Thus, the D accumulator is formed by using the B accumulator as the low byte, bits 0-7, and the A accumulator as the high byte, bits 8-15.

The register shown in the center of the illustration is the *direct page register*, labeled DP. The DP register is an 8-bit register used to address pages of memory. A *page* is simply a block of 256 words. Thus, memory locations 0 to 255 are page 0 of the memory. Since the 6809 has a 16-bit address bus, there are 256 pages. The DP register specifies the page number or high eight bits of an address. The other eight bits are obtained from the instruction being executed. The DP register allows faster and more compact programs to be produced when using blocks of memory smaller than 256 bytes.

The large group of registers to the left of the DP register, are the address registers. As in any microprocessor, we find in the group the program counter (PC) and the stack pointer (S). Recall that the program counter contains the address of the next instruction to be executed. The stack pointer points to the top of the stack in the memory. In the case of the 6809, the stack pointer points to the *last actual entry* in the stack. (In some microprocessors, the stack pointer points just above the last entry.) Also, the stack grows "downwards," i.e., towards the lower addresses. This means that the stack pointer must be *decremented* any time a new word is *pushed* on the stack. Conversely, whenever a word is *removed* (pulled) from the stack, the stack pointer must be incremented by one. In the case of the 6809, PUSH and PULL may involve up to twelve words at the same time, so that the contents of the stack pointer are decremented or incremented by numbers between 1 and 12, inclusive.

The U register is the *user* stack pointer. In the case of PUSH and PULL operations, this register behaves exactly like the S stack pointer. It allows two stacks to be used by the programmer. Recall that with the S

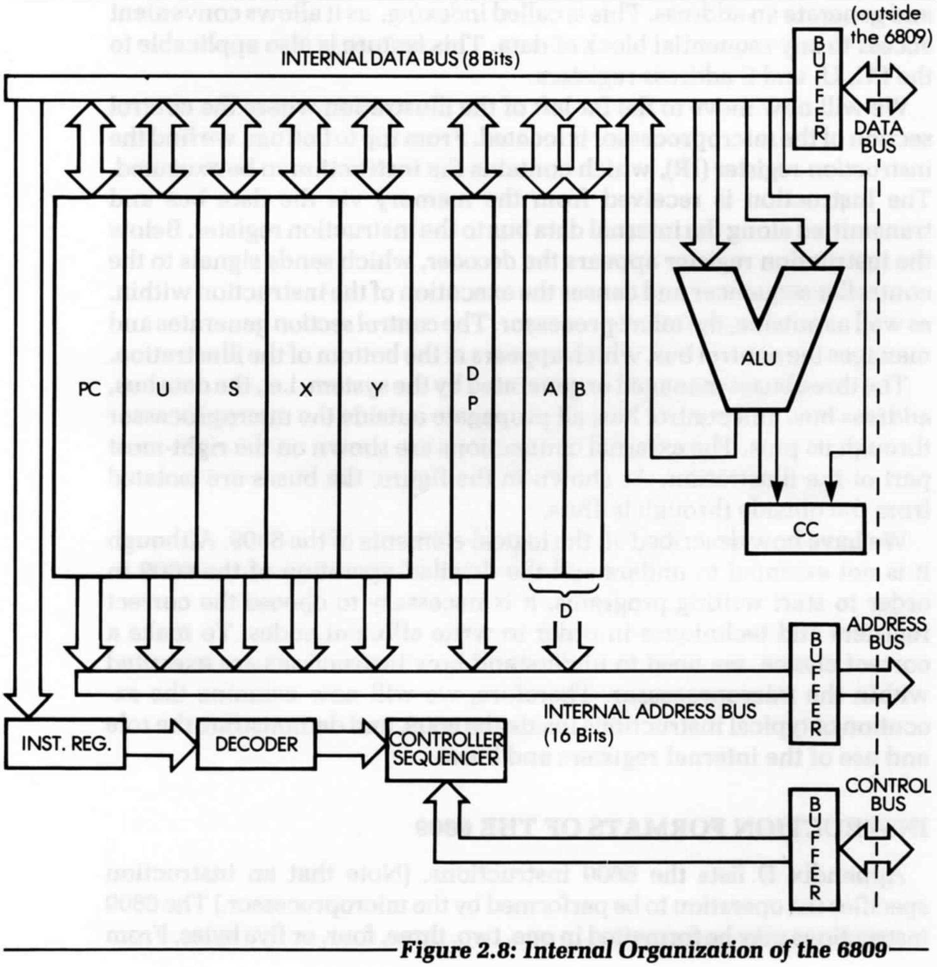


Figure 2.8: Internal Organization of the 6809

stack pointer (also called the *hardware* or *system* stack pointer), certain instructions and outside events cause automatic pushes and pulls. For example, the S register is used in subroutine calls. The U stack pointer is not used by the hardware of the computer, therefore, the programmer has complete control over it.

Looking at the remaining two registers of this group of five registers, we find another type of register: the *index register*. The two index-registers are labeled X and Y. A byte brought along the internal data bus may be added to the contents of X or Y. When using an indexed instruction, this byte is called a *displacement*. Special instructions are provided that will automatically add this displacement to the contents of X or Y and generate an address. This is called *indexing*, as it allows convenient access to any sequential block of data. This feature is also applicable to the PC, U, and S address registers.

We will now move to the far left of the illustration where the control section of the microprocessor is located. From top to bottom, we find the *instruction register* (IR), which contains the instruction to be executed. The instruction is received from the memory via the data bus and transmitted along the internal data bus to the instruction register. Below the instruction register appears the *decoder*, which sends signals to the controller sequencer and causes the execution of the instruction within, as well as outside, the microprocessor. The *control section* generates and manages the control bus, which appears at the bottom of the illustration.

The three buses managed or generated by the system, i.e., the data bus, address bus, and control bus, all propagate outside the microprocessor through its pins. The external connections are shown on the right-most part of the illustration. As shown in the figure, the buses are isolated from the outside through buffers.

We have now described all the logical elements of the 6809. Although it is not essential to understand the detailed operation of the 6809 in order to start writing programs, it is necessary to choose the correct registers and techniques in order to write efficient codes. To make a correct choice, we need to understand how instructions are executed within the microprocessor. Therefore, we will now examine the execution of typical instructions inside the 6809, and demonstrate the role and use of the internal registers and buses.

INSTRUCTION FORMATS OF THE 6809

Appendix D lists the 6809 instructions. (Note that an instruction specifies the operation to be performed by the microprocessor.) The 6809 instructions may be formatted in one, two, three, four, or five bytes. From a more simplified standpoint, every instruction may be represented as an opcode, followed by an optional literal or address field, comprising one or two words. The opcode field specifies the operation to be carried out. In strict computer terminology, the opcode represents only those bits that specify the operation to be performed, exclusive of the register pointers that might be necessary. In the microprocessor world, it is convenient to call the opcode the operation code itself, as well as any register pointers that it might incorporate. This "generalized opcode" must reside in an 8-bit word, for reasons of efficiency. This 8-bit opcode is a limiting factor on the number of instructions available in a microprocessor.

Most microprocessors use instructions that are one, two or three bytes long. (See Figure 2.9.) However, the 6809 is equipped with additional indexed instructions, which require one more byte. In the case of the 6809, opcodes are, in general, one byte long, except for special instructions, which require a 2-byte opcode.

Many instructions require that one byte of data, or a part of an address, follow the opcode. In such a case, the instruction will be a 2-byte instruction—the second byte being data or part of an address (with the exception of indexing, which adds an extra byte). In other cases, the instruction might require the specification of an address. An address requires 16 bits and, therefore, two bytes. Thus, the instruction will be a 3- or 4-byte instruction.

For each byte of the instruction, the control unit must perform a memory fetch, which requires one clock cycle. Thus, the shorter the instruction, the faster the execution.

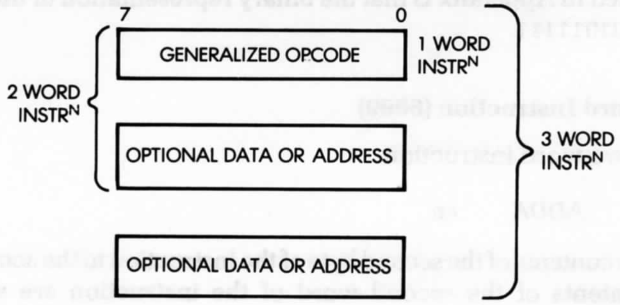


Figure 2.9: Typical Instruction Formats

One Word Instruction (6809)

One word instructions require the smallest amount of memory and are, therefore, favored by the programmer. A typical one word instruction for the 6809 is an increment, for example:

```
        INCA
```

which means: "add 1 to the contents of the A accumulator." This is a typical operation. Every microprocessor is equipped with an instruction like INCA, which allows the programmer to quickly add a 1 to a register, which may then be used as a counter or pointer into memory. Instructions referencing different registers of memory will have different opcodes.

Every instruction must be represented internally in a binary format. The above representation, INCA, is *mnemonic*, or symbolic; it is the *assembly language* representation of an instruction. It is a convenient symbolic representation of the actual binary encoding for that instruction. The binary code that represents this instruction inside memory is: 01001100 (bits 0 to 7).

The placement of the bits in the binary representation of an instruction is not meant for the convenience of the programmer, but for the microprocessor, which must decode and execute the instruction. The assembly language representation, however, is meant for the convenience of the programmer.

Another example of a one word instruction is:

```
        CLRB
```

This instruction clears the contents of the specified accumulator (in this case, B). This operation may be represented symbolically by B=0. It can be verified in Appendix D that the binary representation of this instruction is: 01011111.

Two Word Instruction (6809)

The two word instruction

```
        ADDA      #n
```

adds the contents of the second byte of the instruction to the accumulator. The contents of the second word of the instruction are said to be "literal." They are data and are treated as eight bits without any particular significance. They could be a character or numerical data—a fact that is irrelevant to the operation.

The code for this instruction is:

10001011 followed by the 8-bit byte "n"

The symbol "#" is used to indicate an *immediate* operation. "Immediate" in most programming languages, means that the next word, or words, within the instruction contain a piece of data that should not be *interpreted*, i.e., the next one or two words are to be treated as *literals.*

The control unit is programmed to "know" how many words each instruction has. It will, therefore, always fetch and execute the right number of words for each instruction. However, the longer the instruction, the more complex it is for the control unit to decode.

Three Word Instruction (6809)

The instruction

```
LDB     nn
```

requires three words. It means: "load the B accumulator from the memory address specified in the next two bytes of the instruction." Since addresses are 16-bits long, they require two words. In binary, this instruction is represented by:

11110110	8 bits for the opcode
High Address	8 bits for the upper part of the address
Low Address	8 bits for the lower part of the address

EXECUTION OF INSTRUCTIONS IN THE 6809

We have seen that all instructions are executed in three phases: fetch, decode, and execute. The amount of time it takes to execute an instruction depends on the instruction and the type of memory access being done. In the 6809, time is measured in clock cycles. It always takes an integral number of clock cycles to execute an instruction.

Accessing memory requires one clock cycle. Since each instruction must first be fetched from memory, even the fastest instruction requires more than one clock cycle. The fetch phase of an instruction presents the address of the next instruction to the memory. This address is contained in the program counter. When the contents of memory are available, they can be transferred within the microprocessor to the instruction register. The PC is then incremented to point to the next word in the program.

When an instruction is deposited in the instruction register of the 6809, it is decoded. It takes at least one clock cycle, and possibly more, to decode and execute an instruction. Appendix D gives the execution time for each instruction. Appendix E describes the address bus cycle-by-cycle activity for each instruction, and shows the external activities of the 6809, while the instruction is being executed. These tables offer an in-depth understanding of instruction execution.

Execution Of A 1-Byte Instruction (6809)

Recall that the 1-byte instruction

```
        INCA
```

adds a 1 to the A accumulator. This instruction is fetched during the first clock cycle and is decoded and executed during the second cycle, while the next byte of the program is being fetched. The two cycle execution time of a 1-byte instruction illustrates that all instructions require at least two clock cycles.

Execution Of A 2-Byte Instruction (6809)

Recall that the instruction

```
        ADDA    #n
```

described in the previous section, adds to the A accumulator the contents of the byte that immediately follows the instruction. During the first clock cycle, the instruction is loaded into the IR; and the PC increments. During the second clock cycle the instruction is decoded, *while* the next byte, the data, is fetched. The data from this second fetch is added to the accumulator before the end of the second cycle. It should be observed that two activities occurred during the second cycle: the instruction in the IR was decoded, and the next byte was fetched. Since most instructions in the 6809 need this second byte, execution is speeded considerably.

Execution Of A 3-Byte Instruction (6809)

The instruction

```
        LDB     nn
```

is a 3-byte instruction. Recall that it loads the B accumulator with the contents of the memory location addressed by nn.

This instruction requires 5 cycles to execute. The first cycle fetches the opcode. The next decodes the instruction and fetches the high address byte. The third fetches the low address byte. The fourth forms the address of the data on the *internal* address bus (see Figure 2.8). The fifth uses this address to fetch the data from memory and store it in the accumulator.

The detailed descriptions we have just presented on the execution of typical instructions should help to clarify the role of the registers and internal buses. A second reading of the preceding section may be helpful in gaining a detailed understanding of the internal operation of the 6809.

THE 6809 CHIP

For completeness, we will now examine the signals of the 6809 microprocessor chip. You do not need to understand the functions of 6809 signals in order to program the 6809. If you do not have an interest in the details of hardware, you may want to skip this section.

The 6809 comes in two different forms: the MC6809 and the MC6809E. We will first describe the signals of the MC6809. Then we will describe those signals on the MC6809E that are different from those on the MC6809. The instructions for the two processors are identical, only a few hardware pinouts are different. Figure 2.10 displays the pinout of the MC6809.

The control signals have been divided into four groups. We will now describe them, going from the top of the figure to the bottom.

The first two clock pins, XTAL and EXTAL, are for the connection of an external crystal. The oscillator is contained within the MC6809. The clock cycle frequency is one-fourth the crystal frequency. The other two clock signals, E for enable and Q for quadrature, are used to indicate the times when the data and address bus signals are valid.

The three bus control signals, DMA/BREQ, BS and BA, are used to disconnect the MC6809 from its buses. They are mainly used for DMA, but could also be used by another processor in the system. The DMA/BREQ is the bus request signal issued to the MC6809. In response, the MC6809 places its address bus, data bus, and some output control signals (tristate) in the high-impedance state at the end of the current instruction being executed. The processor status indicators, bus available (BA) and bus status (BS), are used to acknowledge that the buses have been placed in the high-impedance state. There are four possible BA and BS combinations. They are:

BA	*BS*	
0	0	Normal (Running)
0	1	Interrupt or Reset Acknowledge
1	0	Sync Acknowledge
1	1	Halt/Bus Grant Acknowledge

The last state, when BA and BS are 1, is the state that acknowledges the DMA/BREQ. In this section, we discuss interrupt and reset; we describe the sync acknowledge in Chapter 6.

The MC6809 can give the bus to DMA devices for only 15 clock cycles at a time. The processor will then take control of the bus for at least one cycle, to do internal refreshing.

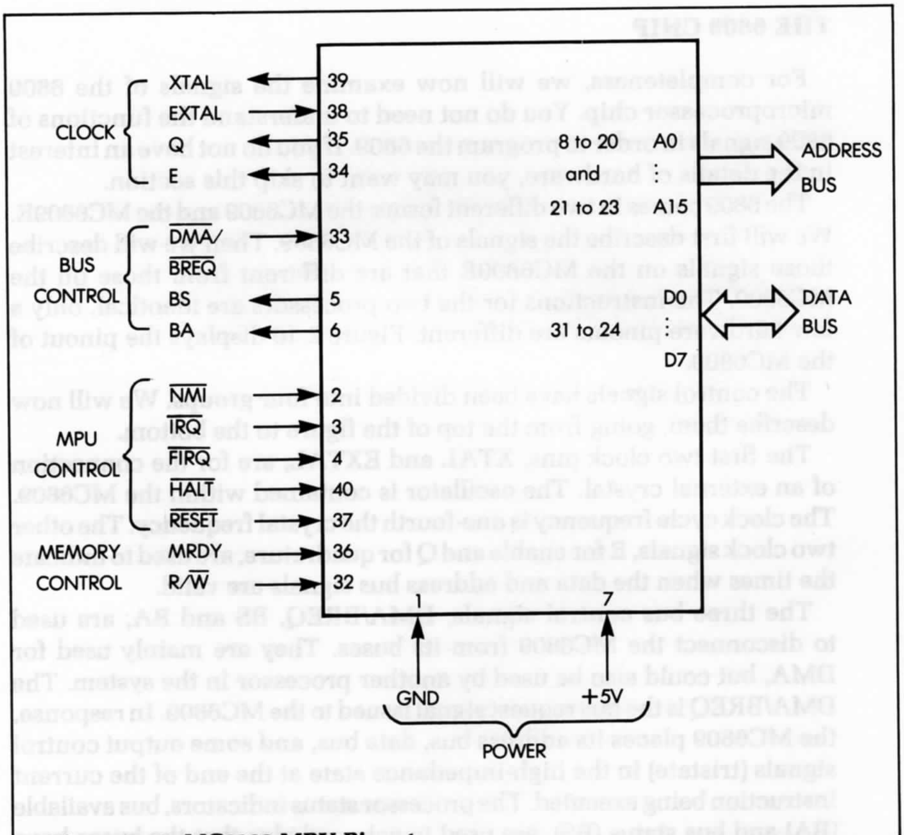


Figure 2.10: MC6809 MPU Pinout

Six MC6809 control signals are related to its internal status or sequencing. IRQ, FIRQ, and NMI are the three interrupt signals. IRQ is the usual interrupt request. A number of input/output devices may be connected to the IRQ interrupt line. Whenever an interrupt request is present on this line, and the internal interrupt enable bit is enabled, the 6809 will accept the interrupt (provided a DMA operation is not in progress). The BA signal will be set to 0, and the BS signal to 1, to indicate an interrupt acknowledge. We describe the rest of this sequence in Chapter 6.

FIRQ is the fast interrupt request signal. It is similar to IRQ but executes faster. NMI is the non-maskable interrupt. It is always accepted by the 6809, assuming no DMA is in progress.

MRDY is a signal used to synchronize the MC6809 with slow memory or input/output devices. When active, this signal indicates that the memory on the device is not yet ready for the data transfer. The MC6809 CPU will wait until the MRDY signal becomes inactive. It will then resume normal sequencing. The MRDY signal may be active for 10 clock cycles at most.

HALT is used to stop the processor. When HALT is active, the processor completes the present instruction and remains halted indefinitely, without loss of data. When the processor is halted, the BA and BS signals are 1, to indicate that the buses are in the high-impedance state and the processor is in the halt/bus grant state. When the HALT signal becomes inactive, processing will resume.

RESET is usually the signal that initializes the MPU. It moves the contents of addresses FFFF and FFFE into the PC. The DP register is set to 0 and both fast and normal interrupts are disabled. The BA signal is 0 and BS is 1, to acknowledge a reset. RESET is usually used after power is applied to the computer.

There is one signal for memory control: the read/write (R/W) control signal. This output indicates whether the next transfer by the processor on the data bus is a read or write .

MC6809E Control Signals

The major difference between the MC6809 and the MC6809E is that the E version requires an external clock generator circuit. This approach allows greater flexibility in the clock circuit capabilities and is useful for multi-processor systems. The differences between the MC6809 and the MC6809E pinouts are detailed below. Figure 2.11 displays the MC6809E pinouts.

Since there is no clock oscillator in the MC6809E, the XTAL and EXTAL pins are not needed. The Q and E clock pins are now inputs, rather than outputs. An external circuit generates Q and E. Otherwise, the definitions of the Q and E signals are the same.

The bus control signals are different. The DMA/BREQ is eliminated and replaced by TSC, the three-state control line. The TSC signal puts the data and address buses and the R/W signal into the high-impedance state in the next clock cycle. The E and Q clocks must then be stopped for the next cycle. The BS and BA signals are not changed. The BUSY control indicates that the processor is executing an instruction that requires more than one clock cycle to stabilize the data in memory. A TSC should not be done when BUSY is active. This is very important for multiprocessor systems.

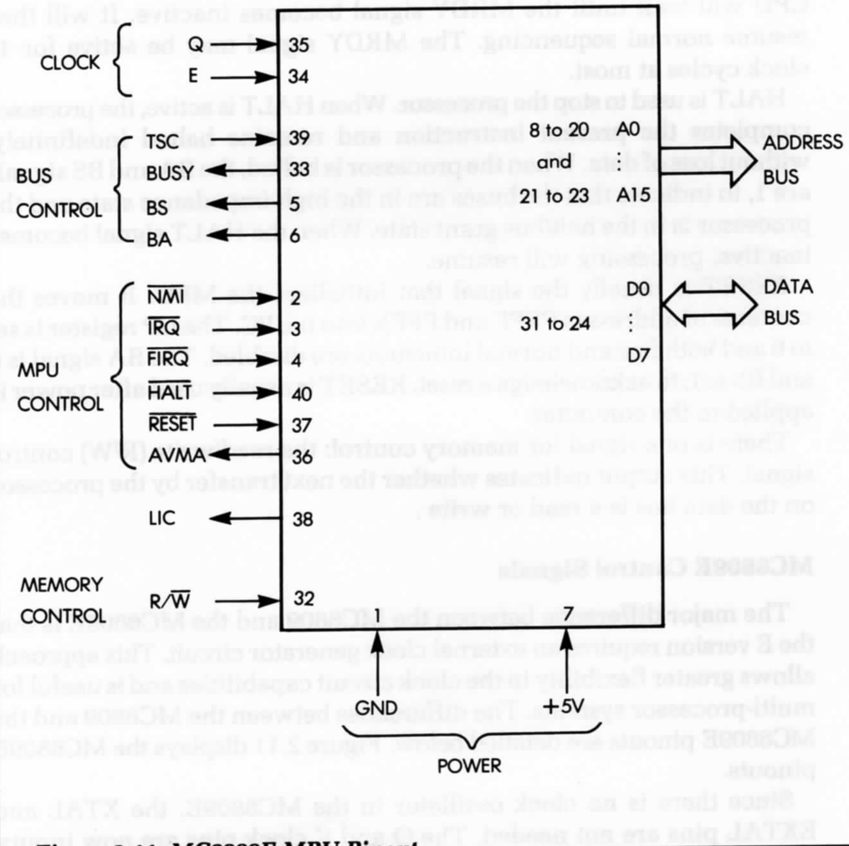


Figure 2.11: MC6809E MPU Pinout

There is only one change in the MPU control signals for the MC6809E: the MRDY input is replaced by the advanced valid memory access (AVMA) signal. This signal indicates that the processor is going to do a valid memory access during the next clock cycle. This indicates to the clock circuit that, if slow memory or I/O is being accessed, the clock times should be extended. The processor itself cannot control the clock in the MC6809E.

There is one new output signal in the MC6809E: the signal that indicates execution of the last instruction cycle (LIC). This signal becomes active during the last cycle of every instruction. When it goes low, it indicates that the first byte of an instruction will be fetched at the end of the present cycle.

SUMMARY

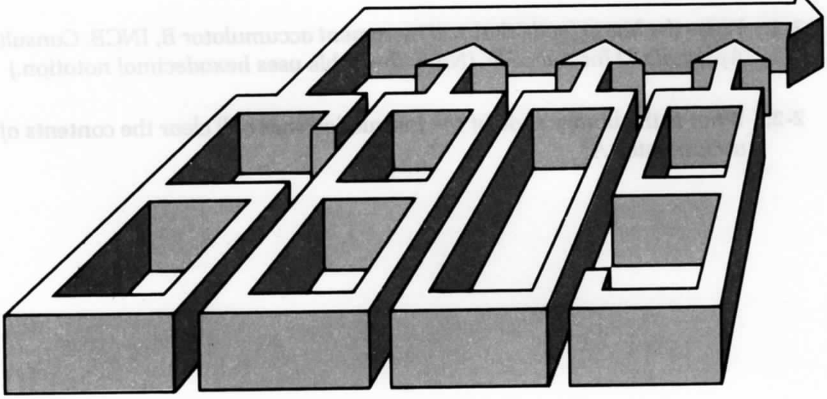
This chapter has presented a description of the internal organization of the 6809. The role of each register is important and should be fully understood before proceeding to the next chapter. Chapter 3 introduces the instructions available on the 6809 and many basic programming techniques for the 6809.

EXERCISES

2-1: *Write the binary code that will increment accumulator B, INCB. Consult Appendix D for the code. (Note: this table uses hexadecimal notation.)*

2-2: *What is the binary code of the instruction that will clear the contents of accumulator A?*

CHAPTER 3



BASIC PROGRAMMING TECHNIQUES

IN THIS CHAPTER, we examine the basic techniques necessary for writing a program for the 6809. In particular, we show how to move information between the memory and the MPU, and how to manipulate it within the MPU itself. We develop programs of increasing complexity, so that we can see how various instructions and registers interact.

We will begin by writing simple arithmetic programs. We will then go on to explain the use of the 6809's excellent 16-bit arithmetic capabilities. Finally, we will discuss the important multiply and divide operations.

ARITHMETIC PROGRAMS

The arithmetic programs in this chapter show how to do addition, subtraction, multiplication, and division. Each uses at least one register. Figure 3.1 shows a conceptual diagram of the 6809 registers. These programs perform integer arithmetic on positive binary numbers and on negative numbers represented as two's complement integers. Let's begin with an example of 8-bit addition.

8-Bit Addition

We begin by writing a program that performs 8-bit addition:

```
        (Instructions)           (Comments)
        LDA      ADR1          LOAD OP1 INTO A
        ADDA     ADR2          ADD OP2 TO OP1
        STA      ADR3          SAVE RESULT RES AT ADR3
```

In this program, we add two 8-bit operands, OP1 and OP2, stored at memory addresses ADR1 and ADR2, respectively. We call the sum RES, and store it at memory address ADR3 (as shown in Figure 3.2).

Each line of the program, expressed here in symbolic form, is called an *instruction*. Each instruction is translated by the *assembler* program into from one to five binary bytes. For this example, we will *not* concern ourselves with this translation; instead we will examine the symbolic representation.

The first line of the program specifies: "load the contents of ADR1 into accumulator A." (Or accumulator B *could* have been used.). Figure 3.2 shows that the contents of ADR1 are the first operand, OP1. Thus, the first instruction transfers OP1 from the memory into the accumulator (see Figure 3.3).

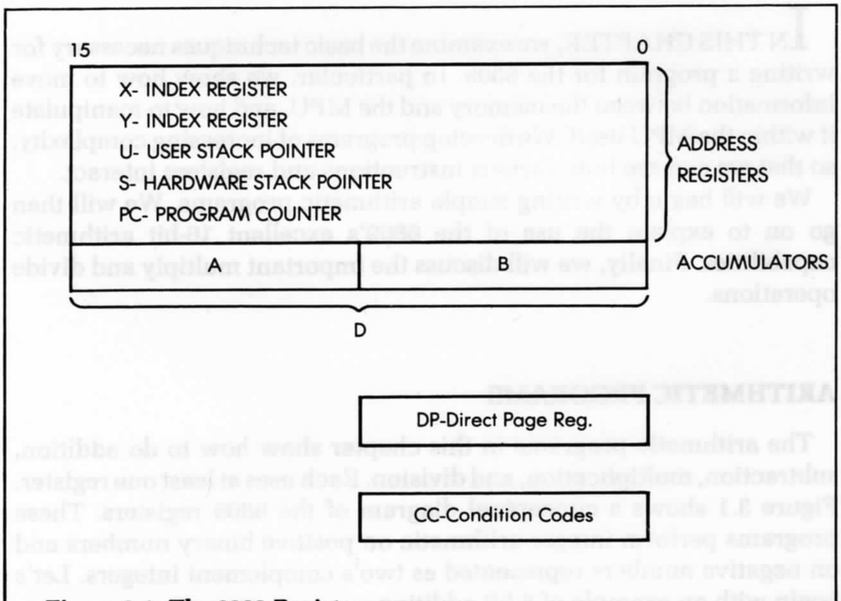


Figure 3.1: The 6809 Registers

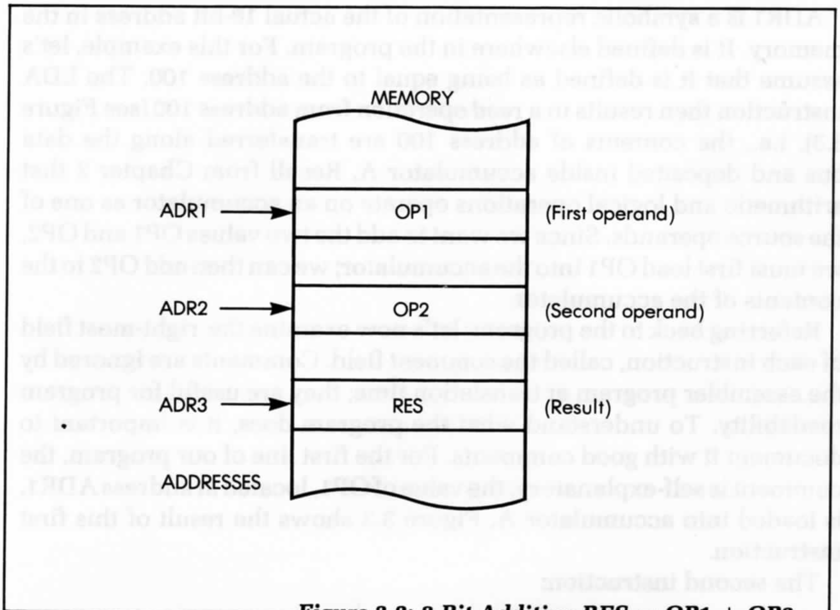


Figure 3.2: 8-Bit Addition RES = OP1 + OP2

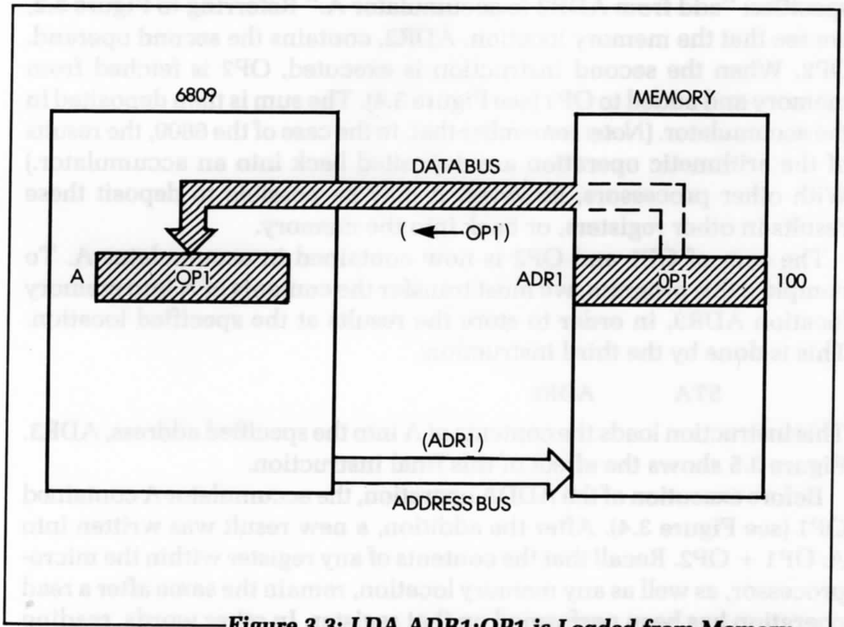


Figure 3.3: LDA ADR1:OP1 is Loaded from Memory

ADR1 is a symbolic representation of the actual 16-bit address in the memory. It is defined elsewhere in the program. For this example, let's assume that it is defined as being equal to the address 100. The LDA instruction then results in a *read operation* from address 100 (see Figure 3.3), i.e., the contents of address 100 are transferred along the data bus and deposited inside accumulator A. Recall from Chapter 2 that arithmetic and logical operations operate on an accumulator as one of the source operands. Since we want to add the two values OP1 and OP2, we must first load OP1 into the accumulator; we can then add OP2 to the contents of the accumulator.

Referring back to the program, let's now examine the right-most field of each instruction, called the *comment* field. Comments are ignored by the assembler program at translation time; they are useful for program readability. To understand what the program does, it is important to document it with good comments. For the first line of our program, the comment is self-explanatory: the value of OP1, located at address ADR1, is loaded into accumulator A. Figure 3.3 shows the result of this first instruction.

The second instruction:

```
        ADDA    ADR2
```

specifies: "add from ADR2 to accumulator A." Referring to Figure 3.2, we see that the memory location, ADR2, contains the second operand, OP2. When the second instruction is executed, OP2 is fetched from memory and added to OP1 (see Figure 3.4). The sum is then deposited in the accumulator. (*Note:* remember that, in the case of the 6809, the results of the arithmetic operation are deposited back into an accumulator.) With other processors, however, it may be possible to deposit these results in other registers, or back into the memory.

The sum of OP1 and OP2 is now contained in accumulator A. To complete this program, we must transfer the contents of A into memory location ADR3, in order to store the results at the specified location. This is done by the third instruction:

```
        STA     ADR3
```

This instruction loads the contents of A into the specified address, ADR3. Figure 3.5 shows the effect of this final instruction.

Before execution of the ADDA operation, the accumulator A contained OP1 (see Figure 3.4). After the addition, a new result was written into A: OP1 + OP2. Recall that the contents of any register within the microprocessor, as well as any memory location, remain the same after a read operation has been performed on that register. In other words, reading the contents of a register or memory location does not change its contents. Only a *write* operation in the register location changes the contents. In this program, the contents of ADR1 and ADR2 remain unchanged throughout the program. However, after the ADD instruction,

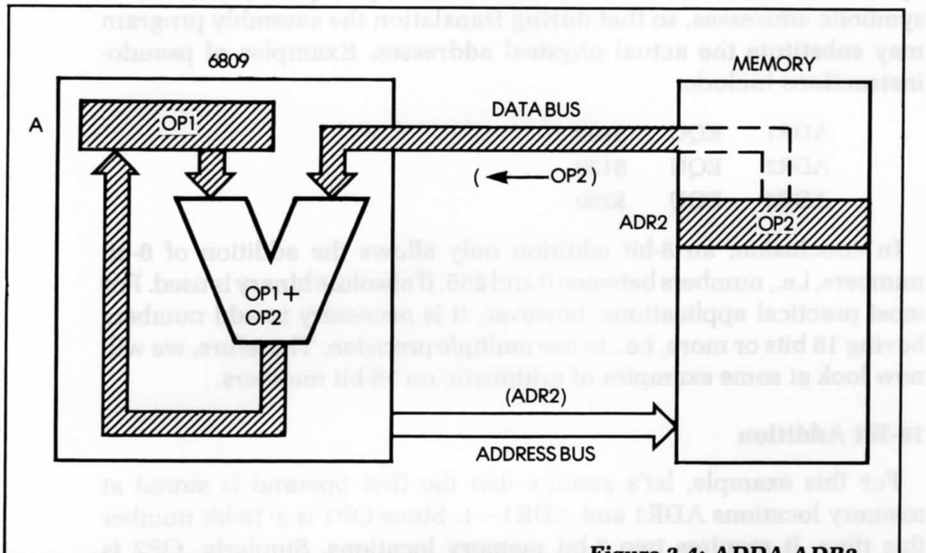


Figure 3.4: ADDA ADR2

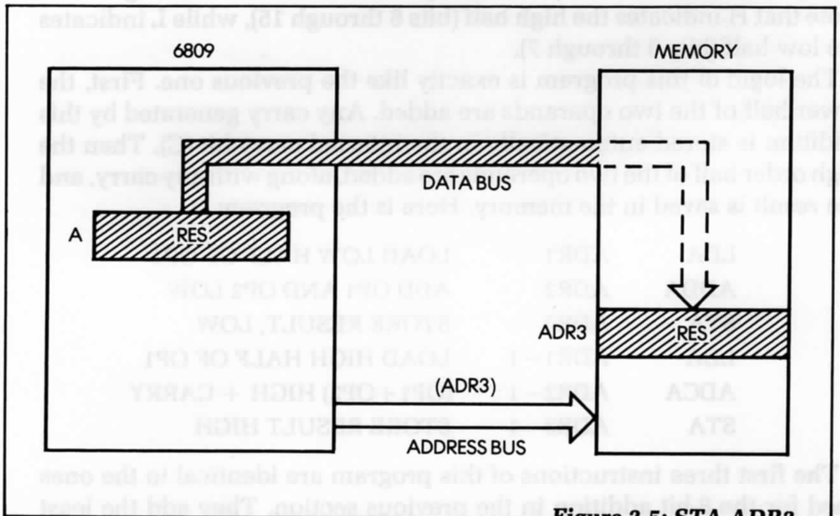


Figure 3.5: STA ADR3

the contents of A are modified, because the output of the ALU is written into the accumulator. The previous contents of A are then lost.

Actual numerical addresses may be used instead of ADR1, ADR2, and ADR3. To keep symbolic addresses, it is necessary to use so-called "pseudo-instructions." Pseudo-instructions specify the value of the symbolic addresses, so that during translation the assembly program may substitute the actual physical addresses. Examples of pseudo-instructions include:

```
ADR1    EQU    $100
ADR2    EQU    $120
ADR3    EQU    $200
```

In conclusion, an 8-bit addition only allows the addition of 8-bit numbers, i.e., numbers between 0 and 255, if absolute binary is used. For most practical applications, however, it is necessary to add numbers having 16 bits or more, i.e., to use *multiple precision*. Therefore, we will now look at some examples of arithmetic on 16-bit numbers.

16-Bit Addition

For this example, let's assume that the first operand is stored at memory locations ADR1 and ADR1−1. Since OP1 is a 16-bit number this time, it requires two 8-bit memory locations. Similarly, OP2 is stored at ADR2 and ADR2−1. The result is to be deposited at memory addresses ADR3 and ADR3−1. This process is illustrated in Figure 3.6. Note that H indicates the high half (bits 8 through 15), while L indicates the low half (bits 0 through 7).

The logic of this program is exactly like the previous one. First, the lower half of the two operands are added. Any carry generated by this addition is stored automatically in the internal carry bit (C). Then the high order half of the two operands are added, along with any carry, and the result is saved in the memory. Here is the program:

```
LDA     ADR1      LOAD LOW HALF OF OP1
ADDA    ADR2      ADD OP1 AND OP2 LOW
STA     ADR3      STORE RESULT, LOW
LDA     ADR1−1    LOAD HIGH HALF OF OP1
ADCA    ADR2−1    (OP1+OP2) HIGH + CARRY
STA     ADR3−1    STORE RESULT HIGH
```

The first three instructions of this program are identical to the ones used for the 8-bit addition in the previous section. They add the least significant halves (bits 0–7) of OP1 and OP2. The sum, called RES, is stored at memory location ADR3 (see Figure 3.6).

Automatically, whenever an addition is performed, any resulting carry (whether 0 or 1) is saved in the carry bit, C, of the condition codes register (register CC). If the two 8-bit numbers generate a carry, then the C bit will be equal to 1. (It will be set.) If the two 8-bit numbers do not generate a carry, then the value of the carry bit will be 0.

The next three instructions of the program are similar to those used in the previous 8-bit addition program. This time, however, they add the most significant half (i.e., the high half—bits 8–15) of OP1 and OP2, plus any carry, and store the result at the address ADR3 − 1.

After execution of this six-instruction program, the 16-bit result is stored at memory locations ADR3 and ADR3 − 1, as specified. Note, however, that there is one difference between the second half of this

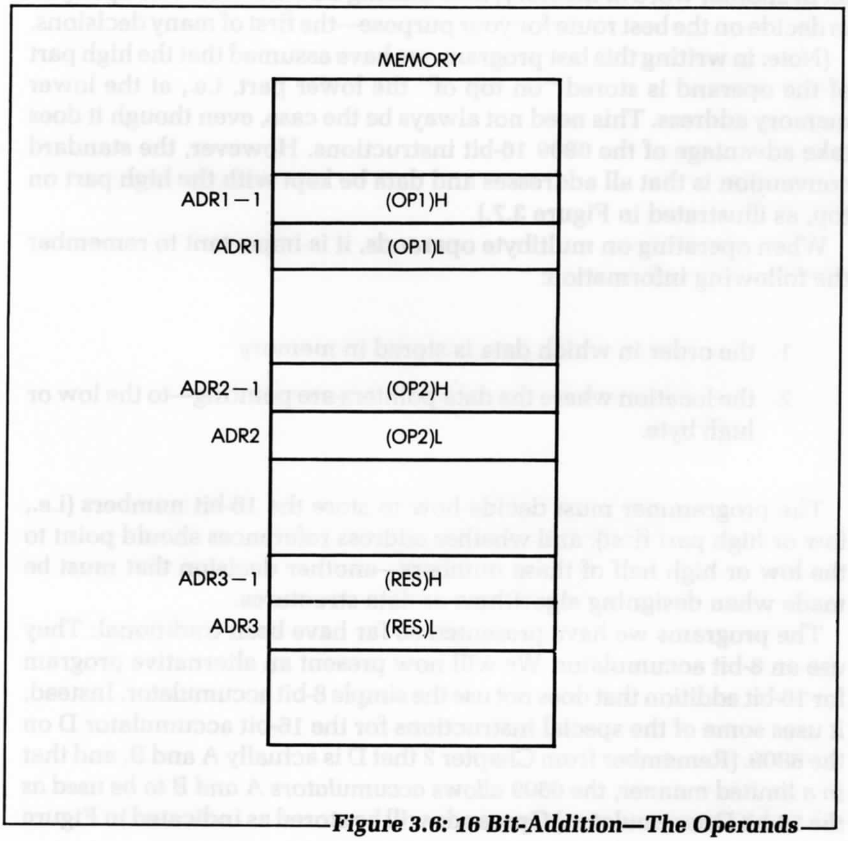


Figure 3.6: 16 Bit-Addition—The Operands

program and the first. The ADC instruction is not the same instruction as the one used in the first half. In the first half of the program, we used the ADD instruction (the 2nd instruction). This instruction adds the two operands, regardless of the carry. In the second half, we used the ADC instruction, which adds the two operands, plus any carry that may have been generated. Here, we must use the ADC instruction to obtain the correct result, as the addition performed on the low operands may result in a carry.

At this point you might ask: "but what if the addition of the high half of the operands also results in a carry?" There are two ways to handle this situation. First, you can assume that this will not happen, unless an error has been made, because the program is designed to work for results of only up to 16 bits—not 17; and that the program will halt when the carry is set. Or, you can include additional instructions that will handle the extra bit in another word of memory, thus making a 24-bit word. It is up to you to decide on the best route for your purpose—the first of many decisions.

(*Note:* in writing this last program, we have assumed that the high part of the operand is stored "on top of" the lower part, i.e., at the lower memory address. This need not always be the case, even though it does take advantage of the 6809 16-bit instructions. However, the standard convention is that all addresses and data be kept with the high part on top, as illustrated in Figure 3.7.)

When operating on multibyte operands, it is important to remember the following information:

1. the order in which data is stored in memory
2. the location where the data pointers are pointing—to the low or high byte.

The programmer must decide how to store the 16-bit numbers (i.e., low or high part first), and whether address references should point to the low or high half of these numbers—another decision that must be made when designing algorithms or data structures.

The programs we have presented so far have been traditional: They use an 8-bit accumulator. We will now present an alternative program for 16-bit addition that does not use the simple 8-bit accumulator. Instead, it uses some of the special instructions for the 16-bit accumulator D on the 6809. (Remember from Chapter 2 that D is actually A and B, and that *in a limited manner, the 6809 allows accumulators A and B to be used as the 16-bit D accumulator.*) Operands will be stored as indicated in Figure

3.7. The program is:

```
LDD     ADR1    LOAD D ACCUMULATOR WITH OP2
ADDD    ADR2    ADD OP2 TO OP1 16 BITS
STD     ADR3    STORE RES INTO ADR3
```

Notice how much shorter this program is, when compared to the previous version.

16-bit numbers can be readily extended to 24, 32, or more bits (always multiples of 8 bits). Let's now try an interesting exercise. Let's use the 16-bit instructions we just introduced to write an addition program for 32-bit operands, assuming the operands are stored as shown in Figure

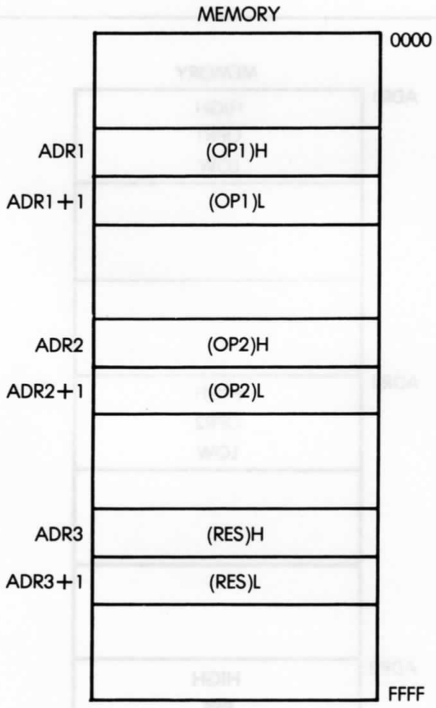


Figure 3.7: Storing 16-Bit Operands in the 6809

3.8. Here is the program:

```
LDD     ADR1+2     LOAD LOW HALF OP1
ADDD    ADR2+2     ADD LOW HALF OP2
STD     ADR3+2     STORE LOW HALF RES
LDD     ADR1       LOAD HIGH HALF OP1
ADCB    #0         ADD 0 AND CARRY TO B
ADCA    #0         ADD 0 AND CARRY TO A
ADDD    ADR2       ADD HIGH HALF OP2
STD     ADR3       STORE HIGH HALF RES
```

(*Note:* There is no instruction that adds a carry to the D accumulator. The carry is handled by adding a zero and any carry to B, and then to A, after the high 16-bits have been loaded into D.)

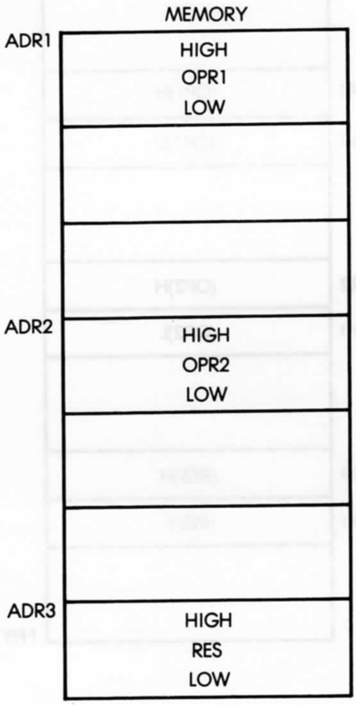


Figure 3.8: A 32-Bit Addition

Now that we have learned to perform a binary addition, let's learn about subtraction.

Subtracting 16-Bit Numbers

Performing an 8-bit or 16-bit subtraction is actually quite simple, so let's try a 16-bit subtraction. As usual, our two numbers, OP1 and OP2, are stored at addresses ADR1 and ADR2. The memory is assumed to be that of Figure 3.7. To perform the subtraction, we use the subtract operation (SUB), instead of the add operation (ADD).

The program appears below. Figure 3.9 shows the data paths.

```
        LDD     ADR1    OP1 INTO D
        SUBD    ADR2    OP1 − OP2
        STD     ADR3    RES INTO ADR3
```

This program is essentially like the one we developed for 16-bit addition.

Recall that in two's complement arithmetic, the final value of the carry indicates a borrow. If a borrow condition has occurred as a result

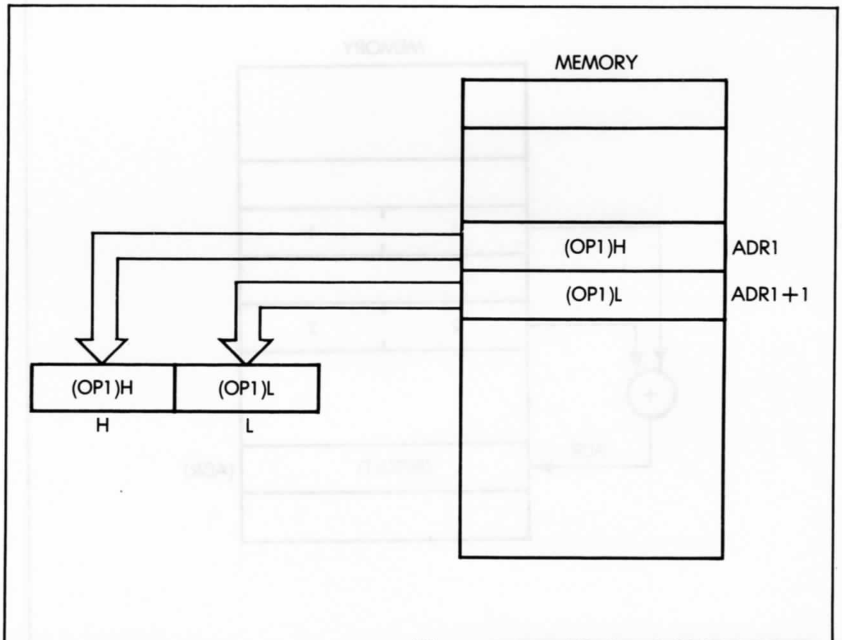


Figure 3.9: 16-Bit Load: LDD ADR1

of the subtraction, the carry bit of the condition codes register will be set, and can be tested.

The examples presented so far in this chapter are simple binary additions and subtractions. However, we may need to use another type of arithmetic, BCD arithmetic.

BCD ARITHMETIC

8-Bit BCD Addition

Chapter 1 discussed the concept of BCD arithmetic. Let's recall its features. It is essentially used for business applications, where it is imperative that every significant digit in a result be retained.

In the BCD notation, a 4-bit nibble is used to store one decimal digit (0 through 9). As a result, every 8-bit byte may store two BCD digits. (This is called a *packed BCD.*) Let's see how BCD works. Let's add two bytes, each containing two BCD digits (see Figure 3.10).

So that we can identify any problems that might come up, let's try

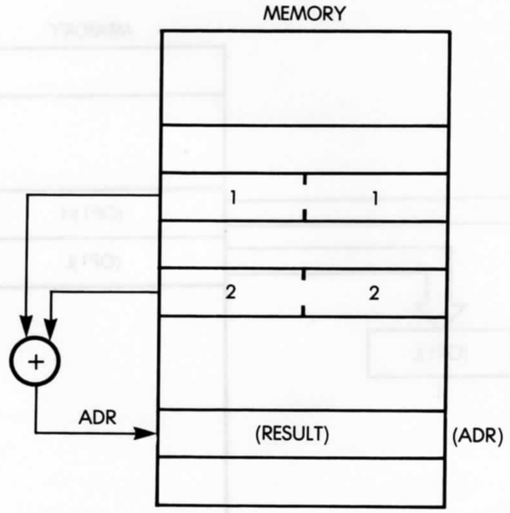


Figure 3.10: Storing BCD Digits

some numeric examples first. Let's add 01 and 02:

01 is represented by:	00000001
02 is represented by:	00000010
The result is:	00000011

This result is the BCD representation for 03. (If you are not sure of the BCD equivalent, refer to the conversion table at the end of this book.) Everything worked very simply in this case. Let's try another example.

08 is represented by:	00001000
03 is represented by:	00000011

If you obtained 00001011 as your result, you have computed the *binary* sum of 8 and 3. You have, indeed, obtained 11 in *binary*. But unfortunately, 1011 is an *illegal code in BCD*. The *BCD* representation of 11 is 00010001.

This difference stems from the fact that the BCD representation uses only the first ten combinations of 4 digits in order to encode the decimal symbols 0 through 9. Thus, the remaining six possible combinations of 4 digits are unused in BCD notation, and the illegal 1011 is one such combination. In other words, whenever the sum of two BCD digits is greater than 9, you must add 6 to the result in order to skip over the 6 unused codes.

Let's try another example. Let's add the binary representation of 6 to 1011:

```
          1011   (illegal binary result)
        + 0110   (+6)
The result is: 00010001
```

The result is, indeed, 11 in the BCD notation. We now have the correct answer.

This example illustrates one of the basic difficulties of the BCD mode: You must compensate for the six missing codes. It is necessary to use a special decimal addition adjust instruction (DAA) to adjust the result of the binary addition. (Add 6 if the result is greater than 9.)

We will use this same example to illustrate another difference. In this example, the carry is generated from the lower BCD digit (the right-most digit) into the left-most one. This internal carry must be taken into account and added to the second BCD digit. The addition instruction takes care of this automatically. However, it is often convenient to detect this internal carry from bit 3 to bit 4 (the half-carry). The H flag is provided for this purpose.

As an example, here is a program to add the BCD numbers 11 and 22:

```
LDA     #$11    LOAD LITERAL BCD 11 INTO A
ADDA    #$22    ADD LITERAL BCD 22
DAA             DECIMAL ADJUST RESULT
STA     ADR     STORE RESULT
```

The A accumulator is used in this program because the decimal addition adjust instruction always uses A. The B accumulator is not affected by the decimal adjust. However, the D accumulator high byte is affected, because it is the A accumulator.

In this program, we are using a new symbol, the $. The $ sign within the operand field of the instruction specifies that the data which follows is expressed in hexadecimal notation. The hexadecimal and BCD representations for digits 0 through 9 are identical.

In this example, we want to add the literals (or constants) 11 and 22. The # symbol indicates literal operands. The result is stored at the address ADR.

This program is analogous to the one given for 8-bit binary addition, but it uses a new instruction: DAA. Let's look at an example which illustrates the role of this instruction. We first add 11 and 22 in BCD:

$$\begin{array}{rl} 00010001 & (11) \\ +\ 00100010 & (22) \\ \hline =\underbrace{0011}_{3}\underbrace{0011}_{3} & (33) \end{array}$$

The result shown is correct, using the rules of binary addition.

Now let's add 22 and 39, using the rules of binary addition:

$$\begin{array}{rl} 00100010 & (22) \\ +\ 00111001 & (39) \\ \hline =\underbrace{0101}_{5}\underbrace{1011}_{?} & \end{array}$$

1011 is an *illegal BCD code*. Recall that BCD uses only the first 10 binary codes, and "skips over" the next 6, in order to obtain the correct result. We must now do the same, i.e. add 6 to the result:

$$\begin{array}{rl} 01011011 & \text{(binary result)} \\ +\quad 0110 & (6) \\ \hline =\underbrace{0110}_{6}\underbrace{0001}_{1} & (61) \end{array}$$

We now have the correct BCD result.

Let's look at BCD subtraction.

BCD Subtraction

BCD subtraction is complex in appearance. To perform a BCD subtraction, you must add the *ten's complement* of the number, just like you add the two's complement of a number to perform a binary subtraction. An example of this is shown by the equation RES = OP1 + 99 − OP2 + 1. Since 99 is the largest BCD number, OP2 can be subtracted from 99 without any decimal adjustment. The number obtained can then be added to OP1, and a DAA can be performed correctly. The following program illustrates this simple BCD subtraction (note that a few instructions have been added).

```
LDA     #$99      LOAD LITERAL BCD 99
SUBA    ADR2      99 − OP2
ADDA    ADR1      OP1 + (99 − OP2)
DAA               DECIMAL ADJUSTMENT
INCA              ADD ONE TO A
STA     ADR3      STORE RES
```

(*Note:* remember, the A accumulator must be used when the DAA instruction is used.)

16-Bit BCD Addition

16-bit addition is performed with a little more work than in the binary case, because the 16-bit D accumulator cannot be used. A program for such an addition appears below:

```
LDA     ADR1 + 1  LOAD (OP1) LOW INTO A
ADDA    ADR2 + 1  (OP1 + OP2) LOW
DAA               DECIMAL ADJUSTMENT
STA     ADR3 + 1  STORE RESULT LOW
LDA     ADR1      LOAD (OP1) HIGH INTO A
ADCA    ADR2      (OP1 + OP2) HIGH + CARRY
DAA               DECIMAL ADJUSTMENT
STA     ADR3      STORE RESULT HIGH
```

Packed BCD Addition

We have now learned how to perform elementary BCD addition and subtraction. However, in actual practice, BCD numbers include any number of bytes. Let's look at a simplified example of a packed BCD addition. We will assume that the two numbers located at N1 and N2 include the same number of BCD bytes and that number is called COUNT. Figure 3.11 shows the register and memory allocation. Here is the program:

```
BCDPAK  LDB     #COUNT
        LDX     #N2
        LDY     #N1
        ANDCC   #0         CLEAR CARRY
PLUS    LDA     ,X+        LOAD N2 BYTE AND INC X
        ADCA    ,Y         ADD N1 BYTE
        DAA
        STA     ,Y+        STORE RESULT INC Y
        DECB               B−1
        BNE     PLUS       LOOP UNTIL B = 0
```

N1 and N2 represent the addresses where the BCD numbers are stored. These addresses are loaded in the index registers X and Y:

```
BCDPAK  LDB     #COUNT
        LDX     #N2
        LDY     #N1
```

In anticipation of the first addition, the carry bit must be cleared. We can clear it in a number of ways. For example, we can use:

```
        ANDCC   #0         CLEAR CARRY
```

The first byte of N2 is loaded into the accumulator, then the first byte of N1 is added to it. The DAA instruction is used to obtain the correct BCD value:

```
PLUS    LDA     ,X+        LOAD N2 BYTE AND INC X
        ADCA    ,Y         ADD N1 BYTE
        DAA
```

The result is then stored in N1:

```
        STA     ,Y+        STORE RESULT INC Y
```

The forms ,X+ and ,Y+ indicate the use of a powerful capability of the index registers: the auto-increment mode. The contents of the index register are first used as the address of the operand. Then, after the instruction is finished, the index register is incremented by one. In the case of the instruction that specifies "add with carry to A," the Y register is used as a simple index register. The counter is decremented and the addition loop is executed until it reaches the value 0:

```
DEC     B       B − 1
BNE     PLUS    LOOP UNTIL B = 0
```

By using the auto-increment mode with the index registers, we can speed up and simplify the program. In this mode, the instruction first executes using the contents of the index register as the address of the operand, and then after the instruction is finished, and before the next

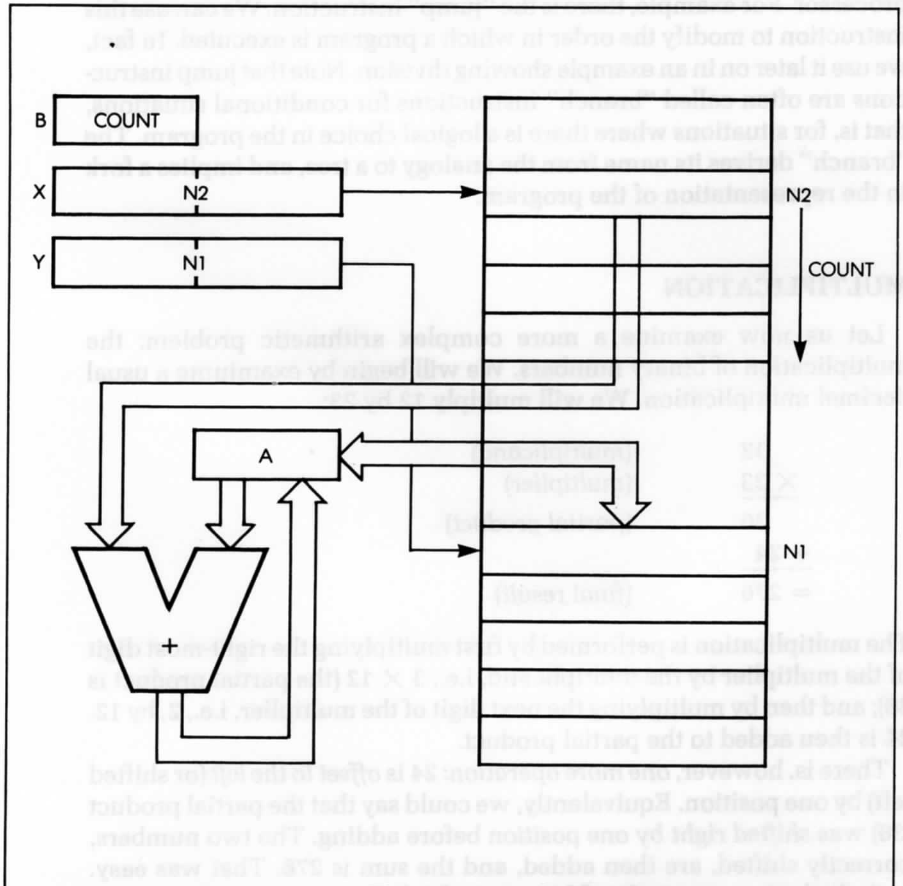


Figure 3.11: Packed BCD Add: N1 ← N2 + N1

instruction starts, the index register is incremented. See Chapter 5 for more information on addressing modes.

Instruction Types

We have now used two types of microprocessor instructions: LD, which loads a register from a memory address, and ST, which stores its contents at the specified address. These are *data transfer* instructions. We have also used *arithmetic* instructions, including ADD, SUB, ADC and SBC, that perform addition and subtraction operations. Later in this chapter we will introduce even more ALU instructions.

There are other types of instructions also available within the microprocessor. For example, there is the "jump" instruction. We can use this instruction to modify the order in which a program is executed. In fact, we use it later on in an example showing division. Note that jump instructions are often called "branch" instructions for conditional situations, that is, for situations where there is a logical choice in the program. The "branch" derives its name from the analogy to a tree, and implies a fork in the representation of the program.

MULTIPLICATION

Let us now examine a more complex arithmetic problem: the multiplication of binary numbers. We will begin by examining a usual decimal multiplication. We will multiply 12 by 23:

12	*(multiplicand)*
× 23	*(multiplier)*
36	*(partial product)*
+ 24	
= 276	*(final result)*

The multiplication is performed by first multiplying the right-most digit of the multiplier by the multiplicand, i.e., 3 × 12 (the partial product is 36); and then by multiplying the next digit of the multiplier, i.e., 2, by 12. 24 is then added to the partial product.

There is, however, *one more operation:* 24 is *offset to the left* (or shifted left) by one position. Equivalently, we could say that the partial product (36) was *shifted right by one position* before adding. The two numbers, correctly shifted, are then added, and the sum is 276. That was easy. Let's look at an example of binary multiplication; it is performed in exactly the same way. Let's multiply 5 × 3:

```
(5)           101  (multiplicand)
(3)         × 011  (multiplier)
-------------------
              101  (partial product)
             101
            000
-------------------
(15)        01111  (final result)
```

The 6809 is one of the few microprocessors with a multiply instruction. This instruction multiplies the A accumulator by the B accumulator and stores the result in the D accumulator. The results of an 8-bit by 8-bit multiplication may require up to 16 bits. This is because $2^8 \times 2^8 = 2^{16}$. A 16-bit register must, therefore, be reserved for the result. The contents of A and B are, of course, lost whenever a MUL instruction is performed.

Multiplying 16-Bit Numbers

At this point, doing an 8-bit multiply would be too easy. We'll leave it as an exercise and go on to perform a 16-bit multiply. As usual, our two numbers, OP1 and OP2, are stored at addresses ADR1 and ADR2. The memory layout is assumed to be that of Figure 3.7, except that ADR3 has four bytes, instead of two.

The 16-bit multiplication requires four 8-bit multiplications in order to obtain the correct result. This process is developed by using the rules of factoring and associativity. Figure 3.12 shows a diagram displaying 16 × 16 multiplication using bytes.

In Figure 3.12, the two low bytes of OP1 and OP2 are first multiplied. Then the low byte of OP2 and the high byte of OP1 are multiplied. This product is aligned 8-bits left because the operation is really OP1H × 2^8 × OP2L, and is, in fact, a 24-bit number with all zeroes in the low 8-bits. The low byte of OP1 is multiplied by the high byte of OP2, and the result is a 24-bit number with a low byte of zero. OP1H is multiplied by OP2H and the 16-bit result is aligned 16 bits left of the first partial product. This is because OP1H × 2^8 × OP2H × 2^8 is, in fact, a 32-bit number with the lowest two bytes zero. Here is the program for this 16 ×16 multiplication:

```
        CLR     ADR3        CLEAR HIGH 16 BITS
        CLR     ADR3+1
        LDA     ADR1+1      LOW BYTE OP1
        LDB     ADR2+1      LOW BYTE OP2
        MUL                 OP1L × OP2L
        STD     ADR3+2      FIRST PARTIAL PRODUCT
        LDA     ADR1        HIGH BYTE OP1
        LDB     ADR2+1      LOW BYTE OP2
        MUL                 OP1H × OP2L
        ADDD    ADR3+1      SECOND HIGHEST BYTE
        STD     ADR3+1
        LDA     ADR1+1      LOW BYTE OP1
        LDB     ADR2        HIGH BYTE OP2
        MUL                 OP1L × OP2H
        ADDD    ADR3+1      LOW 16 BITS DONE
        STD     ADR3+1
        BCC     NOCARY      IF NO CARRY SKIP NEXT
        INC     ADR3        ADD CARRY BIT
NOCARY  LDA     ADR1        HIGH BYTE OP1
        LDB     ADR2        HIGH BYTE OP2
        MUL                 OP1H × OP2H
        ADDD    ADR3        HIGHEST BYTE
        STD     ADR3        FINAL VALUE HIGH 16 BITS
```

This program makes use of the dual role of the accumulators as 8- or 16-bit registers. The MUL instruction puts the results into the 16-bit D accumulator. If an 8-bit number is added, the B accumulator is used. If a 16-bit number is added, the D accumulator is used. The only time a problem may occur is if a carry is created from the 16-bit addition to D. In this case, the next multiply might destroy this carry, because the multiply instruction sets the carry bit, if bit 7 of accumulator B is set. (This is useful for rounding off numbers.)

We must store the carry until it is needed. We will store it in the highest byte of the result at ADR3, which was initialized to zero at the beginning of the program. We will store it, using the following instructions:

```
        BCC     NOCARY      IF NO CARRY SKIP NEXT
        INC     ADR3        STORE A CARRY BIT
NOCARY  LDA     ADR1        HIGH BYTE OF OP1
```

The BCC instruction branches if there is no carry bit set. This means execution of the program continues at the label specified in the instruction (in this case, NOCARY), if the carry bit is clear or 0. If the carry bit is set, execution continues at the next instruction.

BINARY DIVISION

Division is a more complex problem because there is no divide instruction in the 6809. We need to develop an algorithm for writing a division program for the 6809. Let's start by examining a simple decimal division. Let's divide 254 by 12.

$$\begin{array}{rrl} & 21 & \textit{(quotient)} \\ \textit{(divisor)}\ 12 & \overline{)254} & \textit{(dividend)} \\ & \underline{24} & \\ & 14 & \\ & \underline{12} & \\ & 2 & \textit{(remainder)} \end{array}$$

We perform the division by subtracting the largest possible multiple of the divisor from the left-most digits of the dividend. The new dividend is

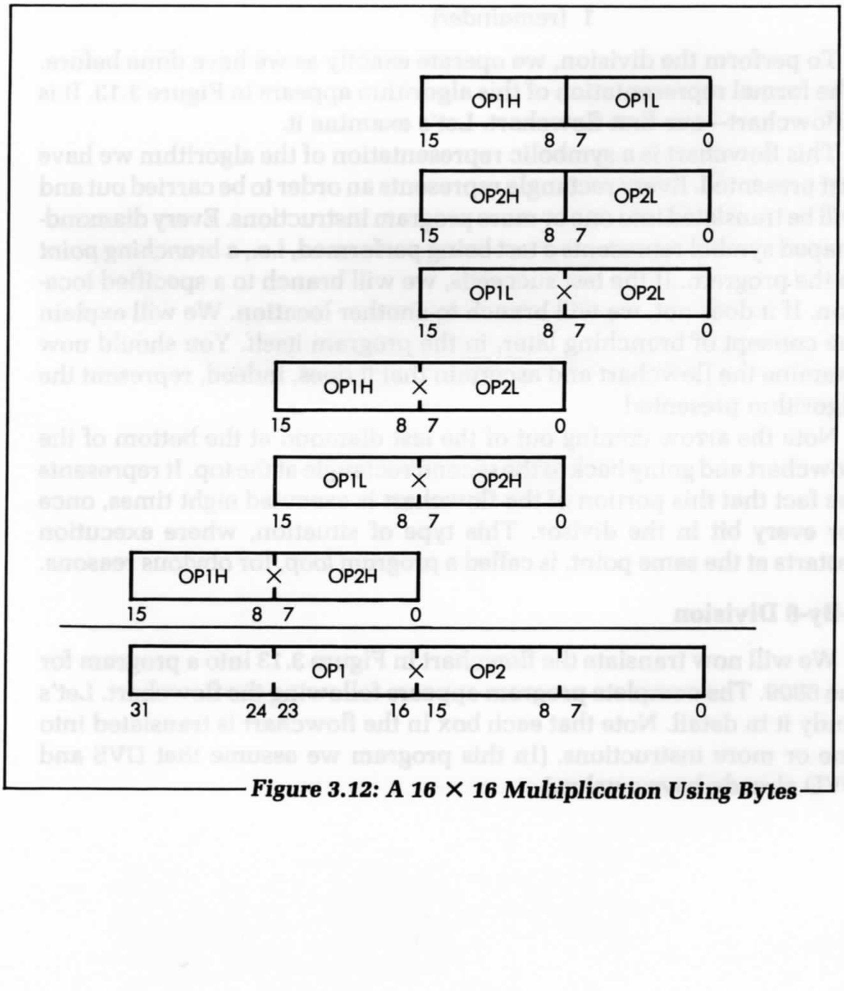


Figure 3.12: A 16 × 16 Multiplication Using Bytes

14. The multiplier of the divisor becomes the second digit of the quotient. The remainder is the result of the last subtraction.

We make trial subtractions or comparisons in order to find the largest multiple of the divisor that can be subtracted from the dividend. It should be noted that in determining the first digit of the quotient, the actual number is 20, not 2, and the number subtracted from the dividend is 240, not 24. By leaving the zeroes out, we make notation convenient, but we must not lose sight of what is actually being done.

Binary division is performed in exactly the same way as is decimal division. Let's look at an example. We divide 10 by 3:

```
             0011  (quotient)
(divisor) 11|1010  (dividend)
             11
             --
             100
              11
             ---
               1  (remainder)
```

To perform the division, we operate exactly as we have done before. The formal representation of this algorithm appears in Figure 3.13. It is a flowchart—our first flowchart. Let's examine it.

This flowchart is a symbolic representation of the algorithm we have just presented. Every rectangle represents an order to be carried out and will be translated into one or more program instructions. Every diamond-shaped symbol represents a test being performed, i.e., a *branching point* in the program. If the test succeeds, we will branch to a specified location. If it does not, we will branch to another location. We will explain the concept of branching later, in the program itself. You should now examine the flowchart and ascertain that it does, indeed, represent the algorithm presented.

Note the arrow coming out of the last diamond at the bottom of the flowchart and going back to the second rectangle at the top. It represents the fact that this portion of the flowchart is executed eight times, once for every bit in the divisor. This type of situation, where execution restarts at the same point, is called a *program loop*, for obvious reasons.

8-By-8 Division

We will now translate the flowchart in Figure 3.13 into a program for the 6809. The complete program appears following the flowchart. Let's study it in detail. Note that each box in the flowchart is translated into one or more instructions. (In this program we assume that DVS and DVD already have a value.)

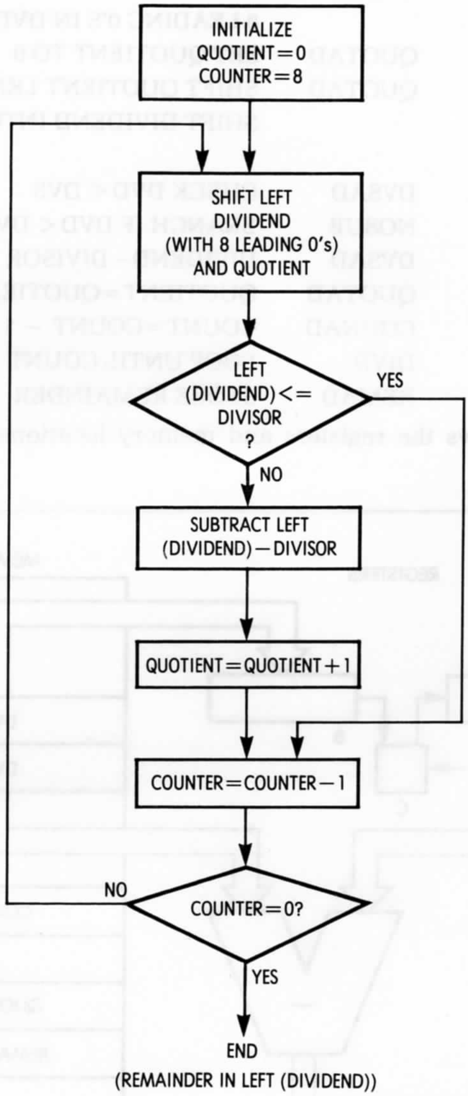


Figure 3.13: 8-Bit Binary Division Flowchart

```
DIV88    LDA     #8        SHIFT COUNTER IS 8
         STA     COUNAD
         LDB     DVDAD     LOAD DIVIDEND IN B
         CLRA              8 LEADING 0'S IN DVD
         CLR     QUOTAD    SET QUOTIENT TO 0
DIVD     ASL     QUOTAD    SHIFT QUOTIENT LEFT
         ASLB              SHIFT DIVIDEND INTO A
         ROLA
         CMPA    DVSAD     CHECK DVD < DVS
         BLO     NOSUB     BRANCH IF DVD < DVS
         SUBA    DVSAD     DIVIDEND-DIVISOR
         INC     QUOTAD    QUOTIENT=QUOTIENT + 1
NOSUB   •DEC     COUNAD    COUNT=COUNT - 1
         BNE     DIVD      LOOP UNTIL COUNT = 0
         STA     REMAD     STORE REMAINDER
```

Figure 3.14 shows the registers and memory locations used by the program.

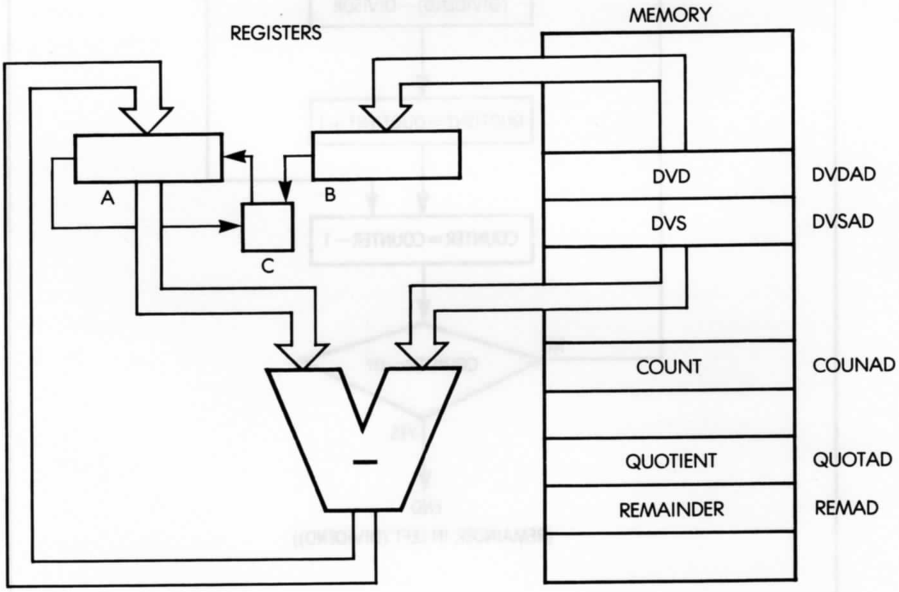


Figure 3.14: 8-By-8 Division—Registers and Memory

The two accumulators of the 6809 and five memory locations are used for this division program. The 8-bit divisor, DVS, is assumed to reside at memory address DVSAD. The dividend, DVD, is assumed to reside at memory address DVDAD. The shift count is loaded with the number 8. The B accumulator is loaded with the dividend, and the A accumulator and the quotient are cleared.

Accumulators A and B will hold the dividend as it is shifted left, one bit at a time. The result of an 8-bit by 8-bit division may require an 8-bit quotient and an 8-bit remainder. As shown in Figure 3.14, two memory locations are reserved for these results.

The first step is to load the shift counter and accumulator with the appropriate contents and to clear A and the quotient, as specified by the flowchart in Figure 3.13. This is accomplished by the following instructions:

```
DIV88   LDA     #8
        STA     COUNAD
        LDB     DVDAD
        CLRA
        CLR     QUOTAD
```

The first three instructions load the shift counter with 8, and the accumulator B with the dividend. The next two instructions clear the accumulator A and the quotient.

In this division program, the dividend and quotient are shifted left, before the dividend and divisor are compared. The dividend, DVD, is shifted into the A accumulator at each step. Accumulator A must, therefore, be initialized to the value 0. This is accomplished by the fourth instruction. Finally, the fifth instruction sets the contents of the quotient to 0.

Referring back to the flowchart in Figure 3.13, the next step is to move the quotient and dividend one bit to the left. After this is done, the divisor should be checked against the dividend to see if a subtraction takes place. This is accomplished by the next five instructions:

```
DIVD    ASL     QUOTAD
        ASLB
        ROLA
        CMPA    DVSAD
        BLO     NOSUB
```

A new type of operation, *shift*, is introduced here in the instruction ASL. It stands for "arithmetic shift left." This operation is performed in the arithmetic and logical unit. A shift left always puts a 0 into bit 0. (There are different types of shift operations; we describe them in the next chapter.)

Figure 3.15 illustrates the effect of the ASL QUOTAD with an arrow that goes from the quotient to the square that designates the carry bit C. The right-most bit of the quotient is set to 0.

The next two instructions shift the dividend left. The first, ASLB, operates like the previous instruction, except that the operand is the B accumulator. As an example, let's assume that the initial contents of B were 00001001. After the ASL instruction, the contents of B are 00010010 and the content of the carry bit is 0.

However, looking back at Figure 3.14, we want to shift the most significant bit (the MSB) of B directly into A; but, there is no instruction that will shift a double accumulator in one operation. Once the contents of B have been shifted, the left-most bit has "fallen into" the carry bit. We must collect this bit from the carry bit and shift it into the A accumulator. This is accomplished by the ROLA instruction.

ROL is still another type of shift operation. It stands for "rotate left." In a *rotation operation*, as opposed to a *shift operation*, the bit coming into the register holds the contents of the carry bit C (see Figure 3.16). This is exactly what we want. The contents of the carry bit C are loaded into the right-most part of A, and we have effectively transferred the left-most bit of B.

Figure 3.17 illustrates this sequence of instructions. The bit in the most significant position of B, marked by an X, is first transferred into the carry bit, then into the least significant position of A. Effectively, it is shifted from B into A.

The next instruction, CMPA DVSAD, is a *compare operation*. It means "compare the contents of the accumulator" (A) to the contents of DVSAD. This instruction subtracts the contents of DVSAD, from A. It is actually subtracting the divisor from the dividend shifted into A from B. It is not, however, a normal subtraction, because the contents of A are not changed. Only the condition codes are affected. For example, if A equals DVS, the

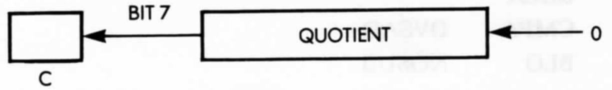


Figure 3.15: Shift Left Quotient

Z-bit in the condition code register is set. The compare operation does an internal subtraction of two operands, a memory location is subtracted from a register, and the condition codes are set according to the result of the subtraction. The operands are not changed. The condition codes are now ready for use by a *branch instruction.*

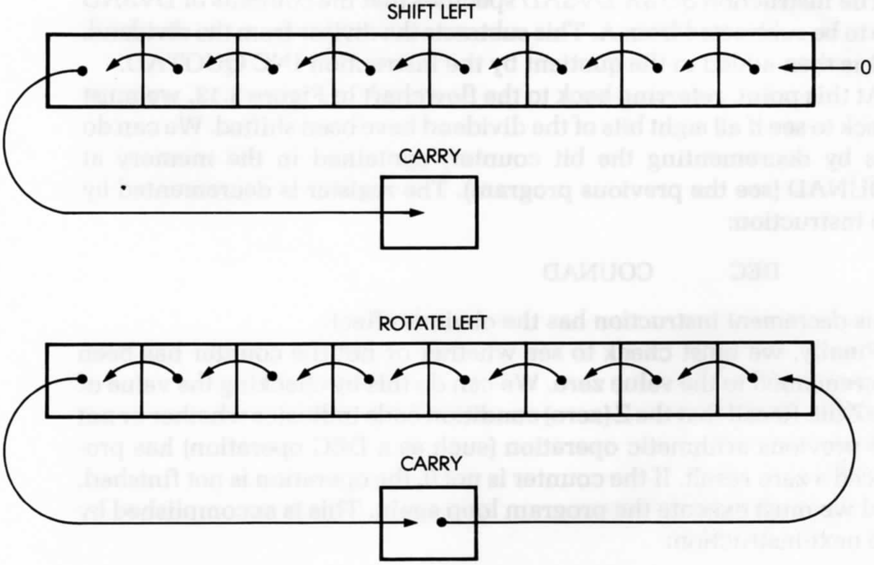


Figure 3.16: Shift and Rotate

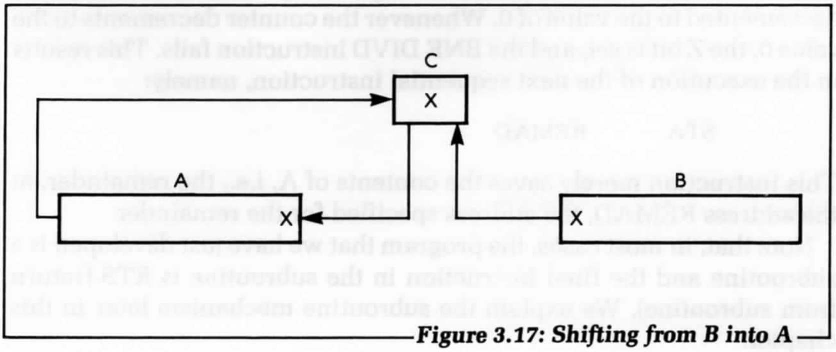


Figure 3.17: Shifting from B into A

The instruction, BLO NOSUB, is a *branch operation*. It means "branch on lower" (C = 1) to the address (label) NOSUB. If the result of a previous compare operation indicates that the accumulator A is less than the divisor, then the program branches to the address NOSUB. If the accumulator A is greater than or equal to the divisor, then no branch occurs, and the next sequential instruction is executed (i.e., the instruction "SUBA DVSAD" is executed).

The instruction SUBA DVSAD specifies that the contents of DVSAD are to be subtracted from A. This subtracts the divisor from the dividend. A 1 is then added to the quotient by the instruction INC QUOTAD.

At this point, referring back to the flowchart in Figure 3.13, we must check to see if all eight bits of the dividend have been shifted. We can do this by decrementing the bit counter, contained in the memory at COUNAD (see the previous program). The register is decremented by the instruction:

```
        DEC       COUNAD
```

This *decrement instruction* has the obvious effect.

Finally, we must check to see whether or not the counter has been decremented to the value zero. We can do this by checking the value of the Z bit. Recall that the Z (zero) condition code indicates whether or not the previous arithmetic operation (such as a DEC operation) has produced a zero result. If the counter is not 0, the operation is not finished, and we must execute the program loop again. This is accomplished by the next instruction:

```
        BNE       DIVD
```

This branch instruction specifies that whenever the Z bit is not set (NE means "not equal to zero"), a branch occurs to location DIVD. This is the *program loop*, which is executed repeatedly until the counter is decremented to the value of 0. Whenever the counter decrements to the value 0, the Z bit is set, and the BNE DIVD instruction fails. This results in the execution of the next sequential instruction, namely:

```
        STA       REMAD
```

This instruction merely saves the contents of A, i.e., the remainder, at the address REMAD, the address specified for the remainder.

Note that, in most cases, the program that we have just developed is a subroutine and the final instruction in the subroutine is RTS (return from subroutine). We explain the subroutine mechanism later in this chapter.

Important Self-Test

This program is the first significant program we have encountered so far. It includes many different types of instructions, including transfer instructions (LD, ST), arithmetic operations (SUB), logical operations (ASL, ROL), and branch operations (BLO, BNE). It also implements a program loop, in which the lower nine instructions, starting at address DIVD, are executed repeatedly. It is longer and more complex than the other arithmetic programs we have developed, therefore, you should study it carefully.

To test your understanding of the program, try the following exercise, and correctly complete it before proceeding. It will be your only real proof that you have understood the concepts presented so far. If you obtain a correct result, then you have proven that you understand how instructions manipulate information in the microprocessor, transfer this information between the memory and registers, and process it. If you do not obtain the correct result, or if you do not do this exercise, it is likely that you will experience difficulties later on when you begin writing programs yourself. Learning to program requires practice. Please pause now, take a piece of paper, or use the illustration in Figure 3.18, and complete the following exercise.

A Sample Exercise

Every time a program is written, it should be verified by hand, to ascertain that the results are correct. The goal of this exercise is to do just that, by accurately completing the table presented in Figure 3.18.

You may want to write directly on the table, or you may want to make a copy of it. For this exercise, you must determine the contents of every relevant register and memory location in the 6809 after the execution of each instruction in the program. Figure 3.19 shows the registers and memory locations used by the previous program. From left to right, they are accumulators A and B, the carry C, and the memory locations for the quotient and counter. If applicable, you should first complete the label on the left side of this table and then fill in the instructions being executed; then, on the right side of the table, you should fill in the contents of each register after each instruction has been executed. If you do not know the contents of a register, use dashes.

We will start by filling in the table together. After that you must fill in the rest of the form by yourself. The first line appears in Figure 3.19. We will assume that we are dividing 28 (DVD) by 4 (DVS).

LABEL	INSTRUCTION	A	B	C (CARRY)	QUOTIENT	COUNTER

—Figure 3.18: Form for Division Exercise—

LABEL	INSTRUCTION	A	B	C	QUOTIENT	COUNTER
		--	--	-	--	--
DIV88	LDA #8	08	--	-	--	--

—Figure 3.19: Division—After One Instruction—

The first instruction to be executed is LDA #8. The accumulator A is loaded with the number 8. This is the number of times the divide loop needs to be executed. After execution of this instruction, the contents of A are set to 8. Note that the contents of B, the quotient, and the counter are still undefined (this is indicated by dashes).

The LD instruction does not condition the carry bit, so the contents of the carry bit, C, are undefined as indicated by the dash. As shown in Figure 3.20, the next instruction loads 8 into the counter.

Figure 3.21 shows the situation after the first five instructions of the program have been executed (just before the DIVD).

The ASLB instruction performs an arithmetic shift left, and the leftmost bit of B falls into the carry bit. Figure 3.22 shows that the contents of B after the shift is 00111000. The carry bit, C, is now set to 0. The other registers are unchanged by this operation. Now that you see how the chart works, you should complete it.

Figure 3.23 shows a second iteration of the divide loop.

LABEL	INSTRUCTION	A	B	C	QUOTIENT	COUNTER
		--	--	-	--	--
DIV88	LDA #8	08	--	-	--	--
	STA COUNAD	08	--	-	--	08

Figure 3.20: Division—After Two Instructions

LABEL	INSTRUCTION	A	B	C	QUOTIENT	COUNTER
		--	--	-	--	--
DIV88	LDA #8	08	--	-	--	--
	STA COUNAD	08	--	-	--	08
	LDB DVDAD	08	IC	-	--	08
	CLRA	00	IC	-	--	08
	CLR QUOTAD	00	IC	-	00	08

Figure 3.21: Division—After Five Instructions

Programming Alternatives

The program we have just finished could have been written in several different ways. As a general rule, even the programmer can usually find ways to modify, and often improve, a program. For example, we have used an algorithm that uses shifts and subtractions; however, we could have used a method that uses only repeated subtractions until the divisor is larger than the dividend. The quotient is incremented by one for each subtraction done. This method is simpler than the first, because it is exactly the definition of division.

Improved Division Program

The program just developed is a straightforward translation of the algorithm to code. However, *effective programming requires close attention to detail*, and the length or execution time of a program can often be reduced. We will now study alternatives for improving this basic program.

To improve our division program, note that three different shift operations are used in the initial program. The quotient is shifted left, and then the dividend is shifted left in two operations, by first shifting accumulator

LABEL	INSTRUCTION	A	B	C	QUOTIENT	COUNTER
		--	--	-	--	--
DIV88	LDA #8	08	--	-	--	--
	STA COUNAD	08	--	-	--	08
	LDB DVDAD	08	IC	-	--	--
	CLRA	00	IC	-	--	08
	CLR QUOTAD	00	IC	-	00	08
DIVD	ASL QUOTAD	00	IC	0	00	08
	ASLB	00	38	0	00	08
	ROLA	00	38	0	00	08
	CMPA DVSAD	00	38	1	00	08
	BLO NOSUB	00	38	1	00	08
NOSUB	DEC COUNAD	00	38	1	00	07
	BNE DIVD	00	38	1	00	07

Figure 3.22: One Pass Through the Loop

B, then rotating accumulator A to the left. Such shifting is time consuming. However, there is a standard programming "trick" used in the case of division that is based on the following observation: every time the dividend is shifted one bit position, another bit position becomes available in the dividend register. Each time the dividend shifts left, a bit position becomes available on the right. Simultaneously, it can be observed that the first quotient (or result) uses, at most, 1 bit. We can use the bit position just vacated by the dividend, to store the first bit of the result.

After the next shift of the dividend, the size of the quotient is increased by one bit again. In other words, the bit positions freed by the dividend can be used as the quotient. To improve this program, we will make the A and B accumulators both the dividend and quotient.

LABEL	INSTRUCTION	A	B	C	QUOTIENT	COUNTER
		--	--	-	--	--
DIV88	LDA #8	08	--	-	--	--
	STA COUNAD	08	--	-	--	08
	LDB DVDAD	08	IC	-	--	--
	CLRA	00	IC	-	--	08
	CLR QUOTAD	00	IC	-	00	08
DIVD	ASL QUOTAD	00	IC	0	00	08
	ASLB	00	38	0	00	08
	ROLA	00	38	0	00	08
	CMPA DVSAD	00	38	1	00	08
	BLO NOSUB	00	38	1	00	08
NOSUB	DEC COUNAD	00	38	1	00	07
	BNE DIVD	00	38	1	00	07
DIVD	ASL QUOTAD	00	38	0	00	07
	ASLB	00	70	0	00	07
	ROLA	00	70	0	00	07
	CMPA DVSAD	00	70	1	00	07
	BLO NOSUB	00	70	1	00	07
NOSUB	DEC COUNAD	00	70	1	00	06
	BNE DIVD	00	70	1	00	06

Figure 3.23: Second Pass Through the Loop

The changed program appears below. Most of the program remains unchanged; however, there is a change, in that the quotient is not cleared during initialization and is in accumulator B at the end.

```
DIV88   LDA     #8
        STA     COUNAD
        LDB     DVDAD
        CLRA
DIVD    ASLB            SHIFT LEFT DVD AND QUOTIENT
        ROLA
        CMPA    DVSAD
        BLO     NOSUB
        SUBA    DVSAD
        INCB            INCREMENT QUOTIENT
NOSUB   DEC     COUNAD
        BNE     DIVD
        STD     RESAD   STORE QUOTIENT AND
                        REMAINDER
```

When we compare this program to the previous one, we see that the length of the division loop (the number of instructions between DIVD and the branch) has been reduced. This program has fewer instructions, which usually results in faster execution. This shows the advantage of selecting the correct registers to contain the information.

A straightforward design generally results in a program that works, although it does not necessarily result in a program that is *optimized.* It is, therefore, important to understand and use the available registers and instructions in the best possible way. This program illustrates a rational approach to register and instruction selection for maximum efficiency.

16-By-16 Bit Division

Our 16-bit division program is very similar to the 8-bit division program. The same algorithm is used; however, for the 16-bit program the A and B registers are treated together, as the D accumulator, whenever possible. Figure 3.24 shows the layout of memory. Both the quotient and remainder may be up to 16-bits long. The quotient is formed in the two memory locations of the dividend, as the dividend is shifted out. Here is our 16-bit division program:

```
DIV16   LDA     #16        SHIFT COUNTER IS 16
        STA     COUNAD
        CLRA               CLEAR ACCUMULATORS
        CLRB
DIVD    ASL     DVDAD+1    SHIFT DIVIDEND AND QUOTIENT
        ROL     DVDAD
        ROLB               SHIFT DIVIDEND INTO B
        ROLA
        CMPD    DVSAD      CHECK DVD > DVS
        BLO     NOSUB      BRANCH IF DVD < DVS
        SUBD    DVSAD      DIVIDEND − DIVISOR
        INC     DVDAD+1    INCREMENT QUOTIENT LOW
                           HALF
NOSUB   DEC     COUNAD     COUNT = COUNT − 1
        BNE     DIVD       LOOP UNTIL COUNT = 0
        STD     REMAD      STORE REMAINDER
```

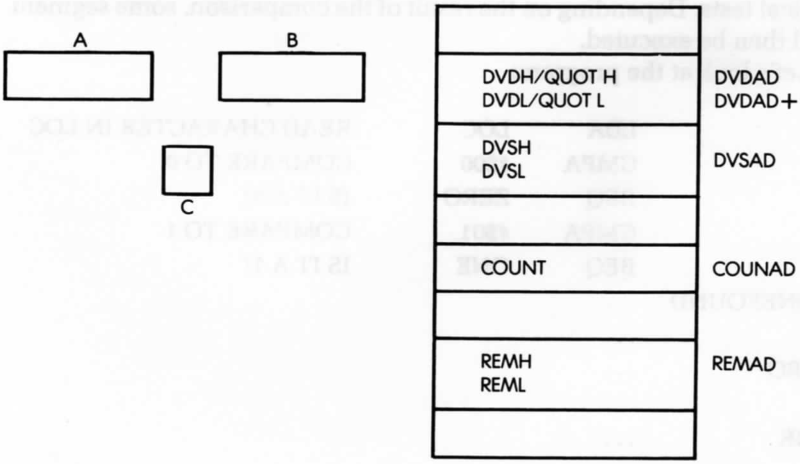


Figure 3.24: 16-By-16 Division—Register and Memory Allocation

The division programs we have presented so far have two possible flaws. One is that there is no check for division by zero; division by zero is undefined, and, therefore, it is an error condition. The divisor should be checked at the beginning of the program. If it is zero, a branch should be made to a code that handles the error. The other problem is that all of the numbers have been assumed to be unsigned numbers. This problem is usually rectified by determining the sign of the result from the signs of the dividend and the divisor before the division is done. Then the dividend and the divisor are converted to positive numbers and the division program is executed. The sign of the result is adjusted to the sign determined before the division was performed.

LOGICAL OPERATIONS

The other class of instructions, which can be executed by the ALU inside the microprocessor, is the set of *logical instructions*. These include AND, OR, and exclusive OR (EOR). In addition, one can also include the shift and rotate operations, which have already been utilized, and the comparison instruction, CMP. We will describe the AND, OR, and EOR instructions in Chapter 4.

We will now develop a brief program that checks whether a memory location, called LOC, contains the value 0, the value 1, or something else. This program uses the comparison instruction, and performs a series of logical tests. Depending on the result of the comparison, some segment will then be executed.

Let's look at the program:

```
          LDA     LOC       READ CHARACTER IN LOC
          CMPA    #$00      COMPARE TO 0
          BEQ     ZERO      IS IT A 0?
          CMPA    #$01      COMPARE TO 1
          BEQ     ONE       IS IT A 1?
NONEFOUND ...
          ...
ZERO      ...
          ...
ONE       ...
          ...
```

The first instruction, LDA LOC, reads the contents of memory location LOC and loads it into accumulator A. The data in LOC is the character we want to test. The instruction

```
        CMPA    #$00
```

compares the contents of A to the hexadecimal value 00 (i.e., the bit pattern 00000000). If this comparison instruction is successful, the Z bit in the condition code register is set to the value 1. This bit is then tested by the next branch instruction:

```
        BEQ     ZERO
```

If this comparison is successful, i.e., if the Z bit has been set to one, then the branch succeeds. The program then jumps to the address ZERO. If the test fails, the next sequential instructions are executed:

```
        CMPA    #$01
        BEQ     ONE
```

Similarly, the next branch instruction branches to location ONE, if the comparison succeeds. If none of the comparisons succeed, then the instruction at location NONEFOUND is executed:

```
NONEFOUND   . . .
```

This program demonstrates the value of the comparison instruction followed by a branch—a combination used in many of the following programs.

INSTRUCTION SUMMARY

We have now used most of the important instructions of the 6809. We have transferred values between the memory and registers. We have performed arithmetic and logical operations on data and introduced the program loop. We have tested data, and depending on the results of these tests, we have executed various portions of the program. In particular, we have made full use of the special 6809 features, such as the 16-bit accumulator and the multiply instructions. We will introduce other special instructions, PSH, PUL, and SWI, throughout the remainder of this book.

We will now examine another important programming structure, the subroutine.

SUBROUTINES

In concept, a subroutine is simply a block of instructions named by the programmer. From a more practical point of view, a subroutine must start with a label, which identifies it to the assembler. It is terminated by a special instruction, called a *return*. We will now illustrate the use of a subroutine in order to demonstrate its value. We will then examine how it is actually implemented.

Figure 3.25 illustrates how a subroutine is used. The main program appears on the left of the illustration and the subroutine appears, symbolically, on the right. Let's examine how the subroutine works. In this program, the lines of the main program are executed successively until a new instruction, CALL SUB, is met. This special instruction is the *subroutine call* and results in a transfer to the subroutine. Thus, the next instruction to be executed after the CALL SUB is the first instruction in the subroutine. This is illustrated by arrow 1 in the illustration.

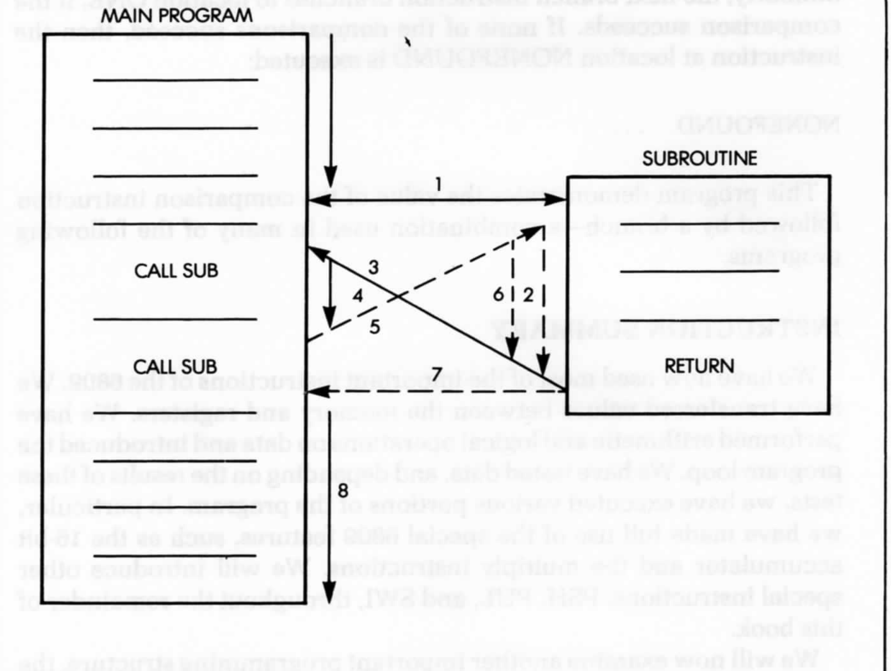


Figure 3.25: Subroutine Calls

The subprogram within the subroutine executes like any other program, as indicated by arrow 2. (We will assume that the subroutine does not contain any other calls.) The last instruction of this subroutine is a RETURN. This is a special instruction which causes a return to the main program. The next instruction to be executed after the RETURN is the one following the CALL SUB in the main program. This is illustrated by arrow 3 in the illustration. Program execution continues then, as illustrated by arrow 4.

Later, a second CALL SUB appears in the body of the main program. A new transfer occurs, as shown by arrow 5. This means that the body of the subroutine is again executed, following the CALL SUB instruction.

Whenever a RETURN is encountered within a subroutine, a return occurs to the instruction that follows the CALL SUB being executed. This is illustrated by arrow 7. Following the return to the main program, program execution proceeds normally, as illustrated by arrow 8.

The effect of the two special instructions, CALL SUB and RETURN, should now be clear. What is the value of the subroutine? The essential value of the subroutine is that it can be called from any number of points in the main program, and used repeatedly *without having to rewrite it.* An advantage of this approach is that it saves memory space, since the subroutine doesn't need to be rewritten each time. Another advantage is that the programmer can design a specific subroutine only once, and then use it repeatedly. This is a significant simplification in program design.

The disadvantage of a subroutine should be clear just by examining the flow of execution between the main program and the subroutine. A subroutine results in a *slower execution*, since extra instructions must be executed (i.e., the CALL SUB and the RETURN).

Implementation of the Subroutine Mechanism

We will now examine how the two special instructions, CALL SUB and RETURN, are implemented internally within the processor. The CALL SUB instruction causes the next instruction to be fetched at a new address. Recall that this address is contained in the program counter (PC). This means that CALL SUB substitutes new contents into register PC. In other words, the start address of the subroutine is loaded into the program counter. *Is that really sufficient*?

To answer this question, let's consider the other special instruction: RETURN. This instruction causes a return to the instruction that follows the CALL SUB. This is possible only if the address of this instruction (that is, the value of the program counter at the time the CALL SUB was executed), has been preserved somewhere.

The next problem is with saving this return address: It must always be saved in a location where it will not be erased.

We will now, however, consider the situation illustrated in Figure 3.26, where subroutine 1 contains a call to SUB2. Our mechanism must work in this case, as well as in other cases, where there may be more than two subroutines, say n "nested" calls. Whenever a new CALL is encountered, the mechanism that stores the return address must again store the program counter. Therefore, we need at least 2n memory locations for this mechanism. Additionally, we need to return from SUB2 first, and SUB1 next. In other words, we need a structure that can preserve the chronological order in which addresses were saved. This structure is the *stack*.

Figure 3.27 shows the actual contents of the stack during successive subroutine calls. The memory layout of the program appears in Figure 3.28. Let's examine the main program first. The first call, CALL SUB1, is encountered at address 100. We will assume that, in this microprocessor, the subroutine call uses 3 bytes. The next sequential address is, therefore, not 101, but 103. The CALL instruction uses addresses 100, 101, and 102. Because the control unit of the 6809 "knows" that the instruction is 3-bytes long, the value of the program counter, when the call has been completely decoded, is 103. The effect of the call is to load the

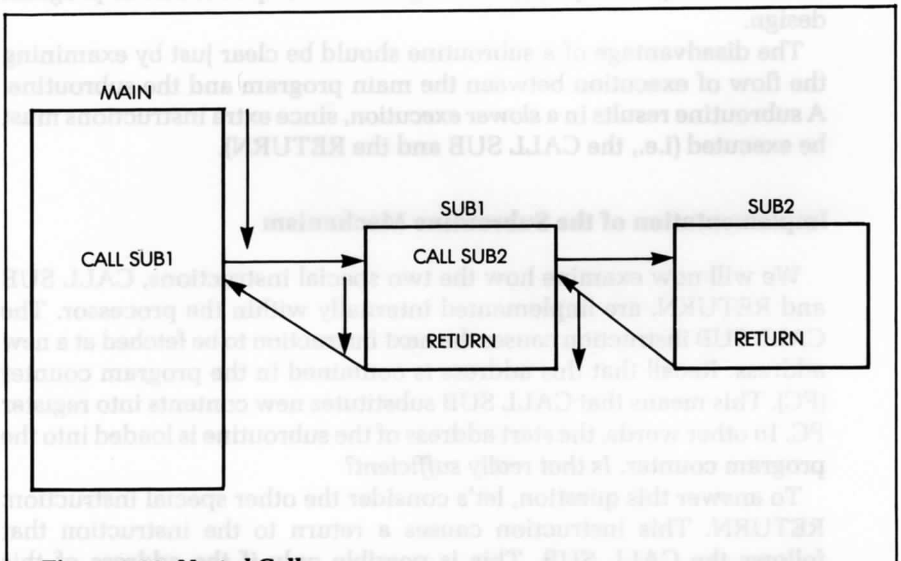


Figure 3.26: Nested Calls

value 280 in the program counter. 280 is the starting address of SUB1. In SUB1, the subroutine SUB2 (at location 900) is called at time 2 from the memory address 300. This pushes 303, the return address, to SUB1 on the stack.

We are now ready to demonstrate the effect of the RETURN instruction and the correct operation of the stack mechanism. Execution proceeds within SUB2 until the RETURN instruction is encountered at time 3. The

STACK	TIME ①	TIME ②	TIME ③	TIME ④
	103	103	103	
		303		

Figure 3.27: Stack Versus Time

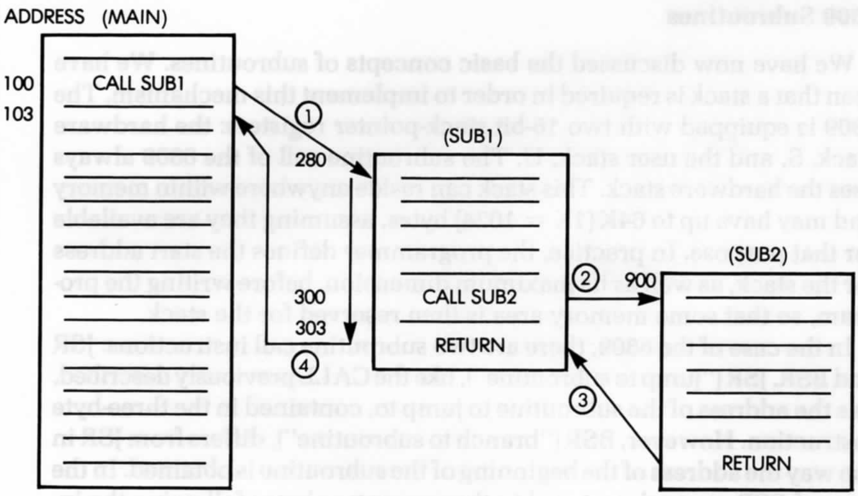


Figure 3.28: The Subroutine Calls

RETURN instruction simply pops the top of the stack into the program counter. In other words, the program counter is restored to the value it had prior to the entry into the subroutine. In our example, the top of the stack is 303. Figure 3.27 shows that, at time 3, value 303 is removed from the stack and put back into the program counter. As a result, instruction execution proceeds from address 303. At time 4, the RETURN of SUB1 is encountered. The value on top of the stack is 103. It is popped and installed in the program counter. As a result, program execution proceeds from location 103 in the main program. That is, indeed, the effect that we wanted. Figure 3.27 shows that at time 4 the stack is again empty. Thus, the mechanism to store return addresses works.

The subroutine call mechanism works up to the maximum dimension of the stack. That is why early microprocessors with 4- or 8-register stacks were essentially limited to 4 or 8 levels of subroutine calls.

Note that for clarity, Figures 3.25 and 3.26 show the subroutines to the right of the main program. In reality, the subroutines are typed as regular instructions in the program. When producing the listing of a complete program, the subroutines may be listed either at the beginning, middle, or end of the text. For this reason, they must be identified, and are, therefore, preceded by a label.

6809 Subroutines

We have now discussed the basic concepts of subroutines. We have seen that a stack is required in order to implement this mechanism. The 6809 is equipped with two 16-bit stack-pointer registers: the hardware stack, S, and the user stack, U. The subroutine call of the 6809 always uses the hardware stack. This stack can reside anywhere within memory and may have up to 64K (1K = 1024) bytes, assuming they are available for that purpose. In practice, the programmer defines the start address for the stack, as well as its maximum dimension, before writing the program, so that some memory area is then reserved for the stack.

In the case of the 6809, there are two subroutine call instructions: JSR and BSR. JSR ("jump to subroutine"), like the CALL previously described, has the address of the subroutine to jump to, contained in the three-byte instruction. However, BSR ("branch to subroutine"), differs from JSR in the way the address of the beginning of the subroutine is obtained. In the case of BSR, a number stored in the one or two bytes following the instruction opcode is added to the PC to form the value of the subroutine starting address. A one byte number can branch only +127 bytes or −128 bytes from the program counter. A two byte number allows the program to branch to a subroutine anywhere in memory. This long branch to a subroutine instruction is called LBSR.

The advantage that BSR and LBSR have over JSR is that if the whole program is moved in memory, the branch to subroutine instructions will still branch to the right address. This is because the start address is calculated *relative* to the present value of the PC, a technique useful for implementing programs stored in ROM. However, LBSR executes more slowly than JSR.

There is only one return instruction which means return from subroutine: RTS. This return instruction operates as previously described. Additionally, there is a special type of return instruction available that is used to terminate interrupt routines. This instruction, RTI, is described in the sections on the 6809 instructions and interrupts.

Finally, there are three other specialized subroutine call instructions which are analogous to a subroutine call. However, these instructions store all the registers, except the hardware stack pointer and the return address, on the stack. These instructions, called software interrupts (SWI), jump to an address stored in the highest memory locations. They are called software interrupts, because their action is the same as an interrupt, however, this action is initiated by software. The three SWI instructions are SWI, SWI2, SWI3.

The SWI instruction takes the PC for the beginning of the subroutine from addresses FFFA:FFFB. The contents of these two memory locations are then transferred to the PC. The new PC does not come from bytes following the SWI instruction. SWI2 uses addresses FFF4:FFF5 to contain the new PC; and SWI3 uses FFF2:FFF3.

Subroutine Examples

Most of the programs developed in this book would normally be written as subroutines. For example, the division program is likely to be used by many areas of the program. To facilitate and clarify program development, it is, therefore, convenient to define a subroutine with a name (for example, DIV88). At the end of the subroutine then, we would simply add the instruction RTS.

Recursion

Recursion indicates that a subroutine is calling itself. Recursive programs are not encountered very often. Their main application is in artificial intelligence programming. We will not discuss recursion any further in this book.

Subroutine Parameters

When calling a subroutine, it is normally expected that the subroutine will work on some data. For example, in the case of multiplication, it is necessary to transmit two numbers, or *parameters*, to the subroutine that performs the multiplication. For example, the multiplication subroutine expects to find the multiplier and the multiplicand in given memory locations. Using fixed memory locations illustrates one of these three methods of passing parameters:

1. through registers
2. through memory
3. through the stack.

Let's now examine each method.

Passing Parameters

Registers are often used to pass parameters. This solution is the most advantageous if registers are available, since a fixed memory location is not needed; therefore, the subroutine remains memory-independent. The disadvantage of a fixed memory location is that when it is used, other users of the subroutine must be careful to use the same convention. Also, other users must make sure that the memory location is indeed available. That is why, in many cases, a block of memory locations is reserved simply for passing parameters among various subroutines.

Using *memory* to pass parameters offers greater flexibility, but results in poorer performance. It also ties the subroutine to a given memory area.

Depositing parameters in the *stack* offers the same advantage as using registers: it is memory-independent. The subroutine simply knows that it is supposed to receive, say, two parameters which are stored on top of the stack. Naturally, this method also has disadvantages. It clutters the stack with data and, therefore, reduces the number of possible levels of subroutine calls. It also significantly complicates the use of the stack, and may require multiple stacks.

The choice is up to the programmer. Generally, it is advantageous to remain independent from actual memory locations as long as possible.

If registers are not available, a possible solution is the stack. However, if a large quantity of information must be passed to a subroutine, this information may have to reside directly in the memory. An elegant way around the problem of passing a block of data is simply to transmit a pointer to the information. Recall that a pointer is the address of the beginning of the block. A pointer can be transmitted in a register, in the stack (two-stack locations can be used to store a 16-bit address), or in a given memory location(s).

Finally, if neither of the two solutions is applicable, then an agreement may be made with the subroutine that the data will be put at some fixed memory location (the "mail-box").

Subroutine Library

There are definite advantages to structuring portions of a program into identifiable subroutines. For example, subroutines can be debugged independently, and they can have a mnemonic name. Also, provided that they can be used in other areas of the program, they become shareable. It becomes advantageous to build a library of useful subroutines. However, there is no general panacea in computer programming. Using subroutines systematically for any group of instructions that can be grouped by function can result in poor efficiency. The alert programmer will have to weigh the advantages against the disadvantages.

SUMMARY

In this chapter, we have described how information is manipulated by instructions inside the 6809. We have introduced increasingly complex algorithms, and translated them into programs. We have also examined the main types of instructions and important structures, such as program loops, stacks and subroutines.

By now you should have acquired a basic understanding of programming, and the major techniques used in standard applications. Let's go on to the next chapter and study the instructions available.

EXERCISES

3-1: *Referring only to the list of instructions at the end of the book, write a program that adds two numbers stored at memory locations LOC1 and LOC2, and deposits the results at memory location LOC3.*

3-2: *Rewrite the addition program in Exercise 3-1, using 16-bit numbers and the memory layout indicated in Figure 3.6.*

3-3: *Refer to Figure 3.6. Assume now that ADR1 does not point to the lower half of OPR1, but, instead, points to the higher part of OPR1, as illustrated in Figure 3.7. Now, write the corresponding program.*

3-4: *Write an 8-bit subtraction program.*

3-5: *Rewrite the subtraction program you wrote in Exercise 3-4, for 16-bit numbers, without using the specialized 16-bit instruction.*

3-6: *Can we place the DAA instruction in the 16-bit BCD addition program after the instruction STA ADR?*

3-7: *Compare the program in Exercise 3-6 to the one for the 16-bit binary addition. What is the difference?*

3-8: *In the packed BCD addition program, can register Y be incremented with the ADCA instruction, instead of STA?*

3-9: *Write a subtraction program for a 16-bit BCD number.*

3-10: *Divide 28 by 4 in binary, using the flowchart, and verify that the result is 7. If the result is not 7, try again. It is only when you obtain the correct result that you are ready to translate this flowchart into a program.*

3-11: *Is it really necessary to clear the quotient at the beginning of an 8-bit division program?*

3-12: *Compute the speed of a division operation, using the improved 8-bit division program. Assume that a branch will occur in 50% of the cases. Look up the number of cycles required by each instruction in the appendix. Assume a clock rate of 2 MHz (one cycle = 2.0 microseconds).*

3-13: *Write an 8 × 8 division program using the algorithm which subtracts the divisor from the dividend, until the divisor is larger than the dividend. The quotient is incremented each time a subtraction is done. Compare it* *to the 8-bit division program in this chapter, and determine whether this approach is faster or slower than the preceding one. The speeds of the 6809 instructions are given in the appendix.*

3-14: *Add a check for divide by zero to the 8 × 8 division program.*

3-15: *Make the 16 × 16 division program so that it can handle signed numbers. (Hint: Be careful when complementing a 16-bit number.)*

3-16: *Refer to the definition of the LDA LOC instruction in the next chapter. Examine the effect, if any, of this instruction on the condition codes. Is it necessary to have the second instruction of the program (CMPA $00) illustrating logical operations?*

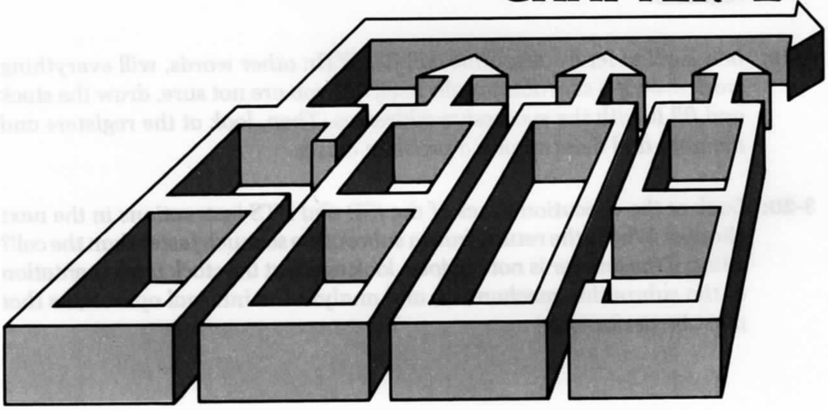
3-17: *Write a program that reads the contents of the memory location 24, and branches to an address called STAR, if there is a * in memory location 24. The bit pattern for a * in binary notation is assumed to be represented by 00101010.*

3-18: *If DIV88 is used as a subroutine, will it "damage" any internal flags or registers?*

3-10: *Is it lcgal to lct a subroutine call itself? (In other words, will everything work even if a subroutine calls itself?) If you are not sure, draw the stock and fill it with the successive addresses. Then, look at the registers and memory and determine if a problem exists.*

3-20: *Look at the execution times of the JSR and RTS instructions in the next chapter. Why is the return from a subroutine so much faster than the call? (Hint: if the answer is not obvious, look again at the stack implementation of the subroutine mechanism, and analyze the internal operations that must be performed.)*

CHAPTER 4



THE 6809 INSTRUCTION SET

IN THIS CHAPTER, we will first analyze the various classes of instructions normally available on a general-purpose microcomputer. We will then examine the variety of instructions that the 6809 offers in each of these categories, and we will see how each of these instructions affects the condition codes. We will also see these instructions used in various addressing modes.

CLASSES OF INSTRUCTIONS

It is possible to classify instructions in a number of different ways; there is no standard set of classifications. For the purpose of this discussion, we will distinguish six main categories of instructions:

1. data transfers
2. data processing
3. data pointer
4. test and branch
5. input/output
6. control.

Data Transfers

Data transfer instructions transfer data between registers, between a register and memory, and between a register and an input/output device. Some registers even offer specialized transfer instructions that can be used to organize data (for example, push and pull operations are provided for efficient stack operation).

Data Processing

Data processing instructions modify data in the computer. These instructions fall into four general categories:

1. arithmetic operations (for example, plus, minus)
2. bit manipulation (for example, set, reset)
3. logical operations (for example, AND, OR, exclusive OR)
4. skew and shift operations (for example, shift, rotate).

Data Pointer

The *data pointer instructions* perform two tasks:

1. They can load 16-bit address registers from other registers.
2. They can add a number to the address register.

They are useful for establishing blocks of data space in a program during execution.

Test and Branch

Test instructions test the bits in the condition code register for values of 0 or 1, and for combinations of these values. It is, therefore, desirable to have as many flags as possible in this register.

It is useful to have instructions that will test for:

1. combinations of bits
2. a single bit position in a word
3. the value of a register compared to the value of a memory location (greater than, less than, or equal to).

Generally, microprocessor instructions are limited to testing single bits of the flags register; in comparison to other processors, the 6809 offers better test facilities than most.

Branch instructions generally fall into three categories:

1. the *branch,* which is restricted to an 8-bit displacement field
2. the *long branch,* which specifies a full 16-bit address
3. the *branch to a subroutine,* which is used for subroutine calls.

It is convenient to have two- or even three-way branches, depending, for example, on whether one operand of a comparison is equal to, greater than, or less than the other operand. It is also convenient to have skip operations, that jump forward or backward by a few instructions. Note that a "skip" is equivalent to a "branch."

Input/Output

Input/output instructions are specialized instructions for handling input/output devices. In practice, most 8-bit microprocessors use *memory-mapped I/O,* whereby the input/output devices are connected to the address bus in the same way that the memory chips are connected, and they are addressed as such. (That is, they appear to the programmer as memory locations.)

Memory-type operations (to the address of an I/O device) normally require 3 bytes and are, therefore, slow. For efficient input/output handling in such an environment, it is usually desirable to have a short addressing mechanism. It is possible to use direct page addressing, which requires only two bytes, if the I/O device addresses are all on the same page of memory.

Control

Control instructions supply synchronization signals. These instructions can suspend or interrupt a program. They can also function as breaks or simulated interrupts. (See Chapter 6 for a detailed description of interrupts.)

THE 6809 INSTRUCTION SET

The 6809 microprocessor was designed as an improved version of the 6800, and, therefore, offers all of the capabilities of the 6800, plus several new instructions. In view of the limited number of bits available in an 8-bit opcode, one often wonders how the designers of the 6809 succeeded in implementing additional instructions. They did so by using a few unused opcodes, and adding an additional byte for indexed operations and for those operations that use 16-bit addresses and data. It is for this reason that some of the 6809 instructions can occupy up to five bytes in memory.

In this section, we will review the various instructions of the 6809, we will explore their capabilities, and group them into logical categories. Let's first examine the capabilities provided by the 6809 in terms of the five classes of instructions just described. Later, we will present an individual, in-depth description of each instruction.

Data Transfer Instructions on the 6809

We can classify the data transfer instructions on the 6809 into three categories: 8-bit transfers, 16-bit transfers, and stack operations. Let's examine each category.

8-Bit Data Transfers

Most 8-bit data transfers use load and store instructions to transfer 8-bit data between memory and the two accumulators. For example, the instruction

```
LDA     ADDR1
```

loads accumulator A from memory. Similarly,

```
STB     ADDR1
```

stores accumulator B in memory. To transfer data to and from the DP and CC registers, we use the transfer register and the exchange register instructions. The transfer instruction copies the contents of one register to another. For example, the instruction

```
TFR     A,DP
```

transfers the contents of A to the DP register. The exchange instruction actually exchanges the contents of two registers. For example,

```
EXG     A,B
```

copies the contents of B to A and of A to B.

There are several different addressing modes, *immediate, direct, indexed,* and *extended,* that we can use to access the memory location used in a load or store instruction. We discuss them in detail in Chapter 5.

16-Bit Data Transfers

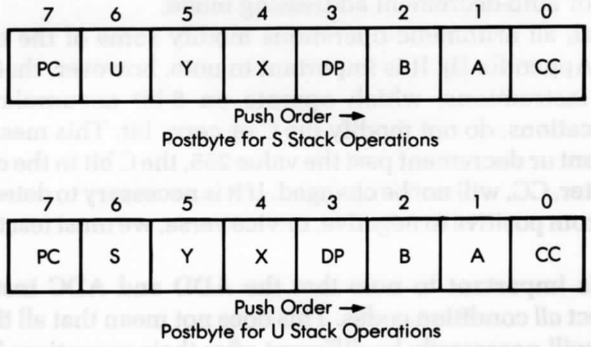
We can use the same instructions that we used for 8-bit transfers to accomplish 16-bit data transfers. For example, we can use the load and store instructions to load five 16-bit registers, D, X, Y, U, and S *from* memory, or to store them *in* memory. We can also use the TFR instruction to transfer a 16-bit register to any other 16-bit register, including the PC; and we can use the EXG instruction to exchange any two 16-bit registers, including the PC. Note that by transferring a new value into the PC, or by exchanging the PC with another register, we can cause the program to continue execution at the memory location addressed by the new value of the PC.

Stack Operations

Recall from Chapter 3 that the stack operations move data between the top of the stack and the registers. The 6809 has two stack instructions: *PUSH* and *PULL*. It has two stack pointers: the hardware stack pointer, S, and the user stack pointer, U.

The registers to be pushed onto the stack are indicated in the byte immediately following a stack instruction opcode. Each bit in this byte, called the *postbyte*, indicates a register. When a bit is set, that register is used in the stack operation (see Figure 4.1).

The stack pointer that we specify in the instruction opcode cannot be pushed or pulled. The two instructions for the S stack pointer are PSHS and PULS. PSHU and PULU are the two instructions for the U stack pointer.



7	6	5	4	3	2	1	0
PC	U	Y	X	DP	B	A	CC

Push Order →
Postbyte for S Stack Operations

7	6	5	4	3	2	1	0
PC	S	Y	X	DP	B	A	CC

Push Order →
Postbyte for U Stack Operations

Figure 4.1: Postbytes for Stack Operations

Whenever an 8-bit register is pushed on a stack, the stack pointer is decremented by 1. Whenever a 16-bit register is pushed on a stack, the stack pointer is decremented by 2. The 16-bit push puts the low byte on the stack first. Pull instructions are the same as push except, of course, they increment the stack pointer.

Data Processing Operations on the 6809

We can classify data processing operations on the 6809 into four categories: arithmetic, logical, skew and shift, and bit manipulation. Let's examine each category.

Arithmetic

As we discussed in Chapter 3, the 6809 provides three main arithmetic operations: addition, subtraction, and multiplication. Addition has two types of instructions: with carry, ADC, and without, ADD. Similarly, subtraction has two types of instructions: with carry, SBC, and without, SUB. The 6809 also provides three special instructions: DAA, COM, and NEG. The decimal addition adjust instruction, DAA, is used to implement BCD operations—usually BCD addition and subtraction. COM and NEG are two available complementation instructions. COM computes the one's complement of an accumulator or memory location, and NEG negates an accumulator or memory location into its complement format (two's complement). (*Note:* All of these instructions operate on 8-bit data. 16-bit operations are more restricted: only ADD and SUB are available on the D accumulator.) Finally, there are also increment and decrement instructions available, which operate on the accumulators and memory in 8-bit data format. We can increment or decrement the index registers and stack pointers in 16-bit format, by using an auto-increment or auto-decrement addressing mode.

In general, all arithmetic operations modify some of the condition codes (see Appendix D). It is important to note, however, that the INC and DEC instructions, which operate on 8-bit accumulators and memory locations, do not modify the C or carry bit. This means that if we increment or decrement past the value 255, the C bit in the condition codes register, CC, will not be changed. If it is necessary to detect a value changing from positive to negative, or vice versa, we must test the N and V bits.

Also, it is important to note that the ADD and ADC instructions always affect *all* condition codes. This does not mean that all the condition codes will necessarily be different *after* their execution; however, they might be.

Logical

The 6809 provides three logical operations, AND, OR (inclusive) and EOR (exclusive), plus a comparison instruction, CMP. The logical operations operate on 8-bit data, and the CMP instruction operates on 8-or 16-bit data. Let's examine these operations.

AND Each logical operation is characterized by a *truth table,* which expresses the logical value of the result as a function of the inputs. Here is the truth table for AND.

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

or

AND	0	1
0	0	0
1	0	1

The AND operation is characterized by the fact that the output is 1, only if both inputs are 1. In other words, if one of the inputs is 0, the result is guaranteed to be 0. This feature, called *masking,* is used to zero a bit position in a word.

The AND instruction is useful for clearing or "masking out" one or more bit positions in a word. Assume, for example, that we want to zero the right-most, four-bit positions in a word. The program is:

```
LDA     WORD        WORD CONTAINS 10101010
ANDA    %11110000   11110000 IS MASK
```

We assume that WORD is equal to 10101010. The result of this program is to leave the value 10100000 in the accumulator. % is used to indicate a binary value.

OR The OR instruction is the inclusive OR operation. It is characterized by the following truth table:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

or

OR	0	1
0	0	1
1	1	1

The logical OR is characterized by the fact that if one of the operands is 1, then the result is always 1. The obvious use of OR, then, is to set any bit in a word to 1.

Let's set the right-most, four bits of WORD to the value 1. The program is:

```
LDA     WORD
ORA     %00001111
```

Let's assume that WORD contains 10101010. The final value of the accumulator is 10101111.

EOR EOR stands for "exclusive OR." The exclusive OR differs from the inclusive OR in one respect: the result is 1 only if one, and only one, of the operands is equal to 1. If both operands are equal to 1, then the normal OR would give a 1 result. The exclusive OR gives a 0 result. The truth table is:

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

or

EOR	0	1
0	0	1
1	1	0

We can use the exclusive OR for comparisons. If any bit is different, then the exclusive OR of two words will be non-zero. In addition, we can use the exclusive OR to *complement* a word. We do this by performing the EOR of a word using all 1s. The program appears below:

```
        LDA       WORD
        EORA      %11111111
```

Let's assume that WORD contains 10101010. The final value of the accumulator is 01010101. We can verify that this is the complement of the original value.

We can use EOR to advantage as a "bit toggle," i.e., the bits in the accumulator will change or toggle each time an EOR is done, if the other byte used does not change.

Skew Operations (Shift and Rotate)

It is necessary here to differentiate between the shift and rotate operations. In a shift operation, the contents of the register are shifted to the left or right by one bit position. The bit falling out of the register goes into the carry bit, C, and the bit coming in is zero.

One exception exists, however: *arithmetic-shift-right*. When we perform operations on negative numbers in the two's complement format, the left-most bit is the sign bit. In the case of negative numbers, it is 1. When we divide a negative number by 2, by shifting it to the right, the sign bit should remain negative, i.e., the left-most bit should remain a 1. This is performed automatically by the ASR (arithmetic shift right) instruction. With this instruction, the bit coming in on the left is identical to the sign bit. It is a 0 if the left-most bit was a 0, and a 1 if the left-most bit was a 1. Figure 4.2 illustrates this.

A rotation differs from a shift in that the bit coming into the register is the one that will fall from the carry bit. The rotation is actually a 9-bit operation. Figure 4.3 illustrates a 9-bit rotation. For example, in the case of a right rotation, the 8 bits of the register are shifted right by one bit position. The bit falling off the right part of the register goes, as usual, into the carry bit. At this time, the bit coming in on the left end of the register is the previous value of the carry bit (before it is overwritten with the bit falling out). In mathematics this is called a 9-bit rotation, since the eight bits of the register, plus the ninth bit (the carry bit), are rotated right by one bit position. Conversely, the left rotation accomplishes the same result in the opposite direction.

Bit Manipulation

We have shown previously how we can use the logical operations to set or reset bits or groups of bits, in accumulators or memory. We can

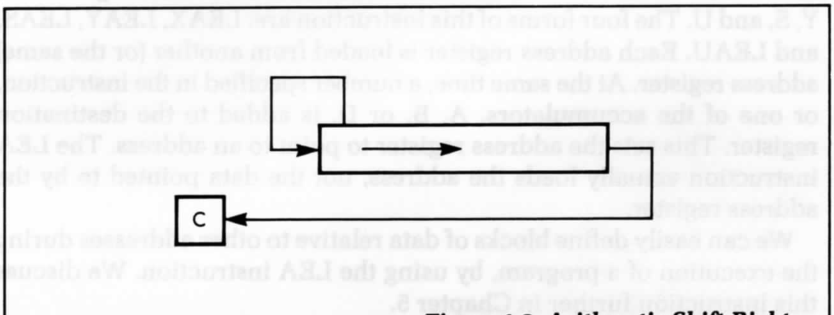


Figure 4.2: Arithmetic Shift Right

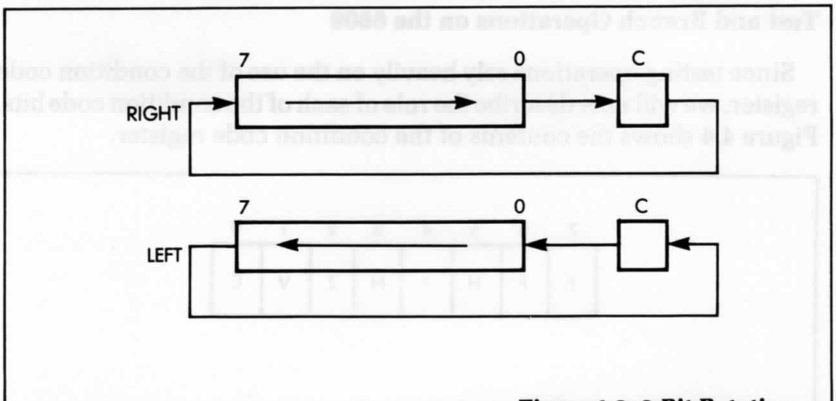


Figure 4.3: 9-Bit Rotation

also use two special instructions for operating on the condition codes register: ANDCC and ORCC. These two instructions perform the logical operations specified on the condition codes register, using the byte immediately following the instruction, as the mask. In this way, bits in CC may be cleared or set. Only the immediate mode of addressing is available with these instructions.

Finally, the bit test instruction, BIT, sets the condition codes from the result of ANDing an accumulator and an 8-bit memory location. In the bit test instruction, neither the accumulator nor the memory location is changed. The AND operation takes place and changes the condition code bits, but not the bytes being tested.

Data Pointer Instructions on the 6809

The load effective address (LEA) instruction is the data pointer instruction on the 6809. This instruction loads four address registers: X, Y, S, and U. The four forms of this instruction are: LEAX, LEAY, LEAS, and LEAU. Each address register is loaded from another (or the same) address register. At the same time, a number specified in the instruction, or one of the accumulators, A, B, or D, is added to the destination register. This sets the address register to point to an address. The LEA instruction actually loads the address, not the data pointed to by the address register.

We can easily define blocks of data relative to other addresses during the execution of a program, by using the LEA instruction. We discuss this instruction further in Chapter 5.

Test and Branch Operations on the 6809

Since testing operations rely heavily on the use of the condition code register, we will now describe the role of each of the condition code bits. Figure 4.4 shows the contents of the condition code register.

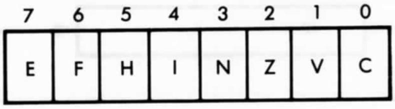


Figure 4.4: The Condition Code Register

C is the carry bit, V is overflow, Z is zero, and N is negative. Bits 4, 6, and 7 are used with interrupts. The code H is used for BCD arithmetic and cannot be tested directly. The other four codes (C, V, Z, N) can be tested in conjunction with conditional branch instructions. We will now describe the role of each condition code bit.

Carry (C)

In the case of nearly all microprocessors, and of the 6809 in particular, the carry bit assumes a dual role. First, it is used to indicate if an addition or subtraction operation has resulted in a carry (or borrow). Second, it is used as a ninth bit in the case of shift and rotate operations. Using a single bit to perform both roles facilitates some operations, such as a dlvision operation. This should be clear from the description of division operations given in Chapter 3.

When learning to use the carry bit, it is important to remember that all arithmetic operations either set or reset it, depending on the result of the instructions. Similarly, all shift and rotate operations use the carry bit and either set or reset it, depending on the value of the bit coming out of the word.

In the case of logical instructions, we can use ANDCC and ORCC to directly reset or set the carry bit. Instructions which affect the carry bit are: ADD, ADC, SUB, SUBC, ANDCC, ORCC, ASL, ASR, LSL, LSR, ROL, ROR, CLR, CMP, COM, NEG, DAA, and MUL. Also, some data transfer instructions and control instructions, including PULS, PULU, TFR, EXG, RTI, and CWAI, affect the C bit, and all other condition code bits, because they load the condition code register.

Overflow (V)

We described the overflow flag in Chapter 1, when we introduced the two's complement notation. The overflow flag detects the fact that, during an addition or subtraction, the sign of the result was "accidentally" changed, due to the overflow of the result into the sign bit. (Recall that, using an 8-bit representation, the largest positive number and the smallest negative number in two's complement are +127 and −128, respectively.)

The V condition code bit is affected by ADC, ADD, ASL, CMP, DEC, INC, LSL, NEG, ROL, SBC, and SUB. The following instructions always reset the V bit: AND, OR, BIT, CLR, COM, EOR, LD, SEX, ST, and TST. The state of the V bit is undefined for the DAA instruction.

The Half-Carry Bit (H)

The half-carry flag indicates a possible carry from bit 3 into bit 4 during an addition operation. In other words, it represents the carry from the low-order nibble (group of 4 bits) into the high-order nibble. Clearly, the half-carry flag is primarily used for BCD operations. In particular, it is used internally by the decimal addition adjust (DAA) instruction, in order to adjust the result to its correct value.

The half-carry flag is set during an 8-bit addition, when there is a carry from bit 3 to bit 4; it is reset when there is no carry. A 16-bit addition does not affect the H bit.

The 8-bit ADD and ADC instructions affect the H bit. The ASL, ASR, NEG, SBC, and the 8-bit forms of the CMP and SUB instructions leave the H bit undefined.

Zero (Z)

The Z condition code bit indicates whether or not the value of a byte which has been computed or is being transferred, is zero. The Z condition code bit is often used with comparison instructions to indicate a match.

In the case of an operation resulting in a zero result, or in the case of a data transfer, the Z bit is set to 1 whenever the byte, or 16-bit word, is zero. Otherwise, Z is reset to 0.

The following instructions condition the value of the Z bit: ADC, ADD, AND, OR, ASL, ASR, BIT, CMP, COM, DAA, DEC, EOR, INC, LD, NEG, ST, TST, LEAX, LEAY, LSL, LSR, MUL, SEX, ROL, ROR, SBC, and SUB. The CLR instruction always sets the Z bit.

Negative (N)

This condition code bit reflects the value of the most significant bit of a result, or of a byte (or 16-bit data) being transferred. In two's complement notation, the most significant bit represents the sign: 0 indicates a positive number, and 1 indicates a negative number. As a result, bit 7 (or bit 15, for 16-bit numbers) is called the negative bit.

In most microprocessors, the sign bit plays an important role when communicating with input/output devices, because it is usually the most convenient bit to test. When examining the status of an input/output device, reading the status register automatically conditions the negative bit, which is then set to the value of bit 7 of the status register and can be conveniently tested by the program. This is why the status register of most input/output chips connected to microprocessor systems have their most important indicator (usually ready/not ready) in bit position 7.

The following instructions affect the negative bit: ADC, ADD, AND, OR, ASL, ASR, BIT, CMP, COM, DAA, DEC, EOR, INC, LD, ST, TST, LSL, SEX, NEG, ROL, ROR, SBC, and SUB. The CLR and LSR instructions always clear the N bit.

Summary of the Condition Code Bits

The condition code bits automatically detect special conditions within the ALU of the microprocessor. We can conveniently test them by using specialized instructions—so that specific actions can be taken in response to the condition detected. It is important to understand the role of the various indicators available, since most decisions made within the program are determined by the value of these condition code bits. All branches executed within a program jump to specified locations, depending on the status of these bits. The only exception involves the interrupt mechanism (described in Chapter 6), which may cause jumping to specific locations whenever a hardware signal is received on specialized pins of the 6809.

At this point, it is only necessary to remember the main function of each bit. When programming, you may want to refer to the description of each instruction in this chapter to verify its effect on the various condition code bits. Most bits can be ignored most of the time, and if you are not yet familiar with them, you should not feel intimidated by their apparent complexity. Their use will become more clear as you continue to examine other application programs.

The Branch Instructions on the 6809

A *branch instruction* causes a forced branching to a specified program address. It changes the normal flow of program execution from a sequential mode into one where a different segment of the program is suddenly executed. Branches may be conditional or unconditional. An *unconditional* branch is one where the branching occurs to a specific address, regardless of any other condition. A *conditional* branch is one where the branching occurs to a specific address only if one or more conditions are met. This is the type of jump instruction used to make decisions based upon data or computed results.

To describe conditional branch instructions, it is necessary to understand the role of the condition code register (explained in the preceding section), since all branching decisions are based upon these condition bits. We will now examine, in more detail, the branch instructions provided by the 6809.

The two main types of branch instructions provided by the 6809 are branch instructions within the main program (called branches), and the special branch instructions used to jump to and from a subroutine (JSR, BSR, and RTS). As a result of any branch instruction, the program counter (PC) is reloaded with a new address, and the usual program execution resumes from that point on. The full power of branch instructions can be understood only in the context of the various addressing modes provided by the microprocessor. (We cover this topic in Chapter 5 when we discuss addressing modes.) We will only consider here the other aspects of these instructions.

Branches may be either unconditional (always branching to a specified memory address) or conditional. In the case of a conditional branch, one or more of the four condition code bits, the Z, C, V, and N bits, may be tested for the value 0 or 1.

The corresponding abbreviations for the individual bits are:

BCC	= carry clear	(C = 0)
BCS	= carry set	(C = 1)
BEQ	= equal to zero	(Z = 1)
BNE	= not equal to zero	(Z = 0)
BMI	= minus	(N = 1)
BPL	= plus	(N = 0)
BVC	= overflow clear	(V = 0)
BVS	= overflow set	(V = 1)

There are several branch instructions which test for combinations of the condition code bits. These are frequently used after a compare (CMP) instruction. Here are the abbreviations for these conditional branch instructions:

BGE	= greater than or equal to
BGT	= greater than
BHI	= higher
BLE	= less than or equal to
BLS	= lower or same
BLT	= less than

There are two branch instructions that have the same opcodes as other branch instructions, and are available in the assembler. (This was done for the convenience of the programmer.) These two instructions are:

BHS = higher or same		duplicates BCC
BLO = lower		duplicates BCS

Even though the same opcode is executed, it is sometimes convenient to give two instruction names to one opcode.

The unconditional branch instruction is BRA (branch always). BRN is the "branch never" instruction, which never branches. It is really a null operation.

The availability of conditional branches is a powerful resource in a computer, although this resource is generally not provided on most 8-bit microprocessors. This resource does, however, improve the efficiency of programs by implementing in a single instruction what normally would require two instructions. There is, however, one drawback to branch instructions on most computers: The address specified with the branch instruction is only one byte in length. This byte is added to the PC to obtain the new address. This means that a branch may move the PC only 127 bytes forward or 128 bytes backwards from the location of the branch instruction. Branching farther is not possible. However, the 6809 does have special *long-branch* instructions.

The long-branch has a 16-bit address specified with the instructions. When added to the PC, branching is allowed to any of the 65,536 memory locations on the 6809. This type of branch instruction removes the need to branch to a jump instruction (JMP). We form the assembly language mnemonic for a long-branch instruction by adding the letter L in front of a branch instruction mnemonic. The opcodes for long-branches are different from their corresponding short-branch instructions. See Appendix D for a list of opcodes.

Finally, a special return instruction, RTI, is provided in the case of interrupt routines. Chapter 6 will discuss this instruction in detail.

One more type of specialized branch is available: the *software interrupt* (SWI) instruction. Recall that the SWI instruction is a single instruction which saves all of the registers on the hardware stack S and then performs a jump by fetching a new PC from one of three addresses at the high end of memory. The three possible locations for the byte pairs which form the new PC are: (FFFA):(FFFB), (FFF4):(FFF5), and (FFF2):(FFF3). SWI is a powerful instruction, because it saves the entire machine state. It is frequently used to jump to a special program, which starts and completes other programs in the computer.

Input/Output Instructions on the 6809

We can address input/output devices in one of two ways: as memory locations (using any one of the instructions described previously), or by using specific input/output instructions. Chapter 6 will examine input/output techniques in detail. The 6809 has no special instructions devoted to input/output. Usual memory addressing instructions use three bytes: one for the opcode and two for the address. As a result, these instructions execute slowly, since they require three memory accesses. However, if we use the special "direct page" addressing mode, where the address is formed by the direct page register and a byte in the instruction, then the instructions to access an input/output device need only be two bytes in length. This allows faster execution.

Control Instructions on the 6809

Control instructions modify the operating mode of the CPU and manipulate its internal status information. The 6809 provides three control instructions: NOP, SYNC, and CWAI.

The NOP instruction is a no-operation instruction which does nothing for two cycles. It is typically used either to introduce a deliberate delay (2 cycles = 2 microseconds with a 4MHZ crystal) or to fill the gaps created in a program during the debugging phase. The opcode of the NOP instruction is 12 hexadecimal. Executing NOPs does not cause damage nor stop program execution.

The SYNC instruction is used in conjunction with interrupts. It actually suspends the operation of the CPU and puts the data and address buses into a high impedance state. The CPU then resumes operation whenever an interrupt signal is received. A sync is often placed at the end of a program during the debugging phase, as there is usually nothing else to be done by the main program. The program must be explicitly restarted when a SYNC is used.

Finally, the last control instruction is clear condition code bits and wait for an interrupt (CWAI). This instruction ANDs an immediate byte with the condition code register, which may clear any bit, stores all the registers on the hardware stack, and then waits for an interrupt. When an interrupt occurs, the machine state need not be saved before servicing the interrupt. Note that the data and address buses are not put in the high impedance state, as they were during the SYNC instruction.

SUMMARY

We have now described the six categories of instructions available on the 6809. Specific details on the individual instructions are presented in the following section of this chapter. It is not necessary to understand the role of each instruction in order to start programming. At the beginning, it is sufficient to know a few essential instructions of each type; however, as you begin writing your own programs, you will want to learn all the instructions on the 6809, so that you can make your programs as efficient as possible.

We have not yet described one important aspect of programming: the addressing techniques implemented on the 6809 that facilitate data retrieval within the memory space. We will cover these addressing techniques in the next chapter.

EXERCISES

4-1: *Write a three-line program that zeroes bits 1 and 6 of WORD.*

4-2: *What will happen if we use a MASK equaling 11111111 with an AND instruction?*

4-3: *What will happen if we use the instruction ORA %10101111 and A contains 10101111?*

4-4: *What is the effect of ORing with FF hexadecimal?*

4-5: *What is the effect of EOR, if we use a register with 00 hexadecimal, instead of 11111111, to complement a byte?*

THE 6809 INSTRUCTIONS: INDIVIDUAL DESCRIPTIONS

Abbreviations and Symbols for Instruction Descriptions

Flags

x — Flag changed according to operation or result of instruction
— Flag unchanged (space)
0 — Flag cleared by instruction
1 — Flag set by instruction
? — Flag unpredictably changed by instruction

Notation

A, B, D, X, Y, S, U, PC, DP, CC	— registers
ACCX	— either A or B
RR	— 16-bit register (D, X, Y, U, S)
←	— data transfer
← →	— exchange data
M	— byte memory operand of valid type for given instruction
MM	— 16-bit memory operand
ADDRM	— Address of M or MM
N	— 8-bit immediate mode operand
NN	— 16-bit immediate mode operand
-high	— most significant byte of 16-bit register
-low	— least significant byte of 16-bit register
<	— direct addressing mode
>	— extended addressing mode
∧	— AND function
∨	— OR function
(XOR)	— exclusive OR function

(*Note:* All of the numbers used in instruction examples are in hexadecimal notation.)

ABX — *Add Accumulator B into Index Register X*

Mnemonic: ABX

Function: X ← X + B

Description: The unsigned 8-bit contents of accumulator B are added into index register X.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: inherent

Example: ABX

before:	*after:*
X:$8006	X:$80D4
B:$CE	B:$CE

ADC — *Add with Carry into Accumulator*

Mnemonics: ADCA M; ADCB M

Function: ACCX ← ACCX + M + C

Description: The carry bit and the memory operand are added into the specified accumulator.

Condition codes:

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

Addressing modes:

immediate
extended
direct
indexed

Example: ADCA ,X

before:	*after:*
X:$3C50	X:$3C50
A:$14	A:$37
CC:$0B	CC:$00
$3C50:$22	$3C50:$22

ADD (8-bit) —*Add Memory into Accumulator*

Mnemonics: ADDA M; ADDB M

Function: ACCX ← ACCX + M

Description: The 8-bit memory operand is added into the specified accumulator.

Condition codes:

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

Addressing modes:

immediate
extended
direct
indexed

Example: ADDB >$55FE

before:	*after:*
B:$F2	**B:$2B**
CC:$13	**CC:$11**
$55FE:$39	**$55FE:$39**

ADD (16-bit) — Add Memory into Accumulator

Mnemonic: ADDD MM

Function: D ← D + MM

Description: The 16-bit memory operand is added into the D accumulator.

Condition codes:

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

Addressing modes:

immediate
extended
direct
indexed

Example: ADDD #$322

before:	*after:*
D:$000F	D:$0331
CC:$00	CC:$00

AND — *AND Memory into Accumulator*

Mnemonics: ANDA M; ANDB M

Function: ACCX ← ACCX ∧ M

Description: The contents of the memory operand and the specified accumulator are logically ANDed; the result is stored in the source accumulator.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing modes:
immediate
extended
direct
indexed

Example: ANDA <EF

before:	*after:*
A:$8B	A:$0B
DP:$7E	DP:$7E
CC:$32	CC:$30
$7EEF:$0F	$7EEF:$0F

AND —*AND Immediate Data into Condition Code—Register*

Mnemonic: ANDCC #N

Function: $CC \leftarrow CC \wedge N$

Description: The condition code register is logically ANDed to the immediate data byte; the result is stored in the condition code register. This instruction may be used to clear a specific bit, e.g., an interrupt mask.

Condition codes:

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

(Changed according to operand.)

Addressing mode: immediate

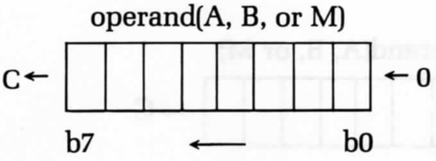
Example: ANDCC #$AF

before:	*after:*
CC:$79	CC:$29

ASL *Arithmetic Shift Left*

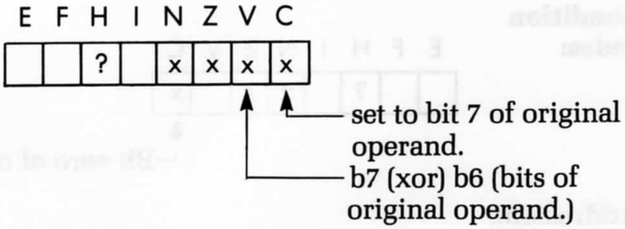
Mnemonics: ASLA; ASLB; ASL M

Function:



Description: All of the bits in the operand are shifted left by one position. Bit 7 is transferred to the carry bit; bit 0 becomes a zero.

Condition codes:



E	F	H	I	N	Z	V	C
		?		x	x	x	x

set to bit 7 of original operand.

b7 (xor) b6 (bits of original operand.)

Addressing modes:

inherent
extended
direct
indexed

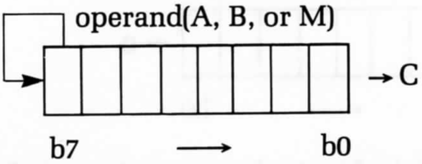
Example: ASLB

before:	*after:*
B:$A5	B:$4A
CC:$04	CC:$03

ASR — *Arithmetic Shift Right*

Mnemonics: ASRA; ASRB; ASR M

Function:



Description: All of the bits in the operand are shifted right by one position. Bit 0 is transferred to the carry bit; bit 7 remains unchanged (this allows the shift to be used on signed binary numbers).

Condition codes:

E	F	H	I	N	Z	V	C
		?		x	x		x

Bit zero of original operand.

Addressing modes: inherent
extended
direct
indexed

Example: ASR >$1A04

before:	after:
CC:$00	CC:$09
$1A04:$E5	$1A04:$F2

BCC *Branch on Carry Clear*

Mnemonics: BCC N; LBCC NN

Function: If C=0 then: PC ← PC + N (or) PC ← PC + NN

Description: If the C bit is clear, then a PC relative branch is executed. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BCS *Branch on Carry Set*

Mnemonics: BCS N; LBCS NN

Function: If C=1 then: PC ← PC + N (or) PC ← PC + NN

Description: If the carry bit is set, then a PC relative branch is executed. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BEQ *Branch on Equal*

Mnemonics: BEQ N; LBEQ NN

Function: If Z=1 then: PC ← PC + N (or) PC ← PC + NN

Description: If the zero bit is set, then a PC relative branch is executed. This condition is true after a subtract or compare on any binary values, if the register was the same as the memory operand. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BGE — *Branch on Greater Than or Equal To*

Mnemonics: BGE N; LBGE NN

Function: If (N(XOR) V) = 0 then: PC ← PC + N (or)
PC ← PC + NN

Description: If the N and V bits are either both set or both clear, then PC relative branch is executed. These conditions are true after a subtract or compare on two's complement values if the register was greater than or equal to the memory operand. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BGT *Branch on Greater Than*

Mnemonics: BGT N; LBGT NN

Function: If Z ∧ (N (XOR) V)=0 then: PC ← PC + N (or)
PC ← PC + NN

Description: If the N and V bits are either both set or both clear and the Z bit is clear, then a PC relative branch is executed. These conditions are true after a subtract or compare on two's complement values if the register was strictly greater than the memory operand. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BHI *Branch on Higher*

Mnemonics: BHI N; BHI NN

Function: If $(C \vee Z)=0$ then: $PC \leftarrow PC + N$ (or)
$PC \leftarrow PC + NN$

Description: If the C and Z bits are both clear, then a PC relative branch is executed. These conditions are true after a subtract or compare on unsigned values if the register was strictly greater than the memory operand. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BHS *Branch on Higher or Same*

Mnemonics: BHS N; BHS NN

Function: If C=0 then: PC ← PC + N (or) PC ← PC + NN

Description: If the C bit is clear, then a PC relative branch is executed. This is a duplicate mnemonic for the BCC instruction. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BIT — *Test Bits*

Mnemonics: BITA M; BITB M

Function: $ACCX \wedge M$

Description: The specified accumulator and the memory operand are logically ANDed and the result is discarded. Only the condition code bits are affected; neither operand is affected.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing modes:

immediate
extended
direct
indexed

BLE — *Branch on Less than or Equal to*

Mnemonic: BLE N; LBLE NN

Function: If Z ∨ (N (XOR) V) = 1 then: PC ← PC + N (or)
PC ← PC + NN

Description: If either, but not both, of the N and V bits is set, or if the Z bit is set, then a PC relative branch is executed. These conditions are true after a subtract or compare on two's complement values, if the register was less than or equal to the memory operand. The short branch can access any instruction in the range $+129$ to -126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BLO *Branch on Lower*

Mnemonics: BLO N; LBLO NN

Function: If C=1 then: PC ← PC + N (or) PC ← PC + NN

Description: If the carry bit is set, then a PC relative branch is executed. This condition is true after a subtract or compare on unsigned values, if the register was strictly lower than the memory operand. This is a duplicate mnemonic for the BCS instruction. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BLS *Branch on Lower or Same*

Mnemonics: BLS N; LBLS NN

Function: If $(C \vee Z) = 1$ then: $PC \leftarrow PC + N$ (or)
$PC \leftarrow PC + NN$

Description: If either or both of the C or Z bits is set, then a PC relative branch is executed. These conditions are true after a subtract or compare on unsigned values, if the register was lower than or the same as the memory operand. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BLT — *Branch on Less Than*

Mnemonics: BLT N; LBLT NN

Function: If (N (XOR) V)=1 then: PC ← PC + N (or)
PC ← PC + NN

Description: If either, but not both, of the N and V bits is set, then a PC relative branch is executed. This condition is true after a subtract or compare on two's complement values, if the register was strictly less than the memory operand. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BMI *Branch on Minus*

Mnemonics: BMI N; LBMI NN

Function: If N=1 then: PC ← PC + N (or) PC ← PC + NN

Description: If the N bit is set, then a PC relative branch is executed. This condition is generally true after an operation, if the sign bit of the result was set. It is preferable to use the BLT instruction when testing two's complement results. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BNE *Branch on Not Equal*

Mnemonics: BNE N; LBNE NN

Function: If Z=0 then: PC ← PC + N (or) PC ← PC + NN

Description: If the Z bit is clear, then a PC relative branch is executed. This condition is true after a subtract or compare on any binary values, if the register was not equal to the memory operand. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BPL *Branch on Plus*

Mnemonics: BPL N; LBPL NN

Function: If N=0 then: PC ← PC + N (or) PC ← PC + NN

Description: If the N bit is clear, then a PC relative branch is executed. This condition is generally true after an operation, if the sign bit of the result was clear. It is preferable to use the BGE instruction on two's complement results. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BRA *Branch Always*

Mnemonics: BRA N; LBRA NN

Function: PC ← PC + N (or) PC ← PC + NN

Description: A PC relative branch is unconditionally executed.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BRN —*Branch Never*

Mnemonics: BRN N; LBRN NN

Function: No operation.

Description: No branch is executed. This instruction is, essentially, a NOP and is included to maintain the symmetry of the instruction set.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BSR *Branch to Subroutine*

Mnemonics: BSR N; LBSR NN

Function:

S ← S − 1 ; (S) ← PC-low
S ← S − 1 ; (S) ← PC-high
PC ← PC + N (or) PC ← PC + NN

Description: The program counter is pushed onto the hardware stack and a PC relative branch is executed. The RTS (return from subroutine) instruction reverses this procedure and returns control to the instruction following the BSR instruction. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BVC — *Branch on Overflow Clear*

Mnemonics: BVC N; LBVC NN

Function: If V=0 then: PC ← PC + N (or) PC ← PC + NN

Description: If the V bit is clear, then a PC relative branch is executed. This condition is true after an operation of two's complement values, if the result was valid, i.e., there was no overflow. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

BVS *Branch on Overflow Set*

Mnemonics: BVS N; LBVS NN

Function: If V=1 then: PC ← PC + N (or) PC ← PC + NN

Description: If the V bit is set, then a PC relative branch is executed. This condition is true after an operation on two's complement values, if the result was invalid, i.e. there was an overflow. The short branch can access any instruction in the range +129 to −126 bytes, relative to the first byte of the branch instruction. The long branch can access any instruction in the 64K memory area.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: relative

CLR *Clear*

Mnemonics: CLRA; CLRB; CLR M

Function: ACCX ← 0 (or) M ← 0

Description: The specified operand is cleared to 0. *Note:* A memory operand will be read before being cleared. This may have significance in non-memory (i.e., I/O) accesses.

Condition codes:

E	F	H	I	N	Z	V	C
				0	1	0	0

Addressing modes:

inherent
extended
direct
indexed

Example: CLR >$F23

before:	*after:*
$0F23:$E2	$0F23:$00

CMP (8-bit) — *Compare Memory from Accumulator*

Mnemonics: CMPA M; CMPB M

Function: ACCX − M

Description: The memory operand is subtracted from the specified accumulator, and the result is discarded. Only the condition code bits are affected; neither operand is affected.

Condition codes:

E	F	H	I	N	Z	V	C
		?		x	x	x	x

Addressing modes:

immediate
extended
direct
indexed

Example: CMPA #6

before:	after:
A:$05	A:$05
CC:$52	CC:$59

CMP (16-bit) *Compare Memory from Register*

Mnemonics: CMPD MM; CMPX MM; CMPY MM;
CMPU MM; CMPS MM

Function: RR − MM

Description: The 16-bit memory operand is subtracted from the specified register and the result is discarded. Only the condition code bits are affected; neither the memory operand nor the register is affected.

Condition codes:

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

Addressing modes:

immediate
extended
direct
indexed

Example: CMPX >$3B33

before:	*after:*
X:$5410	X:$5410
CC:$23	CC:$24
$3B33:$54	3B33:$54
$3B34:$10	3B34:$10

COM *Complement*

Mnemonics: COMA; COMB; COM M

Function: ACCX ← $\overline{\text{ACCX}}$ (or) M ← $\overline{\text{M}}$

Description: The operand byte is replaced by its logical or one's complement.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	1

Addressing modes:
inherent
extended
direct
indexed

Example: COM $E,Y

before:	*after:*
Y:$03F2	Y:$03F2
CC:$04	CC:$01
$0400:$9B	$0400:$64

CWAI — *Clear CC bits and Wait for Interrupt*

Mnemonic: CWAI #N

Function:

```
CC ← CC∧N
E ← 1
S ← S − 1 ; (S) ← PC-low
S ← S − 1 ; (S) ← PC-high
S ← S − 1 ; (S) ← U-low
S ← S − 1 ; (S) ← U-high
S ← S − 1 ; (S) ← Y-low
S ← S − 1 ; (S) ← Y-high
S ← S − 1 ; (S) ← X-low
S ← S − 1 ; (S) ← X-high
S ← S − 1 ; (S) ← DP
S ← S − 1 ; (S) ← B
S ← S − 1 ; (S) ← A
S ← S − 1 ; (S) ← CC
```

Description: The immediate byte operand is logically ANDed with the condition code register. This action may clear specific bits, e.g., the interrupt masks. The E bit is set next. Then the entire processor state is saved on the hardware stack, and the processor waits for an interrupt. An RTI instruction will restore the entire processor state upon finding the E bit set in the recovered condition code register.

Condition codes:

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

(Changed according to operand)

Addressing mode: immediate

DAA—*Decimal Addition Adjust*

Mnemonic: DAA

Function: A ← A + Correction
Correction:
Least Significant Nibble:
  6: if H=1, or if LSN>9
  0: otherwise
Most Significant Nibble:
  6: if C=1, or if MSN>9, or if MSN>8 and LSN>9
  0: otherwise

Description: The appropriate correction factor is computed based on the values of the most significant nibble of A (MSNA), the least significant nibble of A (LSNA), and the condition code bits. It is then added to A. This instruction may be used after the addition of two BCD numbers to assure a proper BCD result. The carry bit generated by this instruction is the correct carry of the BCD addition.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	?	x

Addressing mode: inherent

Example: DAA

before:	*after:*
A:$7F	A:$85
CC:$00	CC:$08

DEC *Decrement*

Mnemonics: DECA; DECB; DEC M

Function: ACCX ← ACCX − 1 (or) M ← M − 1

Description: One is subtracted from the specified operand. Note that the carry bit is not affected.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	x	

↑ set only if original operand was $80, cleared otherwise.

Addressing modes:

inherent
extended
direct
indexed

Example: DECA

before:	*after:*
A:$32	A:$31
CC:$35	CC:$31

EOR — *Exclusive OR*

Mnemonics: EORA M; EORB M

Function: ACCX ← ACCX (XOR) M

Description: The memory operand is logically Exclusive ORed into the specified accumulator.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing modes:

immediate
extended
direct
indexed

Example: EORA 8,Y

before:	*after:*
Y:$32F0	Y:$32F0
A:$F2	A:$6A
CC:$03	CC:$01
$32F8:$98	$32F8:$98

EXG — *Exchange Registers*

Mnemonic: EXG R1,R2

Function: R1 ←→ R2

Description: The registers'values specified by the postbyte of the instruction are exchanged. The low and high nibbles of the postbyte specify the registers to be exchanged in the following way:

0 = D
1 = X
2 = Y
3 = U
4 = S
5 = PC

8 = A
9 = B
A = CC
B = DP
6, 7, C, D, E, F = undefined

Only registers of like size may be exchanged.

Condition codes:

E	F	H	I	N	Z	V	C

(No change unless one register is CC)

Addressing mode: immediate

Example: EXG A,DP

before:	*after:*
A:$42	A:$00
DP:$00	DP:$42

INC *Increment*

Mnemonics: INCA; INCB; INC M

Function: ACCX ← ACCX + 1 (or) M ← M + 1

Description: One is added to the operand. Note that the carry bit is not affected.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	x	

set if original operand was 7F

Addressing modes: inherent
extended
direct
indexed

Example: INCA

before:	*after:*
A:$35	A:$36
CC:$00	CC:$00

JMP — *Jump*

Mnemonic: JMP M

Function: PC ← ADDRM

Description: The value of the memory operand is transferred to the PC, and program execution continues at that address.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing modes:

extended
direct
indexed

Example: JMP ,X

before:	*after:*
X:$B290	X:$B290
PC:$0341	PC:$B290

JSR *Jump to Subroutine*

Mnemonic: JSR M

Function:

S ← S − 1 ; (S) ← PC-low
S ← S − 1 ; (S) ← PC-high
PC ← ADDRM

Description: The PC is pushed onto the hardware stack. The value of the memory operand is transferred to the PC and execution continues at that point. An RTS (return from subroutine) instruction will return control to the instruction following the JSR instruction.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing modes:

extended
direct
indexed

Example: JSR $320D

before:	*after:*
S:$03F2	S:$03F0
PC:$10CB	PC:$320D
$03F0:$03	03F0:$10
$03F1:$4B	03F1:$CB

LD (8-bit) — *Load Register From Memory*

Mnemonics: LDA M; LDB M

Function: ACCX ← M

Description: The memory operand is loaded into the specified register.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing modes:

immediate
extended
direct
indexed

Example: LDB >$EE01

before:	*after:*
B:$05	B:$F2
CC:$13	CC:$19
$EE01:$F2	$EE01:$F2

LD (16-bit) — *Load Register From Memory*

Mnemonics: LDD MM; LDX MM; LDY MM; LDS MM; LDU MM

Function: RR ← MM

Description: The 16-bit memory operand is loaded into the specified register.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing modes:
immediate
extended
direct
indexed

Example: LDD #$14A2

before:	*after:*
D:$0330	D:$14A2
CC:$54	C:$50

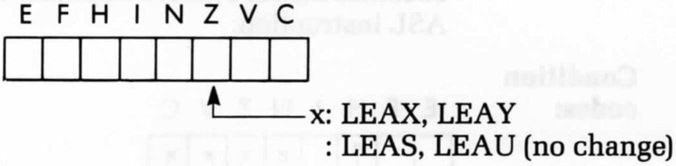
LEA — *Load Effective Address*

Mnemonics: LEAX M; LEAY M; LEAS M; LEAU M

Function: RR ← ADDRM

Description: The specified register is loaded with the address of the memory operand. The only addressing mode allowed is indexed.

Condition codes:



Addressing mode: indexed

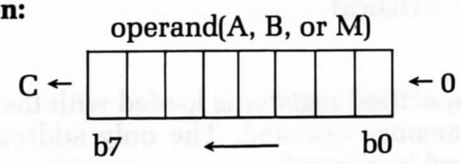
Example: LEAU $A,U

before:	*after:*
U:$0455	U:$045F

LSL *Logical Shift Left*

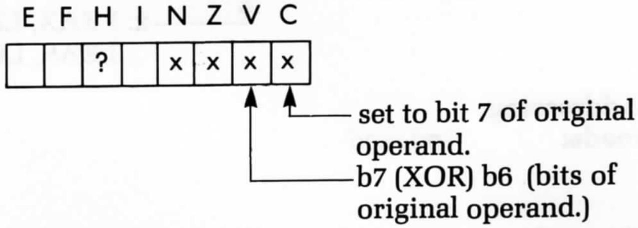
Mnemonics: LSLA; LSLB; LSL M

Function:



Description: All of the bits in the operand are shifted left by one position. Bit 7 is transferred to the carry bit; bit 0 becomes a zero. This is a duplicate mnemonic for the ASL instruction.

Condition codes:



set to bit 7 of original operand.

b7 (XOR) b6 (bits of original operand.)

Addressing modes: inherent
extended
direct
indexed

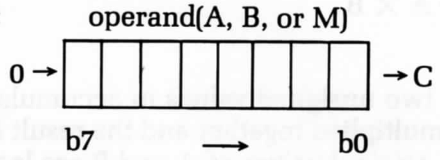
Example: LSL [$0310]

before:	*after:*
CC:$00	CC:$03
$0310:$4B	$0310:$4B
$0311:$28	$0311:$28
$4B28:$B8	$4B28:$70

LSR *Logical Shift Right*

Mnemonics: LSRA; LSRB; LSR M

Function:



Description: All of the bits in the operand are shifted right by one position. Bit 0 is transferred to the carry bit; bit 7 becomes a zero.

Condition codes:

E	F	H	I	N	Z	V	C
				0	x		x

C: set to bit 0 of original operand

Addressing modes:

inherent
extended
direct
indexed

Example: LSRA

before:	*after:*
A:$3E	A:$1F
CC:$0F	CC:$00

MUL —*Multiply*

Mnemonic: MUL

Function: D ← A × B

Description: The two unsigned values in accumulators A and B are multiplied together and the result is placed in D (i.e., original values of A and B are lost, and A contains the most significant byte of A × B.)

Condition codes:

E	F	H	I	N	Z	V	C
					x		x

set only if b7 of B in result is set.

Addressing mode: inherent

Example: MUL

before:	*after:*
A:$0C	A:$04
B:$64	B:$B0
	(D:$04B0)

NEG —*Negate*

Mnemonics: NEGA; NEGB; NEG M

Function: ACCX ← 0 − ACCX (or) M ← 0 − M

Description: Replaces operand with its two's complement.

Condition codes:

E	F	H	I	N	Z	V	C
		?		x	x	x	x

V: set only if original operand was $80

Addressing modes:

inherent
extended
direct
indexed

Example: NEG >$4002

before:	*after:*
CC:34	CC:10
4002: F3	4002: 0D

NOP — *No Operation*

Mnemonic: NOP

Function: Does nothing

Description: No registers or memory locations are affected. Uses time and program memory space.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: inherent

OR — *OR Memory Into Register*

Mnemonics: ORA M; ORB M

Function: ACCX ← ACCX∨M

Description: The specified accumulator and memory operand are logically ORed, and the result is stored in the accumulator.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing modes:

immediate
extended
direct
indexed

Example: ORA #$0F

before:	*after:*
A:$DA	A:$DF
CC:$43	CC:$49

OR — *OR Immediate Data into Condition Code Register*

Mnemonic: ORCC #N

Function: $CC \leftarrow CC \vee N$

Description: The condition code register and the immediate memory byte are logically ORed, and the result is stored in the condition code register. This instruction may be used to set specific flags.

Condition codes:

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

(Changed according to operand)

Addressing mode: immediate

Example: ORCC #$50

before:	*after:*
CC:$13	CC:$53

PSHS -*Push Registers onto Hardware Stack*-

Mnemonics: PSHS register-list; PSHS #N

Function:

```
If b7 of postbyte set:  S ← S − 1 ;  (S) ← PC-low
                        S ← S − 1 ;  (S) ← PC-high
If b6 of postbyte set:  S ← S − 1 ;  (S) ← U-low
                        S ← S − 1 ;  (S) ← U-high
If b5 of postbyte set:  S ← S − 1 ;  (S) ← Y-low
                        S ← S − 1 ;  (S) ← Y-high
If b4 of postbyte set:  S ← S − 1 ;  (S) ← X-low
                        S ← S − 1 ;  (S) ← X-high
If b3 of postbyte set:  S ← S − 1 ;  (S) ← DP
If b2 of postbyte set:  S ← S − 1 ;  (S) ← B
If b1 of postbyte set:  S ← S − 1 ;  (S) ← A
If b0 of postbyte set:  S ← S − 1 ;  (S) ← CC
```

Description: Any combination of registers, including no registers, is pushed onto the hardware stack. The postbyte, n, is determined by the register list. The postbyte has the following structure:

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

push order ⟶

One register may be pushed with an autodecrement store. Example: STY ,−−S pushes Y, but also changes condition code bits.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing: mode: immediate

PSHU — *Push Registers onto User Stack*

Mnemonics: PSHU register-list; PSHU #N

Function:

If b7 of postbyte set: U ← U − 1 ; (U) ← PC-low
U ← U − 1 ; (U) ← PC-high
If b6 of postbyte set: U ← U − 1 ; (U) ← S-low
U ← U − 1 ; (U) ← S-high
If b5 of postbyte set: U ← U − 1 ; (U) ← Y-low
U ← U − 1 ; (U) ← Y-high
If b4 of postbyte set: U ← U − 1 ; (U) ← X-low
U ← U − 1 ; (U) ← X-high
If b3 of postbyte set: U ← U − 1 ; (U) ← DP
If b2 of postbyte set: U ← U − 1 ; (U) ← B
If b1 of postbyte set: U ← U − 1 ; (U) ← A
If b0 of postbyte set: U ← U − 1 ; (U) ← CC

Description: Any combination of registers, including no registers, is pushed onto the user stack. The postbyte, n, is determined by the register list. The postbyte has the following structure:

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

push order ⟶

One register may be pushed with an autodecrement store. Example: STY ,−−U pushes Y, but also changes condition code bits.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing mode: immediate

PULS *–Pull Registers from Hardware Stack–*

Mnemonics: PULS register-list; PULS #N

Function:
If b0 of postbyte set: CC ← (S); S ← S+1
If b1 of postbyte set: A ← (S); S ← S+1
If b2 of postbyte set: B ← (S); S ← S+1
If b3 of postbyte set: DP ← (S); S ← S+1
If b4 of postbyte set: X-high ← (S); S ← S+1
X-low ← (S); S ← S+1
If b5 of postbyte set: Y-high ← (S); S ← S+1
Y-low ← (S); S ← S+1
If b6 of postbyte set: U-high ← (S); S ← S+1
U-low ← (S); S ← S+1
If b7 of postbyte set: PC-high ← (S); S ← S+1
PC-low ← (S); S ← S+1

Description: Any combination of registers, including no register, is pulled from the hardware stack. The postbyte, n, is determined by the register list. The postbyte has the following structure:

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

← pull order

One register may be pulled with an autoincrement load. Example: LDY ,S++ pulls Y, but also changes condition code bits.

Condition codes:

E	F	H	I	N	Z	V	C

(no change—unless CC pulled)

Addressing: mode: immediate

PULU — *Pull Registers from User Stack*

Mnemonics: PULU register-list; PULU #N

Function:

If b0 of postbyte set:	CC	← (S) ;	S ← S+1
If b1 of postbyte set:	A	← (S) ;	S ← S+1
If b2 of postbyte set:	B	← (S) ;	S ← S+1
If b3 of postbyte set:	DP	← (S) ;	S ← S+1
If b4 of postbyte set:	X-high	← (S) ;	S ← S+1
	X-low	← (S) ;	S ← S+1
If b5 of postbyte set:	Y-high	← (S) ;	S ← S+1
	Y-low	← (S) ;	S ← S+1
If b6 of postbyte set:	S-high	← (S) ;	S ← S+1
	S-low	← (S) ;	S ← S+1
If b7 of postbyte set:	PC-high	← (S) ;	S ← S+1
	PC-low	← (S) ;	S ← S+1

Description: Any combination of registers, including no register, is pulled from the user stack. The postbyte, n, is determined by the register list. The postbyte has the following structure:

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

← pull order

One register may be pulled with an autoincrement load. Example: LDY ,U++ pulls Y, but also changes condition code bits.

Condition codes:

E	F	H	I	N	Z	V	C

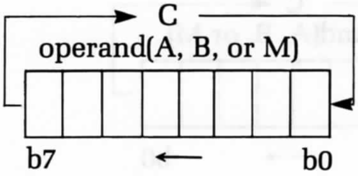
(no change—unless CC pulled)

Addressing: mode: immediate

ROL *Rotate Left*

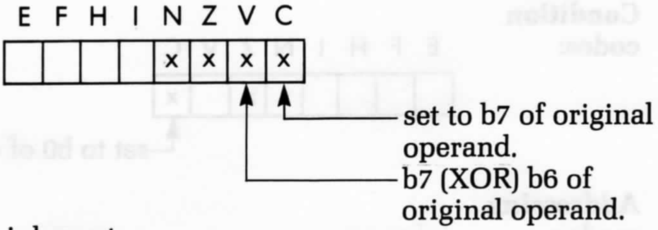
Mnemonics: ROLA; ROLB; ROL M

Function:



Description: All of the bits in the operand are rotated left by one position through the carry bit (9-bit rotation); that is, b7 is transferred to the carry bit and the original value of the carry bit is transferred to b0.

Condition codes:


E F H I N Z V C
x x x x
set to b7 of original operand.
b7 (XOR) b6 of original operand.

Addressing: modes: inherent
extended
direct
indexed

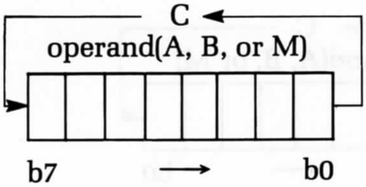
Example: ROLB

before:	*after:*
B:$89	B:$13
CC:$09	CC:$03

ROR Rotate Right

Mnemonics: RORA; RORB; ROR M

Function:



Description: All of the bits of the operand are rotated right by one position through the carry bit (9-bit rotation); that is, b0 is transferred to the carry bit and the original value of the carry bit is transferred to b7.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x		x

set to b0 of original operand

Addressing modes:
inherent
extended
direct
indexed

Example: RORB

before:	after:
B:$89	B:$C4
CC:$09	CC:$09

RTI —*Return from Interrupt*

Mnemonic: RTI

Function:

```
CC ← (S) ;  S ← S+1
If E=1 then:  A       ← (S) ;  S ← S+1
              B       ← (S) ;  S ← S+1
              DP      ← (S) ;  S ← S+1
              X-high  ← (S) ;  S ← S+1
              X-low   ← (S) ;  S ← S+1
              Y-high  ← (S) ;  S ← S+1
              Y-low   ← (S) ;  S ← S+1
              U-high  ← (S) ;  S ← S+1
              U-low   ← (S) ;  S ← S+1
              PC-high ← (S) ;  S ← S+1
              PC-low  ← (S) ;  S ← S+1
```

Description: The condition code register is pulled from the hardware stack. If the E bit is set, then the entire machine state is pulled from the stack, otherwise, only the PC is pulled from the stack. This instruction reverses the effects of an interrupt and should be placed at the end of an interrupt routine.

Condition codes:

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

(Pulled from stack)

Addressing: mode: inherent

RTS *Return from Subroutine*

Mnemonic: RTS

Function:
PC-high ←(S);S←S+1
PC-low ←(S);S←S+1

Description: The PC is pulled from the hardware stack. This instruction reverses the effects of the BSR and JSR instructions and should be placed at the end of a subroutine.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing: mode: inherent

SBC *Subtract with Borrow*

Mnemonics: SBCA M; SBCB M

Function: ACCX ← ACCX − M − C

Description: The memory operand and the C bit are subtracted from the specified accumulator. The resulting C bit is a borrow and is set to the complement of the carry of the internal addition.

Condition codes:

E	F	H	I	N	Z	V	C
		?		x	x	x	x

Addressing modes:
immediate
extended
direct
indexed

Example: SBCB <$3

before:	*after:*
DP:$45	DP:$45
B:$35	B:$31
CC:$01	CC:$20
$4503:$03	$4503:$03

SEX *Sign Extend*

Mnemonic: SEX

Function: If b7 of B = 1 then: A ← FF
else: A ← 0

Description: The 8-bit two's complement value in accumulator B is sign extended to a 16-bit value in accumulator D. The original value of A is lost.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x		

Addressing: mode: inherent

Example: SEX

before:	*after:*
B:$E6	D:$FFE6

ST (8-bit) *Store Register into Memory*

Mnemonics: STA M; STB M

Function: M ← ACCX

Description: The contents of the specified accumulator are stored at the memory operand.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing: modes:

extended
direct
indexed

Example: STB [$F,X]

before:	*after:*
B:$E5	B:$E5
X:$556A	X:$556A
$5579:$03	$5579:$03
$557A:$BB	$557A:$BB
$03BB:$02	$03BB:$E5

ST (16-bit)—*Store Register into Memory*

Mnemonics: STD MM; STX MM; STY MM; STS MM; STU MM

Function: MM ← RR

Description: The contents of the specified register are stored at the 16-bit memory operand.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing: modes:

extended
direct
indexed

Example: STX >$12B0

before:	*after:*
X:$660C	X:$660C
$12B0:$37	$12B0:$66
$12B1:$BF	$12B1:$0C

SUB (8-bit) — *Subtract Memory from Register*

Mnemonics: SUBA M; SUBB M

Function: ACCX ← ACCX − M

Description: The memory operand is subtracted from the specified accumulator. The C bit is a borrow and is set to the complement of the carry of the internal binary addition.

Condition codes:

E	F	H	I	N	Z	V	C
		?		x	x	x	x

Addressing modes:

immediate
extended
direct
indexed

Example: SUBB ,Y

before:	*after:*
B:$03	B:$E2
Y:$0021	Y:$0021
CC:$44	CC:$69
$0021:$21	$0021:$21

SUB (16-bit) — *Subtract Memory from Register*

Mnemonic: SUBD MM

Function: D ← D − MM

Description: The 16-bit memory operand is subtracted from the D accumulator. The carry bit represents a borrow and is set to the complement of the internal binary addition carry.

Condition codes:

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

Addressing modes:
immediate
extended
direct
indexed

Example: SUBD #$020F

before:	*after:*
D:$6B90	D:$6981
CC:$59	CC:$50

SWI *Software Interrupt*

Mnemonic: SWI

Function:

E ← 1
S ← S − 1 ; (S) ← PC-low
S ← S − 1 ; (S) ← PC-high
S ← S − 1 ; (S) ← U-low
S ← S − 1 ; (S) ← U-high
S ← S − 1 ; (S) ← Y-low
S ← S − 1 ; (S) ← Y-high
S ← S − 1 ; (S) ← X-low
S ← S − 1 ; (S) ← X-high
S ← S − 1 ; (S) ← DP
S ← S − 1 ; (S) ← B
S ← S − 1 ; (S) ← A
S ← S − 1 ; (S) ← CC
I ← 1 ; F ← 1
PC-high ← FFFA ; PC-low ← FFFB

Description: The entire machine state is pushed onto the hardware stack. Program control is transferred via the software interrupt 1 vector. Fast and normal interrupts are disabled.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing: mode: inherent

SWI2 —*Software Interrupt 2*

Mnemonic: SWI2

Function:

```
E ← 1
S ← S − 1 ; (S) ← PC-low
S ← S − 1 ; (S) ← PC-high
S ← S − 1 ; (S) ← U-low
S ← S − 1 ; (S) ← U-high
S ← S − 1 ; (S) ← Y-low
S ← S − 1 ; (S) ← Y-high
S ← S − 1 ; (S) ← X-low
S ← S − 1 ; (S) ← X-high
S ← S − 1 ; (S) ← DP
S ← S − 1 ;  (S) ← B
S ← S − 1 ;  (S) ← A
S ← S − 1 ; (S) ← CC
PC-high ← FFF4 ;  PC-low ← FFF5
```

Description: The entire machine state is pushed onto the hardware stack. Program control is transferred through the software interrupt 2 vector. The F and I interrupt masks are not affected.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing: mode: inherent

SWI3 — *Software Interrupt 3*

Mnemonic: SWI3

Function:
E ← 1
S ← S − 1 ; (S) − PC-low
S ← S − 1 ; (S) ← PC-high
S ← S − 1 ; (S) ← U-low
S ← S − 1 ; (S) ← U-high
S ← S − 1 ; (S) ← Y-low
S ← S − 1 ; (S) ← Y-high
S ← S − 1 ; (S) ← X-low
S ← S − 1 ; (S) ← X-high
S ← S − 1 ; (S) ← DP
S ← S − 1 ; (S) ← B
S ← S − 1 ; (S) ← A
S ← S − 1 ; (S) ← CC
PC-high ← FFF2 ; PC-low ← FFF3

Description: The entire machine state is pushed onto the hardware stack. Program control is transferred through software interrupt vector 3. The F and I interrupt masks are not affected.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing: mode: inherent

SYNC — *Synchronize to External Event*

Mnemonic: SYNC

Function: Halt processor.

Description: The processor halts and waits for an interrupt to occur. If the interrupt is masked (disabled) or is shorter than 3 cycles, then the processor continues execution with the instruction following the SYNC instruction. If the interrupt is enabled and lasts more than 3 cycles, then a normal interrupt sequence begins. The return address pushed onto the stack is that of the instruction following the SYNC instruction. This instruction can be used to synchronize the processor with high speed, critical events, such as reading data from a disk drive.

Condition codes:

E	F	H	I	N	Z	V	C

(no change)

Addressing: mode: inherent

TFR *Transfer Register to Register*

Mnemonic: TRF R1,R2

Function: R2 ← R1

Description: The contents of the register specified by the high nibble of the postbyte are transferred to the register specified by the low nibble of the postbyte. The nibbles of the postbyte specify the registers in the following way:

0 = D	8 = A
1 = X	9 = B
2 = Y	A = CC
3 = U	B = DP
4 = S	6, 7, C, D, E, F: undefined
5 = PC	

Only registers of like sizes may be paired up.

Condition codes:

E	F	H	I	N	Z	V	C

(no change—unless R2 is CC)

Addressing: mode: immediate

TST — *Test*

Mnemonics: TSTA; TSTB; TST M

Function: ACCX − 0 (or) M − 0

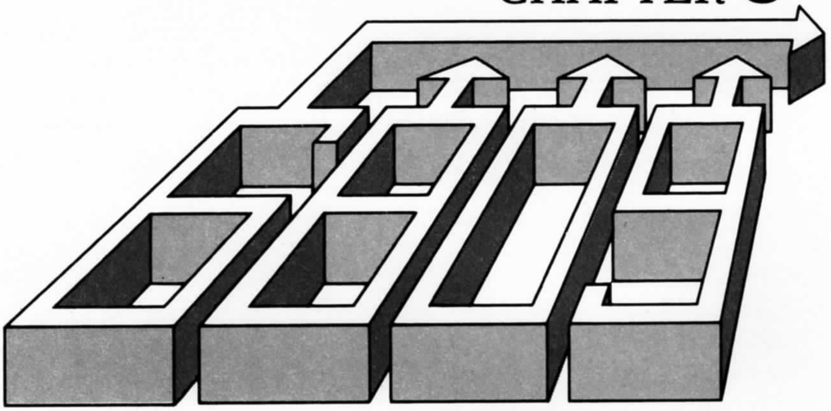
Description: The Z and N bits are affected according to the value of the specified operand. The V bit is cleared.

Condition codes:

E	F	H	I	N	Z	V	C
				x	x	0	

Addressing modes:
inherent
extended
direct
indexed

CHAPTER 5



ADDRESSING TECHNIQUES

IN THIS CHAPTER, we will begin by discussing the general theory of addressing and examining the various techniques used for accessing data. We will then go on to examine the most important aspect of the 6809's architecture—the area where its special power is most apparent—the extensive 6809 addressing capabilities. The most uniquely important of these are indexed and relative addressing.

The special registers and modes provided for indexed addressing make the 6809 an excellent machine for writing efficient routines to handle complex data structures. The 6809's relative addressing modes make it possible to write position independent code (especially important in ROM-based applications)—a task which would be impossible on any other 8-bit microprocessor.

Although complex data accessing methods are not necessary in the beginning stages of programming, it is crucial to understand the addressing modes in order to realize the full power of the 6809. Once you have mastered the addressing techniques that we present in this chapter, it will then be a straightforward matter to write efficient data handling routines.

POSSIBLE ADDRESSING MODES

Addressing refers to the specification within an instruction, of the location of the operand on which the instruction will operate. We begin by examining the six basic addressing modes (shown in Figure 5.1).

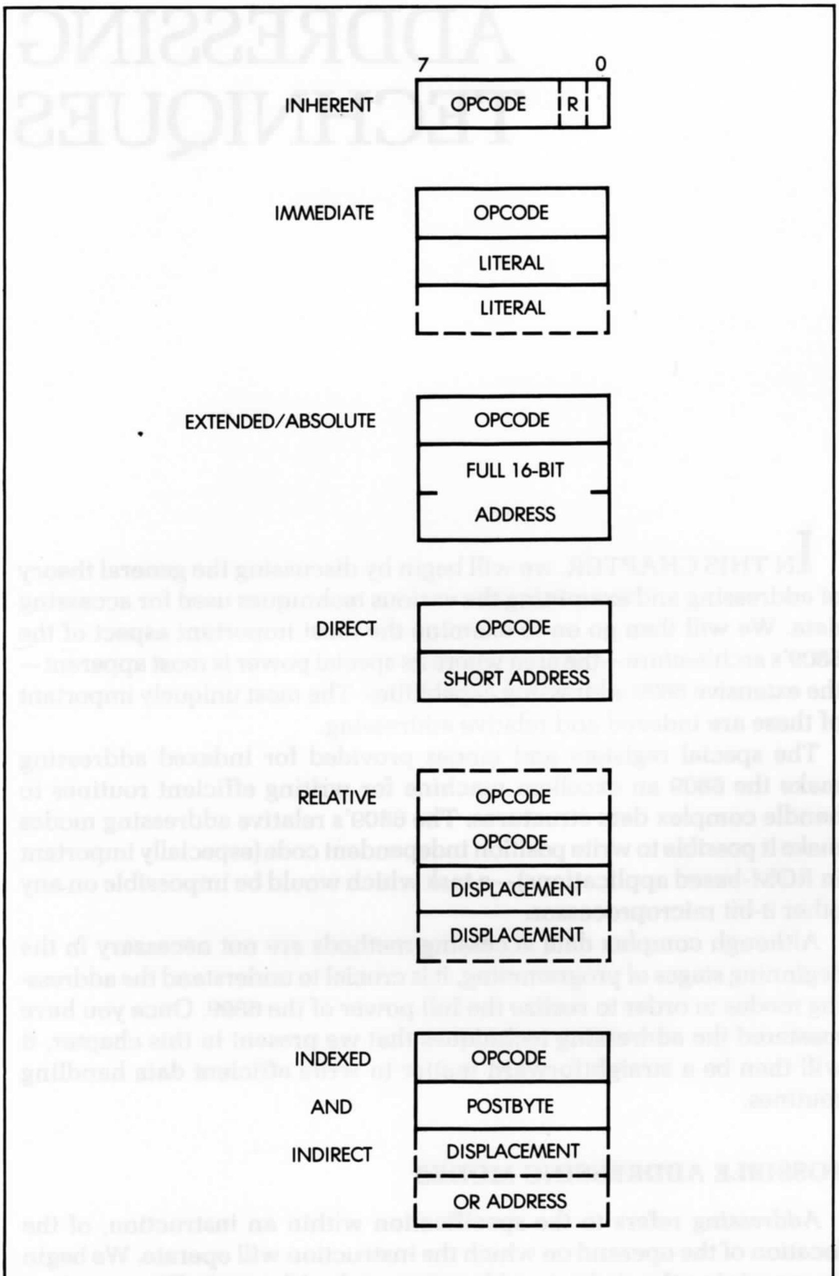


Figure 5.1: Basic Addressing Modes

Inherent (Implied or Register) Addressing

Instructions that operate exclusively on registers normally use *inherent addressing* (as illustrated in Figure 5.1). An inherent instruction derives its name from the fact that it does not specifically contain the address of the operand on which it operates; instead, its opcode specifies one or more registers. Since internal registers are usually few in number (commonly eight), only a small number of bits are needed to specify a particular register in the opcode.

An example of an inherent addressing instruction is:

```
        DECA
```

This instruction specifies: "decrement the contents of A by 1."

Immediate Addressing

In the *immediate addressing* mode, an 8- or 16-bit literal (a constant) follows the 8-bit opcode (see Figure 5.1). Since the microprocessor is equipped with 16-bit registers, it may be necessary to load 8- or 16-bit literals. An example of an immediate instruction is:

```
        ADDB    #$5
```

The second word of this instruction contains the literal 5, which is added to accumulator B.

Another form of immediate addressing uses a byte (called the postbyte) following the opcode, to specify the registers to be used in the instruction. Here is an example of an immediate instruction using the postbyte:

```
        TFR     A,B
```

The second word of this instruction contains the codes for registers A and B, because A is transferred to B.

Extended (or Absolute) Addressing

In *extended addressing*, the 16-bit address of the operand follows the opcode. Extended addressing, therefore, requires three-byte instructions. Here is an example using the extended addressing mode:

```
        STA     $1234
```

This instruction specifies that the contents of the accumulator are to be stored at memory location 1234 hexadecimal. Extended addressing is also called *absolute addressing*, because an absolute memory address is specified.

A disadvantage of extended addressing is that it requires a three-byte instruction. To improve the efficiency of the microprocessor, there may be another addressing mode available, direct addressing, which requires that only one word be used for the address.

Direct Addressing

In *direct addressing*, the opcode is followed by an 8-bit address (see Figure 5.1). The advantage of this approach is that it requires only two bytes, instead of three, for extended addressing. A disadvantage is that on *most* microprocessors it limits all addressing within this mode to addresses 0 to 255. (*Note:* the 6809 does not have this limitation.) When addresses 0 to 255 are used, this type of addressing is also known as *short* or *0-page addressing*.

Relative Addressing

You use *relative addressing* with branch instructions. If the state of the condition codes satisfies the test made by the branch instruction, then the branch instruction loads the PC with a new address. The byte following the opcode, called the *displacement*, is added to the PC to form the new PC, to which the instruction branches. Figure 5.1 shows the structure of the relative addressing mode.

Since the displacement is a positive or negative number, a relative branch instruction allows a branch forward of 127 bytes or backward of 128 bytes (usually +129 or −126, since the PC will have been incremented by 2). The branch instructions are used in program loops. Because most loops are short, relative branching with a one byte displacement is the most common. Relative branching usually results in significantly improved performance for short routines.

If you need a larger branch displacement, you can use the long branch instruction with a 16-bit displacement. This instruction also has an extra opcode byte, so that it is four bytes long (see Figure 5.1). The long branch can branch to any address in the memory because the displacement ranges from −32768 to 32767. Since long branch instructions take longer to execute than the simple branch instructions, you normally use them only when the shorter branch will not work. Relative addressing provides improved speed performance with branch instructions. If a program uses relative addressing, it can be easily moved to different areas of memory. In addition, if you do not use absolute addresses, then it is possible to relocate the program to other areas of memory. The jump instruction, JMP, allows the use of absolute addressing. The absolute addressing mode should generally be avoided in favor of relative addressing.

Indexed Addressing

You use *indexed addressing* to access, in succession, the elements of a block or table. This addressing mode appears in examples given later in this chapter. With indexed addressing, the instruction specifies both an index register and a base address. The contents of the register and base address are added to provide the final address. In this way, the address could be the beginning of a table in memory. The index register would then be used to efficiently access all the elements of a table successively. However, there must be a way to increment or decrement the index register.

Pre-Indexing and Post-Indexing

There are two modes of indexing: *pre-indexing* and *post-indexing*. Pre-indexing is the usual indexing mode in which the final address is the sum of a displacement or address, plus the contents of the index register. Figure 5.2 illustrates this approach (assuming an 8-bit displacement field and a 16-bit index register).

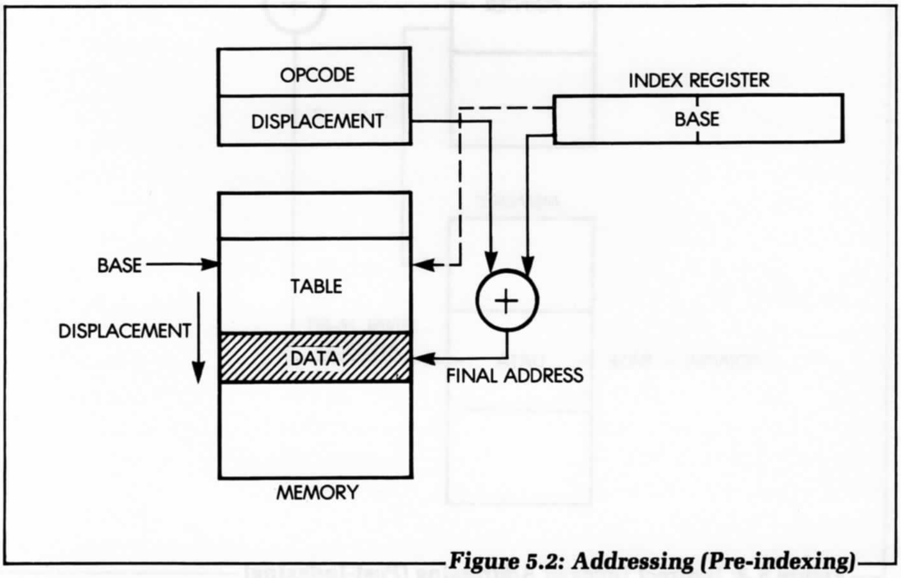


Figure 5.2: Addressing (Pre-indexing)

On the other hand, post-indexing treats the contents of the displacement field like the *address* of the actual displacement, rather than like the displacement itself. In post-indexing, the final address is the sum of the contents of the index register, plus the contents of the memory word *designated by the displacement field* (see Figure 5.3). This feature, in fact, utilizes a combination of indirect addressing and pre-indexing. Let's now define indirect addressing.

Indirect Addressing

At times, it is necessary for two subroutines to exchange a large quantity of data stored in the memory. More generally, several programs or subroutines may need to access a common block of information. To preserve the generality of the program, it is desirable not to keep such a block at a fixed memory location. In particular, the size of the block may grow or shrink dynamically, and thus, it may have to reside in various

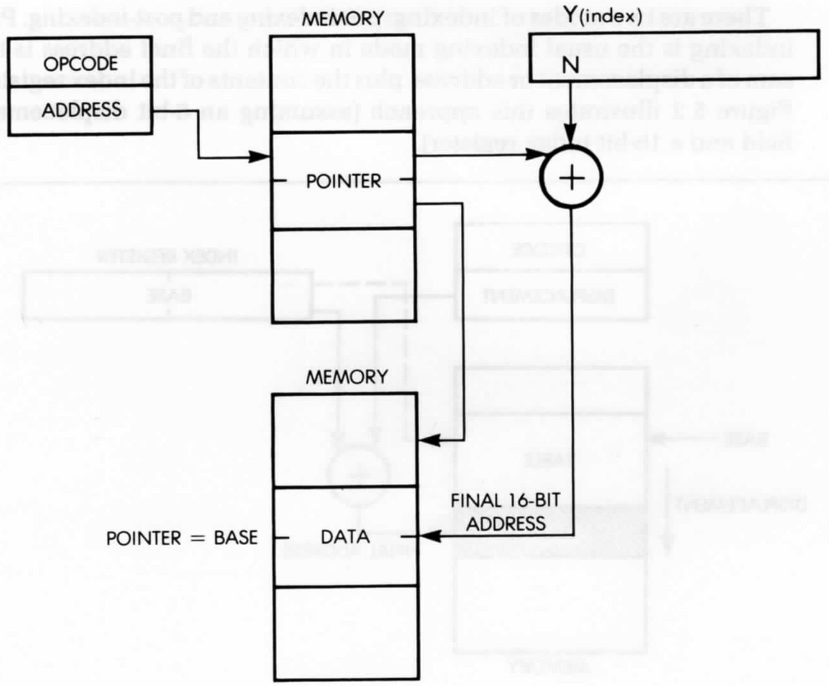


Figure 5.3: Indirect Indexed Addressing (Post-Indexing)

areas of the memory, depending on its size. It would, therefore, be impractical to try to access this block using absolute addresses—that is, without rewriting the program every time.

The solution to this problem then is to deposit the starting address of the block at a fixed memory location. *Indirect addressing,* therefore, normally uses an opcode (16 bits in the case of the 6809), followed by a 16-bit address. This address is used to retrieve a 16-bit word from the memory. This is used as the address of the operand. Figure 5.4 illustrates the structure of an instruction using indirect addressing, where the two bytes at the specified address A1 contain A2. A2 is then interpreted as the actual address of the data to be accessed.

Indirect addressing is particularly useful any time pointers are used. Various areas of the program can then refer to these pointers to conveniently and elegantly access a word or block of data. Another form of indirect addressing, *indexed indirect addressing,* uses an index register, rather than a memory location, to contain the address of the address of the desired data.

Combinations of Modes

It is possible to combine addressing modes. In particular, it is possible in a completely general addressing scheme to use many levels of indirection. For example, in Figure 5.4 the address A2 could be interpreted as an indirect address again, and so on.

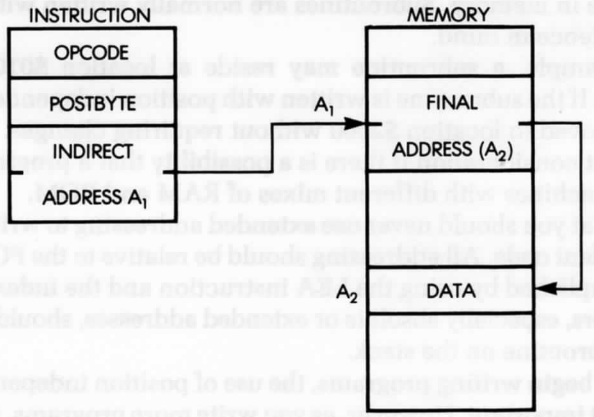


Figure 5.4: Indirect Addressing

You can also combine indexed addressing with indirect access. This allows efficient access to word n of a block of data, provided you know the location of the pointer to the starting address (see Figure 5.2).

Mode Summary

We are now familiar with all the usual addressing modes that can be provided in a system. Most microprocessor systems, because of the limitation of the MPU (i.e., that it must be realized within a single chip), do not provide all possible modes, but only a small subset of them. The 6809 provides a good subset of possibilities. Let's examine them.

6809 ADDRESSING MODES

The 6809 addressing modes are an important feature of the 6809 processor. They can be used with most instructions to offer great power and flexibility. Even though there are fewer instructions on the 6809 than there were on its predecessor, the 6800, the new addressing modes make the 6809 a more capable machine. To make good use of the 6809 processor and to write better programs, it is important to learn to use all the addressing modes.

Position Independent Code (PIC)

The powerful addressing modes on the 6809 make it possible to write position independent code. A position independent program can be run anywhere in memory. Subroutines are normally written with position independence in mind.

For example, a subroutine may reside at location $0100 on one machine. If the subroutine is written with position independent code, it can be moved to location $2000 without requiring changes. This is an important consideration if there is a possibility that a program may be run on machines with different mixes of RAM and ROM.

Note that you should never use extended addressing to write position independent code. All addressing should be relative to the PC. This can be accomplished by using the LEA instruction and the index registers. Parameters, especially absolute or extended addresses, should be passed to the subroutine on the stack.

As you begin writing programs, the use of position independent code is not that important. However, as you write more programs, it becomes more important, as it becomes necessary to combine different parts of a program, without modifying or re-assembling the program.

Inherent Addressing (6809)

On the 6809, *inherent addressing* is primarily used by single-byte instructions which operate on internal registers. Many of these instructions require only two cycles to execute. Instructions using inherent addressing are:

ABX
ASLA
ASLB
ASRA
ASRB
CLRA
CLRB
COMA
COMB
CWAI
DAA
DECA

DECB
EXG
INCA
INCB
LSLA
LSLB
LSRA
LSRB
MUL
NEGA
NEGB
NOP

ROLA
ROLB
RORA
RORB
RTI
RTS
SEX
SWI
SYNC
TFR
TSTA
TSTB

Some instructions, such as MUL, require more than two cycles to execute. Other instructions, such as TFR and EXG, require more than one byte. Inherent addressing is also called *register addressing*.

Immediate Addressing (6809)

Since the 6809 has both single-length (8-bit) and double-length (16-bit) registers, it provides two types of immediate addressing, with both 8-and 16-bit literals. Instructions are then either two or three bytes long.

Here are examples of instructions using the *immediate addressing* mode:

```
LDA     #n          (one byte)
LDX     #nn         (two bytes)
```

and

```
ADDA    #n          (one byte)
```

Extended (or Absolute) Addressing (6809)

By definition, *extended addressing* requires three bytes. The first is the opcode and the next two are the 16-bit address specifying the memory location (i.e., the absolute address).

Extended addressing always specifies a particular address, which does not change while the program executes. Thus, position independent code cannot be written when extended addressing is used. Input and output programs often use extended addressing. Examples of instructions using extended addressing are:

```
          LDA        >$0100
```

and

```
          JMP        >$1234
```

where the two hexadecimal numbers represent the 16-bit addresses of data or instructions.

Direct Addressing (6809)

On most microprocessors, *direct addressing*, if available, addresses only the first 256 bytes, 0 page, of memory (addresses 0 to 255). This is because only an 8-bit address is specified, allowing the instruction to use two bytes instead of three. On the 6809 it is possible to address any byte in memory by using direct addressing and manipulating the direct page (DP) register.

When direct addressing is used, the low byte of the address is the byte immediately following the opcode, and the high byte is the contents of the DP register. By changing the DP register appropriately, any page in memory may be addressed. When the DP register contains zero, the 6809 direct addressing mode operates in the same manner as other microprocessors.

Relative Addressing (6809)

By definition, relative addressing requires two bytes. The first is the "branch relative" opcode; the second specifies the displacement and its sign. A long branch requires an extra opcode byte to indicate a long branch, and a second byte for the displacement, thus, making a total of four bytes.

From a timing standpoint, this instruction should be examined with caution. Whether a test succeeds or fails, (i.e., whether or not there is a branch), all short branch instructions require three cycles. However, the long branch instruction requires five cycles when the test fails, and six when it succeeds and the branch is taken.

Caution must be exercised when computing the duration of the execution of a program segment. If you are not sure that the long branch will succeed, you must remember that sometimes the instruction *will require six cycles (if the condition is met) and sometimes five (if the condition is not met).* An average value is often used for the duration of a long branch.

This timing problem does not apply to the long branch always instruction, LBRA, as this instruction does not test any condition, and always lasts five cycles.

(*Note:* In order to differentiate the absolute jump instruction from the relative branch, the jump instruction is labeled JMP.)

Indexed Addressing (6809)

The *indexed addressing* mode is very powerful on the 6809 microprocessor. In all indexed addressing, one of the address registers (X, Y, U, S, and sometimes the PC) is used to calculate the effective address of the data used by the instruction. There are five different types of indexed addressing. The second byte or postbyte of an instruction using indexed addressing specifies the type of addressing mode, as well as the address register to be used. The structure of an indexed instruction appears in Figure 5.5.

Appendix F gives for each variation, the assembler form and number of cycles and bytes added to the basic values for indexed addressing.

Zero-Offset Indexed

In the *zero-offset indexed* mode, an address register contains the effective address of the data to be used by the instruction. Zero-offset indexed is the fastest indexed mode, because a displacement is not needed. The instruction is two bytes long. Examples of zero-offset indexed addressing are:

```
        LDD     0,Y
```

and

```
        LDB     ,U
```

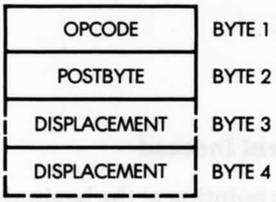


Figure 5.5: Indexed Addressing Needs Two or More Bytes

Constant Offset Indexed

The *constant offset indexed* mode uses a two's complement displacement and the contents of one of the address registers added together to form the effective address of the operand. The address register's initial content is unchanged by the addition. Three sizes of displacements are available:

1. 5-bit (−16 to +15)
2. 8-bit (−128 to +127)
3. 16-bit (−32768 to +32767)

The 5-bit displacement is included in the postbyte and is, therefore, most efficient in size and speed. The 8-bit displacement is contained in the byte following the postbyte. The two byte 16-bit displacement immediately follows the postbyte, high byte first. Examples of constant-offset indexing are:

```
LDA     33,X
LDY     −2,S
LDX     400,Y
```

Accumulator-Offset Indexed

The accumulator-offset indexed mode uses the two's complement value of one of the accumulators, A, B, or D. This value is added to one of the address registers to form the effective address of the operand. The contents of both the accumulator and the address register are unaffected by the addition. The postbyte specifies the accumulator to use, so no displacement bytes are needed. The accumulator offset mode is also advantageous in that the offset may be calculated while the program is running. Examples of instructions using this mode are:

```
LDA     B,Y
LDX     D,Y
LEAX    B,X
```

Auto Increment/Decrement Indexed

If an address register is pointing at the beginning (or end) of a table of data, the auto increment and decrement modes provide a method for efficiently accessing successive elements. In the auto increment mode, the address register is *first* used as the effective address to fetch the operand. Then the address register is incremented by one or two before the next instruction is fetched. This allows stepping through a table from low addresses to high addresses. The address register is incremented by one if 8-bit data is used, and by two if 16-bit data is used. A + *after* the register indicates auto increment mode. One + is used for 8-bit data; two +'s are used for 16-bit data. Here are examples:

```
LDA     ,X+
STD     ,Y++
```

In the second example, two +'s are necessary because a 16-bit word is stored in the D accumulator. It is up to the programmer to decide if single or double incrementing is needed. Auto decrement mode is the opposite of auto increment mode.

When the auto decrement mode is used, the address register is decremented by one or two, *before* it is used to fetch the operand. This is different from the auto increment mode, where the increment takes place *after* the operand is fetched. With auto decrement, a table may be accessed from high addresses to low addresses. A "−" before the name of the register indicates auto decrement mode. Here are examples:

```
LDB     ,-Y
LDX     ,--S
```

In the second example, −− means that S is decremented by two before it is used as the effective address of the operand.

The auto increment/decrement mode is useful for moving data and for creating software stacks. The pre-decrement, post-increment nature of these modes allow them to be used to create stacks with the X and Y registers, that behave in the same way as the stacks with the U and S registers.

Indexed Indirect

The indexed indirect mode is a combination of indirect and indexed addressing modes. All but two of the indexing modes may have a level of indirection added. In this mode, the effective address of the operand is contained in the memory location formed by the contents of an address register, plus any offset. Indirection is indicated by enclosing the operand specification in square brackets, [ ]. In the example below, the A accumulator is loaded indirectly, using an effective address calculated from the X register and an offset:

```
$0100    LDX    #$F000    LOAD X IMMEDIATE
$0103    LDA    [$10,X]   EA IS NOW $F010
$F010    $F1              $F150 IS NOW THE
$F011    $50              NEW EA

$F150    $AA
```

After execution, A contains the data $AA, and X contains $F000.

The two modes that cannot be used for indexed indirect addressing are auto increment/decrement by one, and a constant offset of 5 bits. In the first case, if an indirect auto increment or decrement mode is being used, the *addresses*, which are two bytes long, are pointed to by the indexed register. An increment or decrement of one is not sensible, because the index register would point to the low byte of the address, rather than the high byte.

The mode containing a 5-bit offset in the postbyte is not available, because it would have a postbyte identical to the other indexed indirect postbytes (see Appendix F). An offset of 5 bits or less is contained in the first displacement byte.

Extended Indirect (6809)

Extended indirect is the name of the indirect addressing mode on the 6809. The opcode is followed by a postbyte (which indicates extended indirect addressing) and two other bytes (which contain the address which points to the effective address of the data). The instruction is at least four bytes long when extended indirect addressing is used. In the example below, the data is contained in the address pointed to by the address in the instruction:

```
$0100    LDA    [$FFFE]    EA IS $FFFE FIRST
$E081    $55               THE DATA TO LOAD INTO A
$FFFE    $E0               EA IS NOW $E081
$FFFF    $81
```

After execution, A contains $55, but no other register is changed.

Program Counter Relative (6809)

The *program counter relative addressing* mode is a cross between constant-offset indexed, and relative addressing. When program counter relative mode is used, either an 8- or 16-bit two's complement offset is added to the current PC to create the effective address. This effective address is used to fetch the data. The PC is not changed by the addition, as is the case in normal relative addressing. A postbyte after the opcode specifies program counter relative addressing.

It is often desirable to have the data tables and the programs that access them, maintain the same addressing relationship after the program and table are moved to a different place in memory. This can be done with program counter relative addressing. Also, this addressing mode is needed for position independent code.

In the following example, the starting address of a table is loaded into the X register:

```
        LEAX      TABLE,PCR
```

The number of memory locations between this instruction and the beginning of the table are contained in the symbol TABLE. As long as this number does not change, the program and the table may be moved anywhere in memory, and the program will still run correctly. This is the essence of position independent code.

Program counter relative addressing is a type of indexed addressing. Therefore, it may also have a level of indirection, as in this example:

```
        LEAX      [TABLE,PCR]
```

USING THE 6809 ADDRESSING MODES

This section contains short program examples illustrating the use of several addressing modes. These programs are often used as parts of larger programs.

Use of Indexing for Sequential Block Access

Indexing is primarily used for addressing successive locations within a table. It is sometimes desirable to limit the table size to 256, so that an 8-bit register may count the entries.

Let's now search a table of 100 elements for the * character. The starting address for this table is called BASE. The table has only 100 elements. Figure 5.6 shows the algorithm. Here is the program:

```
SEARCH  LDX       #BASE
        LDA       #'*
        LDB       #COUNT
TEST    CPA       ,X+
        BEQ       FOUND
        DECB
        BNE       TEST
NOTFND  . . .
```

A Block Transfer Routine for Fewer than 256 Elements

In the following program, "COUNT" is the number of elements in the block to be moved. The number is assumed to be less than 256. FROM is the base address of the block, and TO is the base of the memory area where it should be moved. The algorithm is quite simple: We will move a word at a time, and keep track of the word we are moving by storing its position in the counter B. Let's examine the program:

```
BLKMOV  LDX     #FROM
        LDY     #TO
        LDB     #COUNT
NEXT    LDA     ,X+
        STA     ,Y+
        DECB
        BNE     NEXT
```

The first three instructions:

```
BLKMOV  LDX     #FROM
        LDY     #TO
        LDB     #COUNT
```

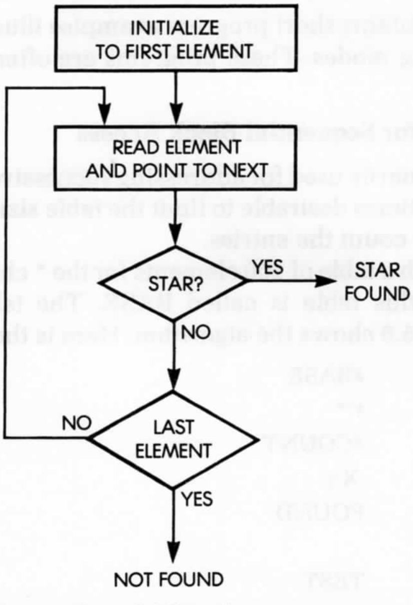


Figure 5.6: Character Search Flowchart

initialize registers X, Y, and B, respectively. The process appears in Figure 5.7.

Index register X is used as the source pointer, and is incremented by using the auto-increment addressing mode. Index register Y is used as the destination pointer, and is also incremented by using the auto-increment addressing mode. Register B is loaded with the number of elements to be transferred (limited to 256, since this is an 8-bit register), and is decremented regularly. Whenever B decrements to zero, all elements have been transferred. The next two instructions:

```
NEXT      LDA       ,X+
          STA       ,Y+
```

load the contents of the memory location pointed to by X into the accumulator A and increment the register X. Then, A is stored in the memory location pointed to by the register Y, and Y is incremented. In other words, these two instructions transfer an element of the source

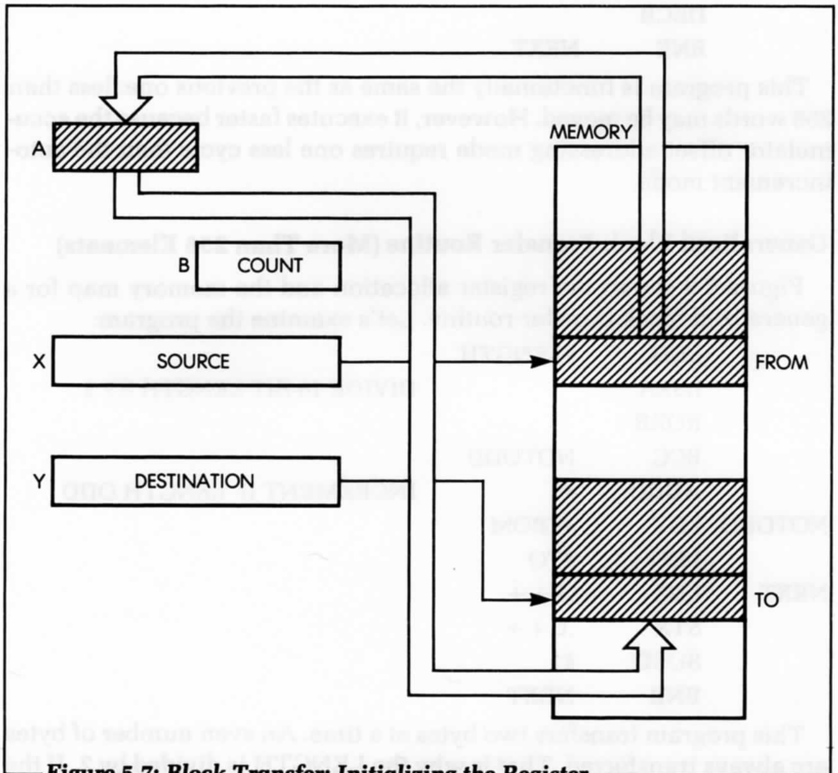


Figure 5.7: Block Transfer: Initializing the Register

block into the destination block, and increment the X and Y registers. The counter register is decremented:

```
        DECB
```

Finally, as long as the counter is not 0, the program loops back to the label NEXT:

```
        BNE     NEXT
```

This program is an example of the possible utilization of index registers. However, let's now compare it to the same program using the accumulator offset indexing mode. In this program, accumulator B is used as the offset, as well as the count:

```
BLKMOV  LDX     #FROM
        LDY     #TO
        LDB     #COUNT
NEXT    LDA     B,X
        STA     B,Y
        DECB
        BNE     NEXT
```

This program is functionally the same as the previous one: less than 256 words may be moved. However, it executes faster because the accumulator offset addressing mode requires one less cycle than the autoincrement mode.

Generalized Block Transfer Routine (More Than 256 Elements)

Figure 5.8 shows the register allocation and the memory map for a generalized block transfer routine. Let's examine the program:

```
        LDD     #LENGTH
        LSRA                    DIVIDE 16-BIT LENGTH BY 2
        RORB
        BCC     NOTODD
        ADDD    #1              INCREMENT IF LENGTH ODD
NOTODD  LDY     #FROM
        LDU     #TO
NEXT    LDX     ,Y++
        STX     ,U++
        SUBD    #1
        BNE     NEXT
```

This program transfers two bytes at a time. An even number of bytes are always transferred. That is why the LENGTH is divided by 2. If the length is an odd number before division, the last byte is not transferred. This is taken into account by testing the carry after the division. If there is a carry, one is added to D, so the last byte and one extra byte will be transferred.

The X register is used as the transfer register, while Y and U are used as index registers. The auto-increment by two address mode is used because two bytes are transferred each time the loop is executed.

Finally, a decrement is not available for the D accumulator, so a one is subtracted. When the D accumulator is zero, the program is finished.

Adding Two Blocks

We will now develop a program that adds, element-by-element, two blocks that start at addresses BLK1, and BLK2, respectively, and that have an equal number of elements, COUNT. Here is the program:

```
BLKADD  LDX     #BLK1
        LDY     #BLK2
        LDB     #COUNT
        CLRA
LOOP    LDA     ,X
        ADCA    ,Y+
        STA     ,X+
        DECB
        BNE     LOOP
```

The memory layout appears in Figure 5.9.

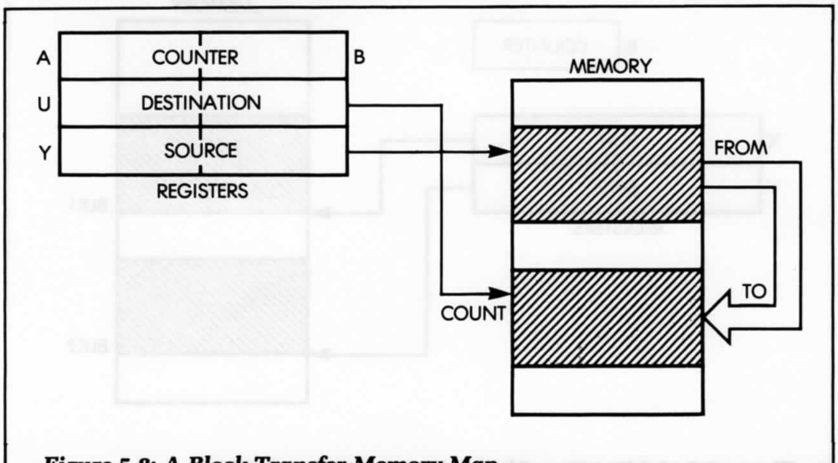


Figure 5.8: A Block Transfer-Memory Map

The program is straightforward. The number of elements to be added is loaded into the counter register B, and the two index registers, X and Y, are initialized to their values BLK1 and BLK2:

```
BLKADD  LDX     #BLK1
        LDY     #BLK2
        LDB     #COUNT
```

The carry bit is then cleared in anticipation of the first addition:

```
        CLRA
```

The first element is loaded into the accumulator:

```
LOOP    LDA     ,X
```

The corresponding element of BLK 2 is then added to it, and the Y register is incremented:

```
        ADCA    ,Y+
```

and finally saved in the memory element of BLK1 pointed to by X:

```
        STA     ,X+
```

The X register is incremented when the byte is stored in BLK1. The counter register is decremented:

```
        DECB
```

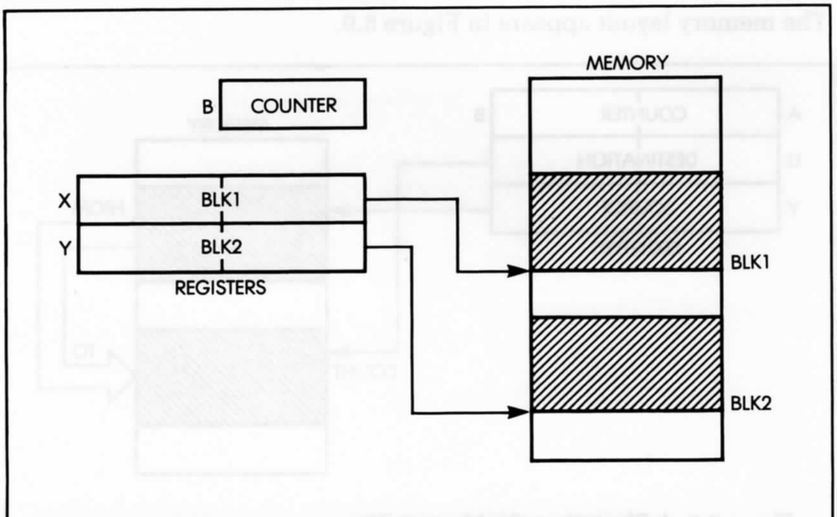


Figure 5.9: Adding Two Blocks: BLK1 = BLK1 + BLK2

As long as the counter register is not 0, the addition loop is executed:

```
        BNE     LOOP
```

SUMMARY

We have now discussed addressing modes and analyzed those available on the 6809. We have seen that the 6809 offers many possible addressing mechanisms. To program the 6809 efficiently, it is necessary to understand these mechanisms. They will be used throughout the remainder of this book.

EXERCISES

5-1: *Use the block addition program to perform a 32-bit addition.*

5-2: *Use the block addition program to perform a 64-bit addition.*

5-3: *Modify the block addition program, so that the result is stored in a separate block starting at address BLK3.*

5-4: *Modify the block addition program to perform a subtraction, rather than an addition.*

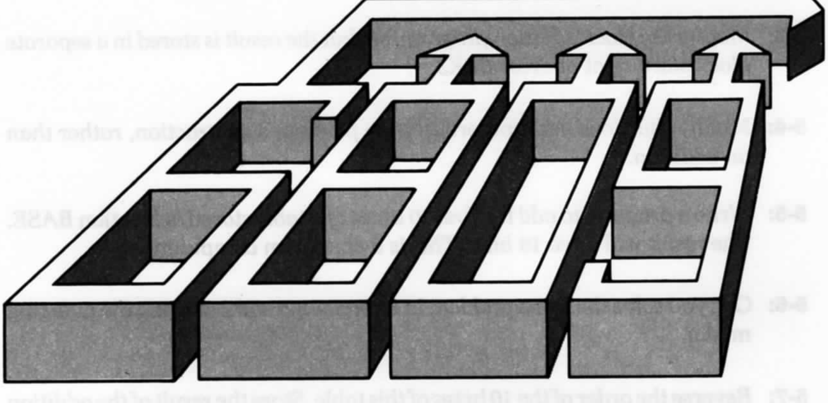
5-5: *Write a program to add the first 10 bytes of a table stored at location BASE. The result will have 16 bits. (This is a checksum computation.)*

5-6: *Can you solve the same problem in Exercise 5-5 without using the indexing mode?*

5-7: *Reverse the order of the 10 bytes of this table. Store the result of the addition at address REVER.*

5-8: *Search the same table for its largest element. Store it at memory address LARGE.*

CHAPTER 6



INPUT/OUTPUT TECHNIQUES

SO FAR IN THIS BOOK, we have seen how to exchange information between the memory and the various registers of the processor; we have learned how to manage registers; and we have learned how to use a variety of instructions to manipulate data. We will now examine input/output techniques and learn how to communicate with the external world.

The principal advantage of the 6809 architecture in this important area is its powerful interrupt structure, which provides, in addition to a regular interrupt mode, a fast and a non-maskable interrupt mode. Also important in the use of these interrupt modes are the 6809's unique SYNC and CWAI instructions, which we will also explore in this chapter.

Input is the transfer of data from an external peripheral (keyboard, disk, or physical sensor) to *internal* computer storage. *Output* is the transfer of data from within the microprocessor or the memory to an *external* device, such as a printer, CRT, disk, or actual sensors and relays. In this chapter, we will perform the input/output operations required in most computer applications. We will be managing several input/output devices simultaneously, and, finally, we will discuss the subject of polling versus interrupts.

THE 6809 INPUT/OUTPUT INSTRUCTIONS

For input or output on the 6809, we can use any instruction that transfers data to or from the memory. Input/output interfacing on the 6809 is called *memory mapped interfacing,* because input/output devices are interfaced to the 6809 in the same way that memory is interfaced. We can use *any* addressing mode for input or output; however, extended addressing is commonly used, because the addresses of input/output devices rarely change once a system has been built.

Generating a Signal

To generate a signal, the computer must turn an output device off or on. To do this, we must change an electrical voltage level in the device from a logical 0 to a logical 1, or from a 1 to a 0. For example, let's assume that an external relay is connected to bit 0 of a register called OUT1. To turn the relay on, we simply write a 1 in the appropriate bit position of the register. We assume here that OUT1 represents the address of the device output register in the system. Here is a program that will turn the relay on:

```
TURNON  LDA     #%00000001    LOAD PATTERN INTO A
        STA     >OUT1         OUTPUT IT TO DEVICE
```

STA is the output instruction. The > symbol indicates extended addressing.

In this example, we have assumed that the states of the other seven bits of the register OUT1 are irrelevant. However, this is often not the case, as these bits might be connected to other relays. Let's, therefore, improve this simple program by turning the relay on, without changing the state of any other bit in the register. We will assume that we can read and write the contents of this register. The improved program is:

```
TURNON  LDA     >OUT1         READ CONTENTS OF OUT1
        ORA     #%00000001    FORCE BIT 0 TO 1 IN A
        STA     >OUT1
```

This program first reads the contents of OUT1, then performs an inclusive OR on its contents. It changes bit position 0 to 1, and leaves the rest of the register intact (see Figure 6.1).

Pulses

We can generate a *pulse* in the same way that we change the voltage *level*. We first turn an output bit on, then we turn it off. This results in a pulse, as illustrated in Figure 6.2. In this example, however, we must solve an additional problem: We need to generate a pulse for a specified length of time. Thus, we must generate a computed delay.

Delay Generation and Measurement

We can generate a delay by using both software and hardware methods. Let's first generate one using software; then we will later generate one using a hardware counter, called a *programmable interval timer* (PIT).

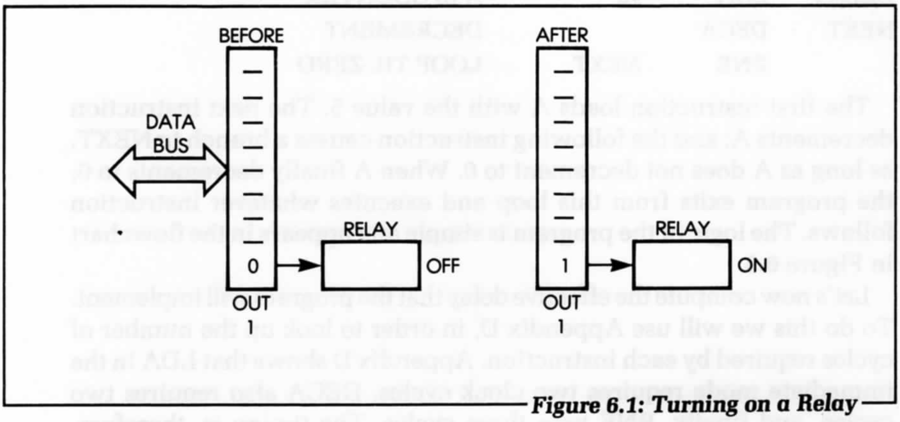
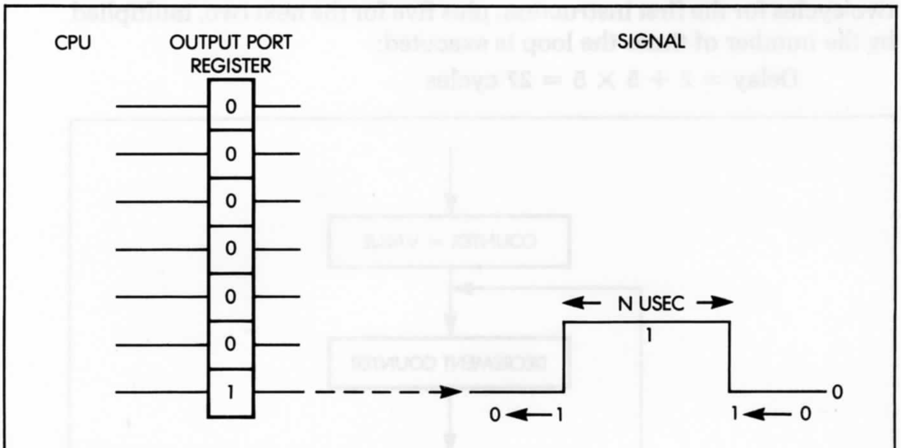


Figure 6.1: Turning on a Relay



THE PROGRAM:

SELECT OUTPUT PORT
LOAD OUTPUT REGISTER WITH PATTERN
WAIT (LOOP FOR N USEC)
LOAD OUTPUT PORT WITH ZERO
RETURN

Figure 6.2: A Programmed Pulse

Programmed delays are achieved by *counting*. A counter register is first loaded with a value, then decremented. The program loops on itself and continues decrementing until the counter reaches the value 0. The total length of time used by this process implements the required delay. As an example, let's generate a delay of 27 clock cycles:

```
DELAY   LDA    #5       A IS COUNTER
NEXT    DECA            DECREMENT
        BNE    NEXT     LOOP TIL ZERO
```

The first instruction loads A with the value 5. The next instruction decrements A; and the following instruction causes a branch to NEXT, as long as A does not decrement to 0. When A finally decrements to 0, the program exits from this loop and executes whatever instruction follows. The logic of the program is simple and appears in the flowchart in Figure 6.3.

Let's now compute the effective delay that the program will implement. To do this we will use Appendix D, in order to look up the number of cycles required by each instruction. Appendix D shows that LDA in the immediate mode requires two clock cycles. DECA also requires two cycles, and finally, BNE uses three cycles. The timing is, therefore, two cycles for the first instruction, plus five for the next two, multiplied by the number of times the loop is executed:

$$\text{Delay} = 2 + 5 \times 5 = 27 \text{ cycles}$$

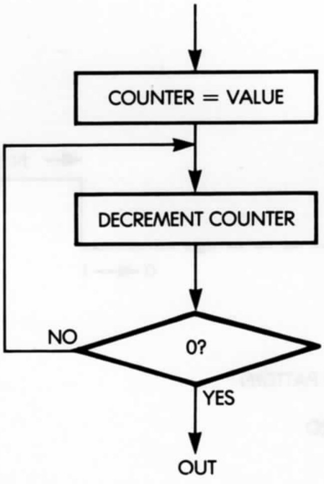


Figure 6.3: Basic Delay Flowchart

Assuming a 1 microsecond clock (a 4 MHz crystal), the programmed delay will be 27 microseconds.

(*Note:* The delay loop just described is used by most input/output programs. Be sure you understand it.)

To implement a longer delay, we simply add extra instructions in the program between the instructions DECA and BNE. The simplest way to do this is to add several NOP instructions. (The NOP instruction does nothing for two cycles.)

Longer Delays

To generate longer delays using software, we can use a wider counter. For example, we can use a *register pair* to hold a 16-bit count. To simplify, let us assume that the lower count is 0. We load the lower byte with 0 (the maximum count), and it will go through a decrementation loop. When it is decremented to 0, the upper byte of the counter is decremented by 1. When the upper byte is decremented to the value 0, the program terminates. If more precision is required in the delay generation, the lower count can have a non-null value. In this case, we would write the program as explained and add the three-line delay generation program described above.

Here is a 24-bit delay program:

```
DEL24   LDA    #COUNTH   COUNTER HIGH (8 BITS)
        STA    COUNTR
DEL16   LDD    #COUNTL   COUNTER LOW (16 BITS)
LOOP    SUBD   #1        DECREMENT IT
        BNE    LOOP      LOOP UNTIL ZERO
        DEC    COUNTR    DECREMENT HIGH COUNTER
        BNE    DEL16     REPEAT UNTIL ZERO
```

Note that a SUBD must be used because there is no DECD.

Naturally, we could generate still longer delays by using more than three words. Actually, this example is analogous to the way an odometer works on a car. When the right-most wheel goes from 9 to 0, the next wheel to the left is incremented by 1. This is the general principle when counting with multiple discrete units.

The main disadvantage of this method, however, is that when the computer is counting delays, the microprocessor does nothing else for hundreds of milliseconds or even seconds. If the computer has nothing else to do, this is acceptable. However, in general, the microcomputer should be available for other tasks. Therefore, long delays are normally not implemented by software. In fact, even short delays may be objectionable in a system, if the system is to provide guaranteed response time in certain situations. (In such situations, we must use hardware delays.) Another disadvantage of the software delay is that, if the program is interrupted, timing accuracy may be lost.

Hardware Delays

Hardware delays are implemented by using a *programmable interval timer*, or "timer," in short. When using a programmable interval timer, a register of the timer is loaded with a value. The timer automatically decrements the counter periodically. The programmer can usually adjust or select the amount of time between decrements. When the timer has decremented to 0, it normally sends an interrupt to the microprocessor. It may also set a status bit, that can be sensed periodically by the computer. (We discuss interrupts later in this chapter.)

Other timer operating modes may include starting from 0 and counting the duration of the signal or the number of pulses received. When functioning as an interval timer, the timer is said to operate in a *one-shot* mode. When counting pulses, it is said to operate in a *pulse counting* mode. Some timer devices may even include multiple registers and a number of optional facilities, which the programmer can select.

Sensing Pulses

The problem with sensing pulses is the reverse of the problem with generating pulses, and includes one more difficulty: an output pulse is generated under program control; an input pulse occurs *asynchronously* with the program. We can use two methods to detect a pulse: *polling* and *interrupts*. Let's first discuss polling.

Using the polling technique, the program continuously reads the value of a given input register and tests a bit position, perhaps bit 0. We will assume that bit 0 was originally 0. (Thus, when a pulse is received, this bit takes the value 1.) The program continuously monitors bit 0 until it takes the value 1. When a 1 is found, the pulse has been detected. Here is a program that does this:

```
POLL    LDA     >INPUT       READ INPUT REGISTER
        BITA    #%00000001   TEST FOR 0
        BEQ     POLL         KEEP POLLING IF ZERO
```

Conversely, let's assume that the input line is normally 1, and we want to detect a 0. This is the usual case for detecting a START bit, when monitoring a line connected to a Teletype. Here is the program:

```
POLL    LDA     >INPUT      READ INPUT REGISTER
        BITA    #%00000001  SET Z BIT
        BNE     POLL        TEST IS REVERSED
START   . . .
```

Monitoring the Duration

We monitor the duration of a pulse in the same way that we compute the duration of an output pulse. We may use either a hardware or software technique. When we monitor a pulse by using software, a counter is regularly incremented by 1, then the presence of the pulse is verified. If the pulse is still present, the program loops upon itself. If the pulse disappears, the count contained in the counter register is used to compute the effective duration of the pulse. Here is a program that monitors pulse duration:

```
DURTN   CLRB                CLEAR COUNTER
AGAIN   LDA     >INPUT      READ INPUT
        BITA    #%00000001  MONITOR BIT 0
        BEQ     AGAIN       WAIT FOR A 1
LONGER  INCB                INCREMENT COUNTER
        LDA     >INPUT
        BITA    #%00000001  CHECK BIT 0
        BNE     LONGER      WAIT FOR A 0
```

Naturally, we assume that the maximum duration of the pulse will not cause register B to overflow. However, if B does overflow, the program will have to be changed to take that into account (or there will be a programming error!).

Since we now know how to sense and generate pulses, let's learn how to capture and transfer large amounts of data. We will later apply this knowledge to actual input/output devices.

PARALLEL WORD TRANSFER

We will assume here that eight bits of transfer data are available in parallel at address INPUT (see Figure 6.4). We will also assume that the status information is contained in bit 7 of address STATUS. The microprocessor must read the data word at this location whenever a status word indicates that it is valid.

We will now write a progam that reads and automatically saves each word of data as it comes in. For simplicity, we will assume that the number of words to be read is known in advance and is contained in location COUNT. But, if this information is not available, we will test for a so-called *break character,* such as a *rubout,* or perhaps the character "*". We have learned how to do this already.

The flowchart for this example appears in Figure 6.5. We will test the status information until bit 7 becomes 1, indicating that a word is ready. When the word is ready, we will read it and save it at an appropriate memory location. We will then decrement the counter and test whether it has decremented to 0. If so, the task is completed; if not, we will read

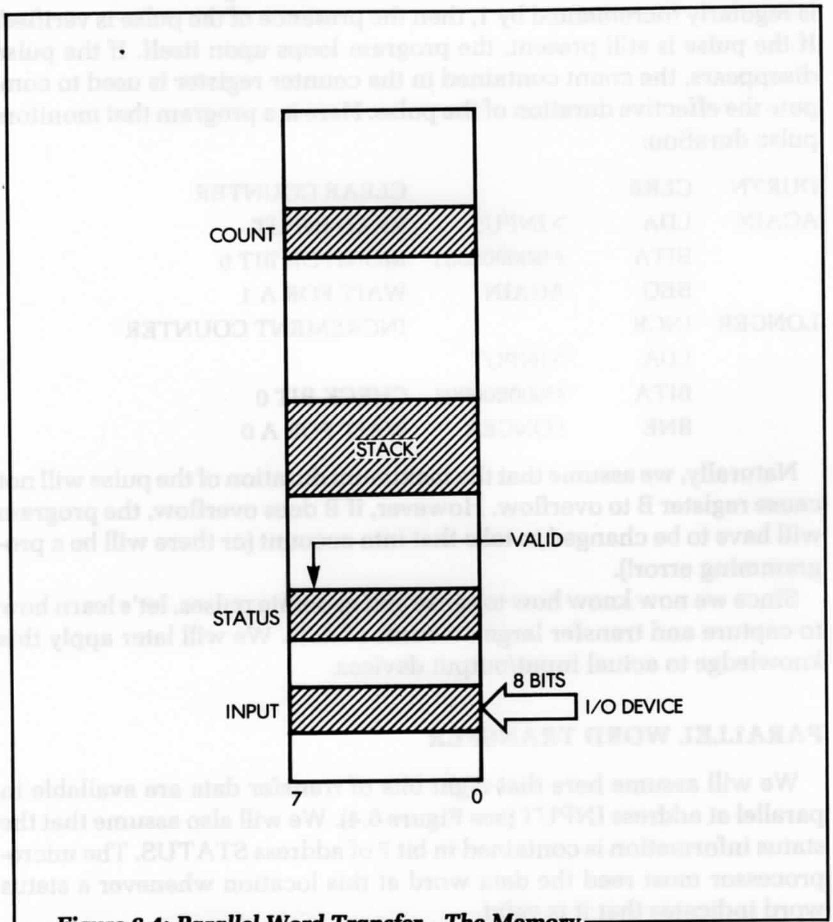


Figure 6.4: Parallel Word Transfer—The Memory

the next word. Here is a simple program that implements this algorithm:

```
PARAL   LDB     COUNT       READ COUNT INTO A
WATCH   LDA     >STATUS     LOOK FOR DATAREADY TRUE
        BPL     WATCH       LOOP TIL READY
        LDA     >INPUT      READ DATA
        PSHS    A           SAVE DATA ON STACK
        DECB                DECREMENT COUNT
        BNE     WATCH       REPEAT UNTIL ZERO
```

We assume here that the "data ready" flag is automatically cleared when STATUS is read. This is usually the case on a device controller.

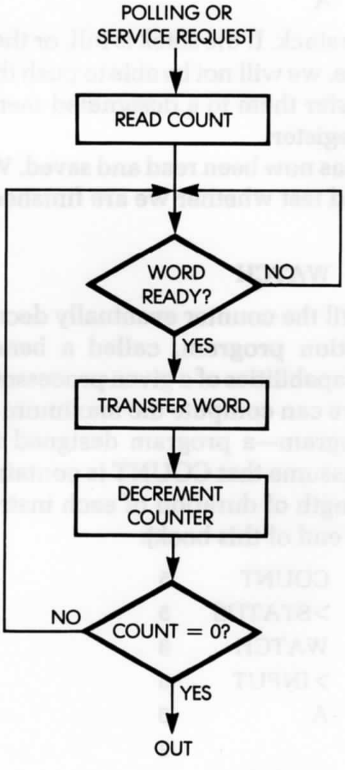


Figure 6.5: Parallel Word Transfer: Flowchart

The first instruction initializes the counter register B:

```
PARAL   LDB     COUNT
```

The next few instructions read the status information and cause a loop to occur when bit 7 of the status register is 0. The LD instruction sets the condition code bits. Bit 7 causes the N bit to be set.

```
        LDA     >STATUS
        BPL     WATCH
```

When BPL fails, the data is valid and we can read it:

```
        LDA     >INPUT
```

The word has now been read from address INPUT and must be saved. Assuming that a sufficient stack area is available, we can use the following instruction:

```
        PSHS    A
```

which saves A on the stack. If the stack is full, or the number of words to be transferred is large, we will not be able to push them on the stack, and we will have to transfer them to a designated memory area, using, for example, an index register.

The word of data has now been read and saved. We simply decrement the word counter and test whether we are finished:

```
        DECB
        BNE     WATCH
```

We keep looping until the counter eventually decrements to 0.

This nine-instruction program, called a *benchmark program*, is designed to test the capabilities of a given processor for a specific operation. For example, we can compute the maximum transfer speed of the parallel transfer program—a program designed for maximum speed and efficiency. We assume that COUNT is contained in memory. Let's now examine the length of duration of each instruction (these figures are also given at the end of this book):

```
PARAL   LDB     COUNT      5
WATCH   LDA     >STATUS    5
        BPL     WATCH      3
        LDA     >INPUT     5
        PSHS    A          6
        DECB               2
        BNE     WATCH      3
```

We can obtain the minimum execution time by assuming that the data is ready every time we sample STATUS. In other words, if we assume that the BPL will fail every time, the length of time necessary to transfer a block is then:

$$5 + (5 + 3 + 5 + 6 + 2 + 3) \times \text{COUNT}$$

If we neglect the first 5 cycles necessary to initialize the counter register, the time used to transfer one word is 24 clock cycles, which is 24 microseconds with a 4MHz crystal. The maximum data transfer rate is:

$$\frac{1}{24(10^{-6})} = 42 \text{ K bytes per second}$$

We have now learned to perform high-speed parallel transfers. Let's examine a more complex case.

BIT SERIAL TRANSFER

A serial input is one in which the bits of information (0s or 1s) come in successively on a line. These bits may come in at regular intervals, called *synchronous* transmission, or at random intervals as bursts of data, called *asynchronous* transmission. We will now develop a program that works in both cases.

The principle of the capture of sequential data is simple. We watch an input line, which we assume to be line 0. When a bit of data is detected on this line, we read the bit in, and shift it into a holding register. When we have assembled eight bits, we preserve the byte of data in the memory and assemble the next one.

To simplify this example, we will assume that we know the number of bytes to be received. Otherwise, we might have to watch for a special break character, and stop the bit-serial transfer at that point. Figure 6.6 shows the flowchart for this program. Here is the program:

```
SERIAL   CLRB               CLEAR INPUT WORD
         LDA    #COUNT      PUT BYTE COUNT INTO
         STA    COUNTR      COUNTR WORD
LOOP     LDA    >INPUT      READ PORT
         BPL    LOOP        WAIT FOR BIT 7 = 1
         LSRA               SHIFT DATA BIT INTO CARRY
         ROLB               SAVE CARRY IN B
         BCC    LOOP        CONTINUE UNTIL 8 BITS IN
         PSHS   B           SAVE WORD ON STACK
         LDB    #$01        RESET MARKER BIT
         DEC    COUNTR      DECREMENT BYTE COUNTER
         BNE    LOOP        ASSEMBLE NEXT WORD
```

This program has been designed for efficiency. It uses new techniques, which we will explain later in this chapter (see Figure 6.7). The conventions are the following: memory location COUNTR is assumed to contain a count of the number of words to be transferred. Register B is used to assemble eight consecutive bits coming in. Address INPUT refers to an input register. It is assumed that bit position 7 of this register is a status flag, or a clock bit. (When it is 0, the data is invalid; when it is 1, the data is valid.) We assume that the data itself appears in bit position 0 of this same address. (In many instances, the status information appears on a different register than the data register. Since this is in the same address, it should be a simple task, then, to modify this program accordingly.) In addition, we assume that the first bit of data to be received by this program is guaranteed to be a 1. This 1 indicates that the real data follows. However, if this is not the case, as we will later see, there is an obvious modification that will correct this problem.

The program corresponds to the flowchart in Figure 6.6. The first few lines of the program implement a waiting loop, which tests whether a bit is ready. To determine whether a bit is ready, we first read the input register, then we test the negative bit (N). As long as this bit is 0, the instruction BPL will succeed, and we will branch back to the loop. Whenever the status (or clock) bit becomes true (1), then BPL will fail and the next instruction will be executed. This initial sequence of instructions corresponds to arrow 1 in Figure 6.7.

At this point, A contains a 1 in bit position 7, and the actual data bit is in bit position 0. The first data bit to arrive will be a 1. However, the following bits may be either 0 or 1. We will now preserve the data bit that has been collected in position 0. The instruction:

```
        LSRA
```

shifts the contents of A to the right by one position. This causes the right-most bit of A, the data bit, to fall into the carry bit. Next, we preserve this data bit into register B (this process is illustrated by arrows 2 and 3 in Figure 6.7) with the instruction:

```
        ROLB
```

This instruction reads the carry bit into the right-most bit position of B. At the same time, the left-most bit of B falls into the carry bit. (If you have any doubts about the rotation operation, refer to Chapter 4.)

It is important to remember that a rotation operation both saves the carry bit (here into the right-most bit position), and reconditions the carry bit with the value of bit 7. In this case, a 0 falls into the carry.

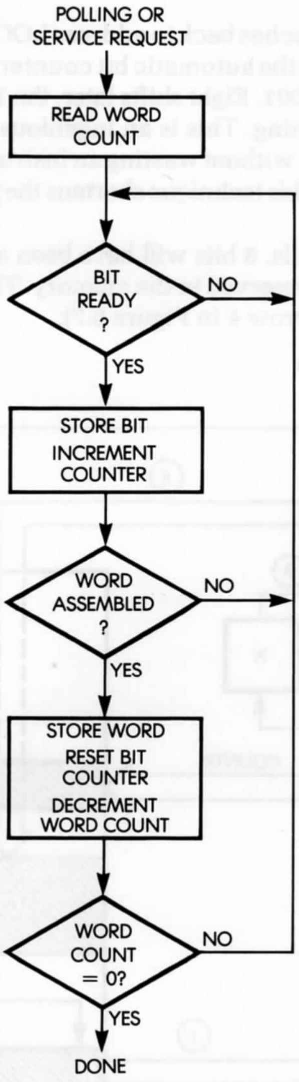


Figure 6.6: Bit Serial Transfer—Flowchart

The next instruction:

```
        BCC         LOOP
```

tests the carry and branches back to address LOOP, as long as the carry is 0. This instruction is the automatic bit counter. As a result of the first ROL, B contains 00000001. Eight shifts later, the 1 will fall into the carry bit and stop the branching. This is an ingenious way to implement an automatic loop counter without wasting an instruction to decrement the contents of a register. This technique shortens the program and improves its performance.

When BCC finally fails, 8 bits will have been assembled into B. This value should then be preserved in the memory. This is accomplished by the next instruction (arrow 4 in Figure 6.7):

```
        PSHS        B
```

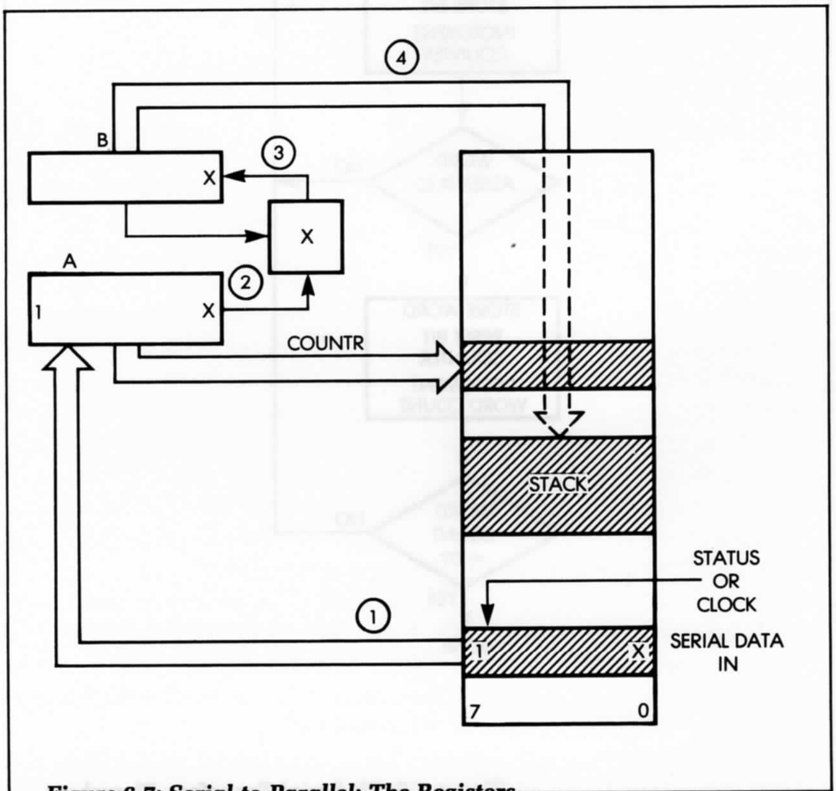


Figure 6.7: Serial-to-Parallel: The Registers

This instruction saves the contents of B on the stack. But, this is possible only if there is enough room in the stack. Provided this condition is met, this is usually the fastest way to preserve a word in the memory. The stack pointer is updated automatically. If we were not pushing a word on the stack, we would have to use one more instruction to update a memory pointer. We could equivalently perform an indexed addressing operation by using an auto increment or decrement addressing mode.

After the first word of data has been saved, there is no guarantee that the first data bit to come in will be a 1. It could be a 0. We must, therefore, reset the contents of B to 00000001, so that we can continue to use the carry bit as a bit counter. We do this with the next instruction:

```
        LDB         #$01
```

Finally, we decrement the word counter, since a word has been assembled, and test whether we have reached the end of the transfer. This is accomplished by the next two instructions:

```
        DEC         COUNTR
        BNE         LOOP
```

The above program has been designed for speed, so that we may capture a fast input stream of data bits. Once the program terminates, it is naturally advisable to immediately read away from the stack the words that have been saved there, and transfer them into another part of the memory where they may be processed. We performed such a block transfer in Chapter 5.

This program is more complex than the previous ones. Let's look at it again in more detail, and examine some possible trade-offs (see Figure 6.6).

Referring to the bit serial transfer program, we see that from time to time a bit of data comes into bit position 0 of INPUT. For example, there might be three 1s in succession. We must, therefore, *differentiate between the successive bits* coming in. This is the function of the clock signal.

The *clock* (or STATUS) signal tells us when the input bit is valid. Therefore, before we read a bit, we must test the status bit. If the status is 0, we must wait. If it is 1, the data bit is good. We assume here that the status signal is connected to bit 7 of register INPUT.

Once we have captured a data bit, we want to preserve it in a safe location. Then, we want to shift it left, so that we can get the next bit.

Unfortunately, in this program we use the accumulator to read and test both data and status. If we were to accumulate data in the A accumulator, bit position 7 would be erased by the status bit.

In this example, we have assumed that the first bit to come in is a special signal, guaranteed to be a 1. However, in general, it could be a 0.

The program could be modified to handle data as the first bit. In addition, note that we have saved the assembled word in the stack; however, we could have saved it in some other memory area.

The Hardware Alternative

As usual for most standard input/output algorithms, we can implement the serial to parallel conversion through hardware. The hardware chip to do this is called a UART. The UART automatically accumulates the bits. If we want to reduce the component count, we should use this program, or a variation of it.

BASIC I/O SUMMARY

So far, we have learned to perform elementary input/output operations and to manage a stream of parallel data or serial bits. We are now ready to communicate with real input/output devices.

COMMUNICATING WITH INPUT/OUTPUT DEVICES

To exchange data with input/output devices, we must first ascertain whether data is available, and if so, if we want to read it; or, we must ascertain whether the device is ready to accept data, and if so, if we want to send it. We can use two procedures to do this: handshaking and interrupts. Let's first discuss handshaking.

Handshaking

Handshaking is generally used as a communication tool between two asynchronous devices, i.e., between two devices that are not synchronized. For example, if we want to send a word to a parallel printer, we must first make sure that the input buffer of the printer is available. We must, therefore, ask the printer: "Are you ready?" The printer will respond either "yes" or "no." If it is not ready, we must wait. If it is ready, we can send the data (see Figure 6.8).

Conversely, before reading data from an input device, we must verify whether the data is valid. We ask: "Is data valid?" The device will respond either "yes" or "no." The "yes" or "no" may be indicated by status bits, or by some other means (see Figure 6.9).

As an analogy, if we wish to exchange information with someone who is doing something else at the time, we need to ascertain that that person is ready to communicate with us. The usual rule of courtesy is to shake hands—data exchange may then follow. This is also the procedure normally used in communicating with input/output devices. Let's examine a simple example.

Sending a Character to the Printer

In this example, the character we wish to print is assumed to be in memory location CHAR. Here is the program that we can use to print it:

```
WAIT    LDA     >STATUS
        BPL     WAIT        WAIT TIL READY
        LDA     CHAR        GET CHARACTER
        STA     >PRINTD     PRINT IT
        BR      WAIT        GO FOR NEXT
```

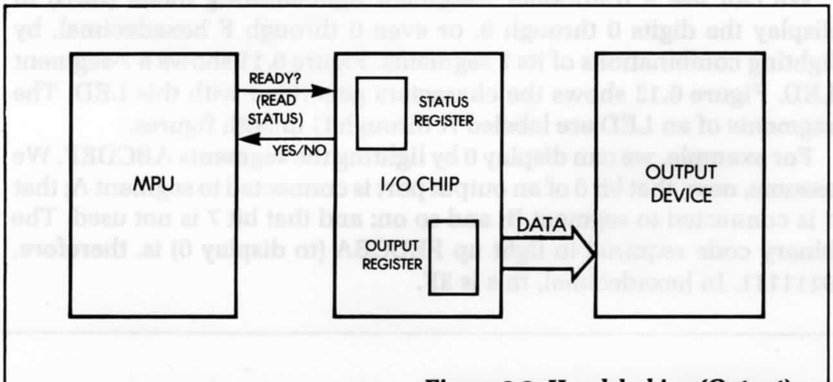


Figure 6.8: Handshaking (Output)

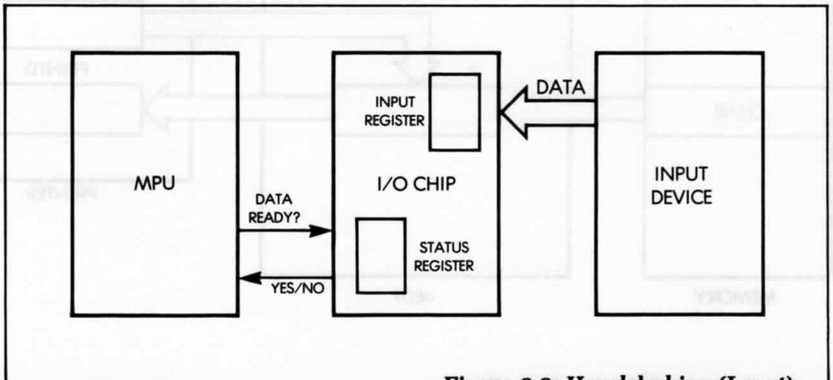


Figure 6.9: Handshaking (Input)

This program is straightforward and uses the handshaking procedure described previously. The data paths appear in Figure 6.10.

The character (called DATA) is located at memory location CHAR. First, the status of the printer is checked. Whenever bit 7 of the status register becomes 1, it indicates that the printer is ready for output, i.e., its output buffer is available. At this point, the character is loaded into the accumulator, and then output to the printer, via the accumulator. As long as the status bit remains 0, the program will remain in a loop, called WAIT.

Let's now complicate the output procedure by requiring a code conversion and by outputting to several devices at a time.

Output To a 7-Segment LED

We can use a traditional 7-segment light-emitting diode (LED) to display the digits 0 through 9, or even 0 through F hexadecimal, by lighting combinations of its 7 segments. Figure 6.11 shows a 7-segment LED. Figure 6.12 shows the characters generated with this LED. The segments of an LED are labeled A through G in both figures.

For example, we can display 0 by lighting the segments ABCDEF. We assume, now, that bit 0 of an output port is connected to segment A; that 1 is connected to segment B; and so on; and that bit 7 is not used. The binary code required to light up FEDCBA (to display 0) is, therefore, 0111111. In hexadecimal, this is 3F.

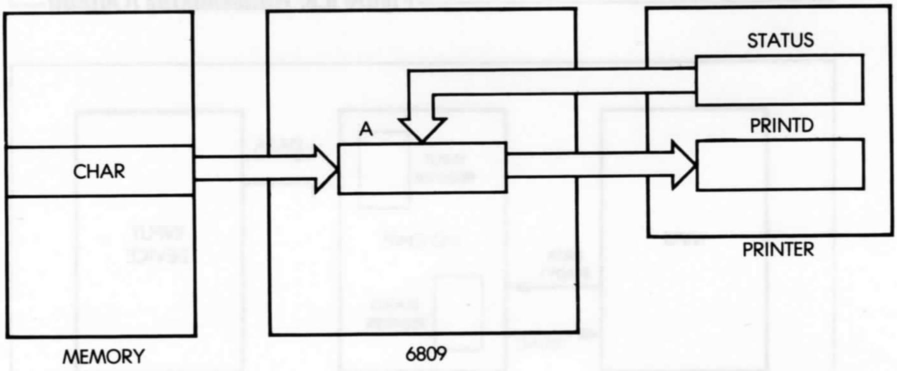


Figure 6.10: Printer—Data Paths

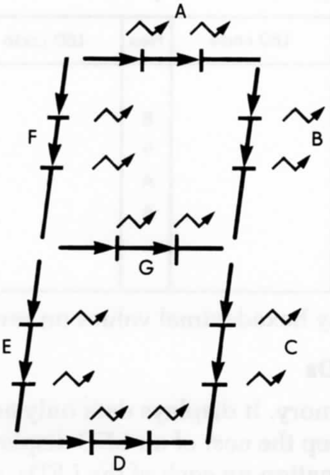


Figure 6.11: 7-Segment LED

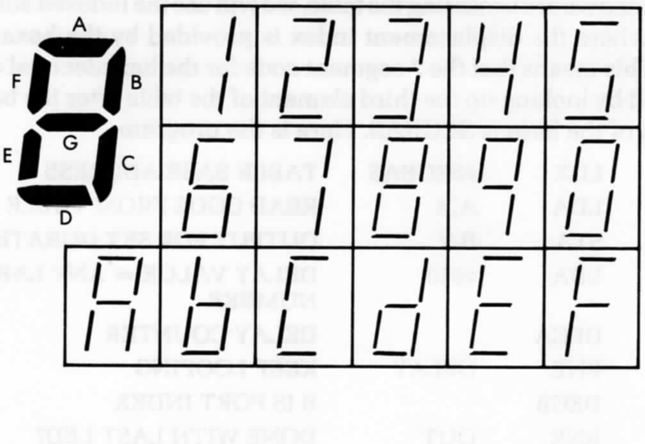


Figure 6.12: Hexadecimal Characters Generated with a 7-Segment LED

As an exercise, try computing the 7-segment equivalent for the hexadecimal digits 0 through F, and complete the following table.

Hex	LED code	Hex	LED code	Hex	LED code	Hex	LED code
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

We will now display hexadecimal values on *several* LEDs.

Driving Multiple LEDs

An LED has no memory. It displays data only as long as its segment lines are active. To keep the cost of an LED display low, the microprocessor displays information on *each of the LEDs*, in turn. The rotation between the LEDs must be fast enough so that there is no apparent blinking. This implies that the time spent going from one LED to the next is less than 100 milliseconds. Let's design a program that accomplishes this.

We will use register B to point to the LED on which we want to display a digit. A is assumed to contain the hexadecimal value to be displayed on the LED. Our first concern is to convert the hexadecimal value into its 7-segment representation. In the last section, we built an equivalence table. Since we are accessing the table, we will use the indexed addressing mode, where the displacement index is provided by the hexadecimal value. This means that the 7-segment code for the hexadecimal digit 3 is obtained by looking up the third element of the table after the base. The address of the base is SEGBAS. Here is the program:

```
LEDS     LDX      #SEGBAS    TABLE BASE ADDRESS
         LDA      A,X        READ CODE FROM TABLE
         STA      B,Y        OUTPUT FOR SET DURATION
         LDA      #$50       DELAY VALUE = ANY LARGE
                             NUMBER
DELAY    DECA                DELAY COUNTER
         BNE      DELAY      KEEP LOOPING
         DECB                B IS PORT INDEX
         BNE      OUT        DONE WITH LAST LED?
         LDB      #MAXLED    IF SO, RESET B TO TOP LED
OUT      RTS
```

This program assumes that the Y register points to the base address of the LEDs, and that B is added to Y in order to point to the next LED to be illuminated. The A accumulator contains the digits to be displayed.

The program first looks up the 7-segment code corresponding to the hexadecimal value contained in the accumulator. The A register is used as a displacement field, and the X register is used as a 16-bit index register. The code for the hexadecimal digit is added to the base address of the table:

```
LEDS    LDX    #SEGBAS
        LDA    A,X
```

The next instruction outputs the 7-segment code to the address specified, by using B as a displacement for the Y index register:

```
        STA    B,Y
```

A delay loop is then implemented, so that the code from the table is displayed for an appropriate duration. Here we have arbitrarily chosen the constant, 50 hexadecimal. The next three instructions implement the delay loop:

```
        LDA    #$50
DELAY   DECA
        BNE    DELAY
```

Once the delay has been implemented, we simply decrement the LED pointer displacement, and make sure we loop around to the highest LED address, if the smallest LED address has been reached.

```
        DECB
        BNE    OUT
        LDB    #MAXLED
OUT     RTS
```

It is assumed here that this program was written as a subroutine; the last instruction is, therefore, RTS: "return from subroutine."

We have now solved common input/output problems. Let's consider the case of a common peripheral: the Teletype.

Teletype Input/Output

The *Teletype* is a serial device that sends and receives words of information in a serial format. Each character is encoded in an 8-bit ASCII format. (The ASCII table appears in Appendix B.) In addition, every character is preceded by a "start" bit, and terminated by two "stop" bits. In the 20-milliamp current loop interface, which is most frequently used, the state of the line is normally a 1. This is used to indicate to the processor that the line has not been cut. A start is a 1-to-0 transition. This indicates to the receiving device that data bits follow. The standard Teletype is a 10-characters-per-second device. We have just established that each character requires 11 bits. This means that the Teletype will transmit 110 bits per second, i.e., that it is a 110-baud device. We will now design a program to serialize bits out to the Teletype at the correct speed.

One hundred ten bits per second implies that bits are separated by 9.09 milliseconds. This will have to be the duration in a program of the delay loop to be implemented between transmission or reception of successive bits. Figure 6.13 shows the format of a Teletype word. Figure 6.14 displays the flowchart for bit input. Here is the program:

```
TTYIN   LDA     >STATUS
        BPL     TTYIN       DATA READY?
        BSR     DELAY1      CENTER OF PULSE
        LDA     >TTYBIT     START BIT
        STA     >TTYBIT     ECHO IT
        BSR     DELAY9      NEXT PULSE 9MS
        LDB     #$08        BIT COUNT
        STB     COUNTR      COUNTER WORD
NEXT    LDA     >TTYBIT     READ DATA BIT
        STA     >TTYBIT     ECHO IT
        LSRA                SAVE IT IN CARRY
        RORB                PRESERVE IT IN B
        BSR     DELAY9      NEXT PULSE 9MS
        DEC     COUNTR      DECREMENT BIT COUNT
        BNE     NEXT
        LDA     >TTYBIT     READ STOP BIT
        STA     >TTYBIT     ECHO IT
        BSR     DELAY9      SKIP SECOND STOP
        RTS
```

Let's examine this program in detail.

First, we test the status of the Teletype to determine if a character is available:

```
TTYIN   LDA     >STATUS
        BPL     TTYIN
```

Then, we implement a 4.5 ms delay, in order to sense the start bit in the middle of the pulse:

```
         BSR      DELAY1
```

DELAY1 is the delay subroutine that implements the required delay. The first bit to come is the start bit. It should be echoed to the Teletype, but ignored by the rest of the program. This is done by the next few instructions:

```
         LDA      >TTYBIT
         STA      >TTYBIT
```

We must now wait for the first data bit. The necessary delay is equal to 9.09 ms and is implemented by a subroutine:

```
         BSR      DELAY9
```

Memory location COUNTR is used as a counter and loaded through the B register with the value 8, because 8 data bits are captured:

```
         LDB      #$08
         STB      COUNTR
```

Next, each data bit is read into A, in turn, then echoed. The data bit is assumed to arrive in bit position 0 of A. The data bit is then preserved in register B, where it is shifted in. The transfer from A to B is performed through the carry bit:

```
NEXT     LDA      >TTYBIT
         STA      >TTYBIT
         LSRA
         RORB
```

Figure 6.15 illustrates this sequence.

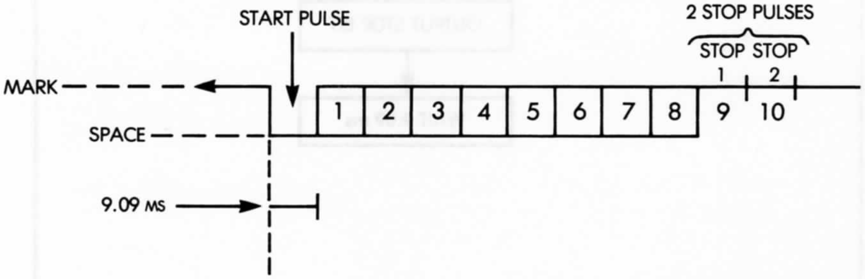


Figure 6.13: Format of a Teletype Word

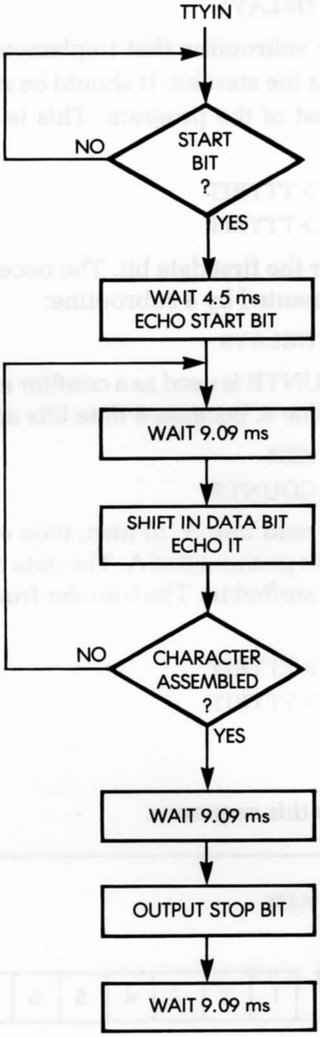


Figure 6.14: TTY Input with Echo

Next, the usual 9 ms delay is implemented, the bit counter is decremented, and the loop is entered again—as long as the eight bits have not been captured:

```
BSR     DELAY9
DEC     COUNTR
BNE     NEXT
```

Finally, the STOP bit is captured, and echoed. It is usually sufficient to send a single STOP bit; however, both could be sent back by using two more instructions:

```
LDA     >TTYBIT
STA     >TTYBIT
BSR     DELAY9
RTS
```

The logic of this program is quite simple: whenever a bit is read from the Teletype (at address TTYBIT), it is echoed back to the Teletype. This is a standard feature of the Teletype. Whenever a user presses a key, the information is transmitted to the processor and then back to the printing mechanism of the Teletype. This verifies that the transmission lines are working and that the processor is operating when a character is, indeed, printed correctly on the paper.

Using the above program, we will now write a PRINTC program that will print the contents of memory location CHAR on the Teletype.

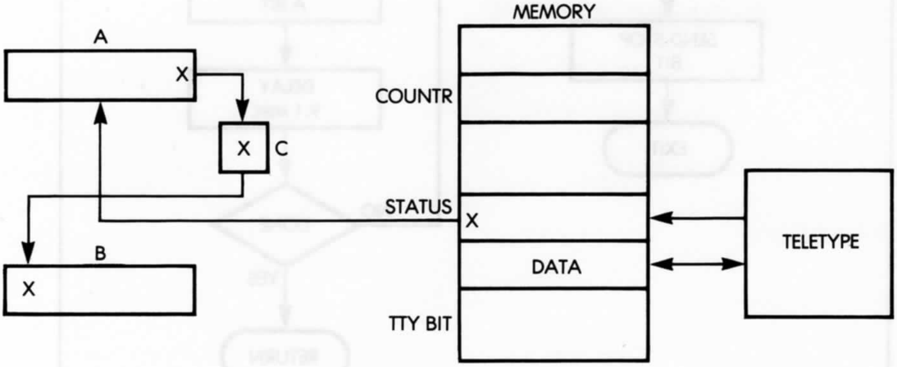


Figure 6.15: Teletype Input

Figure 6.16 shows the relevant flowcharts. Here is the program:

```
PRINTC  LDB     #11        COUNTER = 11 BITS
        CLRA               CLEAR CARRY = START BIT
        LDA     CHAR       GET CHARACTER
        ROLA               CARRY BIT INTO A
NEXT    STA     >TTYBIT    OUTPUT BIT
        BSR     DELAY9
        RORA               NEXT BIT
        ORCC    #$01       SET CARRY BIT
        DECB               BIT COUNT
        BNE     NEXT
        RTS
```

The B register is used as a bit counter for the transmission. The contents of bit 0 of register A are sent to the Teletype line (TTYBIT). Note how the carry is used to provide a ninth bit (the START bit). Also, note that the

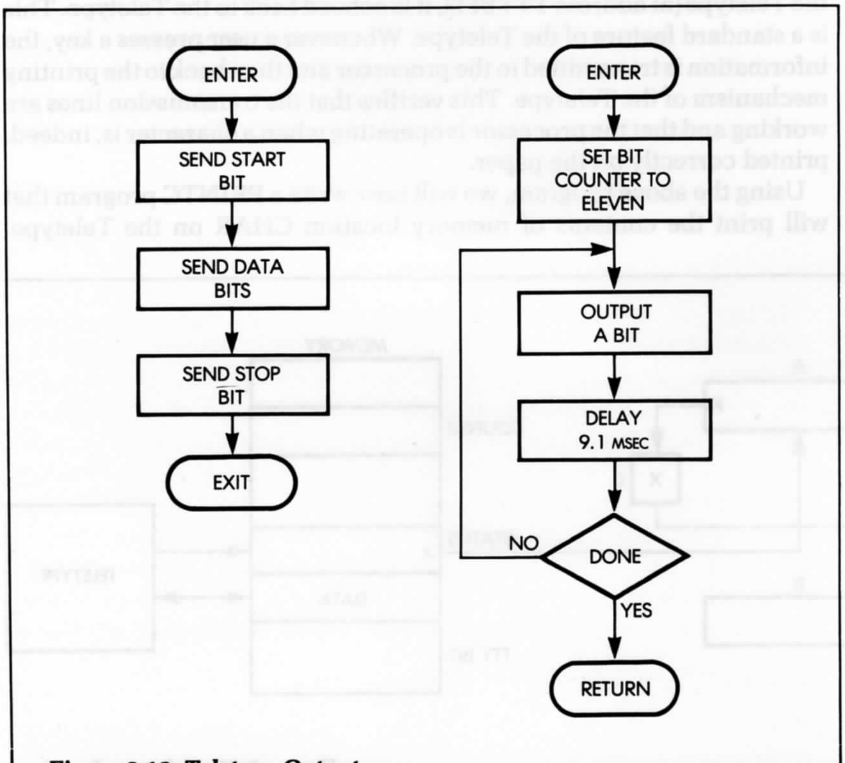


Figure 6.16: Teletype Output

carry is cleared by:

```
        CLRA
```

At the end of the program, the carry is set to 1 to generate a stop bit:

```
        ORCC    #$01
```

Let's now print a string of characters.

Printing a String of Characters

We will assume that the PRINTC routine prints a character on the printer, the display, or any serial output device. Let's now print the contents of memory locations START to (START) + N. Figure 6.17 shows the memory and registers used. Here is the program:

```
PSTRING LDB     #NBR        LENGTH OF STRING
        LDX     START       BASE ADDRESS
        PSHS    B           SAVE B PRINTC DESTROYS
NEXT    LDA     ,X+         GET CHARACTER
        STA     CHAR        PUT IT WHERE PRINTC WANTS IT
        BSR     PRINTC      PRINT IT
        PULS    B           GET COUNT BACK
        DEC     B
        BNE     NEXT        DO IT AGAIN
        RTS
```

PERIPHERAL SUMMARY

We have now described the basic programming techniques used to communicate with typical input/output devices. In addition to the data transfer, it is necessary to condition one or more control registers within each I/O device, in order to correctly condition the transfer speeds, the interrupt mechanism, and various other options. Consult the user's manual to obtain the appropriate information for each device. (See reference C207 in the bibliography for more details on the specific algorithms for exchanging information with all the usual peripherals.)

We have now learned to manage single devices. However, in a real system, all peripherals are connected to the buses and may request service simultaneously. How can we then schedule the processor's time?

INPUT/OUTPUT SCHEDULING

Since input/output requests may occur simultaneously, it is necessary to implement a scheduling mechanism in every system, to determine the order that service will be granted. Three basic input/output techniques are used: polling, interrupt, and DMA. Figure 6.18 illustrates these three techniques. The techniques can all be combined with each other. We will now describe polling and interrupts. Since DMA is a hardware technique, we will not describe it here. (See references C201A and C207 in the bibliography for further information on DMA.)

Polling

Conceptually, polling is the simplest method for managing multiple peripherals. With this strategy, the processor interrogates, in turn, each device that is connected to the buses. If a device requests service, the service is granted. If it does not, the next peripheral is examined. Polling is used not only for devices, but for *any device service routine.*

As an example, if the system is equipped with a Teletype, a tape recorder, and a CRT display, the polling routine would interrogate the Teletype: "Do you have a character to transmit?" It would also interrogate the Teletype *output routine:* "Do you have a character to send?" Then, assuming that the answers are negative, it would interrogate the tape-recorder routines, and finally, the CRT display. Even if only one device is connected to a system, polling would be used to determine whether it needs service. As examples of polling, Figures 6.19 and 6.20 show the

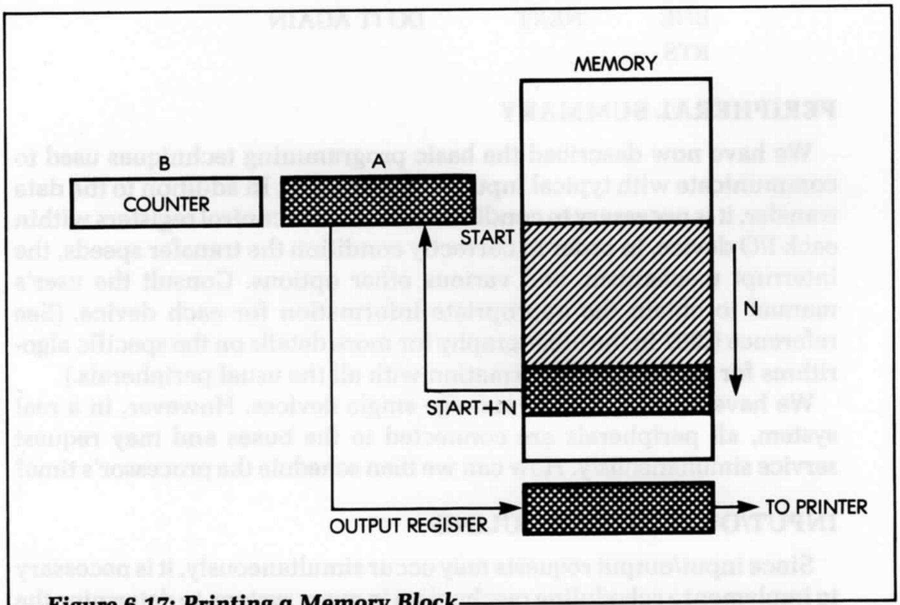


Figure 6.17: Printing a Memory Block

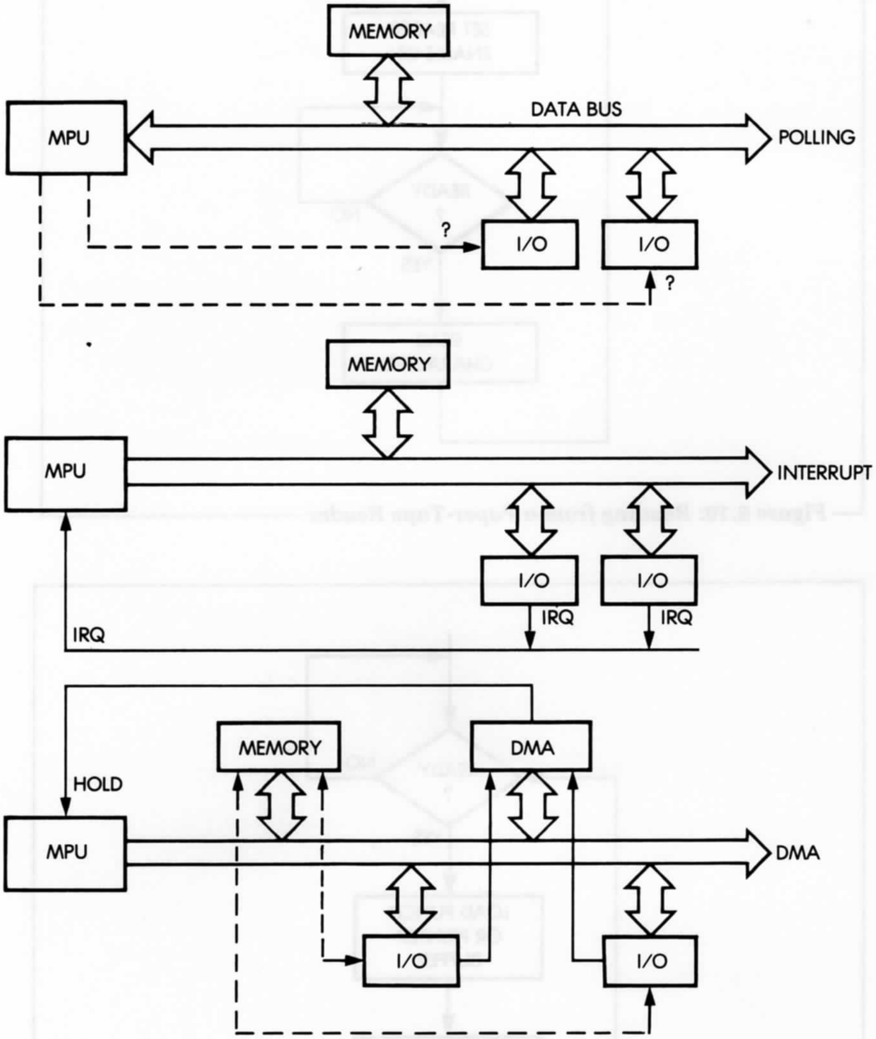


Figure 6.18: Three Methods of I/O Control

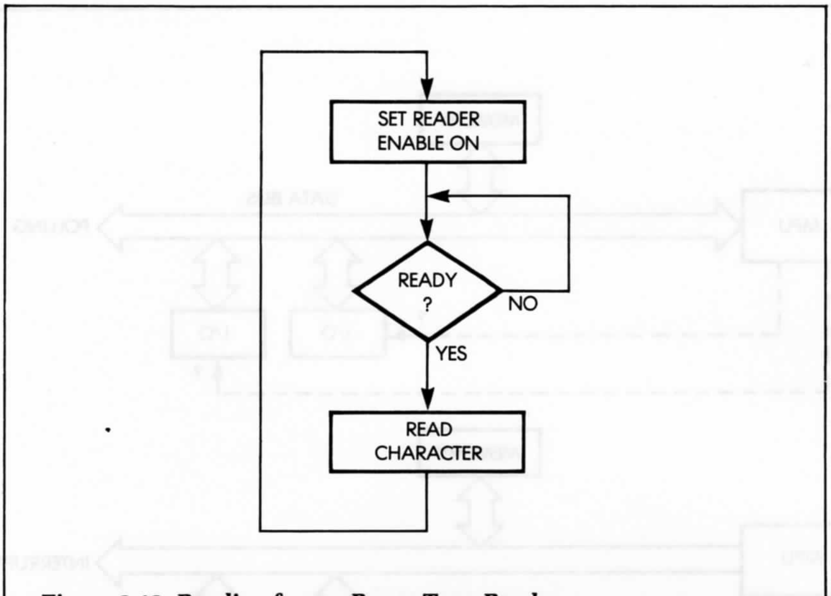


Figure 6.19: Reading from a Paper-Tape Reader

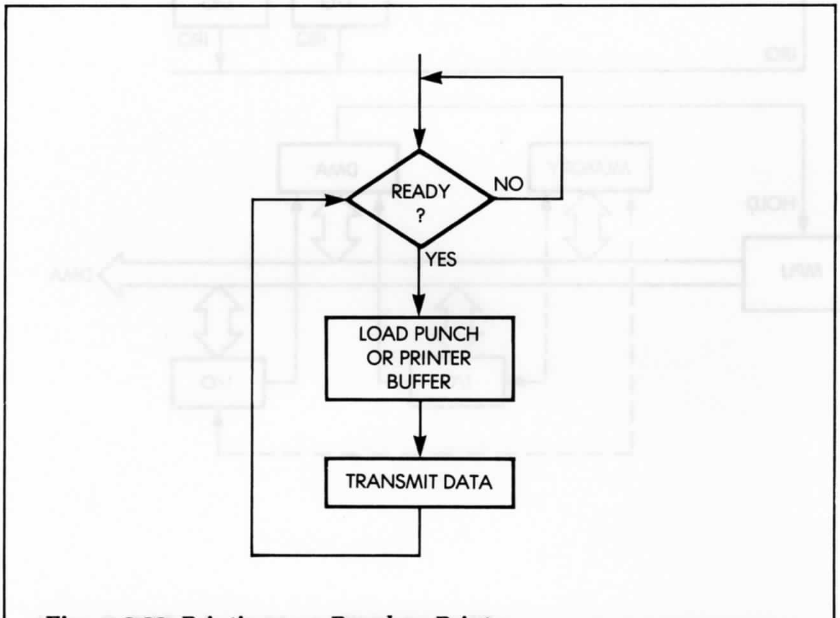


Figure 6.20: Printing on a Punch or Printer

flowcharts for reading a paper-tape reader and printing on a printer. Figure 6.21 shows a polling loop flowchart for 3 devices.

A program for a polling loop of four devices follows. The devices are

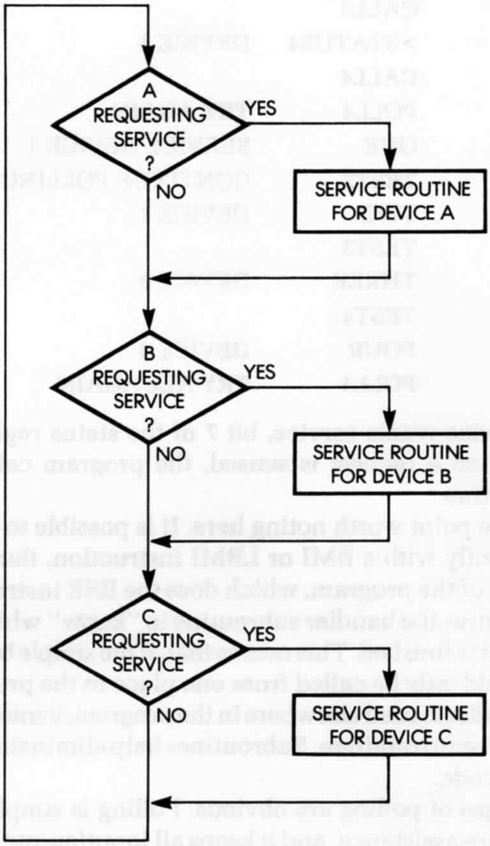


Figure 6.21: Polling Loop Flowchart

called 1, 2, 3, and 4:

```
POLL4   LDA   >STATUS1   GET STATUS OF DEVICE 1
        BMI   CALL1      SERVICE REQUEST
TEST2   LDA   >STATUS2   DEVICE 2
        BMI   CALL2
TEST3   LDA   >STATUS3   DEVICE 3
        BMI   CALL3
TEST4   LDA   >STATUS4   DEVICE 4
        BMI   CALL4
        BR    POLL4      TRY AGAIN
CALL1   BSR   ONE        SERVICE DEVICE 1
        BR    TEST2      CONTINUE POLLING
CALL2   BSR   TWO        DEVICE 2
        BR    TEST3
CALL3   BSR   THREE      DEVICE 3
        BR    TEST4
CALL4   BSR   FOUR       DEVICE 4
        BR    POLL4      TRY ALL AGAIN
```

When the device wants service, bit 7 of the status register for each device is 1. When a request is sensed, the program calls the device handler subroutine.

There is a fine point worth noting here. It is possible to branch to the subroutine directly with a BMI or LBMI instruction, thus eliminating the second part of the program, which does the BSR instruction. Use of the branch requires the handler subroutine to "know" which address to return to when it is finished. This means that, if the simple branch is used, the handler could only be called from one place in the program and no other. If the handler is used elsewhere in the program, it must be rewritten with a different return address. Subroutines help eliminate unnecessary duplication of code.

The advantages of polling are obvious. Polling is simple. It does not require hardware assistance, and it keeps all input/output synchronous with the program operation. The disadvantages are just as obvious. Most of the processor's time is wasted looking at devices that do not need service. In addition, by wasting so much time, the processor might then be late in giving service to a device.

Another mechanism is, therefore, desirable in order to guarantee that the processor's time is used for performing useful computations, rather than the needless continuous polling of devices. However, let us stress that polling is used extensively whenever a microprocessor has nothing better to do, as it keeps the overall organization simple. Let's examine an essential alternative to polling: interrupts.

Interrupts

Figure 6.18 illustrates the concept of interrupts. A special hardware line, the interrupt line is connected to a specialized pin of the microprocessor. Multiple input/output devices may be connected to this interrupt line. Then, when any one of them needs service, it sends a level or pulse on this line. An interrupt signal is the service request from an input/output device to the processor. Let's examine the response of the processor to this interrupt.

In all cases, when an interrupt occurs, the processor completes the instruction that it is currently executing (otherwise, such an interruption would create chaos inside the microprocessor). Next, the microprocessor branches to an interrupt-handling routine, which processes the interrupt. Branching to this subroutine implies that the contents of the program counter must be saved on the stack. *An interrupt must, therefore, cause the automatic preservation of the program counter on the stack.* In addition, the condition code register, CC, should also be preserved automatically, as its contents will be altered by any subsequent instruction. Finally, if the interrupt-handling routine should modify any internal registers, these internal registers should also be preserved on the stack (see Figures 6.22 and 6.23).

6809 Interrupts

An *interrupt* is a signal sent to the microprocessor, which may request service at any time. This signal is asynchronous to the program. Whenever a program branches to a subroutine, such branching is *synchronous* to program execution, i.e., scheduled by the program. An interrupt, however, can occur at any time, and it generally suspends the

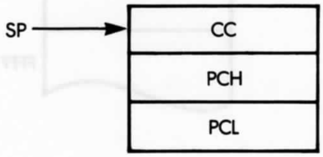


Figure 6.22: 6809 Stack After Interruption

execution of the current program (without the program knowing it). Because it may happen at any time relative to program execution, it is called *asynchronous.*

Four interruption mechanisms are provided on the 6809:

1. the bus request (DMA/BREQ)
2. the non-maskable interrupt (NMI)
3. the fast interrupt request (FIRQ)
4. the usual interrupt request (IRQ).

Let's examine them.

The Bus Request

The bus request is the highest priority interrupt mechanism on the 6809. As a general rule, no interrupt will be sensed by the 6809 until the current machine cycle is finished; and the NMI, FIRQ, and IRQ interrupts will not be taken into account until the current instruction is finished. The DMA/BREQ (TSC on the MC6809E), however, will be handled at the end of the current machine cycle, without necessarily waiting for the end of the instruction. It is used for a direct memory access (DMA), and causes the 6809 to go into DMA mode.

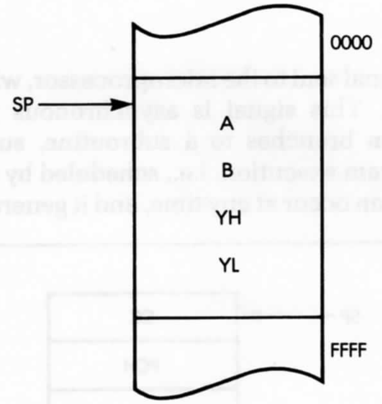


Figure 6.23: Saving Some Working Registers

In DMA mode, the 6809 suspends operation and releases its data-bus and address-bus in the high-impedance state. This mode is normally used by a DMA controller to perform transfers between a high speed input/output device and memory, using the microprocessor address-bus and data-bus. The end of a DMA operation is indicated to the 6809 by DMA/BREQ changing levels. At this point, the 6809 will resume normal operation.

The DMA should normally not be of concern to the programmer, unless timing is important. If a DMA controller is present in the system, the programmer must understand that the DMA may delay the response to one of the other three interrupts.

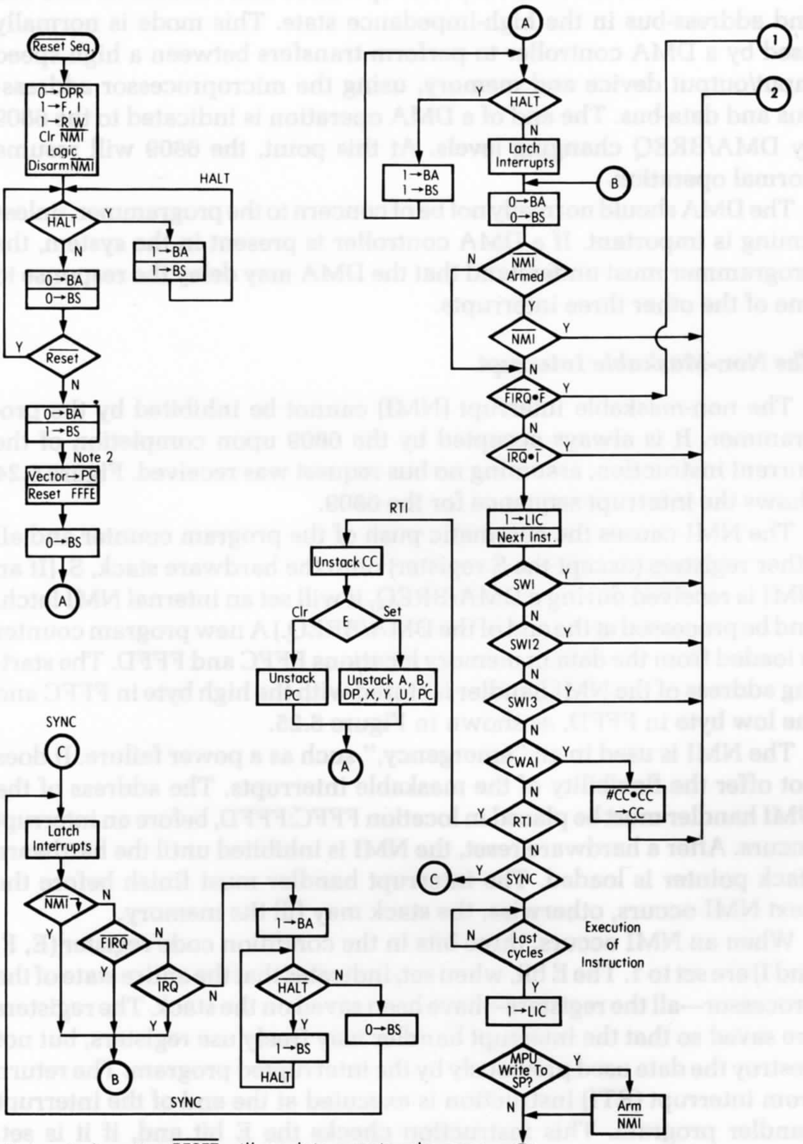
The Non-Maskable Interrupt

The non-maskable interrupt (NMI) cannot be inhibited by the programmer. It is always accepted by the 6809 upon completion of the current instruction, assuming no bus request was received. Figure 6.24 shows the interrupt sequence for the 6809.

The NMI causes the automatic push of the program counter and all other registers (except the S register) onto the hardware stack, S. (If an NMI is received during a DMA/BREQ, it will set an internal NMI latch, and be processed at the end of the DMA/BREQ.) A new program counter is loaded from the data in memory locations FFFC and FFFD. The starting address of the NMI handler is stored with the high byte in FFFC and the low byte in FFFD, as shown in Figure 6.25.

The NMI is used in an "emergency," such as a power failure. It does not offer the flexibility of the maskable interrupts. The address of the NMI handler must be placed in location FFFC:FFFD, before an interrupt occurs. After a hardware reset, the NMI is inhibited until the hardware stack pointer is loaded. The interrupt handler must finish before the next NMI occurs, otherwise, the stack may fill the memory.

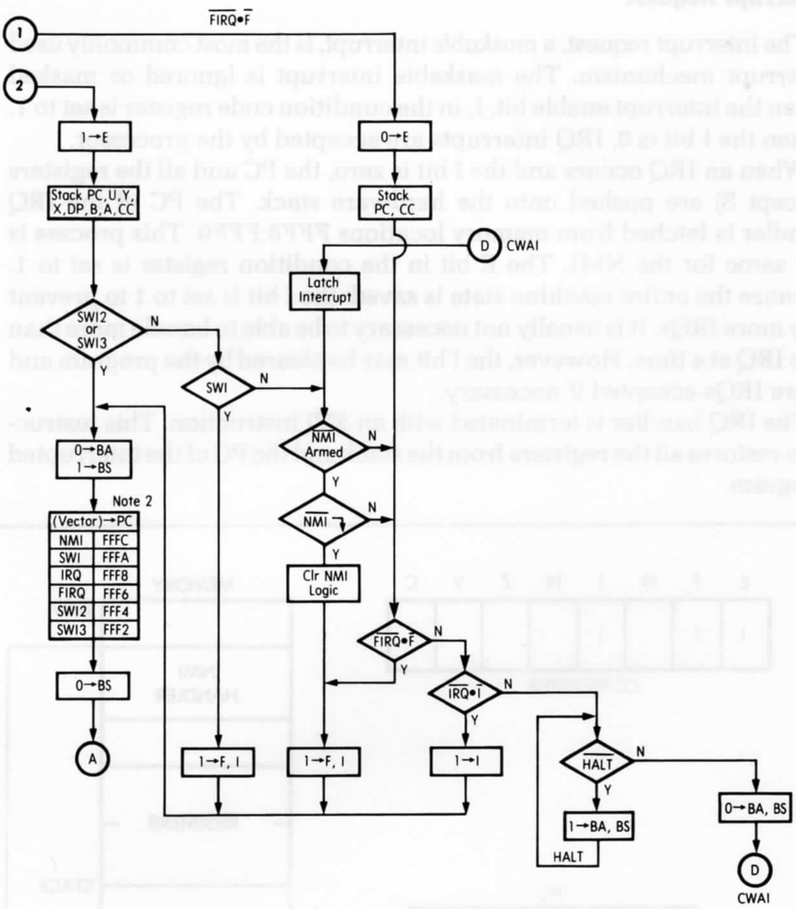
When an NMI occurs, three bits in the condition code register (E, F, and I) are set to 1. The E bit, when set, indicates that the entire state of the processor—all the registers—have been saved on the stack. The registers are saved so that the interrupt handler may freely use registers, but not destroy the data used previously by the interrupted program. The return from interrupt (RTI) instruction is executed at the end of the interrupt handler program. This instruction checks the E bit and, if it is set, restores all the registers and the PC of the interrupted program. If the E bit is clear when the RTI is executed, only the condition code register and the PC will be restored from the stack. The I and F bits enable, or disable (when they are 0 or 1) the IRQ and FIRQ interrupts.



Notes: 1. Asserting RESET will result in entering the reset sequence from any point in the flow chart.
2. BUSY is high during first vector fetch cycle.

Courtesy of Motorola, Inc.

Figure 6.24: Flowchart for MC6809E Interrupts and Instructions



Bus State	BA	BS
Running	0	0
Interrupt or Reset Acknowledge	0	1
Sync Acknowledge	1	0
Halt Acknowledge	1	1

Courtesy of Motorola, Inc.

Figure 6.24: Flowchart for MC6809E Interrupts and Instructions (cont.)

Interrupt Request

The interrupt request, a *maskable* interrupt, is the most commonly used interrupt mechanism. The maskable interrupt is ignored or masked when the interrupt enable bit, I, in the condition code register is set to 1. When the I bit is 0, IRQ interrupts are accepted by the processor.

When an IRQ occurs and the I bit is zero, the PC and all the registers (except S) are pushed onto the hardware stack. The PC of the IRQ handler is fetched from memory locations FFF8:FFF9. This process is the same for the NMI. The E bit in the condition register is set to 1, because the entire machine state is saved; the I bit is set to 1 to prevent any more IRQs. It is usually not necessary to be able to handle more than one IRQ at a time. However, the I bit may be cleared by the program and more IRQs accepted if necessary.

The IRQ handler is terminated with an RTI instruction. This instruction restores all the registers from the stack and the PC of the interrupted program.

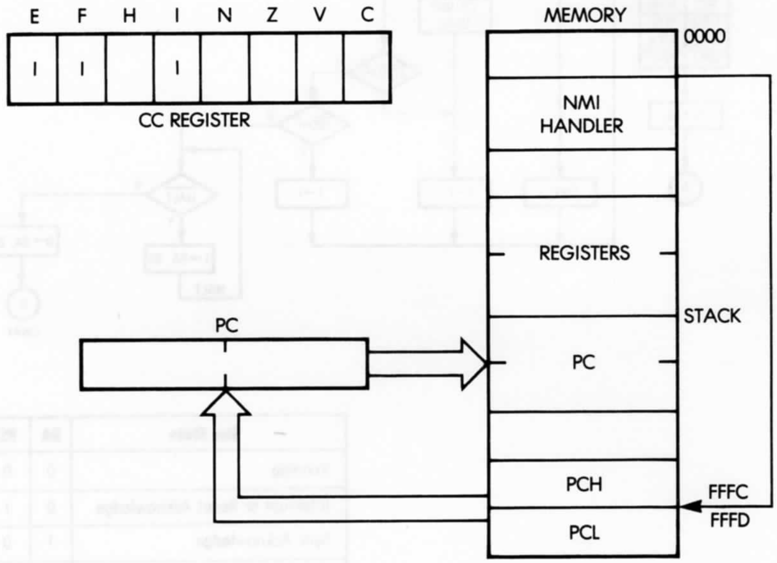


Figure 6.25: Non-Maskable Interrupt Sequence

Fast Interrupt Request

The *fast interrupt request* is similar to the IRQ, as it is maskable by setting the F bit in the condition code register to 1. When an FIRQ is received, only the PC and condition code register are saved on the hardware stack. The E bit is not set, because the entire machine state has not been saved. The PC for the FIRQ handler is fetched from locations FFF6:FFF7. Both the F and I bits are set to 1 to prevent any more interrupts.

The fast interrupt request executes much more quickly than the NMI or IRQ, because only three bytes are pushed onto the stack. The FIRQ takes ten cycles to execute. The NMI and IRQ require nineteen. The fast interrupt request is very useful when speed is essential, but the registers are not used extensively. If a register is used, it must first be pushed and then pulled, before execution of the RTI instruction. The RTI restores the condition code register and the PC of the interrupted program.

Interrupt Dependent Instructions

Two instructions on the 6809 depend on interrupts. They are the *synchronize to external event* (SYNC) instruction and the *clear condition code bits and wait for interrupt* instruction (CWAI).

The SYNC instruction stops the 6809 from processing until an interrupt occurs. It also sets the bus available (BA) pin on the 6809 chip to 1 and the bus status (BS) pin to 0. This is the sync acknowledge state of the processor. If the mask bit for that interrupt is 0, the interrupt handler is executed. If the interrupt is not enabled, execution of the program proceeds immediately after the SYNC instruction. This instruction can be useful for very fast I/O from a device.

The CWAI instruction ANDs the byte immediately following the instruction with the condition code register, saves all of the processor registers on the stack, and suspends program execution until an interrupt occurs. If the interrupt is not masked, the interrupt handler is executed. Otherwise the processor stays in a suspended state. This instruction is provided for compatability with the MC6800 microprocessor.

Interrupt Overhead

Figure 6.18 gives a graphic comparison of the polling process versus the interrupt process—the polling process is illustrated on top, and the interrupt process below. It can be seen in the illustration that the program wastes a lot of time waiting in the polling technique.

When using interrupts, the program is interrupted, the interrupt is serviced, and the program resumes. However, an obvious disadvantage of an interrupt is that it introduces several additional instructions at the beginning and end of the device handler program, thus resulting in a delay before execution of the first instruction of the device handler. This delay is additional overhead.

Now that we have clarified the operation of the interrupt lines, let's consider two remaining problems, involving:

- multiple devices triggering an interrupt at the same time
- an interrupt occurring while another is being serviced.

Multiple Devices Connected to a Single Interrupt Line

Whenever an interrupt occurs, the processor branches to a specified address. Before it can do any effective processing, the interrupt handler must determine which device triggered the interrupt. A polling method can be used to find the device that interrupted the processor. The microprocessor asks each device in turn, "Did you trigger the interrupt?" If the answer is negative, it interrogates the next one. The following program illustrates this process:

```
POLINT   LDA    >STATUS1    READ STATUS
         BMI    ONE         HANDLE DEVICE IF IT
                            INTERRUPTED
         LDA    >STATUS2
         BMI    TWO
                . . .
```

Simultaneous Interrupts

A second problem is that a new interrupt may be triggered during the execution of an interrupt-handling routine. Let's examine what happens when this occurs, and see how the stack can solve this problem. We previously indicated that this was another essential role of the stack; the time has now come to demonstrate its use. The illustration in Figure 6.26 shows the concept of multiple interrupts.

The contents of the stack are shown at the bottom of the illustration. Time elapses from left to right. Looking at time T0 on the left, program P is executing. Moving to the right, at time T1, interrupt I1 occurs. We assume that the interrupt mask was enabled, thus authorizing I1. Program P is suspended, as shown at the bottom of the illustration. The stack contains, at the least, the program counter and the status register of program P, plus any optional registers that might be saved by the interrupt handler or I1 itself.

At time T1, interrupt I1 starts executing until time T2. At time T2, interrupt I2 occurs. We assume that interrupt I2 has a higher priority than interrupt I1. If it had a lower priority, it would be ignored until I1 was completed. At time T2, the registers for I1 are stacked (as shown at the bottom of the illustration). Again, the contents of the program counter and the condition code register are pushed onto the stack. In addition, the routine for I2 might decide to save additional registers. I2 executes to completion at time T3.

When I2 terminates, the contents of the stack are automatically popped back into the 6809 (as illustrated at the bottom of Figure 6.26). Thus, I1 resumes execution automatically. Unfortunately, at time T4, an interrupt I3 of higher priority occurs again. We can see at the bottom of the illustration that the registers for I1 are again pushed onto the stack. Interrupt I3 executes from T4 to T5 and terminates at T5. At that time, the contents of the stack are popped into the 6809, and interrupt I1 resumes execution. This time it runs to completion and terminates at T6. At T6, the remaining registers that have been saved in the stack are popped into the 6809, and program P can resume execution. At this point, we can verify that the stack is empty. In fact, the number of dashed lines indicating program suspension *also* indicate the number of levels in the stack.

We must stress here, however, that, in practice, microprocessor

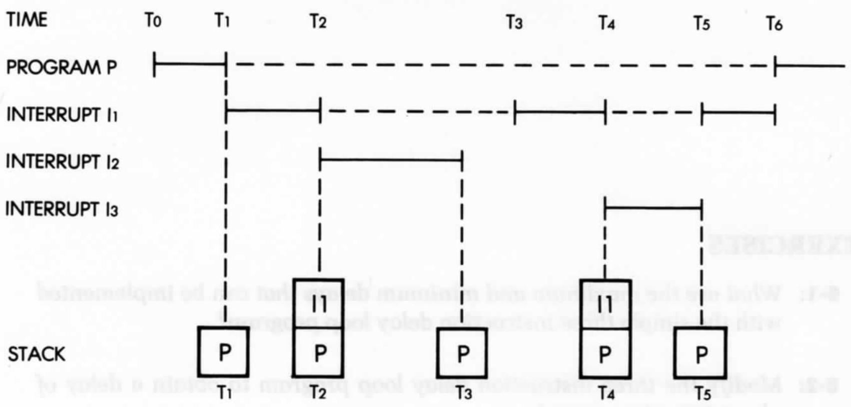


Figure 6.26: Stack Contents During Multiple Interrupts

systems are normally connected to a small number of devices that use interrupts. It is, therefore, unlikely that a high number of simultaneous interrupts will occur in such a system.

We have now solved all the problems usually associated with interrupts. Their use is, in fact, simple, and they should be used to advantage by even the novice programmer.

SUMMARY

In this chapter, we have presented programming techniques that can be used to communicate with the outside world. These techniques have ranged from elementary input/output routines to more complex programs for communication with actual peripherals. We have learned to develop all the usual programs and have even examined the efficiency of benchmark programs in the case of a parallel transfer and a parallel-to-serial conversion. Finally, we have learned to schedule the operation of multiple peripherals, using polling and interrupts.

Naturally, many exotic input/output devices may be connected to a system. With the array of techniques presented so far, and with an understanding of the peripherals involved, we should now be able to solve most common problems.

In the next chapter, we will examine the actual characteristics of the input/output interface chips usually connected to a 6809 processor. We will then discuss the basic data structures available for use.

EXERCISES

6-1: *What are the maximum and minimum delays that can be implemented with the simple three instruction delay loop program?*

6-2: *Modify the three instruction delay loop program to obtain a delay of about 100 microseconds.*

6-3: *Write a program to implement a 100 ms delay (typical of a Teletype).*

6-4: *Assume that the number of words to be transferred to memory is greater than 256. Modify the parallel word transfer program accordingly, and determine the impact on the maximum data transfer rate.*

6-5: *Compute the maximum speed at which the serial bit transfer program will be able to read serial bits. Look up in the appendix the number of cycles required by every instruction in the table, then compute the time that will elapse during execution of this program. To compute the length of time used by a loop, simply multiply the total duration of this loop, expressed in microseconds, by the number of times it will be executed. Also, when computing the maximum speed, assume that a data bit will be ready each time the input location is sensed.*

6-6: *Can you explain why bit 7 is used for status and bit 0 for data in the bit serial transfer program? Does it matter?*

6-7: *Modify the bit serial transfer program, assuming that the first bit to come in is valid data (not to be discarded), and that it can be 0 or 1. (Hint: our bit counter should still work correctly, if you initialize it with the correct value.)*

6-8: *Modify the bit serial transfer program to save the assembled word in the memory area starting at BASE.*

6-9: *Modify the bit serial transfer program so that the transfer stops when the S character is detected in the input stream.*

6-10: *Modify the bit serial transfer program, assuming that data is available in bit position 0 of location INPUT, while the status information is available in bit position 0 of address INPUT + 1.*

6-11: *When using an actual printer, it is usually necessary to send a start order before using the device. Modify the printer program to generate such an order, assuming that the start command is obtained by writing a 1 in bit position 0 of the STATUS register, which is assumed to be bidirectional.*

6-12: *Modify the printer program to print a string of n characters, where n is assumed to be less than 255.*

6-13: *Modify the printer program to print a string of characters, until a carriage-return code is encountered.*

6-14: *It is usually necessary to turn off the segment drivers for an LED, prior to displaying new digits. Modify the LED program by adding the necessary* *instructions (output 00 as the character code, prior to outputting the character).*

6-15: *What would happen to the LED display if the DELAY label in the LED program was moved up by one line position? Would this change the timing? Would it change the appearance of the display?*

6-16: *Assuming that the LED program is a subroutine, notice that it uses the register X internally and modifies its contents. If the subroutine freely uses the memory area designated by SAVEX, can you add instructions at the beginning and end of this program which guarantee that, when the subroutine returns, the contents of the register X will be the same as when the subroutine was entered?*

6-17: *Same exercise as above, but assume that the memory area SAVEX, etc., is not available to the subroutine. (Hint: remember that there is a built-in mechanism in every computer for preserving information in chronological order.)*

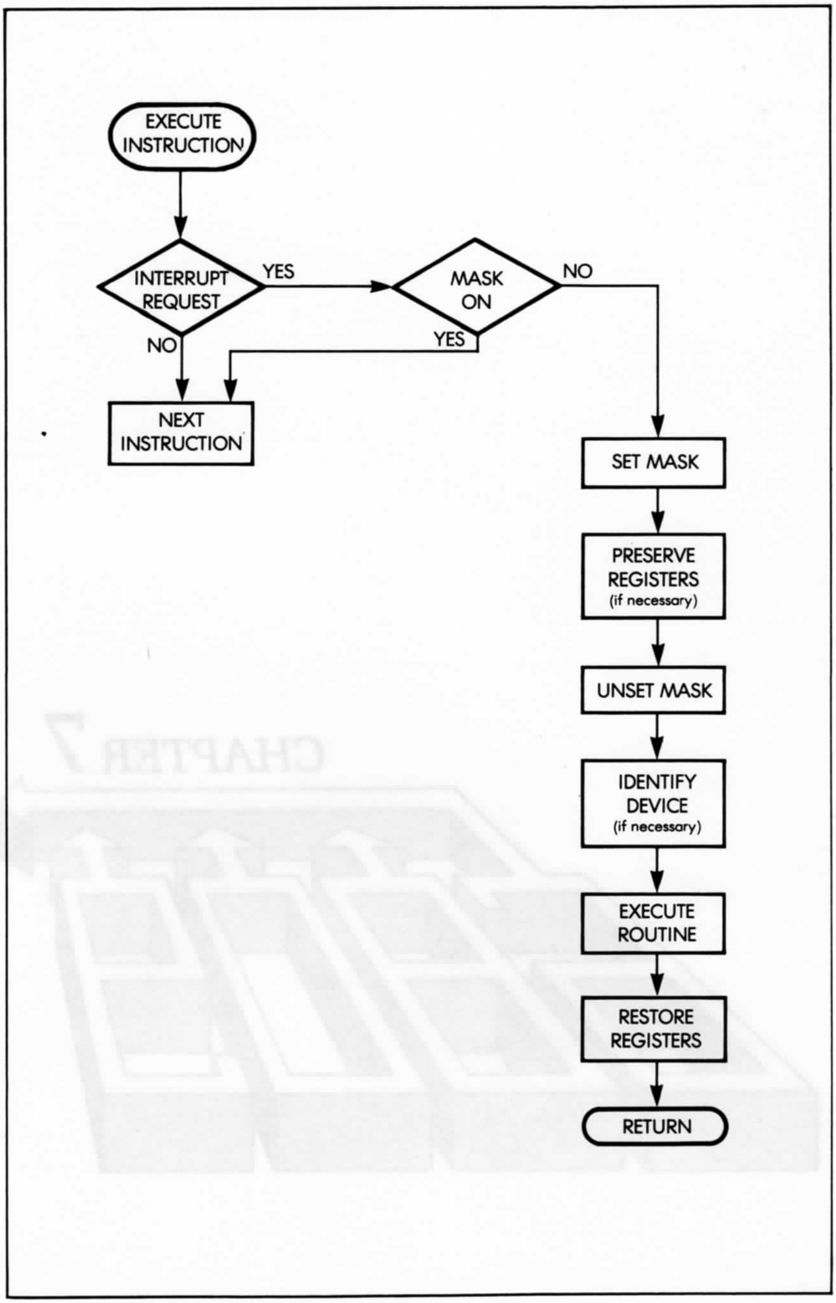
6-18: *Write the delay routine which results in the 9.09 millisecond delay. (DELAY9 subroutine.)*

6-19: *Assume that the area available to the stack is limited to 300 locations in a specific program. Also, assume that all the registers must always be saved and that the programmer allows interrupts to be nested, i.e., to interrupt each other. What is the maximum number of simultaneous interrupts that can be handled? Will any other factor contribute to reducing further the maximum number of simultaneous interrupts?*

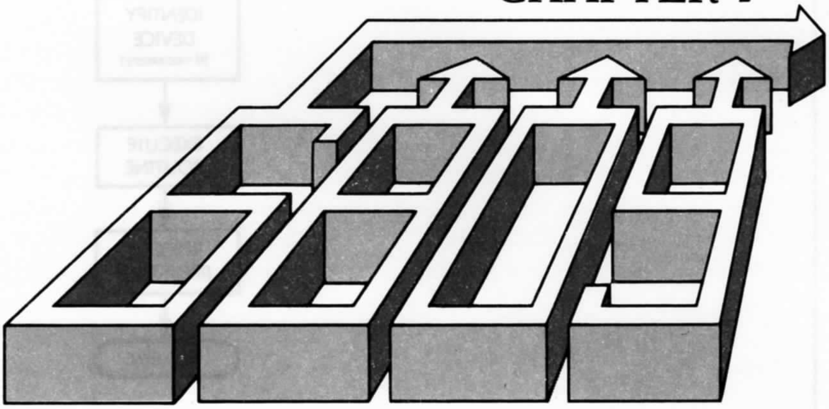
6-20: *A 7-segment LED display can also display digits other than the hex alphabet. Compute the codes for: H, I, J, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, r, t, u, y.*

6-21: *The flowchart for interrupt management appears on the next page. Answer the following questions:*

a. What is done by hardware? What is done by software?
b. What is the use of the mask?
c. How many registers should be preserved?
d. How is the interrupting device identified?
e. What does the RTI instruction do? How does it differ from a subroutine return?
f. Suggest a way to handle a stack overflow situation.
g. What is the overhead ("lost time") introduced by the interrupt mechanism?


EXECUTE INSTRUCTION
INTERRUPT REQUEST
YES
NO
MASK ON
NO
YES
NEXT INSTRUCTION
SET MASK
PRESERVE REGISTERS (if necessary)
UNSET MASK
IDENTIFY DEVICE (if necessary)
EXECUTE ROUTINE
RESTORE REGISTERS
RETURN

CHAPTER 7



INPUT/OUTPUT DEVICES

WITH THE PROGRESS OF LSI, more and more elaborate input/output chips have been developed. As a result, the task of programming a system includes not only programming the microprocessor itself, but also programming the *input/output chips.* In fact, it is often more difficult to remember how to program the various control options of an input/output chip than it is to program the microprocessor itself. This is not because the programming is more difficult, but because each device has its own idiosyncrasies. In this chapter, we will examine the most general input/output device—the programmable input/output chip (the PIO). We will also examine some input/output devices designed by Motorola.

The 6809 was designed to provide 16-bit microprocessor capability, while interfacing easily with any of the extensive 68xx family of I/O chips developed for 8-bit processors. The 6809 will also interface with most 6502 I/O devices, such as those used in the Apple, Atari, Commodore, and many other personal computers.

THE "STANDARD" PIO

A *PIO* provides a multi-port connection for input/output devices. (A port is a set of 8 input/output lines.) At the very least, each input/output device needs a *data buffer* to stabilize the contents of the data bus on output. Most PIO's are equipped with a buffer for each port. Although there is no "standard" PIO, most manufacturers produce PIOs that are similar in function.

We previously established that a microcomputer uses a *handshaking* procedure, or *interrupts* to communicate with an I/O device. The PIO also uses a similar procedure to communicate with a peripheral. Therefore, to implement a handshaking function, each PIO must be equipped with at least *two control lines per port.*

A microprocessor also needs to read the status of each port. Thus, each port must be equipped with one or more *status bits.* In addition, the PIO has a number of options for configuring its resources. To specify these programming options, a programmer must be able to access a special register in the PIO, called the *control register.* In some cases, the status information is part of the control register.

One essential faculty of the PIO is that each line may be configured as *either* an input or output line. Figure 7.1 shows a diagram of a PIO. It is up to the programmer to specify whether a line will be input or output. To program the direction of the lines, a *data-direction register* is provided for each port. A 0 in a bit position of the data-direction register specifies an input. A 1 specifies an output.

It may be surprising that a 0 is used for input and a 1 for output, when usually a 0 corresponds to output and a 1 to input. However, this change is quite deliberate: whenever power is applied to the system, it is important that all the I/O lines are configured as *input.* Otherwise, if the microcomputer is connected to some dangerous peripheral, it may be activated by accident. When a reset is applied, all registers are normally zeroed, which results in configuring all input lines of the PIO as inputs. The connection to the microprocessor appears on the left of the illustration in Figure 7.1. The PIO connects to the 8-bit data bus, the microprocessor address bus, and the microprocessor control-bus. The programmer simply specifies the address of any register to be accessed within the PIO.

THE INTERNAL CONTROL REGISTER

The control register of the PIO provides a number of options for generating or sensing interrupts, or for implementing automatic handshake functions. We will not provide a complete description of these facilities here. However, very simply, when using a practical system that uses a PIO, it is usually necessary to refer to the data-sheet showing the effects of setting the various bits of the control register. When the system is initialized, the programmer must load the control register of the PIO with the correct contents for the expected application.

PROGRAMMING A PIO

Let's now look at a typical sequence, using a PIO channel (assuming an input):

1. *Load the control register* by using a programmed transfer between a 6809 register (usually an accumulator) and the PIO

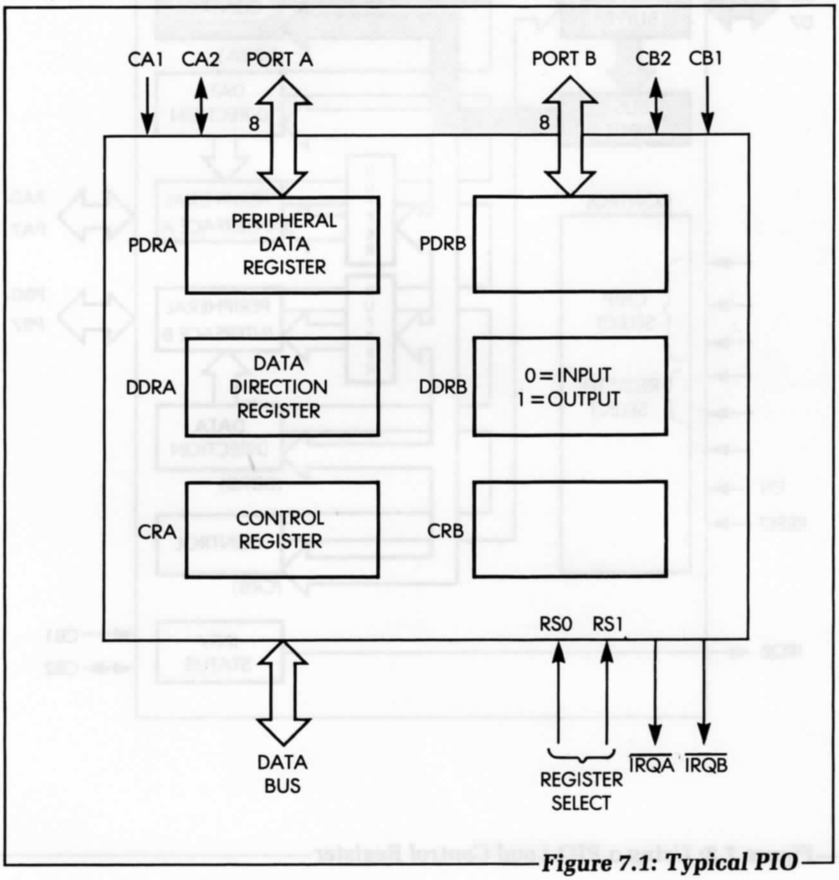


Figure 7.1: Typical PIO

control register. The options and operating mode of the PIO are set when the register is loaded (see Figure 7.2). The loading is normally done only once, at the beginning of a program.

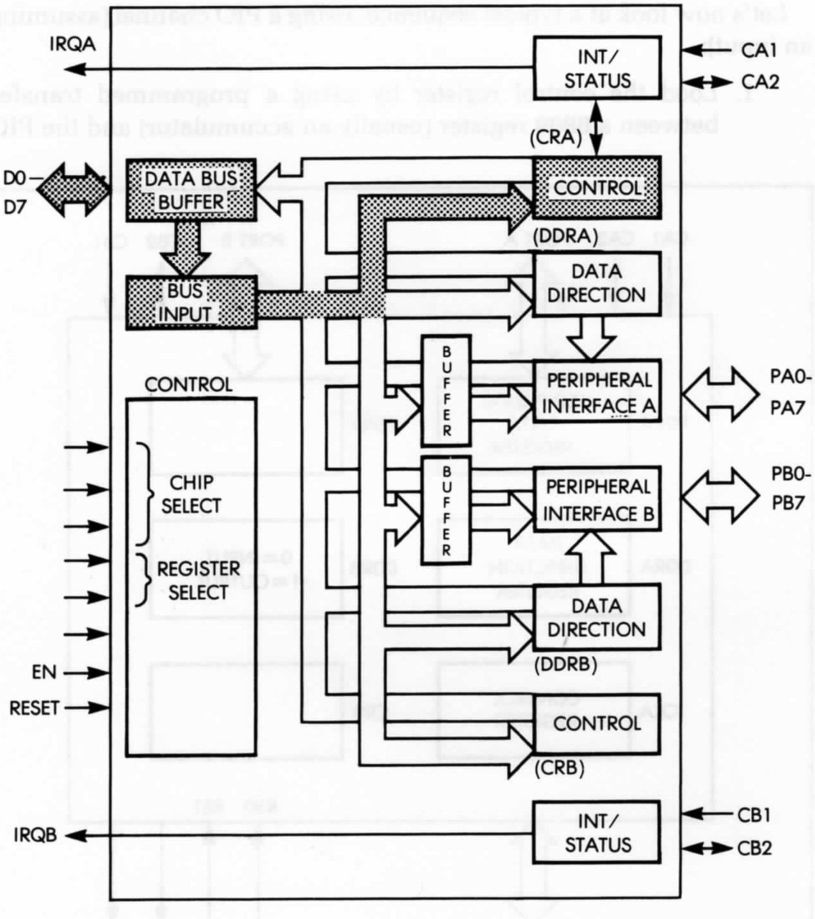


Figure 7.2: Using a PIO-Load Control Register

2. *Load the direction register* to specify the direction in which the I/O lines will be used. (See Figure 7.3.)

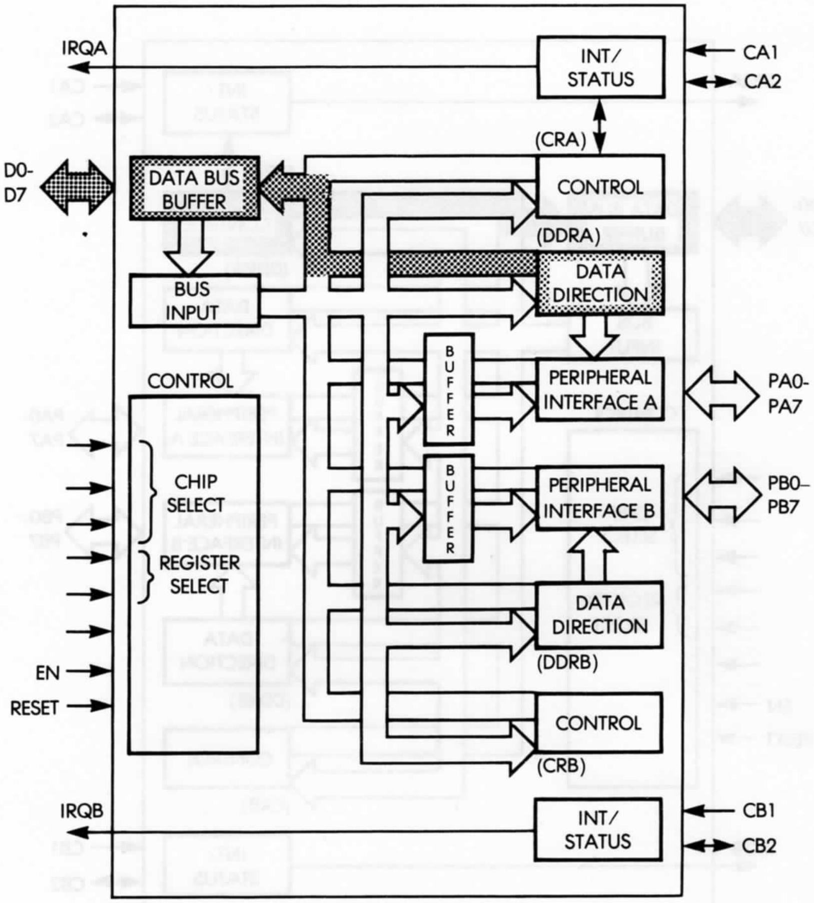


Figure 7.3: Using a PIO-Load Data Direction

3. *Read the status register* to check if a valid byte is available on input. (See Figure 7.4.)
4. *Read the port;* the byte is read into the 6809. (See Figure 7.5.)

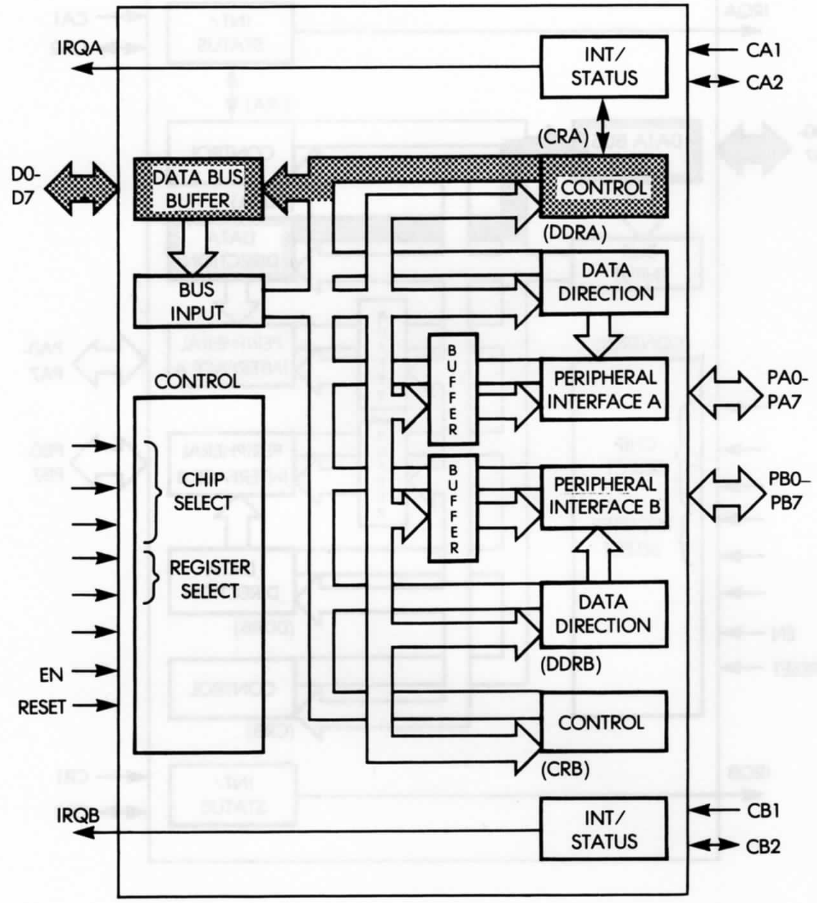


Figure 7.4: Using a PIO-Read Status

THE MOTOROLA 6821 PROGRAMMABLE INTERFACE ADAPTER

The 6821 PIA is a two-port PIO with an architecture that is essentially the same as the standard model we have just described. Figure 7.6 shows the actual pinout of a 6821.

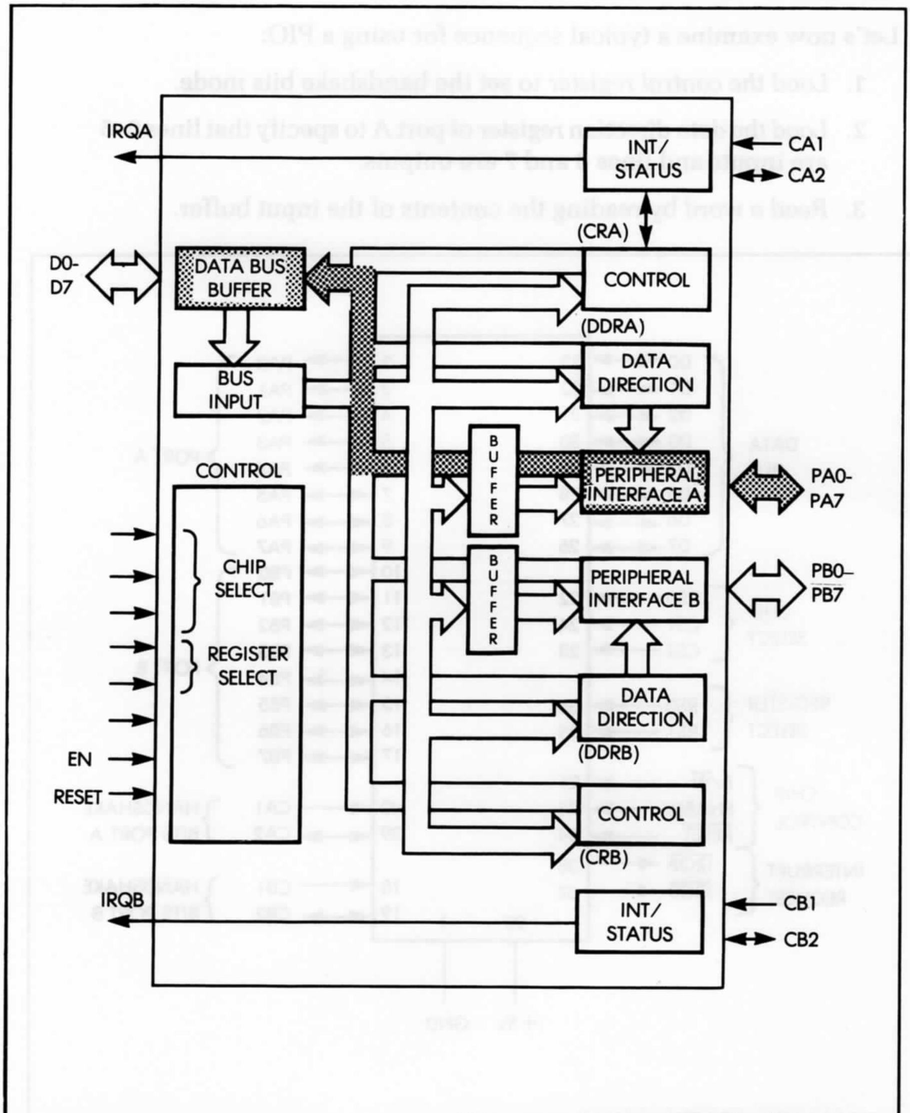


Figure 7.5: Using a PIO-Read INPUT

The control register for each port has bits which control the conditions in which an interrupt can be generated and the conditions when the handshake bits can change state.

PROGRAMMING THE MOTOROLA PIO

Let's now examine a typical sequence for using a PIO:

1. *Load the control register* to set the handshake bits mode.
2. *Load the data direction register* of port A to specify that lines 0–5 are inputs and lines 6 and 7 are outputs.
3. *Read a word* by reading the contents of the input buffer.

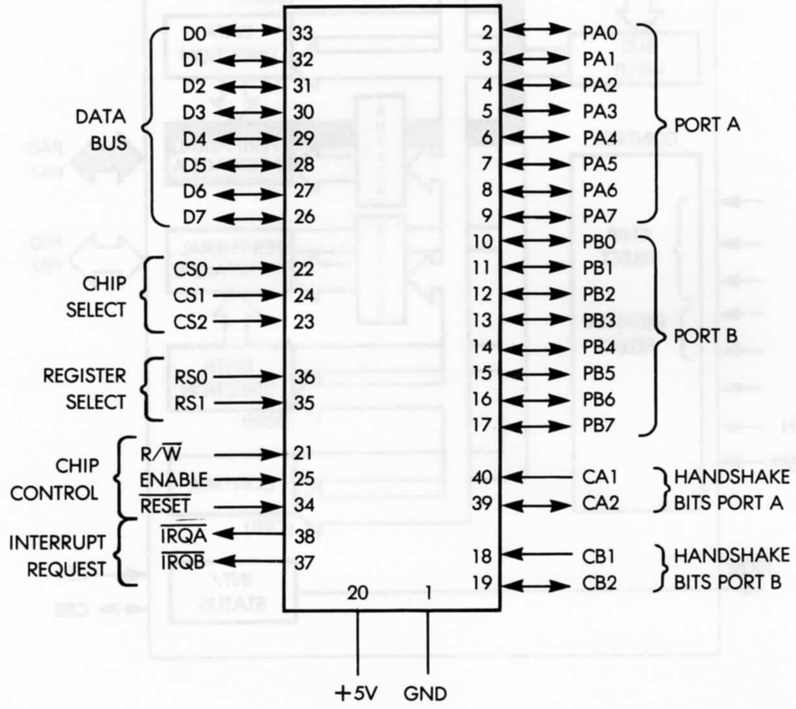


Figure 7.6: 6821 PIA Pinout

THE MC6850 ACIA FOR THE 6809

The MC6850 ACIA (Asynchronous Communications Interface Adapter) is a peripheral chip designed to facilitate asynchronous communications in serial form. It includes a universal asynchronous receiver-transmitter (a UART). The essential function of the ACIA is serial-to-parallel and parallel-to-serial conversion. The ACIA also offers a choice of data format and interrupt modes.

OTHER I/O CHIPS

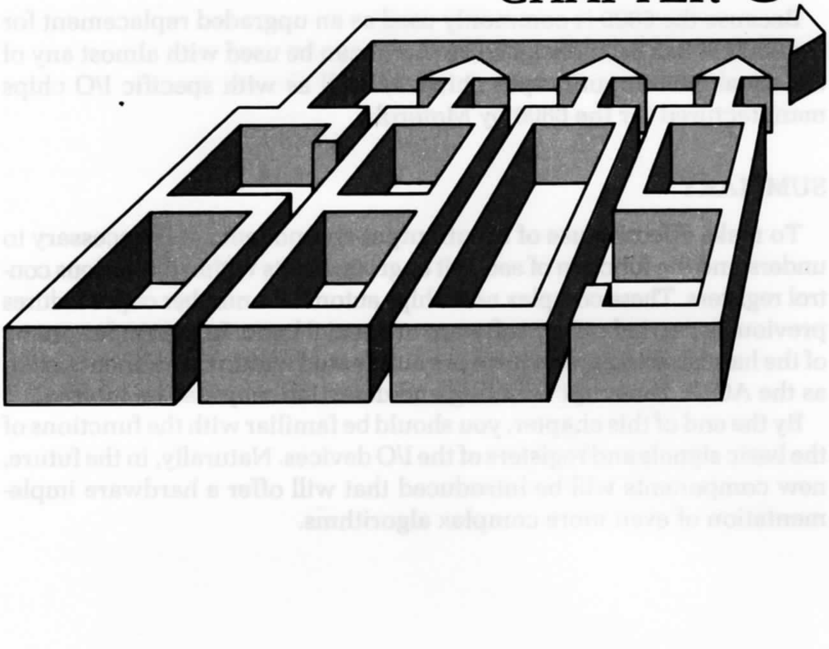
Because the 6809 is commonly used as an upgraded replacement for the 6800, it has been designed so that it can be used with almost any of the usual 6800 input/output chips, as well as with specific I/O chips manufactured for the 6809 by Motorola.

SUMMARY

To make effective use of input/output components, it is necessary to understand the function of each bit or group of bits within the various control registers. These complex new chips automate a number of procedures previously carried out by software or special logic. In particular, many of the handshaking procedures are automated within components, such as the ACIA. Interrupt handling and detection may also be internal.

By the end of this chapter, you should be familiar with the functions of the basic signals and registers of the I/O devices. Naturally, in the future, new components will be introduced that will offer a hardware implementation of even more complex algorithms.

CHAPTER 8



APPLICATION EXAMPLES

IN THIS CHAPTER, you can sharpen your new programming skills by developing a collection of utility programs that fetch characters from an I/O device and process them in various ways. These programs give your a chance to apply the knowledge and techniques you have learned so far, in the development of a number of routines that are useful in many applications. The development of these routines demonstrates how the architecture of the 6809 can make the programming of such common algorithms exceptionally straightforward.

Before we begin, we will clear an area of the memory in which we will put the characters from the I/O device. Clearing memory is not always necessary; we do it here as a programming example.

CLEARING A SECTION OF MEMORY

We will start by clearing (zeroing) the contents of the memory from address BASE to address BASE + LENGTH, where LENGTH is less than 256 bytes. The program is:

```
ZEROM    LDB    #LENGTH    LOAD B WITH LENGTH
         LDX    #BASE      POINT TO BASE
CLEAR    CLR    ,X+        CLEAR LOCATION AND POINT TO
                           NEXT
         DECB              DECREMENT COUNTER
         BNE    CLEAR      END OF SECTION?
         RTS
```

In this program, we assume that the length of the section of memory is equal to LENGTH. We use the index register, X, as a pointer to the current word to be cleared, and register B as a counter.

We could use this utility in a memory test program to zero the contents of a block. The memory test program would then verify that the contents of the block remain zero.

Let's now improve this routine:

```
ZEROM   LDB    #LENGTH
        LDX    #BASE
        CLRA                SET A TO ZERO
CLEAR   STA    ,X+
        DECB
        BNE    CLEAR
        RTS
```

We have improved the program by storing the A register, rather than by using the CLR instruction. When using the indexed addressing modes, the STA instruction requires 6 cycles, rather than the 8 required by CLR.

This example demonstrates that *every time a program is written, even though it may be correct, it can usually be improved.* It is necessary, however, to be familiar with the complete instruction set in order to implement such improvements. These improvements are not simply cosmetic; they can often improve the execution time of the program; they might also require fewer instructions and less memory space, and they may improve the readability of the program and, therefore, its chances of being correct.

GETTING CHARACTERS IN

We will now write a program that reads characters from an I/O device. Assuming that the computer we are using has a keyboard as an input device, each time we type a character, the character will be saved in an area of memory called the BUFFER, until a special character called SPACE is encountered. (Appendix B gives the code number for SPACE.) The subroutine GETCHAR fetches one character from the keyboard and puts it in the A accumulator. We assume that 256 characters (maximum) will be fetched before a SPACE character is encountered:

```
STRING  LDX    #BUFFER    POINT TO BUFFER
NEXT    JSR    GETCHAR    GET A CHARACTER
        CMPA   #SPC       CHECK FOR SPACE
        BEQ    OUT        FOUND IT?
        STA    ,X+        STORE CHAR IN BUFFER
        BRA    NEXT       GET NEXT CHAR
OUT     RTS
```

At the end of this routine, we have a string of characters in the memory buffer. We will now process them in various ways.

TESTING A CHARACTER

This program determines if the character at memory location LOC is equal to 0, 1, or 2:

```
ZOT     LDA     LOC      GET CHARACTER
        CMPA    #0       IS IT A ZERO?
        BEQ     ZERO     BRANCH ROUTINE
        CMPA    #1       A ONE?
        BEQ     ONE
        CMPA    #2       A TWO?
        BEQ     TWO
        BRA     NOTFND   FAILURE
```

This routine simply reads the character, and then uses the CMP instruction to check its value.

We will now run a different test.

BRACKET TESTING

This program determines if the ASCII character at memory location LOC is a digit between 0 and 9:

```
BRACK   LDA     LOC      GET CHARACTER
        ANDA    #$7F     MASK OUT PARITY BIT
        CMPA    #$30     ASCII 0
        BLT     OUT      CHAR TOO LOW?
        CMPA    #$39     ASCII 9
        BGT     OUT      CHAR TOO HIGH?
        CLRA             FORCE ZERO FLAG
OUT     RTS
```

ASCII 0 is represented in hexadecimal by 30 or by B0, depending upon whether the parity bit is used or not. Similarly, ASCII 9 is represented in hexadecimal by 39 or by B9.

The purpose of the second instruction of the program is to delete bit 7, the parity bit, in case it was used, so that the program is applicable to both cases. The value of the character is then compared to the ASCII values for 0 and 9. When using a comparison instruction, the Z bit is set, if both the contents of the register and the operand are equal. The carry bit is set, if there is a borrow. This means that the carry bit is set, if the value of the operand is greater than the contents of the register.

The instruction CLRA forces a 0 into the Z bit. The Z bit is used to indicate to the calling routine that the character in LOC was indeed in the interval (0,9). Other conventions, such as loading a digit in the accumulator, could also be used to indicate the results of the test.

When using an ASCII table, note that parity is often used. For example, the ASCII representation for 0 is 0110000, a 7-bit code. If, however, we use odd parity and guarantee that the total number of 1s in a word is odd, then the code becomes 10110000 (or B0 in hexadecimal). An extra 1 is added to the left side of the code. Let's now develop a program to generate parity.

GENERATING PARITY

This program generates even parity in bit position 7:

```
PARITY   LDA     CHAR      GET CHARACTER
         PSHS    A         SAVE CHAR ON STACK
         CLRA
         LDB     #7        COUNT 7 BITS
BITCNT   LSR     ,S        SHIFT CHAR RIGHT
         BCC     NOINC     C = ZERO SKIP
         INCA              COUNT CARRY BITS
NOINC    DECB              LOOP TILL
         BNE     BITCNT    7 BITS ARE TESTED
         LSRA              CHECK IF A IS EVEN
         BCC     DONE      IF EVEN THEN DONE
         LDA     CHAR      GET CHARACTER
         ORA     #$80      SET BIT 7
         STA     CHAR
DONE     PULS    A         CLEAN UP STACK
         RTS
```

This program shifts a character and then counts the number of 1s in it. If the number of 1s is even, the parity bit is not set, if the number is odd, the parity bit is set.

The stack is used as working space for this program. Shifting destroys the character, but it is preserved in CHAR. It is important to note that the stack pointer, S, is restored to its previous value by the PULS A instruction. If this is not done, the stack will eventually overflow memory.

CODE CONVERSION: ASCII TO BCD

Converting ASCII to BCD is very simple. In this example, we see that the hexadecimal representation of ASCII characters 0 to 9 is 30 to 39 or B0 to B9, depending on parity. The BCD representation is simply obtained by dropping the 3 or the B, i.e., masking off the left nibble (4 bits). Here is the program:

```
ASCBCD  JSR      BRACK       CHECK THAT CHAR IS 0 TO 9
        BNE      ILLEGAL     EXIT IF ILLEGAL CHAR
        LDA      CHAR        GET CHARACTER
        ANDA     #$0F        ZERO HIGH NIBBLE
        STA      BCDCHR      STORE RESULT
```

In full BCD notation, the first word contains the count of BCD digits, the next contains the sign, and every successive nibble contains a BCD digit (we assume no decimal point). The last nibble of the block may not be used.

CONVERTING HEX TO ASCII

In the example, the A register contains one hexadecimal digit. We simply need to add a 3 (or a B) into the left nibble. Here is the program:

```
        ANDA     #$F         ZERO LEFT NIBBLE
        ADDA     #$30        ASCII
        CMPA     #$3A        CORRECTION NEEDED?
        BLT      OUT
        ADDA     #7          CORRECTION FOR A THRU F
```

FINDING THE LARGEST ELEMENT OF A TABLE

The beginning address of the table is contained at memory address BASE. The first entry of the table is the number of bytes it contains. The following program searches for the largest element of the table. Its value is then stored in A, and its position is stored in Y.

This program uses registers A, B, X, and Y, and indexed addressing, to search a table anywhere in memory (see Figure 8.1):

```
MAX     LDX     #BASE       TABLE ADDRESS
        LDB     ,X+         BYTES IN TABLE
        CLRA                CLEAR MAXIMUM VALUE
        TFR     X,Y         INITIALIZE Y
LOOP    CMPA    ,X+         COMPARE ENTRY
        BHI     NOSWIT      BRANCH IF LESS THAN MAX
        LEAY    -1,X        SET NEW POSITION
        LDA     ,Y          LOAD NEW MAX
NOSWIT  DECB                DECREMENT COUNTER
        BNE     LOOP        KEEP GOING UNTIL ZERO
        RTS
```

This program tests the nth entry. If it is greater than A, the entry goes into A, and its location is remembered in Y. The (n + 1st) entry is then tested, etc. This program works for positive integers only.

SUM OF N ELEMENTS

This program computes the 16-bit sum of N entries of a table. The starting address of the table is contained at memory address BASE, in

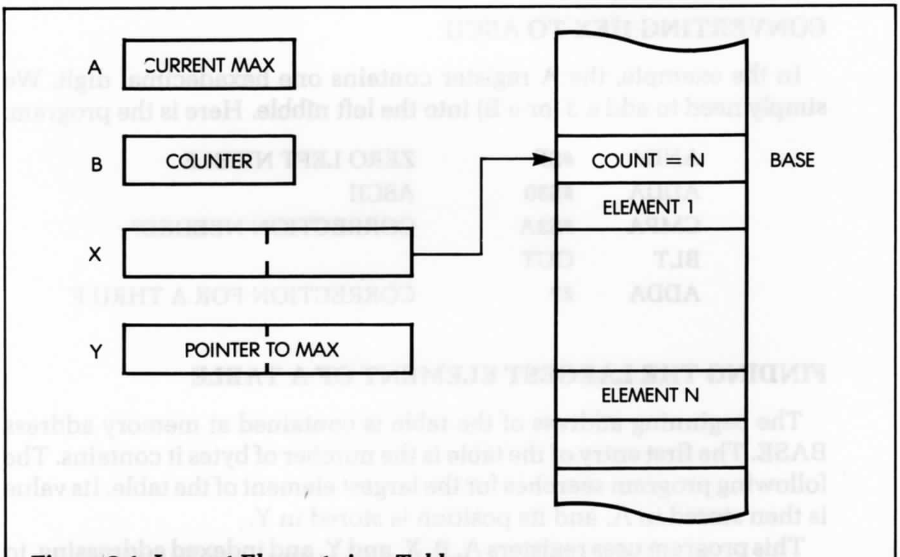


Figure 8.1: Largest Elements in a Table

page zero. The first entry of the table contains the number of elements in N. The 16-bit sum is left in memory locations SUMLO and SUMHI. If the sum requires more than 16 bits, only the lower 16 bits are kept. (The high order bits are said to be truncated.)

This program modifies registers A, B, X, and Y. It assumes 256 elements maximum (see Figure 8.2):

```
SUMIG   LDX    #BASE    POINT TO TABLE BASE
        LDB    ,X+      READ LENGTH INTO COUNTER
        LDY    #SUMLO   POINT TO RESULT LO
        CLR    SUMLO    CLEAR RESULT
        CLR    SUMHI
ADLOOP  LDA    ,X+      GET TABLE ENTRY
        ADDA   ,Y       COMPUTE PARTIAL SUM
        STA    ,Y       STORE IT
        BCC    NOCARY   CHECK FOR CARRY
        INC    1,Y      ADD CARRY TO HIGH BYTE
NOCARY  DECB            DECREMENT BYTE COUNT
        BNE    ADLOOP   KEEP ADDING TIL END
        RTS
```

This program should be self-explanatory.

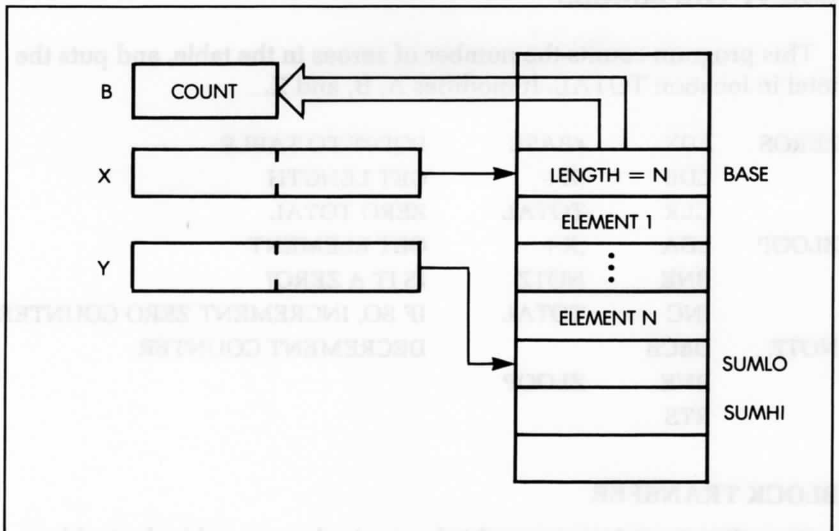


Figure 8.2: Sum of N Elements

A CHECKSUM COMPUTATION

A *checksum* is a digit or set of digits computed from a block of successive characters. The checksum is computed at the time the data is stored; it is then put at the end. To verify the integrity of the data, the data is read, and the checksum is recomputed and compared with the stored value. A discrepancy indicates an error or failure.

We can use several algorithms. In this example, we exclusive-OR all the bytes in a table of N elements, and leave the results in the accumulator. As usual, the base of the table is stored at address BASE. The first entry of the table is its number of elements, N. The program then modifies A, B, and X. N must be less than 256 elements:

```
CHKSUM  LDX     #BASE      POINT TO TABLE
        LDB     ,X+        GET LENGTH
        CLRA               CLEAR CHECK SUM
CHLOOP  EORA    ,X+        COMPUTE CHECKSUM
        DECB               DECREMENT COUNTER
        BNE     CHLOOP     REPEAT UNTIL END
        STA     ,X         PUT CHECKSUM AT END OF TABLE
        RTS
```

COUNT THE ZEROES

This program counts the number of zeroes in the table, and puts the total in location TOTAL. It modifies A, B, and X.

```
ZEROS   LDX     #BASE      POINT TO TABLE
        LDB     ,X+        GET LENGTH
        CLR     TOTAL      ZERO TOTAL
ZLOOP   LDA     ,X+        GET ELEMENT
        BNE     NOTZ       IS IT A ZERO?
        INC     TOTAL      IF SO, INCREMENT ZERO COUNTER
NOTZ    DECB               DECREMENT COUNTER
        BNE     ZLOOP
        RTS
```

BLOCK TRANSFER

We will now pick up every third entry in the source block at address

FROM and store it in a block at address TO:

```
FER3    LDX     #FROM
        LDY     #TO           SET UP POINTERS
        LDB     #LENGTH
LOOP    LDA     ,X            GET AN ENTRY
        STA     ,Y+           STORE IT
        LEAX    3,X           POINT TO THIRD
        DECB
        BNE     LOOP
```

BUBBLE-SORT

Bubble-sort is a sorting technique used to arrange the elements of a table in ascending or descending order. The bubble-sort technique derives its name from the fact that the smallest element "bubbles up" to the top of the table; every time it "collides" with a "heavier" element, it jumps over it.

Figures 8.3 and 8.4 show practical examples of a bubble-sort. The list to be sorted contains the numbers 10, 5, 0, 2, and 100, and must be sorted in descending order (0 on top). The algorithm is simple. The flowchart for the algorithm appears in Figure 8.5.

The two top (or else the two bottom) elements are compared. If the lower element is less (lighter) than the top element, they are exchanged. Otherwise, they are left alone. For practical purposes, the exchange, if it occurs, is indicated by a flag, called "EXCHANGED." The process is then repeated on the next pair of elements, etc., until all elements have been compared, two by two.

Figure 8.3 illustrates this first pass in steps 1, 2, 3, 4, 5, and 6, going from the bottom up. (Equivalently, we could go from the top down.) If no elements have been exchanged, the sort is complete. If an exchange has occurred, we must start over again. Looking at Figure 8.4, we see that four passes are necessary in this example. This process is simple, and widely used.

One possible complication resides in the actual mechanism of the exchange. When exchanging A and B, we may not write:

$$A = B$$

or

$$B = A$$

as this would result in the loss of the previous value of A. (Try it on an example.) The correct solution is to use a temporary variable or location

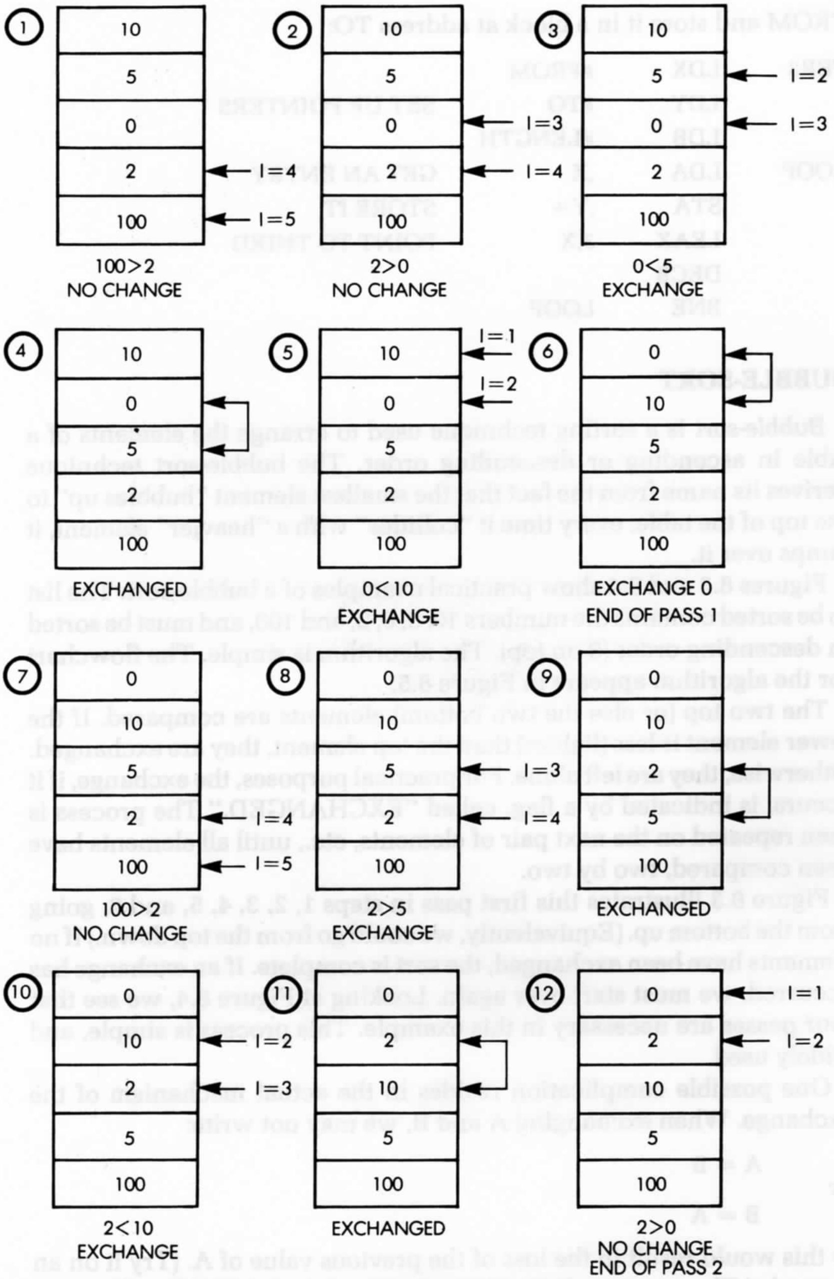


Figure 8.3: Bubble-Sort Example: Phases 1 to 12

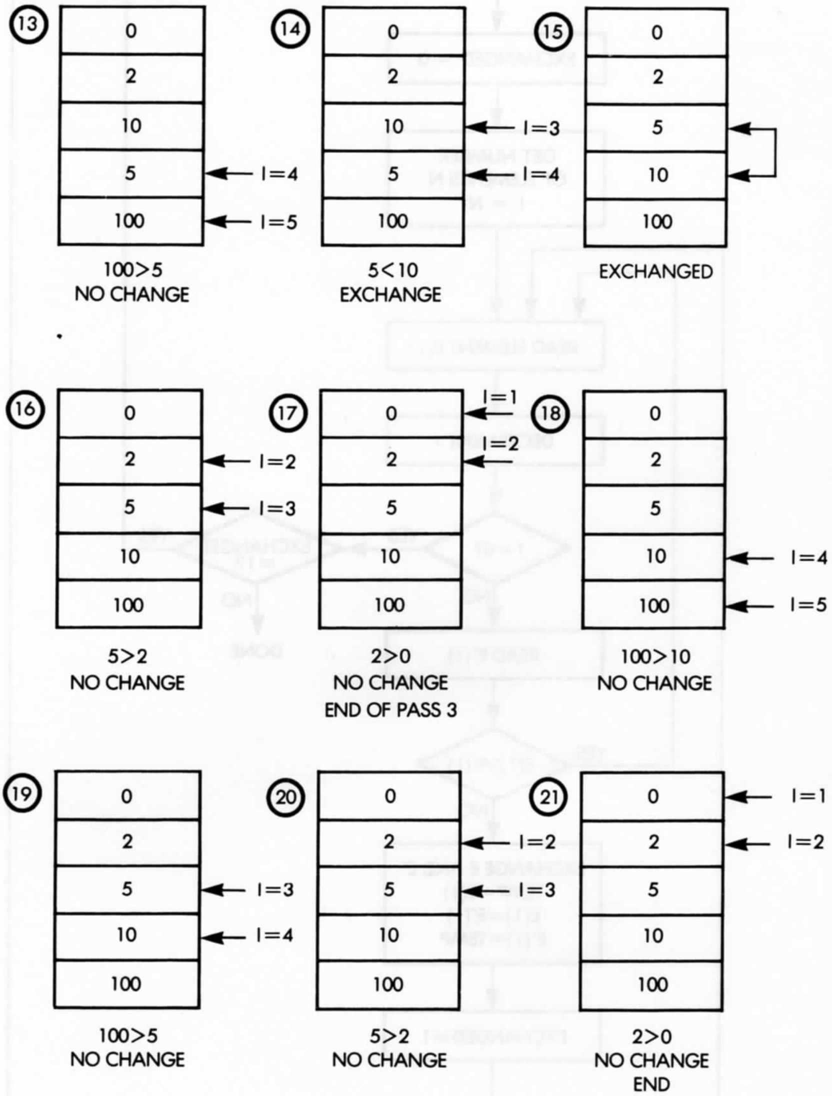


Figure 8.4: Bubble-Sort Example: Phases 13 to 21

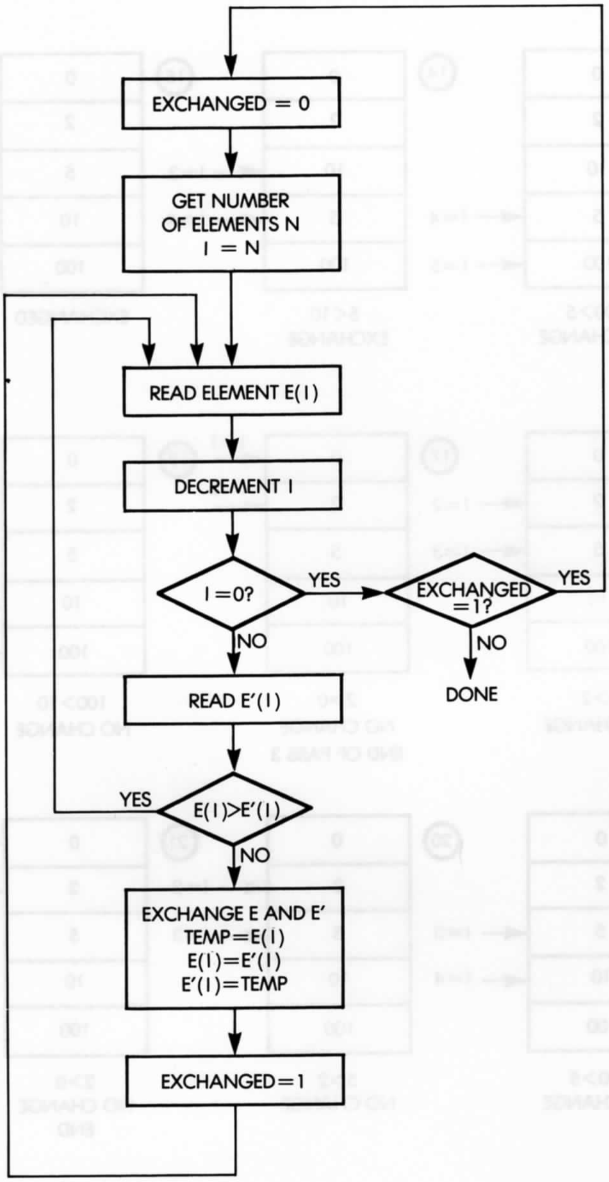


Figure 8.5: Bubble-Sort Flowchart

to preserve the value of A. For example, we may use:

```
TEMP = A
A    = B
B    = TEMP
```

This process, called *circular permutation*, works. (Try it on an example.)

All programs implement the exchange in this way. Figure 8.5 illustrates the process. Figure 8.6 shows the register and memory assignments.

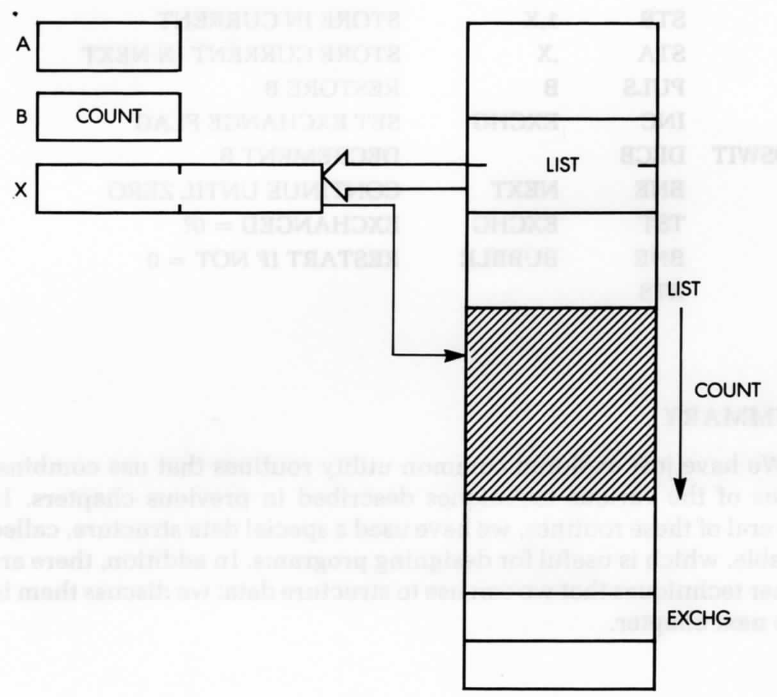


Figure 8.6: Bubble-Sort

The program is:

```
BUBBLE   LDX     #BASE      GET TABLE
         LDB     #LENGTH    GET LENGTH
         DECB
         LEAX    B,X        POINT TO END
         CLR     EXCHG      CLEAR EXCHANGE FLAG
NEXT     LDA     ,X         A = CURRENT ENTRY
         CMPA    ,-X        COMPARE WITH NEXT
         BGE     NOSWIT     GO TO NOSWITCH IF CURRENT
                            >= NEXT
         PSHS    B          SAVE B
         LDB     ,X         GET NEXT
         STB     1,X        STORE IN CURRENT
         STA     ,X         STORE CURRENT IN NEXT
         PULS    B          RESTORE B
         INC     EXCHG      SET EXCHANGE FLAG
NOSWIT   DECB               DECREMENT B
         BNE     NEXT       CONTINUE UNTIL ZERO
         TST     EXCHG      EXCHANGED = 0?
         BNE     BUBBLE     RESTART IF NOT = 0
         RTS
```

SUMMARY

We have just explored common utility routines that use combinations of the various techniques described in previous chapters. In several of these routines, we have used a special data structure, called a table, which is useful for designing programs. In addition, there are other techniques that we can use to structure data; we discuss them in the next chapter.

EXERCISES

8-1: *Write a memory test program that:*

- *zeroes a 256-word block and verifies that each location is 0*
- *writes all 1s and verifies the contents of the block*
- *writes 01010101 and verifies the contents*
- *and, finally, writes 10101010 and verifies the contents.*

8-2: *Modify the program you wrote for Exercise 8-1, so that it fills the memory section with alternating 0s and 1s (i.e., 0s, then all 1s).*

8-3: *Try to improve the STRING program by:*

- *Echoing the character back to the device (for a Teletype, for example).*
- *Checking that the input string is no longer than 256 characters.*

8-4: *Is the following program equivalent to the Bracket Testing program?:*

```
LDA     LOC
SUBA    #$30
BMI     OUT
SUB     #10
BPL     OUT
ADDA    #10
```

8-5: *Determine if an ASCII character contained in an accumulator is a letter of the alphabet.*

8-6: *Using the parity generation program as an example, verify the parity of a word. Compute the correct parity, then compare it to the one that is expected.*

8-7: *Write a program to convert BCD to ASCII.*

8-8: *Write a program to convert BCD to binary (more difficult). (Hint: $N_3N_2N_1N_0$ in BCD is $(((N_3 \times 10) + N_2) \times 10 + N_1) \times 10 + N_0$ in binary.)*

8-9: *Convert HEX to ASCII, assuming a packed format (two hex digits in A).*

8-10: *Modify the program that finds the largest element in a table, so that it also works for negative numbers in two's complement.*

8-11: *Will the program in Exercise 8-10 also work for ASCII characters?*

8-12: *Write a program that sorts n numbers in ascending order.*

8-13: *Write a program that sorts n names (3 characters each) in alphabetical order.*

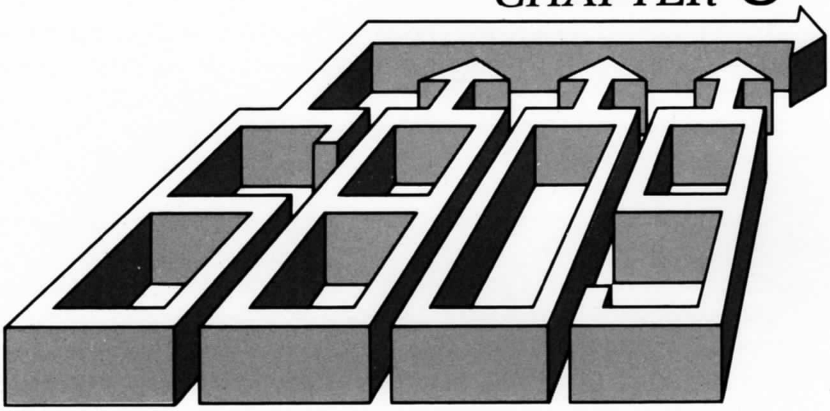
8-14: *Modify the sum of the n elements program to:*

- *compute a 24-bit sum*
- *compute a 32-bit sum*
- *detect any overflow.*

8-15: *Modify the Count the Zeroes program to count:*

- *the number of stars (the character "*")*
- *the number of letters of the alphabet*
- *the number of digits between 0 and 9.*

CHAPTER 9



DATA STRUCTURES

PART I—THEORY

TO DESIGN A GOOD PROGRAM you need both a good algorithm design and a good data structure design. Most simple programs do not involve significant data structures, therefore, up to this point, we have only concentrated on designing and coding good algorithms in a given machine language. We will now turn our attention to the design of data structures, so that we can develop more complex programs. We have already used two data structures in this book: the table and the stack. We will now examine several other, more general, data structures.

The material presented in this chapter is theoretical in concept; it involves the logical organization of data in any system. However, the aptness of the 6809 is particularly apparent here as its addressing modes (often combined with its multiplication instruction) yield particularly efficient implementations of more complex data structures. We have limited the material in this chapter to only that which is essential for understanding common data structures. We will begin by reviewing the most common data structure: the pointer.

POINTERS

A *pointer* is a number that designates the location of actual data. Every pointer is an address. However, every address is not necessarily a pointer. An address is a pointer only if it points to some type of data or structured information. In this book, we have already encountered a typical pointer, the stack pointer, which points to the top of the stack (or just over the top of the stack). The stack, called an LIFO structure, is a common data structure. As another example, when using indirect addressing, the indirect address is always a pointer to the data that is to be retrieved.

LISTS

Almost all data structures are organized as lists. We will now examine several types of lists.

A Sequential List

A *sequential list*, table, or block is probably the simplest data structure (see Chapter 8). *Tables* are normally ordered in function of a specific criterion, such as an alphabetical or numerical ordering. Because of this, it is easy to retrieve an element in a table, by using, for example, indexed addressing.

A *block* normally refers to a group of data that has definite limits, but whose contents are not ordered. A block may contain a string of characters. It may be a sector on a disk, or it may be some logical area (called segment) of the memory. Generally, it is not easy to access a random element of a block. Directories are used to facilitate the retrieval of blocks of information.

A Directory

A *directory* is a list of tables or blocks. For example, the file system normally uses a directory structure. As a simple example, the master directory of a system may include a list of users' names (illustrated in Figure 9.1). In this example, the entry for user "John" points to John's file directory. In this case, the *file directory* is a table of pointers containing the names and locations of all of John's files. For this example, we have designed a two-level directory. This flexible directory system allows the inclusion of additional, intermediate directories—a convenient feature for the user.

A Linked List

In a system, there are often blocks of information that represent data, events, or other structures that cannot be moved around easily. If they could, we would probably assemble them in a table in order to sort or structure them. Let's assume, for example, that we want to leave several blocks where they are, but we also want to establish an ordering among them, such as first, second, third, or fourth. To do this, we will use a linked list (see Figure 9.2).

In the illustration in Figure 9.2, a list pointer, called FIRSTBLOCK, points to the beginning of the first block. A dedicated location within Block 1, such as the first or last word, contains a pointer to Block 2, called PTR1. The process is then repeated for Blocks 2 and 3. Since Block 3 is the last entry in the list, then, by convention, PTR3 contains either a special "nil" value or points to itself. This is done so that the end of the list can be detected. The linked list structure is economical, as it requires only one pointer per block, and frees the user from having to physically move the blocks in the memory.

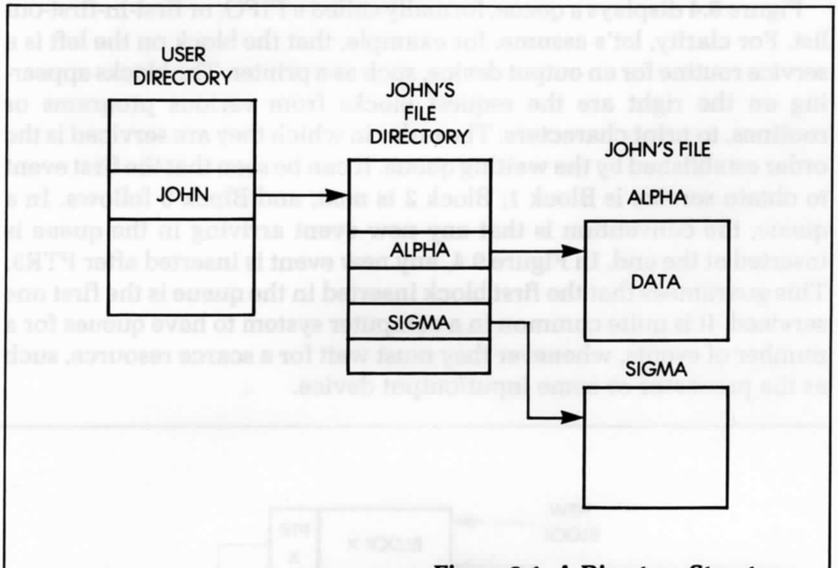


Figure 9.1: A Directory Structure

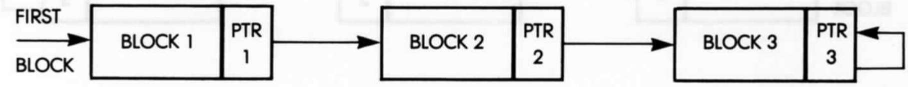


Figure 9.2: A Linked List

Let's now examine how a new block is inserted into a linked list (see Figure 9.3). We will assume that the new block is at address NEWBLOCK, and is to be inserted between Block 1 and Block 2. Pointer PTR1 is simply changed to the value NEWBLOCK, so that it now points to Block X. PTRX now contains the former value of PTR1, i.e., it points to Block 2. The other pointers in the structure are left unchanged. We can see that the insertion of a new block has simply required the updating of two pointers in the structure—a clearly efficient procedure.

Several types of lists have been developed to facilitate specific types of access, insertions, and deletions, to and from the list. We will now examine some of the more frequently used types of linked lists.

A Queue

Figure 9.4 displays a *queue*, formally called a FIFO, or first-in-first-out list. For clarity, let's assume, for example, that the block on the left is a service routine for an output device, such as a printer. The blocks appearing on the right are the request blocks from various programs or routines, to print characters. The order in which they are serviced is the order established by the waiting queue. It can be seen that the first event to obtain service is Block 1; Block 2 is next; and Block 3 follows. In a queue, the convention is that any new event arriving in the queue is inserted at the end. In Figure 9.4, any new event is inserted after PTR3. This guarantees that the first block inserted in the queue is the first one serviced. It is quite common in a computer system to have queues for a number of events, whenever they must wait for a scarce resource, such as the processor or some input/output device.

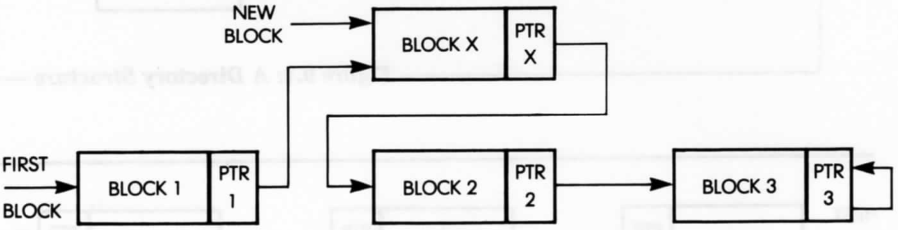


Figure 9.3: Inserting a New Block

A Stack

We have already discussed the stack structure, a last-in-first-out (LIFO) structure. The last element deposited on top is the first one to be removed. A stack may be implemented as either a sorted block or a list. Because most stacks in microprocessors are used for high-speed events, such as subroutines and interrupts, a continuous block is usually allocated to the stack, rather than a linked list structure.

Linked List Versus Block

Similarly, a queue could be implemented as a block of reserved locations. Advantages of using a continuous block include fast retrieval and the elimination of pointers. A disadvantage is that it is usually necessary to dedicate a fairly large block in order to accommodate the worst-case size of the structure. In addition, it is often difficult or impractical to insert or remove elements from within the block. Since memory is traditionally a scarce resource, blocks have usually been reserved for fixed-size structures or structures, such as the stack, that require the maximum speed of retrieval.

A Circular List

"Round robin" is a common name for a *circular list.* A circular list is a linked list in which the last entry points back to the first (see Figure 9.5).

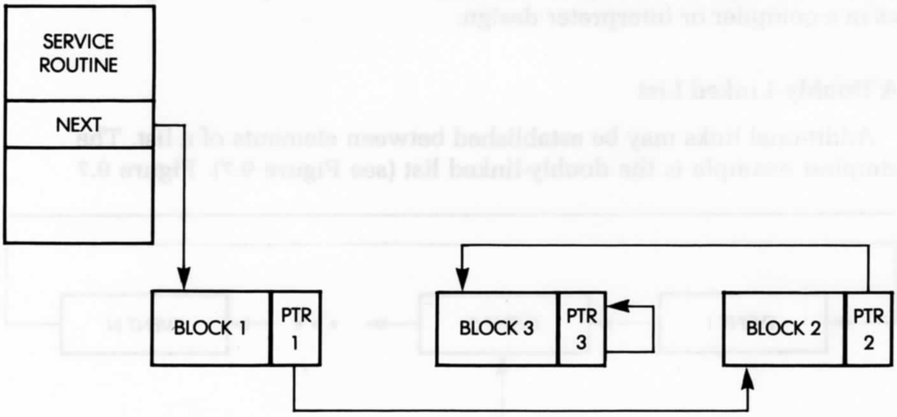


Figure 9.4: A Queue

In the case of a circular list, a *current-block* pointer is often kept. In the case of events, or programs waiting for service, a *current-event* pointer is moved by one position to the left or right each time. A round robin usually corresponds to a structure in which all blocks are assumed to have the same priority. However, a circular list may also be used as a subcase of other structures, in order to facilitate the retrieval of the first block after the last one, when performing a search.

A polling program is a good example of a circular list. It usually goes in a round robin fashion, interrogating all peripherals and then coming back to the first one.

A Tree Structure

A tree structure may be used whenever a logical relationship (called a *syntax*) exists among all elements of a structure. A simple example of a tree structure is a descendant or genealogical tree (see Figure 9.6). The tree in Figure 9.6 shows that Smith has two children: a son, Robert, and a daughter, Jane. Jane, in turn, has three children: Liz, Tom and Phil. Tom, in turn, has two children: Max and Chris. Robert, on the left of the illustration, has no descendants.

This tree is a structured tree. Figure 9.1 showed an example of a simple tree: the directory structure was a two-level tree.

Trees are used to advantage whenever elements can be classified according to a fixed structure, thus facilitating insertion and retrieval. In addition, trees can be used to establish groups of information in a structured way, so that they can be easily used for later processing, such as in a compiler or interpreter design.

A Doubly-Linked List

Additional links may be established between elements of a list. The simplest example is the doubly-linked list (see Figure 9.7). Figure 9.7

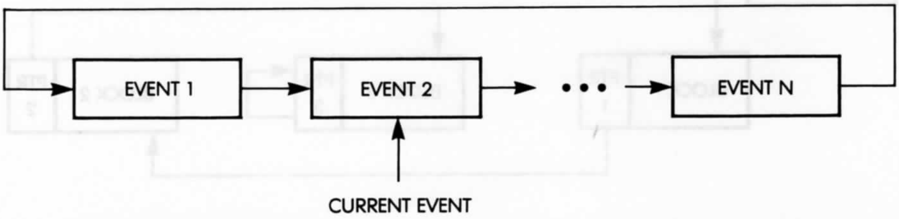


Figure 9.5: *A Round Robin Is A Circular List*

shows the usual sequence of links from left to right, plus another sequence of links from right to left. The goal is to allow easy retrieval of the elements just before and after the element being processed. This method does, however, cost an extra pointer per block.

SEARCHING AND SORTING

The process of searching and sorting elements of a list depends directly on the type of structure used for the list. Many searching algorithms have been developed for the most frequently used data structures. As an example, we used indexed addressing in Chapter 8 to search through a table for a particular element. Recall that we can use indexed addressing

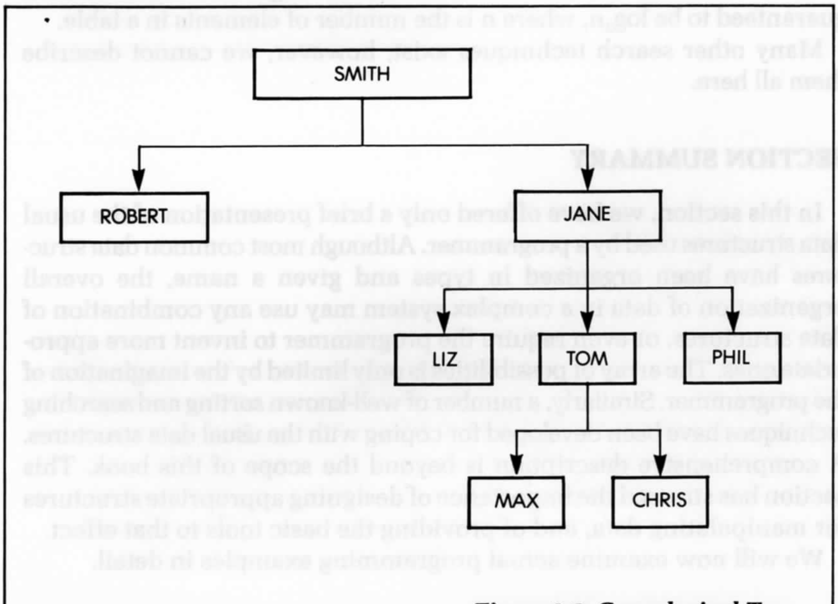


Figure 9.6: Genealogical Tree

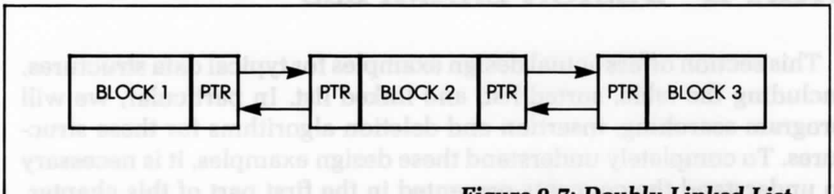


Figure 9.7: Doubly-Linked List

whenever the elements of a table are ordered by a function of known criterion. Such elements can then be retrieved by their numbers.

Sequential searching refers to the linear scanning of an entire block. This technique is clearly inefficient; however, it may be necessary to use it when no better technique is available, for lack of ordering of the elements.

Binary or *logarithmic searching* attempts to find an element in a sorted list, by dividing the search interval in half at each step. For example, let's assume that we are searching an alphabetical list. We might start in the middle of a table and determine if the name we are looking for is before or after that point. If it is after, we will eliminate the first half of the table and look at the middle element of the second half. We compare this entry again to the one we are looking for, and we restrict our search to one of the two halves, and so on. The maximum length of a search is then guaranteed to be $\log_2 n$, where n is the number of elements in a table.

Many other search techniques exist; however, we cannot describe them all here.

SECTION SUMMARY

In this section, we have offered only a brief presentation of the usual data structures used by a programmer. Although most common data structures have been organized in types and given a name, the overall organization of data in a complex system may use any combination of data structures, or even require the programmer to invent more appropriate ones. The array of possibilities is only limited by the imagination of the programmer. Similarly, a number of well-known sorting and searching techniques have been developed for coping with the usual data structures. A comprehensive description is beyond the scope of this book. This section has stressed the importance of designing appropriate structures for manipulating data, and of providing the basic tools to that effect.

We will now examine actual programming examples in detail.

PART II—DESIGN EXAMPLES

This section offers actual design examples for typical data structures, including the table, sorted list, and linked list. In particular, we will program searching, insertion and deletion algorithms for these structures. To completely understand these design examples, it is necessary to understand the concepts presented in the first part of this chapter.

The programs we present in this section use most of the addressing modes of the 6809, and integrate many of the concepts and techniques presented in previous chapters.

We will now introduce three structures: a simple list, an alphabetical list, and a linked-list, plus directory. For each structure, we will develop three programs: search, enter and delete.

DATA REPRESENTATION FOR THE LIST

In the example shown in Figure 9.8, note that both the simple list and the alphabetic list use a common representation for each list element. Each element, or "entry," includes a 3-byte label, and an n-byte block of data, where n is between 1 and 253. Thus, at most, each entry uses one page (256 bytes). Within each list, all elements are the same length (see Figure 9.9). Note that the programs operating on these two simple lists use some common variable conventions, including:

ENTLEN is the length of an element. For example, if each element has 10 bytes of data, ENTLEN = 3 + 10 = 13.

TABASE is the base of the list or table in the memory.

POINTR is the running pointer to the current element.

OBJECT is the current entry to be located, inserted or deleted.

TABLEN is the number of entries.

All labels are assumed to be distinct. Changing this convention would require a minor change in the programs.

A SIMPLE LIST

In this example, we have organized a simple list as a table of n elements. The elements are not sorted (see Figure 9.10). When searching,

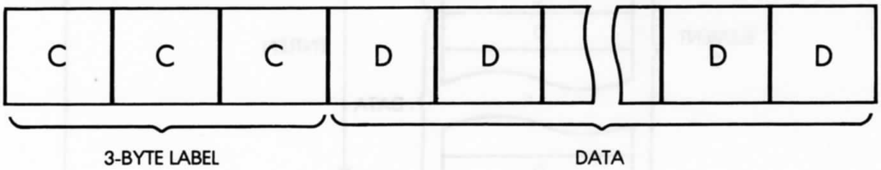


Figure 9.8: A Single List Entry

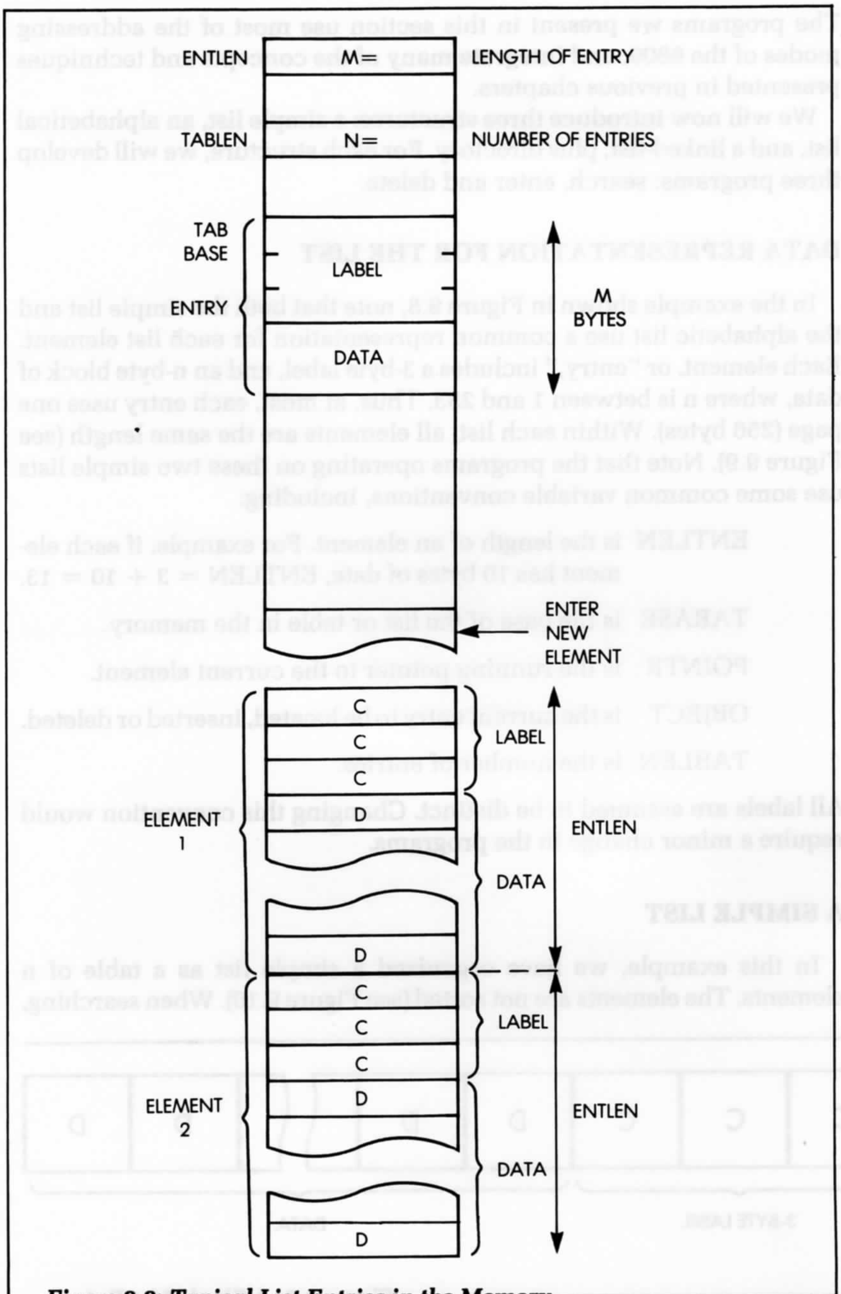


Figure 9.9: Typical List Entries in the Memory

the list is scanned until either an entry is found or the end of the table is reached. When inserting, the new entry is appended to the existing ones. When deleting, the entries in higher memory locations, if any, are shifted down to keep the table continuous. Let's examine these functions in more detail.

Searching

We will now look at an example using a serial search technique, where each entry's label field is compared in turn to the OBJECT's label, letter by letter. We will initialize the running pointer in the X register to the value of the TABASE. In this program, we will use indexed addressing modes and the load effective address (LEA) instruction.

The search proceeds in an obvious way. Figure 9.11 shows the corresponding flowchart. The program appears in Figure 9.14 (program SEARCH).

Inserting

When we insert a new element, the first available memory block ENTLEN bytes long at the end of the list is used (see Figure 9.10). The

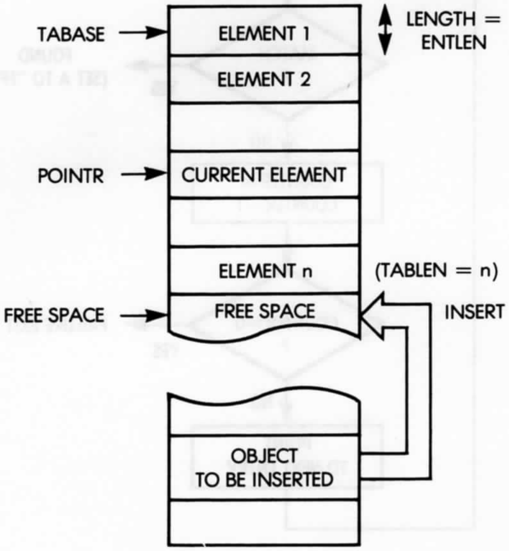


Figure 9.10: The Simple List

program first checks that the new entry is not already in the list. All labels are assumed to be distinct in this example. If the entry is not found, the program increments the list length TABLEN, and moves the OBJECT to the end of the list. Figure 9.12 shows the corresponding flowchart. Figure 9.15 displays the program, called NEW.

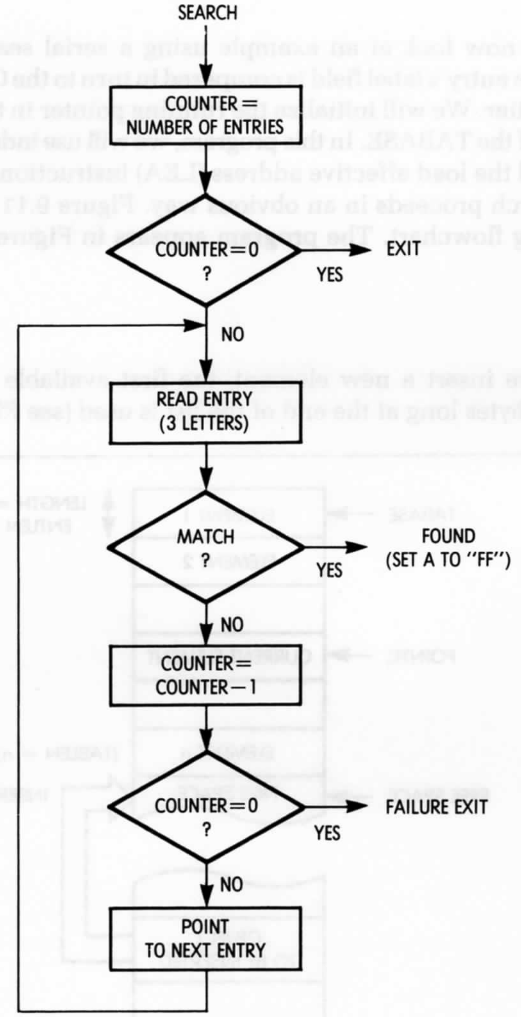


Figure 9.11: Table Search Flowchart

Deleting

To delete an element from the list, the elements following that element in the list at higher addresses are merely moved up by one element position. The length of the list must also be decremented (see Figure 9.13).

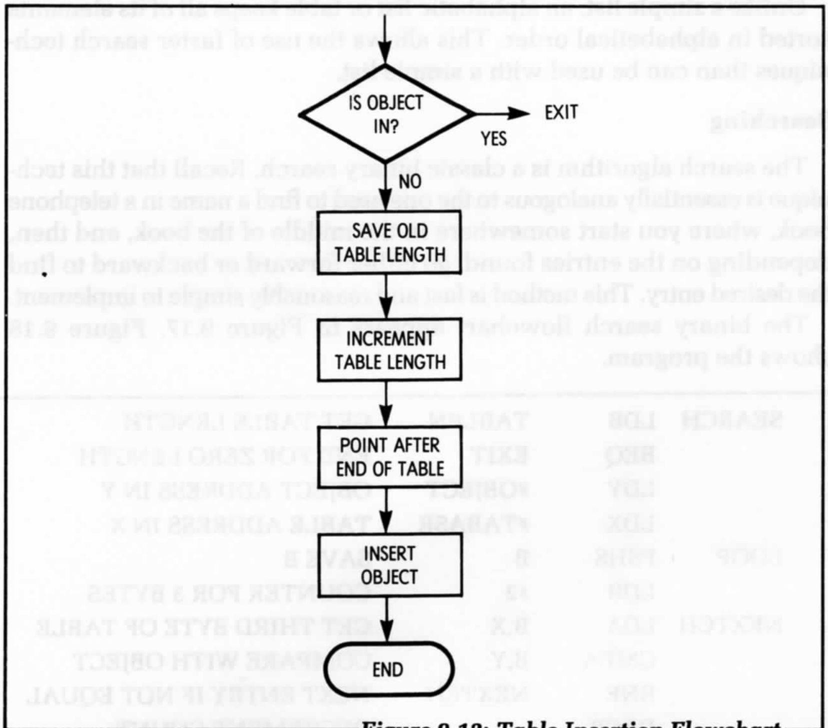


Figure 9.12: Table Insertion Flowchart

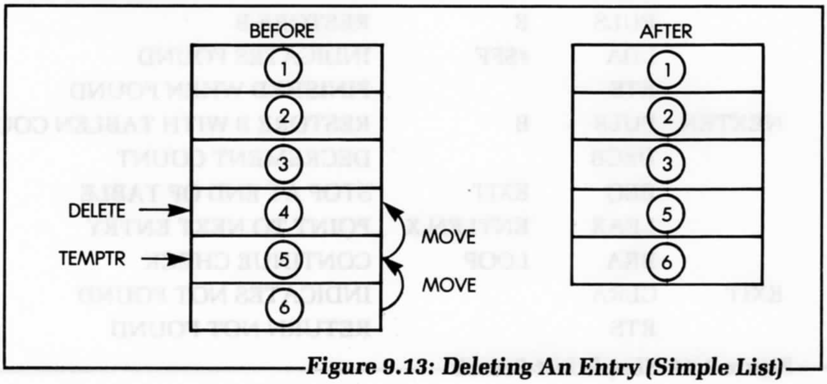


Figure 9.13: Deleting An Entry (Simple List)

The corresponding program, called DELETE, appears in Figure 9.16 at the end of this section.

ALPHABETIC LIST

Unlike a simple list, an alphabetic list or table keeps all of its elements sorted in alphabetical order. This allows the use of faster search techniques than can be used with a simple list.

Searching

The search algorithm is a classic binary search. Recall that this technique is essentially analogous to the one used to find a name in a telephone book, where you start somewhere in the middle of the book, and then, depending,on the entries found, go either forward or backward to find the desired entry. This method is fast and reasonably simple to implement.

The binary search flowchart appears in Figure 9.17. Figure 9.18 shows the program.

```
SEARCH  LDB     TABLEN      GET TABLE LENGTH
        BEQ     EXIT        END FOR ZERO LENGTH
        LDY     #OBJECT     OBJECT ADDRESS IN Y
        LDX     #TABASE     TABLE ADDRESS IN X
LOOP    PSHS    B           SAVE B
        LDB     #2          COUNTER FOR 3 BYTES
NEXTCH  LDA     B,X         GET THIRD BYTE OF TABLE
        CMPA    B,Y         COMPARE WITH OBJECT
        BNE     NEXTEN      NEXT ENTRY IF NOT EQUAL
        DECB                DECREMENT COUNT
        BPL     NEXTCH      CHECK NEXT CHAR TIL B<0
        PULS    B           RESTORE B
        LDA     #$FF        INDICATES FOUND
        RTS                 FINISHED WHEN FOUND
NEXTEN  PULS    B           RESTORE B WITH TABLEN COUNT
        DECB                DECREMENT COUNT
        BEQ     EXIT        STOP AT END OF TABLE
        LEAX    ENTLEN,X    POINT TO NEXT ENTRY
        BRA     LOOP        CONTINUE CHECK
EXIT    CLRA                INDICATES NOT FOUND
        RTS                 RETURN NOT FOUND
```

Figure 9.14: Simple List-Search

```
NEW     BSR     SEARCH      SEE IF OBJECT IS IN TABLE
        TSTA                CHECK RESULT OF SEARCH
        BNE     OUT         QUIT IF ALREADY IN TABLE
        LDA     TABLEN      GET TABLE LENGTH
        INC     TABLEN      INCREMENT TABLE LENGTH
        LDB     #ENTLEN     GET ENTRY LENGTH
        MUL                 MAKE TABLE SIZE
        LDX     #TABASE     GET START ADDRESS
        LEAX    D,X         POINT TO END OF TABLE
        LDY     #OBJECT     GET OBJECT ADDRESS
        LDB     #ENTLEN     GET ENTRY LENGTH
TRLQOP  LDA     ,Y+         GET BYTE
        STA     ,X+         STORE BYTE
        DECB
        BNE     TRLOOP      LOOP UNTIL TRANSFERRED
OUT     RTS                 FINISHED
```

Figure 9.15: Simple List-New

```
DELETE  BSR     SEARCH      SEE IF OBJECT IS IN TABLE
        TSTA                CHECK RESULT OF SEARCH
        BEQ     DONE        QUIT IF NOT THERE
        DEC     TABLEN      DECREMENT TABLE LENGTH
        DECB                B = # OF ENTRIES LEFT IN TABLE
        BEQ     DONE        ... AFTER ONE TO BE DELETED
        TFR     X,Y         X POINTS TO ENTRY TO DELETE
        LEAY    ENTLEN,Y    Y POINTS TO NEXT BLOCK
MOREBK  PSHS    B           SAVE B
        LDB     #ENTLEN     COUNT TO MOVE A BLOCK
MOVBLK  LDA     ,Y+         MOVE BYTE FROM A BLOCK
        STA     ,X+         UP ONE BLOCK
        DECB
        BNE     MOVBLK      LOOP TIL A BLOCK IS DONE
        PULS    B           RESTORE BLOCK COUNT
        DECB
        BNE     MOREBK      LOOP TIL ALL BLOCKS MOVED
DONE    RTS                 ALL FINISHED
```

Figure 9.16: Simple List-Delete

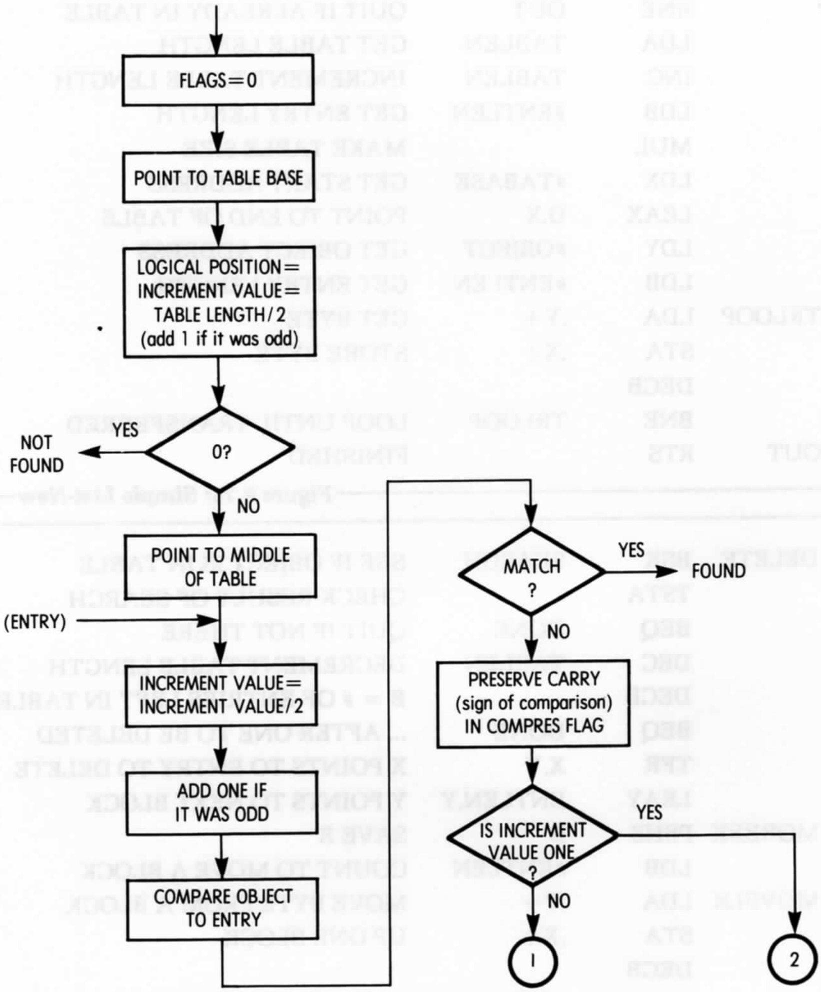


Figure 9.17: Binary Search Flowchart

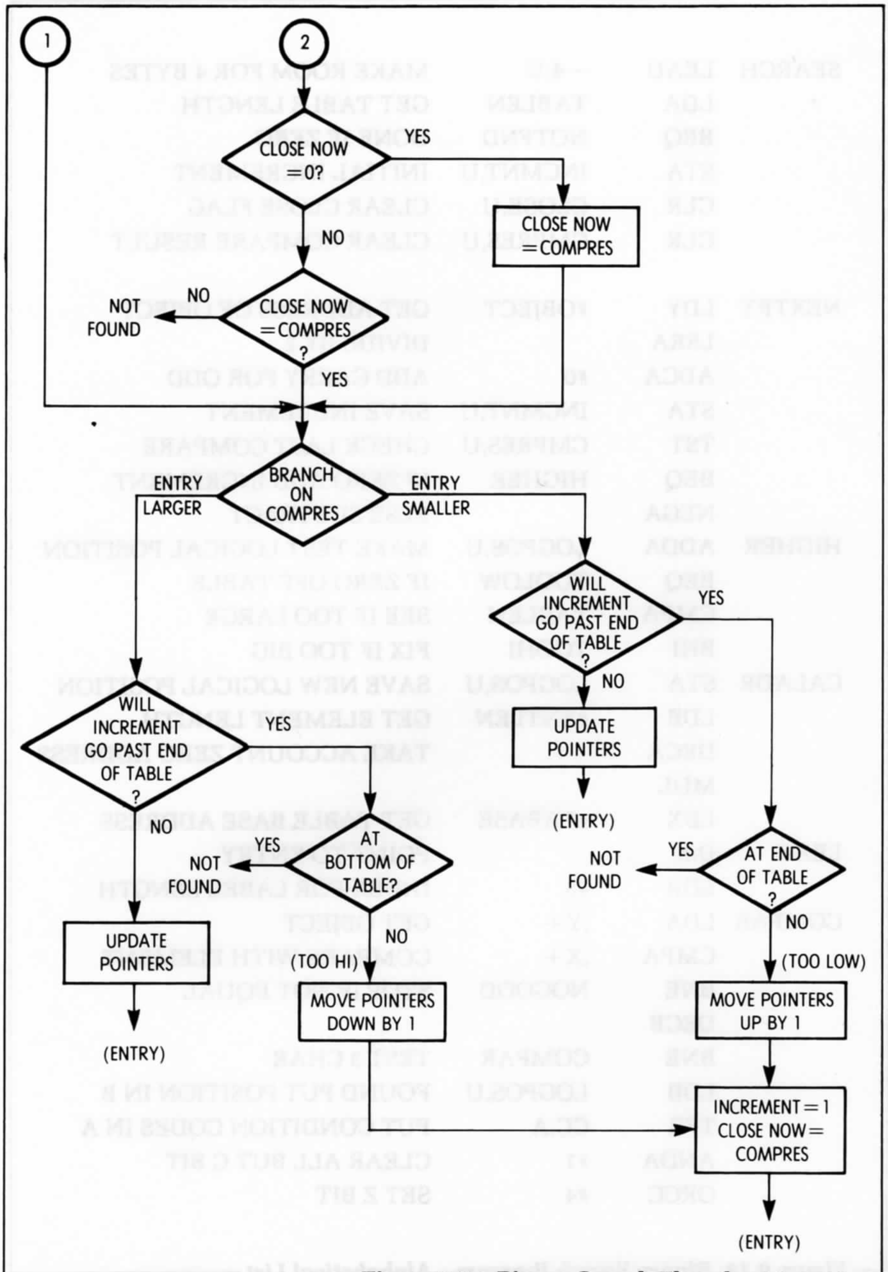


Figure 9.17: Binary Search Flowchart (cont.)

```
SEARCH  LEAU    −4,U        MAKE ROOM FOR 4 BYTES
        LDA     TABLEN      GET TABLE LENGTH
        BEQ     NOTFND      DONE IF ZERO
        STA     INCMNT,U    INITIAL INCREMENT
        CLR     CLOSE,U     CLEAR CLOSE FLAG
        CLR     CMPRES,U    CLEAR COMPARE RESULT

NEXTRY  LDY     #OBJECT     GET ADDRESS OF OBJECT
        LSRA                DIVIDE BY 2
        ADCA    #0          ADD CARRY FOR ODD
        STA     INCMNT,U    SAVE INCREMENT
        TST     CMPRES,U    CHECK LAST COMPARE
        BEQ     HIGHER      IF ZERO ADD INCREMENT
        NEGA                ELSE SUBTRACT
HIGHER  ADDA    LOGPOS,U    MAKE TEST LOGICAL POSITION
        BEQ     TOOLOW      IF ZERO OFF TABLE
        CMPA    TABLEN      SEE IF TOO LARGE
        BHI     TOOHI       FIX IF TOO BIG
CALADR  STA     LOGPOS,U    SAVE NEW LOGICAL POSITION
        LDB     #ENTLEN     GET ELEMENT LENGTH
        DECA                TAKE ACCOUNT ZERO ADDRESS
        MUL
        LDX     #TABASE     GET TABLE BASE ADDRESS
LEAX    D,X                 POINT TO ENTRY
        LDB     #3          INDEX FOR LABEL LENGTH
COMPAR  LDA     ,Y+         GET OBJECT
        CMPA    ,X+         COMPARE WITH ELEMENT
        BNE     NOGOOD      STOP IF NOT EQUAL
        DECB
        BNE     COMPAR      TEST 3 CHAR
        LDB     LOGPOS,U    FOUND PUT POSITION IN B
        TFR     CC,A        PUT CONDITION CODES IN A
        ANDA    #1          CLEAR ALL BUT C BIT
        ORCC    #4          SET Z BIT
```

Figure 9.18: Binary Search Program—Alphabetical List

```
        ANDCC   #4          CLEAR ALL OTHER BITS
        LEAU    4,U         RESTORE U STACK POINTER
        RTS                 ALL DONE WHEN FOUND
NOGOOD  TFR     CC,A        PUT CONDITION CODES IN A
        ANDA    #1          CLEAR ALL BUT C BIT
        TST     CLOSE,U     ARE WE CLOSE?
        BEQ     CHKINC      ZERO THEN NOT CLOSE
        CMPA    CMPRES,U    CLOSE COMPARE C BITS
        BNE     NOTFND      NOT EQUAL NOT FOUND
CHKINC  STA     CMPRES,U    STORE LAST COMPARE RESULT
        LDA     INCMNT,U    GET INCREMENT
        CMPA    #1          SEE IF IT IS 1
        BNE     NEXTRY      NOT 1 NEXT CHECK
        INC     CLOSE,U     IF 1 SET CLOSE FLAG
        BRA     NEXTRY      TRY ONCE MORE
NOTFND  LDB     LOGPOS,U    PUT POSITION IN B
        ANDCC   #0          CLEAR Z BIT
        LEAU    4,U         RESTORE U STACK POINTER
        RTS                 FINISHED NOT FOUND
TOOLOW  LDA     LOGPOS,U    GET LAST POSITION
        CMPA    #1          SEE IF IT WAS 1
        BNE     ADJUST      NOT 1 FIX POSITION
        LDA     #1          SET C BIT IN A
        BRA     NOTFND      NOT IN TABLE
ADJUST  LDA     #1          POSITION IS 1
        BRA     CALADR      CALCULATE ADDRESS
TOOHI   LDA     LOGPOS,U    GET LAST POSITION
        CMPA    TABLEN      SEE IF AT END
        BNE     FIXIT
        CLRA                CLEAR C BIT IN A
        BRA     NOTFND      BEYOND TABLE
FIXIT   LDA     TABLEN      POINT TO LAST ELEMENT
        BRA     CALADR      CALCULATE ADDRESS
```

Figure 9.18: Binary Search Program—Alphabetical List (cont.)

The alphabetic list keeps the entries in alphabetical order and retrieves them using a binary or logarithmic type search. Figure 9.19 shows an example of a binary search. The search is somewhat complicated, because it is necessary to keep track of several conditions. The major problem is to avoid searching forever for an object that is not there. In such a case, the entries with higher and lower alphabetic values would be alternately tested forever. To avoid such an occurrence, a flag is maintained in the program to preserve the value of the carry flag after an unsuccessful comparison. When the INCMNT value, which shows the amount by which the pointer was incremented, reaches the value of 1, another flag called CLOSENOW, is set to 1. A flag called COMPRES (comparison result) stores the carry bit from the last comparison. When CLOSENOW is set, the value of COMPRES is compared with the carry bit of the most recent comparison. If they are not equal, the search terminates because the object cannot be found.

The carry bit for the last comparison is returned in A for use by the NEW program. This allows the NEW program to determine whether a new element goes before or after the entry pointed to by the SEARCH program.

The other major problem that must be dealt with is the possibility of running off one end of the table when adding or subtracting the increment. This is solved by performing a test add or subtract of the increment to the logical position or element number. This number is then compared

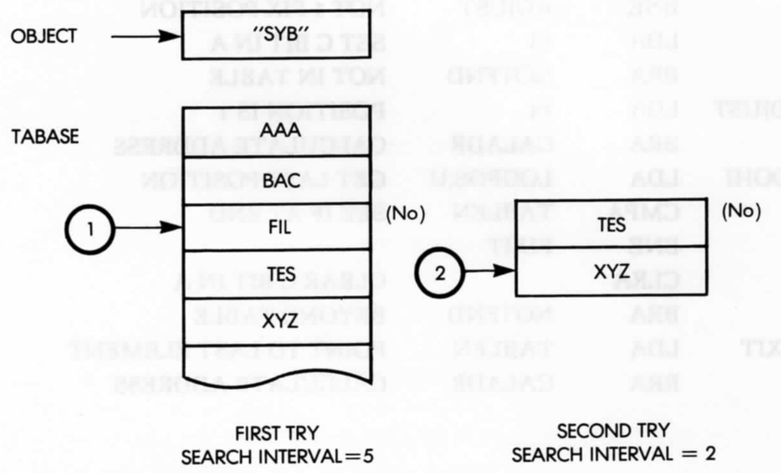


Figure 9.19: A Binary Search

to 1 and the table length. If it is greater than the table length or less than 1, it is adjusted, to fall within the table boundaries.

The following variables are used in the program:

LOGPOS indicates logical position (element number).

INCMNT represents the value by which the pointer will be incremented or decremented if the next comparison fails.

CLOSE is short for CLOSENOW.

CMPRES is short for comparison result.

These variables are accessed by using the U register as an index register. The symbols LOGPOS, INCMNT, CLOSE, and COMPRES have the values 0, 1, 2, and 3, respectively.

An additional complication to this program occurs because the search interval at times can be either even or odd. Since the interval is divided by two to form the increment, we use an LSR instruction. If the bit falling off the right end is not added back into the accumulator, then only even or odd numbered elements would be checked, depending on the value of the table length. This would cause erroneous results.

Study the SEARCH program in Figure 9.18 with care, as it is much more complex than the linear search.

Figure 9.20 shows the insertion process, and Figure 9.21 displays the NEW program.

Element Insertion

In order to insert a new element, a binary search must be conducted. If the element is found in the table, it does not need to be inserted. But if it is not, it must be inserted immediately before or after the last element to which it was compared. The value of the COMPRES flag, returned in register A, indicates whether the new object should be inserted immediately before or after the last element compared. All the elements following the new location are moved down by one block position, and the new object is inserted.

Figure 9.20 shows the insertion process, and Figure 9.21 displays the NEW program.

Element Deletion

Similarly, a binary search is conducted to find the object. If the search fails, the element does not need to be deleted. If the search succeeds, the element is deleted, and all the following elements are moved up by one block position. A corresponding example appears in Figure 9.22. Figure 9.23 shows the flowchart and Figure 9.24 displays the program.

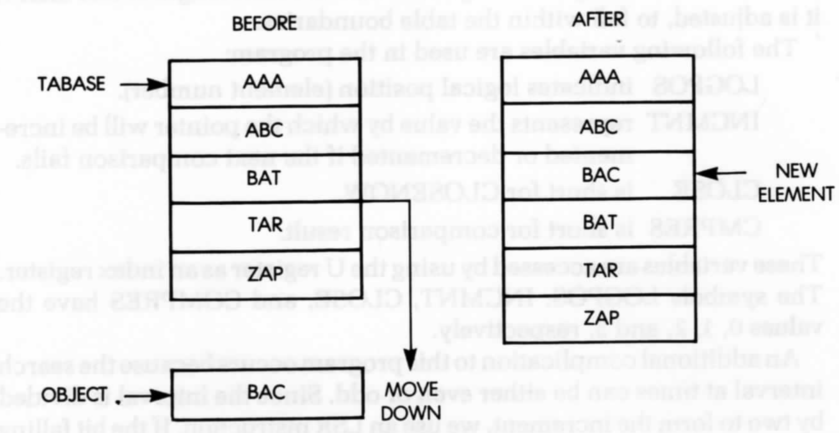


Figure 9.20: Insert: "BAC"

```
NEW       LBSR      SEARCH      SEE IF OBJECT IN LIST
          BEQ       OUT         ALREADY IN LIST
          LDX       #TABASE     GET TABLE BASE
          TST       TABLEN      CHECK TABLE LENGTH
          BEQ       INSERT      IF 0 JUST INSERT
          TSTA                  CHECK LAST CARRY
          BNE       LOSIDE      PUT ABOVE ENTRY IN B
          INCB                  PUT BELOW ENTRY IN B
LOSIDE    TFR       B,A         PUT POSITION IN A
          NEGA                  SUBTRACT IT FROM
          ADDA      TABLEN      ... TABLE LENGTH
          INCA                  A IS NUMBER ELEMENTS TO
                                MOVE
          PSHS      A           SAVE A
          LDA       TABLEN      GET TABLE LENGTH
          LDB       #ENTLEN     GET ELEMENT LENGTH
          MUL
          LEAX      D,X         POINT TO END OF TABLE
          LEAY      ENTLEN,X    POINT ONE ELEMENT BEYOND
          PULS      A           RESTORE A
```

Figure 9.21: NEW Program For An Alphabetical List (continues)

```
BLOOP   TSTA                  CHECK A
        BEQ     INSERT        IF 0 READY TO INSERT
        PSHS    A             SAVE A
        LDB     #ENTLEN       PREPARE TO MOVE A BLOCK
MLOOP   LDA     ,-X           MOVE A BLOCK DOWN
        STA     ,-Y           TO A HIGHER ADDRESS
        DECB
        BNE     MLOOP         LOOP TIL BLOCK DONE
        PULS    A             RESTORE A
        DECA                  DECREMENT BLOCK COUNT
        BRA     BLOOP         CONTINUE
INSERT  INC     TABLEN        ONE MORE ELEMENT
        LDY     #OBJECT       GET OBJECT ADDRESS
        LDB     #ENTLEN       PREPARE TO MOVE OBJECT
MOVOBJ  LDA     ,Y+           GET OBJECT
        STA     ,X+           STORE IN LIST
        DECB
        BNE     MOVOBJ
OUT     RTS                   FINISHED
```

Figure 9.21: NEW Program For An Alphabetical List (cont.)

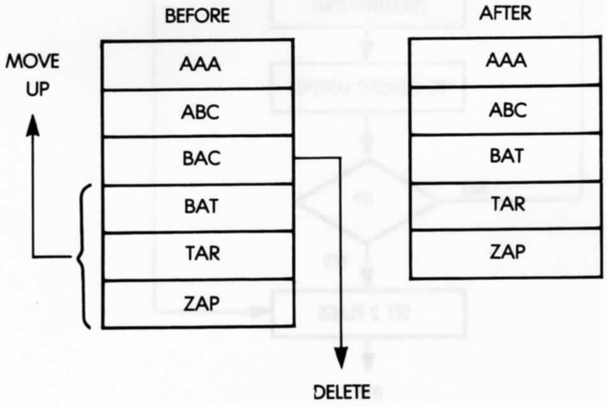


Figure 9.22: Delete "BAC"

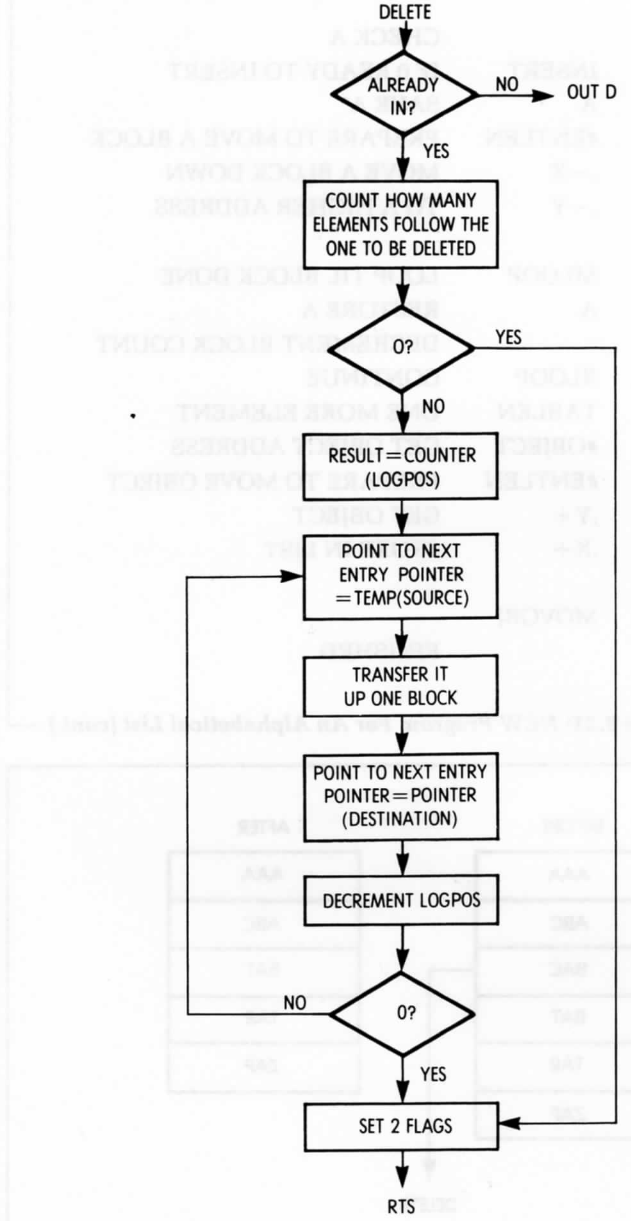

DELETE
ALREADY IN?
NO
OUT D
YES
COUNT HOW MANY ELEMENTS FOLLOW THE ONE TO BE DELETED
0?
YES
NO
RESULT=COUNTER (LOGPOS)
POINT TO NEXT ENTRY POINTER =TEMP(SOURCE)
TRANSFER IT UP ONE BLOCK
POINT TO NEXT ENTRY POINTER=POINTER (DESTINATION)
DECREMENT LOGPOS
NO
0?
YES
SET 2 FLAGS
RTS

Figure 9.23: Deletion Flowchart (Alphabetical List)

LINKED LIST

The linked list is assumed to contain, as usual, the three alphanumeric characters for the label, followed by 1 to 250 bytes of data, then a 2-byte pointer that contains the starting address of the next entry, and finally, a 1-byte marker. Whenever this 1-byte marker is set to 1, it prevents the

```
DELETE  LBSR    SEARCH      FIND OBJECT
        BNE     OUTD        QUIT IF NOT FOUND
        CMPB    TABLEN      SEE IF LAST ELEMENT
        BEQ     TABM1       ... IN TABLE
        PSHS    B           SAVE B
        DECB                ACCOUNT FOR 0 ADDRESS
        LDA     #ENTLEN     GET LENGTH
        MUL
        LDX     #TABASE
        LEAX    D,X         POINT TO ELEMENT TO DELETE
        LEAY    ENTLEN,X    POINT TO NEXT ELEMENT
        PULS    B           RESTORE BLOCK COUNT
        NEGB                SUBTRACT FROM TABLE
        ADDB    TABLEN      ... LENGTH
MOVMOR  TSTB                SEE IF ALL MOVED
        BEQ     TABM1       FINISH IF MOVED
        PSHS    B           SAVE COUNT
        LDB     #ENTLEN     PREPARE TO MOVE A BLOCK
MOVENT  LDA     ,Y+         MOVE A BLOCK UP
        STA     ,X+
        DECB
        BNE     MOVENT
        PULS    B           RESTORE COUNT
        DECB                ANOTHER BLOCK DONE
        BNE     MOVMOR      GO FOR MORE
TABM1   DEC     TABLEN      ONELESS IN LIST
OUTD    RTS                 ALL DONE
```

Figure 9.24: Delete Program — Alphabetical Lists

insert routine from substituting a new entry in place of the existing one. Figure 9.25 shows the structure of an entry.

Further, a directory contains a pointer to the first entry for each letter of the alphabet, in order to facilitate retrieval. It is assumed in the program that the labels are ASCII alphabetic characters. All pointers at the end of the list are set to a NIL value (which has been chosen here to be equal to the table base, minus 1), as this value should never occur within the linked list.

The insertion and deletion programs perform the obvious pointer manipulations. They use the flag INDEXED to indicate if a pointer pointing to an object came from a previous entry in the list or from the directory table. Figure 9.26 shows the data structure.

An application for this data structure would be a computerized address book, where each person is represented by a unique three-letter code (perhaps the usual initials); and the data field would contain a simplified address, plus the telephone number (up to 250 characters). Let us examine the structure in more detail (see Figure 9.25). The entry format also appears in Figure 9.25. As usual, the conventions are:

ENTLEN: total element length (in bytes)
TABASE: address of base list

Here, REFBASE points to the base address of the directory, or the "reference table."

Each two-byte address within this directory points to the first occurrence of the letter to which it corresponds in the list. Thus, each group of entries with an identical first letter in their labels actually forms a separate list within the whole structure. This feature facilitates searching and is

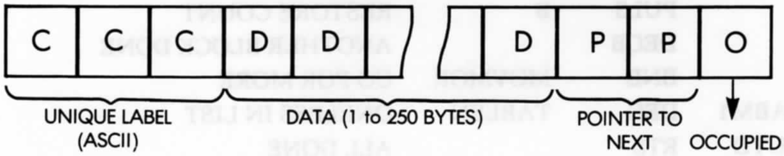


Figure 9.25: Data Structure of a Linked List Entry

analogous to an address book. Note that no data are moved during an insertion or deletion; only pointers are changed, as in every well-behaved linked list structure.

If no entry starting with a specific letter is found, or if there is no entry that alphabetically follows an existing one, the pointers will point to the beginning of the table minus 1 (NIL). The letters in the three-character code are assumed to be alphabetic letters in ASCII code. Changing this would require changing the constant in the PRETAB routine.

The end-of-table marker is set to the value of the beginning of the table minus 1 (NIL). By convention, the NIL pointers, found at the end of a string, or within a directory location that does not point to a string, are set to the value of the table base minus 1, in order to provide a unique identification. Some other convention could be used, but the NIL pointer must never be confused with the address of an entry.

Insertions and deletions are performed in the usual way (see Part I of this chapter), by merely modifying the required pointers. The INDEXED flag is used to indicate if the pointer to the object is in the reference table or in another string element.

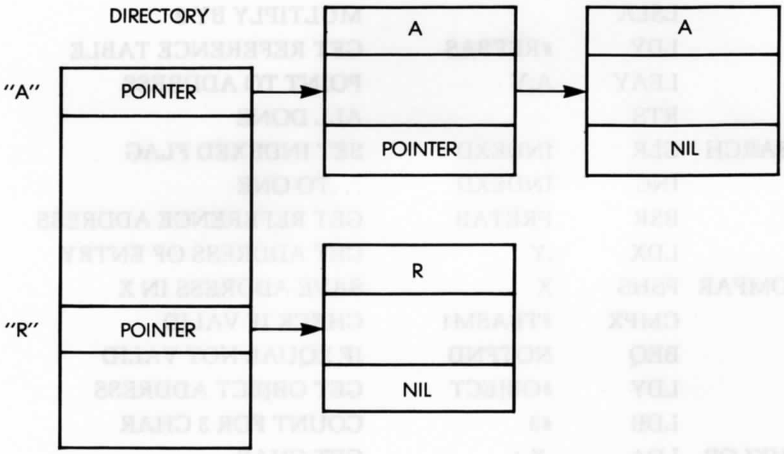


Figure 9.26: Linked List Structure

Searching

The SEARCH program, appearing in Figure 9.27, uses a subroutine called PRETAB. The search principle, as shown in Figure 9.28, is straightforward:

1. Get the directory entry corresponding to the letter of the alphabet in the first position of the OBJECT's label. PRETAB does this.
2. Get the pointer. Access the element. If NIL, the entry does not exist.
3. If not NIL, match the element against the OBJECT. If they are not the same, get the pointer to the next entry down the list.
4. Go back to 2.

```
PRETAB  LDX     #OBJECT       GET OBJECT ADDRESS
        LDA     ,X            GET FIRST LETTER
        SUBA    #$41          REMOVE ASCII OFFSET
        LSLA                  MULTIPLY BY 2
        LDY     #REFBAS       GET REFERENCE TABLE
        LEAY    A,Y           POINT TO ADDRESS
        RTS                   ALL DONE
SEARCH  CLR     INDEXD        SET INDEXED FLAG
        INC     INDEXD        . . .TO ONE
        BSR     PRETAB        GET REFERENCE ADDRESS
        LDX     ,Y            GET ADDRESS OF ENTRY
COMPAR  PSHS    X             SAVE ADDRESS IN X
        CMPX    #TBASM1       CHECK IF VALID
        BEQ     NOTFND        IF EQUAL NOT VALID
        LDY     #OBJECT       GET OBJECT ADDRESS
        LDB     #3            COUNT FOR 3 CHAR
CHKLOP  LDA     ,X+           GET CHAR
        CMPA    ,Y+           COMPARE WITH OBJECT
```

Figure 9.27: Linked List—Search Program (continues)

```
        BLO     NOGOOD       TRY NEXT ENTRY
        BNE     NOTFND       GONE TOO FAR NOT FOUND
        DECB
        BNE     CHKLOP       CHECK 3 CHAR
        PULS    X            RECOVER ORIGINAL ADDRESS
        CLRB                 INDICATE FOUND
        RTS                  DONE WHEN FOUND
NOGOOD  PULS    X            GET ORIGINAL ADDRESS
        TFR     X,U          SAVE AS PREVIOUS ADDRESS
        LEAY    ENTLEN-3,X   POINT TO NEXT POINTER
        LDX     ,Y           GET NEXT POINTER
        CLR     INDEXD       NOT FROM REFERENCE NOW
        BRA     COMPAR       TRY NEXT
NOTFND  PULS    X            GET ORIGINAL ADDRESS
        LDB     #1           NOT FOUND FLAG SET
        RTS                  ALL DONE WHEN NOT FOUND
```

Figure 9.27: Linked List—Search Program (cont.)

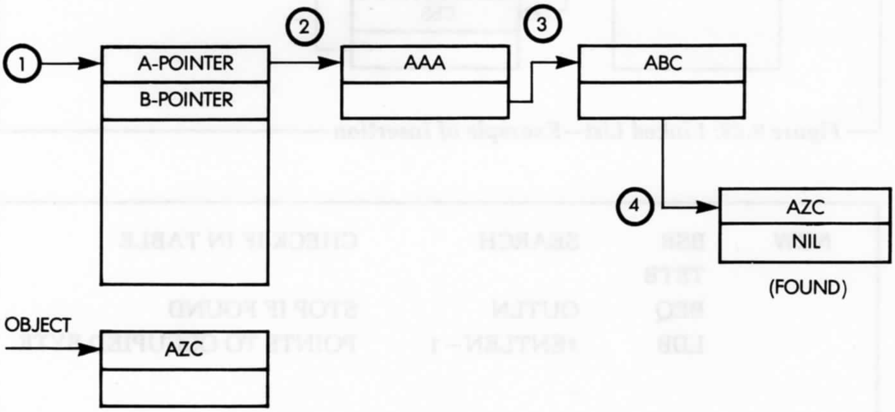


Figure 9.28: Linked List—A Search

Inserting

The insertion is essentially a search followed by an insertion once a NIL has been found (see Figure 9.29). A block of storage for the new entry is allocated by looking for an occupancy marker set at "available." The program, called NEW, appears in Figure 9.30.

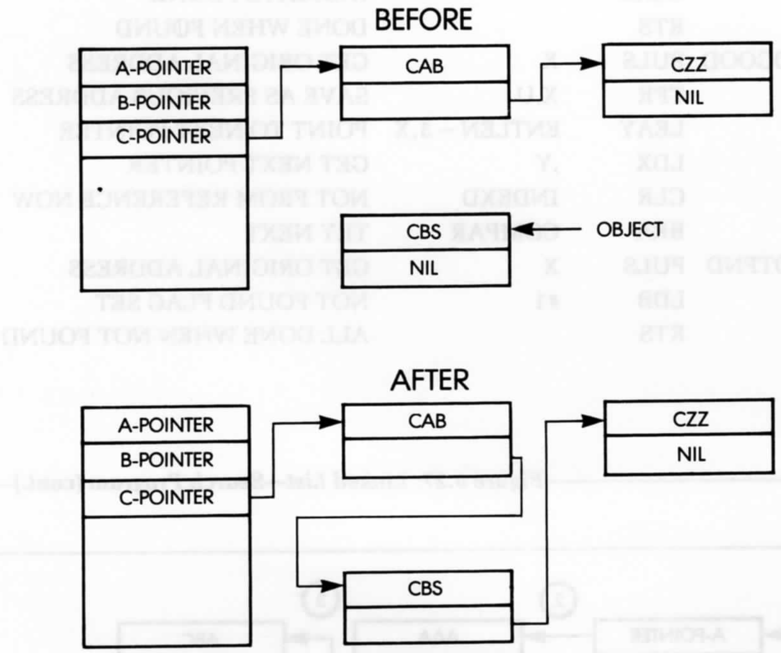


Figure 9.29: Linked List—Example of Insertion

```
NEW     BSR     SEARCH          CHECK IF IN TABLE
        TSTB
        BEQ     OUTLN           STOP IF FOUND
        LDB     #ENTLEN-1       POINTS TO OCCUPIED BYTE
```

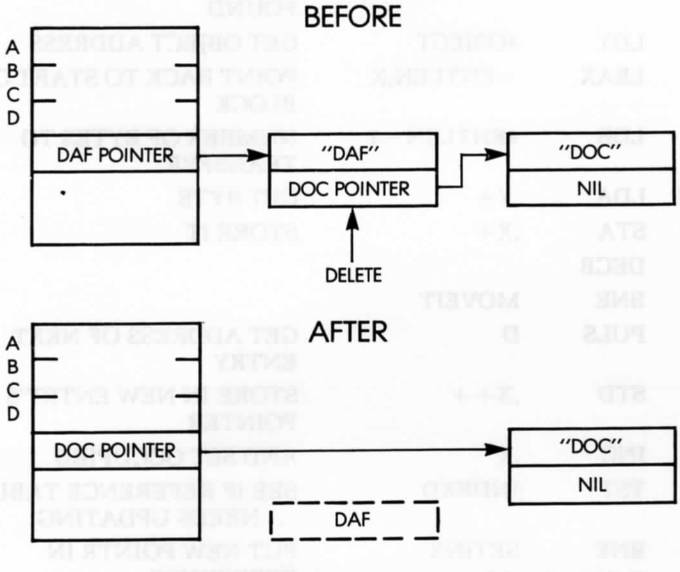
Figure 9.30: NEW Program For a Linked List (continues)

```
        PSHS    X               SAVE POINTER TO
                                FOLLOWING ENTRY
        LDX     #TABASE         START AT BASE OF TABLE
NEXTEN  LEAX    B,X             POINT TO OCCUPIED BYTE
        LDA     ,X+             GET THAT BYTE
        BNE     NEXTEN          SEARCH TIL UNOCCUPIED
                                FOUND
        LDY     #OBJECT         GET OBJECT ADDRESS
        LEAX    -ENTLEN,X       POINT BACK TO START OF
                                BLOCK
        LDB     #ENTLEN-3       NUMBER OF BYTES TO
                                TRANSFER
MOVEIT  LDA     ,Y+             GET BYTE
        STA     ,X+             STORE IT
        DECB
        BNE     MOVEIT
        PULS    D               GET ADDRESS OF NEXT
                                ENTRY
        STD     ,X++            STORE IN NEW ENTRY'S
                                POINTER
        INC     ,X              AND SET OCCUPIED
        TST     INDEXD          SEE IF REFERENCE TABLE
                                . . . NEEDS UPDATING
        BNE     SETINX          PUT NEW POINTR IN
                                REFERENCE
        LEAX    -(ENTLEN-1),X   POINT TO ENTRY AGAIN
        LEAU    ENTLEN-3,U      POINT TO PREVIOUS
                                ENTRY'S POINTER BYTES
        STX     ,U              PUT NEW ENTRY ADDRESS
                                HERE
        RTS                     ALL DONE
SETINX  LEAX    -(ENTLEN-1),X   POINT TO NEW ENTRY
        PSHS    X               SAVE THE ADDRESS
        BSR     PRETAB          GET REFERENCE ADDRESS
        PULS    X               RESTORE ADDRESS
        STX     ,Y              STORE IN REFERENCE
                                TABLE
OUTLN   RTS                     ALL DONE
```

Figure 9.30: NEW Program For a Linked List (cont.)

Deleting

The element is deleted by setting its occupancy marker to "available" and adjusting the pointer to it from the directory or the previous element. An example appears in Figure 9.31. The program, called DELETE, appears in Figure 9.32.



(NOTE: DAF is not erased, but "invisible")

Figure 9.31: Linked List—Example of Deletion

```
DELETE  BSR     SEARCH          GET ADDRESS OF OBJECT
        TSTB
        BNE     OUTLD           QUIT IF NOT FOUND
        LEAX    ENTLEN−3,X      POINT TO POINTER BYTES
        LDY     ,X++            PUT POINTER IN Y
        CLR     ,X              MARK AS UNOCCUPIED
        TST     INDEXD          CHECK IF IN REFERENCE
                                TABLE
        BNE     CHGREF          CHANGE ADDRESS IN
                                TABLE
```

Figure 9.32: DELETE Program For a Linked List (continues)

```
         LEAU    ENTLEN-3,U      POINT TO PREVIOUS ENTRY
                                 LINK POINTER
         STY     ,U              UPDATE LINK POINTER
         RTS                     ALL DONE
CHGREF   PSHS    Y               SAVE FOLLOWING ENTRY'S
                                 ADDRESS
         BSR     PRETAB          GET TABLE ADDRESS
         PULS    X               RESTORE ADDRESS
         STX     ,Y              STORE ADDRESS IN TABLE
OUTLD    RTS                     ALL DONE
```

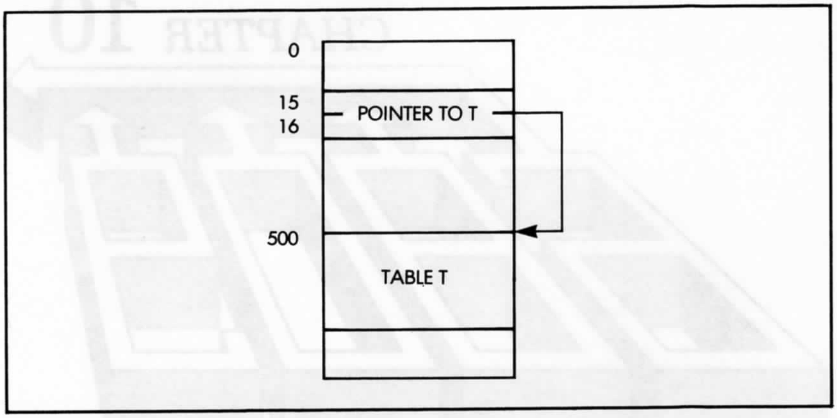
Figure 9.32: DELETE Program For a Linked List (cont.)

SUMMARY

If you are a beginning programmer, it is not essential for you to understand the details of data structure implementation and management. However, as you program more complex problems, you will need to learn more about data structures. The actual examples presented in this chapter have been designed to help you understand and solve all the common problems often encountered with these structures.

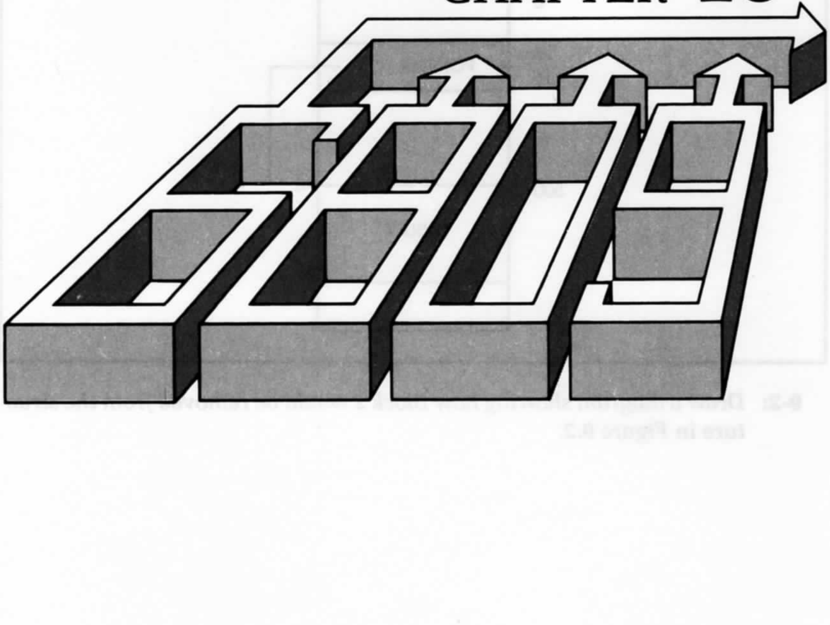
EXERCISES

9-1: *Examine the figure below. At address 15 in the memory, there is a pointer to Table T. Table T starts at address 500. What are the actual contents of the pointer to T?*



9-2: *Draw a diagram showing how Block 2 would be removed from the structure in Figure 9.2.*

CHAPTER 10



PROGRAM DEVELOPMENT

WE HAVE NOW REACHED THE POINT where we should seriously consider developing actual programs. Before proceeding to this task, which is the ultimate goal of our efforts, we should give careful consideration to the options and tools available for developing programs. There are several levels of hardware and software resources to consider. Which level is appropriate depends on the individual application. This chapter presents and evaluates all the available resources.

PROGRAMMING CHOICES

We may write a program either in binary or hexadecimal, in an assembly-level language, or in a high-level language. Let's discuss these alternatives. Figure 10.1 shows the different levels of programming.

Hexadecimal Coding

Most programs are conceived using assembly language mnemonics. The actual translation of such mnemonics into corresponding binary code requires an assembler. When there is no assembler, it is necessary to perform the translation from mnemonics into binary, by hand. Because translating into binary is tedious and error-prone, users often use hexadecimal. Also, many single-board microcomputers require the entry of programs in hexadecimal mode.

(*Note:* in Chapter 1, we showed that one hexadecimal digit represents four binary bits. Therefore, two hexadecimal digits can represent the contents of a byte. (Appendix D offers a table showing the hexadecimal equivalent of the 6809 instructions.))

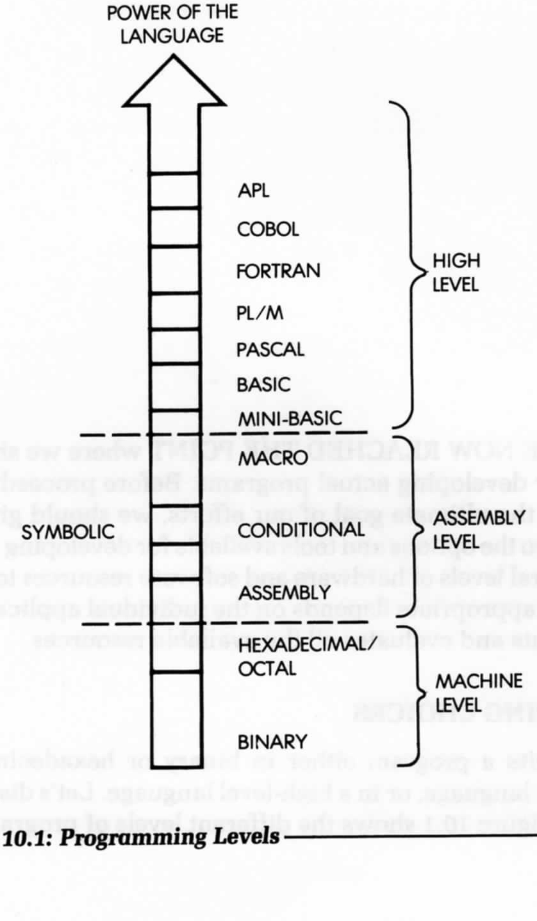


Figure 10.1: Programming Levels

Although it is reasonable to translate a program into hexadecimal by hand for a small number of instructions (for example, 10 to 100), when a program is large, this process becomes tedious and error-prone. Although most single-board microcomputers do not have an assembler and a full alphanumeric keyboard (in order to limit cost), they do provide a hexadecimal keyboard and 7-segment displays for program entry and debugging.

In summary, hexadecimal coding is not a desirable way to enter a program in a computer, it is simply an economical one. The cost of an assembler and the required alphanumeric keyboard is traded-off against the increased time and effort required to enter the program in the memory. Therefore, if it is necessary to use hexadecimal coding, it is wise to first write the program in assembly language mnemonics, then convert it into hexadecimal code. This is because a program written in assembly language is easier to understand and debug.

Assembly Language Programming

Assembly-level programming includes both those programs entered into the system in hexadecimal form and those entered in symbolic assembly-level form. We will now examine the entry of a program directly in its assembly language representation.

When entering a program in assembly language, there must be an assembler program available that will read the mnemonic instructions of the program and translate them into the required bit patterns, using 1 to 5 bytes, as specified by the encoding of the instructions. A good assembler will also offer a number of additional facilities for writing a program. In particular, it might offer *directives* that modify the value of symbols; it might also facilitate symbolic addressing.

(*Note:* by using symbolic labels, it is possible to insert an extra instruction between a branch and the point to which it branches, without rewriting the entire program. The assembler will automatically adjust all the labels during the translation process. In addition, it is possible to debug the program in symbolic form, if an assembler is available.)

Later in this chapter, we will review the various software resources normally available on a system. We will first, however, examine the third alternative: high-level language programming.

High-Level Language

We can also write a program in a high-level language, such as BASIC, APL, or Pascal. A high-level language offers powerful instructions that make programming faster and easier than assembly language. These instructions are then translated by a complex program into the final binary representation that a microcomputer can execute. Typically, each high-level instruction is translated into many individual binary instructions by a program called a *compiler* or an *interpreter*. A compiler translates all the instructions of a program into object code, and then executes the resulting code. By contrast, an interpreter interprets a single instruction, executes it, and then translates the next one, and executes it. An interpreter offers the advantage of interactive response, but results in low efficiency, when compared to a compiler. We will not cover these topics further here. Instead, we will program an actual microprocessor in assembly-level language.

SOFTWARE SUPPORT

We will begin by reviewing the main software facilities available in a complete system for convenient software development. As we proceed, we will summarize the definitions introduced previously and define the remaining important programs available in a software development system.

The *assembler* translates the mnemonic representation of instructions into their binary equivalent. It normally translates one symbolic instruction into one binary instruction (which may occupy between 1 and 5 bytes). The resulting binary code, called the *object code*, is directly executable by the microcomputer. The assembler will also produce a complete mnemonic listing of the program, and a symbol definition list (examples of listings appear later in this chapter). In addition, the assembler will list syntax errors (such as misspelled or illegal instructions), branching errors, duplicate or missing labels. It will not, however, delete *logical* errors. (Such errors are *your* problem.)

A *compiler* translates high-level language instructions into their binary form. An *interpreter*, on the other hand, is similar to a compiler, but it often does not generate an intermediate code; it simply executes the high-level instructions directly.

The *monitor* is the basic program which is indispensable for using the hardware resources of the system. It continuously monitors the input devices for input; it also manages the rest of the devices. As an example, a minimal monitor for a single-board microcomputer, equipped with a keyboard and LEDs, will continuously scan the keyboard for user input, and display the specified contents on the light-emitting diodes. In addition, it must recognize a number of limited commands from the keyboard, such as START, STOP, CONTINUE, LOAD MEMORY, or EXAMINE MEMORY. On a large system that provides complex file management or task scheduling, the monitor is often qualified as the *executive* program. The overall set of facilities is called the *operating system;* and if the files are residing on a disk, the operating system is qualified as the *disk operating system,* or DOS.

An *editor* facilitates the entry and modification of text or programs. It allows the user to conveniently enter, append, and insert characters; add and remove lines of text; and search for characters or strings. The editor is an important resource for convenient and effective text entry.

A *debugger* is a facility necessary for debugging programs. When a program does not work correctly, there may typically be no indication of the cause. In such a case, the programmer may want to insert breakpoints in the program in order to suspend the execution of the program at specified addresses and to examine the contents of registers or memory at these points. The debugger is useful for suspending a program; examining, displaying and modifying the contents of its registers or memory; and then resuming execution. A good debugger also offers a number of additional facilities that allow the programmer to examine data in symbolic form (hexadecimal, binary, or other usual representations), as well as to enter data in this format.

A *loader* or *linking loader* places various blocks of object code at specified positions in the memory and adjusts their respective symbolic pointers, so that they can reference each other.

A *simulator* or an *emulator* program simulates the operation of a device, usually the microprocessor, when developing a program on a simulated processor, prior to placing it on the actual board. Using this approach, it is possible to suspend the program, modify it, and keep it in RAM memory. The disadvantages of a simulator are the following:

1. It usually only simulates the processor itself, not input/output devices.
2. The execution speed is slow, so the instruction cycle times are much longer. It is, therefore, not possible to test real-time devices; and synchronization problems may still occur, even though the logic of the program may be found to be correct.

An *emulator* is essentially a simulator in real time. An emulator uses one processor to simulate another one, and it simulates it in complete detail.

Utility routines are essentially the routines necessary in most applications. They are usually the routines that the user wishes the manufacturer had provided. They may include multiplication, division and other arithmetic operations, as well as block move routines, character tests, input/output device handlers (or drivers), and others. Figure 10.2 shows a memory map for a typical program development system.

THE PROGRAM DEVELOPMENT SEQUENCE

We will now examine a typical sequence for developing an assembly-level program. We will assume that all the usual software facilities are available, so that we may demonstrate their value. If they are not available in a particular system, we can still develop programs, but the convenience will be decreased and, therefore, the amount of time necessary to debug the program is likely to be increased.

Recall that the normal approach for developing an assembly-level program is to, first, design an algorithm and the data structures for the problem to be solved; then, develop a comprehensive set of flowcharts that represent the program flow; and, finally, translate the flowcharts into the assembly-level language for the microprocessor (this is the *coding* phase) and enter the program on the computer. A program can be entered in the RAM memory of the system under the control of the editor. Once entered, we can test a section of the program, such as one or more subroutines.

We must first, however, use the assembler to translate the program into binary code. If the assembler does not already reside in the system, we must load it from an external memory, such as a disk. Assembly will result in an object program that is ready to be executed.

A program is not normally expected to work correctly the first time. To verify its correct operation, we can use the debugger to set a number of breakpoints at crucial locations that will test whether the intermediate results are correct.

Whenever incorrect data is found, an error in the program has been detected. At this point, we should refer to the program listing and verify that the coding is correct. If we cannot find an error in the programming, we should refer to the flowchart—the error might be a logical one.

If we have checked the flowcharts by hand and believe them to be reasonably correct, the error probably stems from the coding. Therefore, we must now modify a section of the program. If the symbolic representation of the program is still in the memory, we can simply re-enter the editor, modify the required lines, and then go through the preceding sequence once again. In some systems, the memory available may not be large enough. In such a case, we will need to flush out the symbolic representation of the program onto a disk or cassette, prior to executing the object code. Naturally, in this case, we will need to reload the symbolic representation of the program from its support medium, prior to entering the editor again.

We can then repeat this procedure, until the results of the program are correct. We stress here that prevention is much more effective than a cure. A correct design typically results in a program that runs correctly very soon after the usual typing mistakes or obvious coding errors have

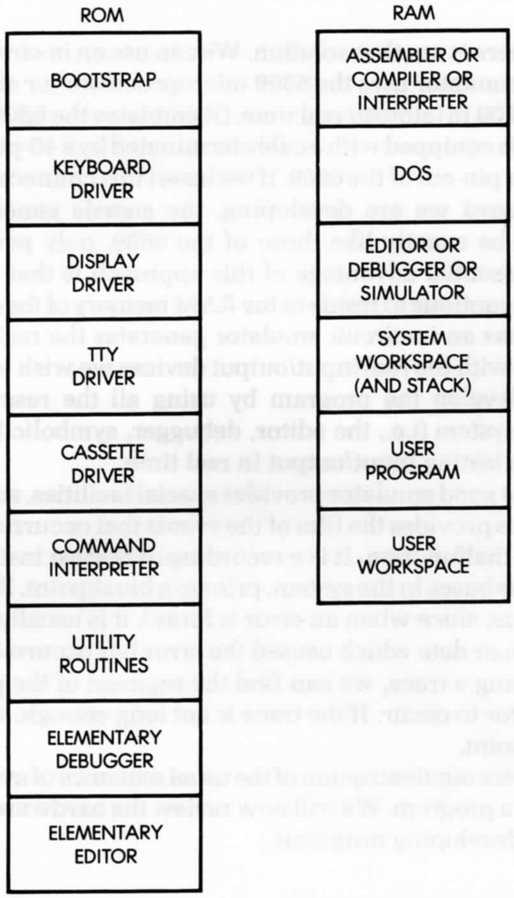


Figure 10.2: A Typical Memory Map

been removed. However, a sloppy design may result in programs that take an extremely long time to debug. The debugging time is generally considered to be much longer than the actual design time. In short, it is always worth investing more time in the design, in order to shorten the debugging phase.

Using the previous approach, we can test the overall organization of the program, but we cannot test it in real time with input/output devices. The direct solution for testing input/output devices is to transfer the program onto EPROM's, install it on the board, and then watch to see if it works.

However, there is another solution. We can use an *in-circuit emulator.* An in-circuit emulator uses the 6809 microprocessor (or any other one) to emulate a 6809 in (almost) real time. (It emulates the 6809 physically.) The emulator is equipped with a cable terminated by a 40-pin connector, identical to the pin-out of the 6809. If we insert this connector on the real application board we are developing, the signals generated by the emulator will be exactly like those of the 6809, only perhaps a little slower. The essential advantage of this approach is that the program under test can continue to reside in the RAM memory of the development system. Because an in-circuit emulator generates the real signals that communicate with the real input/output devices we wish to use, we can continue to develop the program by using all the resources of the development system (i.e., the editor, debugger, symbolic facilities, file system), while testing input/output in real time.

In addition, a good emulator provides special facilities, such as a *trace.* In short, a trace provides the film of the events that occurred prior to the breakpoint or malfunction. It is a recording of the last instructions and the status of the buses in the system, prior to a breakpoint. Such a facility is of great value, since when an error is found, it is usually too late (i.e., the instruction or data which caused the error has occurred prior to the detection). Using a trace, we can find the segment of the program that caused the error to occur. If the trace is not long enough, we can set an earlier breakpoint.

This completes our description of the usual sequence of events involved in developing a program. We will now review the hardware alternatives available for developing programs.

HARDWARE ALTERNATIVES

There are many different hardware systems available for program development. The different systems vary in cost and capabilities. The more expensive and complex the system, the more tools it provides for developing programs.

Single-Board Microcomputer

The single-board microcomputer offers the lowest cost approach to program development. It is normally equipped with a hexadecimal keyboard, plus some function keys, and six LEDs, which can display address and data. Since a single-board microcomputer is equipped with a small amount of memory, an assembler is not usually available. At best, a single-board microcomputer has a small monitor and virtually no editing or debugging facilities, except for a very few commands. All programs must, therefore, be entered in hexadecimal form. They are then displayed in hexadecimal form on the LEDs.

A single-board microcomputer has, in theory, the same hardware power as any other computer. However, because of its restricted memory size and keyboard, it does not support all the usual facilities of a larger system and, therefore, program development is much slower. Because developing programs in hexadecimal format is a tedious task, a single-board microcomputer is best-suited for educational and training purposes, where programs of limited length are developed, and their short length is not an obstacle to programming. Single-boards are probably the least expensive way to learn programming through actual practice. They cannot, however, be used for complex program development, unless additional memory boards are attached, and the usual software aids are made available.

The Development System

A development system is a microcomputer system equipped with a significant amount of RAM memory (32K, 48K), the required input/output devices (a CRT display, a printer, disks, and, usually, a PROM programmer), and, perhaps, an in-circuit emulator. A development system is specifically designed to facilitate program development in an industrial environment. It normally offers all, or most, of the software facilities mentioned in the preceding section. In principle, it is the ideal software development tool.

A limitation of a microcomputer development system is that it may not be capable of supporting a compiler or interpreter. This is because a compiler typically requires a large amount of memory, often more than is available on the system. However, it does offer all of the required facilities for developing programs in assembly-level language. Because development systems sell in relatively small numbers when compared to hobby computers, their cost is significantly higher.

Hobby-Type Microcomputers

The hobby-type microcomputer hardware is naturally analogous to that of a development system. The main difference is that it is normally not equipped with the sophisticated software development aids available on an industrial development system. As an example, many hobby-type microcomputers offer only elementary assemblers and minimal editors and file systems. They normally do not have the facilities to attach a PROM programmer, an in-circuit emulator, or a powerful debugger. Fhey represent, therefore, an intermediate step between the single-board microcomputer and the full microprocessor development system. For a user who wishes to develop programs of modest complexity, they are probably the best compromise. Even though they are quite limited as to their convenience, they can still offer the advantages of low cost and a reasonable array of software development tools.

Time-Sharing System

It is possible to rent terminals that connect to time-sharing networks. These terminals share the time of a larger computer and benefit from the advantages of the large installations. *Cross assemblers* are available for all microcomputers on virtually all commercial time-sharing systems. Formally, a cross assembler is an assembler for microprocessor X, that resides on processor Y. The nature of the computer being used is irrelevant. For example, we can write a program in 6809 assembly-level language, and the cross assembler will translate it into the appropriate binary pattern. The difference, however, is that the program *cannot* be executed at that point. It can only be executed by a simulated processor, if one is available, provided it does not use any input/output resources. Therefore, this solution is used only in industrial environments.

In-House Computer

Whenever a large in-house computer is available, cross assemblers may also be available to facilitate program development. If such a computer offers time-shared service, this option is essentially analogous to the one above. If it offers only batch service, this is probably one of the most inconvenient methods of program development, since submitting programs in batch mode at the assembly level for a microprocessor, results in a very long development time.

Front Panel or No Front Panel?

We can use a front panel, a hardware accessory, to facilitate program debugging. The front panel has traditionally been a tool that conveniently displays the binary contents of a register or memory. However, *all* the functions of the control panel may now be accomplished from a terminal, and the CRT display now offers a service almost equivalent to the control panel, by displaying the binary value of bits. The additional advantage of using the CRT display is that it is possible to switch at will from binary representation to hexadecimal, symbolic, or decimal (if the appropriate conversion routines are available, naturally). The disadvantage of the CRT is that it is necessary to hit several keys to obtain the appropriate display, rather than simply turning a knob. However, since the cost of providing a control panel is quite substantial, most newer microcomputers have abandoned this debugging tool. The value of the control panel is often considered more on the basis of emotional arguments influenced by a user's past experience, than by reason. In other words, the front panel is not indispensable.

Summary of Hardware Resources

We can distinguish three broad categories of hardware systems. Specifically, *single-board microcomputer* is available for those who have only a minimal budget and want to learn how to program. Using a single-board microcomputer, it is possible to develop all the simple programs in this book and many more. Eventually, however, the user will feel the limitations of this approach; for example, when it is necessary to develop programs of more than a few hundred instructions.

A *full development system* is available for the industrial user. Any solution short of the full development system will cause a significantly longer development time. The trade-off is clear: hardware resources versus programming time. Naturally, if the programs being developed are simple, there are less expensive approaches. But if they are complex, it is difficult to justify any hardware savings when buying a development system, since the programming costs will be by far the dominant cost of the project.

For a personal computerist, a *hobby-type microcomputer* typically offers sufficient, although minimal, facilities. Good development software is now becoming available for many of the hobby computers.

Let us now analyze in more detail the most indispensable resource: the assembler.

THE ASSEMBLER

We will now present the formal syntax or definition of assembly-level language. An assembler allows the convenient symbolic representation of a user program, and makes it simple for the assembler program to convert these mnemonics into their binary representation.

Assembler Fields

When typing in a program for the assembler, we have seen that several fields are used. They are:

- *the label field*, which is optional, and may contain a symbolic address for the instruction that follows.
- *the instruction field*, which includes the opcode and any operands. (A separate operand field may be distinguished.)
- *the comment field*, which is optional, and intended to clarify the program.

These fields appear on the programming form in Figure 10.3.

Once a program is fed to the assembler, the assembler produces a *listing* of it. When generating a listing, the assembler will provide four additional fields, usually on the left of the page. An example of assembler output appears in Figure 10.4.

On the far left of the output is the line number. Each line typed is assigned a symbolic line number. The next field to the right is the actual address field, which shows in hexadecimal, the value of the program counter that points to that instruction. Even further to the right is the hexadecimal representation of the instruction, and, finally, to the right of the hexadecimal representation appears the number of cycles required to execute the instruction.

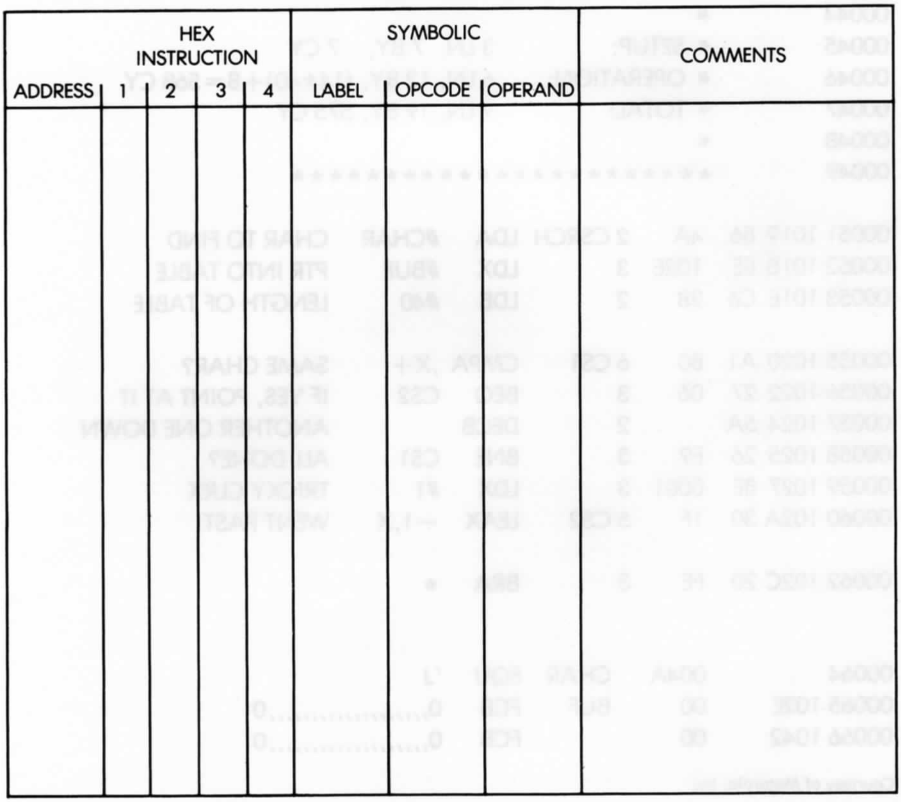
We have now shown one possible use of an assembler. Even if we are designing programs for a single-board microcomputer that accepts only hexadecimal, we should still write the program in assembly-level language, providing we have access to a system equipped with an assembler. We can then run the programs on the system, using the assembler.

The assembler automatically generates the correct hexadecimal codes on the system. This shows, in a simple example, the value of additional software resources.

Tables

When the assembler translates the symbolic program into its binary representation, it performs two essential tasks:

1. It translates the mnemonic instructions into their binary encoding.
2. It translates the symbols used for constants and addresses into their binary representations.

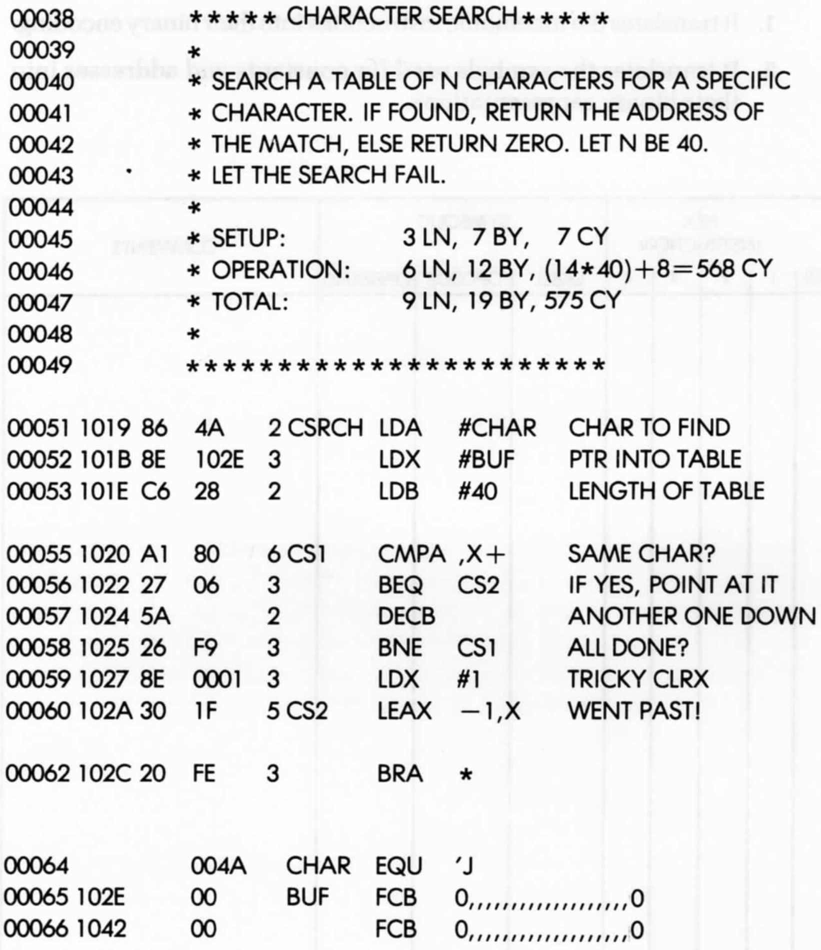


ADDRESS	HEX INSTRUCTION				SYMBOLIC			COMMENTS
	1	2	3	4	LABEL	OPCODE	OPERAND	

Figure 10.3: Microprocessor Programming Form

To facilitate program debugging, the assembler shows, at the end of the listing, the equivalence between the symbol used and its hexadecimal value. This is called the *symbol table.*

Some symbol tables not only list the symbol and its value, but also the line numbers where the symbol occurs—thereby providing an additional facility.



```
00038           ***** CHARACTER SEARCH*****
00039           *
00040           * SEARCH A TABLE OF N CHARACTERS FOR A SPECIFIC
00041           * CHARACTER. IF FOUND, RETURN THE ADDRESS OF
00042           * THE MATCH, ELSE RETURN ZERO. LET N BE 40.
00043     .     * LET THE SEARCH FAIL.
00044           *
00045           * SETUP:          3 LN,  7 BY,    7 CY
00046           * OPERATION:      6 LN, 12 BY, (14*40)+8=568 CY
00047           * TOTAL:          9 LN, 19 BY, 575 CY
00048           *
00049           ***********************

00051 1019 86  4A    2 CSRCH LDA   #CHAR    CHAR TO FIND
00052 101B 8E  102E  3       LDX   #BUF     PTR INTO TABLE
00053 101E C6  28    2       LDB   #40      LENGTH OF TABLE

00055 1020 A1  80    6 CS1   CMPA  ,X+      SAME CHAR?
00056 1022 27  06    3       BEQ   CS2      IF YES, POINT AT IT
00057 1024 5A        2       DECB           ANOTHER ONE DOWN
00058 1025 26  F9    3       BNE   CS1      ALL DONE?
00059 1027 8E  0001  3       LDX   #1       TRICKY CLRX
00060 102A 30  1F    5 CS2   LEAX  -1,X     WENT PAST!

00062 102C 20  FE    3       BRA   *


00064          004A  CHAR    EQU   'J
00065 102E     00    BUF     FCB   0,,,,,,,,,,,,,,,,,,,0
00066 1042     00            FCB   0,,,,,,,,,,,,,,,,,,,0
```

Courtesy of Motorola, Inc.

—*Figure 10.4: Assembler Output—An Example*—

Error Messages

During the assembly process, the assembler detects syntax errors and includes them as part of the final listing. Typical diagnostics include: undefined symbols, label already defined, illegal opcode, illegal address, and illegal addressing mode. Many additional diagnostics are desirable, and are usually provided. Such features vary with each assembler.

The Assembly Language

We have already discussed *opcodes*. We will define here the symbols, constants, and operators that we can use as part of the assembler syntax.

Symbols

Symbols are used to represent numerical values, either data or addresses. Symbols may include up to six characters, and must start with an alphabetic character or a period. The characters are restricted to letters of the alphabet, numbers, a ".", and a "$". Also, we may not choose names identical to the opcodes utilized by the 6809, the names of the registers (A, B, D, X, Y, U, S, PC, DP, and PCR), or the various names used as pseudo-operators by the assembler. The names of these assembler directives are listed later in the corresponding section.

Assigning a Value to a Symbol

Labels are special symbols with values that do not need to be defined by the programmer. The value is automatically defined by the assembler program when it finds that label. Thus, the label value automatically corresponds to the number of the line where it appears. There are special pseudo-instructions available for forcing a new starting value for labels, or for assigning them a specific value. However, any other symbols used for constants or memory addresses must be defined by the programmer, prior to use.

We can use a special assembler *directive* to assign a value to a symbol. This directive is essentially an instruction to the assembler that will not be translated into an executable statement. For example, the constant LOG is defined as:

```
LOG       EQU       $302
```

This assigns the value 302 hexadecimal to the symbol LOG. We examine the assembler directives in detail in a later section.

Constants or Literals

Constants may be expressed in decimal, hexadecimal, octal, binary, or as alphanumeric strings. To differentiate between the bases used to represent numbers, we must use a symbol. To load 0 into accumulator A, we simply write:

```
        LDA     #0
```

The absence of a symbol always means decimal.

A hexadecimal number is preceded by the symbol $ or terminated by H. To load the value FF into A, we write:

```
        LDA     #$FF
```

or

```
        LDA     #0FFH
```

An octal symbol is preceded by an @, or terminated by a Q. A binary symbol is preceded by a %, or terminated by a B. For example, in order to load the value 11111111 into A, we write:

```
        LDA     #%11111111
```

We may also use literal ASCII characters in the literal field. The ASCII symbol must be preceded by a single quote. For example, to load the symbol S into A, we write:

```
        LDA     #'S
```

Operators

To further facilitate the writing of symbolic programs, assemblers allow the use of operators. At a minimum, they usually allow plus and minus, so that the user can specify, for example:

```
        LDA     ADDRESS
        LDB     ADDRESS + 1
```

It is important to understand that the expression ADDRESS + 1 is computed by the assembler, in order to determine the actual memory address that must be inserted as the binary equivalent. An operator is computed *at assembly time*, not at program-execution time.

In addition, there may be other operators available, such as multiply and divide—a convenience when accessing tables in memory. There may also be available more specialized operators, such as greater than and less than, which truncate a two-byte value, respectively, into its high and low byte.

Naturally, an expression must *evaluate* to a positive value. Negative numbers may normally not be used and should be expressed in hexadecimal format.

Finally, it has traditionally been the case that a special symbol represents the current value of the address of the line: "*". This symbol should be interpreted as meaning "current location" (value of PC).

Expressions

The 6809 assembler specifications allow a wide range of expressions with arithmetic and logical operations. Figure 10.5 displays these operations. Let's examine the order of precedence of the various operations:

— Operations within parenthesis are evaluated first.

— Multiplication, division and all of the two-character operations take precedence over addition and subtraction.

— Operators with the same precedence are evaluated from left to right.

Addressing Modes

It is necessary to distinguish the different addressing modes used in the 6809 with special symbols. If a symbol is not used, the assembler normally

OPERATOR	FUNCTION
+	Addition
—	Subtraction
*	Multiplication
/	Division
!^	Exponentiation
!.	Logical AND
!+	Logical OR
!X	Logical Exclusive OR
!<	Shift Left
!>	Shift Right
!L	Rotate Left
!R	Rotate Right

Figure 10.5: Assembler Operators

chooses direct or extended addressing. (*Note:* the assembler chooses direct page addressing whenever possible.) To force direct page addressing, we must put the symbol "<" before the operand. Similarly, we can force extended addressing by putting the symbol ">" before the operand.

The symbol "#" indicates immediate mode. We can use the general form:

```
        OFFSET,R
```

to indicate indexed addressing. By preceding the OFFSET with a "<", the assembler will use an 8-bit offset mode. Placing the symbol ">" before the offset forces a 16-bit offset mode. The assembler will always try to use a zero, 5-bit, or 8-bit offset, if it is not restricted.

The form:

```
        DEST,PCR
```

instructs the assembler to use the indexed mode with the PC. The assembler calculates the relative distance from the present PC and the symbol, DEST. This constant is then added at run time to the PC to fetch the operand.

The symbols + and ++ after an index register, indicate auto increment mode. The symbols – or –– before an index register indicate autodecrement mode. Finally, any operand contained in square brackets "[ ]" indicates indirect addressing.

Assembler Directives

Directives are special orders given by the programmer to the assembler, which result in storing values into symbols or in memory, or in controlling the execution of the assembler. To provide a specific example, we will now review the eight assembler directives available on the 6809 assembler. We begin with:

```
          ORG       nn
```

This directive sets the assembler address counter to the value nn. In other words, the first executable instruction encountered after this directive will reside at the value nn. This directive can be used to locate different segments of a program at different memory locations.

The directive

```
LABEL     EQU       nn
```

assigns a value to a label.

The directive

```
        FCB      n
```

written out as form constant byte, assigns the 8-bit value n to a byte residing at the current program counter. A label may be used with FCB.

The form double byte constant directive

```
        FDB      nn
```

assigns the value nn to the two-byte memory word residing at the current program counter. A label may be used with FDB.

The form constant character string directive

```
        FCC      /string/
```

places the 7-bit ASCII characters in "string" in successive bytes in memory. The character "/" is a delimiter for the string. We can use a number preceding the string to signify the number of characters in the string in place of the "/" delimiter.

The directive

```
        FCC      5,START
```

puts the five characters in ASCII code in successive memory locations. A label may be used with FCC.

The reserve memory bytes directive

```
        RMB      nn
```

allocates nn bytes of space at the present location in the program. A label may be used with RMB.

The set direct page directive

```
        SETDP    n
```

tells the assembler which page of memory to use for the direct page addressing mode. The default page is zero. This directive does not insert instructions to set the register; that must be done by the user. An example follows:

```
        LDA      #DPAGE
        TFR      A,DP
        SETDP    DPAGE
```

The end directive

```
        END
```

marks the end of the program. The assembler does not look for any statement following this directive.

SUMMARY

This chapter has presented the techniques and hardware and software tools required to develop a program; it has also examined various tradeoffs and alternatives. These techniques range, at the hardware level, from a single-board microcomputer to a full development system, and, at the software level, from binary coding to high-level programming.

CONCLUSION

In this book, we have covered all the important aspects of programming the 6809, ranging from the basic definitions and concepts, to the internal manipulation of the 6809 registers, the management of input/output devices, and the implementation of software development aids. These concepts apply to other microprocessors, as well as to the 6809.

What is the next step? There is no substitute for actual experience. Once you have studied the examples in this book and have completed the exercises, you should be ready to move ahead and create *your own programs*.

APPENDIX A

HEXADECIMAL CONVERSION TABLE

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

APPENDIX B

ASCII CONVERSION TABLE

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	—	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	″	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	′	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	—	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

THE ASCII SYMBOLS

NUL	— Null
SOH	— Start of Heading
STX	— Start of Text
ETX	— End of Text
EOT	— End of Transmission
ENQ	— Enquiry
ACK	— Acknowledge
BEL	— Bell
BS	— Backspace
HT	— Horizontal Tabulation
LF	— Line Feed
VT	— Vertical Tabulation
FF	— Form Feed
CR	— Carriage Return
SO	— Shift Out
SI	— Shift In

DLE	— Data Link Escape
DC	— Device Control
NAK	— Negative Acknowledge
SYN	— Synchronous Idle
ETB	— End of Transmission Block
CAN	— Cancel
EM	— End of Medium
SUB	— Substitute
ESC	— Escape
FS	— File Separator
GS	— Group Separator
RS	— Record Separator
US	— Unit Separator
SP	— Space (Blank)
DEL	— Delete

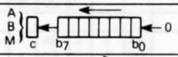
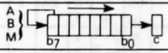
APPENDIX C

DECIMAL TO BCD CONVERSION TABLE

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	91	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

APPENDIX D

6809 INSTRUCTION SET

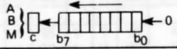
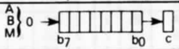
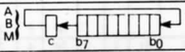
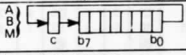
Instruction	Forms	Addressing Modes															Description	5	3	2	1	0
		Immediate			Direct			Indexed			Extended			Inherent				H	N	Z	V	C
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#						
ABX														3A	3	1	B + X → X (Unsigned)	•	•	•	•	•
ADC	ADCA	89	2	2	99	4	2	A9	4+	2+	B9	5	3				A + M + C → A	↕	↕	↕	↕	↕
	ADCB	C9	2	2	D9	4	2	E9	4+	2+	F9	5	3				B + M + C → B	↕	↕	↕	↕	↕
ADD	ADDA	8B	2	2	9B	4	2	AB	4+	2+	BB	5	3				A + M → A	↕	↕	↕	↕	↕
	ADDB	CB	2	2	DB	4	2	EB	4+	2+	FB	5	3				B + M → B	↕	↕	↕	↕	↕
	ADDD	C3	4	3	D3	6	2	E3	6+	2+	F3	7	3				D + M:M + 1 → D	•	↕	↕	↕	↕
AND	ANDA	84	2	2	94	4	2	A4	4+	2+	B4	5	3				A Λ M → A	•	↕	↕	0	•
	ANDB	C4	2	2	D4	4	2	E4	4+	2+	F4	5	3				B Λ M → B	•	↕	↕	0	•
	ANDCC	1C	3	2													CC Λ IMM → CC					7
ASL	ASLA													48	2	1		8	↕	↕	↕	↕
	ASLB													58	2	1		8	↕	↕	↕	↕
	ASL				08	6	2	68	6+	2+	78	7	3					8	↕	↕	↕	↕
ASR	ASRA													47	2	1		8	↕	↕	•	↕
	ASRB													57	2	1		8	↕	↕	•	↕
	ASR				07	6	2	67	6+	2+	77	7	3					8	↕	↕	•	↕
BIT	BITA	85	2	2	95	4	2	A5	4+	2+	B5	5	3				Bit Test A (M Λ A)	•	↕	↕	0	•
	BITB	C5	2	2	D5	4	2	E5	4+	2+	F5	5	3				Bit Test B (M Λ B)	•	↕	↕	0	•
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	6F	6+	2+	7F	7	3				0 → M	•	0	1	0	0
CMP	CMPA	81	2	2	91	4	2	A1	4+	2+	B1	5	3				Compare M from A	8	↕	↕	↕	↕
	CMPB	C1	2	2	D1	4	2	E1	4+	2+	F1	5	3				Compare M from B	8	↕	↕	↕	↕
	CMPD	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M M + 1 from D	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPS	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M M + 1 from S	•	↕	↕	↕	↕
		8C			9C			AC			BC											
	CMPU	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M:M + 1 from U	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPX	8C	4	3	9C	6	2	AC	6+	2+	BC	7	3				Compare M:M + 1 from X	•	↕	↕	↕	↕
	CMPY	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M:M + 1 from Y	•	↕	↕	↕	↕
		8C			9C			AC			BC											
COM	COMA													43	2	1	$\overline{A}$ → A	•	↕	↕	0	1
	COMB													53	2	1	$\overline{B}$ → B	•	↕	↕	0	1
	COM				03	6	2	63	6+	2+	73	7	3				$\overline{M}$ → M	•	↕	↕	0	1
CWAI		3C	≥20	2													CC Λ IMM → CC Wait for Interrupt					7
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	6A	6+	2+	7A	7	3				M − 1 → M	•	↕	↕	↕	•
EOR	EORA	88	2	2	98	4	2	A8	4+	2+	B8	5	3				A ⊻ M → A	•	↕	↕	0	•
	EORB	C8	2	2	D8	4	2	E8	4+	2+	F8	5	3				B ⊻ M → B	•	↕	↕	0	•
EXG	R1, R2													1E	8	2	R1 → R2[2]	•	•	•	•	•
INC	INCA													4C	2	1	A + 1 → A	•	↕	↕	↕	•
	INCB													5C	2	1	B + 1 → B	•	↕	↕	↕	•
	INC				0C	6	2	6C	6+	2+	7C	7	3				M + 1 → M	•	↕	↕	↕	•
JMP					0E	3	2	6E	3+	2+	7E	4	3				EA[3] → PC	•	•	•	•	•
JSR					9D	7	2	AD	7+	2+	BD	8	3				Jump to Subroutine	•	•	•	•	•
LD	LDA	86	2	2	96	4	2	A6	4+	2+	B6	5	3				M → A	•	↕	↕	0	•
	LDB	C6	2	2	D6	4	2	E6	4+	2+	F6	5	3				M → B	•	↕	↕	0	•
	LDD	CC	3	3	DC	5	2	EC	5+	2+	FC	6	3				M M + 1 → D	•	↕	↕	0	•
	LDS	10	4	4	10	6	3	10	6+	3+	10	7	4				M M + 1 → S	•	↕	↕	0	•
		CE			DE			EE			FE											
	LDU	CE	3	3	DE	5	2	EE	5+	2+	FE	6	3				M:M + 1 → U	•	↕	↕	0	•
	LDX	8E	3	3	9E	5	2	AE	5+	2+	BE	6	3				M:M + 1 → X	•	↕	↕	0	•
	LDY	10	4	4	10	6	3	10	6+	3+	10	7	4				M M + 1 → Y	•	↕	↕	0	•
		8E			9E			AE			BE											
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	•	•	↕	•	•

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 result
V	Overflow, 2's complement
C	Carry from ALU
↕	Test and set if true, cleared otherwise
•	Not Affected
CC	Condition Code Register
:	Concatenation
V	Logical or
Λ	Logical and
⊻	Logical Exclusive or

Courtesy of Motorola, Inc.

6809 INSTRUCTION SET

		Addressing Modes																5	3	2	1	0
		Immediate			Direct			Indexed[1]			Extended			Inherent								
Instruction	Forms	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Description	H	N	Z	V	C
LSL	LSLA													48	2	1		•	↕	↕	↕	↕
	LSLB													58	2	1		•	↕	↕	↕	↕
	LSL				08	6	2	68	6+	2+	78	7	3					•	↕	↕	↕	↕
LSR	LSRA													44	2	1		•	0	↕	•	↕
	LSRB													54	2	1		•	0	↕	•	↕
	LSR				04	6	2	64	6+	2+	74	7	3					•	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	60	6+	2+	70	7	3				$\overline{M}$ + 1 → M	8	↕	↕	↕	↕
NOP														12	2	1	No Operation	•	•	•	•	•
OR	ORA	8A	2	2	9A	4	2	AA	4+	2+	BA	5	3				A V M → A	•	↕	↕	0	•
	ORB	CA	2	2	DA	4	2	EA	4+	2+	FA	5	3				B V M → B	•	↕	↕	0	•
	ORCC	1A	3	2													CC V IMM → CC				7	
PSH	PSHS	34	5+[4]	2													Push Registers on S Stack	•	•	•	•	•
	PSHU	36	5+[4]	2													Push Registers on U Stack	•	•	•	•	•
PUL	PULS	35	5+[4]	2													Pull Registers from S Stack	•	•	•	•	•
	PULU	37	5+[4]	2													Pull Registers from U Stack	•	•	•	•	•
ROL	ROLA													49	2	1		•	↕	↕	↕	↕
	ROLB													59	2	1		•	↕	↕	↕	↕
	ROL				09	6	2	69	6+	2+	79	7	3					•	↕	↕	↕	↕
ROR	RORA													46	2	1		•	↕	↕	•	↕
	RORB													56	2	1		•	↕	↕	•	↕
	ROR				06	6	2	66	6+	2+	76	7	3					•	↕	↕	•	↕
RTI														3B	6/15	1	Return From Interrupt					7
RTS														39	5	1	Return from Subroutine	•	•	•	•	•
SBC	SBCA	82	2	2	92	4	2	A2	4+	2+	B2	5	3				A − M − C → A	8	↕	↕	↕	↕
	SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B − M − C → B	8	↕	↕	↕	↕
SEX														1D	2	1	Sign Extend B into A	•	↕	↕	0	•
ST	STA				97	4	2	A7	4+	2+	B7	5	3				A → M	•	↕	↕	0	•
	STB				D7	4	2	E7	4+	2+	F7	5	3				B → M	•	↕	↕	0	•
	STD				DD	5	2	ED	5+	2+	FD	6	3				D → M M + 1	•	↕	↕	0	•
	STS				10 DF	6	3	10 EF	6+	3+	10 FF	7	4				S → M M + 1	•	↕	↕	0	•
	STU				DF	5	2	EF	5+	2+	FF	6	3				U → M M + 1	•	↕	↕	0	•
	STX				9F	5	2	AF	5+	2+	BF	6	3				X → M M + 1	•	↕	↕	0	•
	STY				10 9F	6	3	10 AF	6+	3+	10 BF	7	4				Y → M M + 1	•	↕	↕	0	•
SUB	SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A − M → A	8	↕	↕	↕	↕
	SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B − M → B	8	↕	↕	↕	↕
	SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D − M M + 1 → D	•	↕	↕	↕	↕
SWI	SWI[6]													3F	19	1	Software Interrupt 1	•	•	•	•	•
	SWI[6]													10 3F	20	2	Software Interrupt 2	•	•	•	•	•
	SWI[6]													11 3F	20	1	Software Interrupt 3	•	•	•	•	•
SYNC														13	≥4	1	Synchronize to Interrupt	•	•	•	•	•
TFR	R1, R2													1F	6	2	R1 → R2[2]	•	•	•	•	•
TST	TSTA													4D	2	1	Test A	•	↕	↕	0	•
	TSTB													5D	2	1	Test B	•	↕	↕	0	•
	TST				0D	6	2	6D	6+	2+	7D	7	3				Test M	•	↕	↕	0	•

Notes:

1. This column gives a base cycle and byte count. To obtain total count, add the values obtained from the INDEXED ADDRESSING MODE table, Table 2.
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 (Branch instructions).
6. SWI sets I and F bits. SWI2 and SWI3 do not affect I and 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.

Courtesy of Motorola, Inc.

6809 INSTRUCTION SET

Branch Instructions

		Addressing Mode								
		Relative				5	3	2	1	0
Instruction	Forms	OP	~ 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=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	•	•	•	•	•
BLE	BLE	2F	3	2	Branch ≤ Zero	•	•	•	•	•
	LBLE	10 2F	5(6)	4	Long Branch ≤ Zero	•	•	•	•	•
BLO	BLO	25	3	2	Branch lower	•	•	•	•	•
	LBLO	10 25	5(6)	4	Long Branch Lower	•	•	•	•	•

		Addressing Mode								
		Relative				5	3	2	1	0
Instruction	Forms	OP	~ 5	#	Description	H	N	Z	V	C
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	•	•	•	•	•

SIMPLE BRANCHES

	OP	~	#
BRA	20	3	2
LBRA	16	5	3
BRN	21	3	2
LBRN	1021	5	4
BSR	8D	7	2
LBSR	17	9	3

SIMPLE CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
N=1	BMI	2B	BPL	2A
Z=1	BEQ	27	BNE	26
V=1	BVS	29	BVC	28
C=1	BCS	25	BCC	24

SIGNED CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
r>m	BGT	2E	BLE	2F
r≥m	BGE	2C	BLT	2D
r=m	BEQ	27	BNE	26
r≤m	BLE	2F	BGT	2E
r<m	BLT	2D	BGE	2C

UNSIGNED CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
r>m	BHI	22	BLS	23
r≥m	BHS	24	BLO	25
r=m	BEQ	27	BNE	26
r≤m	BLS	23	BHI	22
r<m	BLO	25	BHS	24

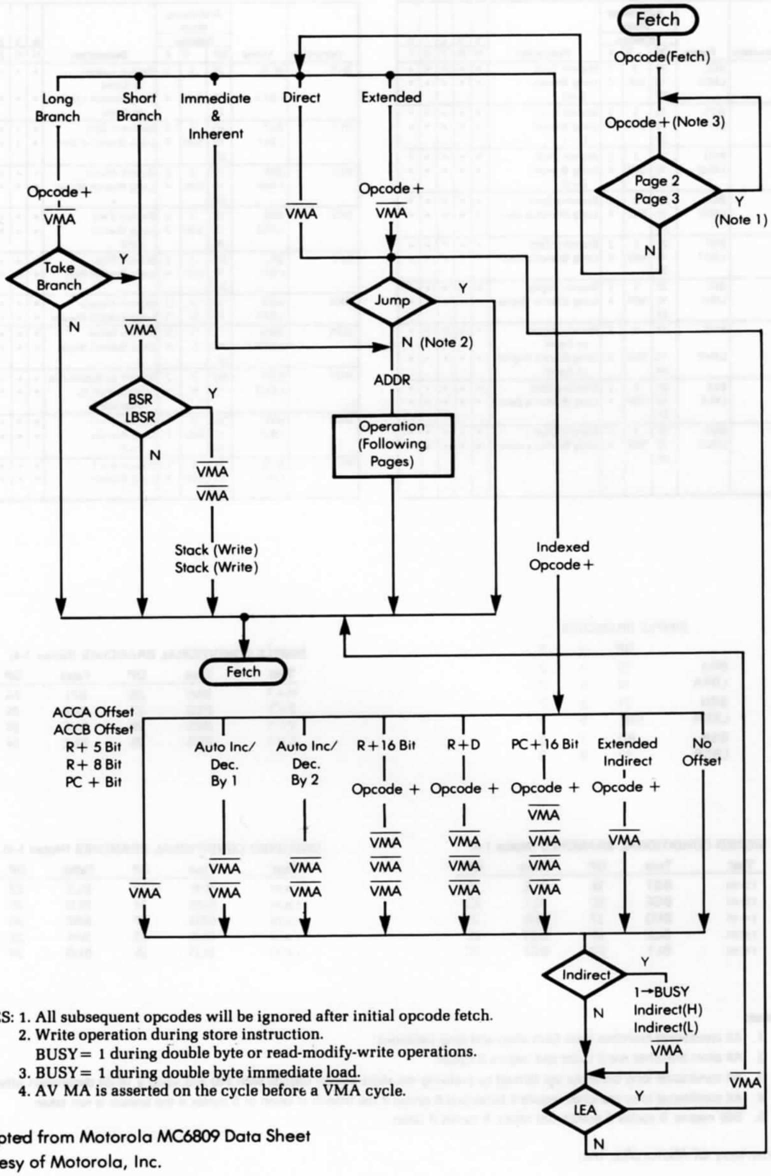
Notes:

1. All conditional branches have both short and long variations.
2. All short branches are 2 bytes and require 3 cycles.
3. All conditional long branches are formed by prefixing the short branch opcode with $10 and using a 16-bit destination offset.
4. All conditional long branches require 4 bytes and 6 cycles if the branch is taken or 5 cycles if the branch is not taken.
5. 5(6) means: 5 cycles if branch not taken, 6 cycles if taken.

Courtesy of Motorola, Inc.

APPENDIX E

ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE

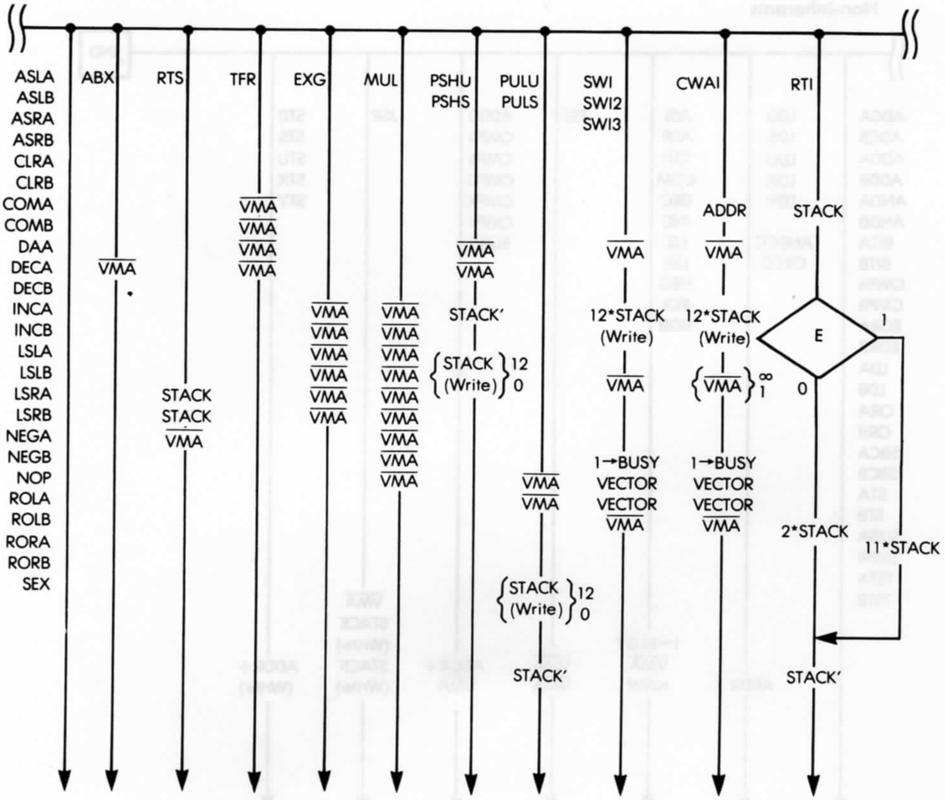


NOTES: 1. All subsequent opcodes will be ignored after initial opcode fetch.
2. Write operation during store instruction.
BUSY= 1 during double byte or read-modify-write operations.
3. BUSY= 1 during double byte immediate load.
4. AV MA is asserted on the cycle before a $\overline{\text{VMA}}$ cycle.

*Adapted from Motorola MC6809 Data Sheet
Courtesy of Motorola, Inc.

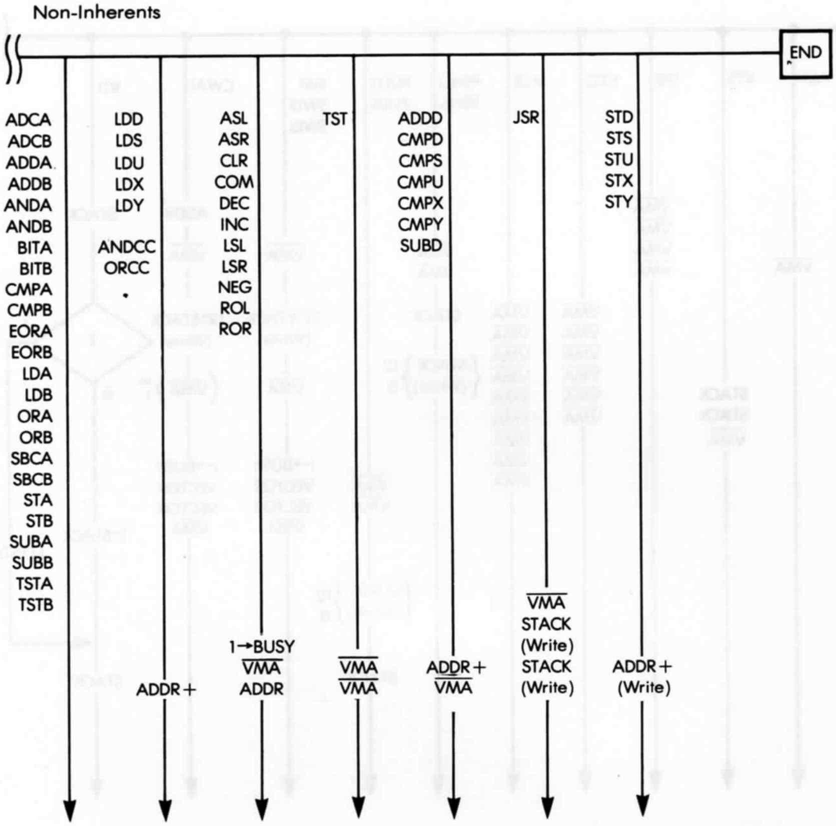
ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE

Inherent Page



Courtesy of Motorola, Inc.

ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE



Courtesy of Motorola, Inc.

APPENDIX F
INDIRECT ADDRESSING MODE POSTBYTES

Type	Forms	Non Indirect				Indirect			
		Assembler Form	Postbyte OP Code	$\underset{\sim}{+}$	$\underset{\#}{+}$	Assembler Form	Postbyte OP Code	$\underset{\sim}{+}$	$\underset{\#}{+}$
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

$\underset{\sim}{+}$ and $\underset{\#}{+}$ indicate the number of additional cycles and bytes for the particular variation.

Courtesy of Motorola, Inc.

BIBLIOGRAPHY

Leventhal, Lance A., *Assembly Language Programming*, Berkeley, California: Osborne/McGraw-Hill, 1981.

Motorola, Inc. *MC 6809–MC 6809E Microprocessor Programmming Manual. 1981.*

Staugaard, Andrew C., Jr., *6809 Microcomputer Programming & Interfacing, With Experiments*, Indianapolis, Indiana: Howard W. Sams & Ço., Inc., 1981.

Warren, Carl R., *The MC6809 Cookbook*, Blue Ridge Summit, Pennsylvania: Tab Books, Inc., 1981.

Zaks, Rodnay. *From Chips To Systems: An Introduction To Microprocessors*, Ref. C201A. Berkeley, California: Sybex, Inc., 1981.

Zaks, Rodnay, and Lesea, Austin. *Microprocessor Interfacing Techniques*, 3rd ed., Ref. C207. Berkeley, California: Sybex, Inc., 1979.

INDEX

The SYBEX Library

YOUR FIRST COMPUTER

by Rodnay Zaks 264 pp., 150 illustr., Ref. 0-045

The most popular introduction to small computers and their peripherals: what they do and how to buy one.

DON'T (or How to Care for Your Computer)

by Rodnay Zaks 222 pp., 100 illustr., Ref. 0-065

The correct way to handle and care for all elements of a computer system, including what to do when something doesn't work.

INTERNATIONAL MICROCOMPUTER DICTIONARY

140 pp., Ref. 0-067

All the definitions and acronyms of microcomputer jargon defined in a handy pocket-size edition. Includes translations of the most popular terms into ten languages.

FROM CHIPS TO SYSTEMS: AN INTRODUCTION TO MICROPROCESSORS

by Rodnay Zaks 558 pp., 400 illustr. Ref. 0-063

A simple and comprehensive introduction to microprocessors from both a hardware and software standpoint: what they are, how they operate, how to assemble them into a complete system.

INTRODUCTION TO WORD PROCESSING

by Hal Glatzer 216 pp., 140 illustr., Ref. 0-076

Explains in plain language what a word processor can do, how it improves productivity, how to use a word processor and how to buy one wisely.

INTRODUCTION TO WORDSTAR™

by Arthur Naiman 208 pp., 30 illustr., Ref. 0-077

Makes it easy to learn how to use WordStar, a powerful word processing program for personal computers.

DOING BUSINESS WITH VISICALC®

by Stanley R. Trost 200 pp., Ref. 0-086

Presents accounting and management planning applications—from financial statements to master budgets; from pricing models to investment strategies.

EXECUTIVE PLANNING WITH BASIC

by X. T. Bui 192 pp., 19 illustr., Ref. 0-083

An important collection of business management decision models in BASIC, including Inventory Management (EOQ), Critical Path Analysis and PERT, Financial Ratio Analysis, Portfolio Management, and much more.

BASIC FOR BUSINESS

by Douglas Hergert 250 pp., 15 illustr., Ref. 0-080
A logically organized, no-nonsense introduction to BASIC programming for business applications. Includes many fully-explained accounting programs, and shows you how to write them.

FIFTY BASIC EXERCISES

by J. P. Lamoitier 236 pp., 90 illustr., Ref. 0-056
Teaches BASIC by actual practice, using graduated exercises drawn from everyday applications. All programs written in Microsoft BASIC.

BASIC EXERCISES FOR THE APPLE

by J. P. Lamoitier 230 pp., 90 illustr., Ref. 0-084
This book is an Apple version of *Fifty BASIC Exercises.*

BASIC EXERCISES FOR THE IBM PERSONAL COMPUTER

by J. P. Lamoitier 232 pp., 90 illustr., Ref. 0-088
This book is an IBM version of *Fifty BASIC Exercises.*

INSIDE BASIC GAMES

by Richard Mateosian 352 pp., 120 illustr., Ref. 0-055
Teaches interactive BASIC programming through games. Games are written in Microsoft BASIC and can run on the TRS-80, Apple II and PET/CBM.

THE PASCAL HANDBOOK

by Jacques Tiberghien 492 pp., 270 illustr., Ref. 0-053
A dictionary of the Pascal language, defining every reserved word, operator, procedure and function found in all major versions of Pascal.

INTRODUCTION TO PASCAL (Including UCSD Pascal™)

by Rodnay Zaks 422 pp., 130 illustr. Ref. 0-066
A step-by-step introduction for anyone wanting to learn the Pascal language. Describes UCSD and Standard Pascals. No technical background is assumed.

APPLE® PASCAL GAMES

by Douglas Hergert and Joseph T. Kalash 376 pp., 40 illustr., Ref. 0-074
A collection of the most popular computer games in Pascal, challenging the reader not only to play but to investigate how games are implemented on the computer.

CELESTIAL BASIC: Astronomy on Your Computer

by Eric Burgess 312 pp., 65 illustr., Ref. 0-087
A collection of BASIC programs that rapidly complete the chores of typical astronomical computations. It's like having a planetarium in your own home! Displays apparent movement of stars, planets and meteor showers.

PASCAL PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan R. Miller 378 pp., 120 illustr., Ref. 0-058
A comprehensive collection of frequently used algorithms for scientific and technical applications, programmed in Pascal. Includes such programs as curve-fitting, integrals and statistical techniques.

BASIC PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan R. Miller 326 pp., 120 illustr., Ref. 0-073

This second book in the "Programs for Scientists and Engineers" series provides a library of problem-solving programs while developing proficiency in BASIC.

FORTRAN PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan R. Miller 320 pp., 120 illustr., Ref. 0-082

Third in the "Programs for Scientists and Engineers" series. Specific scientific and engineering application programs written in FORTRAN.

PROGRAMMING THE 6809

by Rodnay Zaks and William Labiak 520 pp., 150 illustr., Ref. 0-078

This book explains how to program the 6809 in assembly language. No prior programming knowledge required.

PROGRAMMING THE 6502

by Rodnay Zaks 388 pp., 160 illustr., Ref. 0-046

Assembly language programming for the 6502, from basic concepts to advanced data structures.

6502 APPLICATIONS

by Rodnay Zaks 286 pp., 200 illustr., Ref. 0-015

Real-life application techniques: the input/output book for the 6502.

ADVANCED 6502 PROGRAMMING

by Rodnay Zaks 292 pp., 140 illustr., Ref. 0-089

Third in the 6502 series. Teaches more advanced programming techniques, using games as a framework for learning.

PROGRAMMING THE Z80

by Rodnay Zaks 626 pp., 200 illustr., Ref. 0-069

A complete course in programming the Z80 microprocessor and a thorough introduction to assembly language.

PROGRAMMING THE Z8000

by Richard Mateosian 300 pp., 124 illustr., Ref. 0-032

How to program the Z8000 16-bit microprocessor. Includes a description of the architecture and function of the Z8000 and its family of support chips.

THE CP/M® HANDBOOK (with MP/M™)

by Rodnay Zaks 324 pp., 100 illustr., Ref. 0-048

An indispensable reference and guide to CP/M—the most widely-used operating system for small computers.

MASTERING CP/M®

by Alan R. Miller 320 pp., Ref. 0-068

For advanced CP/M users or systems programmers who want maximum use of the CP/M operating system . . . takes up where our *CP/M Handbook* leaves off.

INTRODUCTION TO THE UCSD p-SYSTEM™

by Charles W. Grant and Jon Butah 250 pp., 10 illustr., Ref. 0-061

A simple, clear introduction to the UCSD Pascal Operating System; for beginners through experienced programmers.

A MICROPROGRAMMED APL IMPLEMENTATION

by Rodnay Zaks 350 pp., Ref. 0-005

An expert-level text presenting the complete conceptual analysis and design of an APL interpreter, and actual listing of the microcode.

THE APPLE CONNECTION

by James W. Coffron 228 pp., 120 illustr., Ref. 0-085

Teaches elementary interfacing and BASIC programming of the Apple for connection to external devices and household appliances.

MICROPROCESSOR INTERFACING TECHNIQUES

by Rodnay Zaks and Austin Lesea 458 pp., 400 illustr., Ref. 0-029

Complete hardware and software interconnect techniques, including D to A conversion, peripherals, standard buses and troubleshooting.

SELF STUDY COURSES

Recorded live at seminars given by recognized professionals in the microprocessor field.

INTRODUCTORY SHORT COURSES:

Each includes two cassettes plus special coordinated workbook (2½ hours).

S10—INTRODUCTION TO PERSONAL AND BUSINESS COMPUTING

A comprehensive introduction to small computer systems for those planning to use or buy one, including peripherals and pitfalls.

S1—INTRODUCTION TO MICROPROCESSORS

How microprocessors work, including basic concepts, applications, advantages and disadvantages.

S2—PROGRAMMING MICROPROCESSORS

The companion to S1. How to program any standard microprocessor, and how it operates internally. Requires a basic understanding of microprocessors.

S3—DESIGNING A MICROPROCESSOR SYSTEM

Learn how to interconnect a complete system, wire by wire. Techniques discussed are applicable to all standard microprocessors.

INTRODUCTORY COMPREHENSIVE COURSES:

Each includes a 300-500 page seminar book and seven or eight C90 cassettes.

SB3—MICROPROCESSORS

This seminar teaches all aspects of microprocessors: from the operation of an MPU to the complete interconnect of a system. The basic hardware course (12 hours).

SB2—MICROPROCESSOR PROGRAMMING

The basic software course: step by step through all the important aspects of microcomputer programming (10 hours).

ADVANCED COURSES:

Each includes a 300-500 page workbook and three or four C90 cassettes.

SB3—SEVERE ENVIRONMENT/MILITARY MICROPROCESSOR SYSTEMS

Complete discussion of constraints, techniques and systems for severe environmental applications, including Hughes, Raytheon, Actron and other militarized systems (6 hours).

SB5—BIT-SLICE

Learn how to build a complete system with bit slices. Also examines innovative applications of bit slice techniques (6 hours).

SB6—INDUSTRIAL MICROPROCESSOR SYSTEMS

Seminar examines actual industrial hardware and software techniques, components, programs and cost (4½ hours).

SB7—MICROPROCESSOR INTERFACING

Explains how to assemble, interface and interconnect a system (6 hours).

SOFTWARE

BAS 65™ CROSS-ASSEMBLER IN BASIC

8″ diskette, Ref. BAS 65

A complete assembler for the 6502, written in standard Microsoft BASIC under CP/M®.

8080 SIMULATORS

Turns any 6502 into an 8080. Two versions are available for APPLE II.

APPLE II cassette, Ref. S6580-APL(T)
APPLE II diskette, Ref. S6580-APL(D)

FOR A COMPLETE CATALOG OF OUR PUBLICATIONS

U.S.A.
2344 Sixth Street
Berkeley,
California 94710
Tel: (415) 848-8233
Telex: 336311

SYBEX-EUROPE
4 Place Félix-Eboué
75583 Paris Cedex 12
France
Tel: 1/347-30-20
Telex: 211801

SYBEX-VERLAG
Heyestr. 22
4000 Düsseldorf 12
West Germany
Tel: (0211) 287066
Telex: 08 588 163

SYBEX