Apple IIGS Assembly Language Programming

Apple IIGS Assembly Language Programming

Leo J. Scanlon

BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

Trademarks

Apple, Apple II, II+, //e, //c, IIGS, Applesoft, Disk II, ProDOS, QuickDraw, and SANE are registered trademarks, trademarks, or copyrighted by Apple Computer, Inc.

Macintosh is a trademark licensed to Apple Computer, Inc.

ORCA/M is a trademark of The Byte Works, Inc.

IBM is a trademark of International Business Machines Corporation.

Figures 6-3, 11-3, 12-1, 12-2, 12-3, 12-4, 12-6,
are used with the permission of Apple Computer, Inc.

Apple IIGS Assembly Language Programming
A Bantam Book / August 1987

ISBN 0-553-34395-5

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA
B 0 9 8 7 6 5 4 3 1

Acknowledgments

While my name is the only one on the cover of this book, many people helped along the way, and I want to thank them publicly. Top honors belong to Bill Gladstone of Waterside Productions, Steve Guty of Bantam Books, and Martha Steffen of Apple Computer for their encouragement and unflagging enthusiasm for this project.

Special thanks also to Jim Merritt, Manager of Apple's Developer Technical Support department, who reviewed the manuscript for technical accuracy and overall usability. If you find any errors in the book, blame me, not Jim. (And please write to me about them, in care of Bantam Books.) Pete McDonald, Guillermo Ortiz and the rest of Jim's staff deserve credit, too, for patiently explaining some of the more subtle details of the IIGS.

A few other folks also provided help — without realizing it! I refer especially to Eagle Berns and the others at Apple who developed sample programs to aid IIGS developers. Their listings showed the "right" way to get things done and saved me untold hours (or days or weeks) of programming time.

Danny Goodman's excellent book *The Apple IIGS Toolbox Revealed* was also a valuable resource. It gave me a good starting point for digging deeper into Apple's voluminous documentation.

Finally, I thank Pat, who has always been there to listen.

Contents

Preface

Why Assembly Language?

Many people write all of their computer programs in one of the so-called high-level languages, particularly BASIC. BASIC is easy to learn, easy to use, and fast enough for most computing tasks. That being the case, why would anyone want to use any other language? One reason is that BASIC, like human languages, is not well-suited to everything. Some tasks are much easier in other languages. Imagine, for example, trying to describe fine cooking without some French words, or symphonies without some Italian terms. Similarly, special computing tasks like graphics, music, or word processing are often easier in other languages.

Furthermore, BASIC is quite slow. The term slow may surprise the beginner, since most programs seem to run instantaneously. However, BASIC tends to fall short in the following situations:

1. When large amounts of data are involved. Notice how slow BASIC is when a program must, for example, sort a long list of names and addresses or accounts. Similarly, BASIC is quite slow when a program must search through a 50-page report or keep inventory records on thousands of items.

2. When graphics are involved. If a program is drawing a picture on the screen, it must work quickly or the delay is intolerable. If objects in the picture are supposed to move, the program must be fast enough to make the motion look natural. This is particularly

difficult when the picture contains many objects (such as spaceships, base stations, and alien invaders), all of which are moving in different directions.

3. When a lot of decisions or "thinking" is required. This is often necessary in complex games like checkers or chess. The program has to try many possibilities and decide on a reasonable move. Obviously, the more possibilities there are and the more analysis required, the longer it will take the computer to move.

Why is BASIC slow? In the first place, the computer actually translates each BASIC statement into one or more simple internal commands (so-called machine language). It does this every time it runs the BASIC program. Thus, much of the computer's time is spent decoding the program, not running it.

There are versions of BASIC called *compilers* that perform the translation once and then save the translated version. However, unless the compiler is efficient at "optimizing" programs, even a compiled BASIC program may be slow. A BASIC program is really like an automobile with an automatic transmission; no amount of coaxing can get you the performance or fuel economy that a skillful driver can achieve with a manual transmission. The human being is simply a more flexible, more skillful, and smarter operator than the automatic transmission or the BASIC interpreter or compiler.

Assembly language is like a manual transmission. It gives the programmer greater control over the computer at the cost of more work, more detail, and less convenience. Like an automatic transmission, BASIC is good enough for most programmers. But for those who must get optimum performance from their computers, assembly language is essential. You will find that most complex games, graphics programs, and large business programs are written at least partially in assembly language.

Even if assembly language *is* your likely choice, you may be wondering if you have enough background to learn assembly language programming. You *do* if you have done some programming of *any* kind. If you know BASIC, Pascal, C, or some other so-called "high-level" language, that's even better.

The Contents of This Book

For the benefit of former users of high-level languages, this book has two starting points. If you have never programmed in assembly language,

read Chapter 0, which gives a "crash course" in the *binary* and *hexadecimal* numbering systems. Otherwise, if you already know what these terms mean and understand how to use them, proceed directly to Chapter 1.

Chapter 1 describes the 65816 microprocessor — the "brain" of the Apple IIGS — and the major hardware components inside the IIGS. It also provides a general introduction to assembly language programming and gives an overview of programming the Apple IIGS using its *Toolbox*.

Chapter 2 discusses assemblers in general and then describes the *Apple IIGS Programmer's Workshop*, a software development package offered by Apple Computer. Chapter 2 also presents a simple program and tells how to enter it into the computer, assemble it, and execute (run) it. It also describes the Workshop's *Debugger*, a utility that helps you track down errors in your programs.

Chapter 3 describes the 65816 *addressing modes* you can use to access the data on which your program is to operate.

Chapter 4 discusses the 65816's instruction set, the assembly language commands you can use to communicate with the IIGS. This book treats the instructions in functional groups, rather than alphabetically. That is, I have grouped add with subtract, load with store (the assembly language equivalents of BASIC's Peek and Poke, respectively), and so on. Through this approach, you not only get to *understand* what the instructions do, but you also appreciate how they fit together.

Chapter 5 covers *macros*. A macro is a miniprogram that you can insert in a main program simply by mentioning its name. Macros can make assembly language programs nearly as easy to develop as BASIC programs, and the Programmer's Workshop disk contains *hundreds* of them.

Chapter 6 surveys the Apple IIGS *Toolbox* and the *tool sets* within it. There are tool sets that produce graphics, sound, windows, menus, and virtually any other feature you might want to include in a program. Chapter 6 also includes a *program model* that contains the "boilerplate" programs needed to communicate with the IIGS.

Chapter 7 shows you how to display graphics objects on the screen using the built-in *QuickDraw II* tool set. With QuickDraw, you can easily produce predefined shapes such as rectangles and circles, or objects of your own design.

Chapter 8 tells how to deal with "events," such as the user pressing a key or the mouse button.

Chapter 9 discusses on-screen *windows* and the "controls" the user can employ to operate on them. Chapter 10 covers *menus* that appear on the screen to let the user select a course of action.

The final two chapters are also concerned with user interactions. Chapter 11 provides a brief introduction to buttons, check boxes, and other controls provided by the Control Manager. Chapter 12 shows how to use those controls along with other on-screen items to conduct a conversation or *dialog* with the user.

The book provides four appendixes for your convenience. Appendix A has tables that help you convert hexadecimal numbers to decimal, and vice versa. Appendix B shows the text characters you can display on the screen. Appendix C summarizes the 65816's instruction set in alphabetical order, and shows the legal form for each instruction, how long it takes to execute, how many bytes it occupies in memory, and which status flag it affects. Finally, Appendix D tells what's required to use the tool sets; that is, how they interact and how much working space in memory they need.

For Further Reference

The Apple IIGS is a powerful computer that can do virtually any programming task you want, and I have attempted to describe its most important programming features in the order people generally use them. However, as you gain experience, you will probably want to know more about this fascinating machine. The full details are available in a comprehensive set of technical manuals. They are:

Technical Introduction to the Apple IIGS
Programmer's Introduction to the Apple IIGS
Apple IIGS Hardware Reference
Apple IIGS Firmware Reference
Apple IIGS Toolbox Reference (Volumes 1 and 2)
Apple IIGS Programmer's Workshop
Apple IIGS Programmer's Workshop Assembler Reference
Apple IIGS ProDOS 16 Reference
Apple IIGS Human Interface Guidelines

These manuals are available from your Apple dealer or from the Apple Programmer's and Developer's Association (APDA): 290 SW 43rd Street, Renton, WA 98055.

At the very least, you should have the *Technical Introduction*, the *Programmer's Introduction* and both volumes of the *Toolbox Reference*, to help you proceed from where this book leaves off.

I also recommend two other books in Bantam's "Apple IIGS Library" series. The first, *The Apple IIGS Book* by Jeanne DuPrau and Molly Tyson, is an excellent general-purpose reference on all aspects of the IIGS. Written by two Apple insiders, the book also contains the complete history of the IIGS, including some interesting (and often humorous) anecdotes about the people who designed it. For example, the authors reveal that "GS" stands for . . . are you ready for this? . . . Gumby Software(!), after The Man of Clay, the official mascot of the IIGS design team.

Danny Goodman's *The Apple IIGS Toolbox Revealed* is another worthwhile addition to your library. Danny's plain-English treatment of the Toolbox and the tool sets within it is first-rate. I refer to it often and I'm sure you will, too.

CHAPTER 0

About Computer Numbering

Unless you're visiting from another planet (and if so, welcome!), you have spent your entire life counting things using decimal numbers. Mathematicians call decimal the *base 10* numbering system because it has ten digits, 0 through 9.

Humans are comfortable counting in decimal (probably because we have ten fingers and ten toes), but computers are not. Instead, they count with the base 2, or *binary*, numbering system. The binary system has only two digits, 0 and 1. Hence, to communicate with a computer at its own level — as you do when you program in assembly language — you must be familiar with binary numbering. Besides binary, assembly language programmers also use the base 16, or *hexadecimal*, numbering system, so you must be familiar with it as well. The hexadecimal system has 16 digits, 0 through 9 and A through F.

This chapter is a "crash course" in computer numbering systems, for readers who have had no previous exposure to them. That's why I numbered it Chapter *0*. If you already understand binary and hexadecimal numbering, feel free to skip directly to Chapter 1.

Binary Numbering

A computer gets all program instructions and data from its memory. Memory consists of integrated circuits (or "chips") that contain thousands of electrical components. Like light switches and the power switch on your computer, these components have only two possible settings: "on" and "off." Still, with only these two settings, combinations of memory components can represent numbers of any size.

The on and off settings of a memory component correspond to the 1 and 0 digits of the base 2 or binary numbering system. The switch-like components of memory are called "bits," short for *bi*nary digi*ts*. By convention, a bit that is on has the value 1, while a bit that is off has the value 0. This appears woefully limiting until you consider that a decimal digit (no, it's not called a "det") can only range from 0 to 9. Just as one can combine decimal digits to form numbers larger than 9, one can also combine binary digits to form numbers larger than 1.

As you know, to represent a decimal number larger than 9, you must attach an additional "tens position" digit; to represent a number larger than 99, you must attach a "hundreds position" digit, and so on. Each decimal digit you add "weighs" 10 times as much as the digit to its immediate right.

For example, 324 can be represented as

$$(3 \times 100) + (2 \times 10) + (4 \times 1)$$

or as

$$(3 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$$

Thus, reading right to left, each decimal digit is a power of 10 greater than the preceding digit.

The binary numbering system works the same way, except *each binary digit is a power of 2 greater than the preceding digit.* That is, the rightmost bit has a weight of 2^0 (decimal 1), the second bit has a weight of 2^1 (decimal 2), the third bit has a weight of 2^2 (decimal 4), and so on. For example, the binary value 1001 has a value of decimal 9 because:

$$1001_2 = (1 \times 2^3) + (1 \times 2^0) = (1 \times 8) + (1 \times 1) = 9^{10}$$

In short, to find the value of any given bit position, double the weight of the preceding bit position. Thus, the weights of the first 8 bits are 1, 2, 4, 8, 16, 32, 64, and 128, as shown in Figure 0-1.

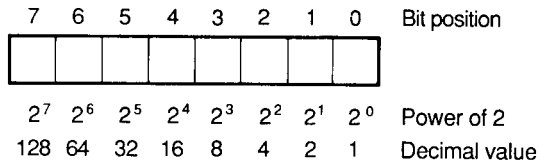


Figure 0-1

Converting Decimal Values to Binary

To convert a decimal value to binary, you make a series of simple subtractions, where each subtraction produces the value of a single binary digit (bit). To begin, subtract the largest possible power of 2 from the decimal value and enter a 1 in the corresponding bit position. Then subtract the next largest possible power of 2 from the result and enter a 1 in *that* bit position. Continue making subtractions until the result is zero. Enter a 0 in any bit position whose weight cannot be subtracted from the current decimal value.

For example, to convert decimal 52 to binary:

```
 52
-32    bit position 5 = 1
 20
-16    bit position 4 = 1
  4
 -4    bit position 2 = 1
  0
```

Entering a 0 in the other bit positions (3, 1, and 0) yields the final binary result of 110100.

To verify that decimal 52 is indeed binary 110100, add the decimal weights of the "1" positions:

```
 32    (bit 5)
 16    (bit 4)
+ 4    (bit 2)
 52
```

Bytes

The Apple II series, Commodore 64, Tandy/Radio Shack TRS 80, and many other popular microcomputers are designed around *8-bit* microprocessors. Eight-bit microprocessors are so named because they transfer data eight bits at a time. To transfer more than eight bits, they must perform additional operations.

In computer terminology, an 8-bit unit of information is called a *byte*. With eight bits, a byte can represent decimal values from 0 (binary 00000000) to 255 (binary 11111111).

Because bytes are a convenient unit of data, microcomputers are usually described in terms of the number of bytes (rather than bits) their memories can hold. Manufacturers typically organize memory in blocks of 1,024 bytes. This particular value reflects the binary orientation of computers in that it represents 2^{10} bytes.

The value 1,024 has a standard abbreviation: the letter *K*. Hence a computer that has a "512K memory" contains 512 × 1024 (or 524,288) bytes.

Adding Binary Numbers

You can add binary numbers the same way you add decimal numbers: by "carrying" any excess from one column to the next. For example, adding the decimal digits 7 and 9 produces 6 in the "ones" column and an excess 1 that you must carry into the "tens" column. Similarly, adding the binary digits 1 and 1 produces a 0 in the "ones" column and an excess 1 that you must carry to the "twos" column.

Adding multibit binary numbers can be somewhat more complex, because you must keep track of multiple carries. For example, this operation involves two carries:

```
 1011
+ 11
------
 1110
```

Adding the rightmost column (1 + 1) produces a result of 0 and a carry of 1 into the second column. With the carry, adding the second column (1 + 1 + 1) produces a result of 1 and a carry of 1 into the third column. Table 0-1 summarizes the general rules for binary addition.

Table 0-1

Inputs			Results	
Operand #1	Operand#2	Carry	Sum	Carry
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	0	1	0	1
1	0	1	0	1
1	1	1	1	1

Signed Numbers

Until now, I have concentrated on unsigned binary numbers, in which each bit has a weight that reflects its position. However, sometimes you want to operate on positive or negative values; that is, on *signed* numbers.

When a byte contains a signed number, only the seven low-order bits (0 through 6) represent data; the high-order bit (7) specifies the sign of the number. *The sign bit is 0 if the number is positive or zero, and 1 if it is negative.* Figure 0-2 shows the arrangement of unsigned and signed bytes.

A byte that holds a signed number can represent positive values between 0 (binary 00000000) and +127 (01111111) and negative values between −1 (11111111) and −128 (10000000). Clearly, the way negative numbers are represented needs an explanation.

Note that −1 in binary is 11111111. Wouldn't it be simpler to make it just 10000001 (that is, 1 with a minus sign bit)? It might be simpler, but it would also produce wrong answers. For example, consider what would happen if you were to add +1 and −1. The answer should, of course, be 0, but instead you would get:

```
00000001    = +1
10000001    = -1
--------    ----
10000010    = +2 (!)
```

Thus, what we need is some way to represent −1 so that adding +1 and −1 produces 0. Indeed, that's how mathematicians arrived at 11111111 to represent −1: it produced the right answer. Now the preceding addition becomes:

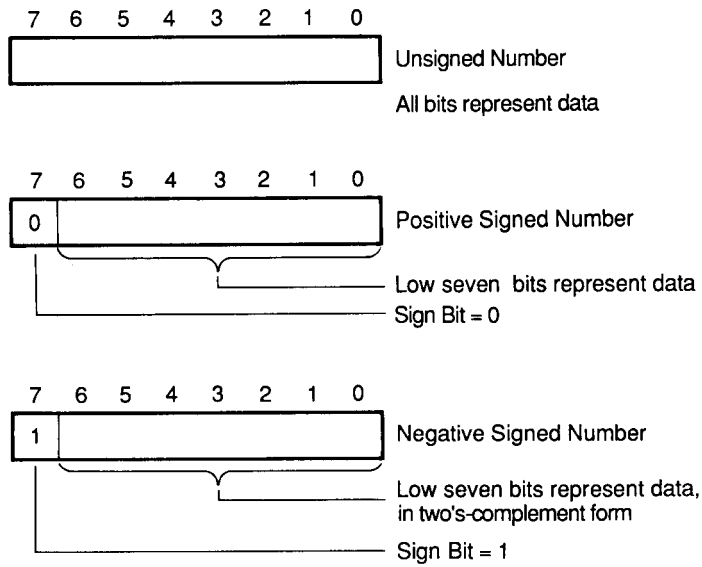


Figure 0-2

```
  00000001  = +1
  11111111  = -1
-------------------
1 00000000  = -0
```

The extra 1 bit at the front is a carry that's been left over from the addition. Since we are operating on 8-bit numbers, we simply ignore this ninth bit.

Two's-Complement

Like −1, all negative signed numbers are represented in a special form that makes additions and subtractions produce the right answers. This is called the *two's-complement* form.

To find the binary representation of a negative number (that is, to find its two's-complement), simply take the positive form of the number and reverse each bit — change each 1 to a 0 and each 0 to a 1 — then add 1 to the result. The following example shows how to calculate the binary representation of −32 in two's-complement form:

```
00100000    +32

11011111    Reverse every bit
+       1   Add 1
--------
11100000    +32 in two's complement form
```

Of course, the two's-complement convention makes negative numbers difficult to decipher. Fortunately, you can use the procedure I just gave to find the positive form of a (two's-complemented) negative number. For example, to find what value 11010000 has, proceed as follows:

```
00101111    Reverse every bit
+       1   Add 1
--------
00110000 =  16 + 32 = +48
```

You'll be happy to hear that you don't normally have to conduct this kind of exercise too often, because the assembler lets you enter numbers in decimal form (signed or unsigned) and does all the converting automatically. However, sometimes you may want to interpret a negative number that is stored in a register or memory, so you should know how to make these conversions manually.

Hexadecimal Numbering

Although computers process only binary values, working with strings of nothing but ones and zeroes can be maddening for humans. It's also easy to induce errors with them, because it's extremely easy to mistype something like 10110101. A single misplaced 1 or 0 can ruin all your calculations.

Years ago, programmers found that they were generally operating on *groups* of bits rather than individual bits. The first microprocessors were 4-bit devices (they processed data four bits at a time), so the reasonable alternative to binary was a system that numbered bits in groups of four.

As you now know, four bits can represent the binary values 0000 through 1111 — decimal 0 through 15 — a total of 16 possible combinations. If a numbering system is to represent 16 combinations, it must have 16 different digits. That is, it must be a *base 16* numbering system.

If base 2 numbers are called "binary" and base 10 numbers are called "decimal," what term is appropriate for the base 16 system? Well, whoever

Table 0-2

Hexadecimal Digit	Binary Value	Decimal Value	Hexadecimal Digit	Binary Value	Decimal Value
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

named the base 16 system combined the Greek word *hex* (for six) with the Latin word *decem* (for ten) to form the word *hexadecimal*. Hence the base 16 system is called the hexadecimal numbering system.

Of the 16 digits in the hexadecimal numbering system, the first ten are labeled 0 through 9 (as in the decimal system), while the last six are labeled A through F (decimal 10 through 15). Table 0-2 lists the binary and decimal values for each hexadecimal digit.

Like binary and decimal digits, each hexadecimal digit has a weight that is some multiple of its base. Since the hexadecimal system is based on 16, each digit weighs 16 times more than the digit to its immediate right. That is, the rightmost digit has a weight of 16^0, the next has a weight of 16^1, and so on. For example, the hexadecimal value 3AF has the decimal value 943 because

$$(3 \times 16^2) + (A \times 16^1) + (F \times 16^0)$$
$$(3 \times 256) + (10 \times 16) + (15 \times 1) = 943$$

While BASIC and other high-level languages usually display numbers in decimal form, the utilities that programmers use to develop assembly language programs generally display numbers in hexadecimal form. This includes memory addresses, instruction codes, and data. Therefore, to get maximum benefit from your programming, try to "think hexadecimal." This is difficult at first, but it will become easier as you gain experience. To help you along, Appendix A provides a table for converting between decimal and hexadecimal.

CHAPTER 1

Introduction to Assembly Language

What is Assembly Language?

Like BASIC, assembly language is a set of commands that tell the computer what to do. However, the commands in the assembly language instruction set refer to computer components directly. It's like the difference between telling someone to walk down to the corner and telling them precisely how to move their muscles and maneuver past obstacles. Obviously, a simple command is sufficient most of the time; only athletes and mountain climbers need the more detailed instructions.

Assembly language programs give the computer detailed commands, such as "load 32 into the X register," "copy the contents of the A register into the Y register," and "store the number in the Y register into memory location 300." As you see, BASIC and assembly language differ in how you instruct the computer. With BASIC, you speak in *generalities*; with assembly language, you speak in *specifics*.

Although assembly language programs take more time and effort to write than BASIC programs, they also run much faster. The level of detail

is the key here. The idea is the same as an athlete who runs faster or jumps further by watching every step of what he or she does. Precise form is essential to achieving maximum performance.

Because assembly language requires you to make direct use of the computer's internal components, you must understand the features and capabilities of the integrated circuit (or "chip") that contains these components, the computer's *microprocessor*. The microprocessor inside the Apple IIGS is a *Western Design Center 65C816*. (The "C" stands for CMOS, short for Complimentary Metal-Oxide Semiconductor. CMOS is the process used to manufacture the chip.) In this book, I'll call it the 65816 — or sometimes just 816 — simply because it's easier to read.

Overview of the 65816 Microprocessor

The 65816 is a 16-bit version of the 8-bit 6502, the microprocessor inside the Apple II series and many other personal computers. However, I don't mean to imply that size is the only difference between the two microprocessors. That would be like saying a Cadillac is an 8-cylinder version of a Volkswagen. The 65816 has quite a few powerful features that the 6502 does not have, and I will note them when necessary.

Data and Address Buses

The 65816 always transfers information to or from memory and I/O devices 8 bits at a time, on its 8-line *data bus*. (Normally, a processor that has an 8-bit data bus would be classified as an 8-bit device. The 65816 is called a 16-bit microprocessor because all its internal circuitry is 16 bits wide.)

Note that I say "memory *and* I/O devices". Like the 6502, the 65816 does not distinguish between memory and I/O — it treats all external devices the same. Hence, it has no special input or output instructions; it has only "load" and "store" instructions, and you use them to transfer data to any external device, regardless of whether it is memory or a peripheral. Like each memory location, each peripheral device has a unique address; the address determines where the 816 sends the data or obtains it.

The 65816 transmits addresses to external devices over 24 lines that are collectively called the *address bus*. Being 24 bits wide, the address bus allows the 816 to access up to 16,777,216 bytes, or *16 megabytes* (abbreviated 16M) — the same range as an IBM System/370! Contrast this with

the 6502, which can address only 64K bytes directly. (Old-timers may recall when 64K was considered a lot of memory!)

Memory Organization

Because the 65816 must be able to work with so much more memory than the 6502 (16M bytes versus 64K bytes), accessing the 65816's memory is slightly more complex.

In the 6502, memory is divided into 256-byte *pages*. Since the 6502 can address up to 64K directly, there are 256 pages in all. Page 0 occupies addresses 0 to 255, page 1 occupies addresses 256 to 511, and so on. (The 6502 treats the lowest page in memory — page 0 or the *zero page* — differently from other pages. The 65816 has an equivalent, special purpose page, called the *direct page*, in low memory. I'll discuss it later.)

If you have a recent Apple //e, you may be thinking, "Scanlon doesn't know what he's talking about. My //e has a 6502, yet it has a 128K memory." Ah, but reread the preceding paragraph, where I state ". . . the 6502 can address up to 64K bytes *directly*". Apple's engineers gave the //e the ability to access 128K bytes by installing circuits that switch between two 64K *banks* of memory. Bank 0 is the lower 64K locations, while bank 1 is the upper 64K. Only one bank can be active at any given time, so my statement about the 6502 addressing only 64K still stands.

The 65816 uses the same page-and-bank scheme as the 6502, except it has bank-switching circuitry built in. Specifically, it has two bank selection registers, where one selects the bank that contains program instructions and the other selects the bank that contains data. With its 16-megabyte addressing range, the 816 can address up to 256 banks. (It's no coincidence that both pages and banks involve the number 256. As I mentioned in Chapter 0, the value 256 and other multiples of 2 reflect the binary nature of computers.) Just because the computer *can* address this much memory doesn't mean that it's all actually available. For example, the standard 256K model of the Apple IIGS has only four banks of read/write memory.

Software Features

The 65816 provides 91 types of assembly language instructions, 35 more than the 6502. It also provides 24 addressing modes, 11 more than the 6502. (An addressing mode is a technique the microprocessor uses to obtain a number it is to add, subtract, or whatever.)

How fast does an instruction execute? That depends on which instruction you're referring to, which addressing mode you're using, and how fast

the microprocessor is operating. Like electronic watches, microprocessors are regulated by quartz crystals. The crystal emits pulses at a fixed rate, which determines how fast the microprocessor operates. In the Apple IIGS, the crystal emits either 1.0 million or 2.5 million pulses per second, depending on which operating speed is active in the Control Panel.

Computerists don't refer to pulses per second, however, but to *cycles per second* or, more often, *Hertz*. Pulses per second, cycles per second, and Hertz all mean the same thing, but in this book I use the term Hertz or its abbreviation, Hz. Hence, one can say that the clock in the IIGS normally runs at 2.5 million hertz, or 2.5 MHz (short for megahertz). At 2.5 MHz, the 816's clock "ticks" every 400 *nanoseconds*, where a nanosecond (abbreviated ns) is one-billionth of a second, or 10^{-9} seconds. At 1.0 MHz, the clock "ticks" every 1,000 nanoseconds; 1,000 nanoseconds, which is one millionth of a second or 10^{-6} seconds, is called one *microsecond* (μs).

The fastest instructions — for example, those that increment a register — execute in 2 cycles, or 800 ns at 2.5 MHz. The slowest instructions — for example, ones that call a subroutine — take 8 cycles, or 3,200 ns at 2.5 MHz. (Note that even the "slow" instructions execute in the remarkable time of 0.0000032 seconds!) Most instructions require between 3 and 6 cycles to execute.

Interrupts

The microprocessor in a computer (an Apple or any other) does not simply run programs. As the chief regulator of the system, it gets involved in one way or another with everything that happens. For instance, when someone presses a key at the keyboard, the processor must find out which key was pressed and do whatever is appropriate for that particular key. Similarly, when a disk drive is transferring data to or from the computer's memory, the processor is responsible for carrying out the instructions that make that transfer happen. As I just said, the microprocessor has a role in *everything* the computer does.

Well, how does a microprocessor get involved with a peripheral device? It doesn't have ESP, so it can't *know* which device needs attention. Then again, it shouldn't sit there asking or "polling" each peripheral whether it needs something. If the processor polled peripherals all day, it wouldn't be able to do anything else — it would have no time to run programs. This would be like having a telephone with no bell. You would have to pick up the receiver every so often just to find out whether anyone was on the line!

In fact, microprocessors and peripherals communicate in a very efficient

fashion. The microprocessor continues to run a program (say, DOS or BASIC or a word processor) until the keyboard, display unit, or some other peripheral says, "Excuse me, micro, but I need your help in getting something done. Would you please stop what you're doing long enough to help?" Of course, peripherals don't actually talk to the microprocessor; they send a special "help me" signal called an *interrupt request*.

Interrupts operate differently between various microprocessors, but in the 65816 (and 6502 and 65C02), here is what happens. When a peripheral device needs servicing, it activates an interrupt request line that is connected to the 816. There is only one interrupt request line; all devices in the system share it.

So, in essence, here is a device ringing the processor's doorbell. Sometimes the processor is doing something so important that it can't stop to respond to the request. In that case, the 816 simply says, "Let the bell ring. I'll answer when I can."

Otherwise, if the processor is doing a job that can wait until later, it makes some notes about the job (so it knows where to resume later), then reads a new program address from a special *interrupt vector* in memory and starts executing that program. The program determines which device is making the request, then transfers to the routine for that particular device. When the 816 finishes servicing the peripheral, it uses the notes it made earlier to get back to the original job.

Operating Modes

The 65816 is actually two microprocessors in one, insofar as it can operate in two different modes, called *native* and *emulation*. In the normal operating mode, native, the 816 can process data either 8 or 16 bits at a time. In emulation mode, it acts like a 6502, and can only process 8-bit data. When you switch the Apple IIGS on, the 65816 starts in native mode; it stays in this mode until a program explicitly switches it to emulation mode.

Internal Registers

Figure 1-1 shows the 65816's internal registers and, because it can operate in emulation mode, the 6502's registers. Note that the 6502's data registers (A, X, and Y) and status register (P) are 8 bits long, whereas its address registers (S and PC) are 16 bits long. This is typical for an 8-bit microprocessor; it has an 8-bit data bus and a 16-bit address bus. (Incidentally, the

shaded "01" that represents the high-order byte of the stack pointer is a hexadecimal value. It indicates that the 6502's stack is always located in page 1 of memory. More about stacks later.)

As you can see, the 65816 microprocessor provides all the 6502 data registers (as the low bytes of its own data registers) and address registers (as the low 16-bit words of its own address registers), but extends data registers to 16 bits and address registers to 24 bits.

Lest you be mistaken, I must emphasize that the 65816 contains only *one* set of registers: those shown in the right-hand column of Figure 1-1. When emulating a 6502, however, the 816 uses those registers as a 6502 would. It interprets any program reference to the A or X register as meaning AL or XL, limits the stack pointer to 16 bits (and puts hex 01 in the upper

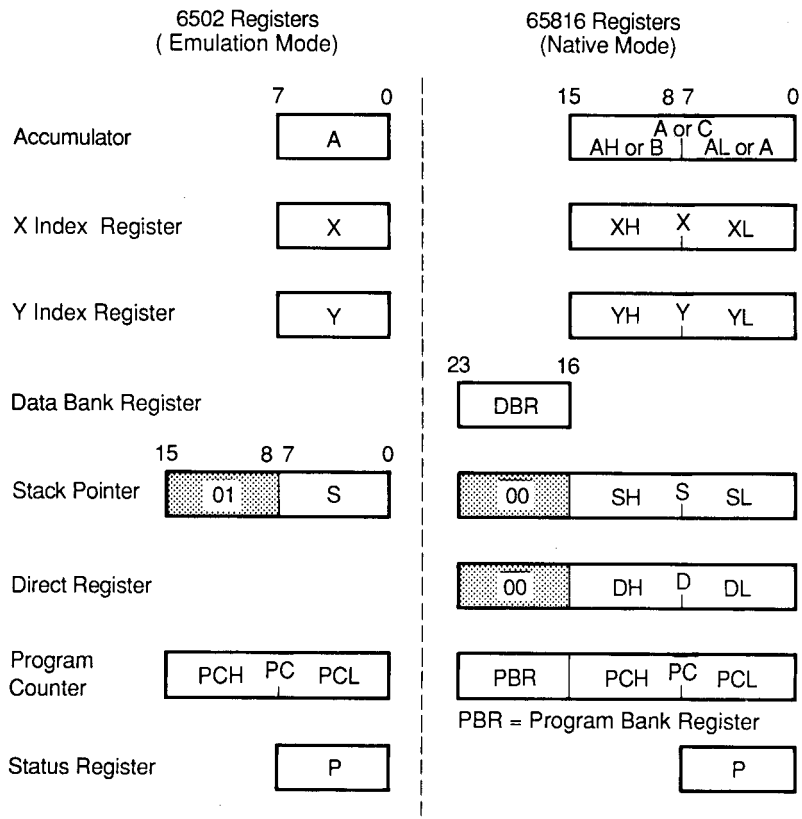


Figure 1-1

byte), and so on. In emulation mode, then, the 816 acts like an adult speaking to a child. It uses only the vocabulary the child can understand, which is something less than the full vocabulary the adult knows.

The following sections describe only the 65816's regular registers, the ones it uses when operating in native mode. To learn about the 6502 registers, which are active in emulation mode, buy one of the many 6502 assembly language books on the market.

General-Purpose Registers

You can treat these registers as either three 16-bit registers or six 8-bit registers, depending on whether you're operating on 16-bit words or 8-bit bytes. The 16-bit registers are named A, X, and Y. The 8-bit registers within them are named AH, AL, XH, XL, YH, and YL. In each case, "H" and "L" indicate the high-order and low-order bytes of the 16-bit registers.

A, the accumulator, is the main register for performing arithmetic and logical operations; it holds one operand and the result of most of these operations. *X* and *Y* serve primarily as index registers to calculate a memory address. However, the 816 also provides instructions that increment and decrement X and Y, which make them useful as counters.

Data Bank Register

The 8-bit data bank register (DBR) provides the bank number in addressing modes that otherwise generate only the lower 16 bits of an address. There, DBR acts as an 8-bit "extension" of the X or Y register that effectively makes X or Y 24 bits long. When you switch the Apple IIGS on, the 65816 initializes DBR to zero, thereby selecting bank 0.

Stack Pointer

As with most other microprocessors, the 65816's memory contains a special data structure called a *stack* that serves as a temporary holding area for data and addresses. When you execute, or "call," a subroutine, the 816 uses the stack to hold the return address — a place marker that eventually brings the processor back to its original spot in the program. You can also use the stack to preserve the contents of registers that a subroutine alters.

The 65816 enters data onto the stack and extracts data from it the same way you (or, with any luck, your children) stack dishes in your kitchen; that is, the last item to be placed on the stack is also the first item to be removed from it. Computerists usually refer to this type of stack as "last in, first out"

(or LIFO). As data items are *pushed* onto the stack, they are stored in memory at ever-lower addresses; the stack "builds" toward address 0.

The stack pointer (S) is a 16-bit register that points to the next available location on the stack; it is the stack's maitre d'. The 65816 decrements S by 1 as each new byte is pushed onto the stack and increments it by 1 whenever a byte is *pulled* off the stack.

As the shaded "00" in Figure 1-1 indicates, the stack is always in bank 0. Note, however, that while the 6502's stack must be in page 1, the 65816's stack can be in any page. In general, though, you shouldn't worry about where the stack is located, what the stack pointer contains, or what's on the stack (unless you specifically stored something there). The computer's software handles all stack operations, so unless you do something awful — such as push two items on the stack but extract only one — you should have no problems.

Direct Register

Recall under "Memory Organization" that I mentioned a special-purpose *direct page* in bank 0. The 24-bit direct register (D) determines which block of 256 bytes is to be used as the direct page. When you switch the Apple IIGS on, the 65816 initializes the direct register to 0.

Program Counter

The 65816 executes (runs) programs by obtaining instructions from memory, one at a time. The *program counter (PC)* determines which memory location the 816 will access next. The 816 increments the PC automatically after each memory access so that it points to the next consecutive location. Because the program counter is 16 bits wide, it can access any location in the active 64K bank. A separate program bank register (PBR) specifies the bank.

Like the stack pointer, the program counter is a register that the microprocessor uses to keep track of addresses (in this case, the addresses of instructions), and you shouldn't be concerned with it unless you're troubleshooting or "debugging" a program.

Program Bank Register

The 8-bit program bank register (PBR) provides the bank number for the instruction that the 816 is to execute next. Thus, it "extends" the program counter to 24 bits. When you switch the Apple IIGS on, the 65816 initializes PBR to zero, thereby selecting bank 0.

Processor Status Register

The 8-bit processor status (P) register contains one-bit indicators. Some of these bits are *status flags* that reflect the result of a previous instruction (generally, the preceding instruction), others are *mode control bits* that determine how the 65816 operates. The P register can take either of two forms, depending on whether the 816 is operating in emulation mode (Figure 1-2) or native mode (Figure 1-3).

In *native mode*, the P register bits do the following:

Bit 0 — The *carry (C) flag* is 1 if an addition produces a carry or a subtraction produces a borrow; otherwise, C is 0. C also holds

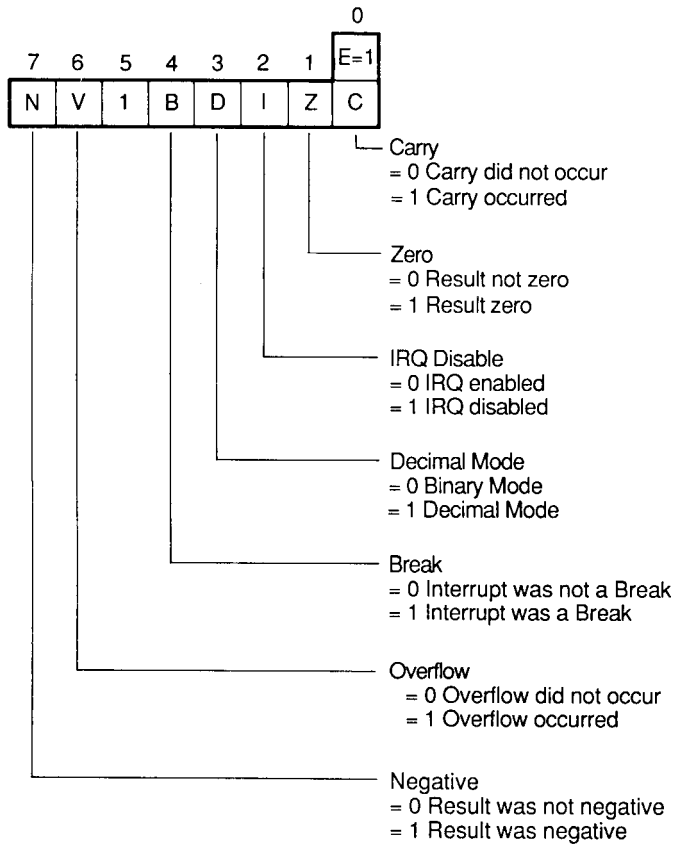


Figure 1-2

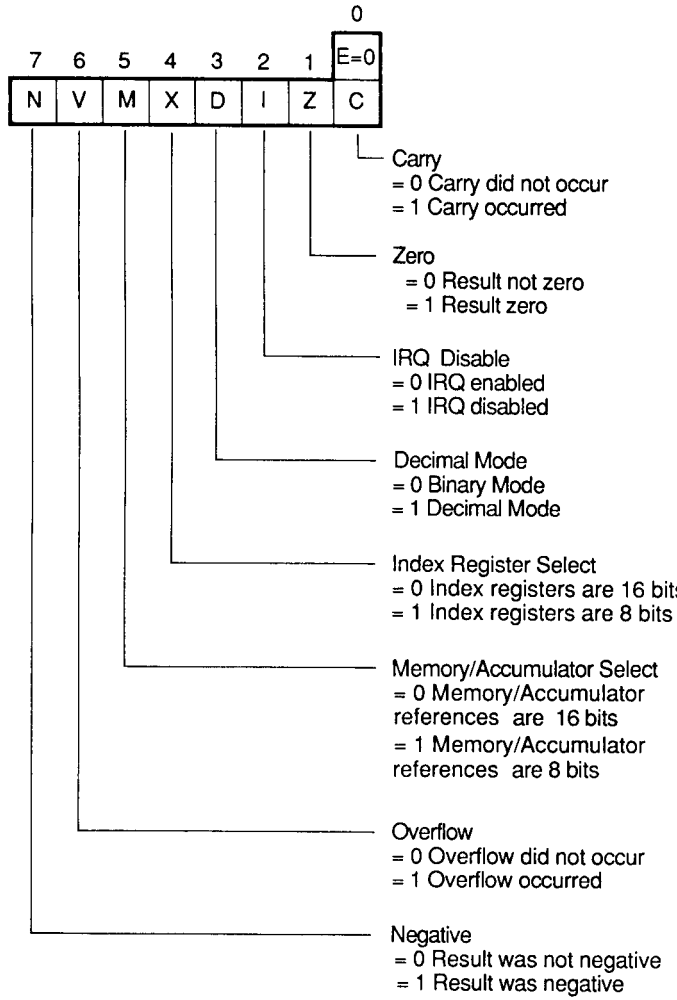


Figure 1-3

the value of a bit that has been shifted or rotated out of a register or memory location, and reflects the result of a compare operation.

Bit 1 — The *zero (Z) flag* is 1 if the result of an operation is zero; a nonzero result clears Z to 0.

Bit 2 — The *IRQ disable (I) flag* allows the 65816 to recognize interrupts from external devices in the system. Setting I to 1 makes the 816 ignore interrupt requests until I becomes 0.

Bit 3 — The *decimal mode (D) flag* controls whether the 65816 is operating on binary numbers (0) or decimal numbers (1). In the decimal mode, the 816 treats arithmetic operands as binary-coded decimal (BCD) digits "packed" two per byte.

Bit 4 — The *index register select bit (X)* specifies whether X and Y are to be treated as 16-bit registers (0) or 8-bit registers (1). Switching the X bit from 0 to 1, or vice versa, leaves XL and YL unchanged, but clears XH and YH to 0.

Bit 5 — The *memory/accumulator select bit (M)* specifies whether operands in memory or the accumulator are 16-bit values (0) or 8-bit values (1). Switching M from 0 to 1, or vice versa, has no effect on AH (B) or AL (A).

Bit 6 — The *overflow (V) flag* is an error indicator for operations on signed numbers. V is 1 if adding two like-signed numbers or subtracting two opposite-signed numbers produces a result that the operand can't hold; otherwise, V is 0.

Bit 7 — The *negative (N) flag* is meaningful only for operations on signed numbers. N is 1 if an arithmetic, logical, shift, or rotate operation produces a negative result; otherwise, N is 0. In other words, N reflects the most-significant bit of the result, regardless of whether it is 8 or 16 bits long.

Unless you are an experienced programmer or a daring one, I suggest you always leave the M and X bits set to 0. Setting either or both to 1 sometimes produces unexpected results.

The processor status bits are the same in *emulation mode*, except bit 4 is a break command flag and bit 5 is unused; it's always 1. The break command (B) flag indicates whether an interrupt request to the processor was generated by a BRK instruction (1) or by an externally-generated interrupt (0).

Inside the Apple IIGS

Although you must understand the internal architecture of the 65816 to be an effective assembly language programmer, you needn't know much about the

hardware within the Apple IIGS. Still, a general understanding of the hardware can't hurt. In fact, you will probably feel more comfortable programming the IIGS if you know what goes on inside it.

Figure 1-4 shows an engineering-style block diagram of the major hardware components in the Apple IIGS. Note that the diagram is divided into a "slow" and "fast" side. On the slow side are components that can only run at 1.0 MHz, the standard Apple II operating speed. The fast side has the components that can run at 2.5 MHz, the speed at which the IIGS operates in native mode.

Fast (2.5-MHz) Components

The "fast" side of the block diagram includes the 65816 microprocessor, a fast processor interface (FPI) chip, a 128K block of read-only memory (ROM), two 64K banks of RAM, and a connector for expanding RAM beyond its standard 256K bytes.

The FPI chip regulates the computer's operating speed. It sends out a 2.5-MHz clock signal when the IIGS is running in native mode and a 1-MHz clock when the IIGS is running in emulation mode. The FPI also synchronizes the processor running at 2.5 MHz with circuits on the "slow" side running at 1 MHz.

The 128K ROM holds all of the computer's built-in programs, or *firmware*. ROM is non-volatile; its contents stay intact even when the power is off. ROM includes, of course, the program that puts the IIGS into some predefined initial state (native mode, registers set to zero, and so on) when you switch it on. But ROM also includes many more "goodies" that I describe later.

The two banks of RAM are, in fact, the banks that regular Apple II programs use. Since Apple IIs are 1-MHz computers, why is their memory on the fast (2.5-MHz) side? The answer to that question is somewhat involved, so I'll postpone discussing it until later.

Slow (1-MHz) Components

As you can see from the block diagram see Figure 1-4, the slow side includes the computer's built-in sound circuitry, the second 128K of standard RAM, plus all the components necessary for the IIGS to communicate with peripheral devices. A major component here is a chip called *Mega II*.

Mega II contains all the circuitry necessary to produce the display modes needed by programs for earlier Apple IIs (low-resolution, high-resolution, and so on). Because this is no easy task, Mega II is virtually packed

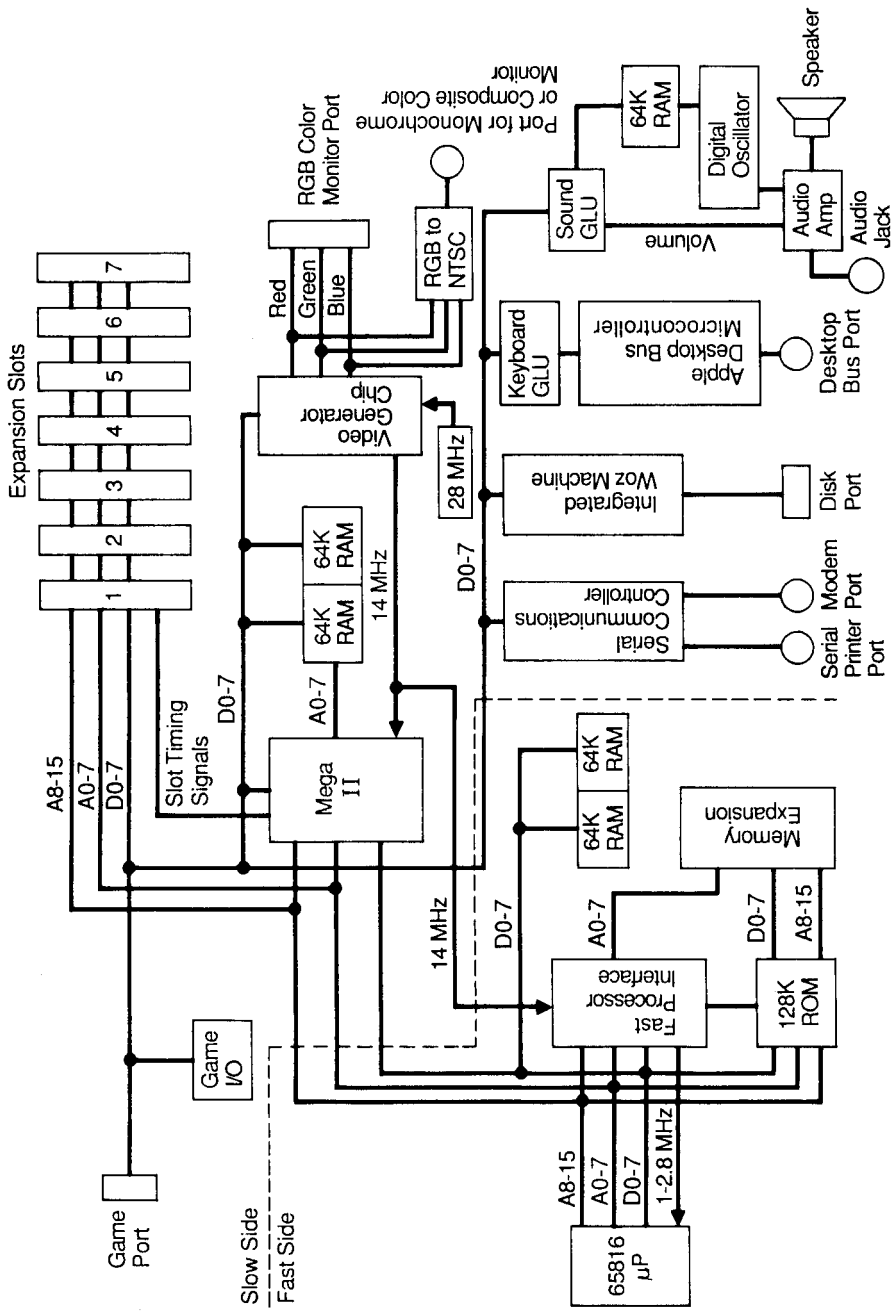

Expansion Slots
1
2
3
4
5
6
7
A8-15
A0-7
D0-7
Slot Timing Signals
Game Port
Game I/O
Slow Side
Fast Side
Mega II
64K RAM
64K RAM
14 MHz
A0-7
D0-7
Video Generator Chip
Red
Green
Blue
RGB Color Monitor Port
RGB to NTSC
Port for Monochrome or Composite Color Monitor
28 MHz
D0-7
Keyboard GLU
Apple Desktop Bus Microcontroller
Desktop Bus Port
Integrated Woz Machine
Disk Port
Serial Communications Controller
Serial Modem Port
Serial Printer Port
64K RAM
64K RAM
Memory Expansion
14 MHz
D0-7
A0-7
D0-7
A8-15
Fast Processor Interface
128K ROM
A8-15
A0-7
D0-7
1-2.8 MHz
65816 µP
Sound GLU
64K RAM
Digital Oscillator
Speaker
Volume
Audio Amp
Audio Jack

Figure 1-4

with components. In all, it contains the equivalent of about 3000 transistors and a 2K ROM that holds display characters. Mega II is, in fact, close to being an Apple II on a chip! Besides its display functions, Mega II also interacts with external devices plugged into the computer's expansion slots and rear-panel connectors (or ports).

The *video generator chip (VGC)* does the actual interfacing with attached display monitors. Its output lines are converted to red, blue, and green signals for RGB monitors and are combined into a single signal for monochrome and composite video monitors, such as home television sets.

The *serial communications controller (SCC)*, *integrated Woz machine (IWM)*, and *Apple desktop bus microcontroller (ADBM)* service the serial, disk, and desktop bus ports on the computer's rear panel. Both the disk and desktop port can handle a number of peripherals that are "chained" together. For example, the IIGS keyboard connects directly to the desktop bus port and the mouse is chained to it.

System Memory

As I mentioned earlier, the 65816 can address 16 megabytes of memory. Figure 1-5 shows how the Apple IIGS uses this space.

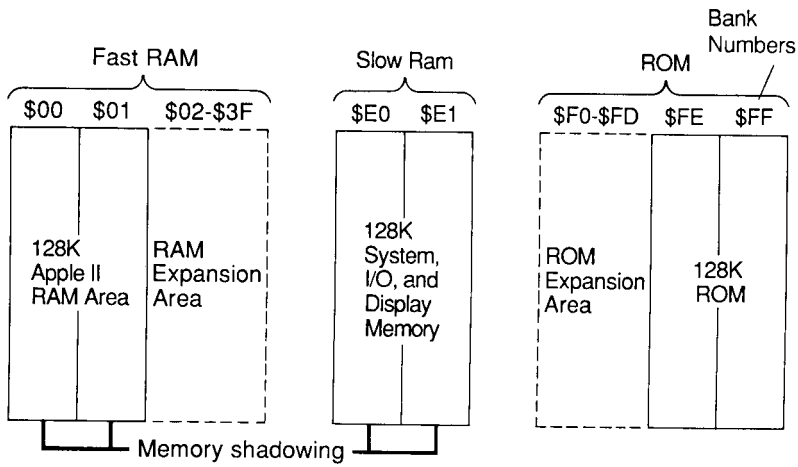


Figure 1-5

Read/Write Memory

The base model IIGS comes with 256K bytes of read/write memory, or *RAM*, consisting of four 64K banks: $00, $01, $E0, and $E1. (RAM, short for random access memory, is really a misnomer, since disk storage is also random access. It might just as well be called RWM, but it's too late to change the industry now.) Apple II programs use the first 128K. The system uses some of the second 128K for working storage, I/O (remember, the 65816 treats external devices just like ordinary memory), and the display. I'll discuss that second 128K, banks $E0 and $E1, shortly. Programs written for the Apple IIGS — that is, those that run in the 65816's native mode — can use about 176K of the 256K bytes available.

Banks $02 through $3F provide memory space for a RAM expansion card that can be plugged into the IIGS motherboard. In all, the 62 banks numbered $02 through $3F provide addressing space for an additional 3.875 additional megabytes of RAM, bringing the total RAM capacity to 4.125 megabytes. All of the expansion RAM is available to application programs; the system doesn't use it.

Memory Shadowing

Note that in Figure 1-5, banks $E0 and $E1 are labeled *slow RAM*, while the rest is *fast RAM*. Slow RAM operates at the Apple II's normal speed, 1 MHz, while fast RAM operates at the IIGS's normal speed, 2.5 MHz.

You're probably wondering how Apple II programs can run in banks $00 and $01, which is fast, 2.5 MHz RAM. The designers took care of this problem by using a technique called *memory shadowing*. When a program writes something into bank $00 or $01, the IIGS writes the same thing into the corresponding location in bank $E0 or $E1. Because the memory in $E0 and $E1 is synchronized to the video hardware, the instruction must execute at the slow speed. However, that only applies to write operations; the 65816 always *reads* from the affected areas of banks $00 and $01 at the faster speed.

To summarize, the trick is: only fast memory is read; both fast and slow memory are written. A write must be constrained to the 1-MHz speed because the operation isn't done until the slower component has been accessed.

Banks $E0 and $E1

Figure 1-6 shows a memory map of banks $E0 and $E1. Starting at the top of the figure, the reserved 1K is a working storage area for various IIGS

system software. There is another working storage area at address $0C00. Starting at address $0400 are the buffers that hold screen data for text and low-resolution graphics. The regular and double high-resolution buffers start at $2000. Note that when super hi-res is active, the buffer extends from location $2000 to location $9FFF; that's 32K total. The last 8K of the free space is the memory that the Memory Manager manages. Finally, you encounter the I/O, banks 0 and 1, and language-cards areas.

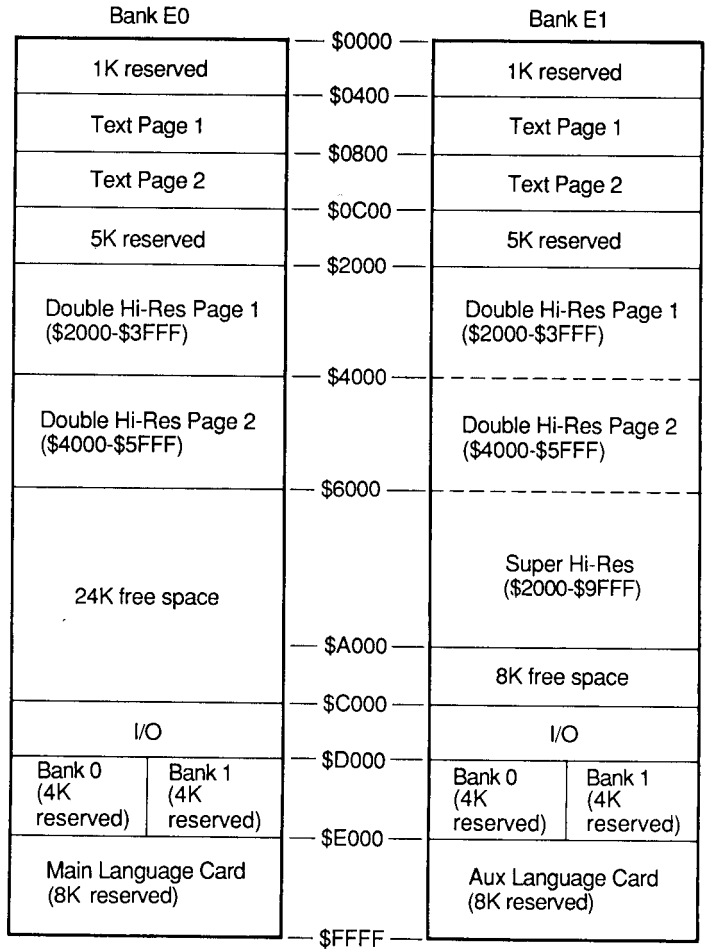


Figure 1-6

ROM

The ROM in banks $FE and $FF contains the following firmware:

1. Monitor
2. Control Panel menu
3. Mouse support
4. Appletalk
5. Support for modem and printer serial ports
6. Disk support
7. Front Desk Bus (FDB) support
8. Diagnostics
9. Desk Accessory Manager support
10. Tools

Programming the Apple IIGS

Like all microcomputers, the Apple IIGS has an internal operating system stored in read-only memory. This internal operating system, called the *Monitor*, holds the program that starts the computer when you switch the power on. It also holds programs that communicate with the keyboard, display information on the screen, and transfer data to the printer and whatever else is attached to your computer. The Monitor does these jobs by running miniprograms called *subroutines* contained in the ROM. There are many useful subroutines within the Monitor and, if you wish, you can use them to do common programming chores such as displaying a message on the screen or reading what the operator types at the keyboard.

In other Apple IIs, the Monitor subroutines are the only built-in aids available to the programmer. While the subroutines provide shortcuts for simple chores, they fall short in several areas. For example, the Monitor has no subroutines that multiply or divide numbers (functions that the 6502 and 65816 microprocessor instruction sets don't do), nor does it have subroutines that produce high-resolution graphics. The Monitor doesn't include these kinds of operations because they aren't needed to make the computer operate — and, after all, that's the Monitor's sole purpose of existence.

Thus, if you want to multiply, divide, or display graphics on an Apple II, II+, //e, or //c, you must either find some software that will do the job for you or (gasp!) create the software yourself.

Fortunately, as an Apple IIGS programmer, you need not rely on the Monitor subroutines to help you do things. In fact, you may *never* use the Monitor, because the IIGS includes a large *Toolbox* that includes powerful programming aids called *tool sets*. Some of these tool sets are built into ROM, while others are fetched from the Apple IIGS System Disk. Each tool set is responsible for a particular variety of tasks. One set displays graphics, another handles sound generation, still another handles communication with the keyboard and mouse, and so on.

Just as a plumber's toolbox contains tools for working with pipes, and an electrician's toolbox has tools for working with wires, sockets, and fuses, each Apple IIGS tool set contains *tools* that relate to its specific area of responsibility. For example, the graphics tool set (called QuickDraw II) has tools that draw shapes on the screen, fill in areas with a specified color, and so on.

Quite simply, there is a tool for virtually every programming task one might want to do. In light of that, your assembly language programs will look quite different from programs people wrote for other Apple IIs. Instead of dozens (or hundreds or thousands) of assembly language instructions and calls to the Monitor, your programs will consist primarily of tool "calls," with just a sprinkling of assembly language instructions here and there.

Because the tools do most of the work, your programs will be much shorter, "cleaner," and easier to understand than equivalent programs written for other Apple IIs. They'll also take you less time to develop.

I'll discuss using tools later in the book, but in the next few chapters, I'll introduce the mechanics of developing assembly language programs and describe the 65816's instruction set. Read this material casually to get an overall understanding of it, but *don't* try to memorize every detail. In later chapters, where I describe actual programs, you will see how the microprocessor instructions and instructions to the assembler (or *directives*) fit in. In writing your own programs, you can always come back to these earlier chapters to look up specific details of an instruction or directive.

CHAPTER 2

Using an Assembler

Assembly language offers the best of two worlds. That is, it lets you write programs at the level the microprocessor understands, which helps ensure that they will be both fast and efficient. Yet assembly language doesn't force you to memorize a lot of numeric codes. Instead, you enter instructions as English-like abbreviations, then run an *assembler* program to convert the abbreviations to their numeric equivalents.

The program comprised of abbreviations is called the *source program*, while the numeric, microprocessor-compatible form of it is the *object program*. Thus, the assembler's job is to convert source programs you can understand into object programs the microprocessor can understand. It does much the same thing a compiler does in a high-level language such as BASIC, C, or Pascal.

There are several assembler software packages available for the Apple IIGS, but this book describes just one: Apple Computer's *Apple IIGS Programmer's Workshop* (or *APW* for short). The assembler within the *APW* is actually an enhanced version of the popular *ORCA/M* assembler from The Byte Works, Inc., so *ORCA* programmers will feel quite comfortable with it. In any case, the features the *APW* provides should be similar to those of any other IIGS-compatible assembler you might have, because they all deal with the same computer.

The *Programmer's Workshop* manual provides complete details, so I won't attempt to describe everything. Instead, I'll concentrate on the features you will probably use most often, and provide some summary tables for quick reference.

Developing an Assembly Language Program

Although assembly language programs look quite different from BASIC, C, or Pascal programs, one follows the same procedures to develop them. However, in assembly language the *mechanics* are more involved. There are seven steps in developing an assembly language program:

1. Define the task and design the program. This often requires drawing a *flowchart*, a road map of how the program should operate.
2. Type the program instructions into the computer using an *editor*, then save the program on disk.
3. Assemble the program using the *assembler*. This produces a disk file called an "object module." If the assembler reports errors, correct them with the editor and reassemble the program.
4. Convert the object module to an executable "shell load file" using the *linker*. Shell load files can be run from the *Programmer's Workshop* disk.
5. If the program is to be run from the System Disk's Program Launcher, redefine the shell load file as a "system load file."
6. Execute (run) the program.
7. Check the results. If they differ from what you expected, you must find the errors or "bugs." The *debugger* is handy for doing this.

If your program is short and simple, you can perform these steps quickly. But longer and more complex programs require more time on each step, especially defining the program. I discuss an efficient approach for developing programs under "Top-Down Program Design" at the end of this section.

Editor

Step 2 above refers to an editor. This is a program that lets you enter and prepare your program. You can use any word processor or editor program that can produce pure ASCII text — regular characters without any special control codes or formatting codes. One such program is *MouseWrite*, from Roger Wagner Publishing, Inc. If you don't have one of these programs, you can use the editor program that comes on the *Programmer's Workshop* disk (see the editor later in the chapter).

Assembler

The computer cannot execute the program you prepare with the editor. You must use the assembler to convert it into an *object* module the computer can understand.

Linker

The linker converts object modules into *shell load files*. A load file contains the numeric (object) form of the program in a form the system loader can load into memory. A shell load file can be run under the *Programmer's Workshop* by entering its name. The linker also does another important job: it combines two or more object modules — a main module and one or more subroutine modules — to form a load file.

Note that *you must run the linker for every program you write*, even those that have only one object module. If a program has only one module, the linker simply puts it in loadable form. If a program has two or more object modules, the linker combines them and makes the *result* loadable. Figure 2-1 illustrates the stages involved in editing, assembling, and linking a program.

If you want to be able to run the program from the System Disk's Program Launcher, there is one more job to do: you must redefine the shell load file as a *system load file*. This takes only a simple command that changes the file's "type."

Debugger

Unfortunately, most programs don't run exactly as expected the first time; they usually contain errors. The easiest kind of error to correct is a syntax error, such as accidentally typing *%E4* when you intended to type *$E4*, or mistyping an *LDA* instruction as *LDQ*. Syntax errors are easy to spot because the assembler identifies them and issues an appropriate error message when it assembles the program.

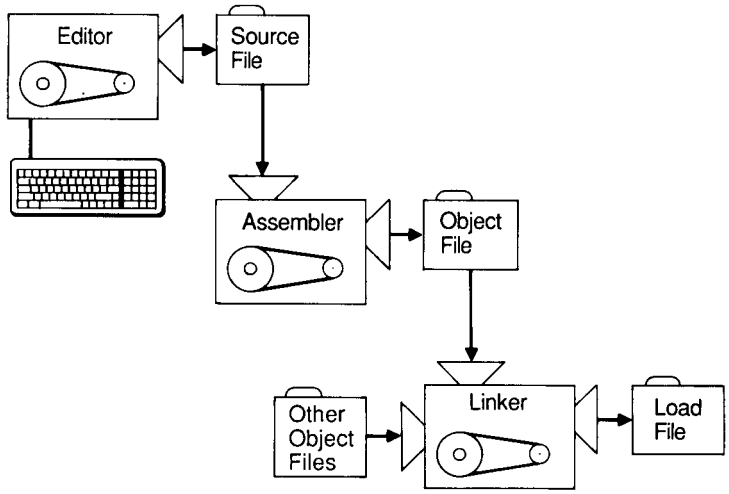


Figure 2-1

Errors that show up when you run the program are also sometimes easy to correct. For example, if the prompt "Please enter your name" shows up as "Please enter your *nabe*", you know at once that there's a typo in your program's message string. Simply correct it with the editor and reassemble the program.

If syntax and typographical errors are the only kinds you encounter, you're either a genius or living a charmed life. Most people occasionally get errors that are difficult to locate. The more minor errors just produce an incorrect result, such as a 26 where you expected a 140. Other errors "crash" the program, sending the computer off to a never-never land in which the screen goes blank. Here, your only recourse is to switch the power off.

When these kinds of things happen, programmers usually list the program and search for obvious errors. More often than not, their search is fruitless. Where does one go from here? Well, generally the best approach is to run the program one instruction at a time, and check each step of the way whether everything is going as expected. The utility that lets you "step" through programs is called the *debugger*.

Among other things, the debugger also lets you display and change values and stop the program at a specific point, to see how things are

proceeding. The debugger thus provides tools that help you to identify and correct errors in programs.

Top-Down Program Design

When creating a program on a computer, you are usually inclined to charge into it just as you would with pencil and paper, entering the first instruction, then the second, third, and so on, to the end. This "brute force" approach may work for programs that are short or simple, but generally it leads to errors and produces programs that are difficult to understand and even more difficult to update later. Thanks to the editing capabilities of word processors and editors, there is an easier, more reliable, and more efficient way to develop programs. It is called *top-down design*.

Top-down design simply means starting with general issues and progressing methodically to specifics. Outlines, tree diagrams, and flowcharts are all valid mechanisms for achieving the ends of top-down design.

One way to approach top-down design is to start with a plain-English outline of the program, then filling in the details gradually. The outline should be a series of lines that tell what steps you want the program to perform. For example, to develop a program that performs one of several tasks based on a choice the user makes from a menu, your outline might look like this:

```
; Display a menu of selections.
; Ask the user to choose.
; Read the user's selection.
; Do what the user has asked.
```

The semicolons here indicate that these lines represent comments rather than instructions. They do the same thing as REMs in BASIC.

From here, you can use the editor to insert instructions between the comment lines. Because each line defines a simple task, you can complete them individually and test each one before proceeding to the next one. That is, begin by inserting the first group of instructions (the ones that display a menu, in my example), then save the program on disk and assemble, link, and execute it. Executing this partially completed program tells you whether it is working correctly so far. If it isn't, debug it and try again. When the first part is working, proceed to the second part, then the third, and so on.

This may seem like a slow way to develop a program, but it has several advantages:

1. It forces you to plan the program in an orderly fashion.
2. The comment lines provide a certain amount of top-level documentation to the finished product.
3. It helps ensure that each step is working correctly before you proceed. This way, you know that any error was probably caused by something you did in the current group of instructions, not in a previous group.

Source Statements

Now that I have discussed the mechanics of developing programs, it's time to take a look at what can go into them. The source program you enter into the computer is a sequence of *statements* that are designed to perform a specific task. A source statement (a line in a program) can be any of five things:

1. A blank line.
2. A comment.
3. An assembly language instruction.
4. An assembler directive.
5. A macro call.

Assembly language instructions are shorthand for the 65816 microprocessor's instruction set. Some manuals refer to them as machine language instructions, because they tell the "machine" (the 65816) what to do. By contrast, assembler directives tell the *assembler* what to do (with the instructions and data you enter).

Macros are labeled groups of instructions. To run the entire group, you simply enter its name and specify any parameters it is designed to accept. You could think of a BASIC statement such as PRINT as a macro, because the interpreter or compiler must replace PRINT with several machine language instructions. (Remember, regardless of which language your program is written in, the microprocessor only understands machine language. Every program file must ultimately translate to machine language.) Macros are discussed in Chapter 5.

Assembly Language Instructions

Each assembly language instruction in a source program can have up to four *fields*, as follows:

```
{Label}    Mnemonic    {Operand}    {;Comment}
```

Of these, only the mnemonic field is always required. The label and comment fields are always optional. The operand field applies only to instructions that require an operand; otherwise, you must omit it. (I show the label, operand, and comment fields in braces to identify them as optional; *don't* type the braces in your programs.)

The label must start in column 1, but you can enter the other fields anywhere on the line, as long as you separate them with at least one space (or tab). An assembly language instruction that uses all four fields is:

```
SetCount    LDX    #4    ;Initialize count
```

Label Field

The label field assigns a name to an assembly language instruction, letting other instructions in the program refer to this instruction. Thus, labels in assembly language programs serve the same purpose as line numbers in BASIC programs.

The label must start in column 1 and cannot contain blanks. It must begin with an alphabetic character, *A* through *Z* — or *a* through *z*, the assembler doesn't distinguish between lowercase and uppercase. The remaining characters may consist of:

- The letters *A* through *Z* (or *a* through *z*).
- The numeric digits 0 through 9.
- The underscore character (_).
- The tilde character (~).

The assembler can process instruction labels of up to 255 characters long (!). However, because most printers produce 80-column lines, the practical limit is somewhat less than 80 characters, to provide for mnemonic and operand fields.

Because the assembler lets you enter various combinations of characters, most labels you can think of are acceptable. However, I recommend the following guidelines for selecting labels:

- Make the name as short as possible, while still being reasonable.
- Make the name easy to type without errors. The usual typing problems are several identical letters in a row (such as HHHH) and similar-looking characters (such as the letter O and the number 0, letter I and number 1, letter S and number 5, and letter B and number 8). There is no reason to invite typing errors; most of us make enough of them anyway.
- Don't use labels that can be confused with each other. For example, avoid using similar names like DataLoc and DateLoc. There's no sense tempting fate and Murphy's law.

Mnemonic (Opcode) Field

The mnemonic field (the first *m* in mnemonic is silent) contains the three-letter abbreviation for the instruction. For example, LDA is the abbreviation for the Load Accumulator instruction and JMP is the abbreviation for the Jump instruction. The assembler uses an internal table to translate each mnemonic into its numeric equivalent.

Many instructions require you to specify an operand as well as a mnemonic. For example, the JMP instruction must know where to jump. The mnemonic tells the assembler what type(s) of operand, if any, it should expect to find in the operand field.

Operand Field

The operand field tells the 65816 where to find the data it is to operate on. This will be either the data itself, a memory address, or a register name. The operand field is mandatory with some instructions and prohibited with others. (The addressing characteristics for each instruction in the 65816's instruction set are discussed in Chapter 3.)

The assembler differentiates constants in the operand field based on which prefix is used. A hexadecimal or binary number must be preceded by a $ or % symbol, respectively, while a decimal number must be unprefixed. For example, the following instructions are equivalent:

```
LDA    #$E        ;Hexadecimal
LDA    #%1110     ;Binary
LDA    #14        ;Decimal
```

Text characters must be enclosed in single quotes ('). For example, *LDA #'Y'* loads the numeric code for capital letter Y into the accumulator.

Comment Field

Like a REM in BASIC, this optional field lets you describe statements in a program, to make the program easier to understand. You must precede a comment with either a semicolon (;), an asterisk (*), or an exclamation point (!) — most programmers use the semicolon — and separate it from the preceding field by at least one space or tab. The assembler ignores comments, but prints them when you list the program.

Put anything you want in a comment field, but to be useful, make comments describe what is happening in the program, not just restate the instruction. For example,

```
LDA    #0     ;Set result register to 0 to begin
```

is more meaningful than

```
LDA    #0     ;Move 0 into 1
```

You may also put a comment on a line by itself, to describe a block of instructions. To do this, enter a semicolon at the beginning of the line. The assembler recognizes ; as the start of a comment line and ignores whatever follows it.

Assembler Directives

Directives are commands to the assembler, rather than to the microprocessor. They can be used to set up subroutines, define symbols, reserve memory for temporary storage, and perform a variety of other housekeeping tasks. Unlike assembly language instructions, however, most directives generate no object code.

Directive statements can have up to four fields. They are:

```
{Label} Directive {Operand} {;Comment}
```

As the braces indicate, only the directive itself is always required. A label is mandatory with some directives and optional with the rest. The same applies to an operand. Of course, the comment is always optional. As with an assembly language instruction, you can put a directive anywhere on a line (except in column 1), but you must separate it from the label with at least one space or tab. Table 2-1 divides the commonly used directives into six groups: program control, file control, space allocation, equate, listing, and mode.

Program Control Directives

In the Apple IIGS, assembly language programs are divided into named groups called *code segments*. Every program has at least one code segment: the segment that holds the main program. That code segment may also contain subroutines that the main program uses. If you want to call a subroutine from some other segment, however, you must define it in a separate code segment.

Each code segment must begin with a *START* directive and end with an *END* directive. The START directive must be preceded by a label, which is the name of the segment. For example, a segment named MainSeg that includes a call to a subroutine named Subr1 would have this general form:

```
MainSeg     START           ;Start of main segment
            . .
            . .
            JSR  Subr1      ;Call Subr1
            . .
            . .
            END             ;End of main segment
Subr1       START           ;Start of subroutine segment
            . .
            . .
            END             ;End of subroutine segment
```

Again, this assumes that Subr1 is a subroutine you want to share between segments. If it is used only by MainSeg, you could put it in MainSeg's segment. (In that case, you wouldn't need the second START and END. Subr1

Table 2-1

Function	Format		
Program Control			
Start code segment	*label*	START	
End segment		END	
Define global entry point	*label*	ENTRY	
Start data segment	*label*	DATA	
Use data segment		USING	*label*
Enable/disable long accumulator and memory		LONGA	ON/OFF
Enable/disable long index registers		LONGI	ON/OFF
File Control			
Append source file		APPEND	*pathname*
Copy source file		COPY	*pathname*
Save object file		KEEP	*pathname*
Space Allocation			
Define storage		DS	*byte-count*
Define constant		DC	{repeat-count} *value(s)*
Set origin		ORG	*address*
Equate			
Equate	*name*	EQU	*expression*
Global equate	*name*	GEQU	*expression*
Listing			
List/don't list output		LIST	ON/OFF
List/don't list absolute addresses		ABSADDR	ON/OFF
List/don't list symbol table		SYMBOL	ON/OFF
Send output to printer/to screen		PRINTER	ON/OFF
Start new page		EJECT	
Print page number and header		TITLE	{'header'}
Mode			
Enable/disable 65C02 instructions	65C02		ON/OFF
Enable/disable 65816 instructions	65816		ON/OFF

would simply be the label of the first instruction in the subroutine.) The JSR here stands for Jump to Subroutine; it does the same thing as GOSUB in BASIC.

START labels are termed "global," because instructions in any segment of the program can refer to them. By contrast, labels *within* code segments are "local." Only instructions within that segment can refer to them. (This has a side benefit: it means that you can use the same label in more than one segment.)

However, you can also declare an instruction global, by preceding it with an *ENTRY* directive. The label on the ENTRY directive assigns a global name to the instruction that follows. As its name implies, the ENTRY directive is convenient for providing an alternate entry point to a subroutine. For example, in a subroutine of the following form:

```
Subr1   START
        . .
        . .
AltEnt  ENTRY
        . .
        . .
        END
```

your program could contain a JSR Subr1 instruction to start at the beginning or a JSR AltEnt instruction to start at the ENTRY point (AltEnt).

You can also set up *data segments* to hold constants and variables that are used by your code segments. Every data segment must begin with a *DATA* directive (rather than a START directive) and end with an END directive. To use a variable or constant in a data segment, a code segment must contain a *USING* directive that names the data segment. For example, the statement USING DS1 must precede the first instruction that refers to an item in the data segment named DS1.

The *LONGA* and *LONGI* directives specify whether the accumulator and index registers are 8 bits long (OFF) or 16 bits long (ON). ON is the default to both directives. You can use these directives outside of a code segment as well as within one. LONGA OFF and LONGI OFF are convenient in emulation-mode programs. Without LONGA OFF, for example, the instruction

```
LDA    #4
```

that is, load decimal 4 into the accumulator would make the assembler assemble 4 as a 16-bit number.

File Control Directives

File control directives specify a disk path name as the operand. *APPEND* is used to construct programs that are too large for the editor to handle all at one time. APPEND simply reads a file in from disk and tacks it onto the current program. You can specify any valid ProDOS path name for the APPENDed file. If it is on a different disk, the assembler will tell you to switch disks before continuing.

COPY does the same thing as APPEND, except COPY may appear anywhere within a program, while APPEND may only appear at the end. COPY is convenient for inserting a file of equates (description upcoming) in a program.

KEEP is used to assign a file name to an object module (which must have a different name than its parent, the source module). You can only use KEEP once, at the beginning of the program and preceding the first START directive. For example, in a program called MYPROG.SRC, you could enter *KEEP MYPROG*. To store the object module on a different disk or in a different directory, you must precede MYPROG with a complete path name. The KEEP directive is optional, by the way. You can also name the object module when you assemble the program (details later).

Space Allocation Directives

Many programs use locations in memory to hold variables, and the assembler provides two directives to reserve space for them. *DS* (Declare Storage) reserves a specified number of bytes without assigning a value to them, whereas *DC* (Declare Constant) both reserves bytes and gives them an initial value. The DC directive is commonly used to define integers, addresses, hexadecimal constants, binary constants, and character strings. Table 2-2 shows the formats for these data types.

Table 2-2

Type	Format
Integer	{repeat-count}I{size}'value1,value2...'
Address	{repeat-count}A'address1,address2,...'
Hexadecimal constant	{repeat-count}H'digit(s)'
Binary constant	{repeat-count}B'digit(s)'
Character string	{repeat-count}C'string'

Note that you can precede the type identifier with a *repeat-count* that tells the assembler how many times to repeat the data between the single quote marks. Note also that you can follow the integer specifier, I, with a *size* value. This specifies the size of the integer in bytes; it can range from 1 to 8. If you omit the size, the assembler makes each data item two bytes long. For example, the following directive stores six integers in memory. The first four are two bytes long (because size is omitted), while the last two are one byte long (size = 1).

```
TABLE  DC  2I'4,5',I1'1,2'
```

If you displayed that portion of memory, you would see:

```
04 00 05 00 04 00 05 00 01 02
```

Here are examples of the other types, including the hexadecimal values they store in memory:

Directive	Memory values
DC A'Table1,Table2'	(16-bit addresses of Table1 and Table2)
DC H'01234ABCD'	01 23 4A BC D0
DC H'AAAA BBBB CCCC'	AA AA BB BB CC CC
DC B'1011 0110'	B6
DC C'Are you sure?'	41 72 65 20 79 6F 75 20 73 75 72 65 3F

(See Appendix B for the numeric values of the characters in the string.)

When you run your program, the system loader will store it at any convenient place in memory. However, you can specify the starting location yourself by putting an *ORG* (short for Origin) directive at the beginning of the program. For example, ORG $3000 will locate the program at $3000.

ORG's operand can also include an asterisk (*) to indicate the current location. Thus, ORG * + 2 makes the assembler skip 2 bytes before storing the next instruction. This statement does the same thing as DS 2.

Equate Directives

The equate directives assign the value of an operand expression to a name. Thereafter, you can use the name anywhere you would normally use the expression. The first of these directives, *EQU*, is a "local" equate; its name can only be used in the segment in which it is defined. The other, *GEQU*,

is a "global" equate; its name can be used in any segment in the program. Global equates are normally defined in data segments rather than code segments. Some examples of equates are:

```
TWO       EQU       2
FOUR      EQU       TWO*TWO
K         GEQU      1024
```

The operand for an equate can also contain an instruction label. If it is the label of a direct (zero) page address or a long address (I'll describe address types in the next chapter), it must precede the labeled instruction; if it is the label of an instruction in any other page, it must follow the labeled instruction. Note that you can also use the ENTRY directive to define a global label.

Listing Directives

The assembler automatically displays the names of your program's segments as it assembles them. However, you can also make it list the numeric machine code for each instruction by entering a *LIST ON* directive at the beginning of the program. If you want to list only a portion of the program, you can shut off the listing with *LIST OFF*. These directives can also be applied selectively within a program. To list a subroutine, for example, precede it with LIST ON and follow it with LIST OFF.

Program listings include an address for each instruction, but they are given relative to the start of the segment. (Each segment starts at address 0.) To obtain addresses relative to the start of the program, enter an *ABSADDR ON* directive at the beginning. These so-called absolute addresses do not indicate where in memory the program will actually be loaded, because loading is the system loader's job, not the assembler's. However, you can easily add them to the starting load address (more on this later) to calculate actual or "effective" addresses. Having effective addresses comes in handy during debugging.

The assembler also generates an alphabetical list of all local symbols after each END directive and a list of all global symbols at the end of the program. The *SYMBOL* directive lets you turn symbol table generation ON or OFF. This speeds up the assembly slightly and saves paper.

The *PRINTER* directive determines whether the assembler listing is sent to the printer (PRINTER ON) or the display screen (PRINTER OFF). The default is OFF. The assembler assumes that you have an 80-column printer connected to IIGS.

EJECT makes the printer start a new page. This is handy for dividing a listing into logical groups — say, to put each subroutine on a page of its own or to separate symbol tables from listings.

The final listing directive, *TITLE*, prints the page number and (optionally) a header at the top of each page. You can use as many TITLE directives as you want. For example, you can enter a title for each subroutine, to give your listing a polished, professional look.

Mode Directives

The assembler is designed to work with 6502 and 65C02 programs, as well as those written for the 65816. The 65816 includes all the 6502 and 65C02 instructions and addressing modes, plus some additional ones of its own (details are upcoming in Chapters 3 and 4). The mode directives tell the assembler which microprocessor's instruction set to recognize. The first one, *65C02*, lets you tell the assembler whether to accept 65C02 instructions (65C02 ON) or only 6502 instructions (65C02 OFF). The first form is convenient for writing programs to run on an Apple //c; the second can be used to write Apple II, II+, and //e programs.

The second mode directive, *65816*, lets you tell the assembler whether to accept 65816 instructions and addressing modes (65816 ON) or only those for the 65C02 and 6502 (65816 OFF). The latter form is convenient if you are writing a program for earlier Apple IIs, because with it, you needn't worry about whether you have used a 65816 feature accidentally.

Advanced Directives

The assembler offers a variety of other directives (see Table 2-3), but most are only of use for advanced applications. If you're interested in any of these directives, refer to Apple's *Assembler Reference* manual. There are also some directives related to macros; they'll be discussed in Chapter 5.

Operators

Note: This is primarily a reference section. If you are a beginner, read it casually, then come back later if you need to look up details.

An *operator* is a modifer used in the operand field of an assembly language or directive statement. There are three kinds of operators: arithmetic, logical, and relational. Table 2-4 summarizes them.

Table 2-3

Function	Format		
Program Control Directives			
Define private code segment	*label*	PRIVATE	
Define private data segment	*label*	PRIVDATA	
Memory Designation Directives			
Align to a boundary		ALIGN	*number*
Reserve memory		MEM	*start, end*
Assembler Option Directives			
Generate IEEE format number		IEEE	ON/OFF
Set maximum error level		MERR	*level*
Set the most-significant bit for characters		MSB	ON/OFF
Specify case sensitivity		CASE	ON/OFF
Specify case sensitivity in object module		OBJCASE	ON/OFF
Listing Option Directives			
Expand DC statements		EXPAND	ON/OFF
Set comment column		SETCOM	*column-number*
Show instruction times		INSTIME	ON/OFF
List/don't list errors		ERR	ON/OFF
Miscellaneous Directive			
Rename opcodes (mnemonics)		RENAME	*original, new*

Arithmetic Operators

The arithmetic operators combine numeric operands and produce a numeric result. The most common arithmetic operators are those that add (+), subtract (−), multiply (*), and divide (/). The divide operator returns the quotient produced by a divide operation. For example,

```
Pi_Quot  EQU  31416/10000
```

returns 3.

Finally, the bit shift operator (|) displaces a numeric operand to the left or right, depending on whether the specified bit count value is positive or negative. About the only time you need this capability is when you're setting up "masks" that you will apply to binary patterns in memory. For example, if you set up a mask with the statement

Table 2-4

Operator	Function
Arithmetic	
+	Format: value1 + value2 Adds *value1* and *value2*.
−	Format: value1-value2 Subtracts *value2* from *value1*.
*	Format: value1*value2 Multiplies *value2* by *value1*.
/	Format: value1/value2 Divides *value1* by *value2*, and returns the quotient.
|	Format: value|count Shifts *value* by *count* bit positions. Shifts left if *count* is positive and right if it is negative.
Logical	
.AND.	Format: operand1.AND.operand2 True (1) if both operands are nonzero; false (0) if either operand is zero.
.OR.	Format: operand1.OR.operand2 True (1) if either or both operands are nonzero; false (0) if both operands are zero.
.EOR.	Format: operand1.EOR.operand2 True (1) if either operand, but not both, is nonzero; false (0) if both operands are nonzero.
.NOT.	Format: .NOT.operand True (1) if the operand is zero; false (0) if it is nonzero.
Relational	
=	Format: operand1 = operand2 True (1) if the operands have the same value; false (0) if they have different values.
<>	Format: operand1<>operand2 True (1) if the operands have different values; false (0) if they have the same value.
<=	Format: operand1<=operand2 True (1) if *operand1* is less than or equal to *operand2*; false (0) if *operand1* is greater than *operand2*.
>=	Format: operand1>=operand2 True (1) if *operand1* is greater than or equal to *operand2*; false (0) if *operand1* is less than *operand2*.

Table 2-4 (cont.)

Operator	Function
>	Format: operand1>operand2 True (1) if *operand1* is greater than *operand2*; false (0) if *operand1* is less than or equal to *operand2*.
<	Format: operand1<operand2 True (1) if *operand1* is less than *operand2*; false (0) if *operand1* is greater than or equal to *operand2*.

```
Mask  EQU  %10110010
```

the statement

```
Mask_Left_2  EQU  Mask|2
```

sets up a new constant with the value %11001000. Similarly,

```
Mask_Right_2  EQU  Mask|-2
```

sets up a new constant that has the value %00101100.

Logical Operators

Logical operators are so named because they operate according to the rules of formal logic, as opposed to the rules of mathematics. The rules of logic are formed with a series of true/false "if" statements that lead to a "then" conclusion. A typical example is "If A is true and B is true, then C is true." In fact, this very statement has an assembly language counterpart in the *.AND.* operator.

.AND. tests two operands and returns a 1 (true) if both have a value other than zero (true). If either operand is zero (false), .AND. produces a result of 0 (false).

.OR. is somewhat more liberal. It produces a 1 (true) result if either or both operands are nonzero.

.EOR. is a variation on .OR. that returns a 1 (true) if either operand, but not both, is nonzero. In fact, the "but not both" exclusion is how .EOR. got its name; EOR is short for Exclusive-OR. Table 2-5 summarizes how .AND., .OR., and .EOR. operate.

Table 2-5

		Result		
Operand #1	**Operand #2**	**.AND.**	**.OR.**	**.EOR**
Nonzero	Nonzero	1	1	0
Nonzero	0	0	1	1
0	Nonzero	0	1	1
0	0	0	0	0

The final logical operator, *.NOT.*, tests only one operand. It produces a 1 (true) if the operand is zero, or a 0 (false) if the operand is nonzero. Note that .NOT. differs from the other three logical operators in that it is looking for a zero value rather than a nonzero value.

Relational Operators

Relational operators compare two numeric values or memory addresses and, like the logical operators, produce 1 if the relationship is "true," or 0 if it is "false." For example, if CHOICE is a predefined constant,

```
LDA  #CHOICE<20
```

assembles as either LDA #1 (if CHOICE is less than 20) or as LDA #0 (if CHOICE is equal to or greater than 20).

Because the relational operators can only produce two values (1 or 0), they are rarely used alone. Instead, they are usually combined with other operators to form a decision-making expression. For example, suppose you want the accumulator to contain 10 if CHOICE is less than 20 and to contain 5 otherwise. A statement that performs this task is:

```
LDA  #5+5*(CHOICE<20)
```

Here, if CHOICE is greater than or equal to 20, CHOICE<20 is "false," and the assembler replaces it with 0. Since 5 times 0 is 0, the accumulator will receive 5 when you run the program.

You can even get fancier than that, by building more complex expressions. For example, suppose you want the accumulator to contain 10 if CHOICE is less than 20 but greater than 0 (i.e., CHOICE is between 1 and 19), and to contain 5 otherwise. This statement will do the job:

```
LDA  #5+5*((CHOICE<20).AND.(CHOICE>0))
```

Recall that .AND. also produces a 1 (true) or 0 (false) result, so it is the perfect operator for letting the computer make range decisions like this one.

Entering, Assembling, and Running Programs

Since I haven't yet discussed the details of the 65816's assembly language instruction set (that's in Chapter 4), I can't expect you to write any useful programs at this point. Thus, I will provide a simple program that sounds a tone through the speaker inside the Apple IIGS.

The details of the program are unimportant, however. The main point is that you will learn how to enter a program into the computer, assemble it, produce a load module (i.e., an executable program file), and execute it. This exercise will give you hands-on experience with the basic steps, and should help you proceed with confidence through more complex material in the rest of the book.

Starting the *Apple IIGS Programmer's Workshop*

The *Apple IIGS Programmer's Workshop (APW)* disk is bootable, so you can start with it directly. You may want to do this to create a program that you will run in some later session. However, the *APW* disk does not contain all the tool sets your programs may need. These sets *are* contained on the Apple IIGS System Disk, however, so you should start using the following procedures:

1. Insert the System Disk into your main disk drive and turn the computer on. (If it is already on, insert the System Disk and press Ctrl-OpenApple-Reset.)
2. When the Program Launcher screen appears, replace the System Disk with the *APW* disk and click the mouse on *Disk*. The Program Launcher loads the *APW* disk (volume name /APW/) and lists its contents.
3. Run the highlighted file, APW.SYS16, by clicking on *Open*. This makes the *APW* prompt, #, appear.

You'll learn what to do next in a moment, but first here's the example program that beeps the speaker.

A Simple Speaker-Beeping Program

Every Apple II has a location in memory that controls the computer's internal speaker. Whenever the microprocessor accesses this location, by either reading from it or writing to it, the speaker emits a "click" sound. A single click isn't very exciting, but if you make the speaker click fast enough, it sounds like a continuous tone, or in some frequency ranges, like a musical note. The program I am about to present reads the contents of the speaker location repeatedly to produce a "beep." In other Apple II models, the speaker "occupies" location $C030 of bank 0; in the Apple IIGS, it occupies location $C030 of bank $E0 (and is copied, with shadowing, into $00C030).

Example 2-1 lists the speaker-beeping program. This program consists of two loops. The outer loop, which starts at BeepIt, controls the *duration* of the beep — how long it lasts. The inner loop, which starts at Loop2, controls the beep's *frequency* — the rate at which the speaker gets clicked. The faster the speaker is clicked, the higher pitched the tone will be.

Briefly, here's what the lines in the program do:

- The *keep beep* directive at the beginning tells the assembler what name to give the object file. The linker will also assign this name to the load file it produces.
- The *Beep START* directive marks the beginning of the program's only code segment. (The final *END* directive marks the end of this segment.)

Example 2-1

```
; BEEP beeps the speaker by repeatedly accessing the speaker
; location in bank $E0.

          keep  beep       ;Name object and load files

Beep      START
speaker   equ   $E0C030    ;This is the speaker location

          ldy   #1000      ;Y is outer loop (duration) counter
BeepIt    lda   speaker    ;Beep the speaker
          ldx   #1000      ;X is inner loop (frequency) counter
Loop2     dex              ;Decrement X by 1
          bne   Loop2      ;Loop until X is 0,
          dey              ; then decrement Y by 1
          bne   BeepIt     ;If Y is not 0, go beep the speaker
          rtl              ;Otherwise, if Y is 0, exit
          END
```

- The *speaker* EQU directive simply allows me to use the word "speaker" in instructions where I would otherwise use the potentially cryptic number $E0C030.
- The first instruction in the program, *ldy #1000*, loads decimal 1000 into the Y register. The program will repeatedly decrement Y by 1 until it contains 0. At that point, it will exit back to the calling program; the Programmer's Workshop "shell," in this case. Thus, Y determines the duration of the beep. (The value 1000 is arbitrary. I have used it only because it produces a beep that lasts long enough to be heard easily.)
- The instruction *lda speaker* at BeepIt clicks the speaker once. It loads the contents of the locations $E0C030 and $E0C031 into the accumulator, but the value it has obtained is inconsequential. I simply want to access the speaker location somehow, and this does the job.
- The *ldx #1000* loads decimal 1000 into the X register.
- The *dex* at Loop2 decrements (decreases) the X register by 1.
- The *bne Loop2* instruction (where *bne* stands for Branch if Not Equal to 0) tests what happened when X was decremented. If X contains anything other than 0, the processor transfers to the preceding *dex* at Loop2; if X contains 0, the processor "drops through" to the next instruction. Hence, this two-instruction loop makes the processor wait until *dex* has been executed 1000 times. The wait interval determines the frequency of the tone in that it controls how often the speaker location is accessed. (As before, the value 1000 is arbitrary. I could have used a lower number to produce a higher-pitched tone or vice versa.)
- The instructions *dey* and *bne BeepIt* complete the outer loop, the one that determines how long the beep lasts. With each pass through the loop, Y gets decremented by 1. As long as it contains a nonzero value, the processor "branches" to the instruction at BeepIt, to click the speaker. When Y reaches 0, the processor executes the *rtl* (Return Long) instruction, which makes it exit the program.

Entering the Program

Now it's time for you to enter the example program into the computer. Start the computer using the *Programmer's Workshop* disk, or reboot by pressing

Control-OpenApple-Reset. When the *Workshop* prompt (#) appears, enter the command:

```
asm65816
```

This tells the editor to format the program you are about to enter as an assembler *source code* file, as opposed to a text file or some other kind of file.

Now, start the editor by entering

```
edit beep.src
```

where *beep.src* (for beep.source) is the name of the new program. You're actually telling the editor to load BEEP.SRC from disk, but since that file does not yet exist, the editor starts with a blank screen. It shows a rectangular cursor at the upper left-hand corner and a format line and status line at the bottom.

The *format line* shows a dot for each column position and a caret symbol (^) for each tab position. The editor provides tabs at columns 10, 16, 41, 48, 56, 64, 72, and 80 (the end of the line) automatically. The first three tabs — at columns 10, 16, and 41 — are convenient for entering the mnemonic, operand, and comment fields (respectively) of assembly language instructions.

The *status line* reports the position of the cursor (it is at "Line: 1" and "Col: 1" initially), the editor mode (EDIT, to begin), and the name of the file being edited (BEEP.SRC, in this case).

Now, enter the source program shown in Example 2-1, line by line, using the Tab key to move between fields and Return to start a new line. If you finish without omitting or mistyping anything, press *Control-Q* to leave or "quit" the editor (see "Leaving the Editor" in this chapter). Otherwise, if you made some mistakes along the way (most of us do), simply go back and correct them.

Correcting Typing Errors

The editor provides a variety of useful commands for manipulating text. Table 2-6 summarizes the ones you will probably use most often.

Note the following points about these commands:

1. Pressing Return from anywhere on a line moves the cursor to column 1 of the next line. This differs from most word processors, where pressing Return moves any remaining characters to a new line.

Table 2-6

(**Note:** The abbreviation *OA* represents the Open Apple key.)

Quit Command	
Quit (Exit to the Editor Menu)	OA-Q or Ctrl-Q
Cursor-Moving Commands	
Bottom of Screen/Page Down Moves the cursor to the bottom of the screen, at the current column position. If it is already at the bottom, the screen scrolls down one page (22 lines).	OA-DownArrow or Control-OA-J
Cursor Down	DownArrow or Control-J
Cursor Left	LeftArrow or Control-H
Cursor Right	RightArrow or Control-U
Cursor Up	UpArrow or Control-K
End of Line	OA-> or OA-. (period)
Start of Line	OA-< or OA-, (comma)
Tab	Tab or Control-I
Tab Left	OA-A or Control-A
Top of Screen/Page Up Moves the cursor to the top of the screen, at the current column position. If it is already at the top, the screen scrolls up one page (22 lines).	OA-UpArrow or Control-OA-K
Word Left (Previous Word)	OA-LeftArrow or Control-OA-H
Word Right (Next Word)	OA-RightArrow or Control-OA-U
Insert Commands	
Insert Line Inserts a blank line ahead of the current line. The cursor may be anywhere on the line when you issue this command.	OA-B or Control-B
Insert Space Inserts a space at the current cursor position by moving the remaining characters one column to the right.	OA-Spacebar
Delete Commands	
Delete Block Puts the editor in "select" mode, where it highlights text as you move the cursor. Pressing Return removes the highlighted block from the screen; pressing Esc cancels the delete operation. *Note:* You cannot Undo the work of the Delete Block command.	OA-Delete
Delete Character	OA-F or Control-F
Delete Preceding Character	Delete or Control-D

Table 2-6 (cont.)

Delete Commands (cont.)	
Delete Line The cursor may be anywhere on the line.	OA-T or Control-T
Delete to End of Line	OA-Y or Control-Y
Delete Word The cursor may be anywhere within the word.	OA-W or Control-W
Remove Blank Lines If the cursor is on a blank line, this command deletes that line and any others up to the next nonblank line.	OA-R or Control-R
Undo (Restore) Last Deletion Restores text deleted by any of the preceding commands except Delete Block.	OA-Z or Control-Z
Search and Replace Commands	
Search Down (Forward)	OA-L
Search Up (Backward)	OA-K
Search Down and Replace	OA-J
Search Up and Replace	OA-H
Block Move and Copy Commands	
Cut Puts the editor in "select" mode, where it highlights text as you move the cursor. Pressing Return removes the highlighted material from the screen and stores it in a temporary disk file called SYSTEMP. Pressing Esc cancels the cut operation.	OA-X or Control-X
Copy Puts the editor in "select" mode, where it highlights text as you move the cursor. Pressing Return copies the highlighted block into a temporary disk file called SYSTEMP. Pressing Esc cancels the copy operation.	OA-C or Control-C
Paste Inserts the previously cut or copied text at the cursor position, by reading the contents of SYSTEMP.	OA-V or Control-V
Tab-Changing Command	
Set or Clear a Tab If there is a tab stop at the cursor position, this command removes it. If there is no tab stop at the cursor position, it sets one.	OA-Tab or Control-OA-I

2. The editor always starts in overstrike mode, replacing existing text with characters you type. However, you can put it in insert mode by pressing Control-E or OA-E (the status line shows *Mode: EDIT*

INSERT). Here, the editor inserts typed characters at the cursor position and shifts the remaining characters on the line to the right. To return to overstrike mode, press Control-E or OA-E again.

3. When you delete a character, line, or word, the editor saves it in an "Undo buffer" in memory. The Undo buffer acts like a barrel, where each newly-deleted unit is placed on top of the unit that was last deleted. The Undo (Control-Z) command removes a unit from the top of the buffer and inserts it at the cursor position. Thus, by doing successive Undos, you can restore everything you deleted!

As you can see, the editor's built-in power and flexibility make it more closely resemble a word processor than an ordinary "editor."

Leaving the Editor

When you finish editing a program, press Control-Q or OA-Q to leave the editor. This produces the editor menu, which looks like Figure 2-2.

Now, to save the program on disk, press S for "Save to the same name" (i.e., save with the name BEEP.SRC). The editor writes the program to disk and shows "Writing . . ." on the screen. When the *Enter selection:* prompt reappears, leave the editor entirely by pressing E for "Exit without updating." This makes #, the APW system prompt, reappear.

To make sure the program's source file is really on the disk, issue a

```
File name: BEEP.SRC

<R> Return to editor.

<S> Save to the same name.

<N> Save to a new name.

<L> Load another file.

<E> Exit without updating.

-------------------------------------------------------------------------------

Enter selection:
```

Figure 2-2

catalog command. The entry for the example program should have the *Name* BEEP.SRC, the *Type* SRC (for assembler source file), and the *Subtype* ASM65816.

Assemble, Link, and Run Commands

Now that the source file is on disk, you can use the commands the *Workshop* provides to assemble, link, and run it. Table 2-7 shows the commands you will probably use most often. To keep things simple, I have omitted some command parameters that are rarely used. Refer to the APW manual for complete details.

Table 2-7

Language Commands

ASM65816
Lets the language to 65816 assembler source, thereby informing the editor to produce a SRC type file.

CHANGE *pathname language*
Changes the language type of a source (SRC) or text (TXT) file. For example,

```
CHANGE MYFILE.SRC ASM65816
```

changes MYFILE.SRC to the assembler source code type, ASM65816. This is handy if the editor was set to the wrong type (say, EXEC) when you created a file.

EXEC
Sets the language to EXEC. EXEC files are used to store a list of *Workshop* commands. You run these commands by entering the name of the file.

Edit Command

EDIT [*pathname*]
EDIT loads a specified text file or assembler source file into the editor. If the file does not exist, the editor starts with a blank screen and assigns the specified filename to it.

Display Commands

CATALOG *[pathname]*
Displays the directory of the deafult disk (CATALOG) or subdirectory (CATALOG *pathname*) you specify. You can also use the = wildcard character in filenames. For example,

Table 2-7 (cont.)

Display Commands (cont.)

CATALOG /MYFILES/P=

displays all the files on the MYFILES disk that begin with the letter P. You may abbreviate CATALOG as CAT.

HELP [*command-name*]
Displays a descriptive summary of an *APW* command. For example, *HELP DELETE* summarizes the DELETE command. If you omit the command name, HELP lists the names of all available commands.

TYPE [+N] *pathname* [*startline* [*endline*]] [>*device*]
Lists the contents of a text, source, or EXEC file. The parameters are:

+N	Makes APW number the lines.
pathname	The pathname of the file.
startline	The number of the first line to be listed. Omitting this parameter causes the entire file to be listed.
endline	The number of the last line to be listed. Omitting this parameter causes all lines between *startline* and the end of the file to be listed.
>*device*	Sends the listing to an output device (e.g., >.printer). Omitting this parameter sends the listing to the screen.

Assembly and Link Commands

ASML [+L/-L] [+S/-S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [. . .]])

Assembles and links a source file. The optional parameters are:

+L/-L	Makes the assembler produce (+L) or omit (-L) a source listing. -L is the default.
+S/-S	Makes the assembler produce (+S) or omit (-S) an alphabetical listing of all global references in the object module. -S is the default.
KEEP	If your source file contains no KEEP directive, you must use this parameter to specify the name of the output (object) file. For a one-segment program, the object file is named *outfile*.ROOT. For a multi-segment program, the first (main segment) is stored in *outfile*.ROOT and the remaining segments are stored in *outfile*.A.
NAMES	Causes the assembler to assembled only the specified segments. The object code for these segments are stored in a file named *outfile*.B.

ASMLG [+L/-L][+S/-S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [. . .]])
Similar to ASML, except ASMLG (for "Assemble, Link, and Go") runs the program after linking it.

ASSEMBLE [+L/-L][+S/-S] *sourcefile* [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2* [. . .]])
Assembles a source file, but does not link it. Hence, ASSEMBLE does not produce a load module. To link ASSEMBLEd modules, use the LINK command.

Table 2-7 (cont.)

Assembly and Link Commands (cont.)

LINK [+S/-S] *objectfile* [KEEP=*outfile*]
or
LINK [+S/-S] *objectfile1 objectfile2* . . . [KEEP=*outfile*]

Links object modules to create an APW shell load file. The first form of the command is used to link object modules that have the same name (e.g., MUSIC.ROOT and MUSIC.A). The second form is used to link modules that have different names (e.g., to link modules named TUNE with those named MUSIC). The +S/-S parameter is as described for ASML. The others are:

objectfile The pathname (minus filename extension) of the object modules to be linked. All modules to be linked must have the same filename (except for extensions) and must be in the same subdirectory.

For example, if the program MUSIC consists of the object modules MUSIC.ROOT, MUSIC.A, and MUSIC.B, all located in directory /MYFILES, use /MYFILES/MUSIC for *objectfile*.

KEEP Use this parameter to specify the name of the load file. **Warning:** If you omit KEEP, the linker will not produce a load file.

objectfilen You can use a single LINK command to link object files having different names into one load file, by giving their pathnames, minus filename extensions.

For example, suppose /MYFILES contains the object modules for a program called MUSIC (say, modules MUSIC.ROOT, MUSIC.A, and MUSIC.B) and the modules for another program called TUNE (TUNE.ROOT and TUNE.A). To link TUNE's modules with MUSIC's, specify

/MYFILES/MUSIC /MYFILES/TUNE

for *objectfile*.

File and Directory Commands

COPY [-C] *pathname1* [*pathname2*]
Copies a file to a different subdirectory, or to a duplicate file with a different name. The parameters are:

-C Normally, if a file named *newpath* already exists, APW asks if you want to replace it. The -C option lets you copy without the prompt.

oldpath The pathname of the file to be copied. Wildcards may be used to copy multiple files. If *oldpath* is a directory, COPY copies the directory and any subdirectories and filesin it.

newpath The pathname of the copy. If you omit the pathname, the file is copied to the current directory.

CREATE *pathname*
Creates a new subdirectory with the specified pathname.

DELETE *pathname*
DELETE deletes the specified file. It can also delete a multiple files if you use a = wildcard in the pathname.

Table 2-7 (cont.)

File and Directory Commands (cont.)

ENABLE D[N][B][W][R] *pathname*
ENABLE enables one or more of the access attributes of a ProDOS file. It is generally used after a FILETYPE command, to guarantee the file has the attributes you want; in most cases, you enable all the attributes by specifying DNBWR.

FILETYPE *pathname filetype-abbrev*
Changes the ProDOS 16 "filetype" of a file. The filetypes are:

Abbreviation	File Type
BIN	ProDOS 8 binary load file
CDA	Classic desk accessory
DIR	Directory
EXE	Shell load
LIB	Library
NDA	New desk accessory
OBJ	Object
S16	ProDOS 16 system load file
SRC	Source
STR	Startup load
SYS	ProDOS 8 system load file
TOL	Toolkit load
TXT	Text

The *Programmer's Workshop* linker produces a shell load (EXE) file. To convert it to a ProDOS 16 system load (S16) file, enter:

filetype *pathname* S16

INIT *device* [name]
Formats a disk as a ProDOS volume. Here, *device* is the device number of the drive containing the disk to be formatted and *name* is the volume name for the disk. If you omit *name*, INIT names the disk BLANK.

MOVE *oldpath newpath*
Moves a file from one location to another; it does the same thing as a COPY command followed by a DELETE. The parameters are the same as for COPY.

PREFIX *directory*
Tells the *Programmer's Workshop* that any pathnames you enter are to be found in the specified *directory*. For example,

PREFIX /MYPROGS/MY.MACROS

tells it to look for files in the MY.MACROS directory of the disk named MYPROGS.

RENAME *oldpath newpath*
Changes the name of a file. The parameters are:

oldpath The pathname of the file to be renamed or moved.
newpath The pathname to which *oldpath* is to be changed or moved.

Table 2-7 (cont.)

File and Directory Commands (cont.)

SHOW [LANGUAGE] [LANGUAGES] [PREFIX] [TIME] [UNITS]
Lists information about the system in one or more of the following categories:

LANGUAGE	Lists the current default language.
LANGUAGES	Lists the available languages.
PREFIX	Lists the current subdirectories to which the ProDOS 16 prefixes are set.
TIME	Lists the current date and time.
UNITS	Lists the available units, including device names and (for disks) volume names.

Debugger Command

DEBUG
Starts the debugger utility.

The commands are divided into six groups: language, edit, display, assembly and link, file and directory, and debugger. You have already used the ASM65816 language command and the EDIT editor command. As Table 2-7 shows, the Programmer's Workshop provides assembly and link commands that let you do as much or as little as you want at one time. That is, you can:

- Assemble a file and link it later,
- Assemble and link it immediately (ASML), or
- Assemble, link, and execute it (ASMLG).

If you're an optimist, you may be inclined to assemble, link, and execute a program directly. You won't lose anything by doing that, because both the assembler and linker will stop if it can't proceed due to errors.

At this point, you should assemble, link, and execute the example program, *BEEP.SRC*, by entering:

```
ASMLG + L BEEP .SRC
```

(ASMLG is short for "Assemble, Link, and Go.") The assembler reads the source file from disk, then assembles it.

When the assembly process finishes, the listing shown in Figure 2-3

appears on the screen. (Well, not exactly; I cheated a little. Since the screen is only 80 columns wide, the comments "wrap" onto a new line. I omitted the comments from the figure so I could show each statement on a single line.)

Unfortunately, the listing scrolls by so fast that it's virtually unreadable. To stop the scrolling temporarily, and to resume after it's stopped, press any key.

Another solution — and probably a better one — is to make the assembler send its listing to your printer, rather than to the screen. To do this, insert a *PRINTER ON* directive at the beginning of your program, ahead of the KEEP statement. This assumes, of course, that you have configured the Apple IIGS to work with your particular printer and that the printer is "on-line."

The Apple IIGS assumes that you have a serial interface card in slot 1 that operates with a specific set of parameters (9,600 baud, 8 data bits, 1 stop bit, no parity, and so on). If you have a serial printer with different

```
0001  0000
0002  0000
0003  0000
0004  0000                              keep    beep
0005  0000
0006  0000                    Beep      START
0007  0000                    speaker   equ     $E0C030
0008  0000
0009  0000 A0 E8 03                     ldy     #1000
0010  0003 AF 30 C0 E0       BeepIt    lda     speaker
0011  0007 A2 E8 03                     ldx     #1000
0012  000A CA               Loop2     dex
0013  000B B0 FD                       bne     Loop2
0014  000D 88                         dey
0015  000E D0 F3                       bne     BeepIt
0016  0010 6B                         rtl
0017  0011                            END

17 source lines
0 macros expanded
0 lines generated
```

Figure 2-3

parameters, change the settings using the Control Panel's *Printer Port* option. If you have a parallel printer attached to slot 1, switch the slot setting to "Your Card" using the Control Panel's *Slots* option.

For this example and other short programs, you probably won't care about the listing. But when you're debugging long or complex programs, and trying to find that elusive "bug" that's making things go haywire, the listing can be quite useful — indeed, at times, indispensable. Therefore, it's worth spending a little time examining exactly what's being shown. The listing shows the following:

- The leftmost column shows the line numbers in decimal. (Except for instruction operands, the rest of the columns show hexadecimal numbers exclusively.)

- The second column shows where the object code for the statement (the instruction or directive) will be stored in memory *relative* to the starting address of the code. For example, if the system loader stores the program starting at location $800 of bank 1, the first instruction, *ldy #1000*, will be stored at location $800 (i.e., $800 + offset $0000); the next instruction, *lda speaker*, will be stored at $803 (i.e., $800 + offset $0003); and so on.

- The third column shows the hexadecimal byte value that will be stored in memory for that particular instruction or assembler directive.

  For an instruction, the byte represents the instruction's *opcode* — the numeric representation of the mnemonic and the addressing mode it uses. For example, ldy, which uses immediate addressing, is translated to the value $A0. (Appendix C summarizes the opcodes for the 65816's instruction set.)

  For an assembler directive that generates data (none of the directives in my program do), the byte in the third column represents the first value that the directive stores in memory. For example, the Declare Constant directive *dc C'H'* stores a byte value of $48 in memory, the ASCII code for the "H" character. (Appendix B summarizes the ASCII character set.)

- The remaining numeric columns show the operand values for each instruction that uses an operand. Some instructions take a 1-byte operand, others take a 2-byte or 3-byte operand, and a few, such as *dex, dey*, and *rtl*, take no operand at all.

For example, the operand "1000" for the *ldy* gets assembled into a 2-byte number, $03E8. Note that on the listing $03E8 appears as "E8 03"! This reflects the low-byte/high-byte order in which the 65816 stores numbers in memory. Similarly, the value of "speaker," $E0C030, is stored in low-byte to high-byte order — 30, then C0, then E0. Remember this arrangement when you're viewing data in memory.

- The remaining columns of the listing are, of course, the source program that you entered using the editor.

The lines below the listing provide additional information about the assembly. The first, "17 source lines," tells you that there are 17 lines in the source program (including blank lines), while "0 macros expanded" and "0 lines generated" refer to macros (a concept explained in a later chapter). There are no macros in this program; hence, the zeros.

When the linker finishes, it displays some information about the segments in the program (this has only one segment, Beep) and shows:

```
There is 1 segment, for a length of $00000011 bytes.
```

This says that the program is 17 ($11) bytes long. (The 8-digit length field suggests that the linker can handle exceptionally large programs!)

Finally, the program is loaded into memory and run. The beep sounds for about three seconds and then the # prompt reappears. CATALOGing the disk at this point will reveal three files for your program:

- BEEP.SRC — The source file you created using the editor.
- BEEP.ROOT — The object file that the assembler produced.
- BEEP — The load file that the linker produced.

BEEP.SRC, BEEP.ROOT, and BEEP have the types *SRC* (for source), *OBJ* (for object), and *EXE* (for shell load), respectively.

If everything went smoothly and you heard the beep, you have successfully entered, assembled, linked, and run the program (congratulations!). Otherwise, if you received any error messages, you must edit the source program, then give another ASMLG command.

Shell Load Files and System Load Files

You can run the shell load (EXE) file that the linker has produced by entering its pathname from the # prompt. However, to run it from the System Disk's Program Launcher, you must convert it to ProDOS 16 system load file format. To do this, enter the command:

```
filetype pathname sl6
```

After that, cataloging the disk will reveal that your program has a "Type" of S16, which identifies it as a ProDOS 16 system load file.

Automating the Assembly Process

If your program is long or complex (or even short, but error-ridden), you will probably have to assemble and link it many times. Typing the same ASML command over and over is pure drudgery, and if you're fumble-fingered like I am, you often wind up with the added monotony of retyping commands. Fortunately, there is a way to automate the process: put your commands in an *EXEC* (Executive) file.

An EXEC file contains a list of *APW* commands that the *Wordshop's* shell program will execute when you enter the file's pathname. For example, suppose you want to assemble and link a program called MYPROG.SRC, then convert the linker's output to a system load file. An EXEC file that does this would contain:

```
asml myprog.src keep=myprog
delete myprog.root
filetype myprog S16
```

Here, the *keep* on the first line indicates that there is no KEEP directive in the program. Note the command to delete MYPROG.ROOT. Once you have linked a program, its object file is no longer needed; you can discard it.

To create an EXEC file, enter **exec** from the # prompt (that changes the active language to EXEC), then start the editor with a command of the form **edit** *pathname*. EXEC files are, by convention, given the extension BUILD, so you may enter, say, **edit myprog.build**.

Once in the editor, enter the commands that belong in your EXEC file, then save it to disk. To run the file (i.e., to perform the commands in it), enter its name from the # prompt; e.g., enter **myprog.build**.

Multisegment Programs

The example program contains only one segment, Beep. However, if your program is large, you should divide it into functional segments. When the assembler assembles a multisegment program, it puts the first (main) segment in an object file that has the extension ROOT and puts all other segments in a file that has the extension A. For example, if MYPROG.SRC contains segments named Main, Init, and DoIt, the assembler would store Main in MYPROG.ROOT and store Init and DoIt in MYPROG.A.

The linker handles multiple object files automatically. If you enter an ASSEMBLE, ASML, or ASMLG command for MYPROG.SRC or a LINK command for MYPROG, the linker will find and link *every* object file whose name starts with MYPROG. (As I mentioned earlier, object files are unneeded after they have been linked. Your BUILD file should generally contain commands to delete them.)

Debugger

Sometimes a program doesn't do what you expected, but you don't know quite why. If you can't spot the problem by examining the assembler listing, you probably need the services of the *debugger*. The debugger is a handy utility that lets you run a program one instruction at a time, or keep running the program until you tell it to stop. Each time the debugger executes an instruction, it highlights the instruction it will execute next and shows the current values of the registers and the data on the stack.

The debugger can also display the contents of memory and *disassemble* programs; that is, it can display an assembled and linked program in its mnemonic form. Disassembling is convenient, for example, for examining programs stored in ROM.

Starting the Debugger

To start the debugger with the *APW* prompt on the screen, enter the command *debug*. When the debugger's initial screen appears, press Return to clear the copyright information at the bottom. Now you must load the program you want to debug. To do this, put the program's disk in the drive and enter a command of the form

```
load pathname
```

where *pathname* is the program's pathname.

Since you only have one program at this point, load it by entering **load beep**. When BEEP has been loaded, the screen will look similar to Figure 2-4.

Subdisplays on the Debugger Screen

The *Apple IIGS Programmer's Workshop* manual refers to the entire screen as the "master display," and to areas on the screen as "subdisplays." The two lines at the top of the screen are called the *register subdisplay*. They show the following:

- KEY is a keystroke modifier. The debugger normally interprets keystrokes as commands to itself (more about these commands shortly). Thus, if the program you are debugging involves any kind of keyboard input, you must tell the debugger that a keystroke is for your use by pressing another key (a "modifier") with it.

```
KEY  BRK  DebugD  K/PC     B   D    S    A    X    Y    M  Q  L  P  nvmxdizc e
00   o d    1400  01/0CE9  00  2C00 2FFF 100A 0000 0000 00 9E 0  00 00000000 0

3011:D0  00/0000: 71 'q'  00/0000-00-00
3010:8F  00/0000: 71 'q'  00/0000-00-00
300F:7E  00/0000: 71 'q'  00/0000-00-00
300E:80  00/0000: 71 'q'  00/0000-00-00
300D:03  00/0000: 71 'q'  00/0000-00-00
300C:A2  00/0000: 71 'q'  00/0000-00-00
300B:2C  00/0000: 71 'q'  00/0000-00-00
300A:B0  00/0000: 71 'q'  00/0000-00-00
3009:4A  00/0000: 71 'q'  00/0000-00-00
3008:60  00/0000: 71 'q'
3007:20  00/0000: 71 'q'  E1/0000.000F-T
3006:B0  00/0000: 71 'q'  00/0000.0000-?
3005:FD  00/0000: 71 'q'  00/0000.0000-?
3004:23  00/0000: 71 'q'  00/0000.0000-?
3003:20  00/0000: 71 'q'  00/0000.0000-?
3002:06  00/0000: 71 'q'  00/0000.0000-?
3001:90  00/0000: 71 'q'  00/0000.0000-?
3000:02  00/0000: 71 'q'  00/0000.0000-?
2FFF:C9  00/0000: 71 'q'  00/0000.0000-?

: (Command line)
```

Figure 2-4

You can make the debugger recognize Shift, Control, Caps Lock, Option, OpenApple, a keypad key, or a repeating key as a keystroke modifier. Details are in the *Programmer's Workshop* manual.

- BRK is a breakpoint flag. A breakpoint is a location in a program at which the debugger is to stop executing and let you examine the registers or memory. I discuss breakpoints later.
- DebugD points to a direct page that the debugger uses internally. You can generally ignore it.
- The next seven entries are the 65816's registers: K/PC is program bank register and program counter, B (data bank register), D (direct page register), S (stack pointer), A, X, and Y. Here, K/PC points to the first instructions in your program, although that instruction is not yet displayed on the screen.
- M and Q are the so-called machine-state and quagmire registers. These registers aren't very useful for most applications.
- L is the language card bank.
- P is the processor status register. Its individual bits are shown at the end of the register subdisplay. Note that *m*, *x*, and *e* are 0; the debugger assumes you want full native mode.

The debugger provides commands that let you change the contents of any register, and I will discuss them shortly.

The leftmost column on the screen is the *stack subdisplay*. It shows the address and contents of the memory locations that precede and follow the location pointed to by the stack pointer. The entry in inverse video at the bottom indicates the current stack location.

The columns that extend to the end of the K/PC data comprise the *RAM subdisplay*. By entering **mem**, you can display the contents of up to 19 different locations in memory — one memory entry on each line. Once the cursor is in the RAM subdisplay, move to the line you want and type a hexadecimal address; the digits shift left as you type them. Then type one of three letters:

- *H* displays the 1-byte hex contents of that location and its corresponding ASCII character (e.g., FA 'z').
- *P* displays the contents of the location and the next location as a 4-digit number (e.g., E9E6).

- *L* displays the contents of the location and the next two locations as a 6-digit number (e.g., 008721).

To return to the command line, press **Esc**.

The rightmost column in Figure 2-4 contains two subdisplays: the *breakpoints subdisplay* at the top and the *memory protection subdisplay* at the bottom. I discuss the breakpoints subdisplay later, under "Breakpoints." The memory protection subdisplay is only useful for advanced applications, but you might like to read about it in the *Programmer's Workshop* manual.

The blank area at the right of the screen is reserved for the *disassembly subdisplay*, in which the debugger will display your program. Now it's time to look at the available debugger commands.

Table 2-8 lists the most commonly used debugger commands. Note that they are divided into six groups, arranged in order of importance.

Single-Step and Trace Commands

These commands let you execute a program one instruction at a time (single step) or execute it until you tell the debugger to stop (trace). In either case, you can tell the debugger to start at the current K/PC address or at another address of your choice.

The debugger will accept an address in nearly any form, as long as you follow it with an S or T. Hence, 104EDS, 0104EDS, 1/04EDS, and 01/04EDS are all valid commands to enter step mode at location $04ED of bank 1. If you omit the bank number (and enter, say, 04EDS), the debugger assumes you're referring to the current program bank, so it obtains the bank number from the program bank register (K).

Once the debugger is in single-step or trace mode (*SINGLE STEP* or *TRACE* appears at the bottom of the screen), you can enter single-key "subcommands" to trace to the end of a subroutine, skip the next instruction (usually BRK), turn the sound on or off, suspend tracing, or change the trace speed.

The *T* subcommand, which changes the screen to text mode, is important for any program that displays graphics. Once the program starts running, it switches the screen from the debugger's display to the application's, and you lose sight of the instructions being traced. The T subcommand puts the screen back into the debugger, where you can once again follow the trace operation.

Table 2-8

Command	Description
	Single-Step and Trace Commands
S	Enter single-step mode at the current instruction. The current instruction. The current instruction is the one the 65816 will execute next, the one K/PC points to; the debugger highlights it on the screen. To execute that instruction, press the spacebar; to leave single-step mode, press Esc.
*address*S	Enter single-step mode at *address*.
T	Enter trace mode at the current instruction. The debugger begins executing immediately, and stops only when you press Esc or it encounters a breakpoint or BRK instruction.
*address*T	Enter trace mode at *address*.
The following are subcommands. They are only available when the debugger is in single-step or trace mode.	
Esc	Exit single-step or trace mode.
OpenApple	Stop tracing until the OpenApple key is released.
Spacebar	Execute highlighted instruction (in single-step mode).
R	Trace up to the next RTS, RTI, or RTL instruction. This lets you trace through one subroutine at a time.
T	Change the display to text mode.
Left-arrow	Change to the slow trace speed.
Right-arrow	Change to the fast trace speed (default).
Down-arrow	Skip the next instruction. This is convenient for skipping over a BRK, for example.
Q	Turn the sound off if it is on, or vice versa.
	Editing Commands
Control-E	Toggle insert/replace mode. In insert mode, typed characters are inserted betweem existing characters; in replace mode, typed characters replace existing characters.
Delete	Delete the character to the left of the cursor.
Control-F	Delete the character at the cursor position.
Esc	Delete the current line.
Control-Y	Delete to the end of the line.
Return	Execute the command. The cursor can be anywhere on the line.

Table 2-8 (cont.)

Command	Description
	Register Commands
e	Toggle the emulation mode flag.
m	Toggle the index register flag.
x	Toggle the memory/accumulator flag.
register = value	Load the specified register with the specified value. Register names are: KEY — Key modifier K — Program bank register PC — Program counter B — Data bank register D — Direct page register S — Stack pointer A — Accumulator X — Index register X Y — Index register Y M — Machine-state Q — Quagmire L — Language card bank P — Processor status register
	Memory Commands
address:	Display 368 consecutive bytes in memory, starting at *address*. The debugger also shows the ASCII equivalent of each byte in either normal video (values between $20 and $7F) or inverse video (values between $C0 and $FF). ASCII values that can't be displayed appear as either a period ($00 to $1F) or an inverse period ($80 to $BF).
address:value(s)	Store the hexadecimal *value(s)* in memory, starting at *address*. For example, the following command stores $A1 at location $100 in bank 1: 01/0100:A1 You can store up to three bytes at a time. For example, the following command stores $A1 at 01/0100, $A2 at 01/0101, and $A3 at 01/0102: 01/0100:A1A2A3
address:'string	Store the values corresponding to *string* in memory, starting at *address*. The high bit of each byte is 0, which produces normal video if you display the string.
address:"string	Store the values corresponding to *string* in memory, starting at *address*. The high bit of each byte is 1, which produces inverse video if you display the string.

Table 2-8 (cont.)

Command	Description
	Memory Commands (cont.)
address:instruction	Assemble the specified *instruction* and store its opcode and operand at *address*. The debugger will display the disassembled form of the new instruction when you enter single-step or trace mode.
	Disassembly Commands
*address*L	Disassemble 19 instructions, starting at *address*.
L	Disassemble the next 19 instructions, starting at the current K/PC address.
ASM	Clear the disassembly subdisplay. This only removes the disassembled instructions from the screen; it does not affect K/PC.
	Conversion Commands
value =	Convert *value* from hexadecimal to decimal. The *value* can range from 0 to FFFF.
+ *value* =	Convert *value* from decimal to hexadecimal. The *value* can range from 0 to 65535.
− *value* =	Convert *value* from decimal to hexadecimal. The *value* can range from 0 to −32768.

Editing Commands

The editing commands let you correct typing errors on the command line. Note that except for Esc, these are the same commands you use in the editor.

Register Commands

You can change any register by entering a command of the form

```
register=value
```

where *register* is the abbreviation for the register (it must be uppercase) and *value* is a hexadecimal number. For example, X = 10 loads $10 (decimal 16) into the X register. You may want to change a register at some point in a program to see what would happen under some alternate set of circumstances. You may also want to change an index register to terminate a long loop operation that you don't care about.

You can also reverse, or "toggle," the setting of the status register's

emulation, memory/accumulator, or index register bit by entering *e*, *m*, or *x*, respectively. You probably won't do this very often, however; the program should regulate these bits.

Memory Commands

These commands let you display and, optionally, alter the contents of memory. The *address:* command makes the debugger display a screenful of bytes in memory (368 bytes, to be exact), starting at the specified address. It also shows the ASCII equivalent of each byte, where applicable, and shows noncharacter bytes as either a period (.) or inverse period. To leave the memory display and return to the regular debugger screen, press Esc.

Of course, the *mem* command also displays memory, on the RAM subdisplay, but it can only show 3 bytes on a line — or 1 byte, if you want the ASCII representation. The *address:* command provides much more information, but it doesn't let you see the registers or your program at the same time. Alas, nothing comes for free.

You can also change data in memory, by entering a hexadecimal value or string after *address:*. Finally, if you enter an instruction after *address:*, the debugger will assemble it and store its opcode and operand starting at that location. For example,

```
0834:LDY #0010
```

stores $A0 (opcode for LDY immediate), $10, and $00 at locations $834, $835, and $836 of the current program bank. The operand 0010 tells the debugger that the 65816 is running in native mode with 16-bit index registers (status flag X = 0). The debugger isn't smart enough to know that the shorter form, LDY #10, means the same thing if Y is 16 bits long, because it doesn't look at the X flag. It simply assembles what you give it.

Disassembly Commands

Sometimes you may want to look at instructions in other parts of memory, say, a subroutine in ROM or instructions in another part of your own program. To do this, you can make the debugger "disassemble" memory starting at a specified address by entering *address*L. Figure 2-5 shows a typical disassembly that was produced with the command 01/1200L. The debugger always disassembles 19 instructions, but only the first eight are shown here.

Note that for the program counter relative mode and relative long

```
12/1000: AD 15 18     LDA 1815
12/1003: 9D 50 10     STA 1050, X
12/1006: 9F 20 30 05  STA 050320
12/100A: A9 77 66     LDA #6677
12/100D: 82 20 10     BRL 2030 (+1020)
12/1010: 80 20        BRA 1032 (+20)
12/1012: F4 12 34     PEA 3412
12/1015: 62 10 10     PER 2028 (+1010)
. .
. .   11 more instructions
. .
```

Figure 2-5

modes, as used by the BRL, BRA, and PER instructions, the debugger lists the effective address followed in parentheses by the relative displacement. You probably won't care about the displacements, but they're there if you ever need them.

Conversion Commands

Finally, the debugger can perform on-screen conversions of hex-to-decimal or vice versa. You might use one of these commands to determine the decimal form of a hexadecimal operand that the debugger has displayed. Once you give a conversion command, it remains on the command line until you press Esc.

Breakpoints

As mentioned earlier, there are two ways to execute a program using the debugger. You can single step through it one instruction at a time, or you can trace through it, executing instructions until the processor encounters a BRK instruction or you press Esc. The debugger also lets you set *breakpoints* in a program.

A breakpoint is a location at which the debugger is to stop tracing so you can examine the contents of registers and memory. Breakpoints can be valuable for locating an elusive "bug" that's making your program produce incorrect answers or crash. They put *you* (rather than the microprocessor) in control of the program.

How you use breakpoints will depend on your application. You might set one at the instruction that follows a loop, to see what the loop produced

before continuing. You could also set a breakpoint to determine whether a particular instruction is ever executed; if tracing never stops there, the instruction has not been executed.

The debugger lets you set up to 17 breakpoints. The breakpoints subdisplay (above the memory protection subdisplay) has space for nine entries, but you can enter more by using some of the memory protection subdisplay's lines.

Entries in the breakpoints subdisplay are initially set to *00/0000-00-00*, where the first item is (as you may have guessed) the breakpoint address. The second item is the "trigger value"; the number of times the debugger is to execute the breakpoint instruction before stopping. The third item is a repetition counter. During tracing, it displays a running total of the number of times the instruction has been executed.

To set a breakpoint, enter **bp** on the command line. This moves the cursor to the hyphen that separates the address and trigger value in the first entry. To enter a breakpoint there, type its address, then press the right-arrow key and type the trigger value. If that's the only breakpoint you want, press Esc to return to the command line; otherwise, press the down-arrow key to reach the next line where you want to enter a breakpoint. If you make a mistake entering a breakpoint or decide you don't want it, move to its line and either reenter it or press Delete to clear it.

You can also enter three breakpoint commands from the command line: *clr* clears all breakpoints, *in* makes the debugger insert BRK instructions at the breakpoint locations (the register subdisplay's BRK entry shows "i"), and *out* removes the BRKs that *in* inserted (the BRK entry shows "o").

Leaving the Debugger

To leave the debugger and restore the # prompt, press *Q* for Quit.

CHAPTER 3

65816 Addressing Modes

The 65816 provides 24 different ways to obtain the data on which your program is to operate. Table 3-1 lists these addressing modes in the order I describe them in this chapter. It also gives the assembler format for each mode (which is how the 816 differentiates them) and shows, with shading, which modes are new with the 816.

In this table, *Loc* and *LongLoc* represent 16- or 24-bit addresses in a program bank or data bank (which one depends on the instruction), while *DLoc* and *DLongLoc* represent 16- or 24-bit addresses in the direct page. These memory address operands are generally labels, and the assembler converts them to the numeric addresses they represent.

For example, suppose your program contains the instruction JMP NEWLOC (i.e., jump or GOTO the instruction labeled NEWLOC), where NEWLOC is at location $1200 in the program bank. When you assemble the program, the assembler will translate NEWLOC into the numeric address $1200.

The terms *direct*, *indirect*, and *indexed* need a general introduction. Direct means that the address of the data is in the direct page (called the zero page in 6502 literature). Indirect means that the address of the data is in the specified location. Indexed means that the address of the data is obtained by adding the contents of an index register to the specified address.

Table 3-1

Mode	Operand Format
Immediate	#Num
Accumulator	A
Implied	*blank* (no operand)
Absolute	Loc
Absolute Long	LongLoc
Absolute Indirect	(Loc)
Absolute Indexed with X	Loc,X
Absolute Indexed with Y	Loc,Y
Absolute Long Indexed with X	LongLoc,X
Absolute Indexed Indirect	(Loc,X)
Direct (Zero Page)	DLoc
Direct Indirect	(DLoc)
Direct Indirect Long	[DLongLoc]
Direct Indexed with X	DLoc,X
Direct Indexed with Y	DLoc,Y
Direct Indirect Indexed	(DLoc),Y
Direct Indirect Indexed Long	[DLongLoc],Y
Direct Indexed Indirect	(DLoc,X)
Program Counter Relative	Disp8
Program Counter Relative Long	Disp16
Stack	(Various operands)
Stack Relative	Disp8,S
Stack Relative Indirect Indexed	(Disp8,S),Y
Block Move	srcbk, destbk

Notes: (1) Shaded modes are new with the 65816; they are not available on the 6502.
(2) Symbols have the following meanings:
Num = 8- or 16-bit constant
Loc = 16-bit address
LongLoc = 24-bit address
DLoc = 16-bit address in the direct page
DLongLoc = 24-bit address in the direct page
Disp8 = 8-bit signed relative displacement (distance forward or backward)
Disp16 = 16-bit signed relative displacement
srcbk = Source bank number
destbk = Destination bank number

Immediate

The immediate addressing mode lets you specify a *constant* as the operand. Here, you must precede the constant with a # symbol. For example, the instruction

```
LDA  #$4A
```

loads the hexadecimal value 4A (decimal 74) into the accumulator (the A register).

Sometimes you will want to work with the *address* of a memory location, rather than its contents. (Address operations are common in the IIGS Toolbox.) Memory addresses are 3 bytes long — a 1-byte bank number and a 2-byte offset — and are stored in offset/bank number order in memory. Thus, to read the offset into a register, you would do a load immediate of the location's label (e.g., LDA #ThisLabel); to read the bank number, you would do a load immediate of the label *plus 2* (e.g., LDA #ThisLabel + 2).

Accumulator

In the accumulator mode, the operand is in the accumulator. For example, this instruction increments the accumulator (adds 1 to it):

```
INC A
```

Instructions that use the accumulator addressing mode are only 1 byte long, because the opcode provides all the information the processor needs.

Implied

About half of the 65816's instructions perform simple tasks such as setting or clearing a bit in the processor status register, incrementing or decrementing a register, or copying the contents of one register into another. These instructions need no operand because the operand is "implied" in the opcode. Some examples are:

```
CLC       ;Clear Carry Flag
DEY       ;Decrement the Y Register
INX       ;Increment the X Register
TAY       ;Transfer Accumulator to Y Register
```

All implied addressing instructions are 1 byte long.

Absolute and Absolute Long

Absolute addressing allows you to access any of the 64K locations in a bank of memory. Absolute addressing instructions are 3 bytes long: the opcode,

followed by a 16-bit address. When you're accessing data, the 65816 uses the data bank register (DBR) to select the bank. For example, if the location DATALOC is in the active data bank,

```
LDA  DATALOC
```

reads its contents into the accumulator.

The instructions Jump (JMP) and Jump to Subroutine (JSR) — the assembly language counterparts of GOTO and GOSUB in BASIC — also use absolute addressing. However, because they are used to move within a program, the 65816 employs the program bank register (PBR), rather than the data bank register, to select the bank. For example,

```
JMP  THERE
```

makes the 816 continue at the instruction that is labeled THERE, by putting its address in the program counter.

You may be wondering what happens if the target location or instruction is in a different bank than the one to which the bank register (DBR or PBR) is pointing. When that happens, the assembler encodes the instruction using the *absolute long* mode. In absolute long mode, the instruction contains a 24-bit address that points to your target location. Here, the high-order 8 bits of the address specify the bank, while the remaining 16 bits specify the location within that bank. For example, if DATALOC is not in the active data bank, the assembler assembles

```
LDA  DATALOC
```

using absolute long addressing.

The key point is that you needn't be concerned about *where* your data is located. The assembler will determine its location and apply the correct mode, absolute or absolute long, automatically.

In general, absolute operands are labels, but you may (for some reason) want to specify the numeric form of an address. To do this, simply enter the number directly. For example, *LDA $100* reads the contents of location $100 in the active data bank. However, to use a numeric operand for absolute long addressing, you must precede it with a < symbol. For example, *LDA <$100* reads the contents of location $100 in bank 0.

Absolute Indirect

Absolute indirect is really two addressing modes, one used only by the JMP instruction and the other used only by the JML instruction. (As mentioned in the preceding section, JMP is the assembly language version of BASIC's GOTO. JML, short for JumpLong, is the same, except it can transfer to any bank; JMP is limited to the current bank.) The term *indirect* here indicates that the operand *contains* the target address; the operand is not itself the target address, as it is for a JMP absolute instruction.

In the case of JMP, the indirectly addressed location contains a 2-byte address in the 65816's standard order: with the high byte following the low byte. The 65816 combines this address with the 8-bit program bank register (PBR) to produce the 24-bit destination address. For example,

```
JMP  (TADD)
```

makes the 65816 transfer to the location in the active program bank whose address is contained in the 16-bit location TADD. Figure 3-1 shows how this instruction operates if TADD contains $014C.

In the case of JML, the indirectly addressed location contains all 3 bytes of the destination address, with the low byte first. To make the transfer, the 65816 loads the low and middle bytes into the program counter (PC) and the high byte into the program bank register (PBR).

You may be thinking, "Why all this rigmarole? If your destination is some specific address, why not simply use regular absolute long addressing to load it into the program counter and PBR?" The answer is that absolute

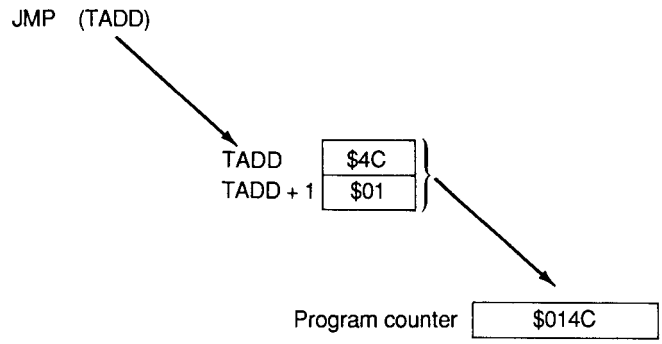


Figure 3-1

indirect addressing allows you to work with *variable* destination addresses. For example, in a program where a user chooses from options in a menu, you can use the option number to look up the address of that option's subprogram in a table, store the address in memory, then jump to it using absolute indirect addressing. Absolute indirect addressing also allows the effective address to be in read/write memory (RAM) even when the program is in ROM or PROM.

The absolute indirect mode has one restriction that limits its use: *the location that holds the indirect address must be in bank 0*! That being the case, you may want to make indirect jumps using the absolute indexed indirect mode (discussed shortly), which doesn't have the bank 0 restriction.

Absolute Indexed with X or Y

In absolute indexed addressing, the 65816 computes the effective address by adding the contents of an index register (X or Y) to the absolute address in the instruction. That is,

```
Effective address = Absolute address + X
```

or

```
Effective address = Absolute address + Y
```

All absolute indexed instructions are 3 bytes long. Absolute indexed operands are formed by attaching a ,*X* or ,*Y* to the address or address label.

Absolute indexed addressing is particularly useful for accessing tables. Here, you would enter the table's name as the address operand and use an index register to select the item you want. For example, if DTABLE is a table of 16-bit words and X contains 6, this instruction

```
LDA  DTABLE,X
```

loads the third word of DTABLE into the accumulator. Figure 3-2 shows how this instruction operates.

Note that it is necessary to put 6 into X because you are referring to a table of words (rather than bytes), and words are 2 bytes long. Using 3 here would cause the 816 to load the third and fourth bytes of DTABLE into the accumulator, and that's not what you intended.

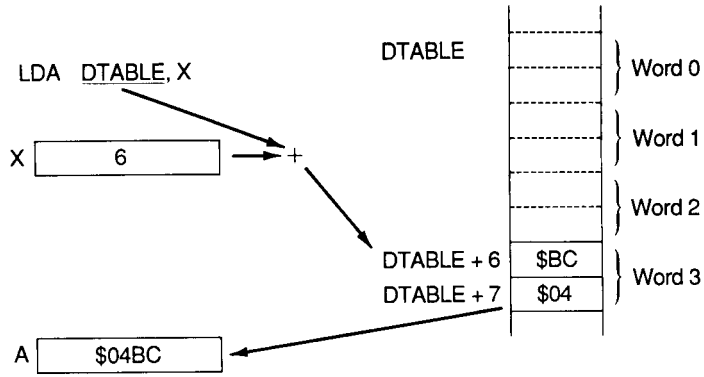


Figure 3-2

Absolute Long Indexed with X

The absolute indexed modes just described always obtain their operands from the active data bank — that is, from the bank to which the data bank register (DBR) is pointing. If you give an absolute indexed with X instruction that refers to a location in some other bank, the assembler assembles it using the absolute *long* indexed with X mode. In that case, the instruction is 4 bytes long, rather than 3, because it contains a 3-byte (24-bit) address. Note that there is no comparable mode for the Y register.

If your operand is a number rather than a label, you must tell the assembler which mode to use by preceding the absolute long indexed operand with a > symbol. For example, *LDA $100,X* refers to location $100 in the active data bank, while *LDA* >$100,X refers to location $100 in bank 0.

Absolute Indexed Indirect

This is a combination of two modes that were discussed earlier: absolute indirect and absolute indexed with X. Recall that with absolute indirect addressing, the operand is the address of the location that *contains* the effective address. Recall also that absolute indexed with X involves adding a displacement in the X register to a base address in the instruction to produce the effective address.

With absolute indexed indirect addressing, the 65816 adds the contents

of the X register to the absolute address in the instruction to obtain the indirect address. That is,

```
Indirect address = Absolute address + X
```

then

```
Effective address = (Indirect address)
```

where the parentheses mean "contents of." Absolute indexed indirect operands have the general form (Loc,X).

To see how this mode is used, suppose your program displays a menu and prompts the user to type a number from 0 to 4 to select an option. Suppose also that it has a table called JTABLE that contains five 16-bit addresses, one for each option. To begin, the program reads the number into the X register and doubles it (because you're accessing words, not bytes). Then, to start the routine for the selected option, it executes:

```
JMP  (JTABLE,X)
```

where JTABLE is assumed to be in the current program bank (*not* the data bank). Figure 3-3 shows how this instruction operates if X contains 4.

Direct

Within bank 0 of memory, there is a certain page (i.e., a 256-byte block of locations) that the 65816 can access faster than any other page. If you think

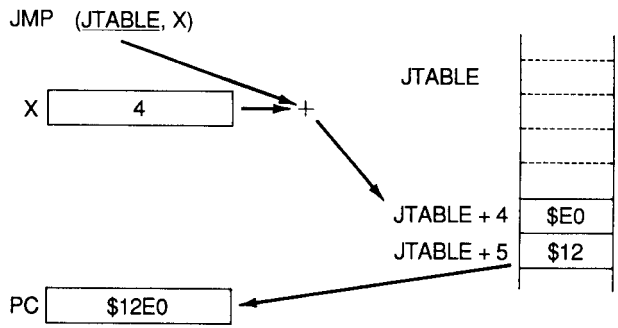


Figure 3-3

of the computer's memory as a group of boxes in a post office — one for each mail carrier in a city — consider this "direct" page (or *zero page*, in 6502 terminology) as the box that is closest to the person sorting mail. Because of this direct box's proximity, the sorter can toss items into it immediately, without even thinking. However, whenever the sorter encounters an item that belongs in any other box, he or she must first search for the box, and that takes more time.

The 65816 acts exactly like the mail sorter in my hypothetical post office; it can transfer data between the direct page — the page in bank 0 at which the direct register (D) is pointing — and a register faster than it can transfer to or from any other page. Thus, when you give an instruction that refers to a location in the direct page, the assembler says, "Oh, that's in the direct page. The 816 can do that operation faster than normal." It then assembles the instruction using *direct* addressing.

Direct addressing is, then, a form of absolute addressing in which the 65816 accesses only a specific 256-byte page within bank 0. Direct addressed instructions are one byte shorter and one clock cycle faster than absolute addressed instructions. To construct the operand address, the 65816 adds the 8-bit offset in the instruction to the 16-bit contents of the direct register (D). Again, the assembler will use direct addressing rather than absolute addressing automatically, where applicable; you needn't tell it which mode to use.

Except for JMP (Jump) and JSR (Jump to Subroutine), all instructions that can use absolute addressing can also use direct addressing. However, considering the savings in storage space and execution time with direct addressing, you should, whenever possible, use the direct page to hold frequently accessed data. The direct page is also useful for storing temporary data values.

Direct Indirect and Direct Indirect Long

With direct indirect addressing, the processor adds an 8-bit offset contained in the instruction to the 24-bit contents of the direct register to produce an address within the direct page. It then combines the 16-bit contents of the addressed location with the 8-bit data bank register (DBR) to produce the 24-bit effective address of the data. The operands for direct indirect instructions have the general form (DLoc). For example, if DLOC is in the direct page,

```
LDA  (DLOC)
```

uses the contents of DLOC as an address in the active data bank from which it obtains the data to be loaded into the accumulator.

The direct indirect long mode is the same, except the processor obtains a 24-bit (3-byte) address from the indirect location. Direct indirect long operands have the general form [DLongLoc]. For example,

```
LDA  [DLOC]
```

Direct Indexed with X or Y

In direct indexed addressing, the 65816 computes the effective address by adding the contents of the direct register (D) and an index register (X or Y) to the 8-bit offset in the instruction. That is,

```
Effective address = Offset + D + X
```

or

```
Effective address = Offset + D + Y
```

All direct indexed instructions are 2 bytes long, and their operands are formed by attaching a *,X* or *,Y* to the offset.

Direct indexed addressing is particularly useful for accessing tables. Here, you would enter the table's name as the offset operand and use an index register to select the item you want. For example, if DTABLE is a table of 16-bit words and X contains 6, this instruction

```
LDA  DTABLE,X
```

loads the third word of DTABLE into the accumulator.

Direct Indirect Indexed and Direct Indirect Indexed Long

Direct indirect indexed addressing is similar to direct indirect addressing, except that after obtaining the indirect address, the 65816 adds the contents of the Y register to it to form the 24-bit effective address. That is:

```
Effective address = (DLOC) + Y
```

This is useful if the indirectly addressed location points to a table. Then, adding the Y register produces the address of an element in the table. Direct indirect indexed operands have the general form (DLoc),Y.

Direct indirect indexed long is the same, but the 816 extracts a 24-bit addresss, rather than a 16-bit address, from the indirectly addressed location. Direct indirect indexed long operands have the general form [DLongLoc],Y.

Direct Indexed Indirect

The direct indexed indirect mode is somewhat like the similar-sounding direct indirect indexed mode, except that the 65816 uses the X register to calculate the location in the direct page from which it is to obtain the address of the data. That is,

```
Indirect address = D + offset + X
```

then

```
Effective address = (Indirect address)
```

Direct indexed indirect operands have the general form (DLoc,X).

This mode is useful for selecting an address from a table of addresses in the direct page. If ADDRTBL is such a table,

```
LDA  (ADDRTBL,X)
```

extracts the address at offset X and uses it to obtain the operand that it loads into the accumulator.

Program Counter Relative and Program Counter Relative Long

Program counter relative addressing is just what its name implies: the effective address is some offset from the current value of the program counter (PC). Since the 65816 updates the PC before it executes an instruction, the PC will be pointing to the *next* instruction when the offset is applied. A positive offset produces a higher-numbered address; a negative offset produces a lower-numbered address.

Program counter relative addressing is only used by the 65816's *branch* instructions. These instructions make the 816 transfer forward or backward if a specified condition is met (e.g., if a preceding add operation produced a zero result). Otherwise, if the condition is not met, the 816 proceeds to the next instruction. For example, the following BCC (Branch on Carry Clear) instruction will make the 816 transfer to the instruction at NEXT if the carry flag is 0, or to NOTTEXT if carry is 1:

```
            BCC   NEXT    ;Is carry =0?
NOTNEXT     LDA   #$3A    ;No. Continue here.
              .
              .
NEXT        LDA   #4      ;Yes. Transfer here.
```

Branch instructions are two bytes long, where the second byte contains the signed offset. With an offset of 8 bits, regular branch instructions can only transfer 127 bytes forward or 128 bytes backward — about half a page in either direction. However, there is a Branch Long (BRL) instruction that can transfer halfway up or down the program bank. BRL uses *program counter realtive long* addressing, which adds a 16-bit offset to the PC.

Stack

This is actually several different modes that are used by various instructions that access the stack, either to "push" data onto it or "pull" data off it.

Stack Relative and Stack Relative Indirect Indexed

These are two modes that you can use to access data on the stack. These modes are rarely used, however; most people only put data on the stack or retrieve data from it. Still, it's nice to know that the 65816 provides some stack-accessing capability if you ever need it.

With stack relative addressing, the 816 adds an 8-bit signed offset in the instruction to the 16-bit contents of the stack pointer (S) to produce an effective address in bank 0. Stack relative operands have the general form Disp8,S.

Stack relative indirect indexed addressing is similar, except that the sum of the offset and the stack pointer is an *indirect* address. The 65816 adds

the value in the Y register to the contents of the indirectly addressed location to produce the effective address. Stack relative indirect indexed operands have the general form (Disp8,S),Y.

Block Move

This mode is only used by the Block Move Negative (MVN) and Block Move Position (MVP) instructions. These instructions copy a specified number of bytes from a "source" bank to a "destination" bank, starting at either the beginning of the source block (MVN) or the end of it (MVP).

The two distinct block move instructions are designed to keep you from overwriting data. For example, if you move a block forward one byte position in memory (to the next higher numbered address), and start at the beginning of the block, the first byte you move will overwrite the second byte. Instead, you must start moving from the end of the block, using MVP.

Similarly, if you move a block downward one byte position (to the next lower numbered address), and start at the end of the block, the first byte you move will overwrite the preceding byte. This time you must start moving from the beginning of the block, using MVN.

The operands for the block move instructions have the general form *srcbk,destbk*, where *srcbk* and *destbk* are the numbers of the source and destination banks, in hexadecimal. For each instruction, the X register contains the offset of its destination, and the A register specifies the number of bytes to bemoved *minus one*. Hence, if X and Y contain 4 and A contains 99, the instruction

```
MVP  5,6
```

tells the 65816 to copy 100 bytes from bank 5 to bank 6, starting with the fifth byte (byte 4).

Addressing Mode Summary

As you now know, the addressing modes that access memory have many variations, and they can be somewhat confusing. To help sort things out, I have included Table 3-2. This table lists the name and operand format for each mode, but in addition it shows — in equation form — how the 65816 calculates the actual address of an operand; that is, its *effective address*.

Table 3-2

Mode	Operand Format	Effective Address =
Absolute	Loc	Loc
Absolute Long	LongLoc	LongLoc
Absolute Indirect	(Loc)	(Loc)
Absolute Indexed with X	Loc,X	Loc + X
Absolute Indexed with Y	Loc,Y	Loc + Y
Absolute Long Indexed with X	LongLoc,X	LongLoc + X
Absolute Indexed Indirect	(Loc,X)	(Loc + X)
Direct (Zero Page)	DLoc	DLoc
Direct Indirect	(DLoc)	(DLoc)
Direct Indirect Long	[DLongLoc]	(DLongLoc)
Direct Indexed with X	DLoc,X	DLoc + X
Direct Indexed with Y	DLoc,Y	DLoc + Y
Direct Indirect Indexed	(DLoc),Y	(DLoc) + Y
Direct Indirect Indexed Long	[DLongLoc],Y	(DLongLoc) + Y
Direct Indexed Indirect	(DLoc, X)	(DLoc + X)
Program Counter Relative	Disp8 or Loc	PC + $Disp^8$
Program Counter Relative Long	Disp16 or Loc	PC + Disp16
Stack Relative	Disp8,S	S + Disp8
Stack Relative Indirect Indexed	(Disp8,S),Y	(S + Disp8) + Y

Notes: (1) Shaded modes are new with the 65816; they are not available on the 6502.
(2) Symbols have the following meanings:

Loc = 16-bit address.
LongLoc = 24-bit address.
DLoc = 16-bit address in the direct page.
DLongLoc = 24-bit address in the direct page.
Disp8 = 8-bit signed relative displacement (distance forward or backward).
Disp16 + 16-bit signed relative displacement.

For example, in an 8-bit mode, an immediate instruction such as LDA #10 will execute in two cycles and occupy two bytes in memory, versus three cycles and three bytes in full native mode. Appendix C gives time and storage data for each instruction.

Note that the table also correlates each addressing mode with the microprocessor on which it was introduced. This is intended to benefit readers who have done some assembly language work on earlier models of the Apple II, and those who are writing programs which are to be 6502 and 65C02 compatible.

Read-Modify-Write Instructions

Four of the modes in Table 3-3 have increased cycle times when they are being used with a Read-Modify-Write instruction. As the name implies,

Table 3-3

Addressing mode	Example	Cycles	Bytes	Introduced in 6502	65C02	65186
Accumulator	INC A	2	1	x		
Immediate	LDA #30	3	3	x		
Implied	INX	2	1	x		
Absolute				x		
Read-Modify-Write ins.	INC Loc	8	3			
Other instructions	LDA Loc	5	3			
Absolute Long	LDA LongLoc	6	4			x
Absolute indexed with X				x		
Read-Modify-Write ins.	INC Loc,X	8[1]	3			
Other instructions	LDA Loc,X	5[1]	3			
Absolute long indexed with X	LDA LongLoc,X	6	4			x
Absolute indexed with Y	LDX Loc,Y	5[1]	3	x		
Absolute indirect	JMP (Loc)	5	3	x		
Absolute indexed indirect	JMP (Loc,X)	6	3		x	
Direct				x		
Read-Modify-Write ins.	INC DLoc	7[3]	2			
Other instructions	LDA DLoc	4[3]	2			
Direct indexed with X				x		
Read-Modify-Write ins.	INC DLoc,X	8[3]	2			
Other instructions	LDA DLoc,X	5[3]	2			
Direct indexed with Y	LDX DLoc,Y	5[3]	2	x		
Direct indirect	LDA (DLoc)	6[3]	2		x	
Direct indirect long	LDA [DLongLoc]	7[3]	2			x
Direct indirect indexed	LDA (DLoc),Y	6[1,3]	2	x		
Direct indirect indexed long	LDA [DLongLoc],Y	7[3]	2			x
Direct indexed indirect	LDA (DLoc,X)	7[3]	2	x		
Relative	BEQ Label	2[2]	2	x		
Relative long	BRL Label	3[2]	3			x
Stack	PHA	3-8	1-4	x		
Stack relative	LDA 3,S	5	2			x
Stack rel. indirect indexed	LDA (4,S),Y	8	2			x
Block move	MVP Source,Dest	7	3			x

[1]Add 1 cycle if adding index crosses a page boundary.
[2]Add 1 cycle if branch is taken.
[3]If direct register low (DL) does not equal zero, add 1 cycle.

Read-Modify-Write instructions operate by reading the contents of a memory location, changing it in some way, then returning the result to memory.

As the table shows, the INC (Increment) instruction is of the Read-

Modify-Write variety. INC adds one to the contents of an operand. If the operand is a memory location, INC must read the location's contents, add one to it, then write the result back to the original location. These extra tasks take time, and it's reflected in the higher cycle count. Among other Read-Modify-Write instructions are DEX (Decrement) and the shift and rotate instructions, which displace the contents of an operand to the left or right. These instructions, along with the rest of the 65816's instructions, will be covered in the next chapter.

Final Thoughts on Addressing Modes

If you are an old hand at Apple II assembly language, you probably breezed through the new 65816 modes. After all, except for the two stack relative modes (which hardly anyone would use) and the block move mode (which only applies to two instructions), the new modes simply provide for "long" operations — those involving inter-bank operations.

To assembly language newcomers, the addressing modes are a different story entirely. There are a lot of them, they often look alike (e.g., LDA Loc,X and LDA Loc,Y), and their names are sometimes baffling (e.g., direct indirect indexed long). In all, if you're just beginning in assembly language, your head may be spinning as you read this. That's natural, and you shouldn't worry about it.

In reality, you will probably use only these eight modes frequently:

Accumulator
Immediate
Implied
Absolute
Absolute indexed with X
Absolute indexed indirect
Relative
Stack

What's more, you will generally use a mode without knowing it. That's because in writing a program, you select the instructions that do what you want. Good programmers don't think in terms of choosing an addressing mode, but rather in terms of (A) which instruction and (B) which form of that instruction does what they want.

For example, to add one to the contents of the accumulator, or "increment" it, a programmer would use an *INC A* instruction, to which the

assembler applies the accumulator mode. Similarly, incrementing a memory location (say, Count) involves using an *INC Count* instruction, to which the assembler applies the absolute mode. Did the programmer consciously chose a mode? No. He or she simply selected an instruction, and left it up to the assembler to translate that instruction using the correct mode.

Just as my hypothetical programmer thinks in terms of instructions, so should you. Concentrate on finding an instruction that does the job, and let the assembler select the mode.

CHAPTER 4

65816 Instruction Set

In earlier chapters you became acquainted with LDA and some other simple instructions. In this chapter, I'll describe the 65816's entire instruction set. The 816 uses the same instructions as the earlier 6502 and 65C02, but it also has a few new ones. For the benefit of readers who have programmed on an Apple //e or some other 6502-based computer, I will note the differences.

In some books authors cover the instructions individually, discussing them in alphabetical order. This is suitable for reference manuals, but it tends to leave readers bored and bewildered after the fifth or sixth instruction. In this book, I group instructions by function, describing similar instructions together. For example, I group add instructions with subtract instructions, shifts with rotates, and so on. This should help you understand the instruction set and appreciate how individual instructions relate to each other, so you don't learn them as just a lot of disjointed entities.

Later, after running a few programs, you will only need to refer to this chapter occasionally, to look up details of specific instructions. Generally, you should be able to resolve most questions by referring to Appendix C, where the instructions are summarized alphabetically.

Instruction Types

As mentioned earlier, the 65816 has 91 different types of instructions. Table 4-1 shows their assembler mnemonics and tells what each stands for.

Alternate Mnemonics

The assembler also provides alternate mnemonics for certain instructions, as listed in Table 4-2. The most useful alternates are BLT (Branch if Less Than) and BGE (Branch if Greater Than or Equal) — branch instructions that test the result of a preceding compare operation. Certainly, people more often want to test whether an operand is "less than" or "greater than or equal to" another operand than whether the carry flag is 0 (clear) or 1 (set).

Table 4-1

Mnemonic	Meaning
ADC	Add to Accumulator with Carry
AND	AND Memory with Accumulator
ASL	Arithmetic Shift Left
BCC	Branch on Carry Clear (C = 0)
BCS	Branch on Carry Set (C = 1)
BEQ	Branch if Equal (Z = 1)
BIT	Bit Test
BMI	Branch if Minus (N = 1)
BNE	Branch if Not Equal (Z = 0)
BPL	Branch if Plus (N = 0)
BRA	Branch Always[1]
BRK	Force Break
BRL	Branch Always Long[2]
BVC	Branch on Overflow Clear (V = 0)
BVS	Branch on Overflow Set (V = 1)
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
COP	Coprocessor[2]
CPX	Compare Memory and X Register
CPY	Compare Memory and Y Register

Table 4-1 (cont.)

Mnemonic	Meaning
EOR	Exclusive-OR Memory with Accumulator
INC	Increment Memory or Accumulator
INX	Increment X Register
INY	Increment Y Register
JML	Jump Long[2]
JMP	Jump
JSL	Jump to Subroutine Long*UP*2
JSR	Jump to Subroutine
LDA	Load Accumulator
LDX	Load X Register
LDY	Load Y Register
LSR	Logical Shift Right
MVN	Block Move Negative (to a lower-numbered address)[2]
MVP	Block Move Positive (to a higher-numbered address)[2]
NOP	No Operation
ORA	OR Memory with Accumulator
PEA	Push Effective Address onto Stack[2]
PEI	Push Effective Indirect Address onto Stack[2]
PER	Push Effective Program Counter Relative Address onto Stack[2]
PHA	Push Accumulator onto Stack
PHB	Push Data Bank Register onto Stack[2]
PHD	Push Direct Register onto Stack[2]
PHK	Push Program Bank Register onto Stack[2]
PHP	Push Processor Status Register onto Stack
PHX	Push X Register onto Stack[1]
PHY	Push Y Register onto Stack[1]
PLA	Pull Accumulator from Stack
PLB	Pull Data Bank Register from Stack[2]
PLD	Pull Direct Register from Stack[2]
PLP	Pull Processor Status Register from Stack
PLX	Pull X Register from Stack[1]
PLY	Pull Y Register from Stack[1]
REP	Reset Processor Status Bits[2]
ROL	Rotate Left
ROR	Rotate Right
RTI	Return from Interrupt
RTL	Return from Subroutine Long[2]
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Bit
SEP	Set Processor Status Bits[2]

Table 4-1 (cont.)

Mnemonic	Meaning
STA	Store Accumulator
STP	Stop the Clock[2]
STX	Store X Register
STY	Store Y Register
STZ	Store Zero[1]
TAX	Transfer Accumulator to X Register
TAY	Transfer Accumulator to Y Register
TCD	Transfer C Accumulator to Direct Register[2]
TCS	Transfer C Accumulator to Stack Pointer[2]
TDC	Transfer Direct Register to C Accumulator[2]
TRB	Test and Reset Bit[1]
TSB	Test and Set Bit[1]
TSC	Transfer Stack Pointer to C Accumulator[2]
TSX	Transfer Stack Pointer to X Register
TXA	Transfer X Register to Accumulator
TXS	Transfer X Register to Stack Pointer
TXY	Transfer X Register to Y Register
TYA	Transfer Y Register to Accumulator
TYX	Transfer Y Register to X Register
WAI	Wait for Interrupt[2]
WDM	Reserved for Future Use (No operation)[2]
XBA	Exchange B and A Bytes of Accumulator[2]
XCE	Exchange Carry and Emulation Bits (Switch modes)[2]

[1]Available only on the 65C02 and 65816, not on the 6502.
[2]Available only on the 65816, not on the 6502 nor 65C02.

Table 4-2

Standard	Alias	Meaning
BCC	BLT	Branch if Less Than
BCS	BGE	Branch if Greater Than or Equal
CMP	CMA	Compare Memory and Accumulator
DEC A	DEA	Decrement Accumulator
INC A	INA	Increment Accumulator
TCD	TAD	Accumulator to Direct Register
TCS	TAS	Transfer C Accumulator to Stack Pointer
TDC	TDA	Transfer Direct Register to C Accumulator
TSC	TSA	Transfer Stack Pointer to C Accumulator
XBA	SWA	Swap Accumulator Bytes

Functional Groups

The instructions are divided into ten functional groups:

1. Data transfer instructions move information between registers, memory locations, and I/O devices.
2. Arithmetic instructions perform add and subtract operations on binary or binary-coded decimal (BCD) numbers.
3. Control transfer instructions can change the sequence in which a program executes; they include jumps and branches.
4. Subroutine instructions perform transfers to and from subroutines.
5. Stack instructions transfer data between registers and the stack, a data structure in memory.
6. Bit manipulation instructions perform logical operations on memory locations and registers.
7. Shift and rotate instructions displace the contents of a register or memory location.
8. The mode control instruction switches the processor between native and emulation mode.
9. Interrupt-related instructions include those that regulate requests for service from external devices.
10. Miscellaneous instructions are ones that don't fit in any other group.

Data Transfer Instructions

Data transfer instructions move information between registers, memory locations, and I/O devices. Table 4-3 summarizes these instructions in three groups: load and store, register transfer, and block move. Remember, *in emulation mode, the processor transfers bytes; in native mode, it can transfer bytes or words.* (Block moves always transfer blocks of bytes.)

Load and Store

The load and store instructions are the assembly language counterparts of PEEK and POKE in BASIC. A load operation reads or copies a memory value or immediate value into a register (either A, X, or Y).

Table 4-3

Mnemonic	Assembler Format	Flags N	V	M	X	D	I	Z	C
Load and Store									
LDA	LDA source	*	.	.	.	.	.	*	.
LDX	LDX source	*	.	.	.	.	.	*	.
LDY	LDY source	*	.	.	.	.	.	*	.
STA	STA destination	.	.	.	.	.	.	.	.
STX	STX destination	.	.	.	.	.	.	.	.
STY	STY destination	.	.	.	.	.	.	.	.
STZ	STZ destination	.	.	.	.	.	.	.	.
Register Transfer									
TAX	TAX	*	.	.	.	.	.	*	.
TAY	TAY	*	.	.	.	.	.	*	.
TCD	TCD	*	.	.	.	.	.	*	.
TCS	TCS	.	.	.	.	.	.	.	.
TDC	TDC	*	.	.	.	.	.	*	.
TSC	TSC	*	.	.	.	.	.	*	.
TSX	TSX	*	.	.	.	.	.	*	.
TXA	TXA	*	.	.	.	.	.	*	.
TXS	TXS	.	.	.	.	.	.	.	.
TXY	TXY	*	.	.	.	.	.	*	.
TYA	TYA	*	.	.	.	.	.	*	.
TYX	TYX	*	.	.	.	.	.	*	.
XBA	XBA	*	.	.	.	.	.	*	.
Block Move									
MVN	MVN ss,dd	.	.	.	.	.	.	.	.
MVP	MVP ss,dd	.	.	.	.	.	.	.	.

Notes: (1) * means changed and . means unchanged.
(2) Shaded instructions are new with the 65816.
(3) *ss* and *dd* are the bank numbers for the source and destination repectively.

Two flags in the processor status register provide information about the value that has been loaded. The negative (N) flag is 1 if the value is negative and 0 if it is zero or positive. (Of course, N is only meaningful if you're working with signed numbers; you don't care about it for unsigned numbers.) The zero (Z) flag indicates whether the loaded value is zero (Z = 1) or something else (Z = 0).

For example, the following instruction loads the contents of memory location ThisLoc into the accumulator:

```
LDA  ThisLoc
```

Again, N and Z reflect the sign of the loaded value and whether it is 0.

The store instructions copy the contents of the A, X, or Y register into a specified memory location. Unlike the load instructions, the stores do not affect the processor status register.

There is also STZ, which stores 0 in a location. The simplest way to store a value other than zero in memory is by using a load and store combination, such as:

```
LDA  #$FFFE
STA  ThatLoc
```

Since I'm discussing numbers in memory, it's worthwhile to mention that the 65816 stores 16-bit numbers in low-byte/high-byte order — that is, with the least-significant byte at the lower address. For example, when storing $ABCD at a location called NUM, it puts $CD at NUM and $AB at the next location, NUM + 1. Keep this storage scheme in mind when you display the contents of memory. Just remember "low data, low address; high data, high address."

Register Transfer

These instructions let you copy between two registers, in the combinations shown in Figure 4-1. Of them, only XBA (Exchange B and A Bytes of Accumulator) affects the contents of the source register.

Note that most of these transfers involve the accumulator (called *A* to be consistent with 6502 terminology or *C* to reflect its new 16-bit length in the 65816). That's because the accumulator is the only register on which the processor can perform arithmetic and logical operations. Hence, there are instructions that copy a value into the accumulator prior to an operation (e.g., TXA and TYA) and copy the result from the accumulator after it (TAX and TAY).

A note about the XBA instruction: The N and Z flags it returns reflect the state of the new A (low) byte.

Block Move

The block move instructions copy a block of bytes from one place in memory to another, working either forward (MVN) or backward (MVP) through the block.

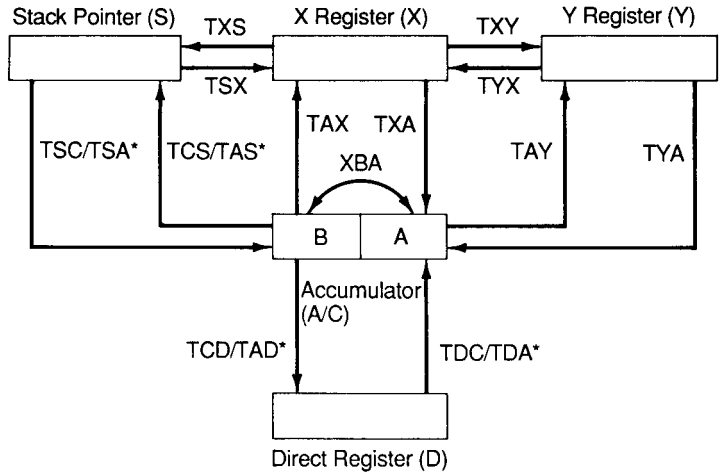


Figure 4-1

They have the general format:

```
MVN  srcbk,destbk
MVP  srcbk,destbk
```

where *srcbk* is the number of the bank that contains the original block (the source) and *destbk* is the number of the bank where the copy belongs (the destination). These numbers are the same if you're copying within a bank. Fortunately, you don't actually have to specify the bank numbers — simply enter the names of the blocks (e.g., OldBlock and NewBlock) and the assembler will calculate their bank numbers.

You must also supply the details of the operation in three registers. Specifically:

- The X register must contain the offset of the first byte to be copied.
- The Y register must contain the offset of the first copied byte.
- The accumulator must contain the number of bytes to be copied less 1.

Note that for MVN, X and Y must point to the beginning of the source and destination blocks, whereas for MVP, X and Y must point to the ends of those blocks.

If the source and destination blocks do not overlap, or if you're copying between banks, you can use either instruction. Make it easy on yourself. If the start addresses are easy to obtain, use MVN; if the end addresses are more convenient, use MVP. For example, to copy 100 bytes from location Here to location There, use:

```
LDX   #Here          ;Put source offset in X,
LDY   #There         ; destination offset in Y,
LDA   #99            ; and count-1 in A
MVN   Here,There     ;Move the block
```

Note the use of the immediate mode to obtain the block offsets.

If the blocks overlap, it's critical to choose the correct block move instruction. For example, if you want to move a block forward one byte position in memory (to the next higher numbered address), and start at the beginning of the block, the first byte to be moved will overwrite the second byte. Instead, you must start moving from the end of the block, using MVP.

Similarly, if you move a block backward one byte position (to the next lower-numbered address), and start at the end of the block, the first byte moved will overwrite the preceding byte. In this case, you must start moving from the beginning of the block, using MVN. In summary:

- To move a block backward (to a lower address), put the start addresses of the source and destination in X and Y, then execute an MVN instruction.
- To move a block forward (to a higher address), put the end addresses of the source and destination in X and Y, then execute an MVP instruction.

Of course, in either case A must contain the byte count minus 1.

Figure 4-2 illustrates how the block move instructions operate when the source and destination blocks overlap. Here, the arrows inside the blocks indicate the sequence in which bytes are copied — from beginning to end for MVN, from end to beginning for MVP.

With MVN, the processor increments the X and Y registers each time it copies a byte; with MVP, it decrements X and Y each time; for both, it decrements the count in A. Hence, at the end of a block move operation, X and Y point to the byte that lies just beyond the last byte copied and A contains −1 (hex FFFF). Moreover, the 65816 assumes that since you copied

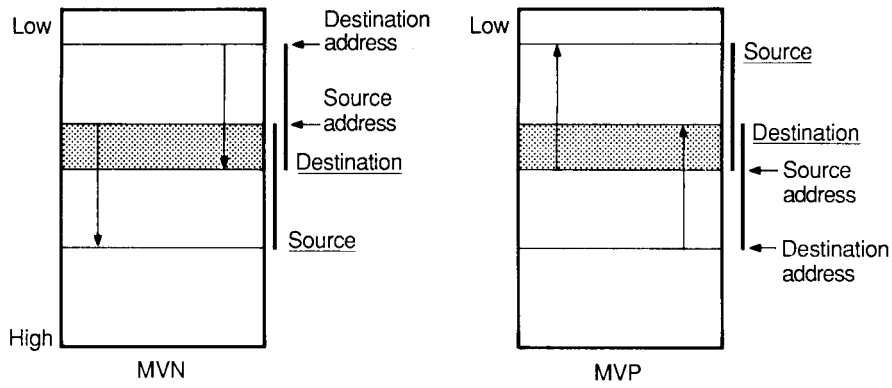


Figure 4-2

something to the destination bank, that's the bank you want to work with, so it makes the data bank register (DBR) point to that bank.

Because the block move instructions assume that X, Y, and A are 16-bit registers, you should only use them in the 65816's full native mode.

Arithmetic Instructions

The 65816 can operate in two different arithmetic modes, binary and decimal. When operating in binary mode (as it is when you switch the computer on), it treats operands as binary values; in decimal mode, it treats them as binary-coded decimal (BCD) numbers.

As Table 4-4 shows, there are instructions that add, subtract, increment, and decrement operands. There are also instructions that manipulate flags in the processor status register; these are used in various ways during arithmetic operations. Finally, there are compare instructions; these actually do a subtraction, but they report the results only in the status flags. Before discussing these instructions, I think it best to spend some time describing the formats of binary and decimal numbers.

Data Formats

As mentioned in chapter 0, binary numbers may be 8 or 16 bits long — depending on whether the 65816 is operating in emulation mode or native mode — and may be either unsigned or signed. In an *unsigned* number, all bits represent data. Therefore, unsigned numbers can range from 0 to 255 (8

Table 4-4

Mnemonic	Assembler Format	Flags N	V	M	X	D	I	Z	C
Flag Manipulation									
CLC	CLC	.	.	.	.	.	.	.	0
CLD	CLD	.	.	.	.	0	.	.	.
CLV	CLV	.	0	.	.	.	.	.	.
SEC	SEC	.	.	.	.	.	.	.	1
SED	SED	.	.	.	.	1	.	.	.
Add and Subtract									
ADC	ADC source	*	*	.	.	.	.	*	*
SBC	SBC source	*	*	.	.	.	.	*	*
Increment									
INC	INC destination	*	.	.	.	.	.	*	.
INX	INX	*	.	.	.	.	.	*	.
INY	INY	*	.	.	.	.	.	*	.
Decrement									
DEC	DEC destination	*	.	.	.	.	.	*	.
DEX	DEX	*	.	.	.	.	.	*	.
DEY	DEY	*	.	.	.	.	.	*	.
Compare									
CMP	CMP source	*	.	.	.	.	.	*	*
CPX	CPX source	*	.	.	.	.	.	*	*
CPY	CPY source	*	.	.	.	.	.	*	*

Note: * means changed and . means unchanged.

bits) or 65,535 (16 bits). In a *signed* number, the high-order bit (7 or 15) specifies the sign of the number; the rest hold data. Therefore, signed numbers can range from 127 to −128 (bits) or from 32,767 to −32,768 (16 bits).

The 65816 can operate on decimal numbers that have been stored as a series of "packed" bytes. By packed, I mean that each byte holds two *binary-coded decimal (BCD)* digits, with the most-significant digit in the upper 4 bits. Therefore, a BCD byte can hold values from 00 to 99, while a BCD word can hold values from 0000 to 9999.

Addition

Most microprocessors have two add instructions: one that simply adds two operands and another that includes a carry in the addition. The intent here is that one would use the "add without carry" form to add single-byte or single-word operands or to add the low-order bytes or words of multiprecision

operands. On the other hand, one would use the "add with carry" form to add higher-order bytes of multiprecision operands.

Like the 6502, the 65816 has only one add instruction: *ADC (Add to Accumulator with Carry)*. Here, "Carry" is the carry (C) flag of the processor status register. ADC adds a source operand (an immediate value or the contents of a memory lcoation) and the carry flag to the accumulator, and puts the result in the accumulator. In equation form, this is:

```
A = A + source + C
```

Since carry is always included, you must explicitly clear it to 0 before adding single-unit numbers or the low-order units of multiprecision numbers. The instruction that clears the carry flag is *CLC (Clear Carry Flag)*. For example, to add 15 to the accumulator, enter:

```
CLC           ;Clear the carry flag,
ADC  #15      ; then add 15
```

ADC affects four flags in the processor status register:

- The negative (N) flag is 1 if the result is negative (the high-order bit is 1); otherwise, N is 0.
- The overflow (V) flag is 1 if adding two positive or negative numbers produces a result that exceeds the two's-complement capacity of the accumulator, which changes the sign; otherwise, V is 0.
- The zero (Z) flag is 1 if the result is zero; otherwise, Z is 0.
- The carry (C) flag is 1 if the result cannot be contained in the accumulator; otherwise, C is 0.

N and V are only pertinent when you add signed numbers.

The 65816 has instructions that test flags and base an execution "decision" on the outcome. For example, a negative result (N = 1) may make it execute one set of instructions, while a zero or positive result (N = 0) makes it execute a different set. These decision-making instructions are discussed later in this chapter.

To add multiprecision numbers, clear the carry flag, add the low-order units, and store the result in memory. Then, to add the higher-order units, perform a series of LDA (load), ADC (add), and STA (store) instructions, one for each remaining unit. You can use the same procedure to add multiprecision decimal (BCD) numbers; simply precede the addition with an *SED (Set Decimal Mode)* instruction.

How the 65816 Subtracts

Like other general-purpose microprocessors, the 65816 has an internal addition unit, but no subtraction unit. Still, it *can* subtract numbers — by adding them! Strange as this may seem, the concept is "elementary," as Sherlock Holmes might say.

To see how to subtract numbers by adding them, consider how you subtract, say, 7 from 10. In elementary school, you learned to write this as:

```
10 - 7
```

However, later (in Algebra 101, perhaps) you learned that another way to write it is:

```
10 + (-7)
```

The first form — the straight subtraction — can be performed by a processor that has a subtraction unit. Since the 816 has no such unit, it subtracts in two steps. First, it changes the sign of, or *complements*, the second number (the subtrahend). Then it adds the complemented subtrahend to the minuend (the first number) to produce the result. Because the 816 works with base 2 (binary) numbers, the complement is a *two's*-complement. To obtain the two's-complement of a binary number, take its positive form and reverse each bit — change each 1 to 0 and each 0 to 1 — then add 1 to the result.

Applying this to the "10 − 7" example, the 8-bit binary representation of 10 and 7 are 00001010 and 00000111, respectively. Take the two's-complement of 7 as follows:

```
  1111 1000  (Reverse all bits)
+        1   (Add 1)
-----------
  1111 1001  (Two's complement of 7, or -7)
```

Now the subtraction operation becomes:

```
  0000 1010  (= 10)
+ 1111 1001  (= -7)
-----------
  0000 0011  (= 3)
```

Eureka! That's the right answer!

Subtraction

As in the case of addition, the 65816 has only one subtract instruction, *SBC (Subtract from Accumulator with Borrow)*, in which the carry (C) flag contributes the "borrow." SBC subtracts a source operand (an immediate value or the contents of a memory location) and the inverse, or *complement*, of the carry flag from the accumulator, and puts the result in the accumulator. In equation form, this is:

```
A = A - source + C
```

Since carry is always included, you must explicitly set it to 1 before subtracting single-unit numbers or the low-order units of multiprecision numbers. The instruction that sets the carry flag is *SEC (Set Carry Flag)*. For example, to subtract 15 from the accumulator, enter:

```
SEC            ;Set the carry flag
SBC  #15       ; then subtract 15
```

SBC affects four flags in the processor status register:

- The negative (N) flag is 1 if the result is negative (the high-order bit is 1); otherwise, N is 0.
- The overflow (V) flag is 1 if you subtract a positive number from a negative, or vice versa, and the result exceeds the two's-complement capacity of the accumulator, which changes the sign; otherwise, V is 0.
- The zero (Z) flag is 1 if the result is zero; otherwise, Z is 0.
- The carry (C) flag is 1 if the result is positive or zero; C is 0 if the result is negative, indicating a borrow.

N and V are only pertinent when you add signed numbers.

To subtract multiprecision numbers, set the carry flag, subtract the low-order units, and store the result in memory. Then, to subtract the higher-order units, perform a series of LDA (load), SBC (subtract), and STA (store) instructions, one for each remaining unit. You can use the same procedure to subtract multiprecision decimal (BCD) numbers; simply precede it with an *SED (Set Decimal Mode)* instruction.

Signed Arithmetic

I haven't yet mentioned how adding and subtracting differs between signed and unsigned numbers. That was not an oversight; I didn't mention it because you use the same instructions, ADC and SBC, regardless of whether the operands are signed or unsigned. However, for signed numbers, you must pay attention to the states of the negative (N) and overflow (V) flags.

The N flag simply reflects the sign of the result (positive if N = 0, negative if N = 1), but the V flag indicates whether the result in the accumulator is valid (V = 0) or invalid (V = 1). Recall that V is 1 if adding two numbers having the same sign or subtracting two numbers having different signs produces a result that exceeds the capacity of the accumulator.

Generally, your program should check for overflow after each signed add or subtract operation, and run some kind of error routine (display a message, perhaps) if V is 1. I'll describe instructions that test for overflow later in this chapter. Once set, the overflow flag stays that way until you either begin another ADC or SBC operation or execute a *CLV (Clear Overflow Flag)* instruction, which resets V to 0.

Increment and Decrement

The 65816 has a group of instructions that increment or decrement an operand — that is, add 1 to it or subtract 1 from it. The *INC* and *DEC* instructions require an operand, either *A* (for accumulator) or one of the memory addressing modes.

INX, *INY*, *DEX*, and *DEY* have no operand; the X or Y register is implied in the instruction. These are convenient for increasing or decreasing loop counters when you're doing a repetitive operation (e.g., adding a table of numbers in memory). The X and Y forms are particularly useful for increasing or decreasing the index register when you are accessing consecutive locations in memory.

Example 4-1 illustrates both of these uses; it is a short program that fills a 100-element table (ZTable) with zeros. Here, X serves as the element pointer and Y holds the count. The final instruction, BNE, is one I haven't described yet, but all it does is make the processor continue looping back to NextEel until Y has been decremented to 0. Note also that I have used *STZ (Store Zero)*, an instruction introduced with the 65C02, to store the zeros. With a 6502, this would require two instructions: an *LDA #0* preceding NextEl and a *STA ZTable,X* at NextEl.

The increment and decrement instructions are specialized adds and

Example 4-1

```
; Fill a 100-element table called ZTable with zeros.

          LDX  #0          ;Index points to first element
          LDY  #100        ;Operate on 100 elements
NextEl    STZ  ZTable,X    ;Store zero in next element
          INX               ; and increase the pointer
          DEY               ;Decrease the count
          BNE  NextEl      ;Loop until count is zero
```

subtracts that affect only the negative (N) and zero (Z) flags; they leave the overflow (V) and carry (C) flags alone. This is important because it lets you, for example, use a loop to add multiprecision numbers, yet preserve the carry between operations.

Compare

Most programs don't execute instructions in the order they are stored in memory. Instead, they usually include jumps, loops, and subroutine calls that make the 816 transfer to a different part of a program. The instructions that actually produce these transfers will be discussed in the next section, "Control Transfer Instructions." But I will now discuss the compare instructions, which help the control transfer instructions make their transfer/no-transfer "decisions."

The *CMP*, *CPX*, and *CPY* instructions do a "trial" subtraction. That is, they subtract a source operand (an immediate value or the contents of a memory location) from a destination operand (A, X, or Y), but *do not save the result*. That is, they don't alter the destination. Instead, they only set or clear the flags based on the result (see Table 4-5). Further, unlike SBC, the compares do not include the carry flag in the subtraction.

Table 4-5

Condition	N*	Z	C
A, X, or Y < Memory	1	0	0
A, X, or Y = Memory	0	1	1
A, X, or Y > Memory	0	0	1

*Meaningful only for operations on signed numbers.

Control Transfer Instructions

The control transfer instructions can make the 65816 transfer from one part of a program to another. All but the simplest programs include jumps and subroutine calls that alter the execution path the microprocessor takes. Subroutine instructions are discussed in the next section; this section covers jumps and branches. Table 4-6 divides the control transfer instructions into two groups: unconditional transfer and conditional transfer. Note that none of these instructions affect the flags.

Unconditional Transfer

There are two kinds of unconditional transfer instructions, jumps and branches. The *JMP* instruction is the assembly language equivalent of GOTO in BASIC; it makes the 65816 take its next instruction from some place other than the next consecutive memory location. JMP has the general form

```
JMP  target
```

Table 4-6

		Flags							
Mnemonic	**Assembler Format**	**N**	**V**	**M**	**X**	**D**	**I**	**Z**	**C**
Unconditional Transfer									
JMP	JMP target	.	.	.	.	.	.	.	.
JML	JML (aaaa)	.	.	.	.	.	.	.	.
BRA	BRA target	.	.	.	.	.	.	.	.
BRL	BRL long target	.	.	.	.	.	.	.	.
Conditional Transfer									
BCC	BCC target	.	.	.	.	.	.	.	.
BCS	BCS target	.	.	.	.	.	.	.	.
BEQ	BEQ target	.	.	.	.	.	.	.	.
BMI	BMI target	.	.	.	.	.	.	.	.
BNE	BNE target	.	.	.	.	.	.	.	.
BPL	BPL target	.	.	.	.	.	.	.	.
BVC	BVC target	.	.	.	.	.	.	.	.
BVS	BVS target	.	.	.	.	.	.	.	.

Notes: (1) . means unchanged.
(2) Shaded instructions are new with the 65816.

where *target* is a location (usually a label) in the active program bank. JMP is often used to bypass a group of instructions that are executed from some other part of the program. For example, you may use it in this context:

```
            . .
            . .
            LDA     DATA1
            CLC
            ADC     DATA2
            JMP     THERE
HERE        . .
            . .
THERE       . .
            . .
```

You can also jump to a location *indirectly*, by obtaining its two-byte address from a pointer in memory. For example,

```
            JMP     (MEMPTR)
```

jumps to the location whose address is in MEMPTR.

To transfer to an instruction in a different bank, you must use *JML (Jump Long)*. JML uses only absolute indirect address, but unlike JMP, obtains a 3-byte address — the program bank register (PBR) and the program counter (PC) — from a memory pointer. JMP is either a 3- or 4-byte instruction, depending on whether the target is in the same bank or a different one. JML is always a 3-byte instruction.

If the target address is no more than 127 bytes away, you can use a faster version of JMP: *BRA (Branch Always)*. Similarly, if the target is no further than 32,767 bytes, you can use *BRL (Branch Always Long)*.

Conditional Transfer

There are eight *branch* instructions that let the 65816 make an execution "decision" based on some prescribed condition, such as a register containing 0 or the carry (C) flag being set to 1. If the condition is satisfied, the 816 makes the transfer; otherwise, it continues to the next instruction. Table 4-7 summarizes the branch instructions and the flags they test.

As you can see from Table 4-7, the branch instructions base their transfer decisions on the contents of four flags in the processor status register: carry (C), zero (Z), negative (N), and overflow (V). In general:

Table 4-7

Instruction	Description	Branch if . . .
BCC/BLT	Branch on Carry Clear/if Less Than	Carry (C) = 0
BCS/BGE	Branch on Carry Set/if Greater Than or Equal to	Carry (C) = 1
BEQ	Branch if Equal	Zero (Z) = 1
BNE	Branch if Not Equal	Zero (Z) = 0
BMI	Branch if Minus	Negative (N) = 1
BPL	Branch if Plus	Negative (N) = 0
BVC	Branch on Overflow Clear	Overflow (V) = 0
BVS	Branch on Overflow Set	Overflow (V) = 1

- The carry flag reflects conditions where the result cannot be contained in the result register or memory location. It is affected indirectly by the arithmetic, shift, and compare instructions, and directly by SEC and CLC.
- The zero and negative flags are set by conditions in which the result is 0 or has the most-significant bit equal to 1. These flags can be affected by about half of the 65816 instructions.
- The overflow flag is affected by the arithmetic instructions (ADC and SBC) and by the BIT and CLV instructions.

The branch instructions use only program counter relative addressing. The relative displacement is a signed byte contained in the instruction, so the branch can span up to 127 bytes forward or 128 bytes backward from the instruction that follows the branch. This is not as restricting as you might think, because you can always combine a branch instruction with a JMP or JML to transfer anywhere in memory. For example, here is how a program might branch on the carry-set condition to an instruction (at CSET) that is beyond the normal +127/−128 branch range:

```
          BCC   CCLEAR    ;Go to CCLEAR on carry = 0
          JMP   CSET      ;Go to CSET on carry = 1
CCLEAR    . .
          . .
```

The conditional branch instructions each occupy 2 bytes in memory: opcode followed by the relative displacement. The 65816 takes 2 cycles to execute a branch if the condition is not met and 3 cycles if the condition is

met (4 cycles if the 816 crosses a page boundary). Because of the time difference, construct your programs so that whenever possible, the expected case executes if the branch is *not taken*.

Here are some examples of branch instructions:

1. The following sequence branches to TOOBIG if the addition produces a carry.

```
ADC    MEMLOC
BCS    TOOBIG
```

2. This sequence branches to TOOSML if the subtraction produces a borrow.

```
SBC    MEMLOC
BCC    TOOSML
```

3. These instruction will branch to ZERO if A and MEMLOC hold the same value.

```
CMP    MEMLOC
BEQ    ZERO
```

4. The following sequence will loop to LOOP until the X register has been decremented to 0. This sort of sequence is common in programs that use the X or Y register as a counter.

```
LOOP    . .
        . .
        DEX
        BNE    LOOP
```

Using Branch Instructions with Compares

You can precede conditional branch instructions with any instruction that alters the flags, but they are often preceded with a compare instruction (CMP, CPX, or CPY). Table 4-5 earlier in this chapter shows how the compares affect the flags for various register/memory combinations.

Now, with the variety of conditional branch instructions, it is worthwhile to look at a more practical table — one that shows which conditional branch to use for all possible register/data combinations. Table 4-8 is the one

Table 4-8

To branch if . . .	Follow compare with: For unsigned numbers		For signed numbers	
Register is less than data	BLT	THERE	BMI	THERE
Register is equal to data	BEQ	THERE	BEQ	THERE
Register is greater than data	BEQ	HERE	BEQ	HERE
	BCS	THERE	BPL	THERE
Register is less than or equal to data	BLT	THERE	BMI	THERE
	BEQ	THERE	BEQ	THERE
Register is greater than or equal to data	BGE	THERE	BPL	THERE

you need. In this table, THERE represents the label of the instruction the 65816 executes if the branch test succeeds, while HERE is the label of the instruction the 816 executes if the test fails.

To illustrate a typical application for a compare/branch combination, Example 4-2 shows a program that arranges two unsigned numbers in memory in increasing order, with the larger number in the higher-numbered location.

You can also combine a compare instruction with two branch instructions to test the "less than," "equal to," and "greater than" cases separately. Example 4-3 shows a sequence that executes any of three groups of instructions, based on whether the value in A is below, equal to, or above 10. Since the branch instructions do not affect the status flags, BNE GT10 can base its branch decision on the same flag that the BGE GTEQ10 based *its* decision on.

Example 4-2

```
; This sequence arranges two unsigned 16-bit numbers in memory
; in order of magnitude, with the larger value at the higher
; address.

          LDA  MEMLOC+2  ;Get second value
          CMP  MEMLOC    ;Compare the numbers
          BGE  DONE      ;Done if second is greater than
                         ; or equal to the first
          LDX  MEMLOC    ;Otherwise, swap them
          STA  MEMLOC
          STX  MEMLOC+2
DONE      ..
          ..
```

Example 4-3

```
; This sequence executes one of three different groups of
; instructions, based on whether the unsigned number in A
; is below, equal to, or above 10.

          CMP  #10        ;Compare accumulator to 10
          BGE  GTEQ10
          ..              ;Accumulator is less than 10
          ..
          BRA  DONE
GTEQ10    BNE  GT10
          ..              ;Accumulator is equal to 10
          ..
          BRA  DONE
GT10      ..              ;Accumulator is greater than 10
          ..
DONE      ..
          ..
```

Subroutine Instructions

Sometimes you want to perform a specific operation (say, display a message) at more than one place in your program. One way to do that is to duplicate the entire set of instructions everywhere you need it. However, duplicating instructions is both frustrating and time-consuming. It also makes programs longer than they would be if you could avoid this duplication. As a matter of fact, you *can* eliminate needless duplication by defining a recurring instruction sequence as a *subroutine*. Table 4-9 lists the 65816's subroutine instructions.

As in BASIC, a subroutine is a set of instructions that you write just once, but which you can execute as needed at any place in a program. The

Table 4-9

Mnemonic	Assembler Format	Flags N	V	M	X	D	I	Z	C
JSR	**JSR target**	.	.	.	.	.	.	.	.
RTS	**RTS**	.	.	.	.	.	.	.	.
JSL	**JSL long-target**	.	.	.	.	.	.	.	.
RTL	**RTL**	.	.	.	.	.	.	.	.

Notes: (1) means unchanged.
(2) Shaded instructions are new with the 65816.

process of transferring control from the main program to a subroutine is defined as *calling*. When you *call* a subroutine, the 65816 executes the instructions in it, then returns to the place from which the call was made.

This invites two questions: "How does one call a subroutine?" and "How does the 816 return to the proper place in the program?" The answers involve two subroutine-related instructions: JSR and RTS.

Instructions that execute subroutines must perform three tasks:

1. They must somehow save the contents of the program counter (PC). Once the subroutine has been executed, the 816 uses this address to return to the calling point. Hence, we refer to the saved address as the *return address*.
2. They must make the microprocessor begin executing the subroutine.
3. Upon completion of the subroutine, they must use the stored value of the PC to return to the main program and continue executing at that point.

These three tasks are performed by two instructions: *JSR (Jump to Subroutine)* and *RTS (Return from Subroutine)*. Essentially, JSR and RTS are the assembly language equivalents of GOSUB and RETURN in BASIC.

JSR performs tasks 1 and 2; it stores the return address and begins executing. Specifically, it stores the return address (the address of the instruction that follows it) on the stack. JSR has the format

```
JSR  target
```

where *target* is the name of the subroutine being called.

RTS undoes the work of JSR; that is, it makes the 816 leave the subroutine and return to the calling program by pulling the return address off the stack. RTS must always be the last subroutine instruction the processor executes. (This doesn't mean that RTS must be the last instruction in the subroutine — although it usually is — just the last one the 816 *executes*.)

For example, to call a subroutine named MYSUB, your program might execute this sequence (offsets are also listed):

```
04F0        JSR MYSUB  ;Call the subroutine
04F3 NEXT   TXA        ;Return here after the subroutine
            . .
```

```
                . .
0600 MYSUB LDA #6       ;First instruction of the
                         subroutine
                . .
                . .
061E       RTS          ;Return to calling program
```

When the 65816 executes JSR MYSUB, it pushes the offset of NEXT onto the stack, then loads the offset of MYSUB into the program counter (PC) — and that's where the 816 begins executing. Eventually, when the 816 encounters the RTS instruction, it pulls the return address off the stack and puts it into the PC. This makes it resume at the instruction labeled NEXT, a TXA in this case. Figure 4-3 shows the stack, the stack pointer (S), and the program counter (PC) before and after the JSR, and after the RTS.

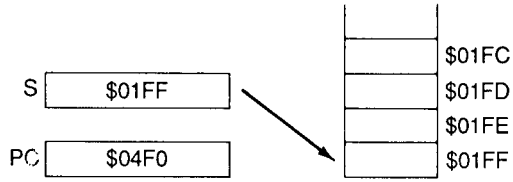
The 65816 also provides two instructions, *JSL (Jump to Subroutine Long)* and *RTL (Return from Subroutine Long)*, for subroutines that are in a different bank than the JSL instruction. JSL pushes 3 bytes (program bank register and program counter) onto the stack, as opposed to 2 bytes (PC) for JSR.

A subroutine may itself call other subroutines. For example, a subroutine that reads a user's menu response from the keyboard may well decode the response character and then call one of several other subroutines based on the result. Calling one subroutine from within another is referred to as *nesting*. Figure 4-4 shows the JSR and RTS instructions for a program in which SUBR1 calls SUBR2 (i.e., SUBR2 is nested within SUBR1).

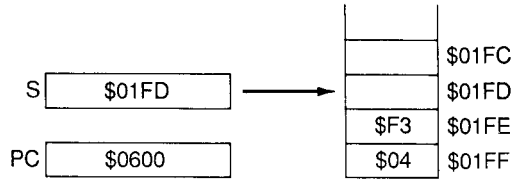
Programmers usually describe nesting in terms of *levels*. An application like the one in Figure 4-4, where the nesting extends only to the JSR to SUBR2 (SUBR2 does not call another subroutine) is said to have one level of nesting. However, SUBR2 might well have called a third subroutine — say, SUBR3 — with SUBR3 calling SUBR4, and so on.

Stack Instructions

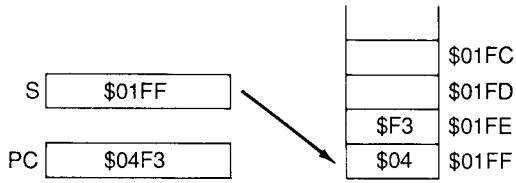
As mentioned in the preceding section, the stack holds return addresses while the 65816 is executing subroutines. The JSR (or JSL) instruction puts the address onto the stack and an RTS (or RTL) instruction retrieves it at the conclusion of the subroutine. In both cases, the processor uses the stack *automatically*; you don't have to tell it to do so.



(A) Before executing JSR MYSUB.



(B) After executing JSR MYSUB.



(C) After executing RTS.

Figure 4-3

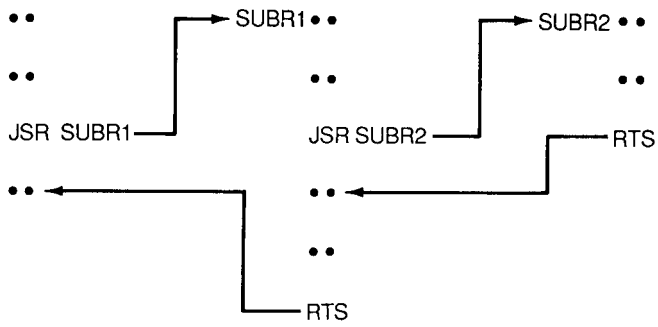


Figure 4-4

The stack is also a convenient place to deposit data from your program temporarily. For example, you might want to save the contents of the accumulator while you put it to some other use.

Overview of the Stack

Earlier, I mentioned that the stack is a "last-in/first-out" type of data structure in page 1 of memory. That is, the last item to be entered (or *pushed*) onto the stack is the first item to be extracted (or *pulled*) from it. Conversely, the first item pushed onto the stack is the last item to be pulled off it. In short, *stack data is retrieved in the opposite order from which it was stored*, just like plates in a kitchen cabinet.

Stack information is accessed by a dedicated stack address register called the *stack pointer (S)*, which always points to the next available location on the stack. The 65816 decrements the stack pointer whenever a byte is pushed onto the stack, and increments it whenever a byte is pulled from the stack. Hence, the stack "builds" downward in memory, in the direction of location 0. When you switch the computer on, the stack pointer points to location $01FF, the end of page 1.

Among the data transfer instructions discussed earlier in this chapter, I described several that are used to copy between the stack pointer and another register: TCS, TSB, TSC, TSX, and TXS. You wouldn't normally use these instructions, however, because for most applications, you simply let the 65816 take care of regulating the stack pointer.

Table 4-10 shows the more important stack instructions, the ones you use to push information onto the stack and pull information off it. These instructions are divided into two groups: *push and pull registers* and *push immediate data or effective address*.

Push and Pull Registers

As Table 4-10 shows, the 65816 provides instructions that push and pull the contents of the accumulator (*PHA* and *PLA*), data bank register (*PHB* and *PLB*), direct register (*PHD* and *PLD*), processor status register (*PHP* and *PLP*), and the X (*PHX* and *PLX*) and Y (*PHY* and *PLY*) registers. It also has a *PHK* instruction that pushes the contents of the program bank register.

Aside from the fact that they operate on different registers, the push instructions work identically. In each case, the 816 pushes the contents of the register onto the stack at the location to which the stack pointer is pointing. Then it decrements the stack pointer by 1 (in emulation mode) or 2 (in native mode), making it point to the next lower location. Push instructions

Table 4-10

Mnemonic	Assembler Format	Flags N	V	M	X	D	I	Z	C
Push and Pull Registers									
PHA	PHA	.	.	.	.	.	.	.	.
PHB	PHB	.	.	.	.	.	.	.	.
PHD	PHD	.	.	.	.	.	.	.	.
PHK	PHK	.	.	.	.	.	.	.	.
PHP	PHP	.	.	.	.	.	.	.	.
PHX	PHX	.	.	.	.	.	.	.	.
PHY	PHY	.	.	.	.	.	.	.	.
PLA	PLA	*	.	.	.	.	.	*	.
PLB	PLB	*	.	.	.	.	.	*	.
PLD	PLD	*	.	.	.	.	.	*	.
PLP	PLP	*	*	*	*	*	*	*	*
PLX	PLX	*	.	.	.	.	.	*	.
PLY	PLY	*	.	.	.	.	.	*	.
Push Immediate Data or Effective Address									
PEA	PEA Loc	.	.	.	.	.	.	.	.
PEI	PEI (DLoc)	.	.	.	.	.	.	.	.
PER	PER #Num	.	.	.	.	.	.	.	.

Notes: (1) * means changed and . means unchanged.
(2) Shaded instructions are new with the 65816.

do not alter the contents of the source register, nor do they affect the status flags.

Figure 4-5 shows how a PHA instruction in 16-bit native mode affects the stack. Here, the stack pointer (S) is pointing to the top of the stack, location $01FF, and the accumulator contains $0304. After PHA executes, the stack pointer has been decremented to $01FD and the contents of the accumulator have been stored on the stack.

If the PHA was followed by a PLA instruction, the 65816 would do the reverse; it would increment S, copy the contents of location $01FE into the low byte of the accumulator, increment S again, and copy the contents of location $01FF into the high byte of the accumulator. (Note that while $0304 is still stored in the same location in memory, it is no longer considered as being "on the stack." In fact, *nothing* is on the stack, because the stack pointer is pointing at the top of the stack, location $01FF.)

With seven push and six pull instructions at your disposal, you can save

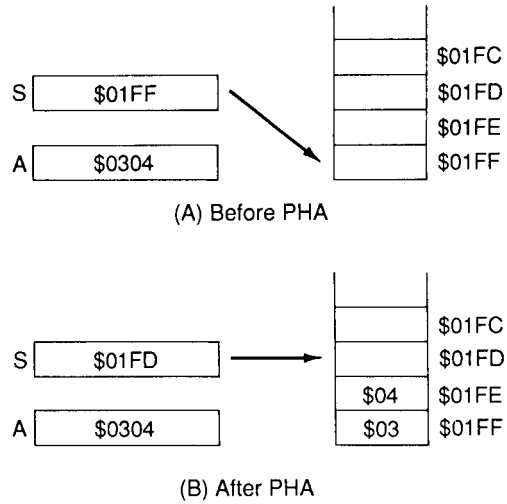


Figure 4-5

all the registers if you want to. That is, you can save the entire "context" in which your program is working. Example 4-4 shows the instructions you would use to save and restore all the general registers and the status. These are the kinds of instructions you should put at the beginning and end of any subroutine that is to save the caller's context.

Note that I have entered the pull instructions in the opposite order from the push instructions. This reflects the "last-in/first-out" nature of the stack. Having mentioned that, I must state the cardinal rule for working with the stack:

Example 4-4

```
; Use these instructions to preserve the accumulator, X, Y,
; and status.

          PHP             ;Save processor status register,
          PHA             ; accumulator,
          PHX             ; X register,
          PHY             ; and Y register
          ..
          ..
          PLY             ;Restore the Y register,
          PLX             ; X register,
          PLA             ; accumulator,
          PLP             ; and status
```

You must always pull data in the reverse order you pushed it.

Note also that I pushed the processor status register first. This lets me pull it last, to cancel the effects of the other pull instructions on the N and Z flags (see Table 4-10).

While I'm giving rules, there's another one you should remember:

Every push must have a corresponding pull later in the program.

By keeping these two rules in mind, you shouldn't have any problems with stack operations.

Later in this book, you will encounter programs that start with the instruction combination:

```
PHK  ;Copy the contents of the program bank register
PLB  ; into the data bank register
```

This is necessary because the system loader makes the PBR point to the bank that contains your program, but it does nothing to the DBR. The PHK/PLB combination tells the microprocessor that data within the program resides in the same bank as the program itself. You might assume that the loader would do this for you, but it does not; at least not in the current version of the software.

Push Immediate Data or Effective Address

The designers of the 65816 included an instruction called *PEA*, which is short for *Push Effective Address onto Stack*. However, it should really be described as "Push Immediate Word onto Stack" (and named, perhaps, PIW), because that's what it does; it pushes a 16-bit constant onto the stack. For example,

```
PEA  #15
```

pushes decimal 15 onto the stack. Of course, there's no "Pull Immediate Data" instruction, so you have to pull it into a register.

Another misnamed instruction is *PEI*, short for *Push Effective Indirect Address onto Stack*. PEI should be described as "Push Direct Page Word onto Stack" (and named, say, PDW), because that's what it does. It takes a

byte operand from the instruction and uses it as an offset into the direct page. PEI forms the effective address by adding the offset to the direct register (D), then pushes the word at that location onto the stack. For example,

```
PEI  #4
```

pushes the word that starts at byte 4 of the direct page onto the stack.

The final instruction of this group, *PER (Push Effective Program Counter Relative Address onto Stack*, takes a 16-bit offset from the instruction and adds it to the value of the program counter, then pushes the result onto the stack. PER does *not* change the program counter or the program bank register.

Bit Manipulation Instructions

These instructions manipulate bit patterns in the accumulator or a memory location. Table 4-11 divides them into three groups: logical, bit testing, and processor status bits.

Logical

Logical instructions are so named because they operate according to the rules of formal logic, rather than those of mathematics. For example, the rule of logic that states, "If A is true and B is true, then C is true" has a 65816 counterpart in the *AND (AND Memory with Accumulator)* instruction. AND applies this rule to corresponding bits in two operands; one operand is the accumulator, the other can be an immediate value or the contents of a memory location.

Specifically, for each bit position where both operands are 1 (true), AND sets the bit in the accumulator to 1. Conversely, for any bit position where the two operands have any other combination — both are 0 or one is 0 and the other is 1 — AND sets the accumulator bit to 0 (see Table 4-12).

In essence, AND masks out (zeros) certain bits so you can do some kind of processing on the remaining bits. Note that *any bit ANDed with 0 becomes 0, and any bit ANDed with 1 retains its original value*. For example, this instruction zeros the high byte of the accumulator:

```
AND #$00FF  (or simply AND #$FF)
```

Table 4-11

Mnemonic	Assembler Format		Flags V	M	X	D	I	Z	C
Logical									
AND	AND source	*	.	.	.	.	.	*	.
EOR	EOR source	*	.	.	.	.	.	*	.
ORA	ORA source	*	.	.	.	.	.	*	.
Bit Testing									
BIT	BIT source-byte	M_7	M_6	.	.	.	.	*	.
BIT	BIT source-word	M_{15}	M_{14}	.	.	.	.	*	.
BIT	BIT #Num	.	.	.	.	.	.	*	.
TRB	TRB destination	.	.	.	.	.	.	*	.
TSB	TSB destination	.	.	.	.	.	.	*	.
Processor Status Bits									
REP	REP #dd	*	*	*	*	*	*	*	*
SEP	SEP #dd	*	*	*	*	*	*	*	*

Notes: (1) * means changed and . means unchanged.
(2) Shaded instructions are new with the 65816.

Table 4-12

Source	Destination	Result AND	ORA	EOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

As in this example, you generally use hexadecimal numbering for an immediate operand. Since the 816 can operate on either bytes or words, you normally deal with either 2 or 4 hexadecimal digits. To help you construct the correct "mask" value for a logical operation, Table 4-13 shows the hexadecimal representation of a 1 in 16 different bit positions. For example, to operate on bit 2, the correct mask value is 4; to operate on bits 2 and 3, the mask is $C ($4 + $8); and so on.

AND is also useful for converting digits that an operator types at the keyboard into binary numbers that your program can process. Typed digits

Table 4-13

Bit Number	Hex. Value	Bit Number	Hex. Value
0	0001	8	0100
1	0002	9	0200
2	0004	10	0400
3	0008	11	0800
4	0010	12	1000
5	0020	13	2000
6	0040	14	4000
7	0080	15	8000

— or characters, for that matter — enter the computer in a form called *ASCII* (short for American Standard Code for Information Interchange). As the ASCII summary in Appendix B shows, the digits 0 through 9 have the ASCII codes $30 through $39, respectively. To convert the code to its binary value, simply chop off the 3. With the code in the low byte of the accumulator, this requires:

```
AND #$000F  (or simply AND #$F)
```

The *ORA (OR Memory with Accumulator)* instruction produces a 1 in the accumulator for each bit position in which either or both operands contain 1 (see Table 4-12). ORA is generally used to set specific bits to 1. For example,

```
ORA  #$C000
```

sets the accumulator's two high-order bits (14 and 15) to 1 and leaves all other bits unchanged.

The *EOR (Exclusive-OR Memory with Accumulator)* can be used to determine which bits differ between two operands or to reverse the setting of selected bits. EOR puts a 1 in the accumulator for every bit position in which the operands differ — that is, where one operand has 0 and the other has 1. If both operands' bits are the same (0 or 1), EOR clears the accumulator bit to 0. For example,

```
EOR  #$C000
```

reverses the state of the accumulator's two high-order bits (14 and 15) and leaves all other bits as they are.

As Table 4-12 shows clearly, EOR is the same as ORA, except that two 1's produce a 0 rather than a 1. EOR excludes the two 1's from the combinations that produce a 1 result — that's why it's called *Exclusive*-OR.

Bit Testing

The *BIT (Bit Test)* instruction ANDs a memory or immediate operand with the value in the accumulator, but affects only the flags, not the accumulator. With an immediate operand, BIT reports the result of the AND operation only in the zero (Z) flag. With a memory operand, BIT affects three status flags, as follows:

- The negative (N) flag receives the initial (un-ANDed) value of bit 7 (byte) or bit 15 (word) of the memory location.
- The overflow (V) flag receives the initial value of bit 6 (byte) or bit 14 (word) of the memory location.
- The zero (Z) flag is 1 if the AND operation produces a zero result; otherwise, Z is 0.

Note that only the Z flag reflects the result of the AND operation; N and V simply report the state of the operand's two high-order bits. If you follow BIT with a BNE (Branch if Not Equal) instruction, the 816 makes the branch if there are any corresponding 1 bits in both operands.

TRB (Test and Reset Bits) and *TSB (Test and Set Bits)* let you set or reset selected bits in a memory location. (In reality, "Test" is a misnomer. These instructions don't test the operand; they set or reset bits unconditionally.) TRB and TSB are useful for manipulating locations that act as indicators, where individual bits are meaningful.

The TRB instruction ANDs the memory operand with the complement of the accumulator (that is, with the accumulator's inverse). Thus, each 1 bit in A resets the corresponding memory bit to 0 and each 0 bit leaves its memory counterpart as it is. For example, if the accumulator contains 9,

```
TRB  MEMFLAG
```

clears bits 0 and 3 of MEMFLAG.

The TSB instruction ORs the memory operand with the accumulator. Thus, each 1 bit in the accumulator sets the corresponding memory bit to 1 and each 0 bit leaves its memory counterpart as it is. For example, if the accumulator contains $40,

```
TSB  MEMFLAG
```

sets bit 6 of MEMFLAG.

Processor Status Bits

Under "Arithmetic Instructions," I described instructions that clear the overflow flag (CLV) and set or clear the carry and decimal mode bits (CLC, CLD, SEC, and SED). There are also instructions that manipulate the IRQ disable bit, and I'll discuss them later. Now I will describe two instructions that let you manipulate the entire processor status register, rather than just individual bits. They are similar to TRB and TSB, except they operate on the status register rather than on a memory location.

The *REP (Reset Processor Status Bits)* instruction ANDs the status register with the complement (i.e., the inverse) of an immediate byte. Thus, each 1 bit in the byte resets the corresponding status bit to 0 and each 0 bit leaves its status counterpart as it is. For example,

```
REP  #%110000
```

clears the M and X bits, thereby making the accumulator and index registers 16 bits long. (You should follow this particular REP with the directives LONGA ON and LONGI ON, to make the assembler assemble for the long registers.)

The *SEP (Set Processor Status Bits)* instruction ORs the status register with an immediate byte. Thus, each 1 bit in the byte sets the corresponding status bit to 1 and each 0 bit leaves its status counterpart as it is. For example,

```
SEP  #%110000
```

sets the M and X bits, thereby making the accumulator and index registers 8 bits long. (Follow this SEP with LONGA OFF and LONGI OFF, to assemble for the short registers.) You will see these M- and X-changing instructions later, when the mode control instruction, XCE, is discussed.

You can also use SEP to set the overflow (V) flag — say, to use it as a 1-bit indicator in a program. The 65816 has no other instruction that sets V directly. The form you would use is:

```
SEP  #%1000000
```

Shift and Rotate Instructions

The 65816 has four instructions that displace the contents of the accumulator or a memory location one bit position to the left or right (see Table 4-14). Two of these instructions *shift* the operand, the other two *rotate* it.

For all four instructions, the carry (C) flag acts as a "9th bit" or "17th bit" extension of the operand. That is, C receives the value of the bit that has been displaced out of the operand. A right shift or rotate puts the value of bit 0 into C; a left shift or rotate puts the value of bit 7 (byte) or bit 15 (word) into it. Figure 4-6 shows how these instructions operate.

Shifts

ASL (Arithmetic Shift Left) shifts the operand one bit position to the left and puts a 0 in the vacated bit 0 position. Similarly, *LSR (Logical Shift Right)* shifts the operand one bit position to the right and puts a 0 in the vacated high-order bit position (bit 7 in a byte, bit 15 in a word). Besides carry, ASL and LSR update the negative and zero flags.

To see how the shift instructions work, assume the processor is in 8-bit mode and the accumulator contains $B4, or binary 10110100. Here is how the shift instructions affect A and C:

After ASL A: (A) = 01101000 C = 1
After LSR A: (A) = 01011010 C= 0

The shift instructions can also serve as multiply-by-2 or divide-by-2

Table 4.14

		Flags							
Mnemonic	**Assembler Format**	**N**	**V**	**M**	**X**	**D**	**I**	**Z**	**C**
Shifts									
ASL	ASL destination	*	.	.	.	.	.	*	*
LSR	LSR destination	0	.	.	.	.	.	*	*
Rotates									
ROL	ROL destination	*	.	.	.	.	.	*	*
ROR	ROR destination	*	.	.	.	.	.	*	*

Note: * means changed and . means unchanged.

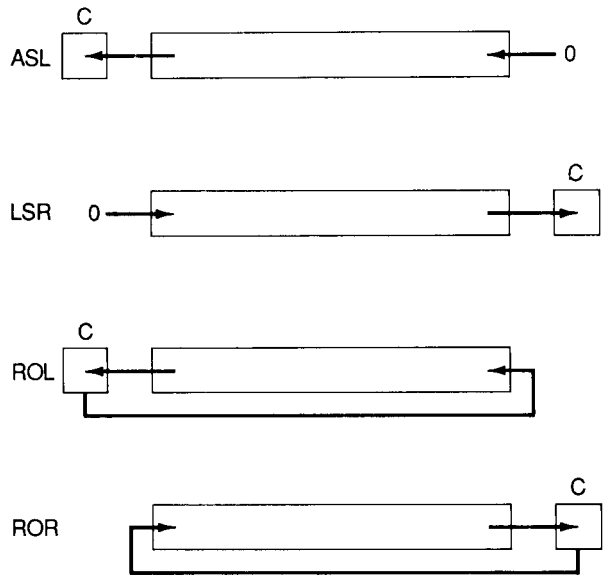


Figure 4-6

instructions, because *shifting an operand one bit position to the left doubles its value and shifting it one bit position to the right halves its value*. Of course, this assumes that the bit you shift out is a 0 rather than a 1.

Rotates

Like the shifts, the rotate instructions, *ROL (Rotate Left)* and *ROR (Rotate Right)*, enter displaced bits into the carry (C) flag. However, the rotates first enter the prerotate value of C into the vacated bit position at the opposite end of the operand. Like the shifts, the rotates also report the result in the negative and zero flags.

To see how the rotate instructions work, consider the operand that was used for the shift examples, $B4 or binary 10110100, and assume that C is 1 initially. Here is how the two rotate instructions affect the accumulator and C:

After ROL A: (A) = 01101001 C = 1
After ROR A: (A) = 11011010 C = 0

Shifting Signed Numbers

Despite the fact that ASL stands for "Arithmetic Shift Left," the four shift and rotate instructions perform what are known as *logical* shifts; that is, they treat the operand strictly as a bit pattern, without regard to sign. Consequently, if a signed number is shifted right, the sign bit is displaced one bit position to the right (like every other bit) and its value is replaced with a 0. If a signed number is shifted left, its sign bit is displaced into the carry flag and its value is replaced by the value of bit 6 (byte) or bit 14 (word). Clearly, your program must deal with this problem somehow.

What *should* happen for a true arithmetic shift left is that the processor should perform the regular logical shift left (which is what ASL does), but set a flag if the sign bit changes. Overflow (V) is the most appropriate flag here, so in your program, enter an instruction sequence similar to the one shown in Example 4-5 (assuming you're shifting the accumulator).

An arithmetic shift right should preserve the sign of the operand by replicating the sign in the vacated high-bit position. Example 4-6 shows an instruction sequence that does this.

Example 4-5

```
; This routine shifts the accumulator left one bit position
; and sets the overflow (V) flag if the sign bit has changed.

           CLV              ;Clear overflow flag to start
           ASL  A           ;Shift the accumulator left
           BCC  POSITIVE
           BPL  OVERFLOW    ;C = 1. If N = 0, set overflow
           BNI  DONE
POSITIVE   BPL  OVERFLOW    ;C = 0. If N = 1, set overflow
OVERFLOW   SEP  #$40        ;Set the overflow flag
DONE       BVS  ERROR       ;Print an error message if V is 1
```

Example 4-6

```
; This routine shifts the accumulator right one bit position
; and preserves the sign.

           CMP  #0          ;Read sign of operand
           BPL  POSITIVE
           LSR  A           ;Operand is negative. Shift it,
           AND  #$8000      ; then set the sign bit
           BRA  DONE
POSITIVE   LSR  A           ;Operand is positive. Shift it
DONE       ..
           ..
```

Mode Control Instruction

XCE (Exchange Carry and Emulation Bits) is used to switch the 65816 between native mode and emulation mode. XCE takes no operand, but it uses the state of the carry flag to determine which mode to switch to. Specifically, C = 1 makes it switch to emulation mode, because E is 1 after the exchange; C = 0 makes it switch to native mode, because E is 0 after the exchange. Hence, switching to emulation mode requires

```
SEC
XCE
```

while switching to native mode requires:

```
CLC
XCE
```

XCE affects only the carry flag. And *because* it affects only C, switching to native mode causes the M and X bits to retain whatever settings bits 4 and 5 had in emulation mode. Recall that in emulation mode, bit 4 is the Break (B) bit and bit 5 is an unused bit that's always 1. Therefore, the CLC/XCE combination makes the accumulator 8 bits long (because bit 5 is 1) and the index registers either 8 or 16 bits long, depending on whether B was 1 or 0. Clearly, you can't leave all this to chance; you must set the M and X bits for the register lengths you want.

To switch to full native mode — i.e., with all registers 16 bits long — you must follow the XCE with a REP (Reset Processor Status Bits) instruction that clears M and X to 0. You must also tell the assembler to assume 16-bit memory and registers, using LONGA ON and LONGI ON directives. Therefore, the statements that switch the 65816 to full native mode are:

```
CLC                 ;Switch to native mode
XCE
REP     #%110000    ;Make registers 16 bits long
LONGA   ON          ; and notify the assembler
LONGI   ON
```

When switching to emulation mode, you needn't set the register length, because registers are always 8 bits long. However, you must still tell the assembler what length you're using. The statements that switch the 65816 to emulation mode are:

```
SEC             ;Switch to emulation mode
XCE
LONGA   OFF     ;Tell the assembler that registers are
LONGI   OFF     ; 8 bits long
```

Interrupt-Related Instructions

Like a subroutine call, an interrupt from an external device makes the 65816 save return information on the stack, then transfer to an instruction sequence elsewhere in memory. However, a subroutine call makes the 816 execute a subroutine, while an interrupt makes it execute an *interrupt service routine*.

While the operands for the subroutine call instructions, JSR and JSL, can only be an absolute, absolute indexed indirect, or absolute long address, an interrupt always makes the 816 obtain the address of the service routine indirectly. That is, the 816 obtains the address from an *interrupt vector*, a 2-byte location at the end of bank 0.

The 65816 provides for 14 interrupt vectors, of which it uses 10 (see Table 4-15). Note that some interrupts are activated by signal lines on the microprocessor chip, while others are activated by the COP and BRK instructions.

There is yet another difference between subroutine calls and interrupts: a subroutine call saves only a return address on the stack, while an interrupt

Table 4-15

Location	Interrupt	Mode
00FFE4,5	COP instruction	Native
00FFE6,7	BRK instruction	Native
00FFE8,9	ABORT line	Native
00FFEA,B	NMI line	Native
00FFEC,D	-	-
00FFEE,F	IRQ line	Native
00FFF0,1	-	-
00FFF2,3	-	-
00FFF4,5	COP instruction	Emulation
00FFF6,7	-	-
00FFF8,9	ABORT line	Emulation
00FFFA,B	NMI line	Emulation
00FFFC,D	RESET line	Emulation and native
00FFFE,F	BRK installation and IRQ line	Emulation

saves the status flags as well. Specifically, when the 816 performs an interrupt, it does the following:

1. In native mode, pushes the program bank register (PBR) onto the stack.
2. Pushes the program counter's high byte, then its low byte, onto the stack.
3. Pushes the processor status register (P) onto the stack.
4. Sets the decimal mode (D) bit to 0, to specify binary mode.
5. Sets the IRQ disable (I) bit to 1, to lock out other interrupts while this one is being serviced.
6. Reads the contents of the interrupt vector into the program counter.

In summary, the PBR (in native mode), PC, and P registers are on the stack; D is 0 and I is 1, and the PC is pointing to the first instruction in the interrupt. That's where the 816 begins executing.

The 65816 has six interrupt-related instructions. Table 4-16 divides them into four groups: interrupt control, return from interrupt, software interrupts, and wait for interrupt.

Interrupt Control

These two instructions determine whether the 65816 accepts interrupt requests from external devices. *CLI (Clear Interrupt Disable Bit)* sets the IRQ disable bit (I) to 0, which lets the 816 respond to interrupt requests. *SEI (Set Interrupt Disable Bit)* sets the I bit to 1, which makes the 816 ignore requests. You generally disable interrupts when you're doing some time-critical or high-priority task that cannot be interrupted.

Return from Interrupt

RTI (Return from Interrupt) is to interrupts what RTS is to subroutines. That is, it undoes the work of the original operation and makes the 816 return to the main program. For this reason, RTI must be the last instruction the 816 executes in an interrupt service routine.

In emulation mode, RTI pulls the processor status register and the program counter off the stack. In native mode, it pulls P, PC, and the program bank register.

Table 4-16

Mnemonic	Assembler Format	Flags N	V	M	X	D	I	Z	C
Interrupt Control									
CLI	CLI	.	.	.	.	.	0	.	.
SEI	SEI	.	.	.	.	.	1	.	.
Return from Interrupt									
RTI	RTI	*	*	*	*	*	*	*	*
Software Interrupts									
BRK	BRK or BRK dd	.	.	.	.	0	1	.	.
COP	COP	.	.	.	.	0	1	.	.
Wait for Interrupt									
WAI	WAI	.	.	.	.	.	.	.	.

Notes: (1) * means changed and . means unchanged.
(2) Shaded instructions are new eith the 65816.

Software Interrupts

BRK (Force Break) makes the 65816 act as it does when it responds to an external interrupt, except BRK also sets the program bank register to 0. As Table 4-15 shows, BRK has two interrupt vectors, one for each mode. In native mode, the 816 loads the contents of locations $00FFE6 and $00FFE7 into the PC's low and high bytes, respectively. In emulation mode, it loads the contents of $00FFFE and $00FFFF into the PC. In either case, executing a BRK instruction sends the Apple IIGS into its Monitor.

Although BRK is a 1-byte instruction, it makes the 816 increment the program counter by 2, thereby allowing you to insert a 1-byte identifier that indicates what condition caused the break; that is, BRK has two formats: BRK and BRK nn.

COP (Coprocessor) is similar to BRK, except that the 816 loads the PC from locations $00FFE4 and 5 (native mode) or $00FFF4 and 5 (emulation mode). Unlike BRK, COP is a true 2-byte instruction; the identifier operand is required. If you omit it, the assembler generates a "Missing Operand" error.

Wait for Interrupt

The final interrupt instruction, *WAI (Wait for Interrupt)*, does what its name says: it puts the processor into an idle state in which it halts and waits for an

interrupt or a hardware reset. (I suppose we all enter an idle state occasionally.) Making the 816 wait with WAI rather than with an endless loop has two advantages: (1) the idle state reduces power consumption and (2) it results in the quickest response to an interrupt.

Miscellaneous Instructions

Table 4-17 shows two instructions that don't fit in any other category. *NOP (No Operation)* is the simplest, because it does nothing whatsoever. That is, it affects no flags, registers (other than the program counter), or memory locations.

Surprisingly, NOP has a variety of uses. For example, it is convenient when you're developing programs. Suppose you have completed a portion of the program that includes a jump or branch to as yet unwritten code; NOP makes a handy target for the jump. You can also use NOP's opcode ($EA) to "patch" an object code file when you want to delete an instruction without reassembling the program.

The final instruction is (appropriately) *STP (Stop the Clock)*. STP stops the processor and its clock, to reduce power consumption. The 816 does not restart until it receives a hardware reset.

There is also a noninstruction, the reserved mnemonic *WDM*. WDM will become an instruction when Western Design Center introduces its 32-bit microprocessor, the 65C832. Rumor has it that WDM stands for William D. Mensch, Jr., the 65816's designer!

Table 4-17

		Flags							
Mnemonic	**Assembler Format**	**N**	**V**	**M**	**X**	**D**	**I**	**Z**	**C**
NOP	**NOP**	.	.	.	.	.	.	.	.
STP	**STP**	.	.	.	.	.	.	.	.

Notes: (1) . means changed.
(2) Shaded instruction is new with the 65816.

CHAPTER 5

Macros

A macro is a subroutinelike "miniprogram" that you can insert in a source program by mentioning its name. This chapter tells how to create macros and use them in programs. Developing a macro can be either a simple or complex task, depending on what you want the macro to do. Still, it's quite possible that you will *never* find it necessary to develop macros of your own, because the *Apple IIGS Programmer's Workshop* (*APW*) disk already contains a wide variety of them.

Most of these macros are actually "tool calls" — calls to subroutines in the Apple IIGS *Toolbox* (described in the next chapter). However, the *APW* disk also has a variety of macros that can be handy for doing general-purpose programming jobs. This chapter summarizes the most useful ones and describes how to use them in your programs.

Introduction to Macros

As just mentioned, a macro is a sequence of assembler statements (instructions and directives) that may appear several times in a program. Like subroutines, macros have names. Once you have defined a macro, you can enter its name in a source program anywhere you would normally enter the instruction sequence.

Macros Vs. Subroutines

Although macros and subroutines both provide a shorthand reference to a frequently used instruction sequence, they are not the same. The statements in a subroutine appear once in a program, and the processor transfers to them (or *calls* the subroutine) as needed. By contrast, the statements in a macro may occur many times within a program; the assembler replaces each mention of a macro name with the statements that name represents. (In computer terminology, the assembler "expands" the macro.) Therefore, when you execute the program, the processor executes the macro instructions "inline"; it does not transfer elsewhere in memory, as it does with a subroutine. Hence, *a macro name is a user-defined assembler directive*; it issues commands to the assembler, rather than to the microprocessor.

Macros have two advantages over subroutines:

1. Macros are *dynamic*. You can easily change the way a macro operates (not merely what it operates on) each time, by changing its input parameters. By contrast, you can only vary the data that gets passed to a subroutine, making subroutines much more inflexible.

2. Macros make for faster-executing programs, because the processor is not delayed by call (JSR) and return (RTS) instructions, and the stack operations they employ, as it is with subroutines.

Nothing comes for free, however. Since a macro gets expanded every time it is used, it tends to make machine-language programs longer by filling memory with repeated instruction sequences. This is a drawback that subroutines do not have.

Macros Speed Up Programming

Like subroutines, macros can speed up your programming and debugging work, as well as any program updating you might do in the future. They speed up programming in that you create a macro just once, then use it wherever you want it in a program. Instead of entering a long sequence of instructions, you enter only the macro name that represents the sequence.

Macros can speed up debugging because you create and debug each macro individually. Once a macro is working properly, you never need to worry about whether *that* portion of your program is correct. You can concentrate on finding errors elsewhere.

Moreover, programs that include macros are generally easier to read and understand. Consequently, they are also easier to update and change.

Change the macro definition and the assembler automatically uses the new version everywhere it previously used the old one.

To see how macros can ease your workload, consider the instructions you need to display a character on the screen. (These instructions are for earlier Apple IIs. I will discuss their equivalents for the IIGS later.) Displaying a character involves loading it into the accumulator, then calling the Monitor's COUT (Character Output) subroutine. To display a *D*, for example, requires:

```
LDA   #'D'       ;Select D for display
JSR   ($36)      ;Display it by calling COUT
```

Suppose you have a program that displays different letters from time to time. What does this involve on your part? It involves entering (and remembering) those same two instructions each time. Although there are only two instructions here, it's annoying to have to reenter them every time you need them. However, if you have defined the sequence as a macro called Cout, you can enter one of the following instead:

```
Cout D    ;Display D
Cout E    ;Display E
Cout y    ;Display y
```

Contents of Macros

Every macro definition has four parts:

1. A *MACRO* directive in the mnemonic (op code) field. This marks the beginning of a macro definition.
2. The *macro definition statement*, which has the following general form:

   [&LAB] *macro-name* [&parm1[,&parm2[, . . .]]]

   Note that in the definition only the macro name (in the mnemonic field) is required. The label specifier, &LAB is always optional. However, you should include it in every macro definition, to allow programs to jump or branch to that line.

   The operand field lists any input parameters for the macro. These are the parameters you can change each time you call the macro. Each parameter name must begin with an ampersand (&)

symbol. Parameter names are separated with commas. For example, the macro definition statement for a macro that adds two values and stores the result in memory might look like this:

```
&LAB ADD &TERM1,&TERM2,&SUM
```

3. The *body* of the macro; the sequence of statements (instructions and directives) that define what the macro does.
4. An *MEND* directive in the mnemonic field, to mark the end of the macro definition.

For example, the following is a simple macro that adds two word-size values:

```
        MACRO
&LAB    ADDW    &TERM1,&TERM2,&SUM
        LONGA   ON
&LAB    CLC
        LDA     &TERM1
        ADC     &TERM2
        STA     &SUM
        MEND
```

The assembler doesn't care whether you specify memory locations or immediate values for the operands (you can't use an immediate value for the sum, of course). As long as the final form is legal, the assembler makes the substitutions without complaining.

For example, at one place in the program you could add two memory locations by entering:

```
ADDW  PRICE,TAX,COST
```

This makes the assembler insert the following instructions in the program:

```
CLC
LDA  PRICE
ADC  TAX
STA  COST
```

Somewhere else, you could add an immediate value to a memory location by entering:

```
LB1  ADDW  MEMLOC,#4,MEMLOC
```

This time, the assembler would insert

```
LB1   CLC
      LDA   MEMLOC
      ADC   #4
      STA   MEMLOC
```

Note that this macro call is labeled LB1. Because the macro definition has a symbolic label, &LAB, on the CLC instruction, the assembler places the label you specify on the CLC instruction when it expands the macro.

These examples demonstrate how much easier it is to pass parameters to macros than to subroutines. With a macro, you enter the parameters on the same line as the macro's name; with a subroutine, you must put them in registers or memory locations.

Macro Directives

Table 5-1 summarizes the macro directives provided by the *Apple IIGS Programmer's Workshop*. They are divided into five groups: macro language, library, symbolic parameter, branching, and listing. (You can use the branching directives in source programs as well as macro definitions, but they are most useful in macros.)

Macro Language Directives

The MACRO and MEND directives have already been discussed; they simply mark the beginning and end of a macro definition. The *MEXIT* directive makes the assembler stop expanding the macro early. In effect, MEXIT does the same thing as MEND, except it ends the expansion somewhere within the macro definition, rather than at the end. You need MEXIT if your expansion can take several different paths, depending on the value of a variable. Path "decision" are controlled by the AIF conditional branch directive, which is described under "Branching Directives" later in this chapter.

Table 5-1

Directive	Function
	Macro Language
MACRO	Format: MACRO MACRO marks the beginning of a macro definition.
MEND	Format: MEND MEND marks the end of a macro definition.
MEXIT	Format: MEXIT MEXIT terminates a macro expansion, usually as the result of a branching directive.
	Library
MCOPY	Format: MCOPY *filename* MCOPY enters the specified *filename* in the list of available macro libraries. Once a file is in this list, the source program can use any macro in it. Up to four macro libraries can be active at any given time.
MDROP	Format: MDROP *filename* MDROP removes the specified *filename* from the list of available macro libraries.
MLOAD	Format: MLOAD *filename* MLOAD enters the specified *filename* in the list of available macro libraries, if it is not already there.
	Symbolic Parameter
LCLA	Format: LCLA *sparm* LCLA (Local Arithmetic) declares an arithmetic type symbolic parameter local to the current macro.
LCLB	Format: LCLB *sparm* LCLB (Local Boolean) declares a boolean type symbolic parameter local to the current macro.
LCLC	Format: LCLC *sparm* LCLC (Local Character) declares a character type symbolic parameter local to the current macro.
GBLA	Format: GBLA *sparm* GBLA (Global Arithmetic) declares an arithmetic type symbolic parameter global for the entire subroutine.
GBLB	Format: GBLB *sparm* GBLB (Global Boolean) declares a boolean type symbolic parameter global for the entire subroutine.
GBLC	Format: GBLC *sparm* GBLC (Global Character) declares a character type symbolic parameter global for the entire subroutine.

Table 5-1 (cont.)

Directive	Function
	Symbolic Parameter (cont.)
SETA	Format: *sparm* SETA *aexp* SETA (Set Arithmetic) resolves the arithmetic expression in the operand field to a four-byte signed hexadecimal number and assigns it to the symbolic parameter in the label field.
SETB	Format: *sparm* SETB *bexp* SETB (Set Boolean) evaluates the boolean expression in the operand field as either true or false, and assigns either 1 or 0 (respectively) to the symbolic parameter in the label field.
SETC	Format: *sparm* SETC *cexp* SETC (Set Character) evaluates the expression in the operand field as a character string and assigns it to the symbolic parameter in the label field.
AMID	Format: *sparm* AMID *string,start-pos,#-chars* AMID extracts a substring from a specified *string* and assigns it to the symbolic parameter in the label field. The substring begins at the character position numbered *start-pos* (the first character in the string is at position 1) and is *#-chars* characters long. Note that AMID does the same thing as the MID$ function in BASIC.
ASEARCH	Format: *sparm* ASEARCH *target$,search$,start-pos* ASEARCH searches the string *target$*, starting at character position *start-pos*, for the first occurrence of the *search$* substring. If the substring is found, its starting position is assigned to the symbolic parameter in the label field; otherwise, the parameter is set to zero.
	Branching
AGO	Format: AGO *ssym* AGO makes processing continue with the statement that follows the specified sequence symbol (see "Branching Directives").
AIF	Format: AIF *bexp,ssym* AIF evaluates the Boolean expression *bexp*. If the expression is true, processing continues with the statement that follows the specified sequence symbol (see "Branching Directives"); if false, processing continues with the statement that follows the AIF directive.
	Listing
GEN	Format: GEN ON/OFF GEN ON causes all lines generated by macro expansions to be included on the assembler's output listing. GEN OFF lists only macro definitions on the output listing.
TRACE	Format: TRACE ON/OFF The assembly normally omits conditional assembly directives from its output listing. TRACE ON makes it list these lines.

Library Directives

The *MCOPY* directive enters the name of a macro library file into a list of available macro libraries. This makes the macros in the library available to the source program. For example,

```
MCOPY NEW.MACROS
```

activates the macro library file called NEW.MACROS.

Up to four macro library files can be active at any given time. Thus, your source program may include up to four MCOPY directives. A more detailed discussion of macro libraries is upcoming, in the "Creating Macro Libraries" section.

The *MDROP* directive removes a specified file from the list of available macro libraries. You only need MDROP if you are juggling more than four libraries.

MLOAD is similar to MCOPY, except MLOAD enters a file name into the list of available macro libraries only if the name is not already in the list.

Symbolic Parameter Directives

These directives operate on symbolic parameters — variables within a macro definition.

The first three directives — *LCLA*, *LCLB*, and *LCLC* — tell the assembler that a symbolic parameter is internal, or "local," to a particular macro expansion; it is unknown throughout the rest of the program. The directives *GBLA*, *GBLB*, and *GBLC* tell the assembler that a symbolic parameter is "global"; its name can be referred to anywhere within the program.

The *SETA*, *SETB*, and *SETC* directives assign the value of an expression to a symbolic parameter. Thus, these directives do for symbolic parameters what the EQU (Equate) directive does for regular symbols.

The *AMID* directive extracts a substring from a string and assigns the substring to a symbolic parameter. (Think of AMID as the assembler counterpart of BASIC's MID$ function.) AMID takes three operands: the string, the character position where the substring begins (position 1 is the first character), and the length of the substring. For example,

```
&SUBS  AMID  'THE STRING',5,3
```

assigns the substring STR to &SUBS.

Finally, *ASEARCH* searches a string for a specified substring. If the substring is found, the symbolic parameter receives the position of its first character; otherwise, the parameter is set to 0.

Branching Directives

The assembler provides two branching directives, one conditional (*AIF*) and the other unconditional (*AGO*). AIF is the assembler's counterpart of the microprocessor's conditional branch instructions (BEQ, BMI, and so on), while AGO is the assembler's BRA. Just as the microprocessor's branch instructions can make the microprocessor continue executing at an instruction label elsewhere in memory, the branching directives can make the assembler continue processing at a *sequence symbol* elsewhere in a macro definition. A sequence symbol is a label that begins with a period (.) and is on a line by itself.

AIF has the general form

```
AIF bexp,ssym
```

where *bexp* is a boolean (logical) expression and *ssym* is the target sequence symbol. If *bexp* is true, the assembler continues processing at *ssym*; otherwise, if *bexp* is false, the assembler continues processing with the statement that follows AIF.

AGO makes the assembler continue processing a specified sequence symbol unconditionally. It's generally used to skip past a block of statements to which a preceding "true" AIF branches.

To see how AIF and AGO might be used, consider the ADDW addition macro described earlier. ADDW takes three parameters, the two terms to be added and the sum. Its macro definition line is:

```
&LAB ADDW &TERM1,&TERM2,&SUM
```

Now suppose we want to make the &SUM term optional. In this case, the sum is stored in &SUM if the user supplies it; otherwise, if he or she omits &SUM, the sum is stored in &TERM1. To make this happen, the macro definition would contain the following kinds of statements:

```
         . .
         . .
.C
         AIF    C:&SUM>0,.D    ;Is &SUM an empty string?
```

```
        LCLC   &SUM             ; Yes. Declare it local
&SUM    SETC   &TERM1           ; and set it to &TERM1
        AGO    .E               ; Skip .D and continue
                                ; at .E
.D
        . .                     ;Process these instruc-
                                ;tions if the user
        . .                     ; entered an &SUM term
.E
        . .                     ;Do the addition
        . .
        . .
```

Listing Directives

The assembler normally shows only macro call statements in its output listing; it does not "expand" macros to show their contents. You can, however, make it list expansions by entering a *GEN ON* directive. A subsequent *GEN OFF* will turn off the expansion listing.

Similarly, the assembler normally omits conditional assembly directives from the output listing. You can make it include them by entering a *TRACE ON* directive.

Creating Macro Libraries

As mentioned earlier, a macro library is a text file that contains one or more macro definitions. To use the macros in a library, your source program must make the library active by reading it with an MCOPY directive.

There are two ways to create a macro library:

1. If all the macros are new, you can enter their definitions using the editor, then save the file with the name you want to use in your MCOPY directive.

   To keep things simple, you should give the library the same name as the program that will use it, and end the name with *.MACROS*; for example, SORT.MACROS is a reasonable name for a file that contains macros used by a program called SORT.

2. To use macros that are contained in one or more existing library files, you can use the *MACGEN* utility contained on the *Programmer's Workshop* disk.

Here's a description of method 2. The MACGEN (Macro Library Generator) utility scans an assembler source file for macro names. To start it, enter a command of the form:

```
MACGEN [+C/-C] source-file out-file macro-lib1
       [macro-lib2. . .]
```

MACGEN's parameters are as follows:

- *+C* (the default) removes all excess blanks from macro definitions; specify *-C* if you have used GEN ON or TRACE ON in your source file.
- *source-file* is the name of your source file.
- *out-file* is the name of the macro library file you want to generate — the file name that appears in the program's MCOPY directive.
- *macro-lib1*, *macro-lib2*, and so on, are the files to be searched for the macro names that appear in *source-file*. Note that the filenames are separated by spaces.

This command makes MACGEN scan *source-file*, and any files named in its COPY and APPEND directives, for macro names mentioned in the program. It then opens a temporary file called SYSMAC on the active disk and reads the macro definitions from the *macro-lib* files into it. When MACGEN has resolved all macros, it changes the name of SYSMAC to *out-file* — the filename in your program's MCOPY directive. For example,

```
MACGEN MYPROG.SRC MYPROG.MACROS FILE1.MACROS
FILE2.MACROS
```

searches FILE1.MACROS, then FILE2.MACROS, for macro definitions whose names are mentioned in MYPROG.SRC, and copies those definitions into a new file called MYPROG.MACROS.

You can also use MACGEN to update a macro library file if you add new macro calls to your program. Simply specify the file that contains the new macro definitions as *macro-lib1* and the existing library file as *macro-lib2* and . For example,

```
MACGEN MYPROG.SRC MYPROG.MACROS FILE3.MACROS
MYPROG.MACROS
```

scans MYPROG.SRC and updates MYPROG.MACROS by adding macro definitions contained in FILE3.MACROS.

You may be tempted to bypass MACGEN, and MCOPY the relevant libraries into your source file directly. There are two reasons why you should resist that temptation:

1. The assembler reads library files *very* slowly, so copying large libraries or multiple libraries takes much longer than copying a smaller file that MACGEN has tailored to your source program.
2. Some macros call other macros. MACGEN will seek out every needed macro definition and put it in your output file. You may overlook some of these internally called macros, and wind up with assembly errors.

Macros on the *Programmer's Workshop* Disk

The *Apple IIGS Programmer's Workshop* (*APW*) disk has a subdirectory called LIBRARIES/AINCLUDE that contains macro files for each tool set in the IIGS Toolbox (described in Chapter 6). However, it also has two files that are unrelated to the Toolbox. M16.PRODOS contains macros that let your programs perform ProDOS 16 commands, while M16.UTILITY provides general-purpose macros for performing both 8- and 16-bit operations.

ProDOS 16 Macros

The ProDOS 16 macros, listed in Table 5-2, are macro versions of the function calls in ProDOS's Machine Language Interface (MLI). Using them saves you from having to remember the numeric "opcode" for each call (shown in italics in the table); each macro supplies it automatically.

Note that each macro name is preceded by an underscore character (_). The *Programmer's Workshop* uses the underscore prefix to identify macros that employ system calls of any kind.

Utility Macros

The macros in file /APW/LIBRARIES/AINCLUDE/M16.UTILITY perform a variety of fundamental operations that are useful for developing

assembly language programs. As Table 5-3 shows, these macros fall into nine groups: push and pull, load and store, add and subtract, define, move, shift, mode, write, and check error. Except for *native*, all utility macros assume the processor is operating in full (16-bit) native mode.

These macros are generally self-explanatory, so I don't provide a formal description. However, if a macro requires one or more operands, I give examples of its various forms. The examples should illustrate clearly what the macro does.

Pay special attention to the *pushword* and *pushlong* macros. Most calls to tools in the Apple IIGS Toolbox require input values or pointers to those values to be on the stack. Hence, you will usually precede a tool call with one or more pushword (for 2-byte inputs) or pushlong (for 4-byte inputs) calls.

The *str* (string) macro is also useful for defining messages to be displayed on the screen.

Push and Pull Macros

push1 [opr] — Push 1 byte onto the stack

```
pushl   Loc          ;Push byte at Loc
pushl   Loc,x        ;Push byte at Loc,x
pushl   #n           ;Push constant n
pushl                ;Push byte from A
```

pushword [opr] — Push 2 bytes onto the stack

```
push2   Loc          ;Push bytes at Loc
push2   Loc,x        ;Push bytes at Loc,x
push2   #n           ;Push constant n
push2                ;Push bytes from A
```

push3 opr[,reg] — Push 3 bytes onto the stack

```
push3   Loc          ;Push bytes at Loc
push3   Loc,x        ;Push bytes at Loc,x
push3   #n           ;Push constant n
```

pushlong addr[,offset] — Push 4 bytes onto the stack

```
pushlong   Loc       ;Push bytes at Loc
```

Table 5-2

Format	Description
__ALLOCINTERRUPT DCB	Install an interrupt handler *$40*
__CHANGEPATH DCB	Change a file's pathname *$04*
__CLEARBACKUPBIT DCB	Clear the backup bit in the file's access byte *$0B*
__CLOSE DCB	End access to file *$CC*
__CREATE DCB	Create file or directory *$C0*
__DEALLOCINTERRUPT DCB	Remove an interrupt handler *$41*
__DESTROY DCB	Delete file or directory *$C1*
__FORMAT DCB	Format a disk *$24*
__FLUSH DCB	Empty file's I/O buffer *$CD*
__GETBOOTVOL DCB	Get name of volume from which PRODOS file was last executed *$28*
__GETDEVNUM DCB	Get device number *$20*
__GETEOF DCB	Get size of file in bytes *$D1*
__GETFILEINFO DCB	Get file information *$C4*
__GETLASTDEV DCB	Get number of the last device accessed *$21*
__GETLEVEL DCB	Get system file level *$1B*
__GETMARK DCB	Get current position in file *$CF*
__GETVERSION DCB	Get ProDOS 16 version number *$2A*
__GETPATHNAME DCB	Get the current application's pathname *$27*
__GETPREFIX DCB	Get current path name prefix *$C7*
__NEWLINE DCB	Enable new line read mode *$C9*
__OPEN DCB	Prepare file for access *$C8*
__QUIT DCB	Terminate the current application *$29*
__READ DCB	Read bytes from file *$CA*
__READBLOCK DCB	Read 512 bytes from file *$80*
__SETEOF DCB	Set size of file *$D0*
__SETFILEINFO DCB	Set file information *$C3*
__SETLEVEL DCB	Set system file level *$1A*
__SETMARK DCB	Set new position in file *$CE*
__SETPREFIX DCB	Set pathname prefix *$C6*
__VOLUME DCB	Get the disk's name, its size in blocks, the number of free (unallocated) blocks, and the file system identification number *$08*
__WRITE DCB	Write bytes to file *$CB*
__WRITEBLOCK DCB	Write 512 bytes to file *$81*

```
pushlong  #Loc                ;Push address of Loc
pushlong  Loc,x               ;Push bytes at Loc,x
pushlong  #n                  ;Push constant n
pushlong  [zeropg],offset     ;Push using indirect
                              ; addressing
```

pushxy Push 4 bytes onto the stack from X and Y

Table 5-3

Format	Description
	Push and Pull Macros
push1 [opr]	Push 1 byte onto the stack
pushword [opr]	Push 2 bytes onto the stack
push3 addr[,reg]	Push 3 bytes onto the stack
pushlong addr[,offset]	Push 4 bytes onto the stack
pushxy	Push 4 bytes onto the stack from X and Y
pushay	Push 4 bytes onto the stack from A and Y
pull1 [opr]	Pull 1 byte from the stack
pullword [opr]	Pull 2 bytes from the stack
pull3 addr	Pull 3 bytes from the stack
pulllong [addr1[,addr2]]	Pull 4 bytes from the stack
pullxy [addr]	Pull 4 bytes from the stack using X and Y
pullay	Pull 4 bytes from the stack into A and Y
pullx [addr]	Pull 2 bytes from the stack using X
	Load and Store Macros
lday addr[,offset]	Load A and Y (4 bytes)
stay addr[,offset]	Store A and Y (4 bytes)
zero bytes,addr	Zero a block
	Add and Subtract Macros
add [term1],term2[,result]	Add 2-byte integers
add4 [term1],term2[,result]	Add 4-byte integers
sub [term1],term2[,result]	Subtract 2-byte integers
sub4 [term1],term2[,result]	Subtract 4-byte integers
	Define Macros
str 'string'	Define string
dp pointer	Define pointer
	Move Macros
move1 from,to[,to2]	Move 1 byte
moveword from,to[,to2]	Move 2 bytes
move3 from,to,[to2]	Move 3 bytes
movelong from,to[,to2]	Move 4 bytes
	Shift Macros
asl4 addr[,count]	Left-shift 4 bytes
lsr4 addr[,count]	Right-shift 4 bytes
	Mode Macros
native [long/short]	Turn on native mode
emulation	Turn on emulation mode
long	Set memory, A, X, and Y to 16 bits
longm	Set memory and A register to 16 bits

Table 5-3 (cont.)

Format	Description
	Mode Macros (cont.)
longx	Set X and Y registers to 16 bits
short	Set memory and registers to 8 bits
shortm	Set memory and A register to 8 bits
shortx	Set X and Y registers to 8 bits
	Write Macros
writech [opr]	Write a character
writestr [opr]	Write a string
writeln [opr]	Write a line (string + CR)
	Check Error Macro
Check__Error error__subr	Call error subroutine if carry is 1

pushay Push 4 bytes onto the stack from A and Y

pull1 [opr] Pull 1 byte from the stack

```
pull1   Loc          ;Push byte and store
                     ; it at Loc
pull1   Loc,x        ;Push byte and store
                     ; it at Loc,x
pull1                ;Push byte into A
```

pullword [opr] Pull 2 bytes from the stack

```
pull2   Loc          ;Pull bytes and store
                     ; them at Loc
pull2   Loc,x        ;Pull bytes and store
                     ; them at Loc,x
pull2                ;Pull bytes into A
```

pull3 addr Pull 3 bytes from the stack

```
pull3   Loc          ;Pull bytes and store
                     ; them at Loc
```

pulllong [addr1[,addr2]] Pull 4 bytes from the stack

```
pulllong   Loc                ;Pull bytes and store
                              ; them at Loc
pushlong   Locl,Loc2          ;Pull bytes and store
                              ; them at both Locl
                              ; and Loc2
pulllong   [zeropg],offset    ;Pull bytes and store
                              ; them using indirect
                              ; addressing
pulllong                      ;Pull 4 bytes. Discard
                              ; first 3, store
                              ; second 2 in A
```

pullxy [addr] Pull 4 bytes from the stack using X and Y

```
pullxy                        ;Pull into X and Y
pullxy  Loc                   ;Pull into X and Y,
                              ; store at Loc
```

pullay Pull 4 bytes from the stack into A and Y

pullx [addr] Pull 2 bytes from the stack using X

```
pullx                         ;Pull into X
pullx   Loc                   ;Pull into X, store
                              ; also at Loc
```

Load and Store Macros

lday addr[,offset] Load A and Y (4 bytes)

```
lday    Loc                   ;Load A from Loc, Y
                              ; into Loc+
lday    #n                    ;Load A and Y with
                              ; constant n
lday    [zeropg],offset       ;Load A and Y
                              ; indirectly
lday    zp,x                  ;Load A from zp,x, Y
                              ; from zp+2,x
```

stay addr[,offset] — Store A and Y (4 bytes)

```
stay    Loc             ;Store A into Loc, Y
                        ; into Loc+2
stay    [zeropg],offset ;Store A and Y
                        ; indirectly
stay    zp,x            ;Store A into zp,x, Y
                        ; into zp+2,x
```

zero bytes,addr — Zero a block

```
zero    #n,block        ;Clear the first n
                        ; bytes of block
zero    #n,[Loc]        ;Clear the first n
                        ; bytes of the block
                        ; whose address is in
                        ; Loc
zero    num,block       ;Obtain the byte count
                        ; from memory
```

Add and Subtract Macros

add [term1],term2[,result] — Add 2-byte integers

```
add     Loc1,Loc2,Loc3  ;Add Loc1 to Loc2 and
                        ; store the sum in
                        ; Loc3
add     Loc1,Loc2       ;Add Loc1 to Loc2 and
                        ; return the sum in A
add     ,Loc2,Loc3      ;Add Loc2 to A and
                        ; store the sum in
                        ; Loc3
add     #4,Loc2         ;Add 4 to Loc2 and
                        ; return the sum in A
```

add4 [term1],term2[,result] — Add 4-byte integers

See *add* for examples.

sub [term1],term2[,result] — Subtract 2-byte integers

```
sub     Loc1,Loc2,Loc3          ;Subtract Loc2 from
                                ; Loc1 and store the
                                ; result in Loc3
sub     Loc1,Loc2               ;Subtract Loc2 from
                                ; Loc1 and return the
                                ; result in A
sub     ,Loc2,Loc3              ;Subtract Loc2 from A
                                ; and store the
                                ; result in Loc3
sub     Loc1,#4                 ;Subtract 4 from Loc1
                                ; and return the
                                ; result in A
```

sub4 [term1],term2[,result] Subtract 4-byte integers

See *sub* for examples.

Define Macros

str 'string' Define string

Generates a Pascal-type string — that is, a 1-byte character count followed by the string. For example,

```
ThisString str 'This is my string'
```

produces the statement:

```
ThisString dc i1'17',c'This is my string'
```

dp pointer Define pointer

Calculates the 4-byte address of the operand and puts it in a DC statement of the form:

```
dc i4'pointer'
```

Move Macros

move1 from,to[,to2] Move 1 byte

```
move1   Here,There              ;Copy the byte at Here
                                ; into There
move1   Here,There,There2       ;Copy the byte at Here
```

```
                                    ; into both There
                                    ; and There2
    movel   #n,There                ;Copy the constant n
                                    ; into There
```

moveword from,to[,to2] Move 2 bytes

```
    moveword   Here,There           ;Copy the bytes at
                                    ; Here into There
    moveword   Here,There,There2    ;Copy the bytes at
                                    ; Here into both
                                    ; There and There2
    moveword   #n,There             ;Copy the constant n
                                    ; into There
    moveword   [zeropg],offset,     ;Copy 2 bytes to There
                                    ; using There in-
                                    ; direct addressing
```

move3 from,to[,to2] Move 3 bytes

```
    move3   Here,There              ;Copy the 3 bytes at
                                    ; Here into There
    move3   Here,There,There2       ;Copy the 3 bytes at
                                    ; Here into both
                                    ; There and There2
    move3   #n,There                ;Copy the constant n
                                    ; into There
```

movelong from,to[,to2] Move 4 bytes

```
    movelong   Here,There           ;Copy the bytes at
                                    ; Here into There
    movelong   Here,There,There2    ;Copy the bytes at
                                    ; Here into both
                                    ; There and There2
    movelong   #n,There             ;Copy the constant n
                                    ; into There
    movelong   [zeropg],offset,     ;Copy 4 bytes to There
                                    ; using There in-
                                    ; direct addressing
```

```
movelong  Here,x,There        ;Copy the bytes at
                              ; Here plus index X
                              ; into There
movelong  Here,y,There        ;Copy the bytes at
                              ; Here plus index Y
                              ; into There
```

Important: For the last two formats, *x* and *y* must be in lowercase.

Shift Macros

asl4 addr[,count] — Left-shift 4 bytes

```
asl4    Loc,#3                ;Left-shift contents
                              ; of Loc by 3 bit
                              ; positions
asl4    Loc,CountLoc          ;Left-shift contents
                              ; of Loc by count
                              ; stored in CountLoc
asl4    Loc                   ;Left-shift contents
                              ; of Loc by count
                              ; stored in X
                              ; register
```

lsr4 addr[,count] — Right-shift 4 bytes

```
lsr4    Loc,#3                ;Right-shift contents of
                              ; Loc by 3 bit positions
lsr4    Loc,CountLoc          ;Right-shift contents of
                              ; Loc by count stored in
                              ; CountLoc
lsr4    Loc                   ;Right-shift contents of
                              ; Loc by count stored in X
                              ; register
```

Mode Macros

native [long/short] — Turn on native mode

```
native long     ;Native mode with 16-bit registers
native          ;Same as preceding
native short    ;Native mode with 8-bit registers
```

emulation Turn on emulation mode

long Set memory, A, X, and Y to 16 bits

longm Set memory and A register to 16 bits

longx Set X and Y registers to 16 bits

short Set memory, A, X, and Y to 8 bits

shortm Set memory and A register to 8 bits

shortx Set X and Y registers to 8 bits

Write Macros

writech [opr] Write a character

```
writech                    ;Write character in A
                           ; register
writech   #'A'             ;Write an "A"
writech   Loc,x            ;Write character
                           ; addressed by Loc,x
```

writestr [opr] Write a string

```
writestr                   ;A and Y point to string
writestr  #'Press a key'   ;Write "Press a key"
writestr  Loc              ;Write string start-
                           ; ing at Loc
```

writeln [opr] Write a line (string + CR)

```
writeln                    ;Write Carriage Return
                           ; only
writeln   #'Press a key'   ;Write "Press a key",
                           ; then return
writeln   Loc              ;Write line starting
                           ; at Loc
```

Check Error Macro

Check_Error error_subr Call error subroutine if carry is 1

The user-defined subroutine *error_subr* can be anywhere in the current program bank.

Using Predefined Macros

The *APW* disk's LIBRARIES/AINCLUDE subdirectory contains more than 20 macro library files. To use a macro in your program, you must copy its definition into your program's macro file using MACGEN.

Eventually, you will know instinctively where specific macros are located. Until then, however, the easiest way to copy the definitions your program needs is to make MACGEN search the *entire* directory, by giving it the equal sign (=) wildcard. Hence, your MACGEN command would be of the form:

```
macgen myprog.src myprog.macros /apw/libraries/
ainclude/ml6.=
```