6502 SOFTWARE DESIGN

Leo J. Scanlon

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

The Blacksburg Continuing Education™ Series

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

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

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

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

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

6502 Software Design

By

Leo J. Scanlon

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

FIRST EDITION
SECOND PRINTING—1980

International Standard Book Number: 0-672-21656-6
Library of Congress Catalog Card Number: 79-67131

Printed in the United States of America.

Preface

The 6502 integrated circuit is a very popular microprocessor. It is currently used in general-purpose microcomputers, video games, and personal computers such as the Apple and the Pet 2001. Many of these microcomputers are programmed in the BASIC* programming language, which makes it very easy to write programs that will perform complex calculations or play games. However, the BASIC language also has its disadvantages. It is relatively slow (only a few hundred statements can be executed every second) and it is not very suitable for controlling peripheral devices. Therefore, if you have a high-speed data-processing or peripheral-control requirement, assembly language programs will probably have to be written.

Once you have decided that assembly language is the language to use, you will need a 6502-based microcomputer that you can use to generate and test your programs. The microcomputer that has been used as the basis for this book is the AIM 65. It is manufactured by Rockwell International. The AIM 65 has a 54-key keyboard, a 20-character alphanumeric LED display, a 20-column thermal printer, a teletypewriter I/O port and two audio cassette I/O ports. As such, it is a very powerful, inexpensive microcomputer system. Even though we have used this microcomputer in our examples, most of the programs listed in this book can be used on all 6502-based microcomputers. However, they may have to be slightly altered to reflect the memory and I/O devices that are wired to your microcomputer.

This book has nine chapters. Chapter 1 discusses the characteristics of the 6502 integrated circuit and the AIM 65 microcomputer.

---

*BASIC is a registered trademark of the trustees of Dartmouth College.

This includes a description of the various *registers* contained within the 6502 integrated circuit. These registers are important because you will use them time and time again in your programming. Chapter 2 presents descriptions of the instructions that the 6502 integrated circuit can actually execute. However, you will not find long detailed programs in this chapter, since you will not have enough familiarity with all of the important instructions to understand long complex programs. Then, Chapter 3 discusses subroutines, which are instruction sequences that are designed to be used at several places in a program.

Chapters 4 through 6 present techniques needed to process structured data (such as lists and tables), perform mathematical operations, and convert data from one number base to another. Finally, Chapters 7 through 9 describe how the 6502 instructions can be used to transfer information between the 6502 integrated circuit and the peripheral input/output devices.

This chapter arrangement, starting with very fundamental material and gradually introducing more complex topics, is intended to increase your understanding of the 6502 integrated circuit in an *orderly* manner.

LEO J. SCANLON

This book is dedicated to my wife, Pat,
and my sons, Roger and Ryan.

Acknowledgments

On a personal note, it is only proper to mention that although the only name on the title page is that of the author, this book reflects the efforts of many people. In particular, the author is indebted to Dr. Christopher A. Titus of Tychon, Inc., the editor (and reader's advocate) for this book, for his keen insight and many constructive suggestions. Special thanks must also go to Dr. Lance A. Leventhal of Emulative Systems, who gave of his valuable time with infectious enthusiasm. Finally, the author owes thanks to many dedicated people at Rockwell International in Anaheim, California, with particular appreciation for the management support of Bob Anslow and Scotty Maxwell and the technical contributions of Gordon Smith, Dick Anderson and Leo Pardo.

Contents

CHAPTER 4

CHAPTER 5

CHAPTER 6

CHAPTER 7

CHAPTER 8

CHAPTER 9

APPENDIX A

APPENDIX B

List of Program Examples

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

CHAPTER 6

CHAPTER 7

CHAPTER 8

CHAPTER 9

1

An Introduction to the 6502 Microprocessor

The purpose of this chapter is to introduce the 6502 microprocessor to those readers who are unfamiliar with its operation. This introduction is sufficiently detailed so that you will gain an understanding of the 6502 integrated circuit and how it functions in a computer system.

WHY THE 6502?

To understand where the 6502 fits in the microprocessor spectrum, a brief look must be taken at the evolution of 8-bit microprocessors. The first 8-bit microprocessor to make a significant impact on the industry was the 8008 produced by Intel Corporation. Fabricated with p-channel metal-oxide semiconductor (PMOS) technology, the 8008 is considered the foremost "first-generation" 8-bit microprocessor. The 8008 was designed with a calculatorlike architecture, and had six scratch-pad registers, an internal stack register, and special instructions to perform input and output. In 1973, Intel Corporation introduced a "second-generation," silicon-gate, NMOS version of the 8008 microprocessor, and called it the 8080.

The 8080 is essentially an improved 8008, with more addressing, more instructions, and faster instruction times. The internal organization is better too, but the overall 8008 architectural philosophy is maintained in the 8080. The 8080 is historically the second-generation de facto standard in microprocessors; the circuit that many people think of first when microcomputers are mentioned. Intel Corporation got a head start on the industry with the 8008, and

preserved it with the 8080 through the early 1970s. Until Motorola, Inc. introduced the 6800 microprocessor in 1974, Intel Corporation had virtually no competition.

Motorola, Inc. saw the tremendous microprocessor market potential evolving, and decided to make an entry of their own. They had essentially two ways to go: (1) they could challenge Intel Corporation on their own ground, by producing a new and improved 8080 (as Zilog, Inc. did in 1976 with the Z80), or (2) they could ignore that approach and design a more *advanced* microprocessor. Realizing that it would be extremely difficult to establish a strong market position (not to mention a *leading* position) by going after Intel Corporation with a "me too" product, Motorola, Inc. decided to challenge with a superior product.

The resulting product, the 6800 microprocessor, was organized along the lines of classic computer architectural concepts, with input and output devices accessed as memory. In the 6800 microprocessor, the load and store instructions used to access memory are the same instructions used to perform input (read) and output (write) operations on peripheral devices. This technique, called *memory-mapped I/O*, eliminates the performance bottlenecks that are associated with having to pass all the data handling and manipulation through a working register, as in the register-based architecture of the 8080.

The preceding brief overview was necessary in order to set the stage for introducing our subject microprocessor, the 6502. The 6502 device was designed by eight ex-employees of Motorola who saw that advances in processes, coupled with a few architectural and software changes, could result in a potentially highly marketable 6800-like microprocessor. They joined a calculator-chip company called MOS Technology.

The MOS Technology design team had two objectives in mind for their next-generation[1] microprocessor—low cost and high performance. Since there is a direct correlation between the manufacturing cost and the die size (the size of the piece of silicon that contains the transistors and resistors which make up the microprocessor), they reduced the complexity of the basic 6800 design as much as possible to minimize the amount of silicon required. Other design decisions included eliminating one of the two accumulators in the 6800 and its tri-state address output buffers. They also replaced the 16-bit index register of the 6800 microprocessor with two separate 8-bit index registers, and they discarded some of the lesser-used instructions of the 6800.

The elimination of instructions opened up some instruction-decode space and permitted the designers to provide the 6502 microprocessor with 13 addressing modes, 7 more modes than the 6800

device has. These modes give the 6502 device capabilities that are normally found only in larger computers. Additionally, the design team realized that although computers are binary machines, man is inherently a decimal-thinking animal, so they added a mode-selection instruction and control bit that allows the 6502 microprocessor to operate on either binary or decimal data. This means that the programmer does not have to remember to write in "decimal adjust" instructions after addition or subtraction operations. For electrical efficiency, the design team employed the newer depletion-load technology, which gives the 6502 clean switching characteristics, low-power dissipation (250 mW typical for the 6502 versus 600 mW typical for the 6800), and good noise immunity.

The 6502 device is one of 10 software-compatible microprocessors that MOS Technology introduced in 1975 as the 6500 Series. Through second-source agreements, the 6500 Series is also produced by Rockwell International and Synertek. All 10 microprocessors have the same instruction set and the same basic architecture, varying only in size and hardware options. The 6500 Series has been very popular since its introduction, and by the end of 1978, more 6500-Series microprocessors were being shipped than all other 8-bit microprocessors, including the 8080 and 6800.

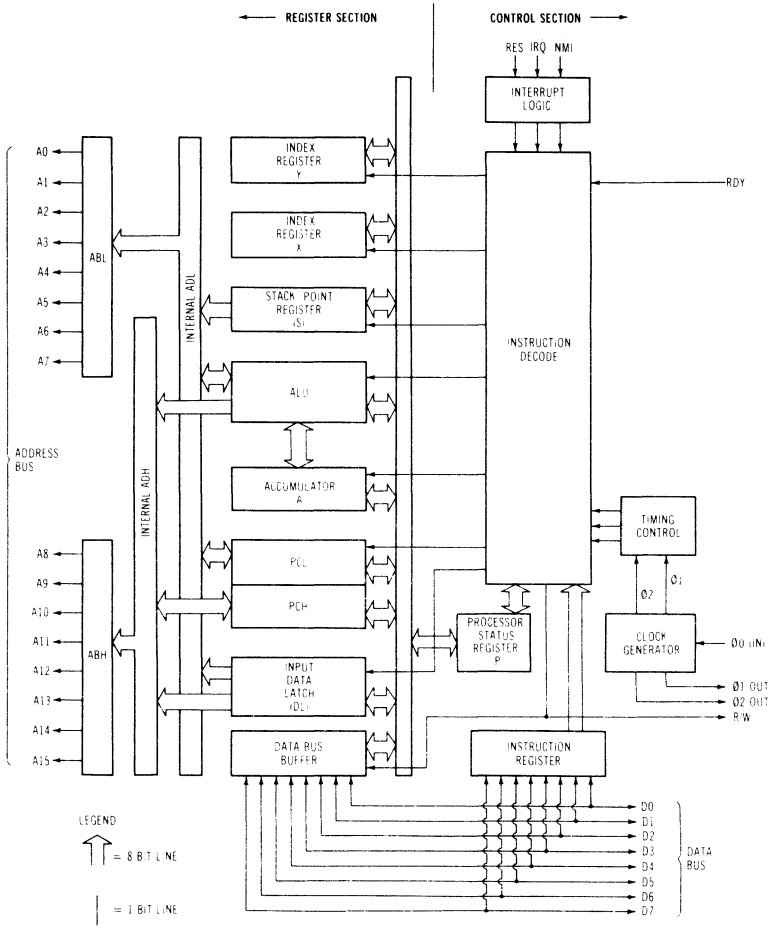
Today, the best-selling 8-bit microprocessors are divided into two distinct families[2]—the 8080/Z80 family, with its register-oriented architecture, and the 6500/6800 family, with its memory-mapped architecture. Which architecture will be the more favored in the 1980s? It is impossible to know which at this time, but the manufacturers of the 6800 and 6500 devices are banking solidly on their opinion that the more minicomputer-like architecture holds the greatest potential for advanced circuits. Intel Corporation has not yet shown signs of sharing this opinion, but it may be significant to note that the latest 16-bit microprocessor from Zilog, Inc., the Z8000, represents a solid break with the 8080/Z80 design concept by including memory-mapped I/O.

THE 6502 MICROPROCESSOR

The 6502 *microprocessor* can be combined with memory and input/output integrated circuits to form a *microcomputer*. As the "heart" of the microcomputer, the 6502 regulates all operations of the microcomputer, based on the sequence of instructions (the program) that it is executing. The 6502 can execute 56 different *types* of instructions. The various combinations of addressing that are available for use by individual instruction types give the microprocessor a total of 151 executable instructions. The 6502 instruction set is described in detail in subsequent chapters of this book;

for now, let us focus our attention on the internal organization (the architecture) of the 6502 and find out *how* it operates.

Fig. 1-1 is a block diagram of the internal architecture of the 6502. It shows the elements of the microprocessor and the buses



Courtesy Rockwell International

Fig. 1-1. Block diagram of the 6502 microprocessor.

by which they communicate with each other and with external circuits. The 6502 contains most of the control and decision-making logic, so only a few additional circuits are required to configure a small microcomputer system. One of the functions of this additional control logic is to provide the 6502 microprocessor with a clock

signal that the internal clock generator will use to generate its two-phase system clock. The 6502 also requires a single +5-volt dc power supply. All of the other inputs and outputs of the 6502 integrated circuit are compatible with standard transistor-transistor-logic (TTL).

Communication With External Devices

The 6502 is an 8-bit microprocessor, which means that the basic unit of the information, the byte, is 8 bits wide. Further, all information is transferred to and from memory and the input/output (I/O) devices 8 bits at a time. To transfer more than 8 bits requires additional transfer operations. All information transfers between the 6502 and external devices are conducted on the 8-line *data bus.* The data bus is bidirectional so the same lines are used to transfer information both into and out of the 6502 microprocessor.

Since a 6502-based system can include a variety of memory and I/O circuits, how does the microprocessor specify whether it wants to communicate with memory or with an I/O circuit? The answer to that is that the 6502 microprocessor, unlike many other microprocessors, *makes no differentiation* between memory and I/O devices—it treats every external device as memory!

Does that mean that the 6502 microprocessor does not have special input or output instructions in its instruction set? Yes, that is so; the 6502 does not have any such instructions. However, since the 6502 does not know whether it is addressing a memory location or peripheral device, the same instructions that read from. or write to, memory can be used to input from, or output to, peripheral devices. If you want to read the contents of a memory location or input a data byte from a peripheral, a load instruction is executed. Similarly, if you want to write a data byte into a memory location or output it to a peripheral, a store instruction is executed. Each peripheral devices as well as each memory location. has a *uniaue address.* The 6502 microprocessor uses this unique address to select one memory location or one peripheral device; therefore, it can exchange with either the memory location or the peripheral device.

The 6502 transmits addresses to all memory and peripheral device over 16 lines that are collectively known as the *address bus.* Being 16 bits wide, the address bus can select any of 65,536 (64K) locations. All external devices, both memory circuits and I/O circuits, must be connected to the address bus.

How can an addressed external device know whether the 6502 wants to input (read) information from or output (write) information to it? The external device knows this by sensing the state of a Read/$\overline{\text{Write}}$ control line (R/$\overline{\text{W}}$) which is high (a logic 1; +2.4 V to +5 V) when a load instruction (read) is executed and

low when a store instruction (write) is executed. The R/$\overline{W}$ signal is one of six signals available on the *control bus* of the 6502. Quite often, the term Read/$\overline{\text{Write}}$ is pronounced "read-write bar." In this and all future discussions, if a signal has a "bar" over it, the signal is active low; that is, it is active when it carries a voltage between 0 (ground) and +0.8 V dc.

How the 6502 Executes a Program

The 6502 microprocessor executes programs by fetching an instruction from memory, executing it, and then fetching the next instruction. A special register, called the *program counter,* determines which memory location will be accessed next. The program counter is automatically incremented after each memory access so that it addresses the next consecutive memory location. Because the program counter is 16 bits wide, it can address any location in the 64K-byte address space of the 6502.

Instructions are comprised of one, two, or three bytes. The first byte always holds the machine-code equivalent of the operation code (*op code*), so this byte is directed into the *instruction register* and routed to the *instruction decode logic.* The instruction decode logic issues appropriate internal-control signals to all other elements of the microprocessor, and possibly to external circuits in the microcomputer system. The second and third bytes, if the instruction has them, are gated into the *data bus buffer,* from which they are routed into either the *Arithmetic Logic Unit* (ALU) if they represent data, or into the *program counter* if they represent an address.

How fast can the 6502 microprocessor process data or communicate with peripheral devices? This will depend on the instructions that are executed. All instructions require a known number of clock cycles in order to be executed, so the speed at which the instructions are executed depends on which version of the 6502 is being used. The microprocessor is available in 1 MHz, 2 MHz, and 3 MHz versions. With a 1-MHz 6502 microprocessor, the simplest instruction will be executed in 2 microseconds (2 μs) and the most complex instruction will require about 7 μs. In general, most instructions require about 3 or 4 μs to be executed. The execution time of all instructions will be listed in Chapter 2.

General-Purpose Registers

The 6502 microprocessor has three 8-bit registers that your programs can use to save temporary data values, to communicate with memory, to maintain counters, and for a variety of other applications. These three registers are the *accumulator,* the *X register* and the *Y register.*

The accumulator is the only register in which arithmetic and logical operations can be performed, and it holds one of the operands for each add, subtract, AND, OR, and Exclusive-OR instruction. The 6502 also has instructions to logically shift the contents of the accumulator to the right or to the left.

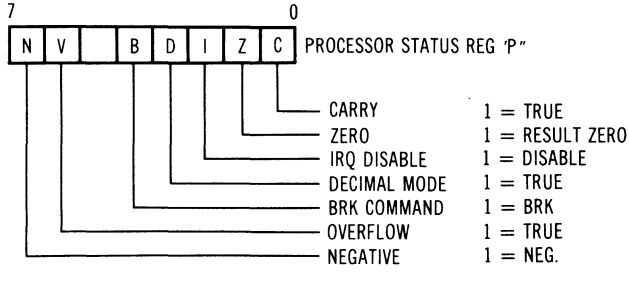
The X and Y registers are primarily employed as index registers (to access sequential data values in memory), but the fact that they can be incremented and decremented under program control also makes them popular as general-purpose counters.

The Processor Status Register

The *processor status register* (Fig. 1-2) contains seven usable bits. Five of these bits are *status flags;* they provide information on the result of a previously executed instruction (in most cases, the preceding instruction). The two other usable bits are control bits.

Let us look at the status flags first. The *Carry flag* (C) is used to save any carry produced by an add operation, any borrow produced by a subtract operation, or the value of a bit after a shift operation. The Carry also reflects the result of a compare operation. The *Zero flag* (Z) indicates whether or not the result of an operation is zero. The *Break Command flag* (B) indicates whether an interrupt request to the 6502 microprocessor was caused by a "break" instruction or by an externally generated interrupt. The *Overflow flag* (V) is applicable only to arithmetic operations on signed numbers. It is set if the addition of two like-signed numbers or the subtraction of two unlike-signed numbers produces a result greater than $+127_{10}$ or less than -128_{10}. The *Negative flag* (N) indicates whether or not the result of a signed arithmetic operation produced a negative result. This bit is also used as a general-purpose indicator of the state of the most-significant bit position in the accumulator.

The 6502 can be programmed to test the condition, or state, of each of these flags. Based on the results of these tests, the 6502 can



Courtesy Rockwell International

Fig. 1-2. The processor status register.

decide whether or not to execute one of two possible sequences of instructions. All flags remain set or cleared after these "test" operations are performed. Therefore, *not all 6502 instructions affect the flags.*

Now, let us discuss the control bits in the processor status register. The first one is the *IRQ Disable bit* (I) which is used to "lock out" external interrupts to the 6502 at times when, for some reason, the microprocessor is not prepared to service an interrupt. Interrupts are automatically disabled by the 6502 while it is being reset or when it is *servicing* a previous interrupt. Programs can also disable interrupts during periods when a certain sequence of instructions must be permitted to be executed uninterrupted.

The second control bit is the *Decimal Mode bit* (D) which controls whether the internal Arithmetic Logic Unit (ALU) of the 6502 is to operate as a straight binary adder or as a decimal adder. In the binary mode, the ALU treats arithmetic operands as 8-bit binary numbers. In the decimal mode, the ALU treats arithmetic operands as two BCD (Binary-Coded Decimal) digits packed into one 8-bit byte.

Reset and Interrupt Signals

There are three separate input signals by which external devices can cause an executing program in the 6502 microprocessor to be interrupted. These signals ($\overline{\text{RES}}$, $\overline{\text{IRQ}}$, and $\overline{\text{NMI}}$) are shown in the top right-hand portion of Fig. 1-1, wired to the *Interrupt Logic Section.* Although they are functionally different, all three signals produce the same general result; they load the program counter with the contents of two consecutive memory locations, which contain the starting address of a program that is unique to that particular signal.

The *Reset* ($\overline{\text{RES}}$) input pin is used to initialize the 6502 microprocessor to a known state, or to start the 6502 when power is first applied to the microcomputer. While $\overline{\text{RES}}$ is grounded, the 6502 can neither transmit nor receive information. When the ground is removed, the microprocessor loads the program counter with the contents of memory locations FFFC and FFFD (hexadecimal), the address from which the *very first* (or initial) instruction will be fetched.

Interrupt Request ($\overline{\text{IRQ}}$) is the input pin by which most peripheral devices request service from the 6502 microprocessor. "Request" is the keyword here. Unlike the Reset signal, which interrupts the 6502 unconditionally, $\overline{\text{IRQ}}$ simply informs the microprocessor that some peripheral device in the system (keyboard, printer, etc.) is waiting to send or receive information. An interrupt request will be acknowledged only if the Interrupt Disable bit (I) of the proc-

essor status register is reset to a logic zero. If "I" is reset, the 6502 will load the contents of the two uppermost memory locations, FFFE and FFFF (hexadecimal), into the program counter. If "I" is set when the interrupt request is received, the microprocessor ignores the request, and continues executing as if no request has been made. The 6502 does not "remember" the request, but when the "I" bit is reset, the 6502 is interrupted.

The name of the third signal, *Non-Maskable Interrupt* ($\overline{\text{NMI}}$), gives a clue as to its nature. The $\overline{\text{IRQ}}$ input is *maskable;* that is, it can be enabled or disabled, depending on the state of the "I" bit in the processor status register. However, $\overline{\text{NMI}}$ is nonmaskable—it cannot be disabled. Like $\overline{\text{RES}}$, $\overline{\text{NMI}}$ does not merely request to interrupt the microprocessor, it *does* interrupt the microprocessor each time it is activated. The $\overline{\text{NMI}}$ line is designed to interrupt the 6502 microprocessor under some condition that requires immediate attention, such as a power failure. The address of the sequence of instructions that service the $\overline{\text{NMI}}$ interrupt is stored in two consecutive memory locations, FFFA, and FFFB (hexadecimal).

The Stack Pointer Register

The 6502 microprocessor can be programmed so that at some point in a program, program execution can be transferred to another sequence of instructions that is stored in another part of memory. Before this transfer actually occurs, the 6502 saves the address of the next instruction in its current sequence of instructions. After the new sequence of instructions has been executed, control can be *returned* to the point that is just after the instruction that caused the first transfer operation. The *return address* is saved for use later in an area of memory called a *stack.*

There is nothing mystical about the stack; it is simply a portion of memory that is designated to accept these return addresses. However, the stack must be implemented in Page 1 of the 6502 address space—the addresses from 0100 to 01FF (hexadecimal). Since this design restriction ensures that the high-order two digits of the address are always 01, the address register for the stack—the Stack Pointer—is only 8 bits wide.

Information is entered onto, and extracted from, the stack of the 6502, in memory, the same way that we stack dishes in the kitchen. The last item to be placed on the stack is also the first item to be removed from it. This type of stack is usually referred to as "last in, first out." As return addresses are entered onto the stack, they are really stored in R/W memory at lower and lower memory addresses; the stack "builds" toward address 0. The Stack Pointer, therefore, is automatically decremented by 1 as each new address byte is *pushed* onto the stack, and is automatically incremented by 1 as

each address byte is *pulled* off of the stack. The accumulator and the processor status register can also be saved on the stack, if desired.

MACHINE CODE AND ASSEMBLY LANGUAGE

For the 6502 microprocessor to perform a specific task, it must be programmed to do so. A program is nothing more than a sequence of instructions stored in sequential memory locations. The 6502 executes the program, one instruction at a time. It fetches an instruction from memory, decodes it, performs the decoded command, and then fetches the next instruction. This cycle is repeated until all instructions in the program have been executed.

What do these instructions look like? Since the 6502 microprocessor is simply a collection of electronic circuits (albeit in microscopic form), the instructions are composed of binary numbers (1's and 0's) that cause some internal electrical signals to be turned on, others to be turned off. The 6502 is an 8-bit microprocessor, so these binary instructions are comprised of multiples of 8 binary bits.

In early computers, all programming was done in the binary form, normally with switches controlling the individual bits—to set a bit to "1," turn the switch on; to reset a bit to "0," turn the switch off. But a string of 1's and 0's presents such a confusing mess that the computer industry soon realized the need for decimal loaders, which allowed the instructions to be written in decimal form. Decimal loaders were eventually replaced by hexadecimal loaders, which allowed the instructions to be written in hexadecimal form. Example 1-1 shows both binary and hexadecimal forms of typical program instructions.

Example 1-1. A Typical 6502 Program, in Binary and Hexadecimal Notation

Binary	Hexadecimal
10100101	A5
00100001	21
11000101	C5
00100000	20
10110000	B0
00101011	2B
10100110	A6
00101100	2C

Hexadecimal representation is some help to the programmer because it frees him from using all those error-prone 8-bit binary numbers. Further, hex instructions do not contain quite so many digits in them, making them somewhat easier to memorize. Unfortunately, though, a hexadecimal number gives no hint as to the function

of an instruction. Does a "C5" instruction perform an addition, a subtraction, store a value in memory, or none of these? Even when using hexadecimal numbers, it is still difficult to program the 6502 microprocessor. Before you enter a hexadecimal number into the microcomputer, you would first have to find the instructions, that you want to store in the memory of the 6502, on a list provided by the microcomputer manufacturer. The appropriate hexadecimal number (op code) can then be found next to the instruction. The time spent in looking up the instructions and op codes could be very costly in developing a program, not to mention the possibilities of errors.

The next higher level of programming permits the programmer to write instructions in an abbreviated form, something closer to a human language, using abbreviations called *mnemonics* that can be correlated directly to the function of the instruction. A computer program can then be executed so that these mnemonics are actually converted to the sequence of 1's and 0's that the 6502 can execute. The program that converts these abbreviations into machine code (1's and 0's) is called an *assembler*, so this form of programming is called *assembly language programming*. An instruction that increments the X register by one has a hexadecimal form of E8 and an assembly language mnemonic of INX. Which do you think is easier to remember? Table 1-1 lists the assembly language mnemonics for several 6502 instructions.

How does the assembler translate instruction mnemonics to binary codes that the 6502 can execute? The assembler contains a large table (the *permanent symbol table*) that contains all the mnemonics (represented by strings of ASCII characters) and their binary equivalents. The assembler compares the mnemonic in your program (a string of ASCII characters) to each ASCII character string in the permanent symbol table. When a "match" occurs, the assembler fetches the binary code associated with the mnemonic, and uses this value during the assembly process. Therefore, the assembler translates the mnemonic INX (Increment X by 1) to E8, and the mnemonic CLC (Clear Carry flag) to 18. In this book, all example programs will be written using the standard mnemonics;

Table 1-1. Some 6502 Mnemonics

Mnemonic	Instruction
ADC	Add memory to accumulator with Carry
CLC	Clear Carry flag
INX	Increment index X by 1
LDA	Load accumulator with memory
TAX	Transfer accumulator to index X

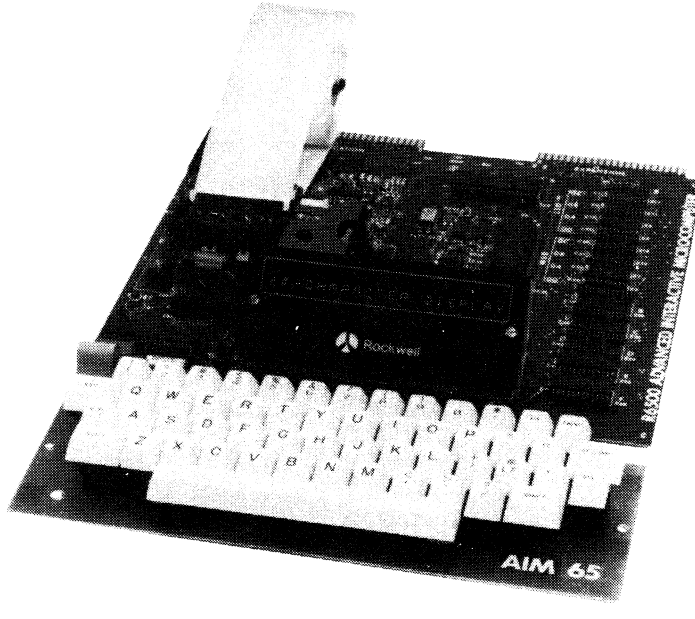
these are the mnemonics that are defined in the literature of the manufacturers.

THE AIM 65 MICROCOMPUTER

The 6502 microprocessor is used in several popular microcomputers, including the Apple II (Apple Computer, Inc.), the Pet 2001 (Commodore Business Machines, Inc.), the Challenger C2 (Ohio Scientific, Inc.), the KIM-1 (MOS Technology) and the SYM-1 (Synertek). In 1978, Rockwell International introduced the AIM 65 as its entry in the 6502-based microcomputer marketplace. The AIM 65 is shown in Fig. 1-3.

The AIM 65 (an acronym for *A*dvanced *I*nteractive *M*icrocomputer R6500) is primarily intended as an educational tool for individuals who are interested in learning about microcomputers, rather than just playing with a sophisticated toy. Its features also make the AIM 65 attractive as a low-cost development system for design engineers.

The features of the AIM 65 are indeed impressive. Perhaps the most exciting feature is the on-board alphanumeric printer, for



Courtesy Rockwell International

Fig. 1-3. The AIM 65 microcomputer.

generating hard-copy listings. The printer can produce up to 120 lines per minute, with up to 20 characters per line. Characters are represented by 5-wide × 7-high dot matrices, and are "printed" on heat-sensitive roll paper. The printer is complemented by a 20-character, 16-segment, alphanumeric display. The printer and display can be used with all 64 upper-case ASCII characters. Like the more expensive Apple II and Challenger C2 microcomputers, the AIM 65 has a 54-key typewriter-style keyboard, which is easier to use and less fatiguing in lengthy programming sessions than the calculator keypads of other microcomputers.

The operating software of the AIM 65 resides in on-board read-only memory (ROM) circuits, and includes a monitor and a symbolic text editor. A portion of the monitor is a pseudo-assembler, which allows instructions to be entered in mnemonic form, rather than in hexadecimal machine codes (as in the KIM-1 and the SYM-1). The monitor also has a disassembler that translates the machine code in memory to its mnemonic form, for printout or display. Rockwell International also offers a two-pass symbolic assembler as a ROM option, which gives you the capability of using symbolic labels in your programs.

The AIM 65 has 1024 bytes of read/write (R/W) memory installed, that can be optionally expanded on-board to 4096 bytes. If desired, the user can interface additional R/W or ROM integrated circuits, up to 64K, to the AIM 65. Also included in the AIM 65 is a 6522 *Versatile Interface Adapter* (VIA) circuit, which is entirely user-dedicated. The VIA has two I/O ports for off-board expansion, an 8-bit shift register, and two 16-bit timer/counters. One nice feature of the 16-bit timer/counters is that once they are started, they need no further intervention by the 6502 microprocessor.

The AIM 65 seems ideal to use for educating the readers of a book on 6502 software design and has, therefore, been selected as the demonstration microcomputer for the remainder of this book. Because the AIM 65 is equipped with a 1-MHz 6502 microprocessor, *all program times quoted* in this book refer to the time of a 1-MHz microprocessor. If you are programming while using a 2-MHz 6502 (6502A) microprocessor, divide the program times by two.

CONVENTIONS USED IN THIS BOOK

Given the proper type of loader, numbers using any base could be entered into the 6502 microprocessor. This includes the octal (base 8), hexadecimal (base 16), or even decimal (base 10) set of numbers. Since most 6502 microprocessor programming is conducted with hexadecimal numbers, they appear extensively through-

out this book. Sometimes they will be preceded by the word "hexadecimal" or its shortened form "hex," but more often a dollar sign ($) prefix will be used. The dollar sign prefix is the assembler prefix that indicates a hexadecimal operand. For example, hexadecimal address 14FB will be written $14FB.

Another way of indicating the base of a number is by applying a subscript to it. For example, $14FB_{16}$ is equivalent to $14FB and "hexadecimal 14FB." There are times when even a decimal number will be subscripted, if there is a possibility of an ambiguity. For example, the sentence "Addresses $1400 through $140F constitute 15_{10} locations" eliminates any uncertainty about whether the number 15 is a decimal number or a hexadecimal number.

REFERENCES

1. Cushman, R. H. "2½-generation μP's—$10 parts that perform like low-end mini's," *EDN*, September 20, 1975, pp. 36–41. (This contains an excellent comparison between the 6800 and the 6502 microprocessors, and includes historical details about the devices.)

2. Cushman, R. H. "μC Support Chip Directory: Solutions keep pouring forth" *EDN*, November 20, 1977, pp. 91–100. (This provides details on microprocessor "families" and the battle being waged on the support-chip front.)

3. *R6500 Microcomputer System Hardware Manual, Sections 1 and 2.* Rockwell International, Anaheim, CA, 1978. (Equivalent documents are also available from MOS Technology and Synertek.)

2

The 6502 Microprocessor Instruction Set

The 6502 microprocessor has 56 different instructions and 13 modes of addressing, making it one of the most versatile microprocessors ever designed. Table 2-1 is a complete list of the instructions, showing both their formal names and the abbreviations (or mnemonic) that you will be using to write your programs.

SUMMARY OF THE INSTRUCTION SET

Table 2-2 is a summary table to which you will be frequently referring in the course of your reading and programming. The Mnemonic columns (one on the left of the table, the other on the right of the table) list the instructions alphabetically. The Operation column gives a symbolic representation of the operation of each instruction. In this column, the internal registers of the 6502 are identified with a single letter—A for Accumulator, X for X Register, Y for Y Register, and S for Stack Pointer, etc. The "M" stands for Memory; not all of the memory, just the memory location being accessed by the instruction. A right-arrow (→) means "replaces," so A → M means that the contents of the accumulator (A) replaces the contents of the addressed memory location (M). The Operation column also uses single-letter identifiers for flags in the Processor Status Register—C for Carry, N for Negative, Z for Zero and V for Overflow.

The next 13 columns (Immediate, Absolute, etc.) represent the 13 addressing modes of the 6502. Each of these 13 columns is subdivided into three smaller columns:

- The OP in the first column is short for *op*eration code. (It is usually called op code.) OP gives the hexadecimal equivalent of the binary code that is stored into memory to represent each instruction. Note that the OP value for a particular instruction is different for each addressing mode. For example, the ADC

Table 2-1. 6502 Instruction Names

ADC	Add to Accumulator with Carry	LDA	Load Accumulator with Memory
AND	AND Memory with Accumulator	LDX	Load Index X with Memory
ASL	Accumulator Shift Left	LDY	Load Index Y with Memory
		LSR	Logical Shift Right
BCC	Branch on Carry Clear		
BCS	Branch on Carry Set	NOP	No Operation
BEQ	Branch on Result Equal to Zero		
BIT	Test Bits in Memory with Accumulator	ORA	OR Memory with Accumulator
BMI	Branch on Result Minus	PHA	Push Accumulator on Stack
BNE	Branch on Result Not Equal to Zero	PHP	Push Processor Status on Stack
		PLA	Pull Accumulator from Stack
BPL	Branch on Result Plus	PLP	Pull Processor from Stack
BRK	Force Break		
BVC	Branch on Overflow Clear	ROL	Rotate Left
BVS	Branch on Overflow Set	ROR	Rotate Right
		RTI	Return from Interrupt
CLC	Clear Carry Flag	RTS	Return from Subroutine
CLD	Clear Decimal Mode		
CLI	Clear Interrupt Disable Bit	SBC	Subtract from Accumulator with Carry
CLV	Clear Overflow Flag		
CMP	Compare Memory and Accumulator	SEC	Set Carry Flag
		SED	Set Decimal Mode
CPX	Compare Memory and Index X	SEI	Set Interrupt Disable Status
CPY	Compare Memory and Index Y	STA	Store Accumulator in Memory
		STX	Store Index X in Memory
DEC	Decrement Memory by One	STY	Store Index Y in Memory
DEX	Decrement Index X by One		
DEY	Decrement Index Y by One	TAX	Transfer Accumulator to Index X
EOR	Exclusive-OR Memory with Accumulator	TAY	Transfer Accumulator to Index Y
		TSX	Transfer Stack Pointer to Index X
INC	Increment Memory by One		
INX	Increment Index X by One	TXA	Transfer Index X to Accumulator
INY	Increment Index Y by One		
		TXS	Transfer Index X to Stack Pointer
JMP	Jump		
JSR	Jump to Subroutine	TYA	Transfer Index Y to Accumulator

instruction with immediate addressing has an op code value of 69, while the same instruction with absolute addressing has an op code value of 6D.

- The n column contains the number of machine cycles that the instruction requires in order to be executed when using the specific listed addressing mode.
- The # column contains the number of memory locations that the instruction occupies.

The Processor Status Codes column, on the right of the page, reflects the flags in the Processor Status Register which may be altered by the instruction. Flags that can be either set or cleared, depending upon operational variations, are identified with the letter code for that flag (e.g., an N indicates that the instruction may alter the Negative flag). Flags that are unconditionally set or cleared by an instruction are identified with a "1" or a "0."

Instruction Formats

All programs in this book are given in the AIM 65 assembler format. This format divides each line in the program (i.e., each line of program *code*) into four fields: label, op code, operand and comments.

The *label field* is used to assign a symbolic name or label to the location of an instruction, so that it can be referenced by other instructions in the program. For example, the instruction JMP THERE will cause the program counter to be unconditionally loaded with the memory address that has been assigned the label THERE. The instruction at label THERE will be the next instruction to be executed after the JUMP (JMP) instruction is executed. The label field is always optional. In fact, most instructions will not be labeled. However, if an instruction is labeled, the label must begin in the leftmost column of the line (Column 1), must begin with an alphabetic character (A through Z), and must be no longer than six characters.

The *op code field* is mandatory for every line in the program that contains an instruction, and must contain one of the 56 valid mnemonics listed in Table 2-1 or Table 2-2. The op code may begin in any column except Column 1, and must be separated from a label (if a label is present) by at least one space.

The *operand field* is used to specify data or an address for instructions that require an operand. (This subject will be discussed shortly.) The operand must be separated from the op code by at least one space. The assembler will accept operands in any of five forms. The form is specified by applying an appropriate prefix character to the operand, as follows:

Table 2-2. R6502 Instruction Set Summary

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND, X)			(IND), Y			Z PAGE, X			ABS, X			ABS, Y			RELATIVE			INDIRECT			Z PAGE, Y			PROCESSOR STATUS CODES	
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7 6 5 4 3 2 1 0 N V • B D I Z C	MNEMONIC
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										N V • • • • Z C	ADC
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										N • • • • • Z •	AND
ASL	C ← [7 6 5 4 3 2 1 0] ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													N • • • • • Z C	ASL
BCC	BRANCH ON C = 0 (2)																															90	2	2								• • • • • • • •	BCC
BCS	BRANCH ON C = 1 (2)																															B0	2	2								• • • • • • • •	BCS
BEQ	BRANCH ON Z = 1 (2)																															F0	2	2								• • • • • • • •	BEQ
BIT	A∧M				2C	4	3	24	3	2																																M7 M6 • • • • • Z •	BIT
BMI	BRANCH ON N = 1 (2)																															30	2	2								• • • • • • • •	BMI
BNE	BRANCH ON Z = 0 (2)																															D0	2	2								• • • • • • • •	BNE
BPL	BRANCH ON N = 0 (2)																															10	2	2								• • • • • • • •	BPL
BRK	BREAK													00	7	1																									• • • 1 • 1 • •	BRK
BVC	BRANCH ON V = 0 (2)																															50	2	2								• • • • • • • •	BVC
BVS	BRANCH ON V = 1 (2)																															70	2	2								• • • • • • • •	BVS
CLC	0 → C													18	2	1																									• • • • • • • 0	CLC
CLD	0 → D													D8	2	1																									• • • • 0 • • •	CLD
CLI	0 → I													58	2	1																									• • • • • 0 • •	CLI
CLV	0 → V													B8	2	1																									• 0 • • • • • •	CLV
CMP	A − M	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										N • • • • • Z C	CMP
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																																N • • • • • Z C	CPX
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																																N • • • • • Z C	CPY
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													N • • • • • Z •	DEC
DEX	X − 1 → X													CA	2	1																									N • • • • • Z •	DEX
DEY	Y − 1 → Y													88	2	1																									N • • • • • Z •	DEY
EOR	A ∀ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										N • • • • • Z •	EOR
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													N • • • • • Z •	INC
INX	X + 1 → X													E8	2	1																									N • • • • • Z •	INX
INY	Y + 1 → Y													C8	2	1																									N • • • • • Z •	INY
JMP	JUMP TO NEW LOC				4C	3	3																												6C	5	3						JMP
JSR	JUMP SUB				20	6	3																																			• • • • • • • •	JSR
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										N • • • • • Z •	LDA

L D X	M → X (1)	A2	2	2	AE	4	3	A6	3	2																					BE	4	3							B6	4	2	N • • • • • Z •	L D X
L D Y	M → Y (1)	A0	2	2	AC	4	3	A4	3	2														B4	4	2	BC	4	3														N • • • • • Z •	L D Y
L S R	0 → [ ] → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0 • • • • • Z C	L S R
N O P	NO OPERATION													EA	2	1																									• • • • • • • •	N O P
O R A	A V M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										N • • • • • Z •	O R A
P H A	A → Ms S − 1 → S													48	3	1																									• • • • • • • •	P H A
P H P	P → Ms S − 1 → S													08	3	1																									• • • • • • • •	P H P
P L A	S + 1 → S Ms → A													68	4	1																									N • • • • • Z •	P L A
P L P	S + 1 → S Ms → P													28	4	1																									(RESTORED)	P L P
R O L	←[ ]←[C]←				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													N • • • • • Z C	R O L
R O R	→[C]→[ ]→				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													N • • • • • Z C	R O R
R T I	RTRN INT													40	6	1																									(RESTORED)	R T I
R T S	RTRN SUB													60	6	1																									• • • • • • • •	R T S
S B C	A − M − C̄ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										N V • • • • Z (3)	S B C
S E C	1 → C													38	2	1																									• • • • • • • 1	S E C
S E D	1 → D													F8	2	1																									• • • • 1 • • •	S E D
S E I	1 → I													78	2	1																									• • • • • 1 • •	S E I
S T A	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										• • • • • • • •	S T A
S T X	X → M				8E	4	3	86	3	2																												96	4	2	• • • • • • • •	S T X
S T Y	Y → M				8C	4	3	84	3	2													94	4	2																• • • • • • • •	S T Y
T A X	A → X													AA	2	1																									N • • • • • Z •	T A X
T A Y	A → Y													A8	2	1																									N • • • • • Z •	T A Y
T S X	S → X													BA	2	1																									N • • • • • Z •	T S X
T X A	X → A													8A	2	1																									N • • • • • Z •	T X A
T X S	X → S													9A	2	1																									• • • • • • • •	T X S
T Y A	Y → A													98	2	1																									N • • • • • Z •	T Y A

(1) ADD 1 TO N IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO N IF BRANCH OCCURS TO SAME PAGE
ADD 2 TO N IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW
(4) IF IN DECIMAL MODE Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
Ms MEMORY PER STACK POINTER

\+ ADD
− SUBTRACT
∧ AND
V OR
∀ EXCLUSIVE OR

M_7 MEMORY BIT 7
M_6 MEMORY BIT 6
n NO. CYCLES
\# NO. BYTES

Courtesy Rockwell International

Prefix	Operand Form
(none)	Base 10 (Decimal)
$	Base 16 (Hexadecimal)
@	Base 8 (Octal)
%	Base 2 (Binary)
'	ASCII

The mnemonic entry mode of the AIM 65 accepts only a hexadecimal input, so AIM 65 users who are entering programs from the keyboard should ignore the preceding operand prefix list and write all operands as unprefixed hexadecimal numbers.

The *comment field* is always optional, and is used to add an explanatory note to a statement. The contents of the operand field are not executed, so you can write any kind of comment that you choose. However, the text of the comment should be preceded by a semicolon (;). Comments may be used alone, too, without being appended to a line that contains an instruction.

HOW THIS CHAPTER IS ARRANGED

This chapter gives a detailed description of the 13 addressing modes of the 6502 microprocessor, followed by descriptions of the instructions. Many books treat the instructions individually, discussing them one by one, alphabetically. Although this approach has definite merit in a reference book, it tends to leave the reader bewildered (and probably bored) after the fifth or sixth instruction. In this book, instructions are grouped by function, with similar instructions together. This approach is designed to aid *understanding*, and will (hopefully) avoid boring the reader. By the time you've finished this chapter, Table 2-2 should provide sufficient reference material for most of your programming. Alternatively, you may refer to Appendix B, where the 6502 instruction set is summarized in more detail.

The instructions that are grouped in this chapter consist of:

1. Load and Store instructions.
2. Arithmetic instructions.
3. Increment and Decrement instructions.
4. Logical instructions.
5. Jump, Branch, Compare and Bit-Test instructions.
6. Shift and Rotate instructions.
7. Register Transfer instructions.
8. Stack instructions.
9. The No Operation instruction.

6502 ADDRESSING MODES

One reason for the increasing popularity of the 6502 microprocessor is the minicomputer-like flexibility it offers in addressing. The 6502 has 13 addressing modes. This is six more addressing modes than the Motorola 6800 microprocessor has, eight more than the Intel 8080 or 8085, three more than the Zilog Z80, and five more than the Texas Instruments 9900. Table 2-3 lists the addressing

Table 2-3. The 6502 Addressing Modes

Mode	Operand Format
Immediate	#aa
Absolute	aaaa
Zero Page	aa
Implied	
Indirect Absolute	(aaaa)
Absolute Indexed, X	aaaa,X or aaaaX
Absolute Indexed, Y	aaaa,Y or aaaaY
Zero Page Indexed, X	aa,X or aaX
Zero Page Indexed, Y	aa,Y or aaY
Indexed Indirect	(aa,X) or (aaX)
Indirect Indexed	(aa),Y or (aa)Y
Relative	aa or aaaa
Accumulator	A

modes, in the order that they are described in this section, and it also gives the assembler form of their operands (which is how the addressing modes are differentiated). In this table, "a" represents a hexadecimal address digit, so "aaaa" is the general form for a four-digit hexadecimal address, such as 34FB.

Immediate Addressing

In immediate addressing, the operand resides in the second byte of the instruction. An immediate operand is specified by placing a # prefix before the operand. For example,

```
LDA  #$3F
```

is an instruction that loads hexadecimal 3F (decimal 127) into the accumulator. All instructions that use immediate addressing are two bytes "long."

Absolute Addressing

Absolute addressing allows the direct addressing of any of the 65,536 memory locations in the address space of the 6502. All instructions that use absolute addressing require three consecutive memory locations for storage. The first byte is the op code of the

instruction, the second and third bytes are the low-order and high-order bytes of the operand address, respectively. For example,

```
LDA  $12B4
```

is an instruction that loads the contents of memory location $12B4 into the accumulator. If memory location $12B4 contains hexadecimal 3F (decimal 127) when the LDA instruction is executed, the accumulator will contain hexadecimal 3F after the LDA instruction is executed. The example instruction looks like this in memory:

Location	Contents	Description
nnnn	$AD	Op code for LDA with absolute addressing
nnnn + 1	$B4	Low-order byte of address
nnnn + 2	$12	High-order byte of address

Zero Page Addressing

Zero page addressing is a form of absolute addressing in which the 6502 microprocessor accesses only the first 256 locations in memory. These are hexadecimal addresses 0000 through 00FF (decimal addresses 0 through 255). Because the high-order byte of a zero page address is always zero, instructions that use zero page addressing are two-byte instructions; the first byte is the op code, the second byte is the low-order byte of a zero page address (00 through FF). The 6502 microprocessor will treat a two-digit operand as a zero page address. For example,

```
LDA  $2A
```

loads the contents of memory location 002A into the accumulator.

Except for two instructions, JMP (Jump) and JSR (Jump to Subroutine), all 6502 instructions that can use absolute addressing can also use zero page addressing. Considering the inherent savings in both storage space and execution time, the zero page should be used, whenever possible, to hold frequently accessed data. ("Zero page" instructions occupy one less byte in memory and take one less cycle to execute than their "absolute address" counterparts.) The zero page is also useful to store temporary data values.

Implied Addressing

Roughly half of the instructions in the 6502 instruction set perform simple workmanlike 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—the 6502 receives enough information from the op code alone—and employ what is called (appropriately) "implied addressing." Some examples are:

Mnemonic	Description
CLC	Clear Carry flag.
DEX	Decrement the X register.
TAX	Transfer Accumulator to X register

All implied addressing instructions occupy one 8-bit memory location.

Indirect Absolute Addressing

Indirect absolute addressing is used by only one 6502 instruction —the Jump (JMP) instruction. The JMP instruction loads the program counter with a new address at which the 6502 is to fetch its next instruction. The JMP instruction can use either absolute addressing or indirect absolute addressing. With absolute addressing, the operand of the JMP instruction is the destination address that is to be put into the program counter. With indirect absolute addressing, the operand of the JMP instruction is the address of the first of two memory locations that *contain* the 16-bit destination address.

An indirect absolute operand is specified by enclosing it in parentheses. For example,

```
JMP  ($0203)
```

causes the program counter to be loaded with the low-order address contained in memory location $0203 and the high-order address contained in memory location $0204. Fig. 2-1 illustrates this example, with $04BC as the final (effective) address and with the instruction stored in memory locations $0110, $0111, and $0112.

You may be asking yourself, "Why all this rigamarole?" If your destination is address $04BC, why not just use absolute addressing to store it in the program counter? The answer is that indirect absolute addressing allows us to work with *variable* destination addresses. For example, if the 6502 is installed in a system in which it must service several peripheral devices, an indirect absolute addressing JMP instruction can be used to access a sequence of instructions appropriate to the peripheral that requires service. In this case, the indirect absolute JMP instruction would always fetch the 16-bit address from the same pair of memory locations, but the 6502 would change the contents of these locations, depending upon which peripheral device requires servicing.

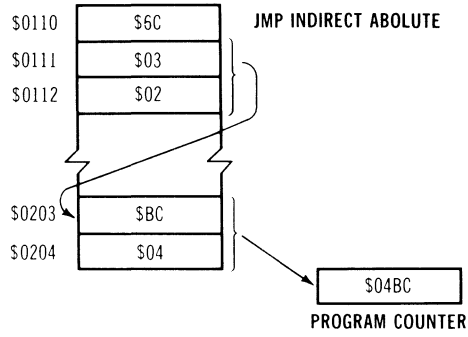


Fig. 2-1. Indirect absolute addressing.

In a data processing application, a single 6502 microprocessor might be accepting data from operators at several keyboards. In this case, the destination address that the 6502 is to jump to depends upon *which keyboard* the 6502 is accepting data from at any particular time. Input from Keyboard No. 1 will be stored in one place, input from Keyboard No. 2 will be stored in another place, and so on. Indirect absolute addressing also allows the effective address to be in R/W memory (changeable memory) even when the program is in ROM or PROM (fixed memory).

Absolute Indexed Addressing

In absolute indexed addressing, the effective address of the operand is computed by adding the contents of the X or Y Register to the absolute address in the instruction. That is,

$$\text{Effective address} = \text{Absolute address} + X$$

or

$$\text{Effective address} = \text{Absolute address} + Y$$

All absolute indexed instructions occupy three consecutive memory locations. Absolute indexed operands are specified by attaching a ",X" or a ",Y" to the address. For example, if the X register contains $03, the instruction

```
LDA  $12B4,X
```

loads the contents of memory location $12B7 (i.e., $12B4 + $03) into the accumulator.

Absolute indexed addressing is particularly useful for accessing data in a list. For this application, you would use the starting address of the list as the operand of the instruction and use the index register (X or Y) to specify the particular element in the list that

you want to access. If you establish a loop in which X is incremented after each access, you can access a series of consecutive elements in the list. Lists will be discussed in more detail in Chapter 4.

Zero Page Indexed Addressing

Zero page indexed addressing is to zero page addressing as absolute indexed addressing is to absolute addressing. With zero page indexed addressing, the effective zero page address of the operand is computed by adding the contents of the X or Y register to the zero page base address contained in the second byte of the instruction.

All zero page indexed instructions are two-byte instructions (one byte less than their absolute-indexed counterparts). Zero page indexed operands are specified by attaching a ",X" or a ",Y" to the address. For example, if the X register contains $03, the instruction

```
LDA  $2A,X
```

loads the contents of location $002D (i.e., $002A + $03) into the accumulator.

Like absolute indexed addressing, zero page indexed addressing offers the potential for list applications. By using zero page indexed addressing, the instruction requires only two memory locations for storage, while absolute indexed addressing instructions require three memory locations for storage. There is also the possibility that the zero page indexed addressing instruction will require one less clock cycle to be executed.

One important point to stress is the effective address is restricted to Page 0 (locations 0 through $FF). If the addition of the index register produces an address larger than $FF, the 6502 will disregard any carry out of the low-order byte. In the previous example, X is restricted to values of $D5 (decimal 213) or less; X = $D5 will produce an effective address of $FF, while X = $D6 will produce a "wrap-around" address of $00.

Indexed Indirect Addressing

Indexed indirect addressing is a combination of two addressing options that have already been discussed in this section—indexed addressing and indirect addressing. Recall that indexed addressing involves adding an index register displacement to a base address contained in the instruction to arrive at a second address—the *effective address* of the data. In indirect addressing, the operand contained in the instruction is the address of the first of two memory locations that contain the address of the data, rather than the data itself.

These two concepts can be combined. With indexed indirect addressing, a displacement in the X register is added to the zero page operand in the instruction to produce an indirect zero page address. The effective absolute address is contained in the memory location addressed by the computed indirect address (low-address byte) and the next consecutive memory location (high-address byte).

Because the operand address is a zero page address, all indexed indirect instructions occupy only two bytes in memory. Further, since the X register is 8 bits long, it can provide a displacement of up to decimal 255, but (as with zero page indexed addressing) the indirect address is restricted to Page 0—locations 0 through $FF. The beauty of indexed indirect addressing is that since the effective address is a 16-bit absolute address, *the full 65K-byte memory space of the 6502 microprocessor can be accessed with a two-byte instruction!* Nothing comes free, though. All the indexed indirect instructions take six cycles to execute, three more than the zero page form of the same instruction and two more than the absolute form.

Indexed indirect operands are of the form (aa,X). For example, if the X register contains $4B, the instruction

```
LDA  ($2A,X)
```

causes the 6502 microprocessor to compute an address of $75 ($2A + $4B) and fetch the effective memory address from zero page locations $75 (low address) and $76 (high address). If memory location $75 contains $B4 and location $76 contains $12, the contents of location $12B4 will be loaded into the accumulator. Fig. 2-2 illustrates this example.

The very nature of indexed indirect addressing, adding a displacement to a zero page base address to fetch an address, hints at one of its uses—selecting one address from a list of addresses in

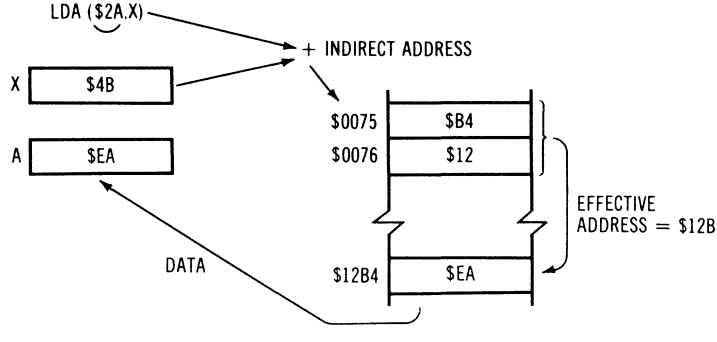


Fig. 2-2. Indexed indirect addressing.

the zero page. Of course, since each address occupies two memory locations, the contents of the X register must be doubled before it is applied to the base address.

Indirect Indexed Addressing

Indirect indexed addressing combines the same two addressing concepts as indexed indirect addressing (indexed addressing and indirect addressing), but applies them in reverse order. In indexed indirect addressing, the index is added to the zero page address in the second byte of the instruction *before* the indirect addressing is performed. This technique is called *pre-indexing*. In indirect indexed addressing, the index is added to the 16-bit memory address *after* the indirect addressing is performed. This technique is called *post-indexing*.

Indirect indexed operands are of the form (aa),Y. For example, if the Y register contains $4B, the instruction

```
LDA  ($2A),Y
```

fetches its base address from zero page locations $2A (low address) and $2B (high address). So, if location $2A contains $B4 and location $2B contains $12, the base address of the data table is $12B4. The value to be loaded into the accumulator is at location $12B4 + $4B, which means that the contents of location $12FF will be loaded into the accumulator. Fig. 2-3 illustrates this example.

Indirect indexed addressing is handy for accessing a certain known element in one of a number of like-structured data tables. For instance this mode might be used in an instruction sequence that is shared by several users. Before executing the indirect in-

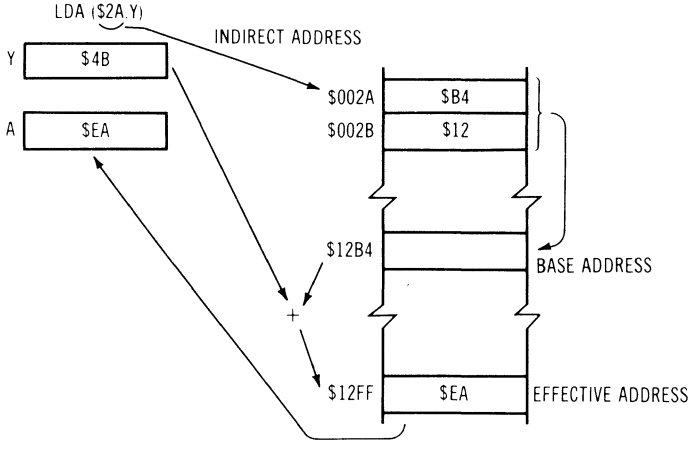


Fig. 2-3. Indirect indexed addressing.

dexed instruction, the calling program of the user stores a unique base address into the zero page operand location (and the next consecutive location), so that the instruction ends up using the correct data table.

Relative Addressing

Relative addressing is just what the name implies. The effective address is specified relative to the address of the next instruction to be executed. That is, the effective address is computed by adding a positive or negative displacement to the current value of the program counter. A positive displacement will address a location following the current instruction (i.e., higher in memory). A negative displacement will address a location preceding the current instruction (i.e., lower in memory).

Relative addressing is used only by the eight *branch instructions* of the 6502 microprocessor. The branch instructions cause program control to transfer forward or backward if a certain condition is met (e.g., if the preceding arithmetic operation produced a zero result); otherwise, execution proceeds to the next sequential instruction. For example, the Branch on Carry Clear (BCC) instruction

```
        BCC  NEXT
NOTNXT  LDA  #$3F
```

will cause the 6502 microprocessor to branch to the instruction at label NEXT if the Carry bit is clear (reset). If the Carry bit is set, the branch is not performed, so the 6502 executes the LDA instruction at label NOTNXT.

All branch instructions occupy two bytes in memory, with the second byte containing the displacement. Being only 8 bits long, the displacement is limited to the range of +127 bytes (forward) to −128 bytes (backward) from the branch instruction.

Accumulator Addressing

The 6502 microcomputer has four instructions that allow shifting or rotating the contents of the accumulator or a memory location one bit position to the right or to the left. If an A operand is specified for these instructions, the 6502 shifts or rotates the accumulator, rather than the memory. Clearly then, accumulator addressing is nothing more than an implied type of addressing that is unique to the four shift and rotate instructions. For example,

```
ASL  A
```

shifts the A contents of the accumulator to the left by one bit position.

LOAD AND STORE INSTRUCTIONS

Having a memory-oriented architecture, the most fundamental operations of the 6502 microprocessor involve moving information into and out of memory. All such transfers are made via three registers—the Accumulator (A) Register, the X Register, and the Y Register.

The process of transferring information from memory into one of the registers of the 6502 microprocessor is called *loading* the information. There are three Load instructions:

Instruction	Description
LDA	Load Accumulator with Memory
LDX	Load X Register with Memory
LDY	Load Y Register with Memory

The source location in memory is unaffected by the load operation, but two flags in the processor status register are altered to provide some information about the value that has been loaded into the register. The Negative flag (N) of the processor status register will be set if Bit 7 (the most-significant bit) of the loaded value is a 1, and will be reset if Bit 7 of the loaded value is a 0. Further, the Zero flag (Z) will be set if the loaded value is 0; otherwise it will be reset. For example,

```
LDA  $1234
```

loads the contents of memory location $1234 into the accumulator.

The process of transferring information from one of the registers of the 6502 microprocessor into memory is called *storing* the information. There are three Store instructions:

Instruction	Description
STA	Store Accumulator in Memory
STX	Store X Register in Memory
STY	Store Y Register in Memory

The store instructions do not change the contents of the source register (A, X, or Y), nor do they alter the processor status register. The simplest way of storing a value in memory is with a Load immediate and Store combination, such as

```
LDA  #00
STA  $21
```

which stores a zero in zero page memory location $0021, effectively clearing that location.

ARITHMETIC INSTRUCTIONS

The 6502 microprocessor has instructions for adding and subtracting both binary-coded and binary-coded-decimal numbers. All add and subtract operations involve two operands, one in the accumulator and the other in memory (or in the second byte of the instruction, if immediate addressing is used).

Representation of Numbers

The 6502 microprocessor can be used to add and subtract both unsigned and signed numbers. The form makes no difference to the microprocessor, but you, as the programmer, should know how to interpret both forms.

In an *unsigned number*, each data bit carries a certain binary weight, according to its position within the number. Data bits are numbered from right to left, with the rightmost bit labeled as Bit 0 and the leftmost bit labeled as Bit 7. Further, the bit numbering scheme has a direct correlation to the binary weights in that Bit 0 has a weight of 2^0 (decimal 1), Bit 1 has a weight of 2^1 (decimal 2), etc. Thus, Bit 7 has a weight of 2^7 (decimal 128). The assignments can be summarized as follows:

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*	**Bit Position**
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	**Binary Weight**
128	64	32	16	8	4	2	1	**Equivalent Decimal Weight**

As you can see, a single byte can represent an unsigned number from 0 (binary 00000000) to decimal 255 (binary 11111111).

In a *signed number*, the seven low-order bits (Bit 0 through Bit 6) represent data, and have the same weights as with unsigned numbers. The most-significant bit (Bit 7) represents the sign of the number. If the number is positive, Bit 7 is a logic 0. If the number is negative, Bit 7 is a logic 1. Positive signed numbers may be within the range of 0 (binary 00000000) to +127 (binary 01111111). Negative signed numbers may be within the range of −1 (binary 11111111) to −128 (binary 10000000).

At this point, some readers may be puzzled and wonder why −1 is represented by binary 11111111, rather than by 10000001. The answer is that negative-signed numbers are represented in their *two's complement* form. The two's complement form was introduced to eliminate the problems that are associated with allowing zero to be represented in two forms, binary combination 00000000 (the positive form) and binary 10000000 (the negative form). Using two's complement, zero is represented by only one form, the binary combination 00000000. To derive the *negative two's complement*

form of a binary number, you simply take the positive form of the number and reverse the sense of each bit. You change each 1 to a 0, and each 0 to a 1, and add 1 to the result. The following example shows the steps required in deriving the binary representation of −32 (in two's complement form).

00100000	$+32_{10}$
11011111	One's complement
+ 1	Add 1
11000000	Two's complement

Decimal Mode Instructions

The 6502 microprocessor has instructions that can cause its internal Arithmetic Logic Unit to operate as either a binary adder or as a decimal adder during addition and subtraction instructions. When operating as a binary adder, the ALU treats both 8-bit operands as binary numbers, with values from 00000000 to 11111111 (hex FF, decimal 255). When operating as a decimal adder, the ALU treats both operands as Binary-Coded Decimal (BCD) numbers, with two 4-bit BCD digits packed into each 8-bit operand.

Since decimal digits range from 0 to 9, BCD digits have values from binary 0000 (hex 0, decimal 0) to binary 1001 (hex 9, decimal 9). Binary combinations 1010 through 1111 are not allowed. Table 2-4 contains the BCD-to-binary relationships.

Table 2-4. The Binary Equivalents of BCD Digits

BCD	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

How do you control the binary and decimal modes of the ALU? It is done with two instructions, *Set Decimal Mode* (SED) and *Clear Decimal Mode* (CLD). The Set Decimal Mode (SED) instruction causes the ALU to function as a decimal (BCD) adder, so that the 8-bit operands of all subsequent add and subtract instructions are treated as packed two-digit BCD numbers. The SED in-

struction also sets the Decimal Mode (D) control bit of the processor status register. The Clear Decimal Mode (CLD) instruction causes the ALU to function as a binary adder, and clears the Decimal Mode (D) bit in the processor status register. At the time when power is applied, *the state of the Decimal Mode (D) bit is undefined,* and it must be either *set or cleared* by the initialization program of your system. The monitor program of the AIM 65 clears the D bit when power is applied.

Addition

Most microprocessors have two add instructions—one that simply adds the operands and another that includes a carry in the addition. The former instruction is used to add single-byte numbers and to add the low-order bytes of two multibyte operands. The latter instruction is reserved for adding the higher-order bytes of two multibyte operands.

The designers of the 6502 microprocessor recognized that since most additions involve multibyte numbers, they could save valuable coding space (and add other functions) in their microprocessor by eliminating the add-without-carry instruction. As a result, the 6502 has only one add instruction. This is Add to Accumulator with Carry (ADC), in which "Carry" is the Carry flag (C) in the processor status register.

Symbolically, the operation of the ADC instruction can be represented as:

$$A = A + M + C$$

where,

A is the accumulator,
M is the memory (or an immediate value),
C is the Carry.

If Carry is set to a logic 1 when the ADC instruction is executed, the addition becomes

$$A = A + M + 1.$$

If Carry is reset to a logic 0 when the ADC instruction is executed, the addition becomes

$$A = A + M + 0.$$

What is the procedure for adding single-byte numbers, or adding the least-significant bytes of two multiple-byte numbers? For these situations, the Carry flag (C) must be reset to zero before performing the addition. The instruction that resets the Carry flag is Clear Carry flag or CLC. With the clear Carry requirement, single-byte add operations usually look like the following:

```
CLC
ADC  $21
```

In this case, the contents of zero page memory location $0021 is added to the accumulator.

There is one situation in which you do not need to precede an ADC instruction with a CLC instruction. If an immediate value is being added to the accumulator (e.g., ADC #36), and the Carry flag has been set by some previous operation, the state of the Carry can be accounted for by using an immediate operand that is one less than the value you want to add. For instance, to add 36 to the accumulator and the Carry is set to one, code an ADC #35. The result will be identical to coding CLC followed by ADC #36.

All ADC instructions are two bytes long except the ADC instructions that use absolute addressing (indexed or unindexed). These are three bytes long. The ADC instruction affects the following four flags in the processor status register:

- The Carry flag (C) is set if the sum of a binary addition exceeds decimal 255 (hex FF), or if the sum of a binary-coded decimal addition exceeds decimal 99 (hex 99); otherwise, it is reset.
- The Zero flag (Z) is set if the sum is zero; otherwise, it is reset.
- The Negative flag (N) is set if Bit 7 of the result is a logic 1; otherwise it is reset. If signed numbers are being added, the N flag is set if the result is negative and reset if it is positive.
- The Overflow flag (V) is set if two like-signed numbers (both positive or both negative) are added and the result exceeds $+127_{10}$ or -128_{10}, which causes Bit 7 of the accumulator to be changed; otherwise it is reset.

For add operations, the status of the N and V flags is pertinent only if signed numbers are being added.

To add multiprecision numbers, simply clear the Carry flag before adding the low-order bytes, and then execute a series of LDA (load), ADC (add), and STA (store) instructions, once for each byte to be added. Example 2-1 shows a routine that adds two double-precision (16-bit) binary numbers. The same routine can be used to add two double-precision decimal (BCD) numbers, by simply inserting a Set Decimal mode (SED) instruction between the CLC instruction and the LDA $20 instruction.

Subtraction

Most microprocessors have two subtract instructions, one that simply subtracts the operands and another that includes a borrow in the subtraction. The former instruction is used to subtract single-

Example 2-1: A Double-Precision Addition Routine

```
;THIS ROUTINE ADDS TWO 16-BIT NUMBERS. ONE NUMBER IS
;STORED IN LOCATIONS $20 AND $21, THE OTHER IS STORED IN
;LOCATIONS $22 AND $23. THE SUM REPLACES THE NUMBER IN
;LOCATIONS $20 AND $21.
  DPADD  CLC          ;CARRY = 0
         LDA  $20     ;ADD LOW-ORDER BYTES
         ADC  $22
         STA  $20
         LDA  $21     ;ADD HIGH-ORDER BYTES
         ADC  $23
         STA  $21
```

byte numbers and, also, to subtract the low-order bytes of two multibyte operands. The latter instruction is reserved for subtracting the higher-order bytes of two multibyte operands.

For the same space-saving reasons given for the single add instruction in the preceding Addition section, the 6502 has only one subtract instruction. This is the Subtract from Accumulator with Borrow (SBC) instruction, in which "Borrow" is contributed by the Carry flag (C) of the processor status register. With borrow always included in the subtraction, you must observe the following rule:

> Preceding a subtraction operation, the Carry flag (C) must be set to a 1 or accounted for in the subtraction. The instruction that sets the Carry flag is Set Carry flag (SEC).

Why set the Carry to 1 if it is to be included in the subtraction? The answer is given in the implementation of the instruction, in which the *complement* of the Carry, rather than its true value, is included in the subtraction. It is given here, in a symbolic representation of the operation of the SBC instruction:

$$A = A - M - \overline{C}$$

where,

A is the Accumulator,
M is the memory,
C is the Carry.

With the Carry set requirement, single-byte subtraction operations look like the following:

```
SEC
SBC  $21
```

In this case, the contents of zero page location $0021 are subtracted from the accumulator. The SBC instruction occupies two bytes in memory if a zero page or an immediate addressing mode is used, and three bytes in memory if an absolute addressing mode is used.

The SBC instruction affects four flags in the processor status register:

- The Carry flag (C) is set if the result is positive or zero, and is reset if the result is negative (indicating a borrow).
- The Zero flag (Z) is set if the result is zero; otherwise, it is reset. Note that if Carry and Zero are both set, the result is zero. If Carry is set and Zero is reset, the result is positive.
- The Negative flag (N) is set if Bit 7 of the result is a logic 1; otherwise it is reset. If signed numbers are being subtracted, the N flag is set if the result is negative and reset if it is positive.
- The Overflow flag (V) is set if two unlike-signed numbers (one number positive, the other number negative) are subtracted, and the result exceeds $+127_{10}$ or -128_{10}. This causes Bit 7 of the accumulator to be changed.

To subtract multiple-precision numbers, you simply set the Carry flag before subtracting the low-order bytes, then execute a series of LDA, and STA instructions, once for each byte to be subtracted. Example 2-2 shows such a subtraction on 16-bit binary numbers. The same routine can be used to subtract two double-precision decimal (BCD) numbers, by simply inserting a Set Decimal mode (SED) instruction between the SEC instruction and the LDA $20 instruction.

Example 2-2: A Double-Precision Subtraction Routine

```
;THIS ROUTINE SUBTRACTS A 16-BIT NUMBER STORED IN LOCATIONS
;$22 AND $23 FROM ANOTHER
;NUMBER STORED IN LOCATIONS $20 AND $21. LOCATIONS $20
;AND $22 HOLD THE LOW-ORDER BYTES OF THE NUMBERS. THE RESULT
;REPLACES THE NUMBER IN LOCATIONS $20 AND $21.

DPSUB  SEC        ;CARRY = 1
       LDA  $20   ;SUBTRACT LOW-ORDER BYTES
       SBC  $22
       STA  $20
       LDA  $21   ;SUBTRACT HIGH-ORDER BYTES
       SBC  $23
       STA  $21
```

The SBC instruction can also be used to negate a number (to put it into two's complement form). You do this by subtracting the positive form of the number from zero. For example, the following routine negates the contents of zero page location $21:

```
SEC        ;CARRY = 1
LDA  #00   ;ACCUMULATOR = 0
SBC  $21   ;SUBTRACT $21 AND
STA  $21   ;  AND RETURN TO MEMORY
```

Signed Number Arithmetic

We have just concluded a brief discussion of the addition and subtraction capabilities of the 6502 microprocessor, but no mention has been made of how the addition or subtraction of unsigned numbers differs from the addition or subtraction of signed numbers. The reason for this is that as far as the actual addition or subtraction operation is concerned, *it makes no difference* whether the numbers are *signed* or *unsigned.* All information given in the preceding Addition and Subtraction sections applies to operands of either form.

That does not mean, though, that every addition and subtraction routine can be used with numbers from both numbering systems, but only that the mechanics of the add and subtract portions of these routines are universal. The signed arithmetic routine may include some additional instructions that will cause a negative result to be treated differently from a positive result. An operation that produces an overflow condition (i.e., the sign bit is altered by the operation) may be treated differently from one in which no overflow occurs.

The resulting sign and overflow status of the final result is reflected by the state of two flags in the processor status register—the Negative (N) and Overflow (V) flags. Signed arithmetic routines often use the final status of these two flags to make some decision about how to process the result. The Negative flag merely reflects the sign of the result (positive or negative), but the Overflow flag indicates whether the accumulator contains a valid result or an invalid result.

The Overflow (V) flag will be set *only* if the result in the Accumulator is invalid. You will recall that the V flag is set if the addition of two like-signed numbers, or the subtraction of two unlike-signed numbers, produces a result more positive than $+127_{10}$ or more negative than -128_{10}; otherwise, the V flag is reset. Once set, the Overflow flag can be reset (cleared) with a special one-byte, two-cycle instruction:

Instruction	Description
CLV	Clear Overflow (V) Flag

The Overflow flag will also be reset automatically at the beginning of the next ADC or SBC instruction.

INCREMENT AND DECREMENT INSTRUCTIONS

The 6502 microprocessor has instructions that increment and decrement the contents of the X register, the Y register, or a memory location.

Increment/Decrement Registers

In the 6502 Addressing Modes section of this chapter, we discussed the use of the X and Y registers as index registers. These two registers can also function as general-purpose counters in a variety of applications.

The designers of the 6502 microprocessor made it easy to access consecutive memory locations, and to count up or down by one, by providing increment and decrement instructions for both of these general-purpose registers. They are:

Instruction	Description
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
INX	Increment Index X by One
INY	Increment Index Y by One

These implied address instructions each occupy one byte in memory and take two cycles to execute. Further, they affect two flags in the processor status register:

- The Negative flag (N) is set if the register contains a negative result after being incremented or decremented (Bit 7 = 1); otherwise, N is reset.
- The Zero flag (Z) is set if the result of the increment or decrement operation is zero; otherwise, it is reset.

Example 2-3 shows a routine that copies the contents of six consecutive bytes in memory (starting at location $20) into another portion of the memory (starting at location $0320). In this routine, the X register acts as an index register and the Y register acts as the byte counter. The final instruction, BNE, has not been described yet, but all it does is make the 6502 loop back and load another byte (via the $20,X instruction at label NXTBYT) until the byte counter—the Y register—has been decremented to zero.

Example 2-3: A Six-Byte Memory Move Routine

```
;THIS ROUTINE COPIES A SIX-BYTE BLOCK OF MEMORY,
;STARTING AT LOCATION $20, INTO ANOTHER PART OF
;MEMORY, STARTING AT LOCATION $0320.

        LDX   #00       ;INDEX = 0
        LDY   #06       ;BYTE COUNT = 6
NXTBYT  LDA   $20,X     ;LOAD NEXT BYTE
        STA   $0320,X   ;STORE NEXT BYTE
        INX             ;INCREMENT INDEX
        DEY             ;DECREMENT BYTE COUNT
        BNE   NXTBYT    ;LOOP UNTIL ALL BYTES COPIED
```

Increment/Decrement Memory

In some applications, counters are maintained in memory, rather than in the X or Y registers. Two instructions allow you to perform a simple increment-by-1 or decrement-by-1 operation on memory without using an ADC or SBC instruction. They are:

Instruction	Description
DEC	Decrement Memory by One
INC	Increment Memory by One

Both instructions permit four addressing modes—absolute, zero page, absolute indexed X, and zero page indexed X. The absolute-addressed instructions occupy three bytes in memory, and the zero page-addressed instructions occupy two bytes in memory. Further, these instructions affect the same flags in the processor status register—Negative (N) and Zero (Z)—as the register increment and decrement instructions.

Unlike the arithmetic add and subtract instructions (ADC and SBC), the INC and DEC instructions do not affect the Carry flag. Does this make them worthless for multibyte counting in memory? No! It simply means that if a multibyte counter is being incremented, you must check the Z flag (rather than the C flag) to determine whether or not to increment the next higher byte; if Z is set, increment. Similarly, if a multibyte counter must be decremented, you must check the N flag to determine whether or not to decrement the next higher byte; if the DEC instruction has caused N to switch from a logic 0 state of a logic 1 state, decrement.

LOGICAL INSTRUCTIONS

There will be situations in which you will want to examine just one or more bits in a memory location or register, rather than the entire 8-bit byte. For instance, you may want to test certain bits in a status byte, or perhaps operate on the lower four bits or the upper four bits of some data byte. The logical instructions of the 6502 microprocessor are useful for these situations.

There are three logical instructions. All operate on the accumulator, using the contents of a memory location (or an immediate value) specified in the operand. The logical instructions are:

Instruction	Description
AND	AND Memory with Accumulator
EOR	Exclusive-OR Memory with Accumulator
ORA	OR Memory with Accumulator

All three logical instructions occupy two bytes in memory if a zero page or immediate addressing mode is used, and three bytes in memory if an absolute addressing mode is used.

Further, all three logical instructions affect two flags in the processor status register:

- The Negative flag (N) is set if the result is negative (Bit 7 = 1); otherwise, it is reset.
- The Zero flag (Z) is set if the result is zero; otherwise, it is reset.

AND Instruction

The AND Memory with Accumulator (AND) instruction is primarily used to filter, mask, or strip out (set to zero) certain bits in the accumulator so that some form of processing can be performed on the remaining bits. For each bit position in which both memory and the accumulator contain a 1, the bit in the accumulator is set to 1; otherwise that bit is reset to 0. Table 2-5 summarizes the AND conditions.

Table 2-5. Logical AND Operation

Memory Bit	Accumulator Bit	Result Bit in Accumulator
0	0	0
0	1	0
1	0	0
1	1	1

The AND instruction is best suited for testing selected accumulator bits for a 1 value (to check a status byte, perhaps, to find out which bits are "on"), or to mask out bits that are of no interest in a particular program application. For example, in the American Standard Code for Information Interchange (ASCII), the characters 0 to 9 are assigned the values listed in Table 2-6. Assume that the most-significant bit (MSB) is always a logic 0.

If the four most-significant bits are masked out of the ASCII values so that they are reset to zero, the binary-coded decimal (BCD) value for each character will remain in the four least-significant bits. Examine the number 5.

$$5_{10} = \text{ASCII } 0011\mathit{0101}_2 \text{ and } 0000\mathit{0101}_2 = 05 \text{ (BCD)}$$

How can the four most-significant bits be masked to logic 0? It can be easily done with a logical AND. If you refer to Table 2-5, you will see that any bit that is ANDed with a 0 will be cleared to 0 and any bit that is ANDed with a 1 will retain its original value. Therefore, ASCII 5 in the accumulator can be easily converted to BCD by

Table 2-6. The ASCII Values for Characters 0 Through 9

Character	ASCII		BCD
	Hex	Binary	
0	30	00110000	0000
1	31	00110001	0001
2	32	00110010	0010
3	33	00110011	0011
4	34	00110100	0100
5	35	00110101	0101
6	36	00110110	0110
7	37	00110111	0111
8	38	00111000	1000
9	39	00111001	1001

ANDing its four most-significant bits with 0s and its four least-significant bits with 1s. This can be done with the instruction AND #$0F. What will be contained in the accumulator when the instructions

```
LDA   #$38
AND   #$0F
```

are executed? The accumulator will contain 00001000, or BCD 8.

Exclusive-OR Instruction

The Exclusive-OR Memory with Accumulator (EOR) instruction is primarily used to determine which bits differ between two operands, but it can also be used to complement selected accumulator bits. Each bit position that contains a 1 in either memory or the accumulator (but not both) is set to 1; all other bit positions are cleared to 0. Table 2-7 summarizes the EOR conditions.

Table 2-7. Logical Exclusive-OR Operation

Memory Bit	Accumulator Bit	Result Bit in Accumulator
0	0	0
0	1	1
1	0	1
1	1	0

The way the EOR operates on a particular bit can be likened to a radio operator waiting for two radio messages. If neither message arrives, it is a zero night. If either message arrives, the night is a success. However, if *both* signals arrive at the same time (cancelling each other), it also results in a zero night. As an example, the instruction

```
EOR   #$0F
```

will complement the four least-significant bits of the accumulator and clear to zero the four most-significant bits of the accumulator. The EOR instruction is also used to determine whether two values are identical; EOR will set the Zero flag (Z) if, and only if, the contents of memory are identical to the contents of the accumulator.

OR Instruction

The OR Memory with Accumulator (ORA) instruction produces a logic 1 result in each accumulator bit position for which both memory and accumulator contain a logic 1. Table 2-8 summarizes the ORA conditions. The ORA instruction is usually used to set bits to a logic 1. For example,

```
ORA  #80
```

sets the most-significant bit (Bit 7) to a logic 1, and leaves all other bits unchanged.

Table 2-8. Logical OR Operation

Memory Bit	Accumulator Bit	Result Bit in Accumulator
0	0	0
0	1	1
1	0	1
1	1	1

JUMP, BRANCH, COMPARE, AND BIT-TEST INSTRUCTIONS

Up to this point, all of the program examples (except Example 2-3) have been straight line programs; instructions were executed in the order in which they appeared in the program. Control has not been transferred to another section of a program based on the results received by executing an instruction or a series of instructions. The next group of instructions that we will discuss demonstrate how program execution can be transferred from one section of a program to another section, and why this is useful.

The Jump Instruction

Whether you know it or not, you are already familiar with the basic concepts of jump operations. Have you ever read a set of instructions for something or other and come across a direction like "Jump to Step 5"? Well, that is a jump operation. Similarly, when an income tax form says "Go to Step 36a," that is also a jump instruction. As in these everyday situations, the jump instruction of the 6502 microprocessor causes the next operation to take place at a point other than the next consecutive memory location. A further

similarity with everyday "jump situations" is that the jump instruction of the 6502 is *unconditional;* it occurs every time the 6502 encounters it in a program.

The 6502 jump instruction, Jump, has the mnemonic JMP. The operand of a JMP instruction is frequently a label, so you will often see JMP instructions being used in the following context:

```
        .
        .
        .
        ADC   $20
        INX
        JMP   THERE
HERE    STA   $24
        .
        .
        .
THERE   SEC
        SBC   $22
        .
        .
        .
```

In the preceding example, the program increments the X register (via INX), and then jumps to the instruction assigned to THERE. It executes that instruction (SEC), then the SBC $22 instruction, and then whatever follows. Will the STA $24 instruction located at label HERE ever be executed? Yes, but only if an instruction in some other portion of the program transfers control (using perhaps a JMP HERE instruction) to that location.

The JMP instruction is used typically to skip over a group of instructions that are executed under some alternative set of conditions, or a group of instructions that are executed during some other part of the program. The JMP instruction can operate with absolute addressing or with indirect absolute addressing. Since both modes are absolute, the jump can be to any place in memory. We might add that although the JMP THERE instruction in the preceding example caused a jump to a location following it in memory, it could just as easily have jumped to a location preceding it in memory.

The JMP instruction occupies three bytes in memory and takes three cycles to execute with absolute addressing and five cycles to execute with absolute indirect addressing. Further, the JMP instruction does not affect any flags in the processor status register.

The Branch Instructions

Branch instructions, like the JMP instruction, cause the program execution to be transferred to a specific memory location. Whereas

the JMP instruction transfers control to some absolute address in memory, the branch instructions *transfer control a specified number of memory locations forward or backward from the next instruction* after the branch instruction. Another dissimiliarity between the JMP and branch instructions is that the branch instructions are *decision-making* instructions. Each of the branch instructions tests the status of a single flag in the processor status register. If the state of the flag meets the requirements specified by the branch instruction, the instruction is executed; otherwise, execution continues with the next consecutive instruction in the program. Branch instructions, and the flags they test, are summarized in Table 2-9.

Table 2-9. The Branch Instructions

Instruction	Description	Causes a Branch If
BCC	Branch on Carry Clear	Carry = 0
BCS	Branch on Carry Set	Carry = 1
BEQ	Branch on Result Equal to Zero	Zero (Z) = 1
BNE	Branch on Result Not Equal to Zero	Zero (Z) = 0
BMI	Branch on Result Minus	Negative (N) = 1
BPL	Branch on Result Plus	Negative (N) = 0
BVS	Branch on Overflow Set	Overflow (V) = 1
BVC	Branch on Overflow Clear	Overflow (V) = 0

As you can see from Table 2-9, the branch instructions are based on the status of four flags in the processor status register—Carry (C), Zero (Z), Negative (N), and Overflow (V). In general:

- The Carry flag reflects conditions where the result cannot be contained in the specified resulting register or memory location. It is affected by the arithmetic instructions, the logical shift instructions, the compare instructions, and the SEC and CLC instructions.
- The Zero and Negative flags are set by conditions in which the result is zero or has Bit 7 = 1, respectively. These flags can be affected by about half of the 6502 instructions.
- The Overflow (V) flag is affected by the arithmetic instructions (ADC and SBC), and by the BIT and CLV instructions.

After reading that the branch instruction transfers are taken relative to the branch instruction, you may have already guessed that these instructions operate in only one addressing mode, *relative addressing*. For this reason, the branch can span up to 127 bytes forward, or 128 bytes backward, from the branch instruction. This does not impose as much of a restriction as you might think, because you can always combine a JMP instruction with the branch instruction to branch anywhere in memory. For example, here is

how a program might execute a branch on the Carry-set condition to an instruction (at CSET) that is more than 127 locations past, or 128 locations ahead of, the BCC instruction:

```
        BCC  CCLEAR  ;GO TO CCLEAR ON CARRY = 0
        JMP  CSET    ;GO TO CSET ON CARRY = 1
CCLEAR  LDA  $20
```

Table 2-10 gives the hexadecimal equivalents for the full range of branches, both forward and backward. If you have an AIM 65 and are using the mnemonic entry mode, you can ignore this table

Table 2-10. Hexadecimal Operands for Branch Instructions

Forward Relative Branch Table

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

Backward Relative Branch Table

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

and enter the absolute (16-bit) address itself, rather than the relative displacement. Otherwise, here is how you can use Table 2-10. If you want to code a Branch on Carry Clear (BCC) instruction that will branch 72 bytes backward if the condition is met, you would look up 72 in the Backward Relative Branch Table. Position 72 is in column 8 and row B. The column gives the least-significant hexadecimal digit (LSD) value and the row gives the most-significant hexadecimal digit (MSD) value, so the BCC instruction should be coded like this:

```
BCC  $B8
```

All branch instructions occupy two bytes in memory. (The first byte holds the op code, and the second byte holds the relative displacement.) These instructions are executed in two cycles if the condition is not met and three cycles if the condition is met (four cycles if the branch crosses a page boundary).

If we assume that a given condition is met 50% of the time, then the 6502 microprocessor will average 2.5 cycles to test a single status flag. The coding of two branch instructions to test a combination of two flags will be executed on the average in 3.75 cycles.

The following are a few examples of branch instructions:

1. The sequence

```
ADC  $24
BCS  TOOBIG
```

   branches to label TOOBIG if the add operation produces a carry.

2. The sequence

```
SBC  $24
BCC  TOOSML
```

   branches to label TOOSML if the subtract operation produces a borrow.

3. The sequence

```
ADC  $24
BEQ  DONE
```

   branches to DONE if the add operation produces a zero result in the accumulator.

4. The sequence

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

   will loop (to LOOP) until the X register has been decremented to zero. You will see this sort of sequence in many programs that use the X or Y register as a counter.

5. The sequence

```
SBC  $24
BMI  MINUS
```

   branches to MINUS if the subtract operation produces a negative result in the accumulator.

Note that except for Example 4, each of the preceding examples employs a bit of inverted logic in which the more expected circum-

stance is executed if the branch *fails*, rather than *succeeds*. That is, the instructions that immediately follow the branch instruction represent the usual flow of the program, and the instructions at the branch target are executed as an exception. (Parents use this same approach when they tell a daughter or son, "If it looks like rain tomorrow, take your raincoat to school," or "If you say one more word, you're going to bed.") In fact, this approach should be used whenever possible, because a branch test that fails takes one less cycle to execute than a branch test that succeeds.

To this point, our branch instruction examples base the branch test on the state of a status flag that reflects the result in an arithmetically altered register. There are many situations, however, where you would like to base a branch decision on the contents of a register or memory location in its unaltered state. For example, you might want to branch if a certain location contains 0, or −3, or a number larger than +6. The 6502 microprocessor has three instructions for that purpose. These instructions are called *compare* instructions.

Compare Instructions

The compare instructions set or clear three of the status flags (Carry, Zero and Negative) that can be tested with branch instructions, *without altering the contents of the operand.* There are three compare instructions:

Instruction	Description
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y

The CMP instruction supports eight different addressing modes, the same ones supported by the ADC and SBC instructions. Since the X and Y registers function primarily as counters and indexes, the CPX and CPY instructions do not require this elaborate addressing capability and operate with just three addressing modes (immediate, absolute, and zero page).

The compare instructions subtract an immediate value or the contents of a memory location from the addressed register, but do not save the result in the register. The only indications of the result are the states of three status flags—Negative (N), Zero (Z), and Carry (C). The combination of these three flags indicate whether the register contents are *less than*, *equal to* (the same as) or *greater than* the operand "data" (the immediate value or contents of the addressed memory location). Table 2-11 summarizes the result indicators for the compare instructions.

Table 2-11. Compare Instruction Results

	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

*Valid only for "two's complement" compare.

The compare instructions serve only one purpose; they provide information that can be tested by a subsequent branch instruction. For example, to branch if the contents of a register are less than an immediate or memory value, you would follow the compare instruction with a Branch on Carry Clear (BCC) instruction, as shown by the following:

```
        CMP   $20     ;ACCUMULATOR LESS THAN LOCATION $20?
        BCC   THERE   ;YES, BRANCH TO THERE
THERE                 ;NO, CONTINUE EXECUTION HERE
        .
        .
        .
HERE                  ;EXECUTE THIS IF A IS LESS THAN LOCATION $20
        .
        .
        .
```

Table 2-12 lists the branch instruction(s) that should follow the compare instruction, for each register/data relationship. In this table, THERE represents the label of the instruction executed if the branch test succeeds and HERE represents the label of the instruction executed if the branch test does not succeed. Besides comparing a memory location and a register, the compare instructions are handy for comparing one memory location with another, by loading one into a register (A, X, or Y).

Example 2-4 contains a routine that tests whether the contents of two memory locations are identical, and sets a flag in memory

Table 2-12. Use of Branch Instructions with Compare

	Follow compare instruction with	
To Branch If	**For unsigned numbers**	**For signed numbers**
Register is less than data	BCC 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	BCC THERE	BMI THERE
	BEQ THERE	BEQ THERE
Register is greater than or equal to data	BCS THERE	BPL THERE

Example 2-4: Testing Two Locations for Equality

```
;THIS ROUTINE SETS MEMORY LOCATION $22 TO "ONE" IF THE CONTENTS
;OF LOCATIONS $20 AND $21 ARE EQUAL, AND TO "ZERO" IF OTHERWISE.

        LDX   #00    ;INITIALIZE FLAG TO ZERO
        LDA   $20    ;GET FIRST VALUE
        CMP   $21    ;IS SECOND VALUE IDENTICAL?
        BNE   DONE
        INX          ;YES, SET FLAG TO ONE
DONE    STX   $22    ;STORE FLAG
```

to so indicate. Example 2-5 contains another memory-to-memory comparison routine. It stores the larger of the two values in the higher memory location. Example 2-6 contains a register-to-constant comparison routine in which two branch instructions are used with

Example 2-5: Arranging Two Numbers in Order of Value

```
;THIS ROUTINE ARRANGES TWO NUMBERS IN LOCATIONS $20 AND $21 IN
;ORDER OF VALUE, WITH THE LOWER-VALUED NUMBER IN LOCATION $20.

        LDA   $21    ;LOAD SECOND NUMBER INTO ACCUMULATOR
        CMP   $20    ;COMPARE THE NUMBERS
        BCS   DONE   ;DONE IF FIRST IS LESS THAN OR EQUAL
                     ;  TO SECOND,
        LDX
        STA   $20    ;OTHERWISE SWAP THEM
        STX   $21
DONE    .
        .
        .
```

one compare instruction, so that the "less than," "equal to," and "greater than" conditions are tested. Since the branch instructions do not affect any flags in the processor status register, BNE GT3 is basing its branch decision on the same flag that the BCS EQGT3 based its branch decision on!

Example 2-6: A Three-Way Decision Routine

```
;THIS ROUTINE STORES THE CONTENTS OF THE ACCUMULATOR INTO LOCATION
;$20, $21 OR $22, DEPENDING UPON WHETHER THE ACCUMULATOR HOLDS
;A VALUE LESS THAN 3, EQUAL TO 3, OR GREATER THAN 3, RESPECTIVELY.
        CMP   #03    ;COMPARE ACCUMULATOR TO 3
        BCS   EQGT3
        STA   $20    ;ACCUMULATOR LESS THAN 3
        JMP   DONE
EQGT3   BNE   GT3
        STA   $21    ;ACCUMULATOR EQUAL TO 3
        JMP   DONE
GT3     STA   $22    ;ACCUMULATOR GREATER THAN 3
DONE    .
        .
        .
```

Thus far, our discussion has concentrated on the CMP instruction, which compares the accumulator with memory. Of what value are the other compare instructions (CPX and CPY)? Their primary value is to monitor the contents of X or Y when these registers are being employed as count-up counters. In these cases, CPX or CPY is used to compare the count against a maximum value of 255_{10} in memory. Example 2-7 is a routine that will move up to 256 consecutive memory bytes, with the CPX instruction doing the "all bytes moved" check each time a byte is moved. You may have noted that Example 2-7 is an expanded version of the six-byte move routine given in Example 2-3.

Example 2-7: A Multiple-Byte Move Routine

```
;THIS ROUTINE MOVES UP TO 256 BYTES OF MEMORY, STARTING AT
;LOCATION $20, TO ANOTHER PORTION OF MEMORY, STARTING AT LOCATION
;$0320. THE BYTE COUNT IS CONTAINED IN LOCATION $1F.

        LDX   #00      ;INDEX = 0
NXTBYT  LDA   $20,X    ;LOAD NEXT BYTE
        STA   $0320,X  ;STORE NEXT BYTE
        INX            ;INCREMENT INDEX
        CPX   $1F      ;ALL BYTES MOVED?
        BNE   NXTBYT   ;IF NOT, MOVE NEXT BYTE
```

The compare instructions compare two "entire" 8-bit values. There are situations, however, when you will need to test one or more individual bits in a memory location. It has already been shown how this can be done by masking out the unwanted bits with an AND instruction. However, in the process of masking-out the unwanted bits, the AND instruction destroys the mask contained in the accumulator. Certainly, the mask could be reloaded, but this requires additional processor time and additional instructions. The same job can be done, without altering the accumulator, by executing a *BIT instruction.*

The BIT Instruction

The 6502 microprocessor has an instruction that allows you to test the value of individual bits in memory, by logically ANDing the contents of the memory location with a bit selection mask in the accumulator. This instruction, Test Bits in Memory with Accumulator (BIT), alters neither accumulator nor memory, and, like the compare instructions, records status information in the processor status register. Here is how the status flags are affected.

- The Negative flag (N) receives the initial (un-ANDed) value of Bit 7 of the memory location being tested.
- The Overflow flag (V) receives the initial (un-ANDed) value of Bit 6 of the memory location being tested.

- The Zero flag (Z) is set if the AND operation generates a zero result; otherwise, it is reset.

The BIT instruction supports only absolute and zero page addressing. It is a three-byte, four-cycle instruction in the absolute mode and a two-byte, three-cycle instruction in the zero page mode.

It is important to note that *only* the Zero flag (Z) reflects the result of the simulated AND operation; the Negative and Overflow flags (N and V) indicate the values of the two high-order memory bits in their unaltered state. In reality, then, the BIT instruction performs two very distinct types of test functions. These test functions will be discussed individually, beginning with the AND function.

From the previous description of the AND instruction, you will recall that for each bit position in which both memory and the accumulator contain a 1, the AND instruction sets the corresponding accumulator bit to 1; otherwise that bit is reset to 0. The BIT instruction performs the same operation as the AND instruction, but it does not enter the ANDed result in the accumulator. Further, whereas the AND instruction affects two status flags (N and Z), and both flags reflect the post-AND status, the BIT instruction affects three flags (N, V, and Z), but only the Z flag reflects the post-AND status.

The bit-testing sequence in a program is comprised of three instructions—an LDA instruction (usually with immediate addressing) that loads the bit selection mask into the accumulator, a BIT instruction that specifies the absolute or zero page location to be tested, and a BEQ or BNE branch instruction. The bit selection mask must have a logic 1 in the bit position to be tested and a logic 0 in all other bit positions. Example 2-8 shows a program sequence that tests the state of Bit 2 of memory location $2340, and waits for it to become a logic 0. Bit 2 might represent a motor or a relay which must be off before the program can proceed to its next operation. Note that the sequence in Example 2-8 could just as easily test for a logic 1 in Bit 2, if the BNE LOOP were a BEQ LOOP.

Can a BIT instruction test more than one bit at a time? Yes, but with some limitations. When two or more bits are set in the accu-

Example 2-8: Waiting for a Memory Bit to Become Logic 0

```
        LDA  #04    ;MASK OFF ALL BITS BUT BIT 2
LOOP    BIT  $2340  ;BIT 2 = 0?
        BNE  LOOP   ;KEEP CHECKING UNTIL IT IS
HIT      .          ;  THEN CONTINUE HERE
         .
         .
```

mulator mask, a subsequent BIT instruction will set the Zero flag (Z) only if *both* test bits are logic 0 in memory, but will reset Z if *either or both* test bits are logic 1 in memory.

These multiple-bit testing limitations disappear, however, if only one of the test bits is in the Bit 0 to Bit 5 range, and the other test bit(s) are either or both of the Bit 6 and 7 positions. Since Bits 6 and 7 provide AND-independent status indications, they can be tested with their own branch instructions (BVS and BVC for Bit 6, BMI and BPL for Bit 7).

Example 2-9 is similar to Example 2-8, but instead of waiting for just one memory bit to become logic 0, we are waiting for any of three memory bits (Bits 2, 6, and 7) to become logic 0. Note that the mask only selects Bit 2 (LDA #04) for ANDing; if we had mask-selected Bits 2, 6, and 7 (LDA #$C4), the Z flag would be set only if *all three bits* were logic 0.

Example 2-9: Waiting for Any of Three Memory Bits to Become Logic 0

```
        LDA   #04     ;SELECT BIT 2 FOR TESTING WITH MASK
LOOP    BIT   $2340   ;TEST MEMORY
        BEQ   HIT     ;BRANCH ON BIT 2 = 0
        BVC   HIT     ;BRANCH ON BIT 6 = 0
        BMI   LOOP    ;LOOP IF BIT 7 = 1
HIT     .             ;CONTINUE HERE WHEN 2, 6, OR 7 = 0
        .
        .
```

We have just seen how the AND-independent nature of the Negative (N) and Overflow (V) bits allows you to test several memory bits with a single BIT instruction. Although that facility is useful in many program applications, most programmers will be even more pleased by the fact that the N and V flag design of the BIT instruction allows them to obtain important information about the contents of a memory location *without loading it into a register.*

Since the BIT instruction is restricted to absolute and zero page addressing, it is of little use if you are testing elements of tables, or any other indexed or indirect-accessed data. However, the BIT instructions can often replace a compare instruction for data at known addresses. The easy access to Bits 6 and 7 (via the V and N flags) makes these bits ideal for storing status information. And, with two bits, you get four testable combinations of status; binary 00, 01, 10, and 11.

Perhaps the most widespread use of the N and V flags relates to the *interrupt request scheme* of the input/output structure of the 6500 system. An interrupt request is simply the way for a peripheral device to tell the 6502 microprocessor, "I am ready to send data," or "I am ready to receive data." Interrupt requests are often stored in the most-significant bit (Bit 7) of an interrupt status byte

in memory, so the presence of the Bit 7 in the N flag of the BIT instruction is an indicator of the interrupt status, not the sign of a number. Further, the presence of Bit 6 in the V flag is designed to accommodate the 6520 Peripheral Interface Adapter (PIA), which has interrupt sense bits in the Bit 6 and 7 positions of its status word. Further discussion of input/output is reserved for Chapters 7, 8, and 9.

SHIFT AND ROTATE INSTRUCTIONS

The 6502 microprocessor has four instructions that cause the 8-bit contents of an operand (a memory location or the accumulator) to be displaced 1 bit position to the left or to the right. Two of these instructions "shift" the operand, the other two "rotate" the operand.

For all four instructions, the Carry flag of the processor status register acts as a "ninth bit" extension of the operand in that it receives the value of the bit that is displaced out of one end of the accumulator or memory location (Bit 0 for a right shift, Bit 7 for a left shift). In a *shift* operation, the vacated bit position at the opposite end of the operand (Bit 7 for a right shift, Bit 0 for a left shift) is reset to "0." In a *rotate* operation, the vacated bit position at the opposite end of the operand receives the prerotate value of the Carry flag. These are the shift and rotate instructions:

Instruction	Description
ASL	Accumulator Shift Left
LSR	Logical Shift Right
ROL	Rotate Left
ROR	Rotate Right

The operations of these instructions are illustrated in Fig. 2-4.

As previously mentioned, the shift and rotate instructions can operate on either the accumulator or a location in memory. To operate on the accumulator, you simply put an "A" in the operand field of the instruction, like this:

```
ASL  A
```

Operations on memory can be conducted in any of four addressing modes—absolute, zero page, zero page indexed X, and absolute indexed X. The following are the ASL instruction formats for each mode:

```
ASL  $1234    (Absolute)
ASL  $2A      (Zero Page)
ASL  $2A,X    (Zero Page Indexed, X)
ASL  $1234,X  (Absolute Indexed, X)
```

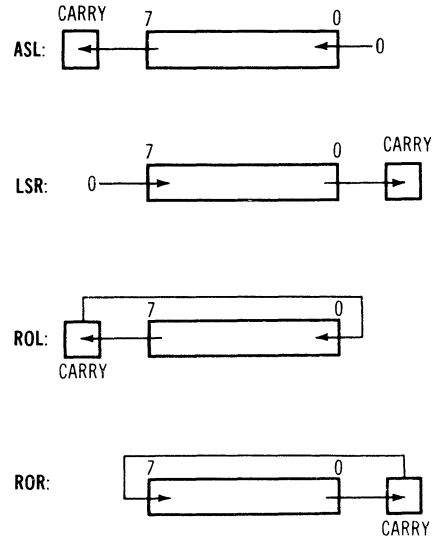


Fig. 2-4. Diagram of the Shift and Rotate instructions.

These instructions occupy one byte in memory for accumulator addressing, two bytes for either zero page addressing mode, and three bytes for either absolute addressing mode.

In addition to their effect on the Carry flag (see Fig. 2-4), the shift and rotate instructions affect two other flags in the processor status register.

- ASL, ROL, and ROR cause the Negative flag (N) to be set if Bit 7 of the shifted result is set to logic 1; otherwise, it is reset. The LSR instruction always causes the Negative flag to be reset, since it shifts a "0" into Bit 7.
- The Zero flag (Z) is set if the shifted result is 0; otherwise, it is reset.

To show how these instructions work, consider an operand that contains hexadecimal 34 (binary 00110100, decimal 52). The Carry flag is set to 1. Here is how the operand and Carry flag would be altered for each of the four shift and rotate instructions:

Carry	Bit Position								
Flag	7	6	5	4	3	2	1	0	
1	0	0	1	1	0	1	0	0	Before shift (hex 34, decimal 52)
0	0	1	1	0	1	0	0	0	After ASL (hex 68, decimal 104)
0	0	0	0	1	1	0	1	0	After LSR (hex 1A, decimal 26)
0	0	1	1	0	1	0	0	1	After ROL (hex 69, decimal 105)
0	1	0	0	1	1	0	1	0	After ROR (hex 9A, decimal 154)

One of the most common applications for shift and rotate instructions is in multiplying and dividing numbers. With a closer examination of the preceding example, you will see why. Look at the decimal equivalents. The unshifted operand has a decimal value of 52. After being left-shifted by ASL, its decimal value is 104. After being right-shifted by LSR, its decimal value is 26. Observe that *each left shift multiplies the operand by two, each right shift divides the operand by two.*

Shift and rotate instructions can also be used to perform serial-to-parallel and parallel-to-serial conversions when the 6502 is communicating with serial-based peripheral devices such as teletypewriters and CRTs.

Shifting Unsigned Numbers

Single-byte numbers can be shifted using only the shift instructions, ASL and LSR. Multiple-byte numbers require a combination of these shift instructions and the rotate instructions, ROL and ROR. In multiple-byte shift operations, the Carry flag is used to propagate bit values that have been displaced out of previously shifted bytes.

Left shifts must operate on multiple-byte numbers in right-to-left order, wherein the low-order byte is shifted first, with an ASL instruction. Each remaining, higher-order byte is shifted with an ROL instruction. Consider, for example, a 24-bit (three-byte) unsigned number that is stored in locations $20 (low-order byte), and $21 and $22 (high-order byte). This number can be left-shifted with the following sequence of instructions:

```
ASL   $20  ;SHIFT LOW-ORDER BYTE
ROL   $21  ;SHIFT MIDDLE BYTE
ROL   $22  ;SHIFT HIGH-ORDER BYTE
```

Right shifts must operate on multiple-byte numbers in left-to-right order, wherein the high-order byte is shifted first, with an LSR instruction. Each remaining lower-order byte is shifted with an ROR instruction. The right-shift routine for the three-byte number starting at location $20 is:

```
LSR   $22  ;SHIFT HGH-ORDER BYTE
ROR   $21  ;SHIFT MIDDLE BYTE
ROR   $20  ;SHIFT LOW-ORDER BYTE
```

Example 2-10 is a routine that left-shifts an unsigned number of variable length, and can accommodate numbers from two bytes up to 256 bytes long. This routine brings the byte count out of memory, and puts it into the Y register. Actually, it could be retained in memory, and decremented with the DEC instruction, but DEY

Example 2-10: A Left-Shift Routine for Multiple-Precision Unsigned Numbers

```
;THIS ROUTINE LEFT-SHIFTS A MULTIPLE-PRECISION UNSIGNED NUMBER
;STORED IN MEMORY STARTING AT LOCATION $30. THE LENGTH OF THE
;NUMBER, IN BYTES, IS CONTAINED IN LOCATION $2F.
        LDY   $2F       ;LOAD BYTE COUNT INTO Y
        ASL   $30       ;SHIFT LOW-ORDER BYTE
        LDX   #01       ;BYTE INDEX = 1
        DEY             ;DECREMENT BYTE COUNT
NXTBYT  ROL   $30,X     ;SHIFT NEXT BYTE
        INX             ;UPDATE BYTE INDEX
        DEY             ;  AND BYTE COUNT
        BNE   NXTBYT    ;LOOP UNTIL ALL BYTES SHIFTED
```

is a one-byte instruction that takes two cycles to execute, whereas DEC is a two-byte instruction that takes five cycles to execute. If you take the time to go through a speed and space comparison of the two approaches, you will discover that the LDY $2F instruction and two DEY instructions will occupy four bytes in memory and will take *seven* cycles to execute. Conversely, two DEC instructions occupy four bytes in memory and take *ten* cycles to execute. Thus, the memory approach is 30% slower than the Y register approach, and that ignores the fact that one of the decrement instructions falls within the NXTBYT loop!

Shifting Signed Numbers

Our four shift and rotate instructions perform what are known as "logical" shifts. That is, they treat the operand as a pattern of 8 bits, without regard to sign. Consequently, if a signed number is right-shifted, the sign bit will be displaced 1 bit position to the

Example 2-11: A Left-Shift Routine for Multiple-Precision Signed Numbers

```
;THIS ROUTINE LEFT-SHIFTS A MULTIPLE-PRECISION SIGNED NUMBER
;STORED IN MEMORY STARTING AT LOCATION $30. THE LENGTH OF
;THE NUMBER, IN BYTES, IS CONTAINED IN LOCATION $2F.
        LDY   $2F       ;LOAD BYTE COUNT INTO Y
        ASL   $30       ;SHIFT LOW-ORDER BYTE
        LDX   #01       ;BYTE INDEX = 1
        DEY             ;DECREMENT BYTE COUNT
NXTBYT  ROL   $30,X     ;SHIFT NEXT BYTE
        INX             ;UPDATE BYTE INDEX
        DEY             ;  AND BYTE COUNT
        BNE   NXTBYT    ;LOOP UNTIL ALL BYTES SHIFTED
;THE CODE THAT FOLLOWS RESTORES THE SIGN TO THE MOST-SIGNIFICANT
;BYTE (MSBY)
        DEX             ;MAKE INDEX POINT TO MSBY
        LDA   $30,X     ;LOAD MSBY INTO ACCUMULATOR
        BCC   MSB0      ;SIGN = 1?
        ORA   #$80      ;YES. PUT ONE IN SIGN BIT
        JMP   SOVER
MSB0    AND   #$7F      ;NO. PUT ZERO IN SIGN BIT
SOVER   STA   $30,X     ;RETURN MSBY TO MEMORY
```

right (like every other bit), and its value will be replaced with a 0. If a signed number is left-shifted, the sign will be displaced into the Carry flag, and its value will be replaced by the value of Bit 6. Obviously, your program must somehow restore the displaced sign value. The following discussion will explain how to do it.

As mentioned previously, in a *logical left shift,* the sign is shifted out the left end of the operand into the Carry flag. You can restore the sign by simply setting up a mask with Bit 7 equal to the new value of the Carry flag, and, then, ORing (if Carry = 1) or ANDing (if Carry = 0) that mask into the shifted operand. Example 2-11 performs just such an operation in left-shifting a multiple-precision signed number of variable length. You will note that this routine is nothing more than the unsigned number routine (Example 2-10), with seven instructions added to restore the sign.

In a *logical right shift,* the sign is vacated by the operation. You can preserve the sign by recording its original value in the Carry flag and by using an ROR instruction to shift it into the most-significant byte. Example 2-12 is a routine that employs this technique.

REGISTER TRANSFER INSTRUCTIONS

Although the 6502 microprocessor has a memory-oriented architecture, it also has 6 one-byte instructions that allow you to copy the contents of one register into another, without disturbing the source register. The register transfer instructions are:

Instruction	Description
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

The TXS and TSX instructions are specific to stack operations, so their description will be deferred until later in this chapter. The other four instructions do not have one specific application, but they are useful from time to time in conserving space and in reducing execution time.

Arithmetic Operations on the X and Y Registers

Since the 6502 microcomputer can only increment and decrement the X and Y registers, the transfer instructions provide a simple way to copy X and Y into the accumulator for a more complex arithmetic operation and, then, have the result returned. Consider

Example 2-12: A Right-Shift Routine for Multiple-Precision Signed Numbers

```
;THIS ROUTINE RIGHT-SHIFTS A MULTIPLE-PRECISION SIGNED NUMBER
;STORED IN MEMORY STARTING AT LOCATION $30. THE LENGTH OF THE
;NUMBER, IN BYTES, IS CONTAINED IN LOCATION $2F.
        CLC             ;PREPARE FOR SIGN = 0 SHIFT
        LDX   $2F       ;LOAD BYTE COUNT INTO X
        DEX             ;SET INDEX FOR MSBY
        LDA   $30,X     ;LOAD HIGH-ORDER BYTE
        BPL   MSB0      ;SIGN = 1?
        SEC             ;YES. PREPARE FOR SIGN = 1 SHIFT
MSB0    ROR   A         ;SHIFT HIGH-ORDER BYTE
        STA   $30,X     ;  AND RETURN IT TO MEMORY
        DEX             ;DECREMENT INDEX FOR NEXT BYTE
NXTBYT  ROR   $30,X     ;SHIFT NEXT HIGHEST BYTE
        DEX
        BPL   NXTBYT    ;LOOP UNTIL ALL BYTES SHIFTED
```

a case in which you need to access every fifth element in a list, rather than consecutive elements. The access instruction would be an indexed instruction, such as LDA LIST,Y, in which LIST is a label assigned to the starting location of the list. To access every fifth element, you must add 5 to the index register (Y) between load operations.

How can you add 5 to the Y register? One way is to code five consecutive INY instructions. A more efficient way of doing this is to transfer Y to the accumulator, add 5 (immediate), and return the sum to Y for the next list access. The routine in Example 2-13 uses every fifth element of an existing list to construct a new list elsewhere in memory. This routine employs the accumulator to update the index register.

What has the four-instruction add-5-to-Y sequence in Example 2-13 saved us over five INY instructions? The five INY instructions

Example 2-13: Accessing Nonconsecutive Elements in a List

```
;THIS ROUTINE USES EVERY FIFTH ELEMENT (0, 5, 10, ETC.) OF A
;LIST THAT STARTS AT LOCATION $0501 TO CONSTRUCT A NEW LIST,
;STARTING AT LOCATION $21. THE LENGTH OF THE SOURCE LIST, IN
;BYTES, IS CONTAINED IN LOCATION $0500. THE BYTE LENGTH OF
;THE NEW LIST IS ENTERED INTO LOCATION $20.

        LDY   #00       ;SOURCE LIST INDEX = 0
        LDX   #00       ;NEW LIST INDEX = 0
GETEM   LDA   $0501,Y   ;LOAD NEXT SOURCE BYTE
        STA   $21,X     ;  AND STORE IT INTO NEW LIST
        INX             ;UPDATE NEW LIST INDEX BY ONE
        TYA             ;UPDATE SOURCE LIST INDEX BY FIVE
        CLC
        ADC   #05
        TAY
        CMP   $0500     ;FINISHED WITH SOURCE LIST?
        BCC   GETEM     ;NO. GET NEXT ELEMENT
        STX   $20       ;YES. STORE NEW LIST BYTE COUNT
```

would occupy five bytes in memory and take a total of ten cycles to execute. In the four-instruction sequence, CLC, TYA, and TYA will each take one byte and two cycles, and ADC #05 will take two bytes and two cycles, for a total of five bytes and eight cycles. Thus, the transfer method takes the same amount of memory, but it is two cycles faster in execution. Therefore, the conclusion of this exercise is self-evident: *To increment or decrement the X or Y register by five or more, do the arithmetic in the accumulator; for increments and decrements of less than five, do it with the increment and decrement instructions.*

STACK INSTRUCTIONS

You will recall from Chapter 1 that the stack of the 6502 microprocessor is of the "last-in-first-out" variety. That is, the last item (a data byte, in this case) that is entered onto the stack is the first item to be extracted from the stack. Conversely, the first item that is entered onto the stack is the last item to be extracted from the stack. *This scheme causes data to be retrieved in the reverse order from which it was stored.*

Stack information is accessed by a dedicated stack address register called the *Stack Pointer*, which always points to the next free memory location "on" the stack. The Stack Pointer is automatically decremented after a byte is pushed onto the stack, and is automatically incremented before a byte is pulled from the stack, so the stack "builds" in the direction of address 0.

In the 6502 microprocessor, the stack is implemented in Page 1 (locations $0100 through $01FF) of the address space. Thus, the Stack Pointer must be initialized by the user's program to address $01FF, when the power is turned on. (This is accomplished automatically when using an AIM 65.) Since a page is comprised of 256 byte locations, the stack can hold up to 256 bytes of information. What kinds of information can it hold? The stack is normally used for two purposes: (1) to save interrupt or subroutine return addresses and (2) to temporarily save register contents. We will explain how and why addresses are saved on the stack when subroutines are discussed in Chapter 3. For now, let us discuss instructions that are used to initialize the Stack Pointer and to save the register contents.

Stack Pointer Instructions

The contents of the Stack Pointer are undefined when power is turned on, and must be initialized by the user's program. The 6502 microprocessor has a special instruction to initialize the Stack Pointer:

Instruction	Description
TXS	Transfer Index X to Stack Pointer

This instruction uses implied addressing, so it occupies just one byte in memory and takes two cycles to execute.

Since the stack starts at memory location $01FF, most system programs initialize the Stack Pointer to $FF (recall that the high-order address byte, $01, is automatically supplied by the 6502 microprocessor). Example 2-14 initializes the Stack Pointer to $FF.

Example 2-14: Initializing the Stack Pointer to $FF

```
;THESE INSTRUCTIONS INITIALIZE THE STACK POINTER TO ADDRESS
;THE "TOP" LOCATION IN PAGE 1, $01FF.

    LDX  #$FF  ;LOAD $FF INTO X REGISTER
    TXS        ;TRANSFER X REGISTER TO STACK POINTER
```

The contents of the Stack Pointer can also be "read" by a program, using another implied-addressing instruction:

Instruction	Description
TSX	Transfer Stack Pointer to Index X

Although this instruction is available, it is rarely used. Why? Because, in most cases, you do not *care* where in the stack information is being stored. However, the TSX instruction is unique in that it is the only instruction that permits you to access the current value of the Stack Pointer, if needed.

Push and Pull Instructions

The 6502 microprocessor has instructions that permit the contents of both the accumulator and the processor status register to be saved on the stack. They are:

Instruction	Description
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack

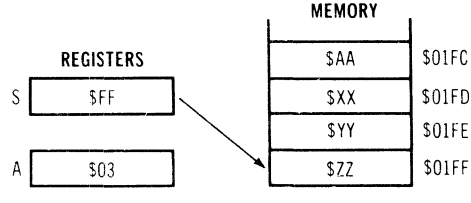
Aside from the fact that they push different registers onto the stack, the PHA and PHP instructions work identically. In each case, the contents of the register are pushed onto the stack at the location being pointed to by the Stack Pointer. Then, the Stack Pointer is decremented by one, to the next lower address. Neither instruction alters the contents of its source register. Similarly, the PLA and PLP instructions increment the Stack Pointer to the next

higher address, and load the contents of the memory location addressed by the Stack Pointer into the appropriate destination register.

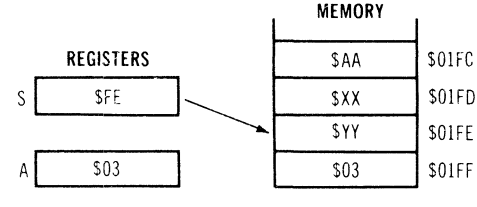
All four instructions are implied-address instructions that occupy one byte in memory. Since the previously discussed Load and Store instructions (LDA and STA) require two bytes of memory in their shortest form (zero page addressing mode), saving the accumulator on the stack rather than in memory means that our instruction sequence is one byte shorter. It should be noted, too, that the PHP and PLP instructions represent the *only* way that the processor status register can be saved.

Fig. 2-5 shows the effect of the PHA instruction on the stack. In Fig. 2-5A, the Stack Pointer (S) is pointing to the top of the stack (location $01FF), and the accumulator contains $03. After PHA is executed (Fig. 2-5B), the Stack Pointer has been decremented to $FE and the contents of the accumulator have been stored in location $01FF. If we were to execute a PLA instruction after the push, the Stack Pointer would be incremented to point to $01FF, and the accumulator would be loaded with the contents of location $01FF.

So far, there has been a lot of talk about how the stack instructions work, but virtually nothing has been said about *why* someone would want to save registers on the stack. There is really only one reason why these values would be saved; to preserve their contents while they are being manipulated by other programming operations. This is particularly true in subroutines, but it is equally applicable to routines where registers are serving two functions, or where the final contents of a register are unpredictable.



(A) Before PHA.



(B) After PHA.

Fig. 2-5. How a register is pushed onto the stack.

With only four instructions, it looks as if only the accumulator and the processor status register can be saved on the stack. That is true as far as the instructions go, but by using the register transfer instructions we can move the X and Y registers into the accumulator, push them onto the stack, and later pull them off and restore them with additional register transfer instructions. Example 2-15 shows the coding required to save not only the accumulator and the processor status register, but also the X and Y registers on the stack. Note that the registers are pulled in the reverse order from which they were pushed. Incidentally, the AIM 65 Monitor contains a subroutine that can be called to push the contents of the X and Y register onto the stack. It is called PHXY (entry address $EB9E), and there is another subroutine that pulls the contents of these registers from the stack, PLXY (entry address $EBAC).

Does it make any difference in which order register contents are saved on the stack? In most cases, it does. The Accumulator contents must always be pushed onto the stack before either X or Y is pushed, because the X and Y pushes must be preceded by a TXA or TYA instruction, which destroys the contents of the Accumulator. Furthermore, if the Processor Status Register is to be saved, it must also be pushed onto the stack before either X or Y is pushed, because both TXA and TYA affect the Processor Status Register's Negative (N) and Zero (Z) flags.

Example 2-15: Saving All Registers on the Stack

```
PHP   ;SAVE PROCESSOR SATUS REGISTER
PHA   ;SAVE ACCUMULATOR
TXA   ;SAVE X REGISTER
PHA
TYA   ;SAVE Y REGISTER
PHA
 .
 .    Some routine is executing here
 .
PLA   ;RESTORE Y REGISTER
TAY
PLA   ;RESTORE X REGISTER
TAX
PLA   ;RESTORE ACCUMULATOR
PLP   ;RESTORE PROCESSOR STATUS REGISTER
```

THE NO OPERATION INSTRUCTION

The No Operation (NOP) instruction is a simple one-byte implied addressing instruction that is generally used during program development. The NOP instruction performs no operation—it does not alter any status flags, registers, or memory locations, but it does perform the very useful function of reserving space in memory.

Many programmers code NOP instructions into a program under development, to leave room for instructions that may have to be added at a later time. Since each NOP instruction occupies only one byte in memory, at least two NOPs should be inserted at the spot where space is to be reserved.

NOP instructions may also be used to replace instructions that have been deleted, without requiring all of the branch and jump instruction addresses to be changed.

SUMMARY

So far, the following instructions have been discussed:

1. Load and store instructions LDA, LDX, LDY, STA, STX, and STY.
2. Addition and subtraction instructions ADC and SBC.
3. Status bit clear and set instructions CLC, CLD, CLV, SEC, and SED.
4. Decrement and increment instructions for registers (DEX, DEY, INX, and INY) and for memory (DEC and INC).
5. Logical instructions AND, EOR, and ORA.
6. Unconditional jump instruction JMP.
7. Conditional branch instructions BCC, BCS, BEQ, BNE, BMI, BPL, BVS, and BVC.
8. Compare instructions CMP, CPX, and CPY.
9. Bit test instruction BIT.
10. Shift and rotate instructions ASL, LSR, ROL, and ROR.
11. Register transfer instructions TAX, TAY, TSX, TXA, TXS, and TYA.
12. Stack push and pull instructions PHA, PHP, PLA, and PLP.
13. No operation instruction NOP.

3

Subroutines

Up to this point, all of the examples have contained instructions that can be included in a program in order to perform a specific function *once*. It has been implied (but not stated) that if a specific operation must be performed at more than one place in the program, the entire sequence of instructions must be repeated in the program at each one of these places. Obviously, repeating a sequence of instructions in a program at many places would not only be frustrating and time-consuming for the programmer, but it would also make programs much longer than they would be if this repetition could be avoided.

As a matter of fact, all microprocessors *do* eliminate this needless repetition by defining the repeated code as a *subroutine*. A subroutine is a sequence of instructions that is written just once, but which can be executed as needed, at any point in the program. The process of transferring control from a program to a subroutine is defined as *calling*. Therefore, subroutines are *called*. Once called, the 6502 microprocessor executes the sequence of instructions contained in the subroutine and, then, returns control to the calling program.

This description invites two questions: "How is a subroutine called?" and "How does the 6502 microprocessor return to the proper point in the program?" These questions will be answered in the following discussion of two subroutine-related instructions, Jump to Subroutine (JSR) and Return from Subroutine (RTS).

SUBROUTINE INSTRUCTIONS

From the description just given, it is apparent that instructions that cause subroutines to be called must perform three functions:

1. They must include some provision for saving the contents of the program counter. Once the subroutine has been executed, this address will be used to return to the program. This address is often called a *return address.*
2. They must cause the microprocessor to begin executing the subroutine.
3. They must use the stored contents of the program counter to return to the program, and continue executing the program at this point.

These three functions are performed by two 6502 instructions, Jump to Subroutine (JSR) and Return from Subroutine (RTS).

Jump to Subroutine (JSR)

The JSR instruction performs the return-address-storing and begin-executing functions (requirements 1 and 2). The return-address stored is the address of the third byte of the JSR instruction. (Clearly, this is *not* the address to which the return will ultimately be made, but this address will be incremented upon return to provide the proper destination. This will be discussed in more detail when we examine the RTS instruction.) Where is the return address stored? It is stored on the stack, which means that the JSR instruction operates like the PHA and PHP push instructions that we encountered in the section on Stack Instructions in Chapter 2. However, unlike the PHA and PHP instructions, which pushed one byte of data onto the stack (the accumulator contents and the processor status register contents, respectively), the JSR instruction pushes *two* bytes onto the stack—the two-byte address in the program counter. After storing the program counter on the stack, the JSR instruction loads the program counter with the absolute address contained in its second and third bytes, which transfers control to the starting address of the subroutine.

The JSR instruction occupies three bytes in memory—one op code byte and two subroutine address bytes (the low-address byte and high-address byte, respectively). The subroutine address is an absolute address in memory or, for assembler source code, the *label* of an absolute address. The 6502 needs six cycles to execute the JSR instruction because of the stack operations involved.

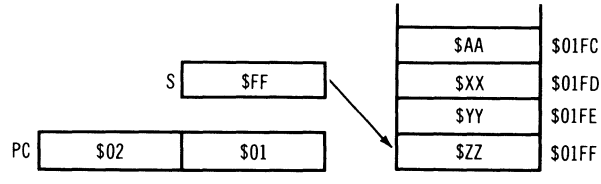
Now, consider a typical Jump to Subroutine instruction, JSR $0503, that is located in memory at locations $0201, $0202, and $0203. Thus:

Memory Location	Instruction
$0201	JSR
$0202	$03
$0203	$05
.	.
.	.
.	.
$0503	(First subroutine instruction)

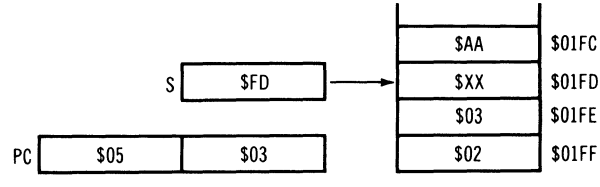
Fig. 3-1 shows the contents of the Stack Pointer (S), the program couter (PC) and the stack in memory both before and after the instruction JSR $0503 is executed. In the drawing of Fig. 3-1A, the program counter is pointing to the first byte of the JSR instruction ($0201) and the Stack Pointer is pointing to the next free location on the stack (the top location of the stack, $01FF, is assumed). In the drawing of Fig. 3-1B, after the JSR $0503 instruction has been executed, the program counter is pointing to the starting address (the *entry address*) of the subroutine ($0503), the Stack Pointer is pointing to a new "next stack location" ($01FD), and the two top bytes on the stack are the MSBY and LSBY addresses of the third byte of the JSR instruction. As previously mentioned, *the 6502 microprocessor must return to the instruction that immediately follows the JSR instruction.* For our example, the 6502 must return to the instruction that starts in location $0204.

Return from Subroutine (RTS)

The Return from Subroutine (RTS) instruction causes the 6502 microprocessor to return from the subroutine to the calling program



(A) Before executing JSR $0503.



(B) After executing JSR $0503.

Fig. 3-1. How a return address is saved on the stack when a JSR instruction is executed.

(the program that contains the JSR instruction). The only difference between a sequence of instructions stored in memory and a subroutine stored in memory is the fact that an RTS instruction is always the last instruction that is executed in a subroutine.

In essence, the RTS instruction causes the 6502 microprocessor to continue program execution at an absolute address that is one greater than the address on the last two bytes of the stack. Recall that before loading the subroutine address into the program counter, the JSR instruction pushed two address bytes onto the stack. These were the least-significant address byte (LSBY) and the most-significant address byte (MSBY) of the memory location that contains the high-address byte (the third byte) of the JSR instruction. In our example, this was address $0203. To retrieve that address, the RTS instruction increments the Stack Pointer, loads the LSBY address into the lower half of the program counter and, then, increments the Stack Pointer again and, then, loads the MSBY address into the upper half of the program counter. The program counter is then incremented so that it addresses the next instruction after the JSR instruction.

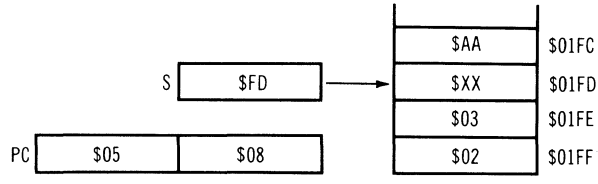
There is an implied requirement, of course, that when an RTS instruction is executed, the Stack Pointer is addressing the stack location used or established by the JSR instruction. Therefore, if any push operations are performed in the subroutine, there must be an equal number of pull operations *before* the RTS is executed. (The RTS instruction occupies one byte in memory, takes six cycles to execute, and affects no bits in the processor status register.)

How JSR and RTS are Used Together

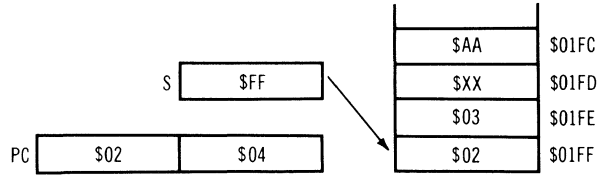
Having described the two subroutine instructions, JSR and RTS, let us look at an example of how they are used together. Example 3-1 shows the Jump to Subroutine instruction that was diagrammed in Fig. 3-1. The instruction JSR $0503 is being used to call a subroutine that simply doubles the value in the Y register. The JSR

Example 3-1: The Subroutine Call and Return Sequence

Memory Location	Mnemonic	Operand	Comment
$0201	JSR	$0503	;SUBROUTINE CALL INSTRUCTION
$0204	•	•	;SUBROUTINE RETURNS HERE
	•	•	
	•	•	
$0503	PHA		;SAVE ACCUMULATOR ON STACK
	TYA		;TRANSFER Y INTO ACCUMULATOR,
	ASL	A	; DOUBLE IT,
	TAY		; TRANSFER RESULT BACK TO Y
	PLA		;RETRIEVE ACCUMULATOR
$0508	RTS		;RETURN



(A) Before executing RTS.



(B) After executing RTS.

Fig. 3-2. How the RTS instruction pulls an address from the stack.

instruction starts in location $0201. The subroutine starts in location $0503 and ends in location $0508, with the RTS instruction.

To what location must the 6502 microprocessor return? The 6502 must return to the location that follows the JSR $0503 instruction. Since the JSR instruction occupies three bytes in memory, the return will be to location $0204. Fig. 3-2 shows the configuration of the Stack Pointer (S), the program counter (PC), and the memory stack before and after the RTS instruction is executed.

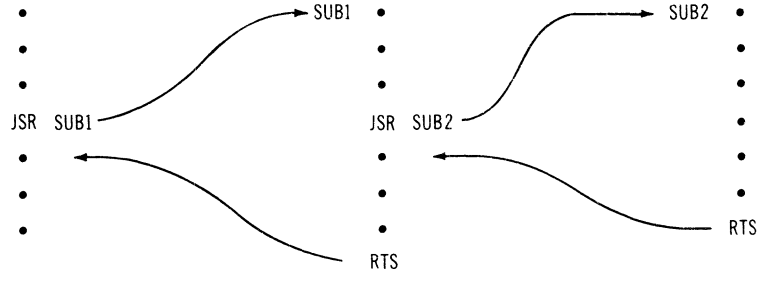
SUBROUTINE NESTING

A subroutine may include one or more JSR instructions that call other subroutines. For example, a subroutine that is called to input a keyboard character from a terminal may very well decode that character and then call one of several other subroutines, based on the decoded result. The process of calling a subroutine from within a subroutine is usually referred to as *nesting*. Example 3-2 shows the JSR and RTS instructions for a program in which subroutine SUB2 is called from within subroutine SUB1 (i.e., SUB2 is nested within SUB1).

Nesting is usually described in terms of *levels*. An application like the one shown in Example 3-2, in which the nesting extended only to the JSR to SUB2 (SUB2 did not call another subroutine), is said to have one level of nesting. There is no reason, though, why SUB2 could not have called another subroutine (SUB3), with SUB3 calling SUB4, and so on. Considering that each JSR instruction pushes two address bytes onto the stack, only the capacity

of the stack limits the amount of nesting. Since the stack can utilize all 256 bytes of page one in memory, a 6502 program can have up to 127 levels of subroutine nesting. However, very few applications will require nesting even *approaching* this limit!

Example 3-2: Subroutine Nesting



MOVING DATA IN MEMORY

Let us continue this discussion using a subroutine that performs an operation that we have already covered in this book—moving a block of data from one portion of memory to another. Chapter 2 includes two examples of this operation; Example 2-3 is a routine that moves the contents of six locations, starting at location $20, to the portion of memory that starts at location $0320. Example 2-7 is a more generalized form of Example 2-3; it moves the contents of a *specified* number of locations (up to 256) from the same starting address ($20) to the same destination address ($0320) as Example 2-3 did.

In this section, we will develop a subroutine that *offers a more generalized solution* to the move-data problem. It allows up to 256 bytes of data that are residing any place in memory *to be moved to any other place* in memory.

Many novice programmers attack this problem by writing a simple loop that fetches the first data byte from the source block, stores it into the first destination block location, decrements a byte counter, and repeats this procedure until all bytes have been moved (byte counter = 0). That approach works very well if the source block does not overlap the destination block, or if the destination block begins at a lower address than the source block. However, that approach is unacceptable if the first destination block location lies *within* the source block, because it will cause data bytes to be overlayed before they have been moved.

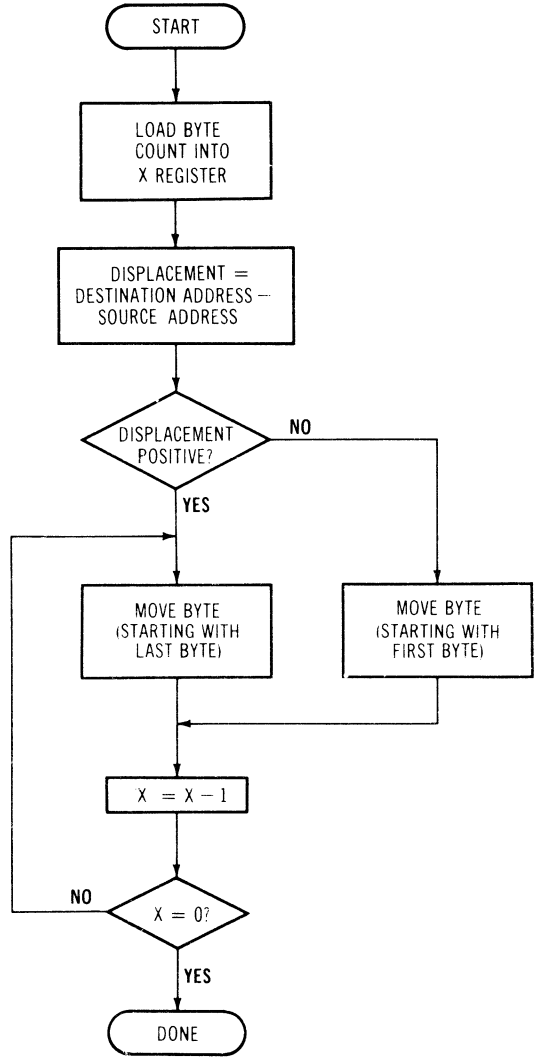


Fig. 3-3. Data block move algorithm.

What is needed, then, is a program or subroutine that has two separate paths—one for moving data downward (toward address $0000) and the other for moving data upward (toward address $FFFF). Fig. 3-3 is a flowchart for an algorithm that includes this two-path approach. This algorithm loads the byte count into the X register, and calculates the difference between the destination address and the source address (the displacement). This calculation is not performed to find the displacement, however, but only the *direction* of the move—upward (positive displacement) or downward (negative displacement). A positive displacement will cause the block to be moved in last-byte-to-first-byte order; a negative displacement will cause the block to be moved in first-byte-to-last-byte order.

Example 3-3 is a subroutine (MOVE8) that can be used to move any block of data (up to 256 bytes) anywhere in memory, using the flowcharted algorithm. The starting address of the data block is contained in locations $40 and $41, and the starting address of its destination is contained in location $42, and the lower address holds the low byte of the address in both cases. Further, the number of data bytes to be moved is specified by the contents of location $44.

Example 3-3: A Data-Block Move Subroutine

```
;THIS SUBROUTINE MOVES A BLOCK OF DATA IN MEMORY. THE STARTING
;ADDRESS OF THE DATA IS CONTAINED IN LOCATIONS $40 (LOW ADDRESS
;BYTE) AND $41 (HIGH ADDRESS BYTE). THE STARTING ADDRESS OF
;THE DESTINATION OF THE DATA IS CONTAINED IN LOCATIONS $42 (LOW
;ADDRESS BYTE) AND $43 (HIGH ADDRESS BYTE). THE NUMBER OF BYTES
;TO BE MOVED IS CONTAINED IN LOCATION $44.

MOVE8   LDX   $44      ;LOAD BYTE COUNT INTO X REGISTER
        SEC            ;CALCULATE MOVE DISPLACEMENT
        LDA   $42
        SBC   $40
        LDA   $43
        SBC   $41
        BMI   DISPN    ;DISPLACEMENT NEGATIVE?
        LDY   $44      ;NO. START MOVE WITH LAST BYTE
CONT1   DEY
        LDA   ($40),Y  ;LOAD BYTE FROM SOURCE LOCATION
        STA   ($42),Y  ;  AND STORE IT AT DESTINATION LOCATION
        DEX            ;DECREMENT BYTE COUNT
        BNE   CONT1    ;LOOP UNTIL ALL BYTES MOVED
        RTS
DISPN   LDY   #00      ;START MOVE WITH FIRST BYTE
CONT2   LDA   ($40),Y  ;LOAD BYTE FROM SOURCE LOCATION
        STA   ($42),Y  ;  AND STORE IT AT DESTINATION LOCATION
        INY            ;INDEX TO NEXT BYTE
        DEX            ;DECREMENT BYTE COUNT
        BNE   CONT2    ;LOOP UNTIL ALL BYTES MOVED
        RTS
```

When a data value is loaded into a register or a section of memory for subsequent access by a subroutine, it is called *passing an argument.* Argument passing is a common practice in programming with subroutines. The values loaded into locations $40 through $44 are the arguments being passed to the subroutine MOVES (Example 3-3).

Here is how the MOVE8 subroutine works. After loading the byte count into the X register (LDX $44), the source address is subtracted from the destination address to produce a displacement. Neither displacement byte is saved, but the sign of the most-significant byte of the result is tested by BMI DISPN. If the displacement is negative, BMI DISPN causes a branch to label DISPN; otherwise the 6502 executes a sequence that moves the block upward in memory, starting with the last byte in the block.

The sequence beginning at label DISPN (initiated by a negative displacement) performs a similar move operation, but starts the move with the first byte of the block and moves the block downward in memory.

TIME-DELAY SUBROUTINES

We will now discuss a type of subroutine that is common in computer applications—a subroutine that generates a time delay. Time delays are important in synchronizing the 6502 microprocessor to specific external events, such as transferring information between a microcomputer and a peripheral I/O device. Time delays are also essential during interactions between a very fast microcomputer and a (relatively) slow human operator. The need for a time delay is evident here if you consider that in the time that it takes you to press a key on a terminal, a microprocessor can execute *several thousand* machine cycles!

How can a time delay be programmed? Certainly the simplest way to program a time delay is by using one or more No Operation (NOP) instructions. Since each NOP instruction requires two cycles to be executed (two microseconds when using a 1-MHz 6502 microprocessor), a brief time delay can be generated by executing several successive NOP instructions. In reality, however, very few applications are so time-critical that they deal in delays of only a few microseconds. Most applications require delays of at least 50 or 100 microseconds.

Delays of these, and longer, durations are usually programmed using a subroutine. With a subroutine, we can take advantage of the fact that each 6502 instruction requires a known number of *clock cycles* to be executed. These clock cycle times are summarized in the "n" columns of Table 2-2 for each addressing mode of the

6502 instruction set. To determine the time required to execute an instruction, the number of clock cycles (n) listed in Table 2-2 must be multiplied by the *cycle time* of the microprocessor. As mentioned in Chapter 1, all execution times in this book are referenced to a 1-MHz clock, so each cycle is one microsecond long. Thus, a two-cycle instruction requires two microseconds to be executed, a three-cycle instruction requires three microseconds to be executed, and so on.

Flowchart for a Time-Delay Subroutine

The time required to execute a sequence of instructions in the 6502 microprocessor is the sum of the times required to execute each instruction in the sequence. This total time can be effectively multiplied by causing the microcomputer to loop through the instruction set a number of times. Both increment and decrement instructions are particularly useful for this task. Fig. 3-4 contains the flowchart for a time-delay subroutine that contains two loops; in one the contents of the X register is decremented to zero, in the other the Y register is incremented to zero. These registers receive their count values from two zero page locations ($20 and $21, respectively) that must be initialized by the user before the time-delay instructions are executed.

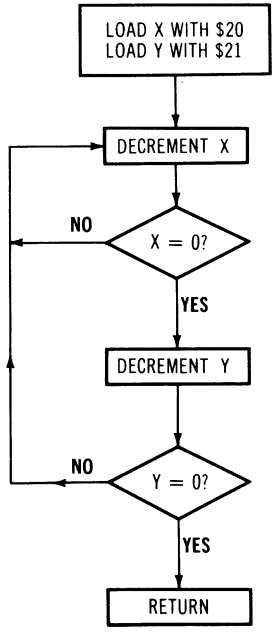


Fig. 3-4. Flowchart for a time-delay subroutine.

In this flowchart, the X register is decremented until it contains zero. When this occurs, the Y register is decremented. If Y is decremented to a non-zero value, the 6502 microprocessor branches back to the decrement-X-register loop. Since the X register already contains zero when this branch occurs, the next decrement will cause the X register to contain $FF (binary 11111111), which means that the decrement-X-register loop will be executed 255 times before the Y register is again decremented.

Decisions in this algorithm are based on whether or not the contents of a register (X or Y) have been decremented to zero. These decision-making steps are represented by the diamonds in the flowchart (Fig. 3-4). Which instruction could be used to make the actual decision in a program? The BNE instruction is the logical candidate for use in this application.

A Two-Loop Time-Delay Subroutine

Example 3-4 is a subroutine (DELAY1) that is constructed using the algorithm that is flowcharted in Fig. 3-4. At the beginning of the DELAY1 subroutine, the X register is loaded with the contents

Example 3-4: A Time-Delay Subroutine

```
;THIS SUBROUTINE CAN BE USED TO GENERATE TIME DELAYS BETWEEN 26
;MICROSECONDS AND 329 MILLISECONDS, WITH A RESOLUTION OF 5
;MICROSECONDS. THE CONTENTS OF LOCATIONS $20 AND $21 SELECT THE
;DURATION OF THE TIME DELAY.
;A VALUE OF ONE IN $20 AND $21 GENERATES THE MINIMUM TIME DELAY,
;26 MICROSECONDS. EACH ADDITIONAL COUNT INCREMENT IN LOCATION $20
;GENERATES A 5-MICROSECOND TIME DELAY. EACH ADDITIONAL COUNT
;INCREMENT IN LOCATION $21 GENERATES A 1284-MICROSECOND TIME DELAY.

DELAY1  LDX  $20    ;LOAD X WITH 5-μSEC COUNT
        LDY  $21    ;LOAD Y WITH 1284-μSEC COUNT
WAIT    DEX
        BNE  WAIT   ;LOOP UNTIL X IS ZERO
        DEY
        BNE  WAIT   ;LOOP UNTIL BOTH X & Y ARE ZERO
        RTS
```

of memory location $20 and the Y register is loaded with the contents of memory location $21. When the 6502 microprocessor executes the DEX instruction at WAIT, the contents of the X register are decremented by 1. If this decrementation produces a non-zero result, the BNE WAIT instruction causes a branch back to WAIT, to decrement the X register again. When the X register has finally been decremented to zero, the 6502 executes the DEY instruction. If the result of this operation is non-zero, the second BNE WAIT instruction branches back to WAIT, which initiates the first of 255

decrement-X-register operations. When the Y register contents have finally been decremented to zero, the RTS instruction causes the 6502 to return from the subroutine.

What range of time delays can be generated by the DELAY1 subroutine? This range can be calculated by examining the execution times of each instruction in the subroutine, and of the JSR instruction that calls the subroutine, for the minimum and maximum count values in locations $20 and $21. Execution times for all instructions in the DELAY1 subroutine are summarized in Table 3-1, for a 6502 microprocessor with a 1-MHz clock.

Table 3-1. Execution Times of Instructions in the DELAY1 Subroutine

Instruction	Execution Time (Microseconds)
JSR DELAY1	6
LDX $20	3
LDY $21	3
DEX	2
BNE WAIT	
Branch not executed	2
Branch executed	3
DEY	2
RTS	6

You will note in Table 3-1 that the BNE WAIT instructions in the subroutine will require either two cycles or three cycles to be executed, depending upon whether the branch occurs or does not occur. Therefore, the minimum execution time for the subroutine (and the minimum time delay generated) occurs when neither BNE WAIT instruction generates a branch. The branches will be bypassed when the first execution of both DEX and DEY cause their respective registers (X and Y) to be decremented to zero. The minimum time delay is, then, generated when locations $20 and $21 are both initialized to a 1. For this condition, the JSR DELAY1 instruction, and the DELAY1 subroutine that it calls, will require a total of 26 cycles to be executed. Therefore, *the DELAY1 subroutine can generate a minimum time delay of 26 microseconds.*

If a value of a 1 in memory locations $20 and $21 generates a 26-microsecond time delay, how much additional time will be used by the subroutine as the contents of these two memory locations are increased? That is, how much of a time delay is generated if location $20 contains a value of 2, 3, or 4? A similar inquiry could be made concerning various count values in location $21. Let us find out how much time is generated by each count increment, for each of these locations.

Let us first look at the time delay contributed by the count in location $20. The first instruction in the DELAY1 subroutine (LDX $20) causes the contents of location $20 to be loaded into the X register. If the DEX instruction at WAIT does not cause the X register to be decremented to 0, the DEX instruction will generate a 2-microsecond time delay and the next instruction, BNE WAIT, will contribute a 3-microsecond time delay. Therefore, *each count increment* in location $20 will generate an *additional* 5-microsecond time delay.

The time delay contributed by each count increment in location $21 is a bit more complicated, as we shall see. The second instruction in the DELAY1 subroutine (LDY $21) causes the contents of location $21 to be loaded into the Y register. If the DEY instruction does not cause the Y register to be decremented to 0, the next instruction, BNE WAIT, will produce a branch to WAIT, which will cause the 6502 microprocessor to execute the two-instruction DEX and BNE WAIT loop 256 times. The second loop, DEY and BNE WAIT, will generate a 5-microsecond time delay for each additional count increment in location $21 (2 microseconds for DEY and 3 microseconds for BNE WAIT). In decrementing the X register contents from a value of 0 to a value of $FF down to a value of $01 ($00, $FF, $FE, . . . $02, $01), the second loop, DEY and BNE WAIT, will generate a 1275-microsecond time delay (255 times 5 microseconds). When the contents of the X register are decremented from $01 to 0, an additional 4-microsecond time delay will be generated. Adding these three time-delay contributions (5 microseconds plus 1275 microseconds plus 4 microseconds), we discover that *each count increment* in location $21 will generate an *additional* 1284-microsecond time delay.

What is the maximum time delay that can be generated by the DELAY1 subroutine? Quite obviously, the maximum time delay will be the delay that is generated when locations $20 and $21 contain their maximum count values. The maximum count value for both locations is 0, which represents a count increment of 255 above the minimum count value (1). The maximum time delay can be calculated as follows:

$$\begin{aligned} TD_{max} &= 26 + (255 \times 5) + (255 \times 1284) \text{ microseconds} \\ &= 328{,}721 \text{ microseconds} \\ &= 328.721 \text{ milliseconds} \end{aligned}$$

Therefore, *the maximum time delay that can be generated by the DELAY1 subroutine is approximately 329 milliseconds.*

In summary, the rules for generating a specific time delay using the DELAY1 subroutine are:

1. A value of $01 in memory locations $20 and $21 generates a minimum time delay of 26 microseconds.
2. Each additional count increment in location $20 generates an additional 5-microsecond time delay.
3. Each additional count increment in location $21 generates an additional 1284-microsecond time delay.

As an example, let us calculate the values that must be stored in locations $20 and $21 to generate a 300-millisecond time delay. Since location $21 provides the larger time delay, we will begin by finding out how many times 300,000 microseconds (300 milliseconds) can be divided by 1284 microseconds. The answer is 233_{10}, with a remainder of 828_{10}. The hexadecimal equivalent of 233_{10} is $E9, so a value of $EA must be stored in location $21 (recall our $01 "minimum," which must be added to increments for both $20 and $21). Of the 828-microsecond remainder, 26 microseconds are contributed by the minimum time delay, so the count value in location $20 must generate an 802-microsecond delay. How many times can 802 be divided by 5? The result is 160_{10} ($A4), so $A5 must be stored into location $20 (the $01 minimum count had been added). Note that the remainder of 2_{10} has been ignored, so the subroutine actually generates a time delay of 299.998 milliseconds; this error is certainly acceptable for most applications.

A 30-Second Time-Delay Subroutine

Suppose you want the 6502 microprocessor to produce a time delay of 30 seconds. Could the DELAY1 subroutine (Example 3-4) be used to generate this time delay? Yes it could, by simply storing the values for a 300-millisecond delay into locations $20 and $21 and, then, calling the DELAY1 subroutine 100 times. A subroutine that generates a delay of 30 seconds is listed in Example 3-5.

Example 3-5: A 30-Second Time-Delay Subroutine

```
;THIS SUBROUTINE GENERATES A 30-SECOND TIME DELAY, BY CALLING
;THE DELAY1 SUBROUTINE 100 TIMES

HAFMIN  LDA   #$A5      ;PREPARE DELAY1 TO GENERATE A
        STA   $20       ;  300-MSEC TIME DELAY
        LDA   #$EA
        STA   $21
        LDA   #100      ;LOAD ACCUMULATOR WITH DECIMAL 100
CDELAY  JSR   DELAY1    ;CALL DELAY1
        SEC
        SBC   #01       ;DECREMENT THE TIMING BYTE
        BNE   CDELAY1   ;LOOP UNTIL ACCUMULATOR IS ZERO
        RTS             ;RETURN AFTER 30 SECONDS
```

In Example 3-5, the accumulator is used to hold a *timing byte.* The timing byte is really a counter for the number of times that the 300-millisecond time-delay subroutine is to be called. Since the accumulator is loaded with 100_{10}, the DELAY1 subroutine is called 100 times. The HAFMIN subroutine in Example 3-5 can be simplified by incorporating the DELAY1 instructions into it, as shown in Example 3-6.

Example 3-6: A Simplified 30-Second Time-Delay Subroutine

```
HAFMIN  LDA  #100     ;LOAD ACCUMULATOR WITH DECIMAL 100
GENDLY  LDX  #$A5     ;LOAD X AND Y FOR A 300-MSEC TIME DELAY
        LDY  #$EA
WAIT    DEX
        BNE  WAIT     ;LOOP UNTIL X IS ZERO
        DEY
        BNE  WAIT     ;LOOP UNTIL BOTH X AND Y ARE ZERO
        SEC
        SBC  #01      ;DECREMENT THE TIMING BYTE
        BNE  GENDLY   ;LOOP UNTIL ACCUMULATOR IS ZERO
        RTS           ;RETURN AFTER 30 SECONDS
```

Other Time-Delay Subroutines

A one-minute time-delay subroutine can be produced by simply replacing the instruction LDA #100 in Example 3-6 with the instruction LDA #200. A one-minute time-delay subroutine, ONEMIN, is shown in Example 3-7.

Example 3-7: A One-Minute Time-Delay Subroutine

```
ONEMIN  LDA  #200     ;LOAD ACCUMULATOR WITH DECIMAL 200
GENDLY  LDX  #$A5     ;LOAD X AND Y FOR A 300-MSEC TIME DELAY
        LDY  #$EA
WAIT    DEX
        BNE  WAIT     ;LOOP UNTIL X IS ZERO
        DEY
        BNE  WAIT     ;LOOP UNTIL X AND Y ARE ZERO
        SEC
        SBC  #01      ;DECREMENT THE TIMING BYTE
        BNE  GENDLY   ;LOOP UNTIL ACCUMULATOR IS ZERO
        RTS           ;RETURN AFTER ONE MINUTE
```

Note that the instructions in Examples 3-6 and 3-7 that begin at GENDLY constitute a general-time-delay subroutine (GENeral DeLaY) in which the duration of the delay depends upon the contents of the accumulator when the calling instruction (JSR GENDLY) is executed. If the accumulator contains 1_{10} when GENDLY is called, the subroutine will generate a 0.300-second time delay before the 6502 microprocessor returns to the calling program. As has been seen in Examples 3-6 and 3-7, if the accumulator contains 100_{10} or 200_{10} when GENDLY is called, the subroutine will generate a 30-second or 1-minute time delay, respec-

tively. The maximum time delay that GENDLY can generate is 76.8 seconds, if the accumulator contains a 0 when GENDLY is called.

Longer delays are also obtainable, by calling the HAFMIN or ONEMIN subroutine (Examples 3-6 and 3-7), or the GENDLY subroutine, a number of times. For instance, to generate a one-hour time delay, the one-minute time-delay subroutine (ONEMIN) could be called 60 times. Example 3-8 shows how this is done, in a subroutine called ONEHR. This subroutine uses the accumulator to hold a timing byte, and must store this timing byte on the stack while ONEMIN is executing, since ONEMIN also uses the accumulator to hold a timing byte of its own.

Example 3-8: A One-Hour Time-Delay Subroutine That Calls the ONEMIN Subroutine

```
ONEHR  LDA   #60     ;LOAD TIMING BYTE INTO ACCUMULATOR
DELAY2 PHA           ;  AND SAVE IT ON THE STACK
       JSR   ONEMIN  ;CALL ONE-MINUTE DELAY SUBROUTINE
       PLA           ;PULL TIMING BYTE FROM STACK
       SEC           ;  AND DECREMENT IT
       SBC   #01
       BNE   DELAY2  ;LOOP UNTIL TIMING BYTE IS ZERO
       RTS           ;RETURN AFTER ONE HOUR
```

You have now seen a number of time-delay subroutine examples. You have also seen how the duration of the time delay can be calculated, by knowing the cycle time of the 6502 microprocessor that will execute the subroutine, and the number of clock cycles required by each instruction. Knowing this, there is no reason why you could not write a time-delay subroutine that would produce a delay of 4.505 milliseconds or a delay of 23 days, 5 hours, 26 minutes, and 19 seconds.

AIM 65 Time-Delay Subroutine

Owners of the AIM 65 microcomputer have a subroutine in the monitor that can be employed to generate programmed time delays. This subroutine, DEBK1 (entry address $ED2C), produces a 5-millisecond time delay each time it is called. The AIM 65 uses the DEBK1 subroutine to "debounce" key depressions (more on this topic in Chapter 9), but it is certainly available for use by your programs.

A Note about Clock Frequency

An important point to remember about the time-delay subroutines listed in this chapter is that the time delay generated is dependent upon the clock frequency of your 6502 microprocessor. The time-delay values generated by these subroutines are based on the assumption that the system clock is providing a cycle time

of *exactly* one microsecond. The accuracy of the cycle time depends on the accuracy of the crystal that is supplying its time base; if your crystal is somewhat faster or slower than its nominal value, you may have to increase or decrease the timing-byte values in your subroutines. The crystal on the AIM 65 microcomputer has a frequency of 4 MHz (it is divided by four by external circuitry), with an accuracy to ±0.00015 MHz, so AIM 65 owners can use the subroutine examples without modifying the timing-byte values.

SUMMARY

The examples presented in this section represent only a small sampling of the applications of subroutines. You will encounter many others in subsequent chapters, and you will certainly find many other uses of subroutines in the programs that you design. Among the essential points to remember about subroutines are these:

1. Each subroutine has an overhead of 12 execution cycles (6 for the JSR, 6 for the RTS) and 4 instruction bytes (3 for the JSR, 1 for the RTS), so you must consider the time and storage tradeoffs in deciding whether to repeat a coding sequence at several points in a program or use a subroutine.
2. Subroutines save only return addresses on the stack, and do not preserve the contents of internal registers other than the program counter. If you wish to preserve the contents of the accumulator, X register, Y register, or processor status register while a subroutine is being executed, you must save those contents on the stack, in some other portion of R/W memory, or in unaffected registers until a return is made to the calling program.
3. An RTS instruction will always cause a return to the instruction which follows the JSR that called the subroutine, if the Stack Pointer has not been altered in the interim.
4. Subroutines can be *nested.* That is, one subroutine can include a call (a JSR) to another subroutine. Since the stack contains 256 locations, and each JSR pushes two address bytes onto the stack, the 6502 microprocessor supports nesting of up to 128 subroutines.

4

Lists and Look-Up Tables

There are many ways in which information in memory can be organized for processing. These organizational techniques vary with the application, and are categorized with such names as lists, arrays, strings, look-up tables, and vectors. As expected, the subject can (and does) fill many volumes, but we will concentrate on just two types of organization, *lists* and *look-up tables.*

Lists are probably the most fundamental data storage technique. They consist of units of data (one or more bytes) called elements, arranged sequentially in memory. The sequence can be consecutive, in which each element occupies one or more adjoining memory locations; or the sequence can be linked, in which each data element is followed by a pointer to the next element in the list. Further, the elements can be arranged randomly, or in ascending or descending order.

Look-up tables are data structures that have one specific purpose—to find information (either data or addresses) that has a defined relationship to a known value. A telephone directory is a good example of a look-up table; knowing a name, you can look up an associated telephone number.

UNORDERED LISTS

In our ordered society, where telephone-book listings are arranged alphabetically and where house numbers increase (or decrease) systematically as you go up or down a street, unordered *anythings* seem somehow inferior to us. Unordered lists are the bane of the programmer too because they are often difficult to process. To find out whether a certain value is in an unordered list, you must search the list from the beginning, element by element,

until you either find the value or you reach the end of the list. But like it or not, unordered lists are a fact of life in many applications, and represent a common way to store random, chronologically derived, or dynamically changing data (especially data from an experiment).

Adding An Entry to an Unordered List

Subroutine ADD2UL, given in Example 4-1, is a sample of the kind of program that you will be using to process an unordered list. This subroutine simply searches the list, element by element, for the occurrence of a value (the contents of memory location $2F). If this value is already in the list, the subroutine does nothing; otherwise, it adds the value to the end of the list, as a new element. The starting address of the list is contained in zero page locations $30 (low-address byte) and $31 (high-address byte), so all elements are accessed with indirect indexed addressing. The first location of the list contains an unsigned number that represents the length of the list, in bytes; if the entry is added to the list, this location is incremented by one.

Example 4-1: Adding an Entry to an Unordered List

```
;THIS SUBROUTINE ADDS THE CONTENTS OF LOCATION $2F TO AN UNORDERED
;LIST, IF IT IS NOT ALREADY IN THE LIST. THE STARTING ADDRESS
;OF THE LIST IS IN LOCATIONS $30 AND $31. THE LENGTH OF THE
;LIST IS IN THE FIRST BYTE OF THE LIST.

ADD2UL  LDY  #00     ;FETCH ELEMENT COUNT
        LDA  ($30),Y
        TAX          ;TRANSFER LENGTH INTO X
        LDA  $2F     ;LOAD ENTRY INTO ACCUMULATOR
NXTEL   INY          ;INDEX TO NEXT ELEMENT
        CMP  ($30),Y ;ENTRY AND ELEMENT MATCH?
        BEQ  ITSIN   ;YES, DONE
        DEX          ;NO, DECREMENT ELEMENT COUNT
        BNE  NXTEL   ;ANY MORE ELEMENTS TO COMPARE?
        INY          ;NO, ADD ENTRY TO END OF LIST
        STA  ($30),Y
        TYA          ;UPDATE ELEMENT COUNT
        STA  ($30,X) ;X IS NOW EQUAL TO ZERO
ITSIN   RTS
```

There is nothing particularly unusual about the design of this subroutine. It gets the element count from the first location of the list, and then searches the list, element by element, for the occurrence of the entry value (the contents of $2F). If the entire list has been searched without finding the contents of $2F, the BNE NXTEL test fails and the entry is tacked on to the end of the list. The entire search is conducted using indirect indexed addressing (via the Y register); however, the subroutine reverts to indexed

indirect addressing to update the element-count byte. Why? The element count could certainly be updated with indirect indexed addressing, by loading the Y register with 0 and executing an STA ($30),Y instruction, but instead it eliminates the LDY #00 instruction and updates the element-count byte with STA ($30,X). This technique takes advantage of the fact that since BNE NXTEL has failed, the X register *must* contain a 0.

How long does it take to search an unordered list? In an N-element unordered list, it takes an average of N/2 comparisons to find a match. (One-half of all search values will lie in the lower half of the list, the other one-half will lie in the upper half of the list.) Since the NXTEL loop in Example 4-1 takes 14 microseconds to execute if no match is found, and 10 microseconds to execute if a match is found, searching a 100-element list will take about 700 microseconds on the average.

Deleting an Element From an Unordered List

To delete an element from a list, you must find the element to be deleted and, then, move all the remaining elements in the list up one location (to eradicate the deleted element). Then, the element count of the list is decremented. The DELUEL subroutine given in Example 4-2 performs just such an operation, using the contents of location $2F to specify the element to be deleted. As in Example 4-1, the list address is defined by the contents of locations $30 (low-address byte) and $31 (high-address byte).

The first portion of the subroutine (DELUEL to DELETE) simply searches for the first occurrence of the desired element (the contents of $2F) in the list. If a matching element is found, the last portion (DELETE to the RTS instruction) moves all subsequent elements up one location and decrements the element count. The element-count update instructions use indexed indirect addressing to access the first byte of the list, taking advantage of the fact that the X register *must* contain a 0 because branch instruction BEQ DECCNT failed.

Finding the Minimum and Maximum Values in an Unordered List

The need to find the minimum and maximum values on a list is a requirement in many applications, particularly when test data or statistical information is being processed. One method that can be used to find these values without ordering the list is to initially establish the first element as both the minimum and maximum value, and then sequentially compare each of the remaining elements in the list to that minimum and maximum value. If an element is found that is less than the minimum value, that element becomes the new minimum value unit. Likewise, if an element is found that

Example 4-2: Deleting an Entry from an Unordered List

```
;THIS SUBROUTINE DELETES THE CONTENTS OF LOCATION $2F FROM AN
;UNORDERED LIST, IF IT IS IN THE LIST. THE STARTING ADDRESS OF
;THE LIST IS IN LOCATIONS $30 AND $31. THE LENGTH OF THE LIST IS
;IN THE FIRST BYTE OF THE LIST.

DELUEL  LDY   #00     ;FETCH ELEMENT COUNT
        LDA   ($30),Y
        TAX           ;TRANSFER LENGTH INTO X
        LDA   $2F     ;LOAD ENTRY INTO ACCUMULATOR
NEXTEL  INY           ;INDEX TO NEXT ELEMENT
        CMP   ($30),Y ;DO ENTRY AND ELEMENT MATCH?
        BEQ   DELETE  ;YES. DELETE ELEMENT
        DEX           ;NO. DECREMENT ELEMENT COUNT
        BNE   NEXTEL  ;ANY MORE ELEMENTS TO COMPARE?
        RTS           ;NO. ELEMENT NOT IN LIST. DONE

;THE INSTRUCTIONS TO FOLLOW DELETE AN ELEMENT, BY MOVING ALL ELEMENTS
;UP ONE LOCATION.

DELETE  DEX           ;DECREMENT ELEMENT COUNT
        BEQ   DECCNT  ;END OF LIST?
        INY           ;NO. MOVE NEXT ELEMENT UP
        LDA   ($30),Y
        DEY
        STA   ($30),Y
        INY
        JMP   DELETE
DECCNT  LDA   ($30,X) ;UPDATE ELEMENT COUNT OF LIST
        SBC   #01
        STA   ($30,X)
        RTS
```

is greater than the maximum value, that value becomes the new maximum.

Subroutine MINMAX in Example 4-3 applies these rules to an unordered list whose starting address is contained in locations $30 (low-address byte) and $31 (high-address byte). The first 7 instructions in the MINMAX subroutine load the element count into the X register and store the value of the first element into both the "minimum" memory location ($32) and the "maximum" memory location ($33). At label AGAIN, the element count in the X register is decremented and the next element is loaded into the accumulator. This value is compared to the minimum in $32, and becomes the new minimum value (STA $32), if appropriate. If the element has a value equal to or greater than the minimum, it is compared to the maximum in $33, and becomes the new maximum value if appropriate.

When the list has been examined entirely, the X register is decremented to zero at label AGAIN. This will cause the branch instruction BEQ BOTHIN to produce a branch to the RTS instruction.

Example 4-3: Find the Minimum and Maximum Values in an Unordered List

```
;THIS SUBROUTINE FINDS THE MINIMUM AND MAXIMUM VALUES IN AN UN-
;ORDERED LIST, STORING THE MINIMUM VALUE INTO LOCATION $32 AND
;THE MAXIMUM VALUE INTO LOCATION $33. THE STARTING ADDRESS OF
;THE LIST IS IN LOCATIONS $30 AND $31. THE LENGTH OF THE LIST
;IS IN THE FIRST BYTE OF THE LIST

MINMAX  LDY   #00
        LDA   ($30),Y
        TAX
        INY             ;INDEX TO FIRST ELEMENT
        LDA   ($30),Y   ;  AND LOAD IT INTO ACCUMULATOR
        STA   $32       ;MAKE FIRST ELEMENT THE INITIAL MIN
        STA   $33       ;  AND MAX VALUE
AGAIN   DEX             ;DECREMENT ELEMENT COUNT
        BEQ   BOTHIN    ;END OF LIST?
        INY             ;NO. INDEX TO NEXT ELEMENT
        LDA   ($30),Y   ;  AND LOAD IT INTO ACCUMULATOR
        CMP   $32       ;IS ELEMENT A NEW MIN?
        BCS   CHKMAX    ;NO. CHECK WHETHER IT'S A MAX
        STA   $32       ;YES. THIS ELEMENT IS NEW MIN
        JMP   AGAIN
CHKMAX  CMP   $33       ;IS THIS ELEMENT A NEW MAX?
        BCC   AGAIN
        BEQ   AGAIN
        STA   $33       ;YES. THIS ELEMENT IS NEW MAX
        JMP   AGAIN
BOTHIN  RTS             ;LIST HAS BEEN CHECKED. RETURN
```

A SIMPLE SORTING TECHNIQUE

Although unordered data are perfectly acceptable for many applications, ordered data are often easier to analyze, and it certainly makes it much easier to locate an element. How can an unordered list be ordered? A considerable amount of literature exists on the subject. (Two good sources are References 1 and 2.) However, one of the simplest techniques is called the *bubble sort.*

Just as bubbles rise upward into the sky, list elements rise upward in memory during a bubble sort. (Data can be sorted in an increasing or decreasing order; we will discuss only increasing order). During a bubble sort, elements of a list are accessed sequentially, starting with the first element, and are compared to the next element in the list. If an element is greater than the next sequential element in the list, the elements are exchanged. The next pair of elements is compared, exchanged if required, and so on. By the time the 6502 microprocessor gets to the last element of the list, the largest element in the list will have "bubbled up" to the last element position of the list.

If the bubble-sort algorithm is used, the microcomputer usually requires several passes to sort a list, as can be seen by the following

example. Consider a 5-element list that is initially arranged in the following order:

05 03 04 01 02

After one pass through the list, the elements will be in the following order:

03 04 01 02 05

Element 05, the largest element of the list, has "bubbled up" to the top of the list. The next pass will produce the order:

03 01 02 04 05

Element 04 is bubbled up the list to a position that is just before element 05. The result of the final pass is:

01 02 03 04 05

This example not only demonstrates how the bubble sort algorithm operates, but it also gives an indication of what type of performance you can expect from it. Note that three passes were required to sort a partially ordered, 5-element list. If the list were totally ordered at the outset, it would still take one pass through the algorithm to deduce this fact. Conversely, if the list were initially arranged in descending order (the worst case), the bubble-sort algorithm would require 5 passes to order the list, 4 passes to sort, and 1 additional pass to detect that no additional elements need to be exchanged. From this observation, we can state that the 6502 microprocessor will have to make from 1 to N number of passes through an N-element list, in order to sort it. On the average, N/2 passes are required to sort an N-element list.

What constitutes a "pass" in terms of instructions and time? This can be found out by examining two typical bubble-sort subroutines; one operating on a list containing 8-bit elements, the other operating on a list containing 16-bit elements. The basic principles that you learn in these two examples should allow you to develop bubble-sort subroutines for lists having even longer elements.

Sorting Lists Having 8-Bit Elements

The subroutine (SORT8) given in Example 4-4 sorts unordered lists that are comprised of 8-bit elements. As in the previous examples in this chapter, the starting address is contained in locations $30 (low-address byte) and $31 (high-address byte). The length of the list is contained in the first byte of the list. Since a byte is 8 bits wide, the list can contain up to 255 elements.

Example 4-4: An 8-Bit Bubble Sort Subroutine

```
;THIS SUBROUTINE ARRANGES THE 8-BIT ELEMENTS OF A LIST IN ASCENDING
;ORDER. THE STARTING ADDRESS OF THE LIST IS IN LOCATIONS $30 AND
;$31. THE LENGTH OF THE LIST IS IN THE FIRST BYTE OF THE LIST. LOCATION
;$32 IS USED TO HOLD AN EXCHANGE FLAG.
SORT8   LDY   #00       ;TURN EXCHANGE FLAG OFF (= 0)
        STY   $32
        LDA   ($30),Y   ;FETCH ELEMENT COUNT
        TAX             ;  AND PUT IT INTO X
        INY             ;POINT TO FIRST ELEMENT IN LIST
        DEX             ;DECREMENT ELEMENT COUNT
NXTEL   LDA   ($30),Y   ;FETCH ELEMENT
        INY
        CMP   ($30).Y   ;IS IT LARGER THAN THE NEXT ELEMENT?
        BCC   CHKEND
        BEQ   CHKEND
                        ;YES. EXCHANGE ELEMENTS IN MEMORY
        PHA             ;  BY SAVING LOW BYTE ON STACK.
        LDA   ($30),Y   ;  THEN GET HIGH BYTE AND
        DEY             ;  STORE IT AT LOW ADDRESS
        STA   ($30),Y
        PLA             ;PULL LOW BYTE FROM STACK
        INY             ;  AND STORE IT AT HIGH ADDRESS
        STA   ($30),Y
        LDA   #$FF      ;TURN EXCHANGE FLAG ON (= -1)
        STA   $32
CHKEND  DEX             ;END OF LIST?
        BNE   NXTEL     ;NO. FETCH NEXT ELEMENT
        BIT   $32       ;YES. EXCHANGE FLAG STILL OFF?
        BMI   SORT8     ;NO. GO THROUGH LIST AGAIN
        RTS             ;YES. LIST IS NOW ORDERED
```

Subroutine SORT8 begins by initializing an *exchange flag*. The exchange flag is an indicator in memory location $32 that can be interrogated upon completion of a sorting pass to find out whether any elements were exchanged during that pass (flag = −1) or if the pass was executed with no exchanges (flag = 0). The latter case indicates that the list is completely ordered and needs no further sorting.

After loading the element count into the X register, the 6502 microprocessors enters an element compare loop at NXTEL. As each element is fetched, it is compared to the next element in the list, with CMP ($30),Y. If this pair of elements are of equal value, or are in ascending (sorted) order, the subroutine then branches to CHKEND, to see if the element count in the X register has been decremented to zero (the end-of-list condition). Otherwise, the elements are exchanged (if the element pair is in the wrong order). The stack is used to save the lower-addressed element while the higher-addressed element is being relocated in memory. A zero page memory location could have been used to save the element, but it was

observed that PHA and PLA both execute in one less cycle than their LDA and STA counterparts. Upon completion of an exchange operation, the exchange flag is turned on, by loading it with −1.

Following the exchange, the element count is decremented with a DEX instruction (label CHKEND) and the subsequent BNE NXTEL instruction branches to NXTEL if the pass has not yet been completed. When the pass is completed, BIT $32 checks whether the exchange is still off (Bit 7 = 0), or has been turned on (Bit 7 = 1) by an exchange operation during the pass. If an exchange occurred, the subroutine is reinitiated at ORDER8, otherwise RTS causes a return, with a now ordered list.

Sorting Lists Having 16-Bit Elements

The sort subroutine discussed in the preceding section was relatively simple because the elements were 8-bit values, and could be compared with a CMP instruction and exchanged without too much difficulty. Unfortunately, the 6502 microcomputer has no 16-bit compare instruction, so a comparison must be made by actually subtracting the elements and testing the status of the result; if a borrow occurs, the elements must be exchanged, otherwise the elements can remain in their present order. The SORT16 subroutine given in Example 4-5 sorts 16-bit elements using the bubble-sort algorithm and a 16-bit "compare" sequence.

Example 4-5: A 16-Bit Bubble-Sort Subroutine

```
;THIS SUBROUTINE ARRANGES THE 16-BIT ELEMENTS OF A LIST IN
;ASCENDING ORDER. THE STARTING ADDRESS OF THE LIST IS IN LOCATIONS
;$30 AND $31. THE LENGTH OF THE LIST IS IN THE FIRST BYTE OF THE LIST.
;LOCATION $32 IS USED TO HOLD AN EXCHANGE FLAG.

SORT16  LDY  #00     ;TURN EXCHANGE FLAG OFF (= 0)
        STY  $32
        LDA  ($30),Y ;FETCH ELEMENT COUNT
        TAY          ;  AND USE IT TO INDEX LAST ELEMENT
NXTEL   LDA  ($30),Y ;FETCH MSBY
        PHA          ;  AND PUSH IT ONTO STACK
        DEY
        LDA  ($30),Y ;FETCH LSBY
        SEC
        DEY
        DEY
        SBC  ($30),Y ;  AND SUBTRACT LSBY OF PRECEDING ELEMENT
        PLA          ;PULL MSBY FROM STACK
        INY
        SBC  ($30),Y ;  AND SUBTRACT MSBY OF PRECEDING ELEMENT
        BCC  SWAP    ;ARE THESE ELEMENTS OUT OF ORDER?
        CPY  #02     ;NO. LOOP UNTIL ALL ELEMENTS COMPARED
        BNE  NXTEL
```

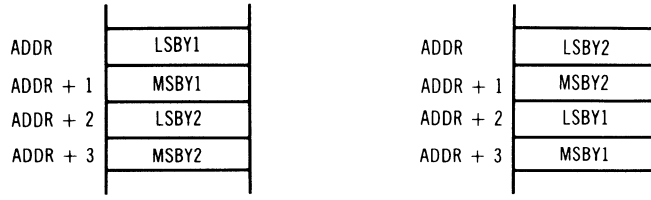
```
        BIT   $32      ;EXCHANGE FLAG STILL OFF?
        BMI   SORT16   ;NO. GO THROUGH LIST AGAIN
        RTS            ;YES. LIST IS NOW ORDERED

;THE ROUTINE BELOW EXCHANGES TWO 16-BIT ELEMENTS IN MEMORY

SWAP    LDA   ($30),Y  ;SAVE MSBY1 ON STACK
        PHA
        DEY
        LDA   ($30),Y  ;SAVE LSBY1 ON STACK
        PHA
        INY
        INY
        INY
        LDA   ($30),Y  ;SAVE MSBY2 ON STACK
        PHA
        DEY
        LDA   ($30),Y  ;LOAD LSBY2 INTO ACCUMULATOR
        DEY
        DEY
        STA   ($30),Y  ;  AND STORE IT AT LSBY1 POSITION
        LDX   #03
SLOOP   INY            ;STORE THE OTHER THREE BYTES
        PLA   ($30),Y
        STA
        DEX
        BNE   SLOOP    ;LOOP UNTIL THREE BYTE STORED
        LDA   #$FF     ;TURN EXCHANGE FLAG ON (= -1)
        STA   $32
        CPY   #04      ;WAS EXCHANGE DONE AT START OF LIST?
        BEQ   SORT16   ;YES. GO THROUGH LIST AGAIN
        DEY            ;NO. COMPARE NEXT ELEMENT PAIR
        DEY
        JMP   NXTEL
```

The SORT16 subroutine is designed with the same algorithm as SORT8, so the two subroutines naturally have several characteristics in common. For example, both SORT8 and SORT16 have an exchange flag (in the same location, $32) that indicates whether or not an exchange occurred during the last pass through the list. Like SORT8, the SORT16 subroutine also compares adjacent elements (albeit with a 16-bit subtraction, as opposed to the simple 8-bit comparison of the SORT8) and has an exchange routine that interchanges misordered elements in memory.

Aside from the fact that SORT8 and SORT16 operate on different size elements, the only other real difference between them is that SORT16 processes the list from the end and works upward, whereas, SORT8 processes the list from the beginning and works downward. Why the difference in procedure? There is no good reason, other than to demonstrate that a bubble sort can operate in either direction.

SORT16 starts by initializing the exchange flag to zero, and fetching the element count from the first byte of the list. Using that value to point Y at the last element, the 6502 microprocessor executes "compare" (subtraction) instructions. These instructions, which start with the NXTEL instruction, perform a double-precision subtraction. (If your recall of double-precision subtraction is hazy, you should briefly refresh your memory by going back to Chapter 2 and studying Example 2-2 and its associated text.) In this subtraction, the least-significant bytes (LSBYs) are subtracted first, and any borrow from that operation is passed, via the Carry, to the subtraction of the most-significant bytes (MSBYs).

Operations on multiple-precision elements typically require a lot of pointer manipulation, as you can see by the instructions that follow the NXTEL label. To get to the higher-addressed LSBY, the Y register must be decremented from its MSBY position. Before decrementing the Y register, however, the higher-addressed MSBY is pushed onto the stack for later use. With several more Y register decrements, the two LSBYs are addressed and subtracted. The result of the subtraction is not saved, since we are only interested in the final status of the operation, not the numerical result.



(A) Before Swap. (B) After swap.

Fig. 4-1. Swapping two 16-bit values in memory.

The most-significant bytes (MSBYs) are subtracted next, by retrieving the higher-addressed MSBY from the stack and subtracting the lower-addressed MSBY from it. With this subtraction, we can make the exchange/no-exchange decision based on the state of the Carry flag. If Carry is set (no borrow occurred), the elements are in the correct order; if Carry is reset (a borrow occurred), the elements must be exchanged.

Fig. 4-1 shows what the SWAP routine actually does, by presenting "before" and "after" diagrams of the exchanged 16-bit elements. The higher-valued element initially resides in symbolic addresses ADDR +2 and ADDR +3, and its bytes are designated as LSBY2 and MSBY2. The lower-valued element initially resides in symbolic addresses ADDR and ADDR +1, and its bytes are designated LSBY1 and MSBY1. The sequence of the SWAP routine will be

more easily understood if you refer to Fig. 4-1 while studying the instructions of the routine.

Due to the previous subtraction routine, the Y register index is pointing at MSBY1 when the SWAP routine is initiated. Taking advantage of this pointer, SWAP saves MSBY1 and the adjacent byte, LSBY1, on the stack. Recalling that information is retrieved from a stack in the opposite order from which it was entered on the stack, the MSBY1-then-LSBY1 push sequence implies which byte will be the next to be pushed onto the stack—it will be MSBY2. With these three bytes on the stack, the final byte LSBY2 is moved from ADDR +2 (again, refer to Fig. 4-1) to ADDR. A short loop (SLOOP) pulls bytes MSBY2, LSBY1, and MSBY1 off the stack and stores them in the locations following LSBY2. The SWAP routine ends by turning on the exchange flag in location $32. If Elements 1 and 2 were exchanged, a BEQ SORT16 instruction branches to the top of the subroutine; otherwise, control jumps to NXTEL for the next comparison.

ORDERED LISTS

Now that we have learned how to order a list, let us discuss how to search the list for a known value and, then, see how two common operations—adding elements and deleting elements—can be programmed.

Searching an Ordered List

Earlier in this chapter we learned that in order to locate a given value in an unordered list, the list must be searched sequentially, element by element. For an N-element list, this requires an average of N/2 comparisons. If a list is *ordered,* however, any of a number of search techniques can be employed. For all but the shortest lists, most of these techniques will be faster and more efficient than the sequential search technique.

One of the most widely known search techniques for ordered lists is called the *binary search.* Its name is derived from the fact that it divides the list into a series of progressively narrower halves, to eventually "zero in" on one element location in the list. A binary search starts in the middle of the list and determines which half of the list the entry value is in. It then takes *that* half of the list and divides it into halves . . . , and so on.

Example 4-6 shows a subroutine (FIND8) that searches an ordered list for the value contained in location $2F. The starting address of the list is contained in zero page locations $30 (low-address byte) and $31 (high-address byte). The first byte in the list contains the element count. If the search entry is found in the

list, the matching element number is returned in location $32; if the entry cannot be found in the list, $32 contains zero upon return.

Example 4-6: An 8-Bit Binary-Search Subroutine

```
;THIS SUBROUTINE SEARCHES FOR THE CONTENTS OF LOCATION $2F IN AN
;ORDERED LIST WHOSE STARTING ADDRESS IS CONTAINED IN LOCATIONS
;$30 AND $31. THE LENGTH OF THE LIST IS IN THE FIRST BYTE OF THE LIST
;(AND IS RETURNED IN LOCATION $33).
;LOCATION $32 HOLDS THE SEARCH RESULT UPON RETURN. IF THE ENTRY
;WAS FOUND, $32 WILL CONTAIN THE NUMBER OF THE MATCHING ELEMENT;
;IF NOT, $32 WILL CONTAIN ZERO.

FIND8   LDY   #00      ;FETCH ELEMENT COUNT
        LDA   ($30),Y
        STA   $32      ;  AND STORE IT IN LOCATIONS $32
        STA   $33      ;  AND $33
        INY            ;INDEX TO FIRST ELEMENT
NXTCHK  LSR   $32      ;CUT SEARCH INCREMENT BY ONE-HALF
        BNE   NOTDUN   ;SEARCH INCREMENT = 0?
        BCS   EVEN     ;ONE LAST COMPARE IF INCREMENT WAS = 1
        RTS            ;NO MATCH. RETURN WITH $32 = 0
NOTDUN  BCC   EVEN     ;WAS SEARCH INCREMENT ODD?
        INC   $32      ;YES. ROUND UPWARD
EVEN    LDA   ($30),Y  ;LOAD ELEMENT INTO ACCUMULATOR
        CMP   $2F      ;DOES THIS ELEMENT MATCH ENTRY?
        BEQ   FOUND    ;YES. GO RETURN
        BCS   GOLOW    ;NO. ELEMENT > ENTRY. SEARCH LOWER.
                       ;NO. ELEMENT < ENTRY. SEARCH HIGHER.
        TYA            ;ADD INCREMENT TO Y INDEX
        ADC   $32
        CMP   $33      ;  BUT LIMIT Y TO UPPER BOUND OF LIST
        BEQ   YOK1
        BCS   NXTCHK
YOK1    TAY
        JMP   NXTCHK
;
GOLOW   TYA            ;SUBTRACT INCREMENT FROM Y INDEX
        SBC   $32
        BEQ   NXTCHK   ;  BUT LIMIT Y TO LOWER BOUND OF LIST
        BCS   YOK2
        BMI   NXTCHK
YOK2    TAY
        JMP   NXTCHK
;
FOUND   STY   $32      ;ELEMENT NUMBER OF MATCH IS IN LOCATION $32
        RTS
```

The search-by-halves technique of the binary-search algorithm is started by loading the element count into memory location $32, and applying a right-shift instruction (LSR $32) to it with each comparison. You will recall from Chapter 2 that a right shift divides a number by two.

If the right-shifting step produces a search increment of 0 (i.e., if the last search increment was a 1), the BNE NOTDUN test fails and the subroutine returns with location $32 containing a 0. Otherwise, the BCC EVEN instruction at label NOTDUN rounds the new search increment upward if the preshift increment was odd, and then proceeds into the comparison routine (at label EVEN).

The comparison routine is very straightforward. If the element in the accumulator matches the search entry in $2F, the subroutine branches to FOUND, to store the element number, and to return. If the element is greater than the entry, the branch to GOLOW will cause the search increment to be subtracted from the Y index and will cause the search to continue lower in the list. Otherwise BCS GOLOW fails, and the search increment is added to Y so that the search will continue higher in the list.

How much more efficient is a binary search than a straight sequential comparison, the kind we used in Example 4-1? A mathematical analysis[1] has shown that whereas a sequential search of an N-element list requires an average of N/2 comparisons, a binary search requires $\log_2$ N comparisons. For a 100-element list, a sequential search will average 50 comparisons, but a binary search will do the same job with about 7 comparisons!

Adding an Entry to an Ordered List

The process of adding an entry to an ordered list can be divided into four basic steps:

1. Find out where the entry must be added.
2. Clear a location for the entry, by moving all higher-valued elements down one position, to the next higher-address location.
3. Insert the entry at the newly vacated element position.
4. Update the list length, by adding one to it.

Certainly one way to locate the insert position is to compare each element of the list with the entry, sequentially. When an element is found with a greater value than the entry, the search is over, and the entry can be inserted immediately ahead of that element. This approach will get the job done, but as you will recall from the discussion of unordered lists, it will average N/2 comparisons for an N-element list.

As a more efficient alternative, the FIND8 subroutine (Example 4-6) can be employed to do the searching. The FIND8 subroutine returns three separate items of information:

- Location $32 contains a 0 if the entry is not already in the list; otherwise, it contains the number of the matching element.

- Location $33 specifies the number of elements in the list.
- The Y register reflects the number of the last element to be processed in the search routine of FIND8.

The subroutine in Example 4-7 (ADD2OL) shows how the 6502 microprocessor is used to add an entry to an ordered list, if it is not already present in the list. Like FIND8, the ADD2OL subroutine operates on a list whose starting address is contained in locations $30 (low-address byte) and $31 (high-address byte). The entry value is contained in location $2F.

The subroutine ADD2OL begins by calling FIND8 with a JSR instruction. If the entry is not in the list, the index value in the Y

Example 4-7: Adding an Element to an Ordered List

```
;THIS SUBROUTINE ADDS THE CONTENTS OF LOCATION $2F TO AN ORDERED LIST,
;IF IT IS NOT ALREADY IN THE LIST. THE STARTING ADDRESS OF THE LIST
;IS IN LOCATIONS $30 AND $31. THE LENGTH OF THE LIST IS IN THE
;FIRST BYTE OF THE LIST.
;THE FIND8 SUBROUTINE (EXAMPLE 4-6) IS CALLED TO PERFORM THE SEARCH.

ADD2OL  JSR   FIND8    ;SEARCH LIST FOR ENTRY
        LDA   $32      ;IS ENTRY IN THE LIST?
        BNE   ITISIN   ;YES. RETURN
        STY   $32      ;NO. SAVE FINAL Y OF THE SEARCH
        SEC            ;CALCULATE NUMBER OF BYTES TO END OF LIST
        LDA   $33      ;  AS DIFFERENCE BETWEEN LIST LENGTH
        SBC   $32      ;  AND FINAL SEARCH Y
        TAX            ;PUT BYTE COUNT IN X
        LDA   $2F      ;LOAD ENTRY INTO ACCUMULATOR
        CMP   ($30),Y  ;ENTRY > FINAL SEARCH ELEMENT?
        BCS   GRTR     ;YES. INSERT ENTRY AFTER ELEMENT
        INX
        JMP   MOVEM
GRTR    INC   $32
        CPX   #00      ;IS BYTE COUNT = 0?
        BEQ   INSERT   ;YES. TACK ENTRY ON TO END OF LIST
MOVEM   LDY   $33      ;INDEX TO LAST ELEMENT IN LIST
MOVNXT  LDA   ($30),Y  ;LOAD ELEMENT
        INY
        STA   ($30),Y  ;  AND STORE IT IN NEXT LOCATION
        DEY            ;BACKTRACK TO PRECEDING ELEMENT
        DEY
        DEX            ;DECREMENT BYTE COUNT
        BNE   MOVNXT   ;LOOP UNTIL ALL ELEMENTS MOVED
INSERT  LDA   $2F      ;INSERT ENTRY INTO LIST
        LDY   $32
        STA   ($30),Y
        INC   $33      ;UPDATE ELEMENT COUNT
        LDA   $33
        LDY   #00
        STA   ($30),Y
ITISIN  RTS            ;RETURN
```

register is saved in location $32, replacing the "not found" indicator. As mentioned above, the Y register points to the list element at which FIND8 concluded its search. This parameter is very meaningful because it indicates the element at which FIND8 *expected* to find the entry if it had been in the list. What does this mean to us? It means that the entry must be inserted *immediately preceding* or *immediately following* the element to which the Y register points.

The value of the Y register also gives an indication of how many elements must be moved to make room for the insertion. Specifically, if the entry is to follow the last search element, the number of elements to be moved is given by:

$$\text{Elements to be moved} = \text{List Length} - Y$$

If the entry is to precede the last search element, the number of bytes to be moved is given by:

$$\text{Elements to be moved} = \text{List Length} - Y + 1$$

Therefore, after saving the final search value of the Y register in $32, the subroutine calculates the number of bytes to be moved, by subtracting the Y value in $32 from the list length in $33, and then puts the result in the X register.

The subroutine then addresses the question of whether the entry must be inserted ahead of or after the final search element. It does this by comparing the entry value to the element value at the final search value of Y. If the entry is greater than the final search value, BCS GRTR branches to label GRTR where the insert Y value in location $32 is incremented (so that the entry will be inserted after the final search element). The routine at label GRTR also checks whether the entry falls outside the upper bound of the list, in which case, it is inserted at the end of the list (without moving any elements).

Returning to the compare operation, if the entry is less than the final search value, the element count in the X register is incremented before jumping to the move routine at MOVEM. The MOVEM routine indexes to the last element in the list (LDY $33), and then moves the preceding elements in the list up one, to a higher memory location, until the X register has been decremented to zero. At that point, the entry in location $2F is stored in the list. With this new element in place, the subroutine adds one to the element count (the first byte in the list), and returns.

Deleting an Element from an Ordered List

It is much easier to delete an element from an ordered list than it is to add one, because deleting only entails finding the proper

element, moving all subsequent elements down to lower memory locations in the list (one location), and decrementing the list length byte.

Example 4-8 shows a typical delete subroutine (DELOL), that uses the FIND8 subroutine (Example 4-6) to locate the position of the intended deletion element. As in all previous examples in this chapter, the starting address of the list is contained in locations $30 (low-address byte) and $31 (high-address byte). The value of the element is in location $2F.

If FIND8 locates the element in the list, the DELOL subroutine addresses the next consecutive element and enters a MOVMOR routine that moves all subsequent elements down to one lower memory location. At the end of the move operation, the element count in the first byte of the list is decremented by one to reflect the deletion.

Example 4-8: Deleting an Element From an Ordered List

```
;THIS SUBROUTINE DELETES THE CONTENTS OF LOCATION $2F FROM AN ORDERED
;LIST, IF IT IS IN THE LIST. THE STARTING ADDRESS OF THE LIST IS IN
;LOCATIONS $30 AND $31. THE LENGTH OF THE LIST IS IN THE
;FIRST BYTE OF THE LIST.
;THE FIND8 SUBROUTINE (EXAMPLE 4-6) IS CALLED TO PERFORM THE SEARCH.

DELOL   JSR  FIND8     ;SEARCH LIST FOR ENTRY
        LDA  $32       ;IS ENTRY IN THE LIST?
        BEQ  ITSOUT    ;NO. RETURN
        INY            ;YES. ADDRESS NEXT ELEMENT,
MOVMOR  LDA  ($30),Y   ;  LOAD IT INTO THE ACCUMULATOR,
        DEY
        STA  ($30),Y   ;  AND MOVE IT DOWN ONE LOCATION IN LIST
        INY
        INY
        CPY  $33       ;HAVE ALL ELEMENTS BEEN MOVED?
        BCC  MOVMOR    ;NO. GO MOVE ANOTHER ELEMENT
        BEQ  MOVMOR
        LDA  $33       ;YES. DECREMENT ELEMENT COUNT
        SBC  #01
        LDY  #00
        STA  ($30),Y
ITSOUT  RTS
```

LOOK-UP TABLES

Many microprocessor programs include applications that require a particular value to be obtained before processing can resume. This value may be a mathematical derivative of a test or calculation result, such as the sine of a calculated angle or the centigrade equivalent of a temperature that has been measured in Fahrenheit. Or, the required value may be a parameter that has some defined

relationship to a program input, but which cannot be calculated, such as a telephone number that corresponds to a name. Applications like these usually call for a *look-up table*. As the name implies, a look-up table is used to obtain an item of information (an *argument*) based on a known value (a *function*).

Look-up tables often replace complicated or time-consuming conversion operations, such as calculating the square root or cube root of a number, or deriving a trigonometric function (sine, cosine, etc.) of an angle. Look-up tables are especially efficient when a function is limited to a very small range of arguments. By using a look-up table, the microcomputer does not have to perform complex calculations each time a function is obtained. In fact, you will find that as a rule, look-up tables reduce execution time in all but the most trivial of relationships. (You would not use a look-up table to store arguments that are always twice the value of a function, for instance.) But since look-up tables typically require large amounts of memory storage space, they are most efficient in applications where storage space can be sacrificed for execution speed.

Look-Up Tables Can Replace Equations

You can save processing time and program development time by implementing the results of complicated equations in look-up tables. In this section, we will examine one common application, that of finding the degrees Celsius equivalent of a temperature that is expressed in degrees Fahrenheit.

The Celsius-to-Fahrenheit conversion is based on the familiar relationship

$$°F = \frac{9}{5}(°C) + 32$$

or

$$°F = 1.8(°C) + 32.$$

To make this conversion in software, a microcomputer program must perform a multiplication followed by an addition. Since the 6502 microprocessor does not have a multiplication instruction, the multiplication operation must be performed by an add-and-shift sequence that can be time-consuming. (This will be discussed further in Chapter 5.) Applications that require very precise conversion results will have to make the conversion using a program, but applications that have less stringent requirements can use a Celsius-to-Fahrenheit look-up table.

Example 4-9 is a Celsius-to-Fahrenheit conversion subroutine (C2F) that is based on the look-up table approach. It can convert

Celsius temperatures between 0 and 100°, and it resolves all temperatures to an accuracy of 1°. In this subroutine, the Celsius value is assumed to be in the accumulator at entry. The Fahrenheit result is returned in both locations $40 and in the accumulator.

The C2F subroutine begins by checking whether the Celsius value is too large (equal to or greater than 101_{10}). If it is, 0 is entered into result location $40 followed by a return to the calling program. With a valid temperature (between 0 and 100° Celsius), the subroutine transfers the Celsius temperature to the Y register and uses it as an index to address the proper Fahrenheit value in the FTEMP look-up table. This table has 100 elements, one for each valid Celsius temperature. Note that the elements are defined by a series of .BYT assembler directives. Each of these directives stores one 8-bit value in memory; the first element ($40) is stored in location FTEMP, the second element ($42) is stored in location FTEMP + 1, and so on.

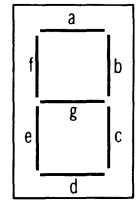
After looking up the appropriate Fahrenheit value in FTEMP and loading it into the accumulator, the subroutine stores this value at location $40 and returns. Note the speed at which the conversion is made using a look-up table; the C2F subroutine takes only 20 microseconds to execute, including the 6-microsecond RTS instruction.

Look-Up Tables for Code Conversion

Look-up tables are also used to hold coded data, such as display codes, printer codes, and messages. As an example of this type of application, let us look at how a microcomputer can communicate with a seven-segment display. Fig. 4-2 shows the components of the seven-segment display and the codes that it recognizes in forming display characters. Each bit of the code controls the "on" or "off" state of a particular segment of the display; Bit 0 controls Segment a, Bit 1 controls Segment b, and so on. Bit 7 is unused,

DIGIT	CODE
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F

(A) Codes.



(B) Diode segments.

Fig. 4-2. Seven-segment arrangement.

Example 4-9: Conversion From Degrees Celsius to Degrees Fahrenheit

```
;THIS SUBROUTINE CONVERTS A CELSIUS TEMPERATURE VALUE IN THE
;ACCUMULATOR TO A FAHRENHEIT TEMPERATURE IN LOCATION $40. THE
;CELSIUS VALUE MUST RANGE BETWEEN 0 AND 100; IF IT IS GREATER
;THAN 100, LOCATION $40 WILL CONTAIN 0 UPON RETURN.

C2F     CMP   #101     ;CELSIUS TEMPERATURE GREATER THAN 100?
        BCC   COK      ;NO. PROCEED WITH CONVERSION
        LDA   #00      ;YES. ENTER ZERO INTO $40
        STA   $40
        RTS            ;  AND RETURN
COK     TAY            ;USE CELSIUS TEMPERATURE AS INDEX
        LDA   FTEMP,Y  ;LOOK UP FAHRENHEIT TEMPERATURE
        STA   $40      ;  AND STORE IT IN $40
        RTS
FTEMP   .BYT  $40      ;0 C = 32 F
        .BYT  $42      ;1 C = 34 F
        .BYT  $44      ;2 C = 36 F
              .        (Remainder of look-up table is stored here,
              .        100 elements total)
              .
        .BYT  $D2      ;99 C = 210 F
        .BYT  $D4      ;100 C = 212 F
```

and is always off (logic 0). A code of $3F turns on Segments a, b, c, d, e, and f, forming a 0. Similarly, a code of $5B turns on Segments a, b, d, e, and g, forming a 2.

Example 4-10 is a subroutine that uses a look-up table to convert a binary-coded decimal (BCD) digit in location $40 to a seven-segment code, and stores the code in location $41. This subroutine,

Example 4-10: A BCD-to-Seven-Segment Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS A BCD DIGIT IN LOCATION $40 TO A
;SEVEN-SEGMENT DISPLAY CODE IN LOCATION $41. IF LOCATION $40
;DOES NOT CONTAIN A DIGIT BETWEEN ZERO AND NINE, LOCATION
;$41 CONTAINS ZERO UPON RETURN.

BCD2SS  LDA   #00
        STA   $41      ;INITIALIZE RESULT LOCATION TO ZERO
        LDY   $40      ;LOAD DIGIT INTO INDEX REGISTER Y
        CPY   #10      ;IS DIGIT GREATER THAN NINE?
        BCS   SSDUN    ;YES. RETURN WITH ZERO IN $41
        LDA   SSEG,Y   ;NO. LOOK UP SEVEN-SEGMENT CODE
        STA   $41      ;  AND STORE IT IN $41
SSDUN   RTS
;
SSEG    .BYT  $3F,$06,
              $5B,$4F,
              $66
        .BYT  $6D,$7D,
              $07,$7F,
              $6F
```

BCD2SS, initializes output location $41 to zero, which will produce an error if the contents of $40 are not in the range of 0 to 9. Two instructions, LDY $40 and CPY #10, check the validity of the BCD numbers ($00 to $09). An invalid (greater than 9) digit will cause a branch to the RTS instruction. With a valid digit, the BCD value in the Y register will be used to address the appropriate seven-segment display code in the SSEG look-up table.

Incidentally, the 16-segment display of the AIM 65 is an "intelligent" display that internally converts ASCII inputs to the proper display codes, without having to resort to the aid of a look-up table.

JUMP TABLES

Look-up tables can contain more than just data. In many cases, the elements of the table are addresses. An error routine, for example, can use a look-up table to find the starting address of an operator error message, based on a code in the accumulator. Similarly, an interrupt routine can use a look-up table to call one of several service routines, based on which device in the system generated the interrupt service request. Another routine may use a look-up table to call one of several control programs, based on a control key pressed by an operator. In all of these applications (there are many more as well), the look-up table containing the addresses is referred to as a *jump table*. Jump tables are used in applications where the control path is dependent on the state of a specific condition.

Example 4-11 illustrates how a jump table can service the needs of five different users in a multiterminal microcomputer system. This subroutine (SELUSR) interprets the contents of zero page location $40 as a user identification code, and uses this code to select and execute one of five user service subroutines. SELUSR checks the validity of the user identification code, and returns to the calling program if the code is greater than a four. However, with a valid code, the subroutine will fetch the appropriate user subroutine address from the jump table at UADDR.

The jump table is formed with a Word (.WOR) directive followed by five subroutine labels, USER0 through USER4. Each of these labels represents an absolute address that will occupy two bytes in memory, a low-address byte followed by a high-address byte. For example, the address of USER0 will occupy locations UADDR and UADDR+1, the address of USER1 will occupy locations UADDR+2 and UADDR+3, and so on. For this reason, the user code in the accumulator is doubled (ASL A) before being transferred to the Y register.

You might expect that at this point the user subroutine can be called with one simple instruction, JSR (UADDR,Y). Unfortunately, this addressing option is not implemented on the 6502 microprocessor. In fact, the JSR instruction can operate only with absolute addressing. How then can the user subroutine be called? As you can see, the situation is not hopeless; we still call the user

Example 4-11: A Multiuser Selection Subroutine

```
;THIS SUBROUTINE CALLS ONE OF FIVE USER SERVICE SUBROUTINES,
;BASED ON A USER IDENTIFICATION CODE (0, 1, 2, 3, OR 4) IN LOCATION
;$40. LOCATIONS $41 AND $42 ARE USED AS SCRATCH MEMORY.

SELUSR  LDA   $40          ;GET USER I.D. CODE
        CMP   #05          ;IS IT GREATER THAN 4?
        BCS   USRDN        ;YES. RETURN
        ASL   A            ;NO. DOUBLE VALUE OF I.D. CODE
        TAY                ;  AND USE IT AS INDEX IN Y
        LDA   UADDR,Y      ;FETCH LSBY OF USER ADDRESS
        STA   $41          ;  AND STORE IT IN $41
        INY
        LDA   UADDR,Y      ;FETCH MSBY OF USER ADDRESS
        STA   $42          ;  AND STORE IT IN $42
        JMP   ($0041)      ;GO EXECUTE USER SUBROUTINE
USRDN   RTS                ;RETURN ON I.D. CODE ERROR
;
UADDR   .WOR  USER0,USER1,
              USER2,USER3,
              USER4
```

subroutine with one instruction, but the instruction is a JMP rather than a JSR! It works like this. Once the index has been calculated, the low- and high-address bytes from the jump table are loaded into the accumulator (with the LDA UADDR,Y instruction) and stored in locations $41 and $42, respectively. The instruction JMP ($0041) "calls" the subroutine using the address in $41 and $42. Since this "call" did not push a return address onto the stack, the RTS in the user subroutine will cause a return to the *calling program,* rather than to the SELUSR subroutine. This technique is a valid, but tricky, "code saver." Some additional examples of jump tables are contained in References 3 and 4.

REFERENCES

1. Bentley, J. B. "An Introduction to Algorithm Design," *Computer*, February 1979, pp. 66–78.

2. Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1973. (This is a comprehensive treatment of sorting and searching.)

3. Leventhal, L. A. "Cut your processor's computation time," *Electronic Design,* August 16, 1977, pp. 82–89.

4. Titus, J. A., Titus, C. A., Rony, P. R., and Larsen, D. G. "Interfacing Fundamentals: Lookup Tables," *Computer Design,* February 1979, pp. 130–134.

5

Mathematical Routines

In this chapter we will discuss the four basic mathematical operations—addition, subtraction, multiplication, and division—and show how these operations are performed using the 6502 instruction set. We do not intend to present a "cookbook" that will cover every situation, but (hopefully) you will have sufficient information to develop your own mathematical subroutines.

The first part of the chapter covers mathematics on fixed-point integer and binary-coded decimal (BCD) numbers. By "fixed-point," we mean that the decimal point is in a fixed location, regardless of the size of the numbers being processed. Fixed-point numbers have no exponents. The final part of the chapter will cover floating-point numbers, which do have exponents.

INTEGER ADDITION

In Chapter 2, we discussed how the add instruction (ADC) of the 6502 microprocessor is used to add one- and two-byte numbers. It is nearly as easy to add numbers of any length, signed or unsigned. Example 5-1 is an addition subroutine (MPADD) that adds two multiple-precision numbers (signed or unsigned) whose least-significant bytes (LSBYs) are in locations $21 and $51, respectively. The length of the operands (the number of 8-bit bytes) is contained in location $20.

Subroutine MPADD uses the indexed addressing capability of the 6502 microprocessor to fetch a byte of one operand (LDA $21,X), add the corresponding byte of the other operand to it (ADC $51,X), and then store the result in the first operand location (STA $21,X). This sequence of operations is repeated for each byte of the operands, with the index located in the X register and the byte count

Example 5-1: A Multiple-Precision Addition Subroutine

```
;THIS SUBROUTINE ADDS TWO MULTIPLE-BYTE NUMBERS, ONE STARTING
;IN LOCATION $21, THE OTHER STARTING IN LOCATION $51. THE
;RESULT REPLACES THE NUMBER THAT STARTS IN LOCATION $21. THE
;BYTE COUNT IS CONTAINED IN LOCATION $20.

MPADD   LDY   $20     ;FETCH BYTE COUNT
        LDX   #00     ;AT START, INDEX = 0
        CLC           ;  AND CARRY = 0
NXTBY   LDA   $21,X   ;ADD 8 BITS
        ADC   $51,X
        STA   $21,X   ;STORE 8 BITS
        INX           ;UPDATE INDEX AND COUNT
        DEY
        BNE   NXTBY   ;LOOP UNTIL ALL BYTES DONE
        RTS
```

in the Y register. Since the X and Y registers are eight bits wide, the operand can be up to 256 bytes long. Note that a slightly modified version of the MPADD subroutine can be used to add two data tables, element by element, if the Carry flag is reset before each operation; that is, if the NXTBY label is assigned to the CLC instruction rather than to the LDA $21,X instruction.

INTEGER SUBTRACTION

In Chapter 2, the subtract instruction (SBC) of the 6502 microprocessor was used to subtract one- and two-byte numbers. As with integer addition, it is nearly as easy to subtract numbers of any length, signed or unsigned. Example 5-2 is a subtraction subroutine (MPSUB) that subtracts a multibyte number starting in location $51 from another multibyte number starting in location $21. The number of bytes in the operands is contained in location $20.

Example 5-2: A Multiple-Precision Subtraction Subroutine

```
;THIS SUBROUTINE SUBTRACTS A MULTIPLE-BYTE NUMBER STARTING IN
;LOCATION $51 FROM A MULTIPLE-BYTE NUMBER STARTING IN LOCATION
;$21. THE RESULT REPLACES THE NUMBER THAT STARTS IN
;LOCATION $21. THE BYTE COUNT IS CONTAINED IN LOCATION $20.

MPSUB   LDY   $20     ;FETCH BYTE COUNT
        LDX   #00     ;AT START, INDEX = 0
        SEC           ;  AND BORROW = 0
NXTBY   LDA   $21,X   ;SUBTRACT 8 BITS
        SBC   $51,X
        STA   $21,X   ;STORE 8 BITS
        INX           ;UPDATE INDEX AND COUNT
        DEY
        BNE   NXTBY   ;LOOP UNTIL ALL BYTES DONE
        RTS
```

Subroutine MPSUB uses the indexed addressing capability of the 6502 microprocessor to fetch a byte of the minuend (LDA $21,X), subtract the corresponding byte of the subtrahend (SBC $51,X), and replace the minuend byte with the result (STA $21,X). This sequence is repeated for each byte of the operands, with the index in the X register and the byte count in the Y register. Since the X and Y registers are eight bits wide, the operands can be up to 256 bytes long. Note that a slightly modified version of the MPSUB subroutine can be used to subtract two data tables, element by element, if the NXTBY label is assigned to the SEC instruction rather than to the LDA $21,X instruction.

INTEGER MULTIPLICATION

Before discussing binary multiplication, let us review the mechanics of decimal multiplication—the kind we do by hand using pencil and paper. As you will recall (in these days of calculators, it may be a bit hazy), you write the multiplicand with the multiplier below it and perform a series of multiplications—one for each digit in the multiplier. Each partial product is written directly below its multiplier digit, causing it to be displaced one digit position to the left of its predecessor. When all of the partial products have been calculated, they are added to produce the final product.

For example, the multiplication of the number 124 by the number 103 looks like this:

124	Multiplicand
×103	Multiplier
372	Partial Product 1
000	Partial Product 2
124	Partial Product 3
12772	Final Product

Of course, you do not normally write down the all-zeroes partial product. We wrote it down in the above problem to emphasize one essential principle of multiplication. A *zero multiplier causes a skip to the next digit position, without producing a partial product.* Remember that:

$$103 \times 124 = (100 \times 124) + (0 \times 124) + (3 \times 124)$$

or

$$103 \times 124 = (1 \times 10^2 \times 124) + (0 \times 10^1 \times 124) + 3 \times 10^0 \times 124).$$

With this groundwork, let us discuss binary multiplication. Binary multiplication is much simpler than decimal multiplication because

binary multipliers consist of only the digits 0 and 1. Therefore, the partial product is a repetition of the multiplicand if the multiplier digit is 1, and the partial product is zero if the multiplier digit is 0.

The binary equivalent of our previous 103 × 124 example looks like the following:

```
    01111100      Multiplicand (= 124)
  ×01100111       Multiplier (= 103)
------------
    01111100
   01111100
  01111100
 00000000
00000000
 01111100
01111100
00000000
------------
011000111100100   Final Product (= 12772)
```

Like most microprocessors, the 6502 has no multiply instruction, so the multiplication must be performed as was just demonstrated. There are some important differences, however. When performing a multiplication by hand, you record the value of the multiplicand if the multiplier bit is 1, and you record eight zeroes if the multiplier bit is 0. You then add the columns to generate a final product. The entry procedure is similar with a computer program, but instead of waiting until the individual partial products are calculated before generating the final product, computer programs update the partial product after each multiplier bit is examined. By doing this, the final product is generated when the last bit of the multiplier has been processed.

Moreover, when multiplying by hand, the digits in the multiplier are examined from right to left, so each digit is ten times (for decimal numbers) or two times (for binary numbers) more significant than the preceding digit. Therefore, the partial result of the multiplication is shifted to the left before it is recorded on paper. In a computer, it is *easier to shift the sum of the partial product and the multiplicand*, thereby aligning it to receive the contribution of the next multiplier bit. The partial product may be shifted either right or left, depending on whether the multiplier bits are being examined from right to left (low-order to high-order) or from left to right (high-order to low-order). For purposes of this discussion, we will process the multiplier in right-to-left order, the way you would do it by hand. In summary, the following applies when multiplying binary numbers by a computer:

If the multiplier is examined from right to left and a digit is a 1, add the multiplicand to the partial product, and then shift the sum one bit position to the right. If the multiplier bit is a 0, shift the current partial product one bit position to the right, with no addition.

This chapter includes subroutines that perform the two most common multiplication operations—multiplying two 8-bit numbers and multiplying two 16-bit numbers. The principles given can be used to develop any additional higher-precision routines that your own application might require. In these subroutines, the addition will be performed with the only add instruction of the 6502 microprocessor—Add to Accumulator With Carry (ADC). The shifting will be performed with the Shift Right (LSR) or Rotate Right (ROR) instruction.

Multiplying Unsigned Numbers

Fig. 5-1 is a flowchart for a subroutine that multiplies an 8-bit unsigned multiplicand (in memory) by an 8-bit unsigned multiplier (in memory), and then stores the 16-bit product in two consecutive memory locations. The multiplier bits are processed by shifting them right, into the Carry bit, and then performing either an add-and-shift (if the Carry contains a 1) or just a shift (if the Carry contains a 0).

Example 5-3 is a subroutine (MLT8) that uses the flowcharted algorithm (Fig. 5-1) to multiply the contents of location $21 by the contents of location $20. The product LSBY is returned in location $22, the product MSBY is returned in location $23. The X register is used to hold the number of unprocessed bits in the multiplier (the bit count).

In the MLT8 subroutine (Example 5-3), the LSR $20 instruction causes the multiplier in memory location $20 to be shifted one bit at a time, into the Carry. If the shifted bit is a 1, the CLC and ADC $21 instructions add the multiplicand to the most-significant byte of the partial product, and the LSR A and ROR $22 instructions shift the new partial product to the right into the least-significant byte, stored in location $22. If the shifted bit of the multiplier is a 0, BCC ALIGN bypasses the add operation by branching to the right-shift sequence at ALIGN. The NXTBT loop is executed eight times, once for each bit in the multiplier.

Multiplying Signed Numbers

The basic principles of the "add-and-shift" method of multiplication that was demonstrated in Example 5-3 can be applied to signed, as well as unsigned, multiplication. A few changes must be made

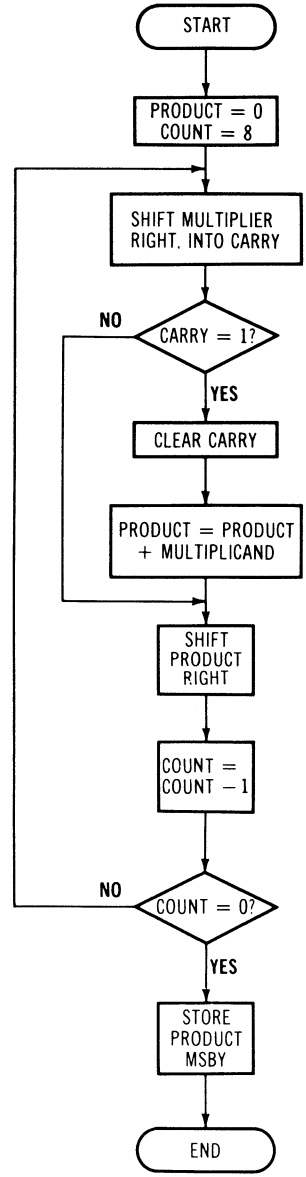


Fig. 5-1. An 8-bit × 8-bit multiplication algorithm.

Example 5-3: An 8-Bit by 8-Bit Unsigned Multiplication Subroutine

```
;THIS SUBROUTINE MULTIPLIES AN 8-BIT UNSIGNED MULTIPLICAND
;IN LOCATION $21 BY AN 8-BIT UNSIGNED MULTIPLIER IN LOCATION
;$20, AND RETURNS THE 16-BIT UNSIGNED PRODUCT IN
;LOCATIONS $22 (LOW BYTE) AND $23 (HIGH BYTE).

MLT8    LDA   #00     ;CLEAR MSBY OF PRODUCT
        LDX   #08     ;MULTIPLIER BIT COUNT = 8
NXTBT   LSR   $20     ;GET NEXT MULTIPLIER BIT
        BCC   ALIGN   ;MULTIPLIER = 1?
        CLC           ;YES. ADD MULTIPLICAND
        ADC   $21
ALIGN   LSR   A       ;SHIFT PRODUCT RIGHT
        ROR   $22
        DEX           ;DECREMENT BIT COUNT
        BNE   NXTBT   ;LOOP UNTIL 8 BITS ARE DONE
        STA   $23     ;STORE PRODUCT MSBY
        RTS
```

for signed (two's complement) multiplication, though, since the 8-bit operands now represent seven data bits prefixed by one sign bit.

The first modification involves the multiplier. You will recall that in "add-and-shift" multiplication, the bits of the multiplier determine whether the multiplicand is added to the partial product before the product is shifted. In signed multiplication a problem arises if the multiplier is negative, since its bits cannot be used in this add-multiplicand function. Example 5-4 shows what happens when 5_{10} (binary 00000101) is multiplied by -3_{10} (binary 11111101).

Example 5-4: Integer Multiplication With a Negative Multiplier

```
  00000101 =    5
  11111101 =   -3
 ----------
  00000101 Partial Product
 00000000
 ----------
  000000101 Partial Product
 00000101
 ----------
  0000011001 Partial Product
 00000101
 ----------
  00001000001 Partial Product
 00000101
 ----------
  000010010001 Partial Product
 00000101
 ----------
  0000100110001 Partial Product
 00000101
 ----------
  00001001110001 Partial Product
00000101
----------
000010011110001 = 1265
```

The final product is not the expected -15_{10} (binary 11111111 11110001), but is 1265_{10} (binary 00000100 11110001)! Therefore, to multiply signed numbers, the ***multiplier must always be positive.***

Now, consider the four sign possibilities for the multiplier and multiplicand. They are:

1. Multiplier and multiplicand are both positive.
2. Multiplier is positive, multiplicand is negative.
3. Multiplier is negative, multiplicand is positive.
4. Multiplier and multiplicand are both negative.

Since the multiplier is positive for combinations 1 and 2, our positive-multiplier requirement is satisfied for these cases. For combination 3, the multiplier is not positive, but you can *get* a positive multiplier (without affecting the result) by simply exchanging the multiplier and the multiplicand. For combination 4, the result will be positive, so you can get a positive multiplier by taking the two's complement of both operands before performing the multiplication.

Another difference between unsigned and signed multiplication arises from the fact that if a logical right shift is performed on a negative partial product, the shift causes the sign bit (most-significant bit of the product) to be set to zero. Clearly then, for a signed multiplication, *the sign of the product must be retained after the shift.* In the next example, this is done by recording the sign of the multiplicand during the sign checks at the beginning of the routine, and then ORing this value (1 or 0) into the most-significant bit of the partial product after it has been right shifted.

Example 5-5 is a subroutine (MLT8S) that multiplies two 8-bit signed integer numbers. The first part of the subroutine (MLT8S through GOMPY) makes any necessary alterations to ensure that the multiplier is positive, and stores a multiplicand sign mask in memory location $24. The mask is 80 if the multiplicand is negative and 00 if the multiplicand is positive. The second part of the subroutine (GOMPY to the RTS instruction) performs the multiplication and ORs the multiplicand sign mask with the partial product. Note that this latter portion is very similar to the MLT8 subroutine in Example 5-3, except for the ORA $24 instruction that follows the LSR A instruction at ALIGN.

Double-Precision Multiplication

Double-precision (16-bit) multiplication is a little more complicated than *single-precision* (8-bit) multiplication due to the additional memory involved, but the basic add-and-shift procedure is still used. With a double-precision multiplication, the multiplier and multiplicand are both 16-bit values, so each occupies two mem-

Example 5-5: An 8-Bit by 8-Bit Signed Multiplication Subroutine

```
;THIS SUBROUTINE MULTIPLIES AN 8-BIT SIGNED MULTIPLICAND IN
;LOCATION $21 BY AN 8-BIT SIGNED MULTIPLIER IN LOCATION $20.
;THE 16-BIT PRODUCT IS RETURNED IN LOCATIONS $22 (LOW BYTE)
;AND $23 (HIGH BYTE). LOCATION $24 IS USED TO HOLD A MULTI-
;PLICAND SIGN BIT MASK.

MLT8S   LDA   #80
        BIT   $20       ;MULTIPLIER POSITIVE?
        BPL   MPOS
        BIT   $21       ;NO. MULTIPLICAND POSITIVE?
        BPL   SWAP      ;IF SO, SWAP OPERANDS
;BOTH OPERANDS ARE NEGATIVE—NEGATE THEM
        ASL   A         ;CLEAR A, BY LEFT-SHIFTING #80
        STA   $24       ;MASK SIGN BIT = 0
        SBC   $20       ;NEGATE MULTIPLIER
        STA   $20
        LDA   #00       ;CLEAR ACCUMULATOR AGAIN
        SEC             ;  AND NEGATE MULTIPLICAND
        SBC   $21
        STA   $21
        JMP   GOMPY
;MULTIPLIER NEG, MULTIPLICAND POS—SWAP THEM
SWAP    STA   $24       ;MASK SIGN BIT = 1
        LDX   $20       ;SWAP OPERANDS
        LDA   $21
        STX   $21
        STA   $20
        JMP   GOMPY
;MULTIPLIER POS. IF MULTIPLICAND NEG, SET MASK SIGN BIT = 1
MPOS    BIT   $21       ;MULTIPLICAND POSITIVE?
        BMI   MSK1      ;IF NOT, MASK SIGN BIT = 1
        ASL   A         ;OTHERWISE, MASK SIGN BIT = 0
MSK1    STA   $24
;THE MULTIPLICATION ROUTINE FOLLOWS
GOMPY   LDA   #00       ;CLEAR MSBY OF PRODUCT
        LDX   #08       ;MULTIPLIER BIT COUNT = 8
NXTBT   LSR   $20       ;GET NEXT MULTIPLIER BIT
        BCC   ALIGN     ;MULTIPLIER BIT = 1?
        CLC             ;YES. ADD IN MULTIPLICAND
        ADC   $21
ALIGN   LSR   A         ;SHIFT PRODUCT MSBY RIGHT
        ORA   $24       ;APPLY SIGN BIT MASK
        ROR   $22       ;SHIFT PRODUCT LSBY RIGHT
        DEX             ;DECREMENT BIT COUNT
        BNE   NXTBT     ;LOOP UNTIL 8 BITS ARE DONE
        STA   $23       ;STORE PRODUCT MSBY
        RTS
```

ory locations. Further, multiplying a 16-bit value by another 16-bit value produces a 32-bit product, which occupies four memory locations.

Fig. 5-2 is a flowchart for a double-precision multiplication subroutine for unsigned numbers. The two bytes of the multiplicand

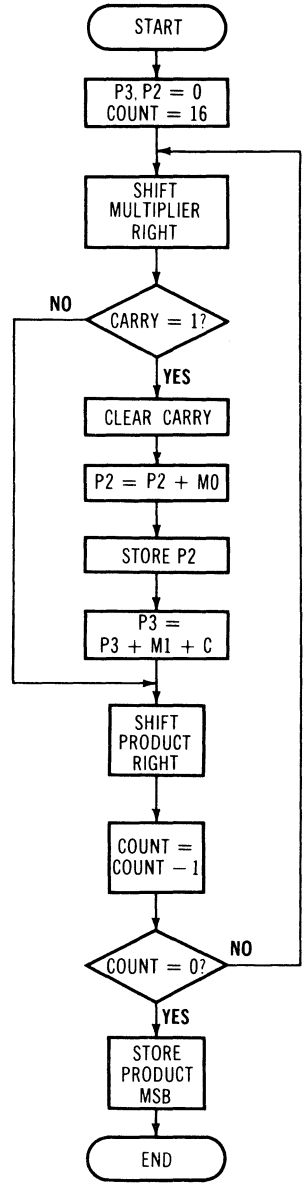


Fig. 5-2. A 16-bit × 16-bit multiplication algorithm.

are represented by the symbols M0 (low-order byte) and M1 (high-order byte). The four bytes of the product are represented by the symbols P0 (low-order byte), P1, P2, and P3 (high-order byte).

The double-precision algorithm shown in Fig. 5-2 operates in a manner similar to the single-precision algorithm that we just discussed. That is, the multiplicand is added to the high-order half of the partial product (P2 and P3) if Carry is a 1. The result is then right-shifted into the low-order half of the partial product (P0 and P1). With double-precision, however, the multiplicand is added in four steps:

1. Add low-order byte of multiplicand (M0) to P2.
2. Store P2.
3. Add high-order byte of multiplicand (M1) to P3, with any Carry out of P2.
4. Store new value of P3.

Shifting the partial product involves four separate operations:

1. Shift P3 right.
2. Shift P2 right, with Carry from P3.

Example 5-6: A 16-Bit by 16-Bit Multiplication Subroutine (With 32-Bit Result)

```
;THIS SUBROUTINE MULTIPLIES THE UNSIGNED CONTENTS OF LOCATIONS
;$22  (LOW) AND $23 (HIGH) BY THE UNSIGNED CONTENTS OF LOCATIONS
;$20 (LOW) AND $21 (HIGH), PRODUCING A 32-BIT UNSIGNED PRODUCT
;IN LOCATIONS $24 (LOW) THROUGH $27 (HIGH).

MLT16   LDA   #00     ;CLEAR P2 AND P3 OF PRODUCT
        STA   $26
        STA   $27
        LDX   #16     ;MULTIPLIER BIT COUNT = 16
NXTBT   LSR   $21     ;SHIFT TWO-BYTE MULTIPLIER RIGHT
        ROR   $20
        BCC   ALIGN   ;MULTIPLIER = 1?
        LDA   $26     ;YES. FETCH P2
        CLC           ;  AND ADD M0 TO IT
        ADC   $22
        STA   $26     ;STORE NEW P2
        LDA   $27     ;FETCH P3
        ADC   $23     ;  AND ADD M1 TO IT
ALIGN   LSR   A       ;SHIFT FOUR-BYTE PRODUCT RIGHT
        STA   $27     ;STORE NEW P3
        ROR   $26
        ROR   $25
        ROR   $24
        DEX           ;DECREMENT BIT COUNT
        BNE   NXTBT   ;LOOP UNTIL 16 BITS ARE DONE
        RTS
```

3. Shift P1 right, with Carry from P2.
4. Shift P0 right, with Carry from P1.

Since the multiplier contains 16 bits, the add-and-shift loop will be performed 16 times.

Example 5-6 is a subroutine (MLT16) that uses the flowcharted algorithm in Fig. 5-2 to perform a double-precision (16-bit) multiplication. The multiplier is in locations $20 and $21, the multiplicand is in locations $22 and $23, and the 32-bit product is returned in locations $24 (LSBY) through $27 (MSBY). As mentioned earlier, the subroutine performs two add operations in order to add the 16-bit multiplicand to the two high-order bytes of the product. It also performs four shift operations (one shift for each byte in the product), and passes data between bytes by means of the Carry.

INTEGER DIVISION

Integer division involves calculating the number of times that one integer number (the divisor) can be subtracted from another integer number (the dividend). The resulting number of subtractions is called the quotient, and the "leftover" amount is called the remainder. For example, the pencil-and-paper division of the number 1265 by the number 13 would look like the following:

```
     0097
13 | 1265
   -0
    12
   - 0
    126
   -117
     95
    -91
      4
```

The division consists of a series of trial subtractions, beginning with the leftmost (most significant) digit of the dividend. The steps are:

1. How many times can 13 be subtracted from 1? The answer is zero, so a 0 is entered in the quotient and the next digit, a 2, is brought down to form 12.
2. How many times can 13 be subtracted from 12? Again the answer is zero, so another 0 is entered in the quotient and the next digit, a 6, is brought down to form 126.

3. How many times can 13 be subtracted from 126? The answer is nine, so a 9 is entered in the quotient and 117 (9 times 13) is subtracted from 126.
4. The subtraction leaves a remainder of 9, so the final digit of the dividend, a 5, is brought down to form 95.
5. How many times can 13 be subtracted from 95? The answer is seven, so a 7 is entered in the quotient and 91 (7 times 13) is subtracted from 95.
6. All of the digits of the dividend have been tested, so the division is complete. The quotient is 97 (the leading zeroes are ignored) and the remainder is 4.

The dividing of binary numbers is similar to the dividing of decimal numbers, but it is much easier. With binary numbers, you never have to worry about multiples in the trial subtractions; if the divisor is less than or equal to the dividend, the appropriate quotient bit will be a 1, otherwise it will be a 0. The binary equivalent of the 1265 ÷ 13 problem is shown in Example 5-7. As with decimal division, a new quotient digit (1 or 0 for the binary version) is always entered to the right of the previously entered quotient digits. In a computer program, this is accomplished by shift-

Example 5-7: Binary Division

```
      00001100001
1101 | 10011110001
     -0
      10
     - 0
      100
     -  0
      1001
     -   0
      10011
     -  1101
       1101
     - 1101
         01
       - 0
         010
       -  0
         0100
       -   0
         01000
       -    0
         010001
       -  1101
            100
```

ing the old partial quotient to the left and entering a new quotient digit into the vacated least-significant bit position.

The computer implementation requires the dividend to be left-shifted, too, to form a partial dividend to which the divisor is compared (and, if possible, subtracted). The fundamental operations for dividing binary numbers by a computer are:

1. Shift the quotient left (initially zero) to provide a (least-significant) bit position for the next quotient digit.
2. Shift the dividend left, so that another bit from the partial dividend is tested.
3. Compare the divisor to the partial dividend.
4. If the divisor is less than or equal to the partial dividend, subtract the divisor from the partial dividend and enter a 1 in the quotient.
5. If any digits remain in the dividend, return to Step 1.

Dividing Unsigned Numbers

The flowchart shown in Fig. 5-3 applies these rules in dividing an 8-bit unsigned dividend in memory by an 8-bit unsigned divisor in memory. The 8-bit quotient and the 8-bit remainder are then

Example 5-8: An 8-Bit by 8-Bit Unsigned Division Subroutine

```
;THIS SUBROUTINE DIVIDES AN 8-BIT UNSIGNED DIVIDEND IN LOCATION
;$21 BY AN 8-BIT UNSIGNED DIVISOR IN LOCATION $20. THE 8-BIT
;QUOTIENT IS RETURNED IN LOCATION $21, REPLACING THE DIVIDEND,
;AND THE 8-BIT REMAINDER IS RETURNED IN LOCATION $22.

DIV8U   LDA   #00     ;CLEAR PARTIAL DIVIDEND
        LDX   #08     ;DIVIDEND BIT COUNT = 8
NXTBT   ASL   $21     ;SHIFT DIVIDEND/QUOTIENT LEFT
        ROL   A       ;  INTO PARTIAL DIVIDEND
        CMP   $20     ;COMPARE DIVISOR TO PARTIAL DIVIDEND
        BCC   CNTDN   ;DIVISOR > PARTIAL DIVIDEND?
        SBC   $20     ;NO. SUBTRACT DIVISOR
        INC   $21     ;  AND SET BIT IN QUOTIENT
CNTDN   DEX           ;DECREMENT BIT COUNT
        BNE   NXTBT   ;LOOP UNTIL 8 BITS ARE DONE
        STA   $22     ;STORE REMAINDER
        RTS
```

stored in two consecutive memory locations. Note that no action is required to enter a 0 into the quotient (required if the divisor is greater than the dividend); left-shifting the quotient at the beginning of the algorithm automatically does this for you!

Example 5-8 illustrates a subroutine (DIV8U) that uses the flowcharted algorithm of Fig. 5-3 to divide the contents of location $21 by the contents of location $20. It then returns the product

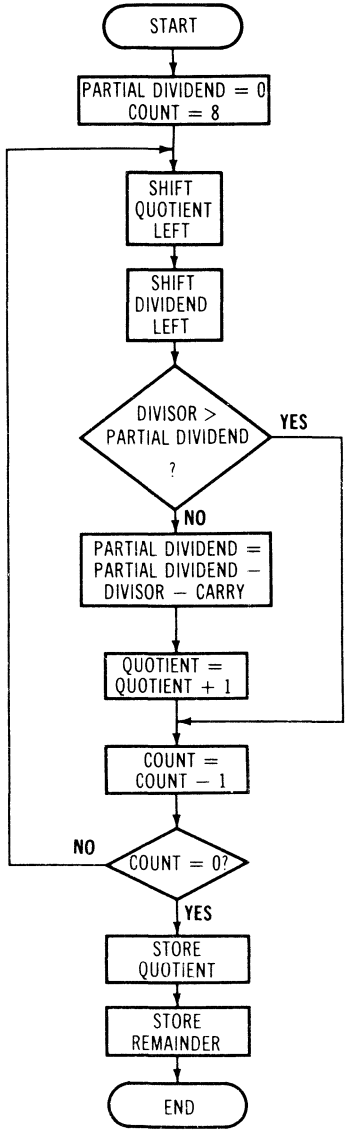


Fig. 5-3. An 8-bit binary division flowchart.

and remainder to locations $21 and $22, respectively. The X register is used to maintain the dividend bit count.

The ASL $21 and ROL A instructions cause location $21 and the accumulator to function as a 16-bit shift register, as illustrated in Fig. 5-4. Location $21 is used to store both the dividend and

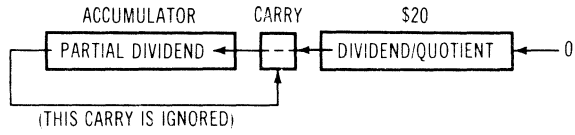


Fig. 5-4. Using the accumulator and location $20 as a 16-bit shift register.

the quotient, with quotient bits replacing dividend bits that have been vacated during shifting. The accumulator is used to store the partial dividend.

The CMP $20 instruction compares the divisor in location $20 to the partial dividend in the accumulator. If the divisor is larger than the partial dividend, the Carry is cleared to 0 and BCC CNTDN causes a branch past the SBC $20 instruction. Otherwise, the Carry is set to a 1 and the BCC CNTDN instruction test fails, continuing execution at SBC $20. At SBC $20, the divisor is subtracted from the partial dividend and a 1 is entered into the quotient (INC $21). After the NXTBT loop is executed eight times (once for each bit in the dividend), STA $22 stores the remainder in location $22.

Dividing Signed Numbers

Recall that in signed multiplication, the multiplier must always be positive. *In signed division, both operands, the divisor and the dividend, must be positive.* The easiest way to do this in software is to convert both the divisor and dividend to positive numbers, perform the division operation, and then convert the quotient and the remainder to the proper sign. If the operands are of the same sign (both positive or both negative), the quotient will be positive; otherwise, it will be negative. The remainder will always have the same sign as the dividend. The four possible sign combinations of quotient and remainder, after a division has been performed, are given in the following list.

Dividend	Divisor	Quotient	Remainder
+	+	+	+
+	−	−	+
−	+	−	−
−	−	+	−

Example 5-9 shows a subroutine (DIV8S) that performs an 8-bit signed division, using the divisor in $20 and the dividend in $21. This subroutine is comprised of three sections. The first section (DIV8S through GODIV) checks the sign of the divisor and dividend. If either operand is negative, the operand is subtracted from zero, which makes it positive. Further, since the original

Example 5-9: An 8-Bit by 8-Bit Signed Division Subroutine

```
;THIS SUBROUTINE DIVIDES AN 8-BIT SIGNED DIVIDEND IN LOCATION $21
;BY AN 8-BIT SIGNED DIVISOR IN LOCATION $20. THE 8-BIT QUOTIENT IS
;RETURNED IN LOCATION $21, REPLACING THE DIVIDEND, AND THE 8-BIT
;REMAINDER IS RETURNED IN LOCATION $22. LOCATION $23 IS USED TO HOLD
;A DIVISOR/DIVIDEND SIGN FLAG.

DIV8S   LDY  #00      ;SIGN FLAGS = 0
        BIT  $20      ;DIVISOR POSITIVE?
        BPL  CHKDD
        TYA           ;NO. CLEAR ACCUMULATOR
        SEC           ;  AND NEGATE DIVISOR
        SBC  $20
        STA  $20
        LDY  #80      ;SIGN FLAG BIT 7 = 1
;IF DIVIDEND IS NEGATIVE, MAKE IT POSITIVE
CHKDD   BIT  $21      ;DIVIDEND POSITIVE?
        BPL  GODIV
        LDA  #00      ;NO. CLEAR ACCUMULATOR
        SEC           ;  AND NEGATE DIVIDEND
        SBC  $21
        STA  $21
        TYA           ;SIGN FLAG BIT 6 = 1
        ORA  #40
        TAY
;THE DIVISION ROUTINE FOLLOWS
GODIV   STY  $23      ;STORE SIGN FLAGS
        LDA  #00
        LDX  #08
NXTBT   ASL  $21
        ROL  A
        CMP  $20
        BCC  CNTDN
        SBC  $20
        INC  $21
CNTDN   DEX
        BNE  NXTBT
        STA  $22
;DIVISION COMPLETE. PUT QUOTIENT AND REMAINDER IN PROPER FORM
        LDA  #$C0     ;TEST SIGN FLAGS
        BIT  $23      ;QUOTIENT AND REMAINDER IN PROPER FORM?
        BEQ  NOMOR    ;YES. RETURN
        BVS  NEGR     ;NO. NEGATE REMAINDER
NEGQ    LDA  #00      ;NO. NEGATE QUOTIENT
        SEC
        SBC  $21
        STA  $21
        RTS
NEGR    LDA  #00      ;NEGATE REMAINDER
        SEC
        SBC  $22
        STA  $22
        BIT  $23      ;DOES QUOTIENT NEED NEGATING TOO?
        BPL  NEGQ     ;YES. GO NEGATE QUOTIENT
NOMOR   RTS           ;NO. RETURN
```

signs of the operands determine the signs of the quotient and remainder, these signs are recorded in the Y register. The original sign of the divisor is recorded in Bit 7, and the original sign of the dividend is recorded in Bit 6. These particular bits were selected because they are readily testable by a BIT instruction (following division).

The second section of the subroutine (GODIV through the ";DIVISION COMPLETE" comment) is nothing more than the DIV8U subroutine. It is preceded by an instruction that stores the Y register sign flags in memory (STY $23). If DIV8U is already in memory, this entire sequence of instructions can be replaced by just two instructions—STY $23 and JSR DIV8U.

The final section of the DIV8S subroutine uses the BIT $23 instruction to determine the state of the original signs of the divisor and dividend. It then alters the quotient and remainder, if required to put them in proper form.

Multiple-Precision Division

The "shift and subtract" method described for single-precision division can be modified for *multiple-precision division.* The fundamental operations of the division algorithm—shift, compare, and subtract—are unchanged with multiple-precision division, but some instructions must be added so that the multiple bytes of the divisor and quotient are accessed properly.

The DIV16 subroutine in Example 5-10 operates with a 16-bit unsigned divisor and dividend, each occupying two consecutive memory locations (with the LSBYs stored in low-address memory). As with the single-byte unsigned division subroutine (Example 5-8), the dividend will be replaced by the quotient as it is shifted to the left.

After clearing the 16-bit partial dividend and loading the dividend bit count into the X register, the subroutine performs a four-byte left shift on the dividend and the partial dividend. You will recall that in the single-precision DIV8 routine, the left shift was followed by a comparison of the divisor to the partial dividend. Because the CMP instruction can compare only single bytes, we must use a different technique to compare a multibyte divisor to a multibyte dividend. The approach used in the DIV16 subroutine is to actually perform a 16-bit subtraction, but enter the result into memory only if the divisor is less than or equal to the partial dividend.

The low-order bytes are subtracted first, with the sequence LDA $24, SEC and SBC $20. The result is saved in the Y register (by TAY). The high-order bytes are then subtracted, with LDA $25 and SBC $21. If the subtraction of the MSBYs generates a borrow

Example 5-10: A 16-Bit by 16-Bit Unsigned Division Subroutine

```
;THIS SUBROUTINE DIVIDES A 16-BIT UNSIGNED DIVIDEND IN LOCATIONS
;$22 AND $23 BY A 16-BIT UNSIGNED DIVISOR IN LOCATIONS $20 AND
;$21. THE 16-BIT QUOTIENT REPLACES THE DIVIDEND. THE 16-BIT
;REMAINDER IS RETURNED IN LOCATIONS $24 AND $25. THE LOW-ORDER
;BYTE OCCUPIES THE LOW ADDRESS IN ALL CASES.

DIV16   LDA   #00     ;CLEAR PARTIAL DIVIDEND
        STA   $24
        STA   $25
        LDX   #16     ;DIVIDEND BIT COUNT = 16
NXTBT   ASL   $22     ;SHIFT DIVIDEND/QUOTIENT LEFT
        ROL   $23
        ROL   $24     ;SHIFT PARTIAL DIVIDEND LEFT
        ROL   $25
        LDA   $24     ;SUBTRACT LOW BYTES
        SEC
        SBC   $20
        TAY           ;SAVE LOW RESULT IN Y
        LDA   $25     ;SUBTRACT HIGH BYTES
        SBC   $21
        BCC   CNTDN   ;DIVISOR > DIVIDEND?
        INC   $22     ;NO. SET BIT IN QUOTIENT
        STA   $25     ;  AND ENTER SUBTRACTION RESULT
        STA   $24     ;  INTO PARTIAL DIVIDEND
CNTDN   DEX           ;DECREMENT BIT COUNT
        BNE   NXTBT   ;LOOP UNTIL 16 BITS ARE DONE
        RTS
```

(Carry = 0), then BCC CNTDN causes a branch to DEX, and BNE NXTBT loops back to test the next bit. In the absence of a borrow (Carry = 1), BCC CNTDN fails, setting the quotient bit (INC $22) and entering the 16-bit result of the subtraction into the partial dividend (STA $25 and STA $24).

Although DIV16 is designed for 16-bit division, larger numbers can be divided in a similar fashion, but the low-order results of their subtraction will have to be saved either on the stack or in separate memory locations. Which is more efficient? Let us investigate the alternatives.

Using memory for temporary storage will require one STA instruction to store a byte (2 instruction bytes, 3 cycles) and, if the byte must be entered into the partial dividend, an LDA instruction to fetch the byte (2 bytes, 3 cycles). Also, another STA instruction is needed to store it into the partial dividend (2 bytes, 3 cycles). Therefore, using memory for temporary storage will take up to 6 instruction bytes and 9 execution cycles.

Using the stack for temporary storage will require one PHA instruction to push the byte onto the stack (1 byte, 3 cycles) and, if the byte must be entered into the partial dividend, a PLA instruction to pull it from the stack (1 byte, 4 cycles). Then, an STA

instruction (2 bytes, 3 cycles) is needed to store it into the partial dividend. Therefore, using the stack for temporary storage will take up to 4 instruction bytes and 10 execution cycles.

With both approaches (memory and stack), if the subtraction result is not entered into the partial dividend (i.e., if BCC CNTDN succeeds, causing a branch to CNTDN), the subtraction result can be ignored—it can either be left on the stack or stored in memory. Leaving this result in memory is normally no problem, but leaving it on the stack will cause the subroutine to return to an improper location. However, you can remedy this by saving the initial Stack Pointer value of the calling program in memory (with a TSX and STX combination) at the beginning of the subroutine, and then restoring the Stack Pointer (with LDX and TXS) before executing the return.

BCD MATHEMATICS

Until now, we have been performing mathematical operations on binary numbers. Although a substantial amount of programming is done with binary numbers (usually in their hexadecimal form), there are certain situations when it is advantageous to use decimal numbers.

If numbers are entered into the 6502 microprocessor using a terminal or teletypewriter (or the AIM 65 keyboard), the microprocessor will receive an ASCII code for each key that is pressed. Since the 6502 is not designed to add, subtract, multiply, or divide ASCII coded data directly, the ASCII data must be converted into either binary or binary-coded decimal (BCD) form, which can be processed by the 6502 microprocessor. Chapter 6 will discuss how to convert ASCII decimal digits to their BCD equivalents but, for now, assume that the conversion has been made and that the data is stored in memory in BCD form.

Because we have been raised in a decimal world, the minor inconvenience of dealing with binary bit patterns can be softened considerably if they are converted to a nice, easy-to-understand decimal number. If you have worked with hexadecimal numbers in your programming up to this point, you have probably cursed the fact that a binary pattern like 1101 does not convert to anything rational like a 6 or a 9, but instead converts to a D. Then, you had to mentally calculate, "What is a D in the real world? Well, if A is a 10, then B is an 11, C is a 12, and D must be a 13." By using BCD numbers in our calculations, this type of conversion does not have to be performed.

The designers of the 6502 recognized a need for dealing with BCD numbers, and they answered this need with two special in-

structions that were designed exclusively for BCD arithmetic. These instructions, Set Decimal Mode (SED) and Clear Decimal Mode (CLD), cause the Arithmetic Logic Unit of the 6502 to operate as a decimal adder when BCD data is being processed. In a decimal adder, the contents of a half-byte (a *nibble*) will not exceed binary 1001. Therefore, if the nibble contains binary 1001 and is incremented, its new value will be binary 0000 (with a Carry to the next nibble) and not binary 1010.

As was discussed in Chapter 2, the Set Decimal Mode (SED) instruction causes the 6502 microprocessor to interpret the operands of all subsequent Add with Carry (ADC) and Subtract with Carry (SBC) instructions as BCD operands. The Clear Decimal Mode (CLD) instruction causes the ALU to revert to functioning as a straight binary adder. Therefore, the addition and subtraction subroutines given in Examples 5-1 and 5-2 can be used to operate on BCD numbers by simply preceding the arithmetic operation with an SED instruction and following it with a CLD instruction. Example 5-11 is the BCD equivalent of our previous multiprecision integer addition routine (Example 5-1). Note that the SED and CLD instructions are inserted outside of the NXTBY addition loop.

Example 5-11 can be converted into a multiple-precision *subtraction* subroutine by simply replacing the CLC with an SEC and the ADC $51,X with an SBC $51,X. BCD numbers can be multiplied and divided too, but because bits rather than bytes are being processed in these operations, BCD representation makes them very complicated. For multiplication and division, it is much easier to convert the operands to binary (see Chapter 6) before performing either of these operations.

Example 5-11: A Multiple-Precision BCD Addition Subroutine

```
;THIS SUBROUTINE ADDS TWO MULTIPLE-BYTE BCD NUMBERS, ONE STARTING
;IN LOCATION $21, THE OTHER STARTING IN LOCATION $51. THE RESULT
;REPLACES THE NUMBER THAT STARTS IN LOCATION $21. THE BYTE
;COUNT IS CONTAINED IN LOCATION $20.

MPAB    SED             ;SET DECIMAL MODE
        LDY   $20       ;FETCH BYTE COUNT
        LDX   #00       ;AT START, INDEX = 0
        CLC             ;  AND CARRY = 0
NXTBY   LDA   $21,X     ;ADD 8 BITS
        ADC   $51,X
        STA   $21,X     ;STORE 8 BITS
        INX             ;INCREMENT INDEX AND COUNT
        DEY
        BNE   NXTBY     ;LOOP UNTIL ALL BYTES DONE
        CLD             ;CLEAR DECIMAL MODE
        RTS
```

FLOATING-POINT MATHEMATICS

The integer and BCD routines covered earlier in this chapter are fine for simple everyday arithmetic, but how would you use them to operate on a very large number such as 16,000,000,000,000, or on a very small number (a fraction) such as 0.000000000003674? With enough memory, numbers like these could be stored and processed by the methods that we have already discussed. However, our "large" number is 14 digits long, which (at two BCD digits per memory location) would require seven memory locations for storage. The situation is just as gloomy for our "small" fraction.

Of course, nobody goes to the extremes of writing out a 16 and 12 zeroes to represent 16 trillion, much less trying to store it in memory that way. Rather, a shorthand notation is usually used in which very large (or very small) numbers are written as a fractional number multiplied by some power of 10. Realizing that $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$, and so on, we can quickly observe that each *positive* exponent of 10 has a direct correlation to the number of digits to the *left* of the decimal point in a number. Similarly, each *negative* exponent of 10 (10^{-1}, 10^{-2}, and so on) has a direct correlation to the number of digits to the *right* of the decimal point. Thus, by using this short form of notation, our "large" number can be written in any of several equivalent forms:

$$16.000 \times 10^{12} \qquad 1.6000 \times 10^{13} \qquad 0.16000 \times 10^{14}$$

Similarly, our fractional number can also be written in a number of equivalent forms:

$$36.74 \times 10^{-13} \qquad 3.674 \times 10^{-12} \qquad 0.3674 \times 10^{-11}$$

The numbers shown in the center and on the right in the preceding examples represent two common notational forms. The center number is given in a format that is commonly called "scientific notation." In scientific notation, numbers are written with one significant (nonzero) digit to the left of the decimal point. The number shown on the right is written in "floating-point notation." In floating-point notation, all digits are written to the right of the decimal point; thus, the numbers are written as fractional numbers.

When calculating by hand, most people use a form of scientific notation, probably because we feel a bit uncomfortable about working with fractions. However, floating-point notation is much better suited for use with computer software. Why? Because a fractional number can be processed with the same arithmetic instructions that are used to process integer numbers; to the arithmetic logic of a computer, there is no difference between the integer 1234 and the fractional number 0.1234.

However, before discussing floating-point programming, let us look closely at how floating-point numbers are operated on using a pencil and paper. To start, consider the following addition:

$$\begin{array}{r} 0.10376 \times 10^6 = 103{,}760_{10} \\ +0.84860 \times 10^4 = \quad 8{,}486_{10} \\ \hline \end{array}$$

The addition cannot yet be performed because the exponents are different; one number is multiplied by 10^6, the other is multiplied by 10^4. To make the exponents agree, and still keep both numbers fractional, the exponent of the smaller number is incremented by one and the number is shifted to the right, until the exponents are the same. Thus, the smaller number in our example will require two such increment-and-shift operations. Therefore:

$$0.84860 \times 10^4 = 0.084860 \times 10^5 = 0.0084860 \times 10^6$$

After doing the above increment-and-shift operations, the addition becomes:

$$\begin{array}{r} 0.103760 \times 10^6 \\ +0.008486 \times 10^6 \\ \hline 0.112246 \times 10^6 \end{array}$$

The same type of exponent adjustment must be made if floating-point numbers are to be subtracted. In summary, we can state this rule: *Before adding or subtracting floating-point numbers, the exponents must be identical.* (To make them identical, shift the number with the smaller exponent to the right, incrementing the exponent with each shift, until the exponents agree.) While we are on the subject of rules, here is one more: *If an addition produces a Carry, shift the sum right one position and increment the exponent.* The Carry here represents an overflow out of the fractional number position, and the shift and increment pull it back to the right of the decimal point.

Multiplying and dividing floating-point numbers is simpler in at least one respect—there is no need to align the exponents. *To multiply floating-point numbers, simply multiply the fractions and add the exponents.* The following is an example:

$$\begin{aligned} &(0.5011 \times 10^{12}) \times 0.3764 \times 10^8) \\ &= (0.5011 \times 0.3764) \times (10^{12+8}) \\ &= 0.18816404 \times 10^{20} \end{aligned}$$

To divide floating-point numbers, divide the dividend by the divisor and subtract the exponent of the divisor from the exponent of the dividend. The following is an example:

$$(0.1936 \times 10^6) \div (0.1017 \times 10^4)$$
$$= (0.1936 \div 0.1017) \times (10^{6-4})$$
$$= 1.9036381 \times 10^2$$
$$= 0.19036381 \times 10^3$$

Most floating-point routines also *normalize* the result (and usually the operands). Normalizing merely means adjusting the fractional number so that the high-order digit has a nonzero value. For example,

$$0.006355 \times 10^{-13}$$

would be normalized to

$$0.6355 \times 10^{-15}$$

As you can see, *to normalize a floating-point number, shift the fractional number left, and decrement the exponent by one for each position shifted.*

That is enough of the basic principles. Let us see how you would store floating-point numbers in memory. For each floating-point number, four separate items of information must be stored—the sign of the fractional number, the value of the fractional number, the sign of the exponent, and the value of the exponent. You are free to use any configuration that feels comfortable to you. The following are two variation:

S_eS_fEEE	.FFFF	FFFF	(First variation)
EE	S_eS_f.FFF	FFFF	(Second variation)

In both cases, S_e and S_f represent the 1-bit signs of the exponent and the fractional number, respectively. A "1" in the sign bit indicates a negative exponent or fractional number, a "0" in the sign bit represents a positive exponent or fractional number. Further, each "E" represents one digit in the exponent and each "F" represents one digit in the fractional number.

You will recall, from the BCD portion of this chapter, that BCD digits can be stored in a "packed" form, two digits per memory byte. Therefore, the upper example (First variation) will occupy six bytes—four for the fractional number and two for the exponent. The two sign bits (S_e and S_f) are stored in Bits 7 and 6 of the location that contains the high-order exponent digit (Bit positions 5 and 4 are used). The lower example (Second variation) will occupy five bytes—four for the fractional number and one for the exponent—with the sign bits residing in Bits 7 and 6 of the location containing the high-order fractional number digit. Which is better? There is no "correct" answer; it will vary with the requirements of each particular application.

SQUARE ROOT

An interesting observation made a few years ago provides us with a simple way of calculating square roots. The observation is this: *The square root of an integer is equal to the number of successively higher odd numbers that can be subtracted from it.* Example 5-12 shows how the square root of 25 can be extracted

Example 5-12: Obtaining a Square Root by Using Odd-Number Subtractions

```
  25
- 1        Partial square root = 1
  24
- 3        Partial square root = 2
  21
- 5        Partial square root = 3
  16
- 7        Partial square root = 4
   9
- 9        Square root         = 5
   0
```

using this method. (Sceptics will want to try a few additional cases of their own.) In this example, a total of five odd numbers—1, 3, 5, 7, and 9—can be subtracted from 25, yielding a square root of 5.

Subroutine SQRT8 (Example 5-13) employs this algorithm to take the square root of the unsigned integer in location $20. Since the square root accumulates at a rate of one count per pass through the AGAIN loop, this subroutine is not particularly fast, but it occupies only 27 bytes in memory.

Example 5-13: A Simple 8-Bit Square Root Subroutine

```
;THIS SUBROUTINE TAKES THE SQUARE ROOT OF THE UNSIGNED INTEGER
;IN LOCATION $20. THE SQUARE ROOT IS RETURNED IN LOCATION $20,
;THE REMAINDER IN LOCATION $21.

SQRT8   LDY   #00     ;SQUARE ROOT = 0
        LDA   #01     ;FIRST ODD NUMBER = 1
        STA   $21
        LDA   $20     ;FETCH INTEGER NUMBER
AGAIN   CMP   $21     ;CAN A SUBTRACTION BE MADE?
        BCC   NOMORE
        SBC   $21     ;YES. MAKE THE SUBTRACTION
        INY           ;INCREMENT SQUARE ROOT
        INC   $21     ;  AND GO TRY NEXT ODD NUMBER
        INC   $21
        JMP   AGAIN
NOMORE  STY   $20     ;ALL DONE, STORE SQUARE ROOT
        STA   $21     ;  AND REMAINDER
        RTS
```

Example 5-14: A Simple 16-Bit Square Root Subroutine

```
;THIS SUBROUTINE TAKES THE SQUARE ROOT OF A DOUBLE-PRECISION
;INTEGER IN LOCATIONS $20 (LOW) AND $21 (HIGH). THE 8-BIT
;SQUARE ROOT IS RETURNED IN LOCATION $20, THE REMAINDER IN
;LOCATION $21.

SQRT16  LDY  #01     ;LSBY OF FIRST ODD NUMBER = 1
        STY  $22
        DEY
        STY  $23     ;MSBY OF FIRST ODD NUMBER
                     ;  (AND SQUARE ROOT) = 0
AGAIN   SEC
        LDA  $20     ;SAVE REMAINDER IN X REGISTER
        TAX          ;SUBTRACT ODD LO FROM INTEGER LO
        SBC  $22
        STA  $20
        LDA  $21     ;SUBTRACT ODD HI FROM INTEGER HI
        SBC  $23
        STA  $21     ;IS SUBTRACT RESULT NEGATIVE?
        BCC  NOMORE  ;NO. INCREMENT SQUARE ROOT
        INY
        LDA  $22     ;CALCULATE NEXT ODD NUMBER
        ADC  #01
        STA  $22
        BCC  AGAIN
        INC  $23
        JMP  AGAIN
NOMORE  STY  $20     ;ALL DONE, STORE SQUARE ROOT
        STX  $21     ;  AND REMAINDER
        RTS
```

Subroutine SQRT16 (Example 5-14) is a double-precision version of SQRT8. It uses locations $20 and $21 to hold the 16-bit integer and locations $22 and $23 to hold the odd numbers as they are accumulated. Because a two-byte number is involved, a true subtraction, rather than a compare, must be performed. Since the final subtraction will destroy the remainder, the remainder (in the low-order integer location) is saved in the X register prior to the byte subtract operation.

The calculation of a new odd number involves the use of a little programming trick. After fetching the low-order byte of the odd number (LDA $22), it seems to be increasing by one (ADC #01), rather than by two. It is, however, actually increased by two, since the branch BCC NOMORE drops through to INY *only* if the Carry flag is set—so ADC #01 is adding a "1" immediate and a Carry = 1. This trick saves clearing the Carry before the ADC instruction.

6

Number-Base Conversion

In all computer systems, there must be a way of communicating digital information between the processor and the external devices. That is, there must be a way to *input* data from a teletypewriter, a card reader, a digital cassette, a floppy disk, or a keyboard or numeric keypad. There must also be a way to *output* data to a printer, a display, a card punch, a digital cassette, a floppy disk, or a teletypewriter.

The data must, of course, be in a form that is recognizable to the recipient (either the peripheral device or the 6502 microprocessor). Depending on its design, a preipheral device can operate on data in one of any number of forms, including ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary-Coded Decimal Interchange Code), binary, or Gray Code. The 6502 microprocessor operates on numeric data in either of two forms, binary or BCD (Binary-Coded Decimal), depending on the state of the Decimal Mode bit (D) in the processor status register.

Although many computer systems convert data with special electronic circuits, most smaller systems use subroutines to perform this conversion. This chapter presents a number of typical number-base conversion subroutines. The discussion will be limited to ASCII, the most commonly used data type for peripheral devices, and to BCD and binary numbers, but the principles learned here can be used to develop conversion subroutines for other data forms and number bases. Furthermore, we will assume all of our character-coded data to be 7-bit ASCII, with the most significant bit zero (see Appendix A).

TWO SIMPLE I/O DEVICES

Up to this point in the book, there has not been any detailed discussion on how data are input or output by the microprocessor,

nor has there been any discussion about how a peripheral device can be *interfaced* (electrically connected) to the 6502 microprocessor. Neither topic will be given an extensive treatment in this chapter either, but it is necessary to introduce a small amount of input/output *programming* here in order to put the conversion subroutines into perspective. After all, data are not entered into the 6502 out of "thin air;" they come from real devices that communicate with the 6502 microprocessor in a definite fashion.

A Simple Input Device (Keyboard)

A keyboard has been selected to represent a typical input device. Let us assume that this keyboard communicates with the 6502 microprocessor through two addresses, $A275 and $A276. When the 6502 places either of these addresses on the *address bus* while a read (i.e., load) operation is being performed, the keyboard interface hardware places information on the *data bus* of the microcomputer. This information is usually read into the accumulator. As far as the 6502 microprocessor knows, it is reading information from memory; however, we know that the information is being read from an I/O (peripheral) device—the keyboard. Therefore, we have just described *memory-mapped I/O*. Of course, we can also "write information out" to a peripheral by placing an address on the address bus and data on the data bus while performing a memory write (i.e., store) operation.

However, to return to our keyboard example, address $A275 represents an 8-bit keyboard status register in which only the most-significant bit (Bit 7) is used. Bit 7 represents a status indicator that is set to logic 1 when a key (*any* key) is pressed. When a key is pressed, a hardware device in the interface identifies the key and loads its ASCII value into a second 8-bit register, which has been assigned the address $A276.

Example 6-1 shows a subroutine (KEYIN) that inputs ASCII characters from the keyboard into the accumulator. This subroutine simply waits for a key to be pressed (waits for Bit 7 of $A275 to become logic 1), and then loads the ASCII character into the ac-

Example 6-1: A Simple Keyboard Input Subroutine

```
;THIS SUBROUTINE INPUTS AN ASCII CODE FROM LOCATION $A276 TO
;THE ACCUMULATOR. LOCATION $A275 CONTAINS A "KEY PRESSED"
;STATUS FLAG IN BIT 7.

KEYIN   BIT   $A275  ;KEY PRESSED?
        BPL   KEYIN  ;NO. WAIT UNTIL IT IS.
        LDA   $A276  ;YES. FETCH ASCII CODE
        LSR   $A275  ;  AND CLEAR STATUS FLAG
        RTS
```

cumulator. It then clears the status flag and returns. An AIM 65 owner who wishes to convert ASCII characters from the keyboard of the microcomputer can use the READ or REDOUT subroutine of the AIM 65 Monitor in place of the KEYIN used for each of the examples in this chapter. The READ subroutine (entry address $E93C) just inputs the keyboard character into the accumulator. The REDOUT subroutine (entry address $E973) not only loads the character into the accumulator, but it also displays and prints it.

A Simple Output Device (Printer)

A printer has been selected to represent a typical output device. Assume that the 6502 microcomputer communicates with the printer through two addresses, $A277 and $A278. Address $A277 represents the 8-bit printer status register, in which only the most-significant bit (Bit 7) is used. Bit 7 holds a status indicator that is set to logic 1 when the printer is ready to receive an ASCII character for printing. When Bit 7 is a logic 1, the ASCII character must be stored into location $A278.

Example 6-2 shows a subroutine (PTROUT) that outputs the contents of the accumulator to the printer. This subroutine waits for the printer to signal that it is ready to accept an ASCII character (Bit 7 of $A277 is a logic 1), stores the character into location $A278, and returns. The OUTPUT subroutine (entry address $E97A) in the AIM 65 Monitor outputs the accumulator contents to both the display and the printer.

Example 6-2: A Simple Printer Output Subroutine

```
;THIS SUBROUTINE OUTPUTS AN ASCII CODE FROM THE ACCUMULATOR TO
;LOCATION $A278. LOCATION $A277 CONTAINS A "PRINTER READY"
;STATUS FLAG IN BIT 7.

PTROUT  BIT   $A277    ;PRINTER READY?
        BPL   PTROUT   ;NO. WAIT UNTIL IT IS
        STA   $A278    ;YES. OUTPUT ASCII CODE
        LSR   $A277    ;  AND CLEAR STATUS FLAG
        RTS
```

TWO-DIGIT ASCII-BASED HEXADECIMAL-TO-BINARY CONVERSION

A substantial amount of 6502 microprocessor programming is done using hexadecimal data. If this data is entered from a keyboard, it might be input in ASCII form. Since the 6502 microprocessor operates on binary data, some way is needed to convert ASCII-based hexadecimal characters to 8-bit binary values. Table 6-1 shows the binary and ASCII form for each of the 16 hexadecimal characters.

Table 6-1. The Hexadecimal Numbering System

Hexadecimal Character	Binary Value	ASCII Value
0	0000	30
1	0001	31
2	0010	32
3	0011	33
4	0100	34
5	0101	35
6	0110	36
7	0111	37
8	1000	38
9	1001	39
A	1010	41
B	1011	42
C	1100	43
D	1101	44
E	1110	45
F	1111	46

This table shows us what conversion is needed, and gives us a few ideas about how to do it. For starters, note that for the values 0 to 9, the low-order hex digit of the ASCII value is identical to the hex character. Obviously, all we will have to do for this range of values is to mask out the high-order hex digit of the ASCII value, which is a 3. If we mask out the high-order hex digit, the 4, of the ASCII value for the remaining values of A to F, the low-order digit is exactly nine less than the required hex value ($1 + $9 = $A, $2 + $9 = $B, etc.).

Subroutine AH2B in Example 6-3 converts two consecutive ASCII-based hexadecimal characters that are entered on the keyboard into an 8-bit binary number. Why *two* characters? Because each hexadecimal character converts to a four bit binary value (0000 through 1111) rather than converting one 4-bit binary value in memory (and thus wasting four bits in every memory location), we convert two ASCII characters and pack the two resulting 4-bit values into a single memory location. One digit is stored in the four most-significant bits, the other digit is stored in the four least-significant bits.

The AH2B subroutine begins by calling an input subroutine, NEWHD, to get the most-significant digit. The NEWHD subroutine uses keyboard input subroutine KEYIN (Example 6-1) to load the keyboard character into the accumulator. It then checks to see whether this character represents one of the 16 valid hexadecimal characters, 0 to 9, or A to F. The ASCII values for hexadecimal characters range from $30 to $39 (0 to 9), and from $41 to $46 (A to F), as can be seen in Table 6-1. Therefore, NEWHD

Example 6-3: An ASCII-Based Hexadecimal-to-Binary Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS A TWO-DIGIT STRING OF HEXADECIMAL ASCII
;CHARACTERS TO AN 8-BIT BINARY VALUE IN THE ACCUMULATOR. LOCATION
;$30 IS USED FOR TEMPORARY STORAGE.

AH2B    JSR   NEWHD   ;INPUT MOST-SIGNIFICANT DIGIT
        ASL   A       ;SHIFT IT INTO THE FOUR MSB'S
        ASL   A
        ASL   A
        ASL   A
        STA   $30     ;SAVE THIS DIGIT IN LOCATION $30
        JSR   NEWHD   ;INPUT LEAST-SIGNIFICANT DIGIT
        ORA   $30     ;  AND INSERT MOST-SIGNIFICANT DIGIT
        RTS

;THE SUBROUTINE BELOW INPUTS THE NEXT VALID HEXADECIMAL DIGIT,
;AND RETURNS WITH THIS DIGIT IN THE FOUR LSB'S OF THE ACCUMULATOR.

NEWHD   JSR   KEYIN   ;FETCH DIGIT
        CMP   #$30    ;CHARACTER LESS THAN $30?
        BCC   NEWHD
        CMP   #$47    ;NO. IS IT MORE THAN $46?
        BCS   NEWHD
        CMP   #$3A    ;NO. IS IT BETWEEN 0 AND 9?
        BCS   A2F
        AND   #$0F    ;YES. MASK OFF THE FOUR MSB'S
        RTS
A2F     CMP   #$41    ;CHARACTER BETWEEN A AND F?
        BCC   NEWHD
        SBC   #$37    ;YES. SUBTRACT $37
        RTS
```

determines whether the keyboard character falls within one of these two ranges (and is, therefore, a valid hex digit), or is not in either range (and, thus, is not a hex digit). If the character is valid and represents a hexadecimal digit between 0 and 9, the subroutine masks off the four most-significant bits, leaving the hexadecimal digit in the four least-significant bits. If the character is valid and represents a hexadecimal digit between A and F, the subroutine subtracts $37 from the ASCII value. If the character is invalid, it is ignored, and the subroutine branches to the JSR KEYIN instruction at NEWHD to fetch another keyboard entry. The branch-on-error operation is arbitrary; the subroutine could have been designed to perform some other operation on an invalid entry, such as returning to the calling program with some type of error indication, like setting the carry flag.

If the keyboard entry represents a valid hexadecimal character, the value in the four LSBs of the accumulator is left-shifted four times (ASL A) when the 6502 microprocessor returns from NEWHD. This moves the digit into the most-significant digit posi-

tion, where it is stored in location $30. The NEWHD subroutine is then called again, to input the least-significant hexadecimal digit. All that remains is to combine these two digits. This is performed by the instruction ORA $30. The 6502 microprocessor then returns from the AH2B subroutine, with the 8-bit binary value in the accumulator.

ASCII-Based Hexadecimal-to-Binary Conversion Using the AIM 65

AIM 65 owners have a hexadecimal-to-binary conversion subroutine at their disposal, in the Monitor. This subroutine, PACK (entry address $EA84), converts an ASCII character in the accumulator to a 4-bit binary value in the four LSBs, and resets the four MSBs to zero. If PACK is called a second time, this subroutine will move the first (most-significant) digit to the four MSBs, and will place the new (least-significant) digit in the four LSBs. The PACK subroutine also checks each ASCII character to determine whether or not it represents a hexadecimal digit, and sets the Carry flag if a nonhexadecimal character is entered.

Example 6-4: The AIM 65 Version of the AH2B Subroutine

```
;THIS SUBROUTINE CONVERTS A TWO-DIGIT STRING OF HEXADECIMAL
;ASCII CHARACTER TO AN 8-BIT BINARY VALUE IN THE ACCUMULATOR.

AH2B  JSR  DIGIT   ;CONVERT DIGIT TO BINARY
DIGIT JSR  REDOUT  ;INPUT KEYBOARD CHARACTER
      JSR  PACK    ;CONVERT IT TO BINARY
      BCS  DIGIT   ;LOOP IF DIGIT IS NOT HEX
      RTS
```

Example 6-4 shows how the PACK subroutine can be used, with the keyboard input subroutine REDOUT of the AIM 65, to form a 5-instruction equivalent of the 22-instruction AH2B subroutine shown in Example 6-3. The subroutine in Example 6-4 executes the final four instructions twice, once for each input character. The instruction JSR DIGIT produces the first execution sequence. The sequence is executed a second time because the JSR DIGIT instruction caused the address at DIGIT to be placed on the stack, so the RTS returns to the JSR REDOUT instruction. Of course, upon completion of the second pass, RTS returns control to the calling program.

AN 8-BIT BINARY-TO-ASCII-BASED HEXADECIMAL CONVERSION

Subroutine B2AH in Example 6-5 works the opposite of subroutine AH2B (Example 6-3) by converting the 8-bit binary contents of the accumulator into two ASCII hexadecimal characters and then outputting them, one by one, to the printer. Because the printer

prints left to right, the digit in the four most-significant bits must be output first. Therefore, the subroutine begins by saving the other digit on the stack. A four-bit right-shift (four LSR A instructions) places the first digit in the four least-significant bit positions.

The actual conversion from binary to ASCII is performed by the CB2AH subroutine. If the digit is between 0 and 9, the CB2AH subroutine uses an ORA instruction to insert 3 as the most-significant digit. (See Table 6-1 for the binary/ASCII relationships.) If the digit is between A and F, the CB2AH subroutine adds $37 (ADC #$36 with Carry set) to the contents of the accumulator.

Upon return from CB2AH, the B2AH subroutine outputs the ASCII character for the most-significant digit to the printer by calling the PRTOUT subroutine (Example 6-2). At this point, the second digit is pulled off the stack and the instruction AND #$0F masks off its most-significant four bits. This 4-bit value is converted to ASCII (with JSR CB2AH), then output to the printer (with JMP PTROUT). The use of the JMP instruction to call PTROUT the second time is not a typographical error; it permits the elimination of an RTS instruction from the B2AH subroutine, so that the RTS instruction in the PTROUT subroutine will cause a return to the calling program rather than to the conversion subroutine.

The AIM 65 Monitor has a subroutine that performs essentially the same function as the B2AH subroutine; it is called NUMA, and has an entry address of $EA46. This completes the discussion of

Example 6-5: An 8-Bit Binary-to-ASCII-Based Hexadecimal Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS AN 8-BIT BINARY VALUE IN THE ACCUMULATOR
;TO A TWO-DIGIT ASCII STRING THAT IS OUTPUT TO THE PRINTER.

B2AH    PHA             ;SAVE BINARY VALUE ON STACK
        LSR   A         ;SHIFT FIRST DIGIT INTO FOUR LSB'S
        LSR   A
        LSR   A
        LSR   A
        JSR   CB2AH     ;CONVERT IT TO ASCII
        JSR   PTROUT    ;  AND OUTPUT IT TO PRINTER
        PLA             ;PULL SECOND DIGIT FROM STACK
        AND   #$0F      ;MASK OFF THE FOUR MSB'S
        JSR   CB2AH     ;CONVERT IT TO ASCII
        JMP   PTROUT    ;  AND OUTPUT IT TO PRINTER

;THE BINARY-TO-ASCII SUBROUTINE FOLLOWS

CB2AH   CMP   #$0A      ;IS DIGIT BETWEEN 0 AND 9?
        BCC   Z29
        ADC   #$36      ;NO. DIGIT IS BETWEEN A AND F.
        RTS
Z29     ORA   #$30      ;YES. ADD MSD = $3
        RTS
```

hexadecimal number-base conversions. Now, we will describe the conversions that are required if *decimal values* are being used.

THREE-DIGIT ASCII-BASED DECIMAL-TO-BINARY CONVERSION

In some respects, it is easier to convert ASCII-based decimal numbers to binary than to convert ASCII-based hexadecimal numbers to binary. However, as we shall see, the conversion process is also more complex. The ASCII/binary correlations are summarized in Table 6-2. As you can see from this table, the only ASCII values that are of interest to us in this section are those that fall between $30 and $39; all values below $30 and above $39 will be ignored. Before proceeding further with this discussion, we must call your attention to the fact that the binary equivalent of a decimal digit is nothing more than the four least-significant bits of the ASCII character.

As you already know, decimal numbers can be expressed as a series of integers multiplied by powers of 10. For example,

$$237 = (2 \times 10^2) + (3 \times 10^1) + (7 \times 10^0)$$

or

$$237 = (2 \times 100) + (3 \times 10) + (7 \times 1)$$

Since only one digit of a number can be entered into the 6502 microprocessor at one time, there will have to be a multiply-by-10 routine or subroutine in a general-purpose ASCII-based decimal-to-binary conversion subroutine. In this type of conversion routine, when the number 93 is entered, the 9 must be multiplied by 10 before being added to the 3.

Example 6-6 shows a subroutine (A3D2B) that converts three ASCII-based decimal digits from the keyboard to an 8-bit binary value in the accumulator. Since the accumulator is eight bits wide, only numbers in the range of 0 to 255 can be properly converted.

Table 6-2. The ASCII-Based Decimal Characters

ASCII Value	Binary Value
30	0000
31	0001
32	0010
33	0011
34	0100
35	0101
36	0110
37	0111
38	1000
39	1001

Example 6-6: A Three-Digit ASCII-Based Decimal-to-Binary Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS A THREE-DIGIT STRING OF DECIMAL ASCII
;CHARACTERS TO AN 8-BIT BINARY VALUE IN THE ACCUMULATOR.
;CARRY IS SET IF THE ACCUMULATOR CANNOT HOLD THE RESULT.

A3D2B   JSR   NEWDIG   ;FETCH HIGH-ORDER DIGIT
        STA   $30      ;  AND STORE IT AS PARTIAL RESULT
        JSR   NEWDIG   ;FETCH SECOND DIGIT
        JSR   MULT10   ;  AND ADD IT TO PARTIAL RESULT
        JSR   NEWDIG   ;FETCH LOW-ORDER DIGIT
        JMP   MULT10   ;  AND FORM FINAL RESULT

;THIS SUBROUTINE INPUTS THE NEXT VALID DECIMAL DIGIT, AND RETURNS
;WITH THIS DIGIT IN THE FOUR LSB'S OF THE ACCUMULATOR.

NEWDIG  JSR   KEYIN    ;FETCH DIGIT
        CMP   #$30     ;CHARACTER LESS THAN $30?
        BCC   NEWDIG
        CMP   #$3A     ;NO. IS IT GREATER THAN $39?
        BCS   NEWDIG
        AND   #$0F     ;NO. MASK OUT THE FOUR MSB'S
        RTS

;THIS SUBROUTINE MULTIPLIES THE PARTIAL RESULT BY 10, THEN ADDS
;THE NEW DIGIT TO THE LEAST-SIGNIFICANT DIGIT POSITION.

MULT10  STA   $31      ;SAVE DIGIT
        LDA   $30      ;FETCH PARTIAL RESULT
        ASL   A        ;MULTIPLY IT BY TWO
        ASL   A        ;MULTIPLY IT BY TWO AGAIN (TOTAL = X4)
        ADC   $30      ;ADD ORIGINAL RESULT TO IT (TOTAL = X5)
        ASL   A        ;MULTIPLY BY TWO (TOTAL = X10)
        ADC   $31      ;ADD NEW DIGIT
        STA   $30      ;  AND UPDATE PARTIAL RESULT
        RTS
```

If a number between 256 and 999 is entered, it will not be converted to its proper binary equivalent, and the Carry will be set to indicate the error condition.

The first instruction in the A3D2B subroutine calls another subroutine, NEWDIG. The NEWDIG subroutine is essentially a keyboard input subroutine, but it accepts only valid decimal digits. This subroutine begins by calling KEYIN, the familiar keyboard subroutine from Example 6-1. After the digit is input by KEYIN, the NEWDIG subroutine checks whether it falls outside of the decimal digit range of ASCII characters ($30 to $39). If the character is not a decimal digit, program control branches back to NEWDIG to await a new keyboard entry. If the character is a decimal digit, the instruction AND #$0F masks out its four most-significant bits.

Upon return from NEWDIG, the binary value of the digit is stored in location $30, as a first partial result. The second digit is

then input by another call to NEWDIG. Upon this return from NEWDIG, another subroutine, MULT10, is called. The MULT10 subroutine multiplies the partial result by 10, and then enters the 4-bit binary value of the new digit. The multiply-by-10 operation takes advantage of the fact that left-shifting a number doubles the value of the number. Therefore, MULT10 left-shifts the partial result twice (i.e., multiplies it by 4), adds the original value (multiply by 5) and left-shifts that result (multiply by 10).

Upon return from MULT10, the A3D2B subroutine makes a third call to NEWDIG to input the low-order digit, and then *jumps* to MULT10 to update the result. The use of the JMP instruction here, rather than a JSR instruction, eliminates the need for an RTS instruction in the A3D2B subroutine, by allowing the RTS instruction of the MULT10 subroutine to cause a return to the *calling program.*

FIVE-DIGIT ASCII-BASED DECIMAL-TO-BINARY CONVERSION

As mentioned previously, one of the difficulties with the three-digit ASCII-based decimal-to-binary conversion subroutine in Example 6-6 is the fact that numbers larger than 255 cannot be converted. What will the contents of the accumulator be after the number 256 is entered? The contents of the accumulator will be zero (and the Carry flag will be set). Remember, *an 8-bit binary number can only represent the decimal numbers 0 through 255.* Therefore, 256 would be too large to be entirely contained in the accumulator.

There are many, many cases in which we need to work with numbers larger than decimal 255. This means that an ASCII-based double- or triple-precision conversion subroutine that will provide 16- or 24-bit results (0–65,535 or $0–1.67 \times 10^7$) is very desirable. Let us look at a double-precision conversion that produces a 16-bit result for decimal numbers up to 65,535. You will recall that we observed at the beginning of this section that decimal numbers can be expressed as a series of integers multiplied by powers of 10. For example,

$$64237 = (6 \times 10^4) + (4 \times 10^3) + (2 \times 10^2) + (3 \times 10^1) + (7 \times 10^0)$$

Clearly, a 16-bit version of the MULT10 subroutine given in Example 6-6 could be written, and it could be called with four consecutive JSR instructions for the first (most-significant) digit, three consecutive JSR instructions for the next digit, and so on, but this would take a large amount of program space in memory. An obvious alternative is to write a loop that calls the multiply-

by-10 subroutine once for each digit entered. Fig. 6-1 is the flowchart for a five-digit ASCII-based decimal-to-binary conversion subroutine that uses the loop approach.

The flowchart in Fig. 6-1 begins by initializing the most-significant byte (MSBY) of the result to zero. It then inputs the high-order digit from the keyboard and converts it to a 4-bit binary value. (These steps were performed earlier using the NEWDIG subroutine in Example 6-6.) The 4-bit binary value is stored in

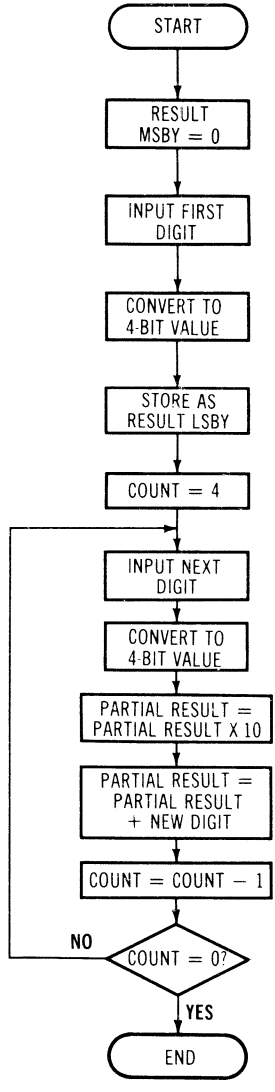


Fig. 6-1. Flowchart for a five-digit ASCII-based decimal-to-binary conversion subroutine.

memory as the initial value of the least-significant byte (LSBY) of the partial result. At this point, four more digits must be input, converted, and entered into the partial result, so a digit count is initialized with the value of 4. The remaining blocks in the flowchart represent a loop to process the four low-order digits that are entered from the keyboard. As a digit is input, it is converted to a 4-bit binary value, in the same manner the high-order digit was converted. In order to prepare the partial result to accept this new digit value, the two-byte partial result is multiplied by 10. The new digit value can now be added to the partial result, to update it. The final two flowchart operations simply involve decrementing the count and looping back for a new keyboard entry if all five digits have not been entered.

The subroutine in Example 6-7 is the program version of the flowchart given in Fig. 6-1. This subroutine (A5D2B) uses locations $32 and $33 to accumulate the partial result as keyboard digits are entered and processed. The subroutine begins by clearing the MSBY of the partial result (location $33), and then calling the keyboard input subroutine NEWDIG to accept and convert the first (high-order) digit from the keyboard. The NEWDIG subroutine in Example 6-7 is the same subroutine that was contained in Example 6-6; it is reproduced here for your convenience. Upon return from NEWDIG, the 6502 microprocessor stores the 4-bit binary code in the LSBY of the partial result (location $32). The next instruction, LDX #04, establishes a remaining-digit count in the X register.

The remaining instructions in the A5D2B subroutine, the instructions that start at NXTDIG, constitute a simple counting loop to process the remaining four ASCII-based decimal digits from the keyboard. This loop includes a call to a subroutine that has not yet been described, MPY10, which performs the same kind of multiply-by-10 function as was provided by the MULT10 subroutine in Example 6-6. However, MPY10 operates on a double-precision result. Admittedly, MPY10 is quite a bit longer than MULT10, but the sequence of operations is identical. As with MULT10, the MPY10 subroutine left-shifts the partial result twice (i.e., multiplies it by 4), adds the original value (multiply by 5) and left-shifts that result (multiply by 10). Because the partial result is a two-byte value, each left-shift requires two instructions, ASL $32 and ROL $33.

AN 8-BIT BINARY-TO-ASCII-BASED DECIMAL CONVERSION

Although computers process numbers in their binary (or BCD) form, it is often desirable to print results in their decimal represen-

Example 6-7: A Five-Digit ASCII-Based Decimal-to-Binary Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS A FIVE-DIGIT SERIES OF ASCII DECIMAL
;CHARACTERS FROM THE KEYBOARD TO A 16-BIT BINARY VALUE IN LOCATIONS
;$32 (LSBY) AND $33 (MSBY). CARRY IS SET IF THE RESULT CANNOT
;BE CONTAINED IN THESE TWO LOCATIONS.
;LOCATION $31 IS ALSO USED BY THE SUBROUTINE, FOR TEMPORARY STORAGE.

A5D2B   LDA   #00      ;PARTIAL RESULT MSBY = 0
        STA   $33
        JSR   NEWDIG   ;FETCH HIGH-ORDER DIGIT
        STA   $32      ;STORE IT AS PARTIAL RESULT LSBY
        LDX   #04      ;REMAINING DIGITS = 4
NXTDIG  JSR   NEWDIG   ;FETCH NEXT DIGIT
        JSR   MPY10    ;  AND ADD IT TO PARTIAL RESULT
        DEX
        BNE   NXTDIG   ;LOOP UNTIL 5 DIGITS CONVERTED
        RTS

;THIS SUBROUTINE INPUTS THE NEXT VALID DECIMAL DIGIT, AND RETURNS
;WITH THIS DIGIT IN THE FOUR LSB'S OF THE ACCUMULATOR.

NEWDIG  JSR   KEYIN    ;FETCH DIGIT
        CMP   #$30     ;CHARACTER LESS THAN $30?
        BCC   NEWDIG
        CMP   #$3A     ;NO. IS IT GREATER THAN $39?
        BCS   NEWDIG
        AND   #$0F     ;NO. MASK OUT THE FOUR MSB'S
        RTS

;THIS SUBROUTINE MULTIPLIES THE PARTIAL RESULT IN $32 AND $33
;BY TEN, THEN ADDS THE NEW DIGIT TO IT.

MPY10   STA   $31      ;SAVE DIGIT JUST ENTERED IN $31
        LDA   $33      ;SAVE PARTIAL RESULT ON STACK
        PHA
        LDA   $32
        PHA
        ASL   $32      ;MULTIPLY PARTIAL RESULT BY TWO
        ROL   $33
        ASL   $32      ;MULTIPLY IT BY TWO AGAIN (TOTAL = X4)
        ROL   $33
        PLA            ;ADD ORIGINAL RESULT TO IT (TOTAL = X5)
        ADC   $32
        STA   $32
        PLA
        ADC   $33
        STA   $33
        ASL   $32      ;MULTIPLY RESULT BY TWO (TOTAL = X10)
        ROL   $33
        LDA   $31      ;ADD DIGIT JUST ENTERED
        ADC   $32
        STA   $32
        LDA   #00
        ADC   $33
        STA   $33
        RTS
```

tations, rather than in binary or hexadecimal, to aid in understanding by decimally oriented human beings. Let us discuss how to convert binary numbers to their ASCII-based decimal equivalents.

Earlier we observed that decimal numbers can be expressed as a series of integers multiplied by powers of 10. For example,

$$237 = (2 \times 10^2) + (3 \times 10^1) + (7 \times 10^0)$$

With this in mind, the primary job of converting an 8-bit binary number to a decimal number reduces to finding out how many 100s, 10s and 1s are contained in the number. We can make this determination by performing a series of successive subtractions on the binary value. The number of times that 100 can be subtracted from the binary value tells us the 100s digit of the decimal representation. Similarly, the number of times that 10 from that remainder tells us the 10s digit of the decimal representation. And the remainder of that operation tells us the 1s digit of the decimal representation.

For example, to determine the number of 100s in the decimal number 237, we will subtract 100 from 237 until the result of the subtraction is negative. A negative result signifies we have gone "too far," so 100 must be added back in before continuing. Each time 100 can successfully be subtracted (no negative result), the *100s count* is incremented by 1. For the number 237, the sequence is:

```
 237
-100
 137     100s count = 1
-100
  37     100s count = 2
-100
- 63
+100
  37
```

Now that the number of 100s is known, we can calculate a *10s count* with a similar subtraction sequence. Working with our previous remainder, 37, the sequence is:

```
 37                          17
-10                         -10
  27    10s count = 1          7    10s count = 3
-10                         -10
  17    10s count = 2        - 3
                            +10
                              7
```

Example 6-8: An 8-Bit Binary-to-ASCII-Based Decimal Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS AN 8-BIT BINARY VALUE IN THE ACCUMULATOR
;TO A THREE-DIGIT ASCII DECIMAL STRING THAT IS OUTPUT TO THE
;PRINTER.

B2AD    LDX   #00     ;INITIALIZE HUNDREDS COUNTER
C100    CMP   #100    ;BINARY VALUE = OR GREATER THAN 100?
        BCC   OUT1    ;NO. GO PRINT DECIMAL DIGIT
        SBC   #100    YES. SUBTRACT 100
        INX           ;INCREMENT DECIMAL COUNT
        JMP   C100    ;  AND COMPARE AGAIN
OUT1    JSR   PUTOUT  ;GO PRINT DECIMAL DIGIT
        LDX   #00     ;INITIALIZE TENS COUNTER
C10     CMP   #10     ;BINARY VALUE = OR GREATER THAN 10?
        BCC   OUT2    ;NO. GO PRINT DECIMAL DIGIT
        SBC   #10     ;YES. SUBTRACT 10
        INX           ;INCREMENT DECIMAL COUNT
        JMP   C10     ;  AND COMPARE AGAIN
OUT2    JSR   PUTOUT  ;GO PRINT DECIMAL DIGIT
        CLC           ;CONVERT REMAINDER TO ASCII
        ADC   #$30
        JMP   PTROUT  ;  AND PRINT IT
;
PUTOUT  PHA           ;SAVE REMAINDER ON STACK
        TXA           ;MOVE DECIMAL COUNT TO ACCUMULATOR,
        ADC   #$30    ;  CONVERT IT TO ASCII,
        JSR   PTROUT  ;  AND PRINT IT
        PLA           ;RETRIEVE REMAINDER
        RTS
```

In order to determine the number of units, a 1 could be subtracted from the remainder of the 10s subtraction. However, the remainder already represents the number of units in the number being converted. Example 6-8 lists an 8-bit, binary-to-decimal conversion subroutine (B2AD) that uses the successive subtraction technique that we have just described.

The B2AD subroutine is very straightforward. It uses the successive subtraction technique to calculate the 100s count value, convert that count to ASCII code, and then output the ASCII character to the printer. It performs the same compare-and-subtract operation for the 10s count and, finally, it converts the remainder to ASCII code and outputs it to the printer. The B2AD subroutine operates in the following manner. The subroutine starts by initializing the 100s count to zero in the X register. The next instruction, CMP #100, compares the binary contents of the accumulator to decimal 100. If the accumulator contains a value that is equal to or greater than 100 (hexadecimal 64), the Carry flag will be set and the instruction BCC OUT1 will allow execution to sequence to the subtraction instruction SBC #100. Following the subtraction, the 100s count in the X register is incremented by one and the instruction

JMP C100 transfers execution back to the compare instruction. When the accumulator finally contains a value of less than 100, the Carry flag will be clear, and BCC OUT1 will branch to the JSR PUTOUT instruction at label OUT1. The PUTOUT subroutine calls the printer output subroutine PTROUT (Example 6-2), but before doing so, saves the remainder on the stack, and then transfers the X register contents to the accumulator and converts it to an ASCII decimal character (by adding $30). Before returning, the PUTOUT subroutine pulls the remainder off the stack.

The 10s count is calculated with the same instruction types as the 100s count, execpt 10 (rather than 100) is used as the operand in the compare-and-subtract instructions. Once the 10s count has been calculated and output to the printer, no further subtractions are required. Therefore, the remainder is converted to an ASCII character by adding $30 to it. With the 1s count in the accumulator in ASCII form, a final jump to PTROUT completes the B2AD subroutine. Why a JMP, rather than a JSR, to PTROUT? A JMP is used rather than a JSR so that the RTS instruction in the PTROUT subroutine will return to the *calling program*, rather than to the B2AD subroutine. This same "trick" was used earlier in Examples 6-5 and 6-6.

A 16-BIT BINARY-TO-ASCII-BASED DECIMAL CONVERSION

In Example 6-8, it was demonstrated that it is fairly easy to convert an 8-bit binary value to an ASCII-based decimal by calculating the 100s count, then the 10s count, and then the 1s count using individual sequences of instructions. However, if your program must convert binary numbers of double- or triple-precision (or more), this individualized approach would require many more instruction sequences than the task warrants. For these higher-precision numbers, we should be thinking in terms of one loop that can be executed for each *decimal weight.*

What types of instructions can be used to form this loop? Can we use a compare instruction, as we did in Example 6-7? No, a compare instruction can only operate on 8-bit numbers. To calculate a digit count using a higher-precision number requires a series of trial subtractions. What is being subtracted? The number to be subtracted must be the power of 10 that corresponds to that particular decimal weight. For example, double-precision (16-bit) binary values can contain numbers from 0 to decimal 65,535. To find the decimal equivalent of this number, you must see how many times 10,000 can be subtracted from it (that will give the 10,000s count), and then see how many times 1000 can be subtracted from the remainder (the 1000s count). From the remainder

of that operation, you must subtract 100 one or more times, and then 10 one or more times. The remainder that results after all of these subtractions have been made represents the 1s count, and you are through.

Example 6-9 lists an actual 16-bit binary-to-decimal conversion subroutine. This subroutine, DPB2AD (Double-Precision Binary to ASCII-Based Decimal), uses a look-up table, called SUBTBL, to hold the subtraction values for the four most-significant counts. The binary value to be converted is contained in two zero page

Example 6-9: A 16-Bit Binary-to-ASCII-Based Decimal Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS A 16-BIT BINARY VALUE IN MEMORY LOCATIONS
;$31 (LSBY) AND $32 (MSBY) TO A FIVE-DIGIT ASCII DECIMAL STRING
;THAT IS OUTPUT TO THE PRINTER.

DPB2AD  LDY   #00      INITIALIZE TABLE POINTER TO ZERO
NXTDIG  LDX   #00      ;INITIALIZE DIGIT COUNT TO ZERO
SUBEM   LDA   $31      ;FETCH LSBY OF BINARY VALUE
        SEC            ;SUBTRACT LSBY OF TABLE VALUE
        SBC   SUBTBL,Y
        STA   $31      ;  AND RETURN RESULT TO MEMORY
        LDA   $32      ;FETCH MSBY OF BINARY VALUE
        INY            ;SUBTRACT MSBY OF TABLE VALUE
        SBC   SUBTBL,Y
        BCC   ADBACK   ;IF RESULT IS NEGATIVE, RESTORE LSBY
        STA   $32      ;OTHERWISE, STORE MSBY OF RESULT,
        INX            ;  INCREMENT DIGIT COUNT,
        DEY            ;  POINT TO LSBY IN TABLE,
        JMP   SUBEM    ;  AND GO SUBTRACT AGAIN


;THE INSTRUCTIONS BELOW RESTORE THE LSBY VALUE IF THE SUBTRACTION
;PRODUCES A NEGATIVE RESULT, THEN OUTPUT THE DIGIT COUNT

ADBACK  DEY            ;POINT TO LSBY IN TABLE
        LDA   $31      ;FETCH LSBY OF BINARY VALUE
        ADC   SUBTBL,Y ;AND ADD LSBY OF TABLE VALUE
        STA   $31
        TXA            ;PUT DIGIT COUNT IN ACCUMULATOR
        ORA   #$30     ;CONVERT IT TO ASCII
        JSR   PTROUT   ;  AND PRINT IT
        INY            ;POINT TO NEXT TABLE VALUE
        INY
        CPY   #08      ;END OF TABLE?
        BCC   NXTDIG   ;NO. CONTINUE WITH NEXT DECIMAL WEIGHT
        LDA   $31      ;YES. PUT REMAINDER IN ACCUMULATOR
        ORA   #$30     ;CONVERT IT TO ASCII
        JMP   PTROUT   ;  AND PRINT IT
SUBTBL  .WOR  $2710    ;10,000
        .WOR  $03E8    ;1,000
        .WOR  $0064    ;100
        .WOR  $000A    ;10
```

memory locations, $31 and $32. The DPB2AD subroutine operates in the following manner. It begins by initializing a look-up table pointer in the Y register, and a digit count in the X register, to zero. In Example 6-9, the instruction LDA $31 at SUBEM represents the first instruction in a loop that extends down to ADBACK.

This loop subtracts the appropriate 16-bit value in SUBTBL from the binary value in locations $31 (LSBY) and $32 (MSBY). If the result of the subtraction is negative (i.e., if Carry is clear), the instruction BCC ADBACK causes the 6502 microprocessor to branch to ADBACK, in order to add the LSBY from the table back into location $31. There is no need to restore location $32 (the MSBY), since the MSBY result of the subtraction is not stored unless the branch test indicates a nonnegative (positive or zero) result. On a nonnegative result, the MSBY of the subtraction result *is* returned to memory (STA $32), and the 6502 microprocessor increments the decimal digit count (INX), decrements the table pointer (DEY), and then jumps to SUBEM to subtract the table value again. In summary, this loop repeatedly subtracts the table value (10,000 for the first count, for instance) from the binary value in $31 and $32 until the result is negative, at which point it executes a branch to ADBACK.

The instructions at ADBACK start by restoring the value of the binary value's LSBY (location $31), and then they execute a three-instruction sequence that outputs the digit count to the printer, using the PTROUT subroutine from Example 6-2. The output operation completes the processing of that particular decimal weight, so the 6502 microprocessor increments the Y register twice in order to point to the next look-up table value. If there are any more table values remaining, compare instruction CPY #08 causes the Carry to be cleared and, then, BCC NXTDIG branches the control back to NXTDIG, the digit count initialization point. If all table values have been processed, location $31 holds the 1s count. This count is loaded into the accumulator, converted to ASCII code, and output to the printer. As in Examples 6-5 and 6-8, the final call to the printer subroutine PTROUT is performed with a JMP instruction so that the RTS instruction of PTROUT will return to the calling program, rather than to the conversion subroutine.

As you already know, the 6502 microprocessor has the capability of performing addition and subtraction operations on binary-coded decimal (BCD) data directly, without converting these data to their binary forms. The next two subroutines that will be discussed convert ASCII-based decimal characters to BCD digits, and vice versa.

TWO-DIGIT ASCII-BASED DECIMAL-TO-BCD CONVERSION

The first 10 decimal digits, 0 to 9, are actually a subset of the 16 hexadecimal digits (0 to 9 and A to F), so the subroutine that converts ASCII-based decimal entries from the keyboard into binary-coded decimal (BCD) entries should be similar to the ASCII-based hex-to-binary conversion subroutine presented in Example 6-3. Indeed, these conversion subroutines *are* similar, as we shall see in this section. If you do not recall the relationships between the ASCII-based decimal characters and the 4-bit BCD values, they are sumarized in Table 6-2.

Example 6-10 lists an ASCII-based decimal-to-BCD conversion subroutine (AD2D) that is similar to, but simpler than, the ASCII-based hexadecimal-to-binary conversion subroutine of Example 6-3. The AD2D subroutine begins by calling a subroutine labeled NEWDIG, which is the same subroutine that was used to input keyboard characters in Example 6-7. After receiving the keyboard character, the subroutine left-shifts it four times (ASL A) in order to place it in the four MSBs of the accumulator, and then stores the result temporarily in location $30. The second (least-significant) digit is input via the same subroutine (NEWDIG), and is then converted to a 4-bit code and merged with the most-significant digit from memory location $30.

Example 6-10: An ASCII-Based Decimal-to-BCD Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS A TWO-DIGIT STRING OF DECIMAL ASCII
;CHARACTERS TO TWO BINARY-CODED DECIMAL DIGITS IN THE ACCUMULATOR.
;THE SUBROUTINE USES LOCATION $30 FOR TEMPORARY STORAGE.

AD2D    JSR   NEWDIG   ;INPUT MOST-SIGNIFICANT DIGIT
        ASL   A        ;SHIFT IT INTO THE FOUR MSB'S
        ASL   A
        ASL   A
        ASL   A
        STA   $30      ;SAVE THIS DIGIT IN LOCATION $30
        JSR   NEWDIG   ;INPUT LEAST-SIGNIFICANT DIGIT
        ORA   $30      ;  AND ADD MOST-SIGNIFICANT DIGIT
        RTS

;
;THE FOLLOWING SUBROUTINE INPUTS THE NEXT VALID DECIMAL DIGIT, AND
;RETURNS WITH THIS DIGIT IN THE FOUR LSB'S OF THE ACCUMULATOR

NEWDIG  JSR   KEYIN    ;FETCH DIGIT
        CMP   #$30     ;CHARACTER LESS THAN $30?
        BCC   NEWDIG
        CMP   #$3A     ;NO. IS IT GREATER THAN $39?
        BCC   NEWDIG
        AND   #$0F     ;NO. MASK OFF THE FOUR MSB'S
        RTS
```

TWO-DIGIT BCD-TO-ASCII-BASED DECIMAL CONVERSION

In the preceding section, it was demonstrated that converting ASCII-based decimal characters to BCD is similar to, but easier than, converting ASCII-based hexadecimal characters to binary. In this section, we will demonstrate that converting BCD digits to ASCII-based decimal characters is similar to, but easier than, converting binary data to ASCII-based hexadecimal characters. The relationships between the 4-bit BCD codes and the ASCII-based hexadecimal characters are summarized in Table 6-2.

If your application requires converting binary values to ASCII-based hexadecimal characters, you can also use the binary-to-hex conversion subroutine B2AH given in Example 6-3 to perform your BCD-to-decimal conversions as well. Otherwise, you should use the BCD-to-ASCII-based decimal conversion subroutine that is listed in Example 6-11. This subroutine (D2AD) begins by storing the contents of the accumulator in location $30, which saves the least-significant BCD digit while the most-significant BCD digit is being processed. The STA $30 instruction is followed by four consecutive right-shift instructions (LSR A) that move the most-significant BCD digit to the four least-significant bit positions. The most-significant BCD digit is then converted into ASCII code by ORing $30 into the four most-significant bit positions. A call to the printer subroutine PTROUT outputs this ASCII character to the printer.

Once the most-significant digit is output, the least-significant digit is retrieved from location $30 and stripped of its four most-significant bits (AND #$0F). The digit is converted to ASCII code by using ORA #$30, and is then output to the printer by a second call to the PTROUT subroutine. This second call is performed with a JMP instruction so that the RTS instruction of the

Example 6-11: A BCD-to-ASCII-Based Decimal Conversion Subroutine

```
;THIS SUBROUTINE CONVERTS TWO BINARY-CODED DECIMAL DIGITS IN THE
;ACCUMULATOR TO A TWO-DIGIT ASCII STRING THAT IS OUPUT TO THE
;PRINTER. LOCATION $30 IS USED FOR TEMPORARY STORAGE.

D2AD    STA   $30      ;SAVE ACCUMULATOR IN LOCATION $30
        LSR   A        ;SHIFT MOST-SIGNIFICANT DIGIT INTO THE
        LSR   A        ;  FOUR LSB'S
        LSR   A
        LSR   A
        ORA   #$30     ;CONVERT THIS DIGIT TO ASCII
        JSR   PTROUT   ;  AND PRINT IT
        LDA   $30      ;GET LEAST-SIGNIFICANT DIGIT
        AND   #$0F     ;MASK OFF THE FOUR MSB'S
        ORA   #$30     ;CONVERT THIS DIGIT TO ASCII
        JMP   PTROUT   ;  AND PRINT IT
```

PTROUT subroutine returns control to the calling program, rather than to the D2AD subroutine. The AIM 65 Monitor has a subroutine NUMA (entry address $EA46) that performs the same function as the D2AD subroutine.

LEADING ZERO SUPPRESSION

Quite often, when numbers are printed on a printer, teletypewriter, or display, the leading zeroes are not printed. They are *suppressed.* This means that instead of printing the number 00302, the number 302 is printed. In the previous output conversion subroutines (Examples 6-5, 6-8, 6-9, and 6-11), a number such as 00302 or 01579 would be printed. Instructions can be easily added to these subroutines that will suppress the printing of the leading zeroes.

For leading-zero suppression, there has to be some type of a *flag* to indicate when the first nonzero character has been printed. After this flag is set by the nonzero character, any zeroes that are encountered in the lesser-digit positions are printed. If a number such as 00302 is to be printed, it is not sufficient to just suppress *all* ASCII zeroes since, then, the number would be printed as 32. Example 6-12 is a new version of the 16-bit binary-to-ASCII-based decimal conversion subroutine DPB2AD (Example 6-9), that suppresses the printing of leading zeroes by using a flag in location $40.

Example 6-12 has seven more instructions than Example 6-9. The first of these additional instructions, STY $40, clears the leading zeroes flag at the beginning of the subroutine; the other six instructions are added to the ADBACK routine. You will recall from the discussion of Example 6-9 that the ADBACK routine is executed when the subtraction of a SUBTBL value from the binary remainder produces a negative result. The function of ADBACK is to restore the subtracted table value to the remainder, print the decimal count for that particular digit, and then branch back to NXTDIG if additional digits must be calculated.

The ADBACK routine in Example 6-12 still performs the remainder-restoration and branch-to-NXTDIG operations, but performs the print operation only if the digit does not represent a leading zero. The first five instructions of the ADBACK routine in Example 6-12 are the same instructions used in the ADBACK routine in Example 6-9. In both cases, these instructions restore the LSBY of the binary remainder (location $31) to its value prior to the subtraction, and transfer the digit count from the X register to the accumulator. At this point, the BNE SETLZF instruction determines whether or not the digit count is zero. If it is nonzero, the 6502 microprocessor branches to SETLZF to set Bit 7 of the

Example 6-12: A 16-Bit Binary-to-ASCII-Based Decimal Conversion Subroutine, With Leading Zero Suppression

```
;THIS SUBROUTINE CONVERTS A 16-BIT BINARY VALUE IN MEMORY LOCATIONS
;$31 (LSBY) AND $32 (MSBY) TO A FIVE-DIGIT ASCII DECIMAL STRING
;THAT IS OUTPUT TO THE PRINTER. LEADING ZEROES ARE SUPPRESSED,
;USING A FLAG IN LOCATION $40.

DPB2AD  LDY   #00      ;INITIALIZE TABLE POINTER TO ZERO
        STY   $40      ;  AND CLEAR LEADING ZEROES FLAG
NXTDIG  LDX   #00      ;INITIALIZE DIGIT COUNT TO ZERO
SUBEM   LDA   $31      ;FETCH LSBY OF BINARY VALUE
        SEC            ;SUBTRACT LSBY OF TABLE VALUE
        SBC   SUBTBL,Y
        STA   $31      ;  AND RETURN RESULT TO MEMORY
        LDA   $32      ;FETCH MSBY OF BINARY RESULT
        INY            ;SUBTRACT MSBY OF TABLE VALUE
        SBC   SUBTBL,Y
        BCC   ADBACK   ;IF RESULT IS NEGATIVE, RESTORE LSBY
        STA   $32      ;OTHERWISE, STORE MSBY OF RESULT,
        INX            ;  INCREMENT DIGIT COUNT,
        DEY            ;  POINT TO LSBY IN TABLE,
        JMP   SUBEM    ;  AND GO SUBTRACT AGAIN
;
;THE INSTRUCTIONS BELOW RESTORE THE LSBY VALUE IF THE SUBTRACTION
;PRODUCES A NEGATIVE RESULT, THEN OUTPUT THE DIGIT COUNT

ADBACK  DEY            ;POINT TO LSBY IN TABLE
        LDA   $31      ;FETCH LSBY OF BINARY VALUE
        ADC   SUBTBL,Y ;  AND ADD LSBY OF TABLE VALUE
        STA   $31
        TXA            ;PUT DIGIT COUNT IN ACCUMULATOR
        BNE   SETLZF   ;IF IT IS NONZERO, GO PRINT IT
        BIT   $40      ;IS THIS ZERO A LEADING ZERO?
        BMI   CNVTA    ;NO. GO PRINT IT
        BPL   UPTBL    ;YES. BYPASS PRINT OPERATION
SETLZF  LDX   #$80     ;SET LEADING ZEROES FLAG
        STX   $40
CNVTA   ORA   #$30     ;CONVERT DIGIT TO ASCII
        JSR   PTROUT   ;  AND PRINT IT
UPTBL   INY            ;POINT TO NEXT TABLE VALUE
        INY
        CPY   #08      ;END OF TABLE?
        BCC   NXTDIG   ;NO. CONTINUE WITH NEXT DECIMAL WEIGHT
        LDA   $31      ;YES. PUT REMAINDER IN ACCUMULATOR
        ORA   #$30     ;CONVERT IT TO ASCII
        JMP   PTROUT   ;  AND PRINT IT
SUBTBL  .WOR  $2710    ;10,000
        .WOR  $03E8    ;1,000
        .WOR  $0064    ;100
        .WOR  $000A    ;10
```

leading zeroes flag (location $40), then converts the 4-bit code to ASCII code and prints it (JSR PTROUT). If the digit count is zero, the next two instructions, BIT $40 and BMI CNVTA, de-

Table 6-3. Conversion Subroutines and Their Characteristics

Name	Operation	Example	Registers Affected	Memory Locations Affected
AH2B	Converts two hex characters from keyboard to binary value in accumulator.	6-3	A	$30
B2AH	Converts 8-bit binary contents of accumulator to two ASCII hex characters, and outputs them to printer.	6-5	A	None
A3D2B	Converts three decimal digits from keyboard to binary value in accumulator. The Carry is set if a number greater than 255 is entered.	6-6	A	$30, $31
A5D2B	Converts five decimal digits from keyboard to 16-bit binary value in $32 and $33. The Carry is set if a number greater than 65,535 is entered.	6-7	A, X	$31, $32, $33
B2AD	Converts 8-bit binary contents of accumulator to three ASCII decimal characters, and outputs them to printer.	6-8	A, X	None
DPB2AD	Converts 16-bit binary contents of $31 and $32 to five ASCII decimal characters, and outputs them to printer.	6-9, 6-12	A, X, Y	$31, $32
AD2D	Converts two decimal digits from keyboard to two BCD digits in accumulator.	6-10	A	$30
D2AD	Converts two BCD digits in accumulator to two ASCII decimal characters, and outputs them to printer.	6-11	A	$30

termine whether or not that zero represents a *leading* zero. If Bit 7 of location $40 is set to logic 1, the 6502 microprocessor branches to CNVTA, where the digit count is converted to ASCII code and output to the printer. If Bit 7 of location $40 is reset to logic 0, the BPL UPTBL instruction causes the 6502 microprocessor to bypass the printing operation. One subtle point is worth mentioning here. If the binary value is $0000, printing is suppressed for all but the *final* zero. This zero is deliberately allowed to be printed in order to inform the user that he has converted a binary value that is equal to zero.

Table 6-4. AIM 65 Conversion Subroutines

Name	Operation	Entry Address	Registers Affected
HEX	Converts ASCII hex character in accumulator to a 4-bit binary value in the four LSBs of the accumulator. The four MSBs of the accumulator contain zeroes on return.	$EA7D	A
NOUT	Converts binary value in the four LSBs of the accumulator to an ASCII hex character, and outputs it to the active output device.	$EA51	A
NUMA	Converts 8-bit binary contents of the accumulator to two ASCII hex characters, and outputs them to active output device.	$EA46	A
PACK	Converts ASCII hex character in the accumulator to a 4-bit binary value in the four LSBs of the accumulator. The result of the last call to HEX or PACK is placed in the four MSBs of the accumulator.	$EA84	A

SUMMARY

The number-base conversion subroutines in this chapter are not intended to serve as a "cook-book" for every application, but they should provide a sufficient foundation from which other similar subroutines can be designed. Table 6-3 lists the number-base conversion subroutines that have been given in this chapter, and summarizes their characteristics. Table 6-4 includes similar information for some conversion subroutines that are included in the AIM 65 monitor.

7

Interrupts and Resets

Up to this point in the book, we have discussed various aspects of the 6502 *microprocessor*, without considering how its operation is influenced by external devices in a *microcomputer system*. Admittedly, the number-base conversion subroutines in Chapter 6 demonstrated input from, and output to, two idealized peripheral devices (a keyboard and a printer), but these devices were included only to establish a realistic context in which the subroutines would normally be found. Other than that brief excursion into the "real world" environment, we have purposely avoided discussing how the 6502 microprocessor is affected by the other elements of a microcomputer system. The remaining chapters in this book are devoted to discussing those external influences on the 6502 microprocessor, the programs that deal with them, and the fundamentals of *interfacing* the 6502 microprocessor to external devices in a microcomputer system. This chapter covers two distinct, but functionally similar, topics—interrupts and resets.

There are two types of *interrupts*, maskable (interrupts that can be temporarily ignored) and nonmaskable (interrupts that require immediate attention). The word "interrupt" is familiar to all readers. Webster defines it as "to break into or in upon." We have encountered both types of interrupts in our everyday life. Suppose, for instance, that you are driving your car and notice that the muffler sounds louder than usual. That is a maskable interrupt; it can be temporarily disregarded (perhaps by turning up the volume on your car radio), but the need for your attention still exists. However, if one of your tires blows out, that constitutes a nonmaskable interrupt; it requires your immediate attention. The 6502 microprocessor provides for both types of interrupts, maskable and nonmaskable. Both types will be discussed in this chapter.

A *reset* is the operation by which a microcomputer is *initialized* to some known state. All microprocessors are designed to be reset when the system power is turned on. Further, most, if not all, microprocessors are designed to allow the internal registers of the microprocessor to be initialized at other times by an external signal. We have just defined two separate conditions, *reset* at power-up time and *restart* at all other times. Both conditions will be covered in this chapter.

We have said that interrupts and resets are functionally similar, but we did not elaborate on that statement. Interrupts and resets are functionally similar because they use *vector pointers* to determine the memory address from which the next instruction will be fetched. When an interrupt or a reset is activated, the 6502 microprocessor loads the program counter with the contents of a particular pair of memory locations. These locations hold the address (the vector pointer) of the next instruction to be executed. The 6502 microprocessor uses locations $FFFA through $FFFF to hold vector pointers. Locations $FFFA and $FFFB hold the vector pointers for the nonmaskable interrupt, locations $FFFC and $FFFD hold the vector pointers for the reset, and locations $FFFE and $FFFF hold the vector pointers for the maskable interrupt request.

THE 6502 MICROPROCESSOR INTERRUPTS

Interrupts are externally generated signals which temporarily suspend the program that is being executed by the microprocessor, and cause program control to be transferred to a subroutine that is designed to *service* that particular interrupt. Peripheral devices use interrupts to "inform" the microprocessor that they have data to be input, or that they need data from the microprocessor. This technique eliminates the need for the microprocessor to waste valuable execution time *polling* the status of the peripheral devices of the system when none of the devices require servicing. In the last chapter, the 6502 microprocessor polled the status indicator of the ASCII keyboard. As we saw, the 6502 microprocessor stayed in the KEYIN loop (Example 6-1) until a key was pressed.

Interrupts are also used to signal some condition that requires the immediate attention of the microprocessor, such as a power failure. The 6502 integrated circuit has separate interrupt pins to handle each of these applications. Peripheral devices *request* interrupt service from the 6502 microprocessor by activating the $\overline{\text{IRQ}}$ (Interrupt Request) line. The critical situations (again, such as power failure) *force* immediate service from the 6502 microprocessor by activating the $\overline{\text{NMI}}$ (Non-Maskable Interrupt) line. (The

"bar" over the IRQ and NMI signal names indicates that these signals are active when they are pulled to ground.) Both types of interrupt signals will now be discussed, beginning with $\overline{\text{IRQ}}$.

INTERRUPT REQUEST ($\overline{\text{IRQ}}$)

In all of the previous examples in this book, the microprocessor was in control of fetching the next instruction to be executed. Most of the time, the next instruction was fetched from the memory locations that followed the current instruction. Instances were discussed, however, where the next instruction was fetched from some other place in memory. This occurred when a branch, Jump (JMP), Jump to Subroutine (JSR), or Return from Subroutine (RTS) instruction was executed. In all cases, though, the instruction fetching was based on the *logic* of the particular program being executed.

How can the 6502 microprocessor "know" whether peripheral devices in the system need servicing? One way is to design the main program so that every so often it stops processing data and *polls* every device in the system to see whether any devices requires servicing. In Example 6-1, the keyboard was constantly polled (once every few microseconds) to see if a key was pressed and, in Example 6-2, the printer was constantly polled to see if it was "ready." Obviously, this is very inefficient, and is somewhat analagous to picking up a telephone receiver every so often to see if anyone is calling you.[1] As an alternative, the 6502 microprocessor has an $\overline{\text{IRQ}}$ input that permits external devices to request servicing from the microprocessor. In our telephone analogy, the $\overline{\text{IRQ}}$ line functions as the bell and indicates to the 6502 microprocessor that some device in the system is "calling." Also, like a telephone bell, the $\overline{\text{IRQ}}$ can be ignored until the 6502 microprocessor is prepared to respond to it.

Interrupt Control Instructions

The IRQ Disable bit (I) in the processor status register determines whether or not the 6502 microprocessor will respond to an interrupt request on the $\overline{\text{IRQ}}$ line. The two instructions that control this bit are:

Instruction	Description
CLI	Clear Interrupt Disable Bit
SEI	Set Interrupt Disable Bit

The first instruction, CLI, clears the Interrupt Disable bit (I), which will cause an external interrupt request to be serviced as soon as it is sensed by the 6502 microprocessor. (The description

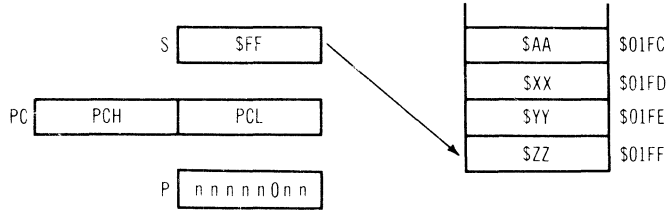
of *how* the 6502 microprocessor responds to an $\overline{\text{IRQ}}$ will be discussed in the next section.) The second instruction, SEI, sets the Interrupt Disable bit (I), which will cause the 6502 microprocessor to ignore all subsequent $\overline{\text{IRQ}}$ interrupt requests. The Interrupt Disable bit (I) is automatically set by the 6502 microprocessor when power is turned on.

How the 6502 Microprocessor Responds to an $\overline{\text{IRQ}}$

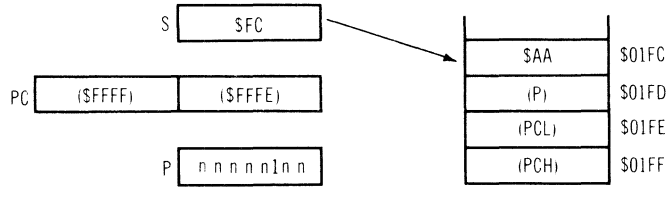
If the Interrupt Disable bit (I) is clear and some external device activates the $\overline{\text{IRQ}}$ signal (pulls it low), the 6502 microprocessor finishes executing the current instruction to complete execution, and then automatically initiates an eight-cycle interrupt sequence. During this sequence, the 6502 microprocessor pushes three bytes of "return" information onto the stack (the high and low bytes of the program counter and the contents of the processor status register). It then loads the contents of the dedicated IRQ vector low ($FFFE) and high ($FFFF) into the program counter. The 6502 microprocessor also sets the IRQ Disable bit (I) in the processor status register, to temporarily "lock out" subsequent interrupt requests.

Fig. 7-1 summarizes how the 6502 microprocessor responds to an $\overline{\text{IRQ}}$, by showing both "before" and "after" diagrams of the Stack Pointer (S), the program counter (PC), the processor status register (S), and the first four locations in the stack ($01FC through $01FF). In the diagram of Fig. 7-1A, the Stack Pointer is pointing to the next free stack location (assumed to be $01FF for this example), the program counter contains the address of the next instruction in the main-line program (high-address is PCH, low-address is PCL), and the processor status register is represented by a binary bit pattern in which the I bit is clear (zero). In the diagram of Fig. 7-1B, it is shown that following $\overline{\text{IRQ}}$, the program counter and processor status register are now on the stack, and the Stack Pointer is pointing to location $01FC. Further, the low-order and high-order bytes (PCL and PCH) of the program counter now contain the contents of memory locations $FFFE and $FFFF, respectively, and the I bit has been set to a 1 in the processor status register. The I bit is the only processor status register bit that was altered by this operation.

At this point, the program counter contains the starting address of a program designed to service $\overline{\text{IRQ}}$-generated interrupts. This program is called *an interrupt service routine* or, in some literature, *an interrupt handler.* The interrupt service routine must perform two functions; it must identify the device that generated the interrupt request (if there is more than one such device in the system) and it must perform the operation required by that device.



(A) Before $\overline{IRQ}$.



(B) After $\overline{IRQ}$.

Fig. 7-1. How the 6502 microprocessor responds to an $\overline{IRQ}$.

How does the interrupt service routine identify the interrupting device? It usually identifies this device by *polling* the status register of each device in the system, to find out which device has its interrupt request bit set. Once the interrupting device is identified, the interrupt service routine must fetch a new address—the starting address of the interrupt service routine for the interrupting device. The 6500-compatible devices maintain their interrupt request bits in either Bit 6 or 7 of a register, so that the polling sequence can be performed with a series of BIT and branch instructions.

Example 7-1 shows a polling sequence for a system that contains four devices. Devices 1, 2, and 4 can generate only one interrupt request, and will indicate this request in Bit 7 of their respective status registers (locations SDEV1, SDEV2, and SDEV4). Device 3 can generate two separate interrupt requests, and can indicate these requests in Bits 6 and 7 of its status register (location SDEV3). The interrupt status of Devices 1, 2, and 3 in the system is interrogated with a BIT instruction, which loads the state of Bit 6 and 7 into the processor status register's Overflow (V) and Negative (N) flag, respectively. For Devices 1 and 2, a BMI instruction determines whether or not the device has an active interrupt request. Device 3 requires two branch instructions (BMI and BVS), since this device is capable of generating either of two separate interrupt requests. Device 4 requires no interrogation, since *it* must have generated the interrupt request if Device 1, 2,

Example 7-1: Interrupt Polling Sequence

```
        .
        .
        .
        BIT   SDEV1   ;INTERRUPT REQUEST FROM DEVICE 1?
        BMI   JISR1   ;IF SO, BRANCH TO JISR1
        BIT   SDEV2   ;INTERRUPT REQUEST FROM DEVICE 2?
        BMI   JISR2   ;IF SO, BRANCH TO JISR2
        BIT   SDEV3   ;INTERRUPT REQUEST FROM DEVICE 3?
        BMI   JISR3A  ;IF SO, BRANCH TO JISR3A
        BVS   JISR3B  ;  OR TO JISR3B
        JMP   ISR4    ;GO SERVICE DEVICE 4 INTERRUPT
JISR1   JMP   ISR1    ;GO SERVICE DEVICE 1 INTERRUPT
JISR2   JMP   ISR2    ;GO SERVICE DEVICE 2 INTERRUPT
JISR3A  JMP   ISR3A   ;GO SERVICE DEVICE 3 INTERRUPT "A"
JISR3B  JMP   ISR3B   ;GO SERVICE DEVICE 3 INTERRUPT "B"
        .
        .
        .
```

or 3 did not. You will note that the polling sequence in Example 7-1 assigns priorities to the devices in the system; Device 1 has the highest priority, Device 4 has the lowest priority. Although it appears that the 6502 microprocessor needs a large amount of time to get to the Device 4 interrupt service routine, due to the number of instructions that are executed before a Device 4 interrupt request can be serviced, the polling takes only six cycles for Devices 1 and 2, and eight cycles for Device 3. Therefore, the 6502 microprocessor needs only 20 cycles (20 microseconds) before it can service Device 4.

Instructions in an Interrupt Service Routine

Besides the instructions that actually transfer information between the interrupting peripheral device and the 6502 microprocessor, what other instructions make up an interrupt service routine? Most interrupt service routines begin with instructions that save, on the stack, the current values of registers that will be altered by the service routine. Of course, the end of the subroutine must have complementary instructions that pull those register values off the stack. Further, most interrupt service routines also include a CLI (Clear Interrupt Disable Bit) instruction, to allow other higher-priority interrupt requests to be serviced. The *location* of CLI in the interrupt service routine will vary with the priority of the device being serviced. In the interrupt service routine for the lowest-priority device, CLI may follow the instructions that save register values on the stack. Conversely, the interrupt service routine for the highest-priority device may not even include a CLI instruction, and will allow interrupt requests to be enabled only

on return from the interrupt service routine. The instruction that performs the return from an interrupt service routine is *a special instruction called RTI.* The RTI instruction must be the final instruction to be executed in *every* interrupt service routine. Let us now look at that instruction in detail.

RETURN FROM INTERRUPT (RTI) INSTRUCTION

The final instruction to be executed in every interrupt service routine is:

Instruction	Description
RTI	Return from Interrupt

This RTI instruction causes the processor status register and the program counter to be reinitialized with their preinterrupt values from the stack. Execution of the RTI instruction will automatically enable IRQ interrupt requests, since the Interrupt Disable bit (I) of the processor status register was clear when this register was pushed onto the stack.

The RTI instruction is, you will note, quite similar to the RTS (Return from Subroutine) instruction that was first encountered in Chapter 3. Both instructions return from a subroutine to the program from which the subroutine was called. For RTI, the call was made automatically, by an externally generated interrupt request. For RTS, the call was made under software control by a JSR (Jump to Subroutine) instruction. Like RTS, the RTI instruction is an implied-address instruction that occupies one byte in memory and requires six cycles to be executed. From a purely operational standpoint, the only difference between RTS and RTI is that RTS pulls two bytes of information from the stack (the low byte and high byte of the program counter), whereas, RTI pulls three bytes of information from the stack (the processor status register, and the low byte and high byte of the program counter). In fact, if you had a subroutine that needed to save the presubroutine value of the processor status register, you could use a PHP (Push Processor Status on Stack) as the first instruction in the subroutine, and use an RTI, rather than an RTS, as the return instruction for the subroutine.

SUMMARY OF IRQ-GENERATED INTERRUPTS

In the preceding sections, it has been shown that the 6502 microprocessor has an interrupt request line ($\overline{IRQ}$) which, when low, indicates that one of the system devices connected to this line requires service. If the $\overline{IRQ}$ Disable bit (I) of the processor status

register is clear when $\overline{\text{IRQ}}$ is pulled low, the 6502 microprocessor will complete the currently executing instruction, push the contents of the program counter and the processor status register onto the stack, and then fetch the address of the next instruction from two dedicated memory locations, $FFFE and $FFFF. This address is the starting address of an *interrupt service routine.*

The interrupt service routine must *poll* all of the devices in the system, to find out which device issued the interrupt request. When the interrupting device has been identified, the 6502 microprocessor jumps to a device-dependent service routine. The device-dependent service routine must perform the proper data transfers to, or from, the peripheral device and clear the IRQ Disable bit so that higher-priority devices can interrupt, if required. The final instruction in the service routine is RTI, which returns control to the program that was being executed when the interrupt occurred.

NONMASKABLE INTERRUPT ($\overline{\text{NMI}}$)

The other interrupt line of the 6502 microprocessor, Nonmaskable Interrupt ($\overline{\text{NMI}}$), provides interrupts for high-priority devices and events (such as a power failure) that cannot afford to wait during the time that the interrupt requests are disabled. Unlike $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$-generated interrupts cannot be masked out or disabled; the 6502 microprocessor will begin processing an $\overline{\text{NMI}}$ interrupt upon completion of the currently executing instruction.

In the 6500 system, an $\overline{\text{NMI}}$ interrupt always has priority over an IRQ interrupt. If an interrupt request and a nonmaskable interrupt occur simultaneously, the 6502 microprocessor will process the nonmaskable interrupt. Other than these differences, how does the operation of $\overline{\text{NMI}}$ differ from that of $\overline{\text{IRQ}}$? They differ very little. The 6502 microprocessor performs an eight-cycle sequence for either type but, for the $\overline{\text{NMI}}$, the vector pointer is fetched from locations $FFFA (low-address byte) and $FFFB (high-address byte), rather than from locations $FFFE and $FFFF (as for the $\overline{\text{IRQ}}$). In fact, Fig. 7-1 also illustrates how the 6502 microprocessor responds to $\overline{\text{NMI}}$ as well, except that for the $\overline{\text{NMI}}$, the post-interrupt value of the program counter would be ($FFFB) and ($FFFA).

Since the $\overline{\text{NMI}}$ line is dedicated to servicing high-priority events, it is important to know just *how fast* the 6502 microprocessor can be expected to respond to the interrupt. Let us find out by taking a look at the "worst case." The worst case occurs if the $\overline{\text{NMI}}$ is activated just as the 6502 microprocessor has fetched the op code of one of its longest instructions—a seven-cycle instruction such as INC (Increment Memory) or DEC (Decrement Memory) with

absolute indexed addressing. The $\overline{\text{NMI}}$ interrupt will not be accepted until completion of this instruction six cycles later, at which time the 6502 microprocessor will initiate an eight-cycle interrupt sequence. In the eighth cycle of this sequence, the 6502 microprocessor will fetch the op-code byte of the first instruction in the interrupt service routine. This instruction can be either the input (LDA) or output (STA) instruction that transfers data from or to the peripheral device, in which case, it will be executed in no less than four cycles. Adding these execution cycle values, we see that *with an $\overline{\text{NMI}}$ interrupt, data can be transferred to or from a peripheral device in no more than 17 cycles* (17 microseconds for a 1-MHz 6502 microprocessor).

In the AIM 65, the $\overline{\text{NMI}}$ vector pointer ($FFFA and $FFFB) holds the address $E075. This address contains a jump indirect instruction, JMP (NMIV2). The parameter NMIV2 is automatically initialized at power-up with the entry address of the Monitor NMI interrupt service routine $E07B. However, NMIV2 is user-alterable, so AIM 65 owners can modify this parameter to address an NMI interrupt service routine of their own.

Before concluding this discussion of interrupts, let us look at an instruction that permits a programmer to simulate an interrupt under program control. This instruction, BRK, is used to halt the 6502 microprocessor so that the current contents of memory and registers can be examined. The BRK instruction is primarily used only during the debugging stage of program development.

THE BREAK (BRK) INSTRUCTION

The Break (BRK) instruction causes the 6502 microprocessor to execute an interrupt sequence under program control. When BRK is executed, the 6502 microprocessor sets the BRK Command flag (B) in the processor status register, increments the program counter by one, and then pushes three bytes of information onto the stack (the high and low bytes of the program counter and the contents of the processor status register). At this point, the 6502 microprocessor loads the contents of the IRQ vector low ($FFFE) and high ($FFFF) into the program counter.

Since the BRK instruction causes a vector to the same location as an $\overline{\text{IRQ}}$-generated interrupt, the IRQ interrupt service routine must have an additional attribute that was not mentioned in the previous discussion of that routine. The IRQ interrupt service routine must include instructions that determine whether the routine is being executed due to an external IRQ or due to a BRK instruction. The easiest way to make this determination is to investigate the state of the BRK Command bit (B) in the processor

status register value on the stack. If the B bit is a 1, the routine is being executed due to a BRK instruction; otherwise, the routine is being executed due to $\overline{IRQ}$. Example 7-2 shows such a decision-making sequence.

Example 7-2: Determining Whether BRK or $\overline{IRQ}$ Caused an Interrupt

```
PLA           ;PULL STATUS REGISTER FROM STACK
PHA           ;  AND RESTORE IT
AND  #$10     ;ISOLATE B FLAG
BNE  BREAK    ;BRANCH IF CAUSED BY BRK
 .            ;OTHERWISE, PROCESS IRQ
 .
 .
```

Where are BRK instructions used? Usually BRK instructions are inserted at one or more points in a program wherever you would like to halt the 6502 microprocessor and "take stock" of what your program has done to that point. Perhaps you might want to check the status of a flag in memory, or examine the current value in a counter; BRK instructions provide the halts that are necessary to make these checks. These BRK instructions can also function as program flow indicators to trace the path a program takes as it is executed; a well-placed scattering of BRK instructions in a program provides a good "road map" to compare against the flowchart of the program. In this application, BRK instructions are often used to identify sequences that are *not* being executed by the program; if the BRK never occurs, you can be sure that that particular portion of code is not being executed.

How are BRK instructions put into a program? Since BRK instructions are temporary instructions, they must *overlay* an existing instruction. BRK is a one-byte instruction, so it will overlay only an op-code byte; if a multibyte instruction is overlayed with BRK, its operand byte(s) will remain intact. The designers of the 6502 microprocessor assumed that, in most cases, the BRK instruction would be used to overlay a multibyte instruction, so *the program counter value on the stack (the return address) addresses the second byte after the BRK instruction.* This presents no problem if BRK replaces a two-byte instruction, but what if you want BRK to replace a three-byte instruction? If you simply overlay the op code of a three-byte instruction with a BRK instruction, the interrupt service routine will return to the third byte of the instruction (the high-order byte of the operand), and the 6502 microprocessor will attempt to execute that byte as an instruction op code. Therefore, if you wish to break at a three-byte instruction, you must code *two* temporary instructions into the program—BRK to overlay the op code and NOP to overlay the high-order operand

Example 7-3: Using BRK to Overlay a Three-Byte Instruction

The Code is written as follows:

```
Location  Instruction
$45F9     INX
$45FA     LDA   $0503
$45FD     JMP   $4706
   .
   .
   .
```

In memory, it is arranged like this:

```
Location  Instruction
$45F9     INX
$45FA     LDA  <------ BRK is to be inserted here
$45FB     $03
$45FC     $05
$45FD     JMP
$45FE     $06
$45FF     $47
```

After BRK and NOP are overlayed, it looks like:

```
Location  Instruction
$45F9     INX
$45FA     BRK  <------ BRK overlays LDA op code
$45FB     $03
$45FC     NOP  <------ RTI causes return to here
$45FD     JMP
$45FE     $06
$45FF     $47
```

byte. Example 7-3 shows the overlays required for a three-byte instruction.

Break-Related Instructions in an Interrupt Service Routine

We previously discussed the interrupt service routine instructions needed to process $\overline{\text{IRQ}}$ interrupts (the device-polling sequence, the CLI instruction, and so on), and the instruction sequence that determines whether the routine was initiated by an active $\overline{\text{IRQ}}$ signal or a BRK instruction. What type of instructions should be included in the break-processing portion of the interrupt service routine? Since the BRK instruction is used to examine program status with the 6502 microprocessor halted, the break-processing instructions must output summary information (the contents of registers, a binary listing of the processor status register, etc.) to a printer, a CRT, or some other type of visual display device.

AIM 65 owners can gain some insight to this by studying the IRQ interrupt service routine of the AIM 65, IRQV3 (entry address $E154). This routine begins by determining whether BRK or $\overline{\text{IRQ}}$ caused the routine to be executed, and uses the same instruction

sequence shown in Example 7-2 to make this determination. If $\overline{IRQ}$ initiated the routine, the instruction JMP (IRQV4) causes a jump indirect through parameter IRQV4 (location $A400) to an IRQ interrupt routine. This parameter, IRQV4, is not initialized by the AIM 65, so the user must initialize it to address an IRQ interrupt routine of his own.

If a BRK instruction initiated the IRQ interrupt service routine IRQV3 of the AIM 65, the execution of the routine continues at label IRQ1. At IRQ1, the processor status register is pulled from the stack and stored in memory, along with the contents of the X register and the Y register. Next, the program counter is pulled from the stack and decremented by one, so that it addresses the location immediately following the BRK instruction. The program counter and the Stack Pointer are then stored in memory. With all register contents stored the routine initiates a long sequence in which the instruction following BRK is disassembled and printed. If the RUN/STEP switch is set to RUN, return is to the Monitor; otherwise return is to the instruction following BRK.

AIM 65 BREAKPOINTS

The AIM 65 microcomputer includes a *breakpoint* feature that allows you to emulate the BRK instruction without modifying any instructions. With this feature, you can specify up to four different instruction addresses as "breakpoint" addresses. The breakpoint feature operates in the following manner. With the AIM 65 in the Step mode, the 6502 microprocessor will halt each time that it is about to execute an instruction whose address has been specified as a breakpoint. At this time, the disassembled form of the breakpoint instruction will be displayed (and printed, if the printer is on) and the 6502 microprocessor will return to the Monitor. If the Register Trace mode is enabled (via the Z command), the register values will also be displayed. With the 6502 microprocessor halted, the contents of any memory location can also be examined, using the M command. As you can see, the breakpoint feature provides you with meaningful information at selected points in a program. Breakpoints are not only useful for examining the status of the program at these points, but they also indicate whether or not a program instruction is *ever* being executed. That is, breakpoints can be used to trace the flow of a program, in order to tell you whether or not the program is operating as you intended it to operate!

To resume execution from a breakpoint, you need only to enter a new G command from the keyboard. In addition to eliminating the need to modify existing program instructions, *the breakpoint feature allows you to break at any address, including those in ROM.*

The only disadvantage of the breakpoint feature is that since it requires the AIM 65 to be in single-step mode, breakpoints cannot be used to debug real-time programs. The AIM 65 breakpoints can also be disabled (but retained intact), using the Monitor's 4 command, in case you want to execute the program without them.

RESET CONSIDERATIONS

As you know, the contents of the program counter determine the memory location from which the 6502 microprocessor will fetch the next instruction to be executed. In previous chapters, we discussed how the program counter is incremented after each instruction byte is fetched in a sequentially executing program, and how the program counter can be changed if a branch, Jump (JMP), Jump to Subroutine (JSR), Return from Subroutine (RTS), or Return from Interrupt (RTI) instruction is executed, but nowhere have we mentioned how the program counter is *initialized.* That is, how it receives its initial contents when power is applied to the system. This omission is comparable to describing a trip without mentioning where the trip originated!

The signal that initializes the program counter and, thereby, initiates 6502 microprocessor operation, is an externally generated signal called $\overline{\text{RES}}$ (Reset). This signal must be applied to Pin 40 of the 6502 integrated circuit, and is used to reset or start the microprocessor from a power-down condition. While $\overline{\text{RES}}$ is held low (to ground), the 6502 microprocessor is in a "disabled" state; information can be neither written to nor read from it, and the contents of the internal registers are undefined.

After power has reached the +5-volt level on $\overline{\text{RES}}$, the 6502 microprocessor will immediately initiate a six-cycle *start sequence*[2]. During this sequence, the 6502 microprocessor sets the IRQ Disable bit (I) of the processor status register to "lock out" external interrupts while the microprocessor is being initialized, and loads the contents of memory locations $FFFC and $FFFD into the low-order and high-order bytes, respectively, of the program counter. These two locations, $FFFC and $FFFD, must contain the address of the first instruction to be executed by the 6502 microprocessor. This is the starting address of an initialization program. (Since the contents of these locations must be preserved while the power is off, they must reside in ROM. In fact, the initialization program that these locations address must also reside in ROM.)

Example 7-4 shows a typical sequence of instructions in a reset program. This program begins by initializing the Stack Pointer. The Stack Pointer is normally initialized to point to location $01FF, as it is here, because that is the starting location of the stack, but

there is no reason why a different reset program could not initialize the Stack Pointer to some other page one location. Why is the Stack Pointer initialization assigned the highest priority in this program? It is assigned this priority so that an immediate nonmaskable interrupt ($\overline{\text{NMI}}$), such as a power failure, can be properly serviced. After initializing the Stack Pointer, the program should initialize the registers of the various peripheral I/O devices in the system. The I/O initialization instructions are not shown because they depend on the configuration of the system. The next three instructions (LDA #00, TAX, and TAY) load zeroes into the accumulator, the X register, and the Y register, respectively.

Example 7-4: A 6502 Microprocessor Reset Program

```
;THIS PROGRAM INITIALIZES A 6502-BASED MICROCOMPUTER SYSTEM
;FOLLOWING POWER-ON OR RESET
RESET   LDX   #$FF    ;INITIALIZE STACK POINTER TO $01FF
        TXS
         .
         .            Configure I/O devices in the system
         .
        LDA   #00     ;INITIALIZE REGISTERS TO ZERO
        TAX
        TAY
        CLD           ;CLEAR DECIMAL MODE
        CLC           ;CLEAR CARRY
        CLI           ;ENABLE INTERRUPTS
        JMP   USERP   ;EXECUTE USER PROGRAM
```

Some system reset programs do not include register initialization, but we did it here to emphasize that the contents of these registers at power-up are undefined. The 6502 microprocessor does not clear them to zero during its six-cycle start sequence. The next two instructions, CLD and CLC, are arbitrary; some reset programs may set one or both of these bits, other reset programs may clear one and set the other. With the stack and all registers now established, the next instruction, CLI, enables IRQ interrupt requests. (You will recall that these interrupts were disabled as part of the start sequence of the 6502 microprocessor.) The final instruction in the reset program executes a jump to the program of the user, which is assigned the label USERP in this example.

Restarting a System

There are times other than power-up when you will want to be able to initialize the microcomputer system to a known state. These include instances in which a program has somehow entered an endless loop, or the malfunction of some device causes the system to "hang up." As an alternative to turning the power off, which

will cause information in R/W memory to be lost, most microcomputer systems allow the RES signal to be activated by a pushbutton (as with the AIM 65) or by some other means.

Regardless of the source of the RES signal, the response of the 6502 microprocessor will be the same. It will execute the six-cycle start sequence and fetch the address of the reset program from locations $FFFC and $FFFD. Our sample reset program (Example 7-4) contained no provision to differentiate a power-up reset from an externally initiated restart, but there is no reason why you cannot design such a provision into a system reset program. For example, the reset program of the AIM 65 Monitor (RSET, entry address $E0BF) does contain such a provision. In the *AIM 65 Microcomputer User's Guide*[3], a power-up reset is referred to as a "cold" reset and a RESET-button-initiated restart is referred to as a "warm" reset.

A "cold" reset (power-up) produces the following results:

- The processor status register, the program counter, the accumulator, the X register, and the Y register all contain a value of zero.
- The Stack Pointer contains a value of $FF, which causes it to point to location $01FF.
- The register trace and instruction trace are off.
- The printer is enabled.
- AIM 65 breakpoints are disabled.
- The on-board printer, display, and keyboard devices are initialized.
- Two Monitor cassette-tape parameters (NPUL and TIMG) in locations $A40A through $$A40D are initialized.
- Five user-alterable Monitor parameters (NMIV2, IRQV2, DILINK, TSPEED, and GAP) in locations $A402 through $A409 are initialized.

A "warm" reset produces the same results, except that it does not alter the contents of the user-alterable Monitor parameters; these parameter locations will contain the same values that they had before the RESET button was pressed.

SUMMARY

In this chapter, we discussed *interrupts* and *resets,* and an interrupt-related topic, *breaks.* There are two types of interrupts—maskable interrupt requests and nonmaskable interrupts. External devices use the $\overline{\text{IRQ}}$ line to present interrupt requests to the 6502 microprocessor. These requests are recognized only if the IRQ

Disable bit (I) of the processor status register is zero, otherwise, they are ignored. The 6502 microprocessor is notified of high-priority events, such as a power failure, on the NMI line. These interrupts cannot be disabled (ignored) and are accepted immediately after the current instruction has been executed.

Resets also come in two forms—power-up reset and restart. Both are initiated by the $\overline{\text{RES}}$ line of the 6502 microprocessor, and differ only in the software used to service them. The 6502 microprocessor services interrupts, reset, and the BRK instruction through vector pointers—these are dedicated pairs of memory locations that hold the starting address of the appropriate service routine. Table 7-1 summarizes the vector pointers.

Table 7-1. The 6502 Vector Pointers

Parameter	Vector Pointer	
	Low Address	High Address
NMI	$FFFA	$FFFB
RES	$FFFC	$FFFD
IRQ	$FFFE	$FFFF
BRK	$FFFE	$FFFF

REFERENCES

1. The idea of comparing "interrupts" to telephones was obtained from an article by Atkins, R. T. "What Is an Interrupt," *BYTE* March 1979, pp. 230–236. This excellent introduction to interrupts is highly recommended.

2. For details of the start sequence, see the *R6500 Microcomputer System Programming Manual, Section 9.2,* Rockwell International, Anaheim, CA, 1978.

3. The "warm" reset and "cold" reset of the AIM 65 are described in the *AIM 65 Microcomputer User's Guide, Sections 1.9.1, 7.3, and 7.6,* Rockwell International, Anaheim, CA.

8

General-Purpose Input/Output Devices

Interfacing a microprocessor with peripheral input/output devices is usually one of the biggest stumbling blocks for system designers. If the interfaces are constructed with discrete components (resistors, capacitors, registers, logic gates, and so on), they can easily occupy one or more printed-circuit boards, and can take months to debug so that everything is working properly. Recognizing this problem, the manufacturers of the 6500-family products have attempted to lessen the design load by offering a variety of *general-purpose I/O circuits*, or devices. These devices are not specific to any one peripheral but, instead, they include various combinations of features (parallel interface ports, counters, timers, and mixes of read/write and ROM memory storage) that are common to many different interfaces. This allows the system designer to quickly implement an interface for a particular peripheral by selecting one general-purpose I/O device and configuring that device to communicate with his peripheral.

How is a general-purpose I/O device configured for use with a particular peripheral? It is configured physically, by making the proper wiring interconnections and, perhaps, by adding a few support components (line drivers or multiplexors, for example). It is also configured by writing software programs that "talk" to the peripheral. After completing these two steps, the general-purpose I/O device *becomes* a keyboard interface, cassette-recorder interface, display interface, or an interface for some other peripheral.

Such an interface is usually not only cheaper and more reliable than the comparable interface constructed with discrete components, but it also cuts the design cycle time to a fraction of the time that it would take to do the job using discrete components.

The 6500 family includes two types of general-purpose I/O devices—I/O interface adapters and memory-I/O-timer combination devices. There are two I/O interface adapters:

- *6520 Peripheral Interface Adapter (PIA)*—The PIA has two 8-bit bidirectional I/O ports and four peripheral control/interrupt lines (two control lines for each port). Each of the 16 lines that forms the two I/O ports can be selected, under program control, to function as either an input line or an output line.
- *6522 Versatile Interface Adapter (VIA)*—The VIA has all of the features of the PIA, but it also includes two 16-bit programmable interval timers/counters and an 8-bit shift register for serial-to-parallel and parallel-to-serial conversion.

Since this book deals with software design, rather than with hardware interfacing, our discussion in this chapter of the PIA and the VIA will focus on the programming aspects of these devices. For hardware interfacing considerations, see Reference 1.

There are currently four different memory-I/O-timer combination devices in the 6500 family. They are:

- *6530 ROM-RAM-I/O-Timer (RRIOT)*—The RRIOT includes a 1024×8 ROM, a 64×8 static Read/Write memory, two 8-bit bidirectional I/O ports, and an 8-bit interval timer.
- *6531 ROM-RAM-I/O-Counter (RRIOC)*—The RRIOC includes a 2048×8 ROM, a 128×8 static Read/Write memory, an 8-bit serial channel, two bidirectional I/O ports (eight lines on one port, seven lines on the other), and a 16-bit timer/counter.
- *6532 RAM-I/O-Timer (RIOT)*—The RIOT includes a 128×8 static Read/Write memory, two 8-bit bidirectional I/O ports, and a programmable 8-bit interval timer.
- *6534 ROM-I/O-Counter (RIOC)*—The RIOC includes a 4096×8 ROM, an 8-bit serial channel, two bidirectional I/O ports (14 data lines total), and a 16-bit counter/latch with interval timer, pulse generator, and event counter modes.

Due to space considerations, the memory-I/O-timer combination devices are not covered in this book. Interested readers should consult References 1 and 2 for descriptions of the 6530 RRIOT and the 6532 RIOT devices, and should contact Rockwell Interna-

tional, Synertek, or MOS Technology for literature on the 6531 RRIOC and the 6534 RIOC devices.

Integrated circuit manufacturers also offer four peripheral controller devices:

- 6541 Programmable Keyboard/Display Controller (PKDC).
- 6545 CRT Controller (CRTC).
- 6592 Printer Controller.
- 6551 Asynchronous Communication Interface Adapter (ACIA).

Again, interested readers should contact the individual manufacturers for details on their products.

THE 6520 PERIPHERAL INTERFACE ADAPTER (PIA)

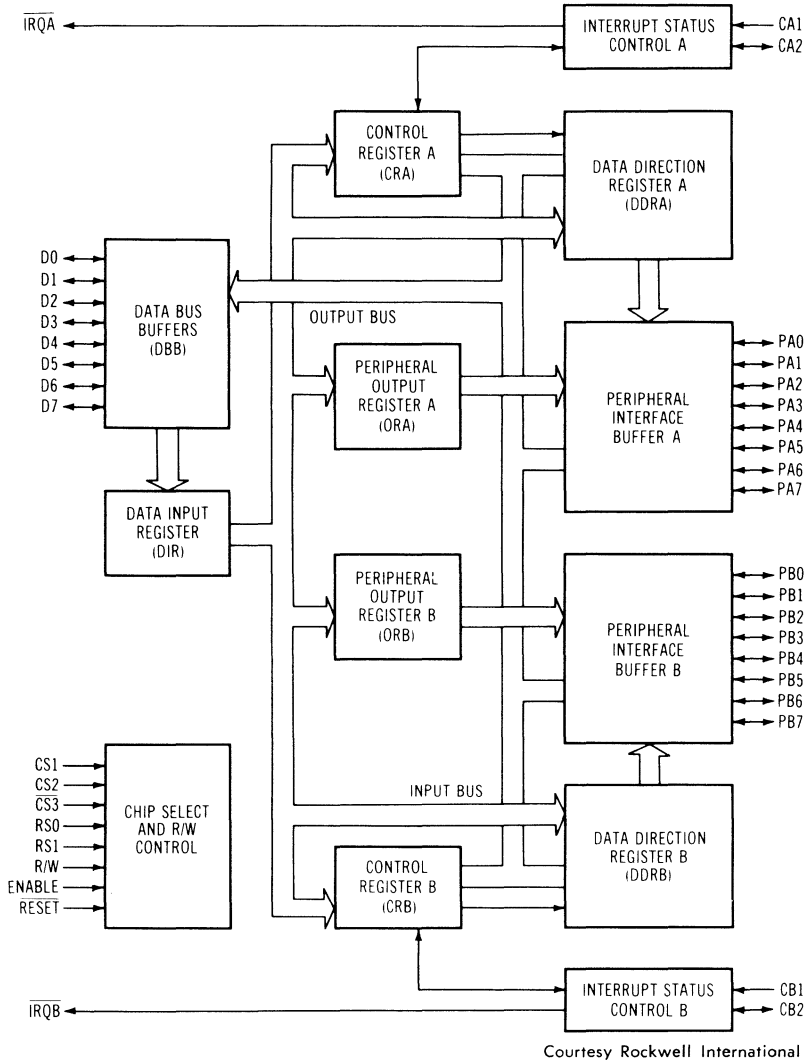
The 6520 IC is one of the most commonly used I/O devices in the 6500 family. A pin-compatible equivalent of the Motorola MC6820, the 6520 PIA provides virtually all of the necessary circuitry to interface a 6500-family microprocessor to a printer, display, keyboard, bank of switches, or any variety of other peripheral devices. The PIA communicates with the microprocessor on the system buses (data, address, and control), and it communicates with attached peripherals via two 8-bit ports, called Port A and Port B. Port A is usually an input port and Port B is usually an output port, but each of the 16 lines that comprise the two ports can be independently programmed to function as either an input or an output port. Each port also has two control lines; one is an input line that presents peripheral status to the PIA, and the other can be selected as either an input line or as a multimode output line.

Fig. 8-1 shows the internal block diagram of the PIA. The microprocessor side of the interface contains:

- *Data Bus Buffers* that transfer the contents of the system Data Bus to and from the microprocessor.
- A *Data Input Register* that latches data from the microprocessor during the $\phi 2$ clock pulse, for subsequent transfer to one of the six addressable registers of the PIA.
- *Chip Select* and *R/$\overline{W}$ Control Logic* that accepts address and control information from the microprocessor.

Each bidirectional port (Port A and Port B) is supported by:

- A *Data Direction Register*. Each bit of the data direction register determines whether its corresponding port line shall function as an input (0) line or an output (1) line.



Courtesy Rockwell International

Fig. 8-1. Block diagram of the 6520 Peripheral Interface Adapter.

- A *Control Register* that holds the interrupt status flags of the port and selects internal logic connections within the PIA.
- A *Peripheral Output Register* that holds data being transferred from the microprocessor to the attached peripheral.
- *Two control lines* that are configured by the contents of the *Control Register.*
- A *Peripheral Interface Buffer* that drives the I/O port lines.

PIA REGISTER ADDRESSING

Six of the internal registers of the PIA are addressable—peripheral interface buffers A and B, data direction registers DDRA and DDRB, and control registers CRA and CRB. The register addressing is performed by register select lines RS0 and RS1, which are normally tied to low-order address lines A0 and A1, respectively. As you know, two address lines can address only four locations in memory. Two of these six locations are *shared* by the peripheral interface buffers and the control registers. Bit 2 of the control register determines whether the peripheral interface buffer (1) or the data direction register (0) is addressed by register select lines RS0 and RS1. This address sharing presents no inconvenience to the programmer because the data direction registers must be accessible only at power-up initialization time, when they are used to configure the individual port lines as inputs or outputs. Once the data direction registers have been initialized, their contents remain constant and the shared addresses are used to write data into, or read data out of, the peripheral interface buffers. Table 8-1 summarizes register addressing for the PIA.

Table 8-1. Addressing PIA Internal Registers

Register Select (Address) Lines		Control Register DDR Access Bit		
RS1	RS0	CRA-2	CRB-2	Register Selected
0	0	1	X	Peripheral interface buffer A.
0	0	0	X	Data direction register A.
0	1	X	X	Control register A.
1	0	X	1	Peripheral interface buffer B.
1	0	X	0	Data direction register B.
1	1	X	X	Control register B.

X = 0 or 1.

With register select lines RS0 and RS1 tied to address lines A0 and A1, the four PIA register addresses represent four consecutive memory locations. For the remainder of this section, we will use four symbolic labels to denote the four addresses:

- PIAD represents the address shared by peripheral interface buffer A and data direction register A.
- PIAC represents the address of control register A.
- PIBD represents the address shared by peripheral interface buffer B and data direction register B.
- PIBC represents the address of control register B.

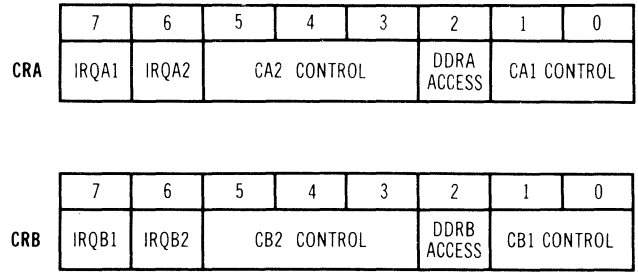


Fig. 8-2. Organization of the PIA control registers.

PIA CONTROL REGISTERS

Fig. 8-2 shows the organization of the PIA control registers CRA and CRB. As can be seen, each of these registers is divided into five fields:

- *Bits 0 and 1, CA1 (CB1) Control*—These bits determine whether the interrupt status bit in Bit 7, IRQA1 (IRQB1), is set on the high-to-low or low-to-high transition of the control line CA1 (CB1), and whether the status bit setting will also activate the interrupt request line, $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$). Table 8-2 sumarizes the combinations of Bits 0 and 1.
- *Bit 2, DDRA (DDRB) Access*—This bit selects the data direction register (0) or the peripheral interface buffer (1), as previously described.
- *Bits 3, 4, and 5, CA2 (CB2) Control*—The CA2 (CB2) control line can function as either an input to the PIA or as an output

Table 8-2. Control of Interrupt Inputs CA1 and CB1

CRA Bit 1	(CRB) Bit 0	Transition of Interrupt Input CA1 (CB1)	Bit 7, Interrupt Flag IRQA1 (IRQB1)	Interrupt Request $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$)
0	0	↓ Active	Set on ↓ of CA1 (CB1)	Disabled—IRQ remains high.
0	1	↓ Active	Set on ↓ of CA1 (CB1)	$\overline{\text{IRQ}}$ goes low when IRQA1 (IRQB1) goes high.
1	0	↑ Active	Set on ↑ of CA1 (CB1)	Disabled—$\overline{\text{IRQ}}$ remains high.
1	1	↑ Active	Set on ↑ of CA1 (CB1)	$\overline{\text{IRQ}}$ goes low when IRQA1 (IRQB1) goes high.

Notes:
1. ↑ indicates positive transition (low to high).
2. ↓ indicates negative transition (high to low).

Table 8-3. Control of CA2 and CB2 as Interrupt Inputs (CRA5, CRB5 are low)

CRA (CRB)			Transition of Interrupt Input CA2 (CB2)	Bit 6, Interrupt Flag IRQA2 (IRQB2)	Interrupt Request $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$).
Bit 5	Bit 4	Bit 3			
0	0	0	↓ Active	Set on ↓ of CA2 (CB2)	Disabled—$\overline{\text{IRQ}}$ remains high.
0	0	1	↓ Active	Set on ↓ of CA2 (CB2)	IRQ goes low when IRQA2 (IRQB2) goes high.
0	1	0	↑ Active	Set on ↑ of CA2 (CB2)	Disabled—$\overline{\text{IRQ}}$ remains high.
0	1	1	↑ Active	Set on ↑ of CA2 (CB2)	$\overline{\text{IRQ}}$ goes low when IRQA2 (IRQB2) goes high.

Notes:
1. ↑ indicates positive transition (low to high).
2. ↓ indicates negative transition (high to low).

to the peripheral, based on the state of Bit 5. If CA2 (CB2) is configured as an *input* (Bit 5 = 0), the four combinations of Bits 3 and 4 produce the same effect on IRQA2 (IRQB2) that Bits 0 and 1 produced on IRQA1 (IRQB1). These combinations are summarized in Table 8-3.

If CA2 (CB2) is configured as an *output* (Bit 5 = 1), the combinations of Bits 3 and 4 can cause CA2 (CB2) to function in one of four different modes—as a "handshake" acknowledgment to the peripheral, as a one-cycle pulse for counting or shifting, as a low-level signal, or as a high-level signal. Tables 8-4 and 8-5 summarize these combinations for CA2 and CB2, respectively.

- *Bit 6, IRQA2 (IRQB2)*—This is an interrupt status flag that is set by an active transition on control line CA2 (CB2). If CA2 (CB2) is configured as an input, it is cleared by reading peripheral interface buffer A (B). If CA2 or CB2 is configured as an output, the clear conditions are defined in Table 8-4 or 8-5, respectively.
- *Bit 7, IRQA1 (IRQB1)*—This is an interrupt status flag that is set by an active transition on control line CA1 (CB1) and is cleared by reading peripheral interface buffer A (B).

It is the various combinations of the two control lines of each port that give the PIA true versatility for input/output. Through the control registers, you can specify which signal transition (low-to-high or high-to-low) will set the status flag of the control register and, whether or not, the setting of a status flag will generate an interrupt request ($\overline{\text{IRQA}}$ or $\overline{\text{IRQB}}$) to the 6502 microprocessor. Further, if CA2 or CB2 is specified as an output signal, the control

Table 8-4. Control of CA2 as an Output (CRA5 is high)

CRA			Mode	Operation of CA2
Bit 5	Bit 4	Bit 3		
1	0	0	Handshake on Read	CA2 is set high on an active transition of the CA1 signal and set low when the 6502 microprocessor reads Peripheral Interface Buffer A. This allows positive control of data transfers from the peripheral to the 6502 microprocessor.
1	0	1	Pulse output	CA2 goes low for one cycle after the 6502 microprocessor reads Peripheral Interface Buffer A. This pulse can be used to signal the peripheral that data was taken.
1	1	0	Level output	CA2 goes low when the 6502 microprocessor writes a 0 into CRA3, and stays low until the 6502 microprocessor writes a 1 into CRA3.
1	1	1	Level output	CA2 goes high when the 6502 microprocessor writes a 1 into CRA3, and stays high until the 6502 microprocessor writes a 0 into CRA3.

register allows you to choose one of four different modes in which this control signal will operate—a handshake mode, a pulse output mode, and two level output modes. Since Port A is normally used as an input port and Port B is normally used as an output port, the handshake and pulse output modes differ between the two ports.

The *handshake mode* is used to control peripheral devices whose operation must be synchronized to the program being executed by the 6502 microprocessor (Fig. 8-3). Such devices include a paper-tape reader (input) and a paper-tape punch (output). On Port A (the input port), CA1 acts as a *data ready* signal to the 6502 microprocessor and CA2 acts as a *data not taken* signal to the peripheral (a paper-tape reader, in our example). When the 6502 microprocessor reads the input data from the PIA, *data not taken* (CA2) is cleared, notifying the peripheral that the 6502 microprocessor is ready to accept a new data byte, if there is one.

On Port B (the output port), CB1 acts as a *peripheral ready* signal to the 6502 microprocessor and CB2 acts as a *data not available* signal to the peripheral (a paper-tape punch, in our example). When the 6502 microprocessor writes the data byte to the PIA, *data not available* (CB2) is cleared, notifying the peripheral that data is available.

The *pulse output mode* is used to inform a peripheral device that data has been read from the PIA or written to the PIA. On

Port A (the input port), CA2 acts as a one-cycle *data taken* strobe to the peripheral device. On Port B (the output port), CB2 acts as a one-cycle *data available* strobe to the peripheral device.

The *level output modes* use CA2 and CB2 to provide an active-high or active-low pulse, of arbitrary length, to the attached pe-

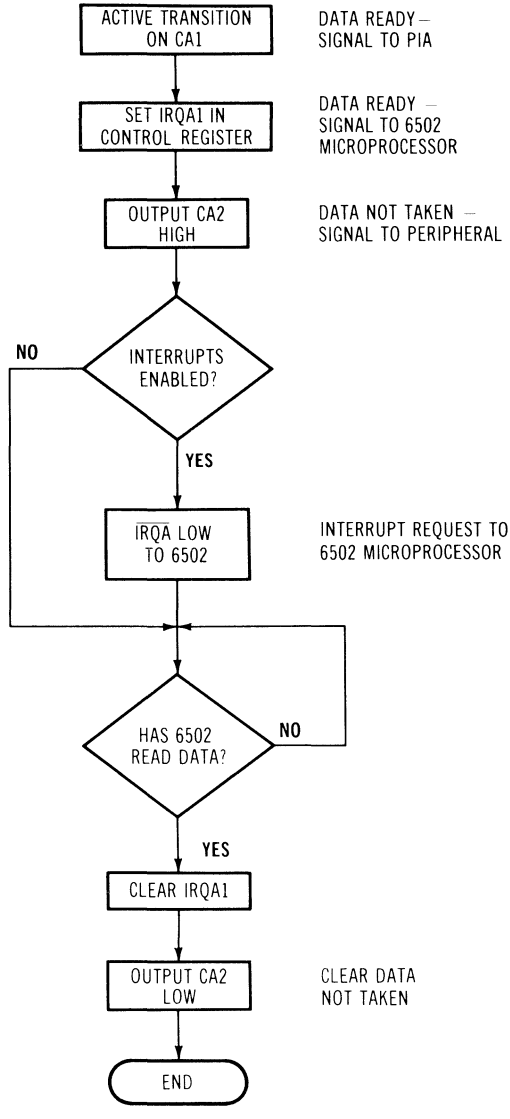


Fig. 8-3. Handshaking on Port A of a PIA.

Table 8-5. Control of CB2 as an Output (CRB5 is high)

CRB			Mode	Operation of CB2
Bit 5	Bit 4	Bit 3		
1	0	0	Handshake on Write	CB2 is set high on an active transition of the CB1 signal and set low when the 6502 microprocessor writes to Peripheral Interface B. This allows positive control of data transfers from the 6502 microprocessor to the peripheral.
1	0	1	Pulse output	CB2 goes low for one cycle after the 6502 writes data to Peripheral Interface Buffer B. This can be used to signal the peripheral that data is available.
1	1	0	Level output	CB2 goes low when the 6502 microprocessor writes a 0 into CRB3, and stays low until the 6502 microprocessor writes a 1 into CRB3.
1	1	1	Level output	CB2 goes high when the 6502 microprocessor writes a 1 into CRB3, and stays high until the 6502 microprocessor writes a 0 into CRB3.

ripheral device(s). These modes can be used to load registers, turn devices on or off, or control operating modes.

CONFIGURING THE PIA

At power-up, the system reset signal ($\overline{RES}$) automatically clears the internal registers of the PIA to zero. What configuration has this produced? It has produced the following:

- Port A and Port B are both input ports.
- Each of the four interrupt flags (IRQA1, IRQA2, IRQB1, and IRQB2) is an input that will set on the high-to-low transition of its associated interrupt input (CA1, CA2, CB1, and CB2).
- Interrupt requests $\overline{IRQA}$ and $\overline{IRQB}$ are disabled.
- The four registers that can be addressed at this time are the data direction register A, the control register A, the data direction register B, and the control register B.

This configuration can be modified to reflect your particular system by writing into the four addressable registers, beginning with data direction register A.

As described previously, the 0 or 1 state of each data direction register bit causes the associated line on the port to function as either an input (0) or an output (1). Some simple examples for Port A are:

```
LDA  #$FF  ;ALL LINES OUTPUTS
STA  PIAD  ;PROGRAM DATA DIRECTION REGISTER A
```

and

```
LDA  #$F0  ;MAKE LINES 0–3 INPUTS, 4–7 OUTPUTS
STA  PIAD  ;PROGRAM DATA DIRECTION REGISTER A
```

The data direction registers have a one-to-one correlation with the port pins, and are easy to configure.

A bit more thought must go into configuring the control registers of the PIA. Let us take a look at a few Port A examples, followed by some Port B examples. In all the examples, you will note that Bit 2 of the control register is set to a logic 1, once the data direction register has been initialized. This allows the 6502 microprocessor to address the peripheral interface buffer on subsequent accesses of the PIAD or PIBD address.

1. Port A is a simple output port, with no control lines:

```
LDA  #$FF           ;ALL LINES OUTPUTS
STA  PIAD
LDA  #%00000100     ;SELECT PERIPHERAL INTERFACE BUFFER A
STA  PIAC
```

2. Port A is an input port. A high-to-low transition on CA1 generates an interrupt request on $\overline{\text{IRQA}}$.

```
LDA  #%00000101  ;SELECT NEGATIVE TRANSITION, WITH IRQA
STA  PIAC
```

   The data direction register (PIAD) was not altered, since $\overline{\text{RES}}$ configured it as an input port.

3. Port A is an input port. A high-to-low transition on CA1 enables CA2 to output a one-cycle pulse to the peripheral on the next read operation.

```
LDA  #%00101101  ;SELECT NEGATIVE TRANSITION, WITH IRQA,
                 ;  AND CA2 PULSE OUTPUT
STA  PIAC
```

   The one-cycle *Input Completed* pulse will follow a read instruction such as:

```
LDA  PIAD
```

   Note that this instruction is reading peripheral interface buffer A, since Bit 2 of the control register is now a 1.

4. Port A is an input port. A high-to-low transition on CA1 sets CA2 high, as a handshake input acknowledge strobe.

```
LDA  #%00100101  ;SELECT NEGATIVE TRANSITION, WITH IRQA,
                 ;  AND CA2 HANDSHAKE ON READ
STA  PIAC
```

The handshake acknowledgment on CA2 will go high with the negative transition of CA1, and will stay high until a read operation such as:

```
LDA  PIAD
```

Some Port B examples are:

1. Port B is an output port. A low-to-high transition on CB1 enables CB2 to output a one-cycle pulse to the peripheral on the next Write operation.

```
LDA  #$FF          ;ALL LINES OUTPUTS
STA  PIBD
LDA  #%00101111    ;SELECT POSITIVE TRANSITION, WITH IRQB,
                   ;  AND CB2 PULSE OUTPUT
STA  PIBC
```

The one-cycle *Data Ready* pulse will follow a Write instruction such as:

```
STA  PIBD
```

Note that this instruction is writing to peripheral interface buffer B, since Bit 2 of the control register is now a 1.

2. Port B is an output port. A low-to-high transition on CB1 sets CB2 high, as a handshake output request strobe.

```
LDA  #$FF          ;ALL LINES OUTPUTS
STA  PIBD
LDA  #%00100111    ;SELECT POSITIVE TRANSITION, WITH IRQB,
                   ;  AND CB2 HANDSHAKE ON WRITE
STA  PIBC
```

The handshake signal on CB2 will go high with the positive transition of CB1, and will stay high until a Write operation such as:

```
STA  PIBD
```

3. Port B is an output port. A low-to-high transition on CB1 sets CB2 high.

```
LDA  #$FF          ;ALL LINES OUTPUTS
STA  PIBD
LDA  #%00111111    ;SELECT POSITIVE TRANSITION, WITH IRQB,
                   ;  AND LEVEL OUTPUT ON CB2
STA  PIBC
```

This signal will stay high until the 6502 microprocessor writes a zero into Bit 3 of control register B (CRB3).

DATA TRANSFERS USING A PIA

Once the data direction registers and the control registers have been properly configured, the PIA is ready to participate in data transfer operations between the 6502 microprocessor and its attached peripheral devices. You will recall that, in each of the preceding examples, we set Bit 2 of the control register. Setting that bit automatically causes the peripheral interface buffers to "occupy" the memory addresses that were previously "occupied" by the data direction registers. Therefore, the registers assigned to our symbolic labels are:

- PIAD represents the address of peripheral interface buffer A.
- PIAC represents the address of control register A.
- PIBD represents the address of peripheral interface buffer B.
- PIBC represents the address of control register B.

Which instructions are used to transfer data using a PIA? To output data to a peripheral device, you could use any one of the *store* instructions. For example, to output data to a peripheral on Port A, you could use STA PIAD, STX PIAD, or STY PIAD. To input data from a peripheral device, you could use any one of the *load* instructions. For example, to input data from a peripheral on Port A, you could use LDA PIAD, LDX PIAD, or LDY PIAD.

Your programs must also include instructions to interrogate the status bits in the control register (IRQA1, IRQA2, IRQB1, and IRQB2). Since these bits occupy Bits 6 and 7, they are readily testable using the BIT instruction, which loads the state of Bit 6 and Bit 7 into the Overflow flag (V) and the Negative flag (N), respectively, of the processor status register. For example, to interrogate the *Peripheral Ready* status on Bit 7 of Port A, you might use the sequence:

```
PNOTRD  BIT  PIAC     ;IS PERIPHERAL READY?
        BPL  PNOTRD  ;NO. LOOP UNTIL IT IS
```

One subtle point that was mentioned in our description of the control register fields, but never emphasized thereafter is:

> *The control register status bits, Bits 6 and 7, can be cleared only by reading the peripheral interface buffer.*

For this reason, programs that write data to the peripheral device must include a "dummy" Read instruction—to clear the status bits. The "dummy" Read may either precede or follow the Write (store) instruction. This requirement can be illustrated by updating the *peripheral ready* sequence just given, as shown in Example 8-1.

Example 8-1: Clearing PIA Status Bits After a Write

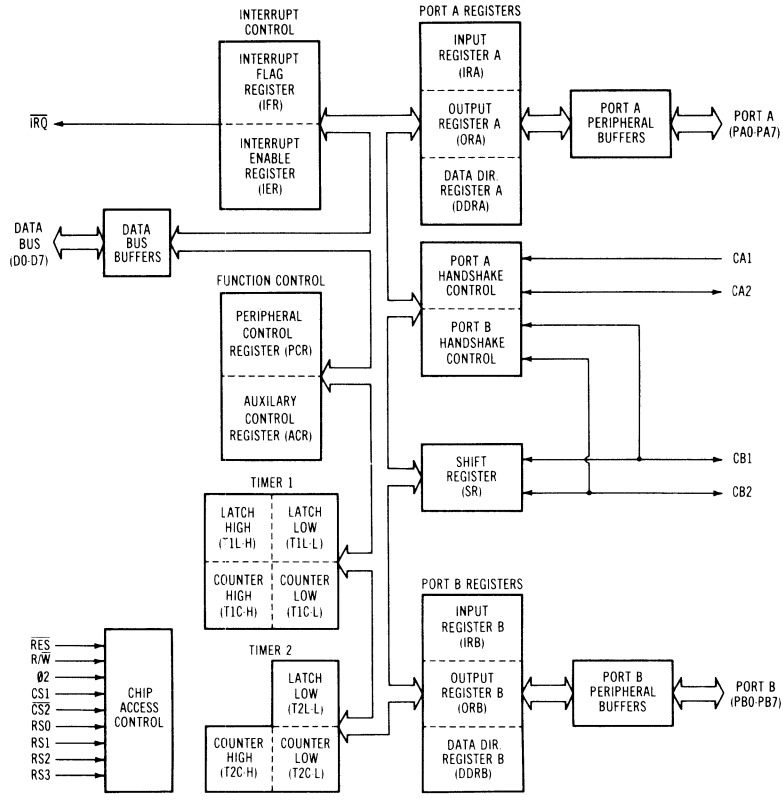
```
PNOTRD  BIT   PIAC    ;IS PERIPHERAL READY?
        BPL   PNOTRD  ;NO. LOOP UNTIL IT IS
        STA   PIAD    ;YES. OUTPUT DATA TO PERIPHERAL
        LDA   PIAD    ;  THEN CLEAR READY FLAG
```

THE 6522 VERSATILE INTERFACE ADAPTER (VIA)

The 6522 Versatile Interface Adapter (VIA) provides all of the capability of the 6520 PIA, but, in addition, it contains a pair of interval timers, a shift register, and input data latching on the peripheral interface ports. The expanded handshaking capability also allows control of data transfers between VIAs in multiple-processor configurations.

Like the 6520 PIA, the 6522 VIA has two bidirectional programmable ports. However, several I/O lines of the VIA can be controlled from the interval timers for generating square waves and for counting externally generated pulses. Fig. 8-4 shows the internal block diagram of the VIA. The internal elements include:

- *Data Bus Buffers* that transfer the contents of the system data bus to and from the microprocessor.
- *Interrupt Control Logic* that is capable of generating an interrupt request signal ($\overline{\text{IRQ}}$) to the microprocessor. This logic contains two 8-bit registers, an interrupt flag register (IFR), and an interrupt enable register (IER). The interrupt flag register maintains interrupt request flags for seven different conditions in the VIA. The interrupt enable register permits the interrupt request flags of the IFR to be selectively enabled or disabled (masked).
- *Function Control Logic* that contains two 8-bit control registers, a peripheral control register (PCR), and an auxiliary control register (ACR). The peripheral control register provides transition and control-line direction functions similar to those provided by the control register in the 6520 PIA. The auxiliary control register provides control over the timers and shift register of the VIA, and performs latch enable/disable selection for the two ports.
- *Two interval timers* (*Timer 1 and Timer 2*). Each timer has a 16-bit counter and a latch that holds the count value.
- An *8-bit Shift Register* (*SR*) that can perform serial-to-parallel and parallel-to-serial conversions.
- Two *8-bit bidirectional ports* (*Port A and Port B*), similar to those provided in the PIA, but with several valuable enhancements.



Courtesy Rockwell International

Fig. 8-4. Block diagram of the 6522 Versatile Interface Adapter (VIA).

Since the VIA is basically an enhanced PIA, the discussion of the VIA will begin with a description of register addressing and data transfers using the two ports of the VIA. We will then proceed to a discussion of two VIA-unique features—the timers and the shift register. The differences between the VIA and the PIA will also be identified where appropriate.

VIA REGISTER ADDRESSING

You will recall that the PIA occupied four locations in memory that were shared by six registers, and that register addressing was based on two register select lines and (for sharing) a bit in the control register. The VIA occupies 16 locations in memory, through which 12 registers (and each 8-bit byte of the 16-bit timers) can be accessed. The register is uniquely addressed by the four register

select lines, RS0 through RS3, of the VIA. Table 8-6 summarizes the addresses for each of the registers of the VIA.

At this point, you need not worry too much about the meanings of all the addressing combinations, but you should note that timer addressing is varied, based on whether a Read or Write operation is taking place. You should also note that the Port A output register (ORA) occupies two locations—one which will cause the handshaking control lines to be changed (select code 0001_2) and another in which the handshaking control lines are not changed (select code 1111_2).

PARALLEL DATA TRANSFERS USING A VIA

Three different VIA registers are used to transfer parallel data between the 6502 microprocessor and a peripheral—the data direction register (DDRA, DDRB), the input register (IRA, IRB), and the output register (ORA, ORB). The VIA data direction registers perform the same function as they do in the PIA; they specify whether each I/O line on the port is to act as an input (0) or an output (1). As in the PIA, the *data direction registers* of the VIA are normally initialized at power-up, and remain unaltered thereafter. The *input registers* provide temporary storage for peripheral data that the 6502 microprocessor will read into its accumulator, X register, or Y register with a load instruction. Similarly, the *output registers* provide temporary storage for microprocessor data that the 6502 microprocessor writes into the VIA with a store instruction. The VIA permits input data on both ports, and output data on Port B, to be *latched.* In order to understand how the input registers and the output registers operate during data transfers, a brief overview of latching will now be presented.

Some peripheral devices generate a strobe signal when valid information is output on their data lines, or when they are ready to accept data from the microprocessor. This type of strobe signal has already been described in the PIA section of this chapter in the discussion on handshaking (Fig. 8-3 and the accompanying text). You will recall that during an input operation, the peripheral device informs the 6502 microprocessor that it has sent data to the PIA by making an "active transition" on control line CA1 or CB1. This transition causes an interrupt status flag, IRQA1 or IRQB1, to be set in the control register.

The VIA operates in a manner similar to the PIA. In the VIA, the active transition on CA1 or CB1 causes a bit to be set in the *interrupt flag register* (*IFR*) of the VIA; CA1 causes the CA1 interrupt flag (IFR1) to be set, CB1 causes the CB1 interrupt flag (IFR4) to be set. The VIA gives you the option of using these

Table 8-6. Addressing the VIA Internal Registers

Select Lines				Register Selected	
RS3	RS2	RS1	RS0	Write Operations	Read Operations
0	0	0	0	Write output register B (ORB). Clear CB2 and CB1 interrupt flags (IFR3 and IFR4).	Read input register B (IRB). Clear CB2 and CB1 interrupt flags (IFR3 and IFR4).
0	0	0	1	Write output register A (ORA), **with effect on handshaking.** Clear CA2 and CA1 interrupt flags (IFR0 and IFR1).	Read input register A (IRA), **with effect on handshaking.** Clear CA2 and CA1 (IFR0 and IFR1).
0	0	1	0	Write data direction register B (DDRB).	Read data direction register B (DDRB).
0	0	1	1	Write data direction register A (DDRA).	Read data direction register A (DDRA).
0	1	0	0	Write Timer 1 latch low byte.	Read Timer 1 counter low byte. Clear T1 interrupt flag (IFR6).
0	1	0	1	Write Timer 1 latch high byte. Clear T1 interrupt flag (IFR6). Initiate counting.	Read Timer 1 counter high byte.
0	1	1	0	Write Timer 1 latch low byte.	Read Timer 1 latch low byte.
0	1	1	1	Write Timer 1 latch high byte. Clear T1 interrupt flag (IFR6).	
1	0	0	0	Write Timer 2 latch.	Read Timer 2 counter low byte. Clear T2 interrupt flag (IFR5).
1	0	0	1	Write Timer 2 counter high byte. Clear T2 interrupt flag (IFR5). Initiate counting.	Read Timer 2 counter high byte.
1	0	1	0	Write shift register (SR). Clear SR interrupt flag (IFR2).	Read shift register (SR). Clear SR interrupt flag (IFR2).
1	0	1	1	Write auxiliary control register (ACR).	Read auxiliary control register (ACR).
1	1	0	0	Write peripheral control register (PCR).	Read peripheral control register (PCR).
1	1	0	1	Write interrupt flag register (IFR).	Read interrupt flag register (IFR).
1	1	1	0	Write interrupt enable register (IER).	Read interrupt enable register (IER).
1	1	1	1	Write output register A (ORA), **with no effect on handshaking.** Clear CA2 and CA1 interrupt flags (IFR0 and IFR1).	Read input register A (IRA), **with no effect on handshaking.** Clear CA2 and CA1 interrupt flags (IFR0 and IFR1).

interrupt flags to latch the data into the input register or (for Port B) the output register of the port, based on whether a latch enable bit in the *auxiliary control register* (*ACR*) of the VIA is set to logic 0 (latching disabled) or logic 1 (latching enabled). Bit 0

of the auxiliary control register is the Port A latch enable and Bit 1 of the auxiliary control register is the Port B latch enable. Fig. 8-5 illustrates input data latching on Port A. We will now summarize Read and Write operations on both Port A and Port B. At this point, we will not describe how to select the polarity of the active transition (high-to-low or low-to-high) or the four control modes (handshake, pulse output, and the two level outputs), but will cover these topics later in the section, when we discuss the peripheral control register of the VIA.

Reading data from a peripheral port causes the contents of the input register (IRA or IRB) to be transferred onto the data bus. With latching disabled on Port A (ACR0 = 0), IRA reflects the data that is currently on the Port A I/O lines. With latching enabled on Port A (ACR0 = 1), IRA reflects the state of the Port A I/O lines at the time that the CA1 interrupt flag (IFR1) was set by an active transition on control line CA1.

Similarly, with latching disabled on Port B (ACR1 = 0), IRB reflects the data that is currently on the Port B I/O lines. With latching enabled on Port B (ACR1 = 1), IRB reflects the state of the Port B I/O lines at the time the CB1 interrupt flag (IFR4) was set by an active transition on control line CB1.

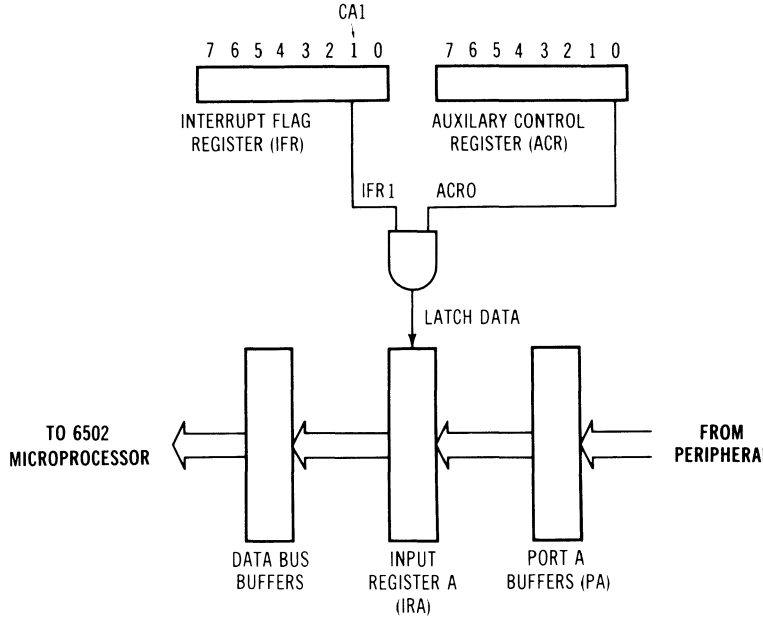


Fig. 8-5. Input data latching on Port A.

Example 8-2: A Simple VIA Input Routine

```
LDA  #FF    ;MAKE PORT A INPUTS
STA  $A003
LDA  $A001  ;INPUT DATA
STA  $40    ;  AND STORE IT
```

Output register A cannot be latched, so during the time that data is being written to a peripheral attached to Port A, this register always reflects the data that is on the Port A I/O lines, regardless of the state of the Port A latch enable (ACR0). With latching disabled on Port B (ACR1 = 0), however, ORB reflects the data that is currently on the Port B I/O lnes. This condition allows proper data to be read if the output pin is not permitted to attain full voltage (for example, driving transistors), because the input register B (IRB) reflects the contents of ORB, rather than the I/O pin. With latching enabled on Port B (ACR1 = 1), ORB reflects the state of the Port B I/O lines at the time that the CB1 interrupt flag (IFR4) was set by an active transition on control line CB1.

Before concluding this introduction to VIA data transfers, we should look at a couple of fundamental programming examples. Example 8-2 shows an instruction sequence that fetches data from a simple input device (e.g., a set of switches) and stores it in memory location $40. Example 8-3 shows an instruction sequence that sends data to a simple output device (e.g., a set of LEDs or relays) from location $40. Both examples assume that the VIA occupies addresses $A000 through $A00F. (The author did not pull this address range out of thin air; it happens to be the addresses that are occupied by the user-dedicated 6522 VIA on the AIM 65 microcomputer board.)

Example 8-3: A Simple VIA Output Routine

```
LDA  #FF    ;MAKE PORT B OUTPUTS
STA  $A002
LDA  $40    ;FETCH DATA
STA  $A000  ;  AND OUTPUT IT
```

VIA PERIPHERAL CONTROL REGISTER (PCR)

The *VIA Peripheral Control Register* (*PCR*) provides one of the functions of the PIA control register—specifying the operation of the port peripheral control lines—but you will see that the control features of the VIA are much more versatile than those of the PIA. Specifically, the VIA gives the option of preventing the interrupt status flags in the interrupt flag register (IFR) from being cleared by data transfer operations (unlike the PIA, in which a data

transfer operation automatically clears the interrupt status flag). The VIA also provides Read and Write handshaking on Port A and Write handshaking on Port B (the PIA has no Write handshaking on Port A). Fig. 8-6 shows the organization of the peripheral control register of the VIA. As you can see, the PCR has four fields:

- *Bit 0 CA1 Control*—This bit determines whether the CA1 interrupt flag (IFR1) will be set by a high-to-low (0) or low-to-high (1) transition of control line CA1.
- *Bits 1, 2, and 3, CA2 Control*—The CA2 control line can function as either an input to the VIA or as an output to the peripheral device, based on the state of Bit 3. If CA2 is configured as an *input* (Bit 3 = 0), then Bit 2 determines whether the CA2 interrupt flag (IFR0) will be set on a high-to-low (0) or low-to-high (1) transition of control line CA2. Bit 1 will determine the conditions under which the CA2 interrupt flag

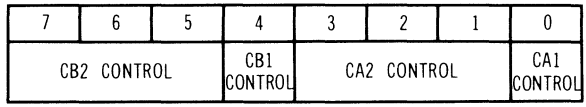


Fig. 8-6. Organization of the VIA peripheral control register.

(IFR0) will be cleared. The CA2 interrupt flag will always be cleared if you write a logic 1 into IFR0, but Bit 1 of the PCR allows you to specify whether or not IFR0 shall also be cleared when output register A (ORA) is accessed with a Read or a Write. If PCR1 = 1, IFR0 can be cleared only by writing a logic 1 into IFR0; if PCR1 = 0, IFR0 will be cleared by reading or writing ORA using either of its select codes (0001_2 or 1111_2), as well as by writing a logic 1 into IFR0. Table 8-7 sumarizes these conditions.

If CA2 is configured as an *output* (Bit 3 = 1), the combinations of Bits 1 and 2 can cause CA2 to function in one of four different modes—as a "handshake" signal to the peripheral during Read and Write operations, as a one-cycle pulse for counting or shifting, as a low-level signal, or as a high-level signal. Table 8-8 summarizes these conditions. Note that in the handshaking and pulse output modes, CA2 is affected only by a Read or Write to output register A (ORA) using the 0001_2 select code. CA2 is unaffected if ORA is accessed via its alternate select code, 1111_2.

- *Bit 4, CB1 Control*—This bit determines whether the CB1 in-

Table 8-7. Control of CA2 as an Input (PCR3 is low)

PCR			CA2 Interrupt Flag (IFR0)	
Bit 3	Bit 2	Bit 1	Set by	Cleared by
0	0	0	↓ of CA2	Read or Write of ORA, or by writing logic 1 into IFR0.
0	0	1	↓ of CA2	Writing logic 1 into IFR0.
0	1	0	↑ of CA2	Read or Write of ORA, or by writing logic 1 into IFR0.
0	1	1	↑ of CA2	Writing logic 1 into IFR0.

Notes:
1. ↑ indicates positive transition (low to high).
2. ↓ indicate negative transition (high to low).

terrupt flag (IFR4) will be set by a high-to-low (0) or low-to-high (1) transition of control line CB1.

- *Bit 5, 6, and 7, CB2 Control*—The CB2 control line can function as either an input to the VIA or as an output to the peripheral device, based on the state of Bit 7. If CB2 is configured as an *input* (Bit 7 = 0), then Bit 6 determines whether the CB2

Table 8-8. Control of CA2 as an Output (PCR3 is high)

PCR				
Bit 3	Bit 2	Bit 1	Mode	Operation of CA2
1	0	0	Handshake on Read or Write	CA2 is set high on an active transition of the CA1 signal and set low when the 6502 microprocessor Reads or Writes output register A (code 0001_2).
1	0	1	Pulse output	CA2 goes low for one cycle after the 6502 microprocessor Reads or Writes output register A (code 0001_2).
1	1	0	Level output	CA2 is held low in this mode.
1	1	1	Level output	CA2 is held high in this mode.

interrupt flag (IFR3) wil be set on a high-to-low (0) or low-to-high (1) transition of control line CB2. Bit 5 determines the conditions under which the CB2 interrupt flag (IFR3) will be cleared. The CB2 interrupt flag will always be cleared if you write a logic 1 into IFR3, but Bit 5 allows you to specify whether or not IFR3 shall also be cleared on an access (a Read or a Write) of output register B (ORB). If PCR5 = 1, IFR3 can be cleared only by writing a logic 1 into IFR3; if PCR5 = 0, IFR3 will be cleared by reading or writing ORB,

Table 8-9. Control of CB2 as an Input (PCR7 is low)

PCR			CB2 Interrupt Flag (IFR3)	
Bit 7	Bit 6	Bit 5	Set by	Cleared by
0	0	0	↓ of CB2	Read or Write of ORB, or writing logic 1 into IFR3.
0	0	1	↓ of CB2	Writing logic 1 into IFR3.
0	1	0	↑ of CB2	Read or Write of ORB, or writing logic 1 into IFR3.
0	1	1	↑ of CB2	Writing logic 1 into IFR3.

Notes:
1. ↑ indicates positive transition (low to high).
2. ↓ indicates negative transition (high to low).

as well as by writing a logic 1 into IFR3. Table 8-9 summarizes these conditions.

If CB2 is configured as an *output* (Bit 7 = 1), the combinations of Bits 5 and 6 can cause CB2 to function in one of four different modes—as a "handshake" signal to the peripheral during Write operations, as a one-cycle pulse for counting or shifting, as a low-level signal, or as a high-level signal. Table 8-10 summarizes these conditions.

Table 8-10. Control of CB2 as an Output (PCR7 is high)

PCR				
Bit 7	Bit 6	Bit 5	Mode	Operation of CB2
1	0	0	Handshake on Write	CB2 is set high on an active transition of the CB1 signal and set low when the 6502 microprocessor Writes output register B.
1	0	1	Pulse output	CB2 goes low for one cycle after the 6502 microprocessor Reads or Writes output register B.
1	1	0	Level output	CB2 is held low in this mode.
1	1	1	Level output	CB2 is held high in this mode.

Programming Examples For the VIA Control Modes

It is time now to look at several simple examples of the types of instruction sequences that one would use with the VIA control modes that have just been discussed. For these examples, assume that the VIA occupies addresses $A000 through $A00F.

Example 8-4: An Input Data Transfer With One Control Signal

```
        LDA   #00      ;MAKE PORT A INPUTS
        STA   $A003
        STA   $A00C    ;SET CA1 INTERRUPT FLAG ON
                       ;  NEGATIVE TRANSITION
CHKCA1  LDA   $A00D    ;FETCH IFR
        AND   #2       ;DATA READY?
        BEQ   CHKCA1   ;LOOP UNTIL DATA READY
        LDA   $A001    ;  THEN INPUT DATA
        STA   $40      ;  AND STORE IT
```

The routine in Example 8-4 fetches data from an input device that produces an active high-to-low data ready signal, and stores the data in memory location $40. Example 8-5 is a similar routine that illustrates an output operation. The routine in this example sends data to an output device that produces an active low-to-high peripheral ready signal; the data is contained in memory loca-

Example 8-5: An Output Data Transfer With One Control Signal

```
        LDA   #$FF     ;MAKE PORT B OUTPUTS
        STA   $A002
        LDA   #$10     ;SET CB1 INTERRUPT FLAG ON
                       ;  POSITIVE TRANSITION
        STA   $A00C
CHKCB1  LDA   $A00D    ;FETCH IFR
        AND   #$10     ;PERIPHERAL READY?
        BEQ   CHCKB1   ;LOOP UNTIL PERIPHERAL READY
        LDA   $40      ;  THEN FETCH DATA
        STA   $A000    ;  AND OUTPUT IT
```

tion $40. Example 8-6 shows a routine that is identical to the routine given in Example 8-4—input from a device that produces an active high-to-low data ready signal—except that Example 8-6 assumes that the input device requires a handshake signal. Adding handshaking requires only one additional instruction, LDA #08. Example 8-7 illustrates the programming required for another one of the four control modes of the VIA, the pulse output mode. The routine in Example 8-7 fetches data from an input device that re-

Example 8-6: An Input Data Transfer With Handshaking

```
        LDA   #00      ;MAKE PORT A INPUTS
        STA   $A003
        LDA   #08      ;SET CA2 HANDSHAKE MODE WITH
        STA   $A00C    ;  CA1 INTERRUPT FLAG SET ON
                       ;  NEGATIVE TRANSITION
CHKCA1  LDA   $A00D    ;FETCH IFR
        AND   #02      ;DATA READY?
        BEQ   CHKCA1   ;LOOP UNTIL DATA READY
        LDA   $A001    ;  THEN INPUT DATA
        STA   $40      ;  AND STORE IT
```

Example 8-7: An Input Data Transfer That Produces a Data-Accepted Pulse

```
        LDA   #00      ;MAKE PORT A INPUTS
        STA   $A003
        LDA   #$0A     ;SET CA2 PULSE OUTPUT MODE WITH
        STA   $A00C    ;  CA1 INTERRUPT FLAG SET ON
                       ;  NEGATIVE TRANSITION
CHKCA1  LDA   $A00D    ;FETCH IFR
        AND   #02      ;DATA READY?
        BEQ   CHKCA1   ;LOOP UNTIL DATA READY
        LDA   $A001    ;  THEN INPUT DATA
        STA   $40      ;  AND STORE IT
```

quires a brief data accepted pulse (for multiplexing or control purposes), and stores the data in memory location $40.

We have thus far discussed register addressing on the VIA, and have described several aspects of configuring the VIA to perform parallel data transfers. In the course of this description, we learned:

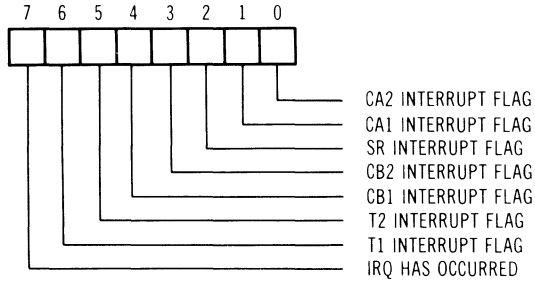
1. The direction of the port I/O pins is defined by the contents of the data direction registers (DDRA and DDRB).
2. Input data can be latched on both ports and output data can be latched on Port B. Latching is enabled or disabled based on a bit in the auxiliary control register (ACR).
3. The polarity, direction, and operation of the port peripheral control lines (CA1, CA2, CB1, and CB2) are defined by the contents of the peripheral control register (PCR).

VIA INTERRUPT REQUESTS

The VIA has only one interrupt request line to the 6502 microprocessor. This line, IRQ, can be activated (driven low) by any of seven different conditions, but in order for one of these conditions to activate IRQ, two conditions must be met:

1. The interrupt flag register (IFR) bit that represents that condition must be set to logic 1.
2. The associated bit in the interrupt enable register (IER) must also be set to logic 1.

Fig. 8-7 shows the organization of the interrupt flag register for the VIA, and summarizes the conditions that set and clear each bit. The seven potential interrupt conditions are represented by the seven low-order bits of IFR. The most-significant bit, *IRQ Has Occurred*, will be set if one or more of the IFR bits is set to logic 1 when its corresponding bit in the interrupt enable register is also set to logic 1. This most-significant bit, *IRQ Has Occurred*, activates the $\overline{\text{IRQ}}$ interrupt request signal. (Of course, $\overline{\text{IRQ}}$ from a VIA, or any other component in the system, will not be recognized by the



BIT	SET BY	CLEARED BY (WITH SELECT CODE)
0	ACTIVE TRANSITION ON CA2	READING OR WRITING ORA (0001_2 OR 1111_2)
1	ACTIVE TRANSITION ON CA1	READING OR WRITING ORA (0001_2 OR 1111_2)
2	COMPLETION OF EIGHT SHIFTS	READING OR WRITING SR (1010_2)
3	ACTIVE TRANSITION ON CB2	READING OR WRITING ORB (0000_2)
4	ACTIVE TRANSITION ON CB1	READING OR WRITING ORB (0000_2)
5	TIMEOUT OF TIMER 2	READING TIMER 2 COUNTER LOW BYTE (1000_2) OR WRITING TIMER 2 COUNTER HIGH BYTE (1001_2)
6	TIMEOUT OF TIMER 1	READING TIMER 1 COUNTER LOW BYTE (0100_2) OR WRITING TIMER 1 COUNTER HIGH BYTE (0101_2)
7	ANY IFR BIT WITH CORRESPONDING IER BIT ALSO SET	WRITING LOGIC 0 TO APPROPRIATE BIT(S) IN IFR (1101_2) OR IER (1110_2)

Fig. 8-7. Organization of the VIA interrupt flag register.

6502 microprocessor unless the IRQ disable bit of the 6502 processor status register is set to logic 0.)

Fig. 8-8 shows the organization of the interrupt enable register (IER). The interrupt enable register can be used to enable or disable individual interrupt requests, based on the state of Bit 7. If you Write to the IER with Bit 7 = 1, each 1 in the Bits 0 through 6 positions will enable the interrupt request that is represented by

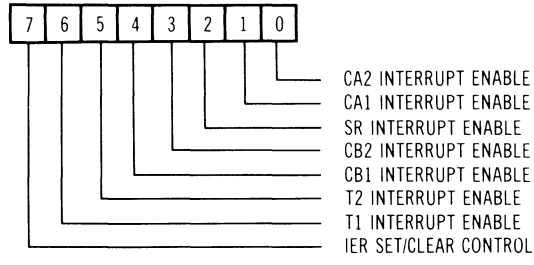


Fig. 8-8. Organization of the VIA interrupt enable register.

that bit position. For example, you could enable CA2 interrupt requests and T1 (Timer 1) interrupt requests by Writing 11000001_2 ($C1) to the IER. If you write to the IER with Bit 7 = 0, each 1 in the Bits 0 through 6 positions will disable the interrupt request represented by that bit position. For example, you could disable CA1 interrupt requests and SR (Shift Register) interrupt requests by Writing 00000110_2 ($06) to the IER. Bit 7 of the IER is active only during Write operations; if you Read the IER, Bit 7 will always appear as a logic 1. As you can see, initializing the IER requires two Write operations, one to select the enabled conditions, the other to select the disabled conditions. This dual control permits the enabled/disabled status of individual conditions to be easily changed during system operation, without altering the status of any other condition in the register.

Since the VIA can cause an interrupt request ($\overline{IRQ}$) to be generated by any of seven different conditions, a single VIA may require up to seven different interrupt service routines to be present in memory. In order to know *which* interrupt service routine to

Example 8-8: Interrupt Polling Sequence For a VIA

```
;THIS SEQUENCE IDENTIFIES THE HIGHEST-PRIORITY INTERRUPT REQUEST
;IN A VIA THAT OCCUPIES ADDRESSES $A000 THROUGH $A00F.
        LDA   $A00D     ;DOES VIA HAVE AN ACTIVE IRQ?
        BPL   NXTDEV    ;NO. GO CHECK NEXT DEVICE
        AND   $A00E     ;YES. LOGICALLY AND IFR WITH IER
        ASL   A         ;CHECK FOR T1 INTERRUPT
        BMI   JT1
        ASL   A         ;CHECK FOR T2 INTERRUPT
        BMI   JT2
        ASL   A         ;CHECK FOR CB1 INTERRUPT
        BMI   JCB1
        ASL   A         ;CHECK FOR CB2 INTERRUPT
        BMI   JCB2
        ASL   A         ;CHECK FOR SR INTERRUPT
        BMI   JSHR
        ASL   A         ;CHECK FOR CA1 INTERRUPT
        BMI   JCA1
        ASL   A         ;CHECK FOR CA2 INTERRUPT
        BMI   JCA2
        JMP   ERROR
JT1     JMP   ISRT1     ;GO SERVICE T1 INTERRUPT
JT2     JMP   ISRT2     ;GO SERVICE T2 INTERRUPT
JCB1    JMP   ISRCB1    ;GO SERVICE CB1 INTERRUPT
JCB2    JMP   ISRCB2    ;GO SERVICE CB2 INTERRUPT
JSHR    JMP   ISRSR     ;GO SERVICE SR INTERRUPT
JCA1    JMP   ISRCA1    ;GO SERVICE CA1 INTERRUPT
JCA2    JMP   ISRCA2    ;GO SERVICE CA2 INTERRUPT
NXTDEV  .
        .               (Interrupt polling for next device)
        .
```

call, the 6502 microprocessor has to know *which* of the seven interrupt conditions of the VIA caused the interrupt request. Maybe more than one condition has an active interrupt request. Perhaps all seven conditions have active interrupt requests! How do we identify active interrupt requests? Active interrupt requests are identified by *polling* the interrupt conditions. You will recall that interrupt polling was discussed earlier in Chapter 7, and an example of an interrupt polling sequence was given (Example 7-1). However, in Chapter 7, we were concerned with identifying which of several *devices* in a system caused an interrupt request, and made little or no mention of devices with multi-interrupt capability, such as the 6520 PIA and the 6522 VIA. Also from Chapter 7, we know that the order in which interrupt conditions are polled establishes a priority structure; you always want to poll the most critical condition first, and go down the line from there, in order of decreasing priority. This applies to interrupt conditions in a single device as well as interrupting devices in a system.

In polling a VIA for an active interrupt request, the system interrupt polling sequence must first find out whether *any* condition in the VIA is generating an interrupt request. How does it do this? This is accomplished by interrogating the state of the most-significant bit (Bit 7) of the interrupt flag register of the VIA. If IFR7 = 1, at least one condition in the VIA is generating an active interrupt request ($\overline{\text{IRQ}}$ is low). If the VIA has an active interrupt request, how do you identify the highest-priority condition that has an active interrupt request? Can you identify this condition by checking only the state of the bits in the interrupt flag register? No, the interrupt flag register does not provide sufficient information. Remember, in order for a condition in the VIA to generate an interrupt request, two things must be true:

1. The interrupt flag register bit must be set.
2. The corresponding bit in the interrupt enable register must be set.

Therefore, if the VIA is generating an interrupt request, the interrupt polling sequence must logically AND the interrupt flag register with the interrupt enable register to extract the "qualified" conditions from the seven interrupt conditions of the VIA, before proceeding with the identification procedure. After the AND operation is performed, the priority structure of your particular system will determine which interrupting condition is checked first, which is checked second, and so on. For illustration purposes, let us assume that we have a VIA whose high-to-low priority corresponds with the left-to-right organization of the interrupt flag register (as

shown in Fig. 8-7). That is, Timer 1 has the highest priority and CA2 has the lowest priority. Example 8-8 shows an interrupt polling sequence for that VIA. You will note that this particular priority structure allows us to check the interrupting conditions using a series of left-shift and branch instructions.

VIA AUXILIARY CONTROL REGISTER (ACR)

The *Auxiliary Control Register* (*ACR*) controls three functions within the VIA—latching on the ports, the shift register, and the two timers of the VIA. Fig. 8-9 shows the organization of the auxiliary control register. Port latching has already been discussed earlier in this chapter. The timers of the VIA and its shift register will now be described.

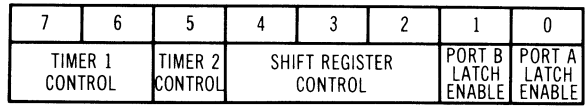


Fig. 8-9. Organization of the VIA auxiliary control register.

VIA TIMERS

The VIA has two separate 16-bit timer/counters, Timer 1 and Timer 2, that can be used to:

- Generate a single time interval, based on an $\phi 2$-system clock count loaded into the timer.
- Count high-to-low transitions on Pin PB6 (Timer 2 only), up to a pulse-count value loaded into the timer.
- Generate continuous time intervals (Timer 1 only), based on a clock-pulse count per interval loaded into the timer.
- Produce a single pulse or a continuous series of pulses on Pin PB7 (Timer 1 only), based on a count loaded into the timer.

Timer 2

Let us look first at Timer 2, which can only be used to generate a single time interval (the so-called *one-shot mode*) or used to count pulses on Pin PB6. Bit 5 of the auxiliary control register selects the operating mode of Timer 2; ACR5 = 0 selects the one-shot mode and ACR5 = 1 selects the pulse-counting mode.

Timer 2 occupies only two addresses in memory (see Table 8-11). These are the addresses associated with select codes 1000_2 and 1001_2. Select code 1000_2 is used to Read or Write the eight least-significant bits of the clock pulse count; Reading from the location

Table 8-11. Timer 2 Select Codes

Select Lines				Register Selected	
RS3	RS2	RS1	RS0	Write Operations	Read Operations
1	0	0	0	Write Timer 2 latch.	Read Timer 2 counter low byte. Clear T2 interrupt flag (IFR5).
1	0	0	1	Write Timer 2 counter high byte. Clear T2 interrupt flag (IFR2). Initiate counting.	Read T2 counter high byte.

at this select code also clears the T2 interrupt flag (IFR5). Select code 1001_2 is used to Read or Write the eight most-significant bits of the clock pulse count; Writing to the location at this select code loads the counter, clears the T2 interrupt flag (IFR5), and initiates the count operation. The completion of the count-down operation sets the T2 interrupt flag (IFR5).

As an *interval timer* (ACR5 = 0), Timer 2 sets the T2 interrupt flag (IFR5) after counting a specified number of ϕ2-system clock pulses. In a 1-MHz 6502 microprocessor, each count will generate a 1-microsecond time interval, so a count of $0010 will generate a 16-microsecond time interval, a count of $03E8 will generate a 1000-microsecond time interval, a count of $FFFF will generate a 65,535-microsecond time interval, and so on. To perform an interval timer operation with Timer 2, your program must:

1. Write to the auxiliary control register (select code 1011_2), with ACR5 set to logic 0.
2. Write the LSBY of the interval count value into the Timer 2 latch (select code 1000_2).
3. Write the MSBY of the interval count value into the high byte of the Timer 2 (select code 1001_2).
4. Clear the T2 interrupt flag (IFR5) upon completion of the interval.

Example 8-9 presents a routine that generates a 1-millisecond (i.e., 1000-microsecond) time interval, using the one-shot interval timer mode of the Timer 2. This routine assumes that the VIA occupies addresses $A000 through $A00F, the addresses of the user-dedicated VIA in the AIM 65. Note that the final instruction, LDA $A008, is a "dummy Read" that is needed for only one reason, to clear the T2 interrupt flag (IFR5). You might also observe that Timer 2 of the VIA can provide the same time-delay functions that were demonstrated in the Time-Delay Subroutines section of Chapter 3.

Example 8-9: A 1-Millisecond Time Interval Using Timer 2

```
;THIS ROUTINE SETS THE T2 INTERRUPT FLAG AT THE END OF A ONE-
;MILLISECOND TIME INTERVAL
        LDA   #00     ;SET T2 ONE-SHOT INTERVAL TIMER MODE
        STA   $A00B
        LDA   #$E8    ;WRITE COUNT LSBY
        STA   $A008
        LDA   #$03    ;WRITE COUNT MSBY AND START TIMER
        STA   $A009
        LDA   #$20    ;SET T2 INTERRUPT MASK
CHKT2   BIT   $A00D   ;HAS T2 COUNTED DOWN?
        BEQ   CHKT2   ;NO. CHECK AGAIN
        LDA   $A008   ;YES. CLEAR T2 INTERRUPT FLAG
```

As a *pulse-counter* (ACR5 = 1), Timer T2 sets the T2 interrupt flag (IFR5) after counting a specified number of negative (high-to-low) transitions on Pin PB6. To perform a pulse-counting operation, the program must supply the same functions as for the interval timer mode, except that PB6 must be specified as an input and ACR5 must be set to logic 1 (Step 1). Example 8-10 counts 10 pulses on PB6, and assumes that the VIA occupies addresses $A000 through $A00F.

Example 8-10: Pulse Counting Using Timer 2

```
;THIS ROUTINE SETS THE T2 INTERRUPT FLAG AFTER 10 PULSES HAVE
;BEEN COUNTED ON PB6
        LDA   #00     ;MAKE PORT B INPUTS
        STA   $A002

        LDA   #01     ;SET T2 PULSE-COUNTING MODE
        STA   $A00B
        LDA   #0A     ;WRITE COUNT LSBY
        STA   $A008
        LDA   #00     ;WRITE COUNT MSBY AND START COUNTING
        STA   $A009
        LDA   #$20    ;SET T2 INTERRUPT MASK
CHKT2   BIT   $A00D   ;HAS T2 COUNTED DOWN?
        BEQ   CHKT2   ;NO. CHECK AGAIN
        LDA   $A008   ;YES. CLEAR T2 INTERRUPT FLAG
```

Timer 1

Timer 1 is somewhat more complex than Timer 2 because it has four operating modes that are selected by Bits 6 and 7 of the auxiliary control register (shown in Table 8-12). Timer 1 can be used to generate a single time interval (a *one-shot mode*) or a continuous series of intervals (a *free-running mode*). Furthermore, each time-out can generate an output pulse on Pin PB7.

Timer 1 occupies four addresses in memory (see Table 8-13). These are the addresses associated with select codes 0100_2, 0101_2,

Table 8-12. Timer 1 Control

ACR		Mode	Operation
Bit 7	Bit 6		
0	0	One shot	Set the T1 interrupt flag (IFR6) once, upon time-out. Output to PB7 is disabled.
0	1	Free running	Continually set the T1 interrupt flag (IFR6), following **each** time-out cycle. Output to PB7 is disabled.
1	0	One shot	Output a low-level signal on PB7 for the time interval, and then set the T1 interrupt flag (IFR6).
1	1	Free running	Continually set the T1 interrupt flag (IFR6), and toggle the output on PB7, following **each** time-out cycle.

0110_2, and 0111_2. Select code 0100_2 is used to Read and Write the eight least-significant bits of the count value; Reading from the location at this select code also clears the T1 interrupt flag (IFR6). Select code 0101_2 is used to Read or Write the eight most-significant bits of the count value; Writing to the location at this select

Table 8-13. Timer 1 Select Codes

Select Lines					
RS3	RS2	RS1	RS0	Write Operations	Read Operations
0	1	0	0	Write Timer 1 latch low byte.	Read Timer 1 counter low byte. Clear T1 interrupt flag (IFR6).
0	1	0	1	Write Timer 1 latch high byte. Clear T1 interrupt flag (IFR6). Initiate counting.	Read Timer 1 counter high byte.
0	1	1	0	Write Timer 1 latch low byte.	Read Timer 1 latch low byte.
0	1	1	1	Write Timer 1 latch high byte. Clear T1 interrupt flag (IFR6).	Read Timer 1 latch high byte.

code loads the counter, clears the T1 interrupt flag (IFR6), and initiates the count operation. Select codes 0110_2 and 0111_2 are used to Read or Write the latches of the Timer without affecting a count-down that is in progress. This allows generation of complex waveforms in the free-running mode. Writing the most-significant byte of the latches also clears the T1 interrupt flag (IFR6).

In the *one-shot mode,* Timer 1 sets the T1 interrupt flag (IFR6) after counting a specified number of ϕ2-system clock pulses. If ACR7 is set to logic 1, PB7 will output a low-level signal for the duration of the count-down. To perform a one-shot operation with Timer 1, your program must:

1. Write to the auxiliary control register (select code 1011_2), with ACR6 = 0 and ACR7 = 0 (PB7 disabled), or ACR7 = 1 (PB7 enabled).
2. Write the LSBY of the interval-count value into the Timer 1 latch low byte (select code 0100_2).
3. Write the MSBY of the interval-count value into the Timer 1 latch high byte (select code 0101_2).
4. Clear the T1 interrupt flag (IFR6) upon completion of the interval.

Example 8-11 presents a routine that generates a 1-millisecond (i.e. 1000-microsecond) time interval, using the one-shot mode of Timer 1. The routine in Example 8-11 is essentially the Timer 1 equivalent of the 1-millisecond Timer 2 routine given in Example 8-9. There is one minor difference, however, in that Timer 1 has a 1½-cycle "overhead" on the time interval, so that *the count value loaded into the timer must be two counts less than the interval desired.* That is why $03E6 (rather than $03E8) was loaded into Timer 1 in Example 8-11.

Example 8-11: A 1-Millisecond Time Interval Using Timer 1

```
;THIS ROUTINE SETS THE T1 INTERRUPT FLAG AT THE END OF A ONE-
;MILLISECOND TIME INTERVAL, WITH NO OUTPUT TO PB7.
        LDA   #00    ;SET T1 ONE-SHOT MODE, WITH NO PB7
        STA   $A00B
        LDA   #$E6   ;WRITE COUNT LSBY
        STA   $A004
        LDA   #$03   ;WRITE COUNT MSBY AND START TIMER
        STA   $A005
        LDA   #$40   ;SET T1 INTERRUPT MASK
CHKT1   BIT   $A00D  ;HAS T1 COUNTED DOWN?
        BEQ   CHKT1  ;NO. CHECK AGAIN
        LDA   $A004  ;YES. CLEAR T1 INTERRUPT FLAG
```

In the *free-running mode,* Timer 1 counts down continuously, reloading the counter with the latch values each time the counter has decremented to zero. Further, the T1 interrupt flag (IFR6) will be set at the end of each count-down cycle. If ACR7 is set to logic 1, the level on PB7 will be inverted at the beginning of each interval (it will go low when the first interval starts). In the free-running mode, timing intervals are usually generated with the interrupts enabled to save the program from polling the T1 interrupt

flag throughout the duration of the time interval. To perform a free-running operation with Timer 1, the program must:

1. Enable the Timer 1 interrupt by writing to the interrupt enable register (select code 1110_2), with IER6 and IER7 both set to logic 1.
2. Write to the auxiliary control register (select code 1011_2), with ACR6 = 1 and ACR7 = 0 (PB7 disabled) or ACR7 = 1 (PB7 enabled).
3. Write the LSBY of the interval count value into the Timer 1 latch low byte (select code 0100_2).
4. Write the MSBY of the interval count value into the Timer 1 latch high byte (select code 0101_2).

Step 4 initiates the timing operation. At the end of each time interval, an $\overline{\text{IRQ}}$ will be generated with the T1 interrupt flag (IFR6) set. If system interrupt requests are enabled, the interrupt polling program (Example 7-1) of the 6502 microprocessor identifies the VIA as the interrupting device, and then determines which of the seven VIA interrupt conditions activated $\overline{\text{IRQ}}$ (Example 8-8). In the VIA interrupt polling sequence (Example 8-8), a T1 interrupt produces a jump to a Timer 1 interrupt service routine labeled ISRT1 (Interrupt Service Routine for Timer 1). What operations should the ISRT1 routine perform? Since the free-running mode of the VIA will generate continuous time intervals until halted, the ISRT1 must count the time intervals. If the interval times are to change at some point, the ISRT1 routine must also then load the new count values into the latches at the proper time.

Example 8-12 shows a routine that generates a 6-millisecond time interval, in which time the output of PB7 is invented 6 times, once each millisecond, to produce a square-wave output. Example 8-12 also includes the interrupt service routine that is called 6 times in the course of this routine. All the interrupt service routine does is clear the T1 interrupt flag (IFR6) and decrement a counter (in location $40). When the counter has decremented to zero, the timer is disabled by clearing the T1 interrupt enable (IER6) and setting PB7 as an input.

A 24-HOUR CLOCK FOR THE AIM 65

The free-running mode of Timer 1 gives this timer potential value as a time-of-day clock. Let us take a brief look at the operations that the 24-hour clock must perform, and then discuss a program that will implement the clock.

Example 8-12: A 6-Millisecond Time Interval With 1-Millisecond Pulses on PB7

```
;THIS ROUTINE GENERATES A SIX-MILLISECOND TIME INTERVAL IN WHICH
;PB7 INVERTS EACH MILLISECOND. LOCATION $40 IS USED TO HOLD A
;TIME INTERVAL COUNT.
        LDA   #$80     ;MAKE PB7 AN OUTPUT
        STA   $A002
        LDA   #$CO     ;SET T1 FREE-RUNNING MODE, WITH PB7 OUTPUT
        STA   $A00B
        STA   $A00E    ;ENABLE TIMER 1 INTERRUPT
        LDA   #$E6     ;WRITE COUNT LSBY
        STA   $A004
        LDA   #$03     ;WRITE COUNT MSBY AND START TIMER
        STA   $A005
        LDA   #06      ;INITIALIZE TIME INTERVAL COUNT = 6
        STA   $40
        CLI            ;ENABLE INTERRUPTS IN 6502
         .
         .
         .
;FOLLOWING IS THE TIMER 1 INTERRUPT SERVICE ROUTINE.
ISRT1   LDA   $A004    ;CLEAR T1 INTERRUPT FLAG
        DEC   $40      ;DECREMENT INTERVAL COUNT
        BNE   CNTNZ    ;SIX MILLISECONDS COUNTED?
        LDA   #$40     ;YES. DISABLE TIMER 1 INTERRUPT
        STA   $A00E
        LDA   #00      ;  AND DISABLE OUTPUT ON PB7
        STA   $A002
CNTNZ   RTI            ;RETURN
```

Basically, a 24-hour clock must maintain separate counts for seconds, minutes, and hours. At the end of each 60-second time interval, it must clear the seconds count and increment the minutes count. Similarly, at the end of each 60-minute time interval, it must clear the minutes count and increment the hours count. And at the end of each 24-hour time interval, it must clear the hours count.

As mentioned, the Timer 1 can be used to generate the time intervals for the 24-hour clock. You will recall that Timer 1 bases its time interval on the cycle count that is loaded into it, and that the cycle count must have a value of two counts less than the required time interval. Since Timer 1 contains a 16-bit counter, it can generate time intervals of as little as three cycles (with a count value of $0001), and as much as 65,537 cycles (with a count value of $FFFF). With a 1-MHz clock, such as the AIM 65 has, each cycle is equivalent to a 1-microsecond time interval. This means that Timer 1 will not be able to generate a 1-second time interval directly, but must generate this interval with multiple count-downs. For this example, we will use the free-running mode of Timer 1 to continuously generate 50,000-microsecond (0.05-second) time intervals,

and to increment the "seconds" count each time the Timer has counted-down *20 times*. A 50,000-microsecond time interval requires a count of $C34E to be loaded into Timer 1. At this point, we can draw a flowchart of the required operation for a 24-hour clock; it is shown in Fig. 8-10.

The flowchart shown in Fig. 8-10 defines the sequence of operations that should occur when Timer 1 times out. Clearly, the 24-hour clock program must perform two additional functions.

1. It must initialize the Timer 1 of the VIA, and start the timing operation.
2. It must output the clock value to the AIM 65 display.

Example 8-13 shows a 24-hour clock program that performs these three functions. The main program, CLK24, performs the required initialization; it includes the display sequence, DPYCLK. Example 8-13 also includes the 24-clock interrupt service routine, CLKINT, which was flowcharted in Fig. 8-10. Let us discuss the details of these three functions.

Example 8-13 assumes that the initial value of the clock has been stored into locations $21 (seconds), $22 (minutes), and $23 (hours) in BCD form. For instance, an initial time of 08:13:36 would be represented as $36 in location $21, $13 in location $22 and $08 in location $23. The main program, CLK24, begins by disabling IRQ interrupts so that the clock can be initialized. The next four instructions store the address of the 24-hour clock interrupt service routine, CLKINT, into the system IRQ vector locations, $A404 and $A405. (Note that we have employed assembler operators < and > here, to truncate the address of CLKINT to its LSBY and MSBY values, respectively. AIM 65 owners who are not using the assembler can replace these LDA instruction operands with the actual address bytes of the CLKINT routine.) Next, Timer 1 is loaded with its count value, $C34E. While Timer 1 is counting, instructions LDA #20 and STA $20 establish an 0.05-second interrupt in counter location $20. With initialization thus completed, the IRQ disable is removed (CLI) and BRK causes a return to the AIM 65 Monitor. The Monitor will return two bytes past the BRK instruction, so a NOP is inserted as a "spacer" between the main program and the output routine DPYCLK.

The display output routine DSPCLK uses two AIM 65 Monitor subroutines, CRLF and NUMA. The first subroutine, CRLF (entry address $E9F0), outputs a CR (Carriage Return) character to the active output device. Unless you have altered the OUTFLG parameter of the AIM 65, location $A413, the display will be the active output device. When the display receives a CR character, its pointer is reset to select position 0, the leftmost character position. The

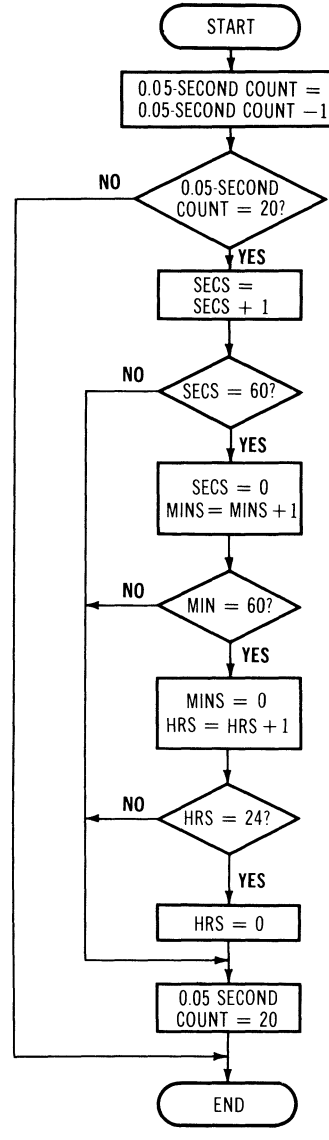


Fig. 8-10. Flowchart for a 24-hour clock program.

NUMA subroutine converts two hex numbers in the accumulator from binary to ASCII code, and outputs them to the active output device, the most-significant digit first. In the DSPCLK routine, NUMA is called three times—to output the hours count, the minutes count, and the seconds count, in that order. The DPYCLK is

Example 8-13: A 24-Hour Clock for the AIM 65

```
;THIS SUBROUTINE DISPLAYS A 24-HOUR CLOCK ON THE AIM 65. UPON
;ENTRY, THE INITIAL BCD VALUES OF SECONDS, MINUTES, AND HOURS MUST
;BE CONTAINED IN LOCATIONS $21, $22, AND $23, RESPECTIVELY. THE
;SUBROUTINE ALSO USES LOCATION $20 TO HOLD A 1/20-SECOND COUNT.

CLK24   SEI             ;DISABLE IRQ INTERRUPTS
        LDA  #<CLKINT   ;CLOCK INTERRUPT VECTOR = CLKINT
        STA  $A404
        LDA  #>CLKINT
        STA  $A405
        LDA  #$C0       ;SET T1 FREE RUNNING MODE
        STA  $A00B
        STA  $A00E      ;ENABLE TIMER 1 INTERRUPT
        LDA  #$4E       ;WRITE COUNT LSBY
        STA  $A004
        LDA  #$C3       ;WRITE COUNT MSBY AND START CLOCK
        STA  $A005
        LDA  #20        ;INTERRUPT COUNT = 20
        STA  $20
        CLI             ;ENABLE IRQ INTERRUPTS
        BRK             ;RETURN TO MONITOR
        NOP

;FOLLOWING IS THE ROUTINE THAT DISPLAYS THE 24-HOUR CLOCK.

DPYCLK  JSR  CRLF       ;RESET DISPLAY
        LDA  $23        ;LOAD HOURS COUNT
        JSR  NUMA       ;  AND OUTPUT IT TO DISPLAY
        LDA  $22        ;LOAD MINUTES COUNT
        JSR  NUMA       ;  AND OUTPUT IT TO DISPLAY
        LDA  $21        ;LOAD SECOUNDS COUNT
        JSR  NUMA       ;  AND OUTPUT IT TO DISPLAY
        JMP  DPYCLK     ;REFRESH THE DISPLAY

;FOLLOWING IS THE 24-HOUR CLOCK INTERRUPT SERVICE ROUTINE.

CLKINT  PHA             ;SAVE ACCUMULATOR ON STACK
        DEC  $20        ;20 INTERRUPTS YET?
        BNE  INTDUN     ;NO. INTERRUPT DONE
        SED             ;YES. SET DECIMAL MODE
        CLC             ;INCREMENT SECONDS COUNT
        LDA  $21
        ADC  #01
        STA  $21
        CMP  #$60       ;SECONDS = 60?
        BCC  RES20      ;NO. REINITIALIZE INTERRUPT COUNTER
        LDA  #00        ;YES. CLEAR SECONDS COUNT
        STA  $21
        ADC  $22        ;INCREMENT MINUTES COUNT
        STA  $22
        CMP  #$60       ;MINUTES = 60?
        BCC  RES20      ;NO. REINITIALIZE INTERRUPT COUNTER
        LDA  #00        ;YES. CLEAR MINUTES COUNT
        STA  $22
        ADC  $23        ;INCREMENT HOURS COUNT
        STA  $23
```

```
        CMP   #$24      ;HOURS = 24?
        BCC   RES20     ;NO. REINITIALIZE INTERRUPT COUNTER
        LDA   #00       ;YES. CLEAR HOURS COUNT
        STA   $23
RES20   LDA   #20       ;INTERRUPT COUNTER = 20
        STA   $20
INTDUN  LDA   $A004     ;CLEAR T1 INTERRUPT FLAG
        CLD             ;CLEAR DECIMAL MODE
        PLA             ;RESTORE ACCUMULATOR
        RTI
```

a closed loop, and will execute continuously until an interrupt occurs. An interrupt will occur each time Timer 1 times out (i.e., every 0.05 second), and will cause control to be transferred to the interrupt service routine CLKINT.

Readers who have studied Fig. 8-10 long enough to understand what is being done will have no difficulty understanding the 24-hour clock interrupt service routine CLKINT. In fact, CLKINT includes only a few features that were not included in the flowchart.

1. It saves the contents of the accumulator at the beginning of the routine (PHA) and restores these contents at the end of the routine (PLA).
2. It updates the seconds, minutes, and hours counts using decimal arithmetic, by enclosing all calculations with an SED instruction at the beginning and a CLD instruction at the end.
3. It clears the T1 interrupt flag at the end of interrupt processing, with a "dummy" Read instruction, LDA $A004.

Like the time-delay subroutines given in Chapter 3, the accuracy of the 24-hour clock depends on the accuracy of the crystal on your particular AIM 65. Although crystals on recent AIM 65 units have been found to be accurate to ±0.00015, you should check your clock program against a reliable source, such as Station WWV, and modify the LSBY count value contents of Timer 1 as necessary.

VIA SHIFT REGISTER

The *shift register* of the VIA can be used to convert data between serial and parallel forms. Bits 2, 3, and 4 of the auxiliary control register (ACR) control the operation of the shift register, as shown in Table 8-14. As you can see from this table, there is one disable mode, three input (serial-to-parallel) modes, and four output (parallel-to-serial) modes. The shift register occupies one location in memory; this is the location whose address is represented by select code 1010_2.

Table 8-14. Shift Register Control

ACR			Mode
Bit 4	Bit 3	Bit 2	
0	0	0	Shift register disabled.
0	0	1	Shift in, under control of Timer 2.
0	1	0	Shift in, under control of $\phi 2$.
0	1	1	Shift in, under control of external clock on CB1.
1	0	0	Free-running output, at rate determined by Timer 2.
1	0	1	Shift out, under control of Timer 2.
1	1	0	Shift out, under control of $\phi 2$.
1	1	1	Shift out, under control of external clock on CB1.

The shift register is rarely used because the serial/parallel conversion function is more readily available using a UART (Universal Asynchronous Receiver-Transmitter) device. Readers who plan to use the shift register should refer to the manufacturers' literature.

REFERENCES

1. Hardware interfacing information for the 6520, 6522, 6530, and 6532 integrated circuits is contained in the *R6500 Microcomputer System Hardware Manual,* Rockwell International, Anaheim, CA, 1978.

2. For a description of 6530 RRIOT programming, see Chapter 11 of the *R6500 Microcomputer System Programming Manual,* Rockwell International, Anaheim,, CA, 1978.

9

Microcomputer Input/Output

In this chapter, we will explore the area of *interfacing* external (or peripheral) devices to the 6502 microprocessor. In previous chapters, it has been assumed that these peripheral devices were properly interfaced to the 6502 microprocessor, and we concentrated on the design and development of the software that is needed to communicate with them. At this point, it is pertinent to explore the design and implementation of the interfacing hardware. Hardware is a broad term that describes the electrical components, resistors, gates, drivers, and latches that are used to electrically connect a peripheral device to the buses of the microcomputer. A discussion of interfacing may seem unusual for a book on assembly language programming, but we believe that assembly language programmers are not only interested in using software to communicate with peripheral devices, but that they are also interested in learning how peripheral devices are actually interfaced to the microcomputer. All of the interfaces in this chapter are implemented with a 6522 VIA (and, in most cases, the addresses of the VIA are those of the AIM 65's user-dedicated VIA). However, similar interfaces could be implemented with a 6520 PIA, or one of the other I/O devices in the 6500 family.

THE 6502 MICROPROCESSOR AND SIMPLE I/O DEVICES

Let us begin this discussion of interfacing by looking at how two very simple I/O devices can be interfaced to the 6502 microprocessor. Perhaps the simplest input device is the two-position spdt toggle switch and the simplest output device is the LED (light-emitting diode). For illustrative purposes, let us develop the circuit that will interface eight spdt switches to the input port

of a 6522 VIA and eight LEDs to the output port of the same VIA. Our program should simply read the switch settings into the accumulator of the 6502 microprocessor and display the setting of each switch on its associated LED. The LED must be on if its corresponding switch is on.

Spdt Switch Interface

Fig. 9-1 illustrates the connection of a single spdt toggle switch to the VIA. With the connection shown, the switch presents a logic 1 (+5 V) to its VIA input port pin when the switch is in the N.O. (normally open) position. It presents a logic 0 (ground) to the same pin when the switch is in the N.C. (normally closed) position.

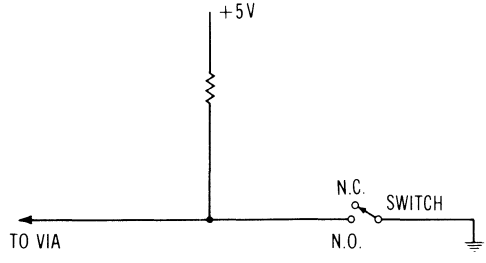


Fig. 9-1. An spdt toggle switch.

Fig. 9-2 shows a group of eight spdt toggle switches connected to Port A of a VIA. Which instructions would be used to read the switch settings into the accumulator? Only three instructions are needed, as we know from our "simple VIA input routine" that was given in Chapter 8 (Example 8-2). These instructions are:

```
LDA  #00    ;MAKE PORT A INPUTS
STA  $A003
LDA  $A001  ;INPUT SWITCH SETTINGS
```

Note that, as in Chapter 8, we are assuming that the VIA occupies addresses $A000 through $A00F, the locations occupied by the user-dedicated VIA on the AIM 65.

LED Display Interface

Fig. 9-3 shows the circuitry necessary to interface a single LED to a 6522 VIA. With this type of connection, the LED turns on when it receives a logic 0 signal from the VIA. The LED is brightest when it operates from currents of between 10 to 50 mA. Since the VIA can (at best) sink a current of 1.6 mA, it cannot drive LEDs directly and needs a supporting transistor-driven circuit and some current-limiting resistors.

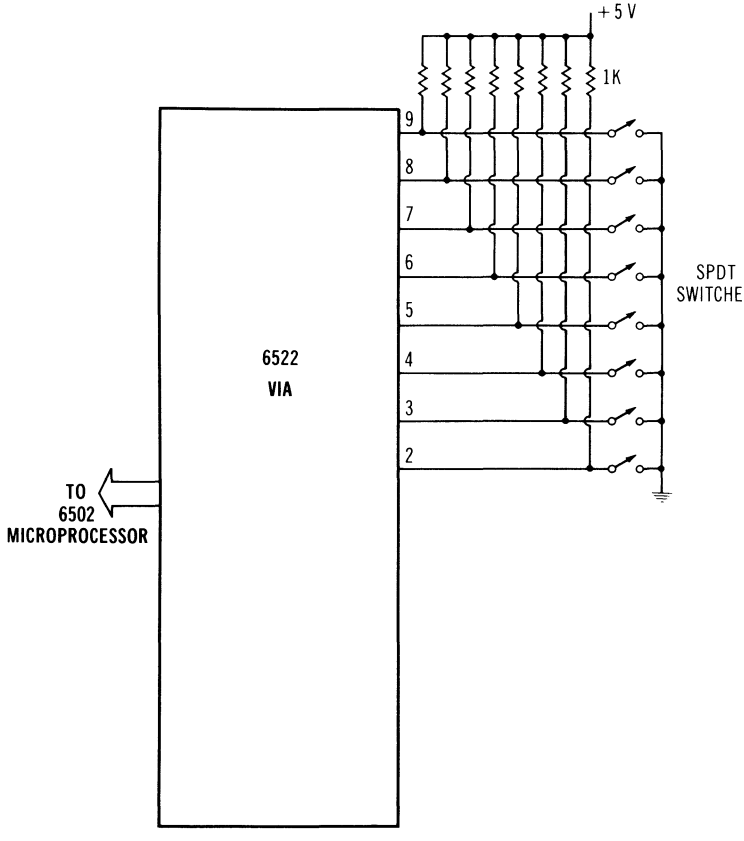


Fig. 9-2. Interfacing eight toggle switches to a VIA.

Fig. 9-4 is an expanded version of Fig. 9-2, showing a group of eight LEDs and their driver circuits that are connected to Port B of the VIA. With the interface now defined, we can write a simple instruction routine that Reads switch data from Port A and then outputs it to the LEDs on Port B. Example 9-1 shows such a routine. This routine simply clears the peripheral control register, initializes Port A as an input port and Port B as an output port, and then loads

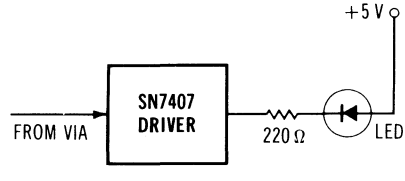


Fig. 9-3. Basic LED circuitry.

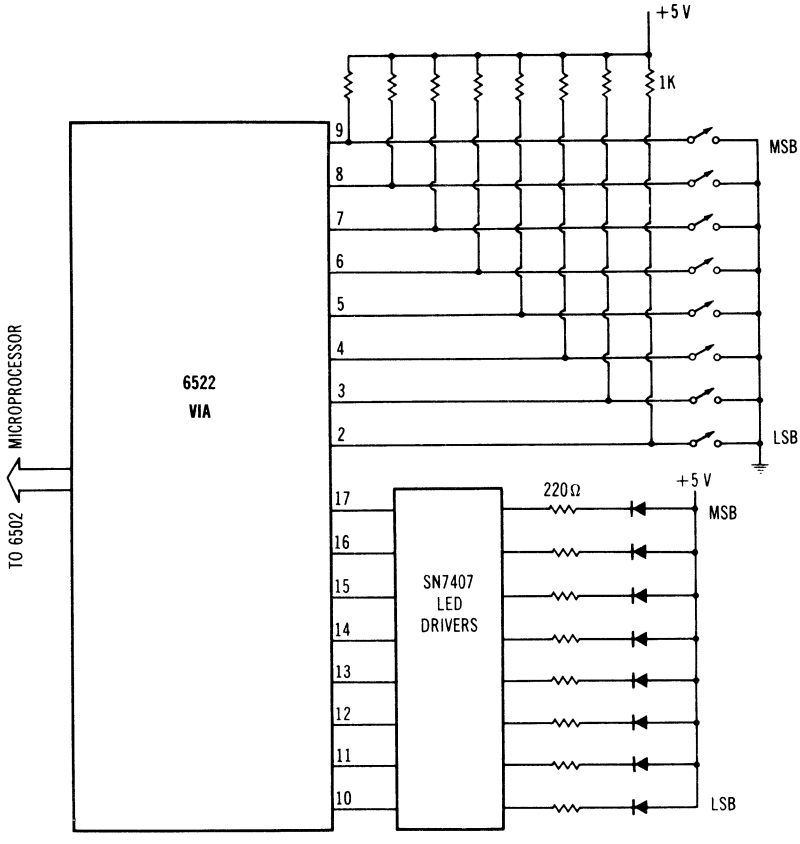


Fig. 9-4. Interfacing eight toggle switches and LEDs to a VIA.

the switch data into the accumulator. Since the LEDs operate with active-low logic (a 0 turns them on), the switch data is saved in the X register, and the accumulator is then complemented (EOR #$FF). The result is sent to the LEDs, and the switch data is returned to the accumulator for subsequent processing. In some applications, the inversion is performed by the drivers, which saves a few instructions.

ANOTHER SIMPLE INPUT DEVICE

We have just discussed one of the two main types of switches that are found in computer systems—the two-position spdt toggle switch. The other main type of switch is the *spst push-button switch.* Push-button switches are commonly used to start or stop a device, to manually initiate a system reset, to enter data, and many other

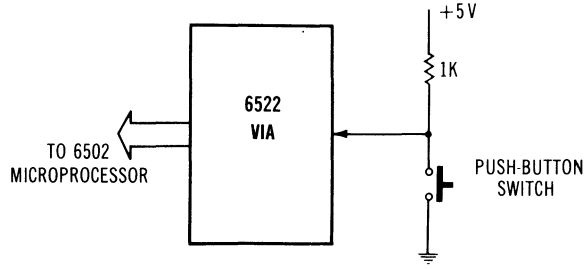


Fig. 9-5. Interfacing a push-button switch to a VIA.

Example 9-1: Read Switch Settings and Display Them on LEDs

```
;THIS ROUTINE READS THE CURRENT SETTINGS OF EIGHT SWITCHES THAT
;ARE CONNECTED TO PORT A OF A VIA, AND SENDS THEM TO EIGHT LED'S
;THAT ARE CONNECTED TO PORT B OF THE VIA.
LDA  #00    ;CLEAR PERIPHERAL CONTROL REGISTER
STA  $A00C
STA  $A003  ;MAKE PORT A INPUTS
LDA  #$FF   ;MAKE PORT B OUTPUTS
STA  $A002
LDA  $A001  ;READ SWITCH DATA
TAX         ;  AND SAVE THEM IN X
EOR  #$FF   ;COMPLEMENT SWITCH DATA
STA  $A000  ;  AND DISPLAY ON LED'S
TXA         ;RETURN SWITCH DATA TO ACCUMULATOR
```

similar functions. The keys on keyboards and the keypads of teletypewriters, terminals, and single board microcomputers are also push-button switches.

Fig. 9-5 shows how a single push-button switch can be interfaced to one of the pins on a VIA input port. As you can see from this drawing, the button presents a logic 0 to the VIA if it is pressed (closed) and a logic 1 if it is not pressed (open). Example 9-2 shows a routine that first checks whether or not the button has been

Example 9-2: Check for Closure of Push-Button Switch

```
;THIS ROUTINE CHECKS TO SEE WHETHER A PUSH-BUTTON SWITCH ATTACHED TO
;PIN PA2 OF A VIA IS PUSHED. IF IT IS, LOCATION $40 IS SET
;TO A ONE.
        LDA  #00    ;CLEAR PERIPHERAL CONTROL REGISTER
        STA  $A00C
        STA  $A003  ;MAKE PORT A INPUTS
        STA  $40    ;BUTTON FLAG = 0
        LDA  $A001  ;READ PORT A
        AND  #04    ;IS BUTTON PUSHED (PA2 = 0)?
        BNE  DONE   ;NO. DONE.
        INC  $40    ;YES. SET BUTTON FLAG = 1
DONE    .
        .
        .
```

pressed, and then sets a flag in memory location $40 to a 1 (pressed) or a 0 (not pressed), based on the test.

The Key Bounce Problem

If your program is simply checking whether a button is pushed (closed) or not pushed (open), the software is fairly straightforward, as you can see from Example 9-2. However, if you are interested in the counting of several separate key closures, most switches, including push-button switches will present a special problem because they cause electrical *bounce*. We all know that if a rubber ball is dropped onto a hard floor, it will bounce a few times, at decreasing heights, before finally coming to rest. Similarly, when you press a push-button switch or a key, you are causing metal contacts to strike together, and they will bounce off each other a few times before settling into their final positions. The bouncing action will cause a series of logic 1s and 0s to be generated, which a microprocessor program may interpret as a sequence of separate key closures, rather than just a single closure. One of the most common ways of avoiding this problem is to insert a time delay into the software that will prevent the program from testing for a second key closure until the first key has been released. The key-bounce period varies from one type of push-button switch to another, but it will probably not exceed 10 milliseconds.

Since the average human key-push period last between $\frac{1}{10}$ second and 1 second, the 10-millisecond key-debouncing time delay will not prevent the program from interpreting the present key closure as a new closure. Clearly, the program must include not only time-delay software, but it must also detect that the key has been released before testing for a new key closure.

Let us now demonstrate how these two techniques, the 10-millisecond debounce time delay and the waiting for key release, could be used in a real application. Assume that a system has a VIA to which two push-button switches are connected. Your task is to write a short program that counts the key closures on push-button switch No. 1 (connected to PA2) until push-button switch No. 2 (connected to PA7) is pushed. Example 9-3 shows a program that will do the job. It uses Timer 1 of the VIA to generate a 10-millisecond time delay, through a subroutine labeled DLY10 (derived from Example 8-11), and it maintains the closure count in memory location $40. Note that push-button switch No. 2 is not debounced, since we are only concerned with its first closure, and do not care whether it is held down or pushed more than once. Incidentally, we could just as easily have used a time-delay subroutine, such as the one given in Example 3-4, to provide the debouncing delay,

Example 9-3: Counting Push-Button Switch Closures, With Debouncing

```
;THIS ROUTINE COUNTS CLOSURES ON PUSH-BUTTON SWITCH NO. 1 UNTIL PUSH-
;BUTTON SWITCH NO. 2 IS PUSHED. PUSH-BUTTON SWITCH NO. 1 IS CONNECTED
;TO VIA PIN PA2, PUSH-BUTTON SWITCH NO. 2 IS CONNECTED TO VIA PIN PA7.
;THE CLOSURE COUNT IS HELD IN MEMORY LOCATION $40.

        LDA   #00     ;CLEAR PERIPHERAL CONTROL REGISTER
        STA   $A00C
        STA   $A003   ;MAKE PORT A INPUTS
        STA   $40     ;CLOSURE COUNT = 0
CHKBTN  LDA   $A001   ;READ PORT A
        BPL   DONE    ;DONE IF BUTTON NO. 2 IS PUSHED (PA7 = 0)
        AND   #04     ;IS BUTTON NO. 1 PUSHED (PA2 = 0)?
        BNE   CHKBTN  ;NO. WAIT UNTIL IT IS.
        INC   $40     ;YES. INCREMENT CLOSURE COUNT.
        JSR   DLY10   ;WAIT 10 MILLISECONDS TO DEBOUNCE
CHKREL  LDA   $A001   ;READ PORT A AGAIN
        AND   #04     ;IS BUTTON NO. 1 STILL CLOSED?
        BEQ   CHKREL  ;YES. WAIT FOR RELEASE
        JSR   DLY10   ;NO. DEBOUNCE THE KEY OPENING
        JMP   CHKBTN  ;  AND WAIT FOR NEXT CLOSURE
DONE    .
        .
        .

;THE FOLLOWING SUBROUTINE USES TIMER 1 TO GENERATE A 10-MILLISECOND
;DEBOUNCE TIME DELAY, BY WRITING 10,000 ($2710) INTO THE COUNTERS.
DLY10   LDA   #00     ;SET T1 ONE-SHOT MODE, WITH NO PB7
        STA   $A00B
        LDA   #$10    ;WRITE COUNT LSBY
        STA   $A004
        LDA   #$27    ;WRITE COUNT MSBY AND START TIMER
        STA   $A005
        LDA   #$40    ;SELECT T1 INTERRUPT MASK
CHKT1   BIT   $A00D   ;HAS T1 COUNTED DOWN?
        BEQ   CHKT1   ;NO. WAIT UNTIL IT HAS
        LDA   $A004   ;YES. CLEAR T1 INTERRUPT FLAG
        RTS           ;  AND RETURN
```

but chose instead to use Timer 1 to further demonstrate the versatility of the 6522 VIA.

THE 6502 MICROPROCESSOR AND KEYBOARDS

Virtually every microcomputer system includes some type of keyboard. Keyboards vary in size, shape, and number of keys, but whatever their physical characteristics, every keyboard can be looked upon as simply a collection of push-button switches in which there is a unique code associated with each button (key). As a programmer, your task will be to detect when a key has been pressed, and then identify *which* key is pressed so that its code can be entered into the microprocessor. Some keyboards include the

internal circuitry to identify the key that was pressed and to generate its code; these are *encoded* keyboards. Other keyboards produce only electrical signals that must be processed entirely by software; these are *unencoded* keyboards. In choosing a keyboard for a microcomputer system, you must consider the trade-off between the convenience of the encoded keyboard and the lower cost of the unencoded keyboard. We will discuss both types of keyboard, unencoded and encoded, and the software that is required to service them.

UNENCODED KEYBOARDS

Realizing that keyboards are simply collections of push-button switches, it is apparent that one way of interfacing a keyboard to a general-purpose I/O device (such as a PIA or a VIA) is to connect each push-button switch to an input line of an I/O device port. This technique would produce a circuit that resembles the circuit shown in Fig. 9-2, in which we interfaced eight toggle switches to a VIA. Moreover, this technique permits a push-button switch to be identified quickly and easily because it involves merely finding out which of the input port lines is in the logic 0 state. With 16 port lines, a VIA can be wired to as many as 16 keys. What happens if we use a keyboard that has more than 16 keys? With the technique just described, we would need one additional VIA for a keyboard with 17 to 32 keys, two additional VIAs for a keyboard with 33 to 48 keys, and so on. Although VIAs are fairly inexpensive, there is a much more cost-efficient way to interface unencoded keyboards. Each of the keys can be wired in a *matrix*, as shown in Fig. 9-6.

As you can see in Fig. 9-6, a 3×3 keyboard requires only six VIA I/O port lines, three lines of an output port and three lines of an input port. Similar circuits could be constructed for a keyboard of size $n \times m$, where n represents the number of rows and m represents the number of columns. Note that a 4×6 keyboard requires 4 rows (output port lines) and 6 columns (input port lines), and an 8×8 keyboard requires 8 rows and 8 columns. This latter size, 8×8, is the largest keyboard configuration that can be interfaced to the 6502 microprocessor using a single VIA, since the VIA has 16 I/O lines.

A program can determine whether any key in a matrix has been pressed by grounding all rows simultaneously (logic zeroes in the output register) and by examining the column lines of the input register. If any key in a column has been pressed, its bit position in the input register will be read as a logic 0. Further, if the program determines that one of the keys in the matrix has been pressed,

it can find out *which* key has been pressed by performing a "row scan" operation. A row scan is a polling technique in which the program begins by grounding only Row 0 and examining the column lines. If any key was pressed in that row, its column line will also be grounded and the corresponding bit position of the input register will be read as a logic 0. If no column line is grounded, the program proceeds to Row 1 and repeats the operation, and then proceeds to Row 2, if necessary.

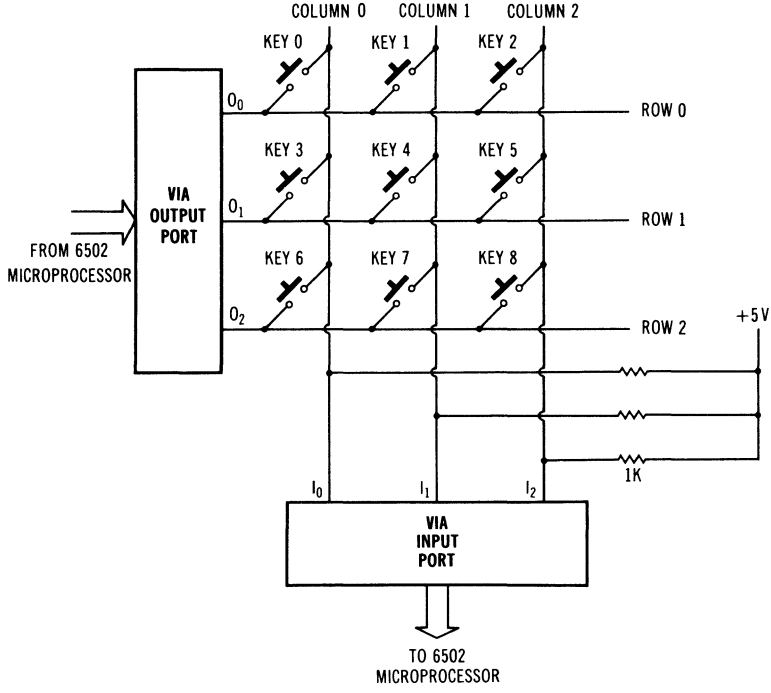


Fig. 9-6. Matrix connections for an unencoded keyboard.

The program we have just described must perform two separate functions:

(1) It must determine whether any key in the matrix was pressed.
(2) Upon sensing a key closure, it must identify the key that was pressed.

The flowchart in Fig. 9-7 shows the sequence of operations for a program that will perform both functions. Let us begin by looking at the instructions for Function 1, those instructions that determine whether any key was pressed. As shown in the flowchart, this por-

tion of the program must initialize the peripheral control register and the data direction registers, and then ground all rows to see if any column is inputting a logic 0. The routine given in Example 9-4 performs those operations for the circuit shown in Fig. 9-6. It

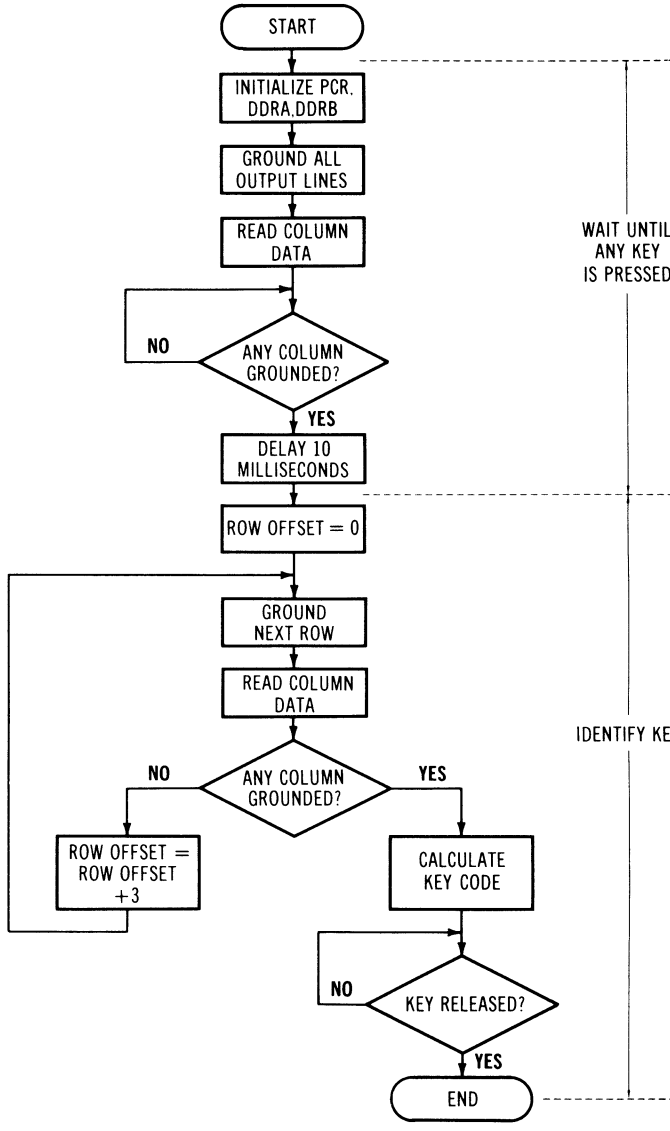


Fig. 9-7. Flowchart for processing key closures.

Example 9-4: Waiting For a Key to Be Pressed

```
;THIS ROUTINE CHECKS WHETHER ANY KEY IN A 3 X 3 UNENCODED KEY-
;BOARD HAS BEEN PRESSED. THE KEYBOARD ROW CONNECTIONS ARE
;ASSUMED TO BE INTERFACED TO PB5 (ROW 0), PB6 (ROW 1), AND PB7
;(ROW 2). THE KEYBOARD COLUMN CONNECTIONS ARE ASSUMED TO BE
;INTERFACED TO PA5 (COLUMN 0), PA6 (COLUMN 1), AND PA7 (COLUMN 2).
        LDA   #$FF     ;MAKE PORT B OUTPUTS
        STA   $A002
        LDA   #00      ;CLEAR PERIPHERAL CONTROL REGISTER
        STA   $A00C
        STA   $A003    ;MAKE PORT A INPUTS
        STA   $A000    ;GROUND ALL OUTPUTS
CHK4GD  LDA   $A001    ;GET COLUMN DATA
        CMP   #$E0     ;IS ANY COLUMN GROUNDED?
        BCS   CHK4GD   ;NO. WAIT UNTIL ONE IS
        JSR   DLY10    ;YES. DEBOUNCE THE KEY
         .
         .             (Identify the key)
         .
```

assumes that the rows are connected to Bits 5, 6, and 7 of Port B and that the columns are connected to Bits 5, 6, and 7 of Port A.

After configuring the ports (Port B is an output port, Port A is an input port) and clearing the peripheral control register, the routine in Example 9-4 grounds all three rows of the keyboard by writing logic zeroes into output register B. It then checks whether the column data is less than $E0, since that is the *only* condition under which Bit 5, 6, or 7 is a logic 0. If none of these three bits is a logic 0, the program waits until a logic 0 is sensed; otherwise, it calls the 10-millisecond time-delay subroutine DLY10 that was included in Example 9-3.

Once a key closure has been sensed and debounced, the next task is to identify the key through row scanning. Example 9-5 constitutes the complete sequence of instructions for identifying a key closure. The first 10 instructions in this routine are the instructions from Example 9-4 (repeated here for your convenience), and are followed by the row-scanning sequence. Before conducting the row scanning, the routine establishes a *row offset* in memory location $40. This offset is used to hold a count of keys in the rows that have been previously scanned, and is initially zero. After initializing the row offset, the routine grounds the outputs on Row 0 and reads the column data for that row. If any column contains a logic 0, the BCC IDKEY test will cause the 6502 microprocessor to branch to the key identification routine, IDKEY. Otherwise, the row offset is updated by three (ADC #*02* is used because the Carry must be set to a logic 1 at this point). Row 1 is tested in a similar manner. If no key in Row 1 is pressed, we assume that Row 2 contains the key that is pressed.

The key identification subroutine, IDKEY, examines the state of the three column lines one-by-one, beginning with Column 2, by left-shifting the input data in the accumulator. When the logic 0 (grounded) column is found, its column number is added to the row offset (in location $40) to produce the key code.

ENCODED KEYBOARDS

The software for encoded keyboards is much simpler than for unencoded keyboards, because encoded keyboards provide a unique code for each key that is pressed. Encoded keyboards have internal electronics that perform all of the scanning and key identification procedures that unencoded keyboards require you to perform with software. Encoded keyboards typically provide key debouncing and rollover (accepting only one key closure when two or more keys are pressed simultaneously), and often include ROMs or PROMs with look-up tables that generate a code in ASCII, EBCDIC, or some other coding format.

Encoded keyboards also provide a *data ready* strobe with each key code transferred, allowing the design of a simple keyboard interface with a single VIA, as shown in Fig. 9-8. An active transition of the keyboard strobe on CA1 will cause the CA1 interrupt flag of the interrupt flag register to be set (IFR1), with the polarity of this transition specified by the CA1 control bit of the peripheral control register (PCR0). Therefore, the keyboard service routine must simply wait for IFR1 to be set, and then read the key code into the accumulator. You will recall from Chapter 8 that reading the data from input register A automatically clears the CA1 interrupt flag. Example 9-6 presents a simple routine to communicate with an encoded keyboard.

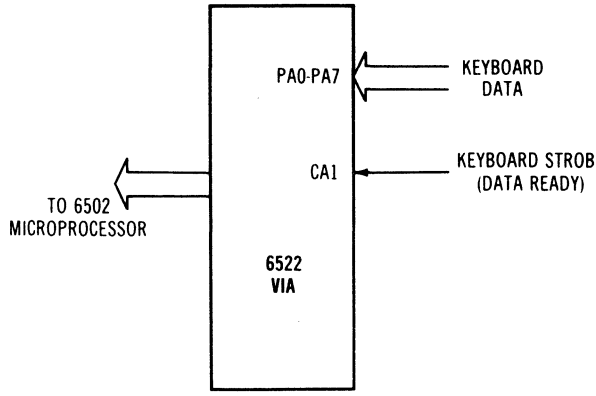


Fig. 9-8. Encoded keyboard interface with a VIA.

Example 9-5: Routine to Identify a Key

```
        LDA   #$FF     ;MAKE PORT B OUTPUTS
        STA   $A002
        LDA   #00      ;CLEAR PERIPHERAL CONTROL REGISTER
        STA   $A00C
        STA   $A003    ;MAKE PORT A INPUTS
        STA   $A000    ;GROUND ALL OUTPUTS
CHK4GD  LDA   $A001    ;GET COLUMN DATA
        CMP   #$E0     ;IS ANY COLUMN GROUNDED?
        BCS   CHK4GD   ;NO. WAIT UNTIL ONE IS
        JSR   DLY10    ;YES. DEBOUNCE THE KEY
;FOLLOWING ARE THE ROW-SCANNING INSTRUCTIONS.
        LDA   #00      ;ROW OFFSET = 0
        STA   $40
        LDA   #$C0     ;GROUND ROW 0
        STA   $A000
        LDA   $A001    ;GET COLUMN DATA FOR ROW 0
        CMP   #$E0     ;IS ANY COLUMN GROUNDED?
        BCC   IDKEY    ;YES. IDENTIFY KEY PRESSED
        LDA   $40      ;NO. ADD 3 TO ROW OFFSET
        ADC   #02
        STA   $40
        LDA   #$A0     ;GROUND ROW 1
        STA   $A000
        LDA   $A001    ;GET COLUMN DATA FOR ROW 1
        CMP   #$E0     ;IS ANY COLUMN GROUNDED?
        BCC   IDKEY    ;YES. IDENTIFY KEY PRESSED
        ASL   $40      ;NO. ROW OFFSET = 6
        LDA   #$60     ;GROUND ROW 2
        STA   $A000
        LDA   $A001    ;GET COLUMN DATA FOR ROW 2
;THE FOLLOWING INSTRUCTIONS IDENTIFY THE KEY.
IDKEY   ASL   A        ;IS KEY 2, 5, OR 8 PRESSED?
        BCS   KEY147   ;NO. CHECK KEY 1, 4, OR 7 PRESSED
        LDA   #02      ;YES. CALCULATE KEY CODE
        JMP   DONE
KEY147  ASL   A        ;IS KEY 1, 4, OR 7 PRESSED?
        BCS   KEY036   ;NO. KEY 0, 3, OR 6 IS PRESSED
        LDA   #01      ;YES. CALCULATE KEY CODE
        JMP   DONE
KEY036  LDA   #00
DONE    ADC   $40
```

INTERFACING WITH TELETYPEWRITERS

A teletypewriter is a serial I/O device; it sends and receives information as strings of logic 1s and 0s that are bounded by control bits. Whereas the basic unit of data transfer is 8 bits of information for parallel devices, the basic unit of data transfer for the teletypewriter is 11 bits of information. These 11 bits are comprised of one *start* bit (always logic 0), a 7-bit ASCII character (trans-

Example 9-6: Reading Data From an Encoded Keyboard

```
;THIS ROUTINE READS A KEY CODE FROM AN ENCODED KEYBOARD ON
;A HIGH-TO-LOW TRANSITION OF A DATA-READY STROBE ON CA1.
        LDA  #00    ;CLEAR PERIPHERAL CONTROL REGISTER
        STA  $A00C
        STA  $A003  ;MAKE PORT A INPUTS
        LDA  #02    ;SELECT CA1 INTERRUPT FLAG
CHKKB   BIT  $A002  ;IS THERE KEYBOARD DATA?
        BEQ  CHKKB  ;NO. WAIT UNTIL THERE IS
        LDA  $A001  ;YES. FETCH DATA
```

mitted LSB first), a *parity* bit and two *stop* bits (both logic 1). This format is illustrated in Fig. 9-9. Most teletypewriters transmit information at a rate of ten 11-bit characters per second, or at a *110-baud* rate. Therefore, each bit has a width of $\frac{1}{110}$ of a second, or 9.09 milliseconds.

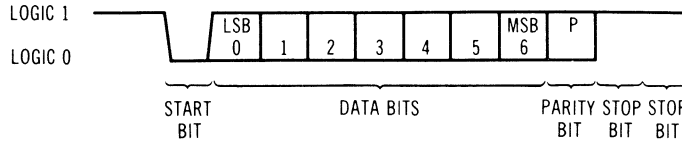


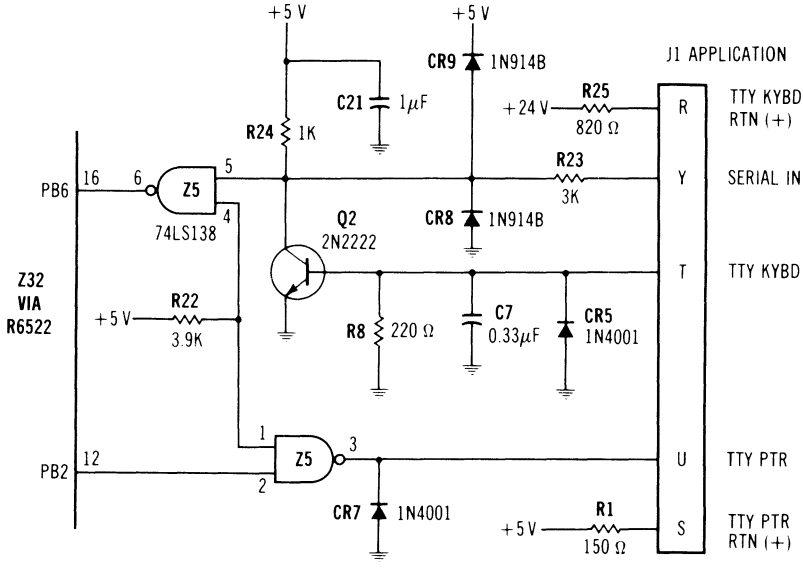
Fig. 9-9. The data format for a teletypewriter.

Hardware Interface

For purposes of discussion, let us examine an existing teletypewriter interface—the interface that is built into the AIM 65 (Fig. 9-10). This interface uses a single VIA, connected in AIM 65 socket Z32, which occupies addresses $A800 through $A80F. Teletypewriter keyboard data is received on application connector pin J1-T (TTY KYBD), buffered by transistor Q2 and associated components, and is presented to the VIA on input line PB6. The teletypewriter printer data originates on VIA output line PB2 and is output to the teletypewriter on application connector pin J1-U (TTY PTR). Application connector pin J1-Y (SERIAL IN) accommodates serial-bit streams at rates of up to 9600 baud, and can be used when the teletypewriter is not operating.

During the system initialization sequence for power-up and reset, the 6502 microprocessor must configure the VIA port lines of the teletypewriter by permanently assigning PB2 as an output and PB6 as an input. These assignments will be made by the instructions

```
LDA  #04    ;PB2 = 1, PB6 = 0
STA  $A802  ;WRITE INTO DATA DIRECTION REGISTER B
```



Courtesy Rockwell International

Fig. 9-10. An AIM 65 teletypewriter interface.

Receiving Data From a Teletypewriter

The basic task in receiving information from a teletypewriter is extracting the data from the 11-bit character that the VIA receives with each data transfer operation. Fig. 9-11 presents a flowchart for a teletypewriter receive program. Here are the steps shown in the flowchart:

1. Wait for a *start* bit (a logic 0) on the data input line.
2. Delay a 1-bit time (9.09 milliseconds), to skip the *start* bit.
3. Delay a ½-bit time (4.545 milliseconds), so that the 6502 microprocessor samples the data stream *in the middle* of the next data bit.
4. Input seven data bits (least-significant bit first), waiting 1-bit time between bits. Assemble the bits into a data byte by right-shifting input register B with each bit received.
5. Delay 1½-bit times, to skip the *parity* bit.
6. Load ASCII data byte into accumulator.

Example 9-7 shows a subroutine that uses the flowcharted algorithm. This subroutine (GETTTY) employs Timer 2 of the VIA to generate 1-bit and ½-bit time delays (with the DELAY and DEHALF subroutines, respectively) by initializing the counters with a value of $233F. This count value accounts for the inherent

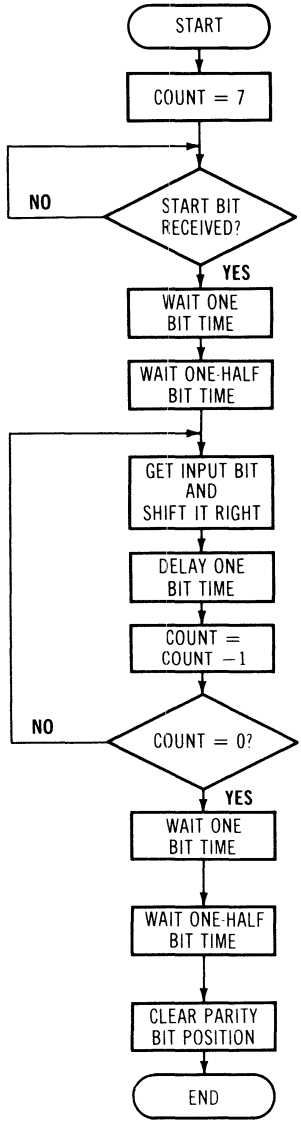


Fig. 9-11. Flowchart for teletypewriter receive operation.

time delays that are induced by the instructions of the delay subroutine and by the JSR and RTS instructions that call and return from the subroutine. Incidentally, AIM 65 owners should be interested in the fact that the GETTTY subroutine given in Example 9-7 is a slightly modified version of the GETTTY subroutine of the AIM 65 Monitor (entry address $EBDB). Likewise, the DELAY and DEHALF subroutines that are included in Example 9-7 are

Example 9-7: A Teletypewriter Receive Subroutine

```
;THIS SUBROUTINE INPUTS AN ASCII CHARACTER FROM THE TELETYPEWRITER
;INTO THE ACCUMULATOR. LOCATION $40 IS USED TO ASSEMBLE THE
;SERIAL DATA AS IT IS RECEIVED.

GETTTY  LDA   #00     ;CLEAR PERIPHERAL CONTROL REGISTER
        STA   $A80C
        LDX   #07     ;DATA BIT COUNT = 7
        STX   $40     ;CLEAR MSB OF DATA STORAGE LOCATION
GET1    BIT   $A800   ;HAS START BIT BEEN RECEIVED?
        BVS   GET1    ;NO. WAIT UNTIL IT IS.
        JSR   DELAY   ;YES. WAIT ONE BIT TIME.
        JSR   DEHALF  ;WAIT ONE-HALF BIT TIME, TO CENTER
GET3    LDA   $A800   ;FETCH INPUT DATA REGISTER B
        AND   #$40    ;MASK OUT ALL BITS EXCEPT BIT 6
        LSR   $40     ;RIGHT-SHIFT DATA BYTE,
        ORA   $40     ;  COMBINE IT WITH NEW BIT,
        STA   $40     ;  AND STORE UPDATED BYTE.
        JSR   DELAY   ;WAIT ONE BIT TIME
        DEX           ;DECREMENT DATA BIT COUNT
        BNE   GET3    ;IF COUNT IS NOT ZERO, GET NEXT BIT
        JSR   DELAY   ;WAIT ONE BIT TIME, TO SKIP PARITY
        LDA   $40     ;FETCH DATA BYTE
        RTS

;THE FOLLOWING SUBROUTINE GENERATES A ONE-BIT TIME DELAY.

DELAY   LDA   #$3F    ;INITIALIZE TIMER 2 LOW COUNT
        STA   $A808
        LDA   #$23    ;INITIALIZE TIMER 2 HIGH COUNT
        STA   $A809   ;  AND START TIMER
DE2     LDA   $A80D   ;FETCH INTERRUPT FLAG REGISTER
        AND   #$20    ;T2 INTERRUPT FLAG SET?
        BEQ   DE2     ;NO. WAIT UNTIL IT IS
        RTS           ;YES. RETURN

;THE FOLLOWING SUBROUTINE GENERATES A ONE-HALF BIT TIME DELAY.

DEHALF  LDA   #$23    ;LOAD TIMER 2 HIGH COUNT
        LSR   A       ;  AND SHIFT IT RIGHT, INTO CARRY
        LDA   #$3F    ;LOAD TIMER 2 LOW COUNT
        ROR   A       ;  AND DIVIDE IT BY 2, WITH CARRY FROM
                      ;  HIGH COUNT
        STA   $A808   ;INITIALIZE TIMER 2 LOW COUNT
        LDA   #$23    ;INITIALIZE TIMER 2 HIGH COUNT,
        LSR   A       ;  DIVIDE IT BY 2,
        STA   $A809   ;  AND START TIMER.
        JMP   DE2     ;GO WAIT FOR TIME-OUT
```

slightly modified versions of the DELAY and DEHALF subroutines of the AIM 65 Monitor (entry addresses $EC0F and $EC23, respectively).

Transmitting Data to a Teletypewriter

From a programming standpoint, transmitting data to a teletypewriter is quite a bit simpler than receiving it. All the program

must do is output the 11 bits that comprise a teletypewriter character, in the following order:

1. Transmit a Start bit (a logic 0) on the data output line.
2. Transmit the seven data bits, with the least-significant bit first.
3. Transmit the Parity bit (a logic 0).
4. Transmit two Stop bits (logic 1s).

The program must generate a 1-bit time delay between the bits being transmitted. Further, since PB2 is being used as the output line, the ASCII character in the accumulator must be initially aligned so that its least-significant bit is in Bit 2 (this can be done by rotating the character left two bit positions). Each subsequent transmit operation must be preceded by a 1-bit right-shift. Fig. 9-12 is a flowchart of the teletypewriter transmit operation.

Example 9-8: A Teletypewriter Transmit Subroutine

```
;THIS SUBROUTINE TRANSMITS AN ASCII CHARACTER IN THE ACCUMULATOR
;TO THE TELETYPEWRITER. LOCATION $40 IS USED FOR TEMPORARY STORAGE.
OUTTTY  PHA             ;SAVE ASCII CHARACTER ON STACK
        STA   $40       ;  AND IN MEMORY
        LDA   #00       ;CLEAR PERIPHERAL CONTROL REGISTER
        STA   $A80C
        STA   $A800     ;TRANSMIT START BIT (LOGIC 0)
        JSR   DELAY     ;WAIT ONE BIT TIME
        LDX   #08       ;DATA BIT COUNT = 8
        LDA   $40       ;FETCH ASCII CHARACTER
        ROL   A         ;ALIGN LSB WITH BIT 2
        ROL   A
OUTBIT  STA   $A800     ;TRANSMIT DATA BIT
        ROR   A         ;SHIFT NEXT BIT INTO BIT 2
        JSR   DELAY     ;WAIT ONE BIT TIME
        DEX             ;ALL BITS TRANSMITTED?
        BNE   OUTBIT    ;NO. TRANSMIT NEXT BIT
        LDA   #04       ;TRANSMIT STOP BITS (LOGIC 1'S)
        STA   $A800
        JSR   DELAY
        JSR   DELAY
        PLA             ;RESTORE ACCUMULATOR CONTENTS
        RTS             ;  AND RETURN
```

Example 9-8 shows a subroutine that uses the flowcharted algorithm. This subroutine (OUTTTY) calls the DELAY subroutine from Example 9-7 to generate the 1-bit time delays between the bit transmit operations. AIM 65 users will note that the example subroutine is similar to the teletypewriter transmit subroutine of the AIM 65, (OUTTTY, entry address $EEA8), but it does not include all of the overhead instructions that are required by the AIM 65 Monitor.

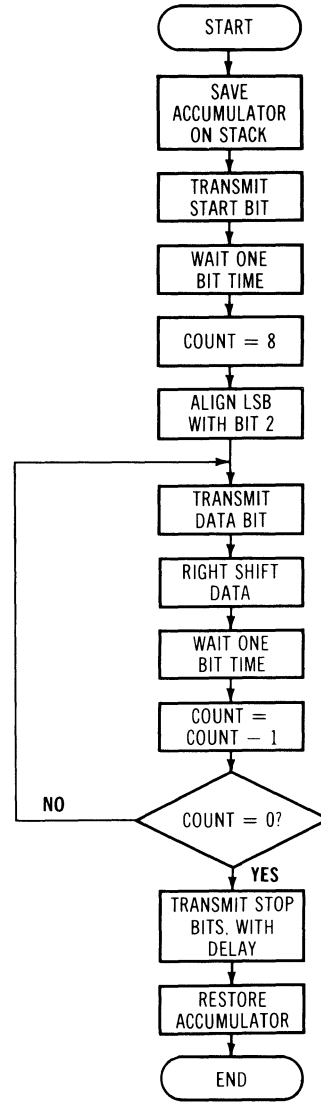


Fig. 9-12. Flowchart for teletypewriter transmit operation.

THE 6502 MICROPROCESSOR AND SEVEN-SEGMENT LED DISPLAYS

At some point in your programming, you may want the 6502 microprocessor to output data to a peripheral device. As we have discussed previously, this device could be a teletypewriter, a CRT,

a digital cassette recorder, or a floppy disk. However, one of the devices that is most often interfaced to a microcomputer is some form of low-cost display. Some microcomputer systems have very simple displays, such as a group of eight LEDs that will display only binary values (as described at the beginning of this chapter). Other microcomputer systems have 5×7 LED matrix displays or sophisticated 16-segment alphanumeric displays, such as the AIM 65 has. However, one of the devices most often interfaced to a microcomputer is the simple seven-segment LED display. This is the same type of display that is used in hand-held calculators and digital clocks. For this reason, we should discuss how to interface seven-segment LED displays to a 6502-based microcomputer.

As we learned in Chapter 4 (Example 4-10 and the accompanying text), the individual segments of a seven-segment LED display are controlled by a set of specific codes. In that chapter, we showed those codes (Fig. 4-2) and presented a subroutine that converted a BCD digit in location $40 to the proper display code in location $41. If the display code is output to a general-purpose I/O device, such as a VIA, that is connected to a set of drivers and a seven-segment display, the interface circuit becomes as simple as the circuit illustrated in Fig. 9-13.

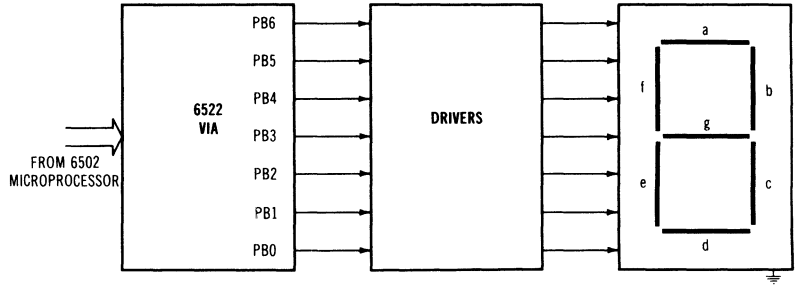


Fig. 9-13. Interfacing to a seven-segment display.

Seven-segment displays are available in two varieties, called the common-anode type and the common-cathode type. If a *common-anode* display is used, the anodes will be connected to +5 volts and a logic 0 will turn a segment on. If a *common-cathode* display is used, the cathodes would be connected to ground and a logic 1 would turn a segment on. Example 9-9 shows a subroutine that can be used to communicate with a common-cathode seven-segment LED display (Fig. 9-13). This subroutine, DISPL, converts a BCD value in location $40 to a seven-segment display code, and then displays that code if it represents a decimal digit (0 to 9); otherwise, it blanks the display (turns all segments off).

Example 9-9: A Seven-Segment Display Conversion and Output Subroutine

```
;THIS SUBROUTINE CONVERTS A BCD DIGIT IN LOCATION $40 TO A
;SEVEN-SEGMENT CODE AND DISPLAYS IT, IF THE BCD DIGIT REPRESENTS
;A DECIMAL DIGIT (0 TO 9). IF THE BCD DIGIT DOES NOT CONVERT
;TO A DECIMAL DIGIT, THE SUBROUTINE BLANKS THE DISPLAY.
DISPL   LDA  #00      ;CLEAR PERIPHERAL CONTROL REGISTER
        STA  $A00C
        LDA  #$FF     ;MAKE PORT B OUTPUTS
        STA  $A002
        LDY  $40      ;FETCH BCD DIGIT INTO Y
        CPY  #10      ;IS DIGIT GREATER THAN NINE?
        BCC  CONV7    ;NO. GO CONVERT AND OUTPUT
        LDA  #00      ;YES. BLANK DISPLAY
        JMP  OUT7
CONV7   LDA  SSEG,Y   ;LOOK UP SEVEN-SEGMENT CODE
OUT7    STA  $A000    ;OUTPUT TO DISPLAY
        RTS

SSEG    .BYT $3F,$06,
             $5B,$4F,
             $66
        .BYT $6D,$7D,
             $07,$7F
             $6F
```

A more common way of interfacing seven-segment LED displays is to *perform the decoding function in hardware*, using a *BCD-to-seven-segment decoder/driver*, such as a 7447 or a 7448. The 7447 decoder/driver interfaces to common-anode displays (a logic 0 lights a segment) and the 7448 interfaces to common-cathode displays (a logic 1 lights a segment). Both types of decoders have a lamp-test input that turns all segments on and, also, blanking inputs and outputs for suppressing leading zeroes. In addition to saving software conversion code, these decoders permit an 8-bit VIA port to service *two* displays directly (see Fig. 9-14), with one display controlled by the four low-order bits and the other display controlled by the four high-order bits. Since 4-bit BCD digits are normally *packed* two per byte, the use of two decoders makes the display subroutine considerably simpler, as you can see by looking at Example 9-10. The subroutine in this example, DISPL1, displays two digits that are packed into location $40.

Example 9-10: A Two-Digit Seven-Segment Display Subroutine for Use With Hardware Decoders

```
DISPL1  LDA  #00    ;CLEAR PERIPHERAL CONTROL REGISTER
        STA  $A00C
        LDA  #$FF   ;MAKE PORT B OUTPUTS
        STA  $A002
        LDA  $40    ;FETCH TWO BCD DIGITS
        STA  $A000  ;  AND DISPLAY THEM
        RTS
```

To study a more complex problem, let us discuss the hardware and software necessary to display a 10-digit BCD number that is stored in memory. We could certainly construct such an interface by using five of the previous circuits (such as Fig. 9-14), but this would require 5 VIAs, 10 type-7448 decoders, and 70 resistors, which would prove relatively expensive. An easier, and more economical way to construct this interface is by *multiplexing* our

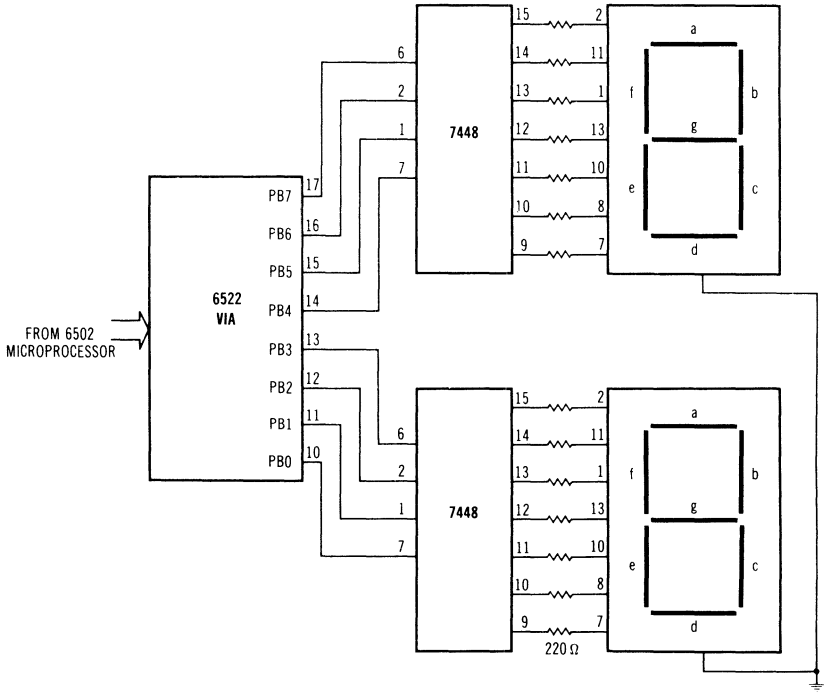


Fig. 9-14. A two-digit seven-segment LED display interfaced to a 6522 VIA.

single VIA and two decoders, as shown in Fig. 9-15. The interface in Fig. 9-15 uses a count-by-five circuit to select the LED displays in pairs. A brief strobe on control line CB2 clocks the counter and directs data to the next pair of displays. Reset ($\overline{\text{RES}}$) initializes the counter to five, so that the first output operation clears the counter and directs data to the first pair of digits.

Example 9-11 shows a routine that could be used to drive the display circuit given in Fig. 9-15. This routine assumes that 10 BCD digits are packed two per byte into locations $40 through $44. Each digit pair is displayed for 3 milliseconds, generated by a subroutine called DLY3. The 3-milliseconds time period is such a short interval that you would not be able to notice that the dis-

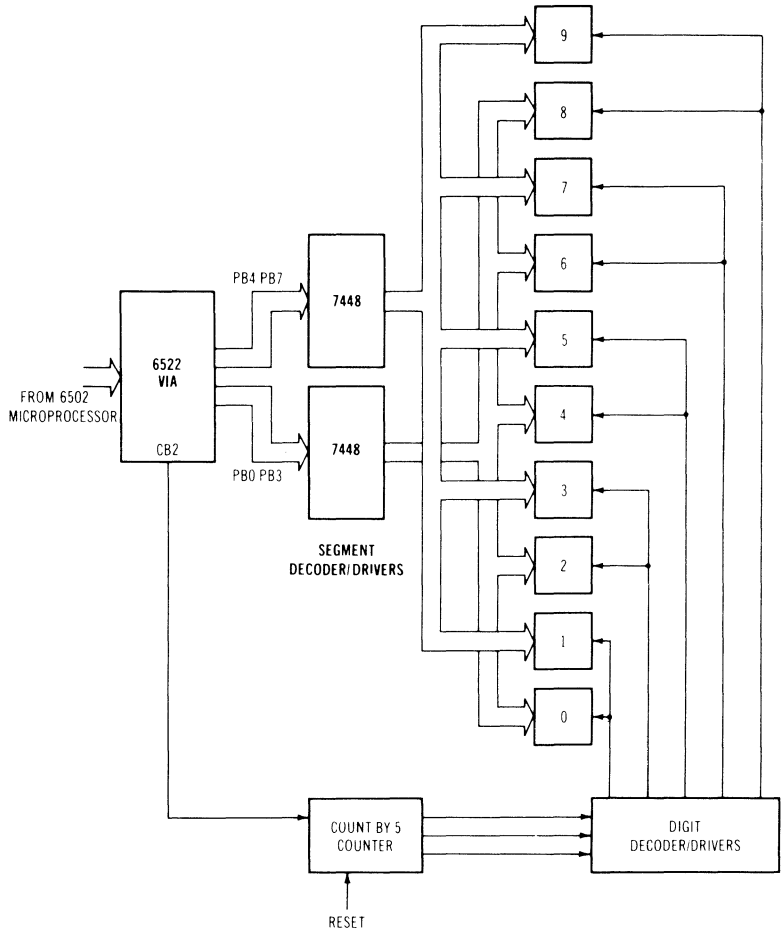


Fig. 9-15. A multiplexed 10-digit LED display.

plays are blinking on and off. Each pair of digits will be on for only 3 milliseconds, which means that all 10 digits (5 pairs) will be displayed once every 15 milliseconds. This means that all of the digits will be turned on and off at a rate of slightly more than 60 times per second. At this rate, your eyes will not be able to detect that the displays are flashing on and off.

SUMMARY

This chapter was not intended to present an exhaustive treatment of interfacing, but is rather an introduction to the funda-

Example 9-11: The Software for a Multiplexed 10-Digit Seven-Segment Display

```
;THIS ROUTINE DISPLAYS THE CONTENTS OF MEMORY LOCATIONS $40
;THROUGH $44 ON A 10-DIGIT SEVEN-SEGMENT DISPLAY.

          LDA  #$A0      ;SET CB2 TO PULSE OUTPUT MODE
          STA  $A00C
          LDA  #$FF      ;MAKE PORT B OUTPUTS
          STA  $A002
RECYCL    LDX  #00       ;POINT TO FIRST LOCATION
          LDY  #05       ;NUMBER OF DIGIT PAIRS = 5
OUTMUX    LDA  $40,X     ;FETCH NEXT TWO DIGITS
          STA  $A000     ;  AND DISPLAY THEM
          JSR  DLY3      ;WAIT 3 MILLISECONDS
          INX
          DEY            ;ANY MORE DIGITS?
          BNE  OUTMUX    ;YES. GO FETCH THEM
          BEQ  RECYCL    ;NO. CYCLE THROUGH AGAIN

;THE FOLLOWING SUBROUTINE USES TIMER 1 TO GENERATE A 3-MILLISECOND
;DELAY, BY WRITING 3000 ($0BB8) INTO THE COUNTERS.

DLY3      LDA  #00       ;SET T1 ONE-SHOT MODE, WITH NO PB7
          STA  $A00B
          LDA  #$B8      ;WRITE COUNT LSBY
          STA  $A004
          LDA  #$0B      ;WRITE COUNT MSBY AND START TIMER
          STA  $A005
          LDA  #$40      ;SELECT T1 INTERRUPT MASK
CHKT1     BIT  $A00D     ;HAS TIMER 1 COUNTED DOWN?
          BEQ  CHKT1     ;NO. WAIT UNTIL IT HAS
          LDA  $A004     ;YES. CLEAR T1 INTERRUPT FLAG
          RTS            ;  AND RETURN
```

mental *concepts* of interfacing. Indeed, there are many types of devices that were not covered at all, such as cassette recorders, CRTs, analog-to-digital converters, digital-to-analog converters, and a variety of other devices. Readers who are interested in further information on interfacing should read the books listed in References 1 through 5.

REFERENCES

1. De Jong, M. L. *Programming and Interfacing the 6502*. Howard W. Sams & Co., Inc., Indianapolis, IN, 1979.

2. Larsen, D. G. and Rony, P. R. *Interfacing and Scientific Data Communications Experiments*. Howard W. Sams & Co., Inc., Indianapolis, IN, 1978. (An excellent book for interfacing information on UART and USART.)

3. Larsen, D. G. and Rony, P. R. *Logic and Memory Experiments Using TTL Integrated Circuits*. Howard W. Sams & Co., Inc., Indianapolis, IN, 1978. (This is a very good volume for the reader who wants to know more about gates, logic counters, decoders, multiplexers, flip-flops, LED displays, and the like.)

4. Titus, J. A., Titus, C. A., Rony, P. R., and Larsen, D. G. *Microcomputer–Analog Converter Software and Hardware Interfacing.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978. (Analog-to-digital and digital-to-analog converters are discussed in this book.)

5. Peatman, J. *Microcomputer-Based Design.* McGraw-Hill, New York, NY, 1977.

APPENDIX A

ASCII Character Set (7-Bit Code)

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

Courtesy Rockwell International

APPENDIX B

Summary of the 6502 Instruction Set

The following notation applies to this summary:

A	Accumulator
X, Y	Index registers
M	Memory
P	Processor status register
S	Stack Pointer
$\surd$	Change
–	No change
+	Add
$\wedge$	Logical AND
–	Subtract
V	Logical Exclusive-OR
$\rightarrow, \leftarrow$	Transfer to
$\forall$	Logical (inclusive) OR
PC	Program counter
PCH	Program counter high
PCL	Program counter low
#dd	8-bit immediate data value (2 hexadecimal digits)
aa	8-bit zero page address (2 hexadecimal digits)
aaaa	16-bit absolute address (4 hexadecimal digits)
$\uparrow$	Transfer from stack (Pull)
$\downarrow$	Transfer onto stack (Push)

ADC

Add to Accumulator with Carry

Operation: A + M + C → A, C

N Z C I D V
✓ ✓ ✓ – – ✓

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	ADC	#dd	69	2	2
Zero Page	ADC	aa	65	2	3
Zero Page, X	ADC	aa,X	75	2	4
Absolute	ADC	aaaa	6D	3	4
Absolute, X	ADC	aaaa,X	7D	3	4*
Absolute, Y	ADC	aaaa,Y	79	3	4*
(Indirect, X)	ADC	(aa,X)	61	2	6
(Indirect), Y	ADC	(aa),Y	71	2	5*

*Add 1 if page boundary is crossed.

AND

AND Memory with Accumulator

Logical AND to the accumulator
Operation: A ∧ M → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	AND	#dd	29	2	2
Zero Page	AND	aa	25	2	3
Zero Page, X	AND	aa,X	35	2	4
Absolute	AND	aaaa	2D	3	4
Absolute, X	AND	aaaa,X	3D	3	4*
Absolute, Y	AND	aaaa,Y	39	3	4*
(Indirect, X)	AND	(aa,X)	21	2	6
(Indirect), Y	AND	(aa),Y	31	2	5*

*Add 1 if page boundary is crossed.

ASL

Accumulator Shift Left

Operation: C ← [7|6|5|4|3|2|1|0] ← 0

N Z C I D V
√ √ √ – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ASL A	0A	1	2
Zero Page	ASL aa	06	2	5
Zero Page, X	ASL aa,X	16	2	6
Absolute	ASL aaaa	0E	3	6
Absolute, X	ASL aaaa,X	1E	3	7

BCC

Branch on Carry Clear

Operation: Branch on C = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCC aa	90	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BCC aaaa), and convert it to a relative address.

BCS

Branch on Carry Set

Operation: Branch on C = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCS aa	B0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to next page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BCS aaaa), and convert it to a relative address.

BEQ

Branch on Result Equal to Zero

Operation: Branch on Z = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BEQ aa	F0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to next page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BEQ aaaa), and convert it to a relative address.

BIT

Test Bits in Memory with Accumulator

Operation: A M, $M_7 \rightarrow$ N, $M_6 \rightarrow$ V
Bit 6 and 7 are transferred to the Status Register. If the result of A M is zero then Z = 1, otherwise Z = 0

N Z C I D V
M_7 √ – – – M_6

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	BIT aa	24	2	3
Absolute	BIT aaaa	2C	3	4

BMI

Branch on Result Minus

Operation: Branch on N = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BMI aa	30	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BMI aaaa), and convert it to a relative address.

BNE

Branch on Result Not Equal to Zero

Operation: Branch on Z = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BNE aa	D0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BNE aaaa), and convert it to a relative address.

BPL

Branch on Result Plus

Operation: Branch on N = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BPL aa	10	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BPL aaaa), and convert it to a relative address.

BRK

Force Break

Operation: Forced Interrupt PC + 2 ↓ P ↓

B N Z C I D V
1 – – – 1 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	BRK	00	1	7

BVC

Branch on Overflow Clear

Operation: Branch on V = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVC aa	50	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BVC aaaa), and convert it to a relative address.

BVS

Branch on Overflow Set

Operation: Branch on V = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVS aa	70	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BVS aaaa), and convert it to a relative address.

CLC

Clear Carry Flag

Operation: 0 → C

N Z C I D V
– – 0 – – –

Addressing Mode	Assembly Language Form	OP CODE	No Bytes	No. Cycles
Implied	CLC	18	1	2

CLD

Clear Decimal Mode

Operation: 0 → D

N Z C I D V
– – – – 0 –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLI

Clear Interrupt Disable Bit

Operation: 0 → I

N Z C I D V
– – – 0 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLV

Clear Overflow Flag

Operation: 0 → V

N Z C I D V
– – – – – 0

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CMP

Compare Memory and Accumulator

Operation: A — M

N Z C I D V
√ √ √ — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CMP #dd	C9	2	2
Zero Page	CMP aa	C5	2	3
Zero Page, X	CMP aa,X	D5	2	4
Absolute	CMP aaaa	CD	3	4
Absolute, X	CMP aaaa,X	DD	3	4*
Absolute, Y	CMP aaaa,Y	D9	3	4*
(Indirect, X)	CMP (aa,X)	C1	2	6
(Indirect), Y	CMP (aa),Y	D1	2	5*

*Add 1 if page boundary is crossed.

CPX

Compare Memory and Index X

Operation: X — M

N Z C I D V
√ √ √ — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPX #dd	E0	2	2
Zero Page	CPX aa	E4	2	3
Absolute	CPX aaaa	EC	3	4

CPY

Compare Memory and Index Y

Operation: Y — M

N Z C I D V
√ √ √ — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPY #dd	C0	2	2
Zero Page	CPY aa	C4	2	3
Absolute	CPY aaaa	CC	3	4

DEC

Decrement Memory by One

Operation: M − 1 → M

N Z C I D V
√ √ − − − −

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	DEC aa	C6	2	5
Zero Page, X	DEC aa,X	D6	2	6
Absolute	DEC aaaa	CE	3	6
Absolute, X	DEC aaaa,X	DE	3	7

DEX

Decrement Index X by One

Operation: X − 1 → X

N Z C I D V
√ √ − − − −

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEX	CA	1	2

DEY

Decrement Index Y by One

Operation: Y − 1 → Y

N Z C I D V
√ √ − − − −

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR

Exclusive-OR Memory with Accumulator

Operation: A V M → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	EOR	#dd	49	2	2
Zero Page	EOR	aa	45	2	3
Zero Page, X	EOR	aa,X	55	2	4
Absolute	EOR	aaaa	4D	3	4
Absolute, X	EOR	aaaa,X	5D	3	4*
Absolute, Y	EOR	aaaa,Y	59	3	4*
(Indirect, X)	EOR	(aa,X)	41	2	6
(Indirect), Y	EOR	(aa),Y	51	2	5*

*Add 1 if page boundary is crossed.

INC

Increment Memory by One

Operation: M + 1 → M

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Zero Page	INC	aa	E6	2	5
Zero Page, X	INC	aa,X	F6	2	6
Absolute	INC	aaaa	EE	3	6
Absolute, X	INC	aaaa,X	FE	3	7

INX

Increment Index X by One

Operation: X + 1 → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY

Increment Index Y by One

Operation: Y + 1 → Y

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP Code	No. Bytes	No. Cycles
Implied	INY	C8	1	2

JMP

Jump

Operation: (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JMP aaaa	4C	3	3
Indirect	JMP (aaaa)	6C	3	5

JSR

Jump to Subroutine

Operation: PC + 2 ↓, (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JSR aaaa	20	3	6

LDA

Load Accumulator with Memory

Operation: M → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDA	#dd	A9	2	2
Zero Page	LDA	aa	A5	2	3
Zero Page, X	LDA	aa,X	B5	2	4
Absolute	LDA	aaaa	AD	3	4
Absolute, X	LDA	aaaa,X	BD	3	4*
Absolute, Y	LDA	aaaa,Y	B9	3	4*
(Indirect, X)	LDA	(aa,X)	A1	2	6
(Indirect), Y	LDA	(aa),Y	B1	2	5*

*Add 1 if page boundary is crossed.

LDX

Load Index X with Memory

Operation: M → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDX	#dd	A2	2	2
Zero Page	LDX	aa	A6	2	3
Zero Page, Y	LDX	aa,Y	B6	2	4
Absolute	LDX	aaaa	AE	3	4
Absolute, Y	LDX	aaaa,Y	BE	3	4*

*Add 1 when page boundary is crossed.

LDY

Load Index Y with Memory

Operation: M → Y

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDY	#dd	A0	2	2
Zero Page	LDY	aa	A4	2	3
Zero Page, X	LDY	aea,X	B4	2	4
Absolute	LDY	aaaa	AC	3	4
Absolute, X	LDY	aaaa,X	BC	3	4*

*Add 1 when page boundary is crossed.

LSR

Local Shift Right

Operation: 0 → [7|6|5|4|3|2|1|0] → C

N Z C I D V
0 √ √ – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Accumulator	LSR	A	4A	1	2
Zero Page	LSR	aa	46	2	5
Zero Page, X	LSR	aa,X	56	2	6
Absolute	LSR	aaaa	4E	3	6
Absolute, X	LSR	aaaa,X	5E	3	7

NOP

No Operation

Operation: No Operation (2 cycles)

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	NOP	EA	1	2

ORA

OR Memory with Accumulator

Operation: A V M → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	ORA	#dd	09	2	2
Zero Page	ORA	aa	05	2	3
Zero Page, X	ORA	aa,X	15	2	4
Absolute	ORA	aaaa	0D	3	4
Absolute, X	ORA	aaaa,X	1D	3	4*
Absolute, Y	ORA	aaaa,Y	19	3	4*
(Indirect, X)	ORA	(aa,X)	01	2	6
(Indirect), Y	ORA	(aa),Y	11	2	5*

*Add 1 on page crossing.

PHA

Push Accumulator on Stack

Operation: A ↓

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHA	48	1	3

PHP

Push Processor Status on Stack

Operation: P↓

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHP	08	1	3

PLA

Pull Accumulator from Stack

Operation: A ↑

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP

Pull Processor Status from Stack

Operation: P ↑

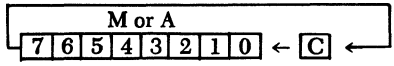
N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL

Rotate Left

Operation:

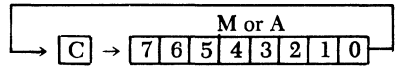


N Z C I D V
✓ ✓ ✓ – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Accumulator	ROL	A	2A	1	2
Zero Page	ROL	aa	26	2	5
Zero Page, X	ROL	aa,X	36	2	6
Absolute	ROL	aaaa	2E	3	6
Absolute, X	ROL	aaaa,X	3E	3	7

ROR

Rotate Right

Operation: 

N Z C I D V
√ √ √ – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROR A	6A	1	2
Zero Page	ROR aa	66	2	5
Zero Page, X	ROR aa,X	76	2	6
Absolute	ROR aaaa	6E	3	6
Absolute, X	ROR aaaa,X	7E	3	7

RTI

Return from Interrupt

Operation: P↑ PC↑

N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTI	40	1	6

RTS

Return from Subroutine

Operation: PC↑, PC + 1 → PC

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTS	60	1	6

SBC

Subtract from Accumulator with Carry

Operation: $A - M - \overline{C} \rightarrow A$

Note: $\overline{C}$ = Borrow

N Z C I D V
√ √ √ – – √

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	SBC	#dd	E9	2	2
Zero Page	SBC	aa	E5	2	3
Zero Page, X	SBC	aa,X	F5	2	4
Absolute	SBC	aaaa	ED	3	4
Absolute, X	SBC	aaaa,X	FD	3	4*
Absolute, Y	SBC	aaaa,Y	F9	3	4*
(Indirect, X)	SBC	(aa,X)	E1	2	6
(Indirect), Y	SBC	(aa),Y	F1	2	5*

*Add 1 when page boundary is crossed.

SEC

Set Carry Flag

Operation: $1 \rightarrow C$

N Z C I D V
– – 1 – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEC	38	1	2

SED

Set Decimal Mode

Operation: $1 \rightarrow D$

N Z C I D V
– – – – 1 –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI

Set Interrupt Disable Status

Operation: 1 → I

N Z C I D V
– – – 1 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA

Store Accumulator in Memory

Operation: A → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STA aa	85	2	3
Zero Page, X	STA aa,X	95	2	4
Absolute	STA aaaa	8D	3	4
Absolute, X	STA aaaa,X	9D	3	5
Absolute, Y	STA aaaa,Y	99	3	5
(Indirect, X)	STA (aa,X)	81	2	6
(Indirect), Y	STA (aa),Y	91	2	6

STX

Store Index X in Memory

Operation: X → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STX aa	86	2	3
Zero Page, Y	STX aa,Y	96	2	4
Absolute	STX aaaa	8E	3	4

STY

Store Index Y in Memory

Operation: Y → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STY aa	84	2	3
Zero Page, X	STY aa,X	94	2	4
Absolute	STY aaaa	8C	3	4

TAX

Transfer Accumulator to Index X

Operation: A → X

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAX	AA	1	2

TAY

Transfer Accumulator to Index Y

Operation: A → Y

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAY	A8	1	2

TSX

Transfer Stack Pointer to Index X

Operation: S → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TSX	BA	1	2

TXA

Transfer Index X to Accumulator

Operation: X → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXA	8A	1	2

TXS

Transfer Index X to Stack Pointer

Operation: X → S

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXS	9A	1	2

TYA

Transfer Index Y to Accumulator

Operation: Y → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TYA	98	1	2

Index

TO THE READER

This book is one of an expanding series of books that will cover the field of basic electronics and digital electronics from basic gates and flip-flops through microcomputers and digital telecommunications. We are attempting to develop a mailing list of individuals who would like to receive information on the series. We would be delighted to add your name to it if you would fill in the information below and mail this sheet to us. Thanks.

1. I have the following books:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

2. My occupation is: ☐ student ☐ teacher, instructor ☐ hobbyist

☐ housewife ☐ scientist, engineer, doctor, etc. ☐ businessman

☐ Other: _____

Name (print): _____

Address _____

City _____ State _____

Zip Code _____

Mail to:

Books
P.O. Box 715
Blacksburg, Virginia 24060

The Blacksburg Group

According to Business Week magazine (Technology July 6, 1976) large scale integrated circuits or LSI "chips" are creating a second industrial revolution that will quickly involve us all. The speed of the developments in this area is breathtaking and it becomes more and more difficult to keep up with the rapid advances that are being made. It is also becoming difficult for newcomers to "get on board."

It has been our objective, as The Blacksburg Group, to develop timely and effective educational materials and aids that will permit students, engineers, scientists and others to quickly learn how to apply new technologies to their particular needs. We are doing this through a number of means, textbooks, short courses, and through the development of educational "hardware" or training aids.

Our group members make their home in Blacksburg, found in the Appalachian Mountains of southwestern Virginia. While we didn't actively start our group collaboration until the Spring of 1974, members of our group have been involved in digital electronics, minicomputers and microcomputers for some time.

Some of our past experiences and on-going efforts include the following:

—The development of the Mark-8 computer, an 8008-based device that was featured in Radio-Electronics magazine in 1974, and generally recognized as the first widely available hobby computer. We have also designed several 8080-based computers, including the Mini-Micro Designer (MMD-1). More recently we have been working with 8085-based computers and the TRS-80.

—The Blacksburg Continuing Education Series™ covers subjects ranging from basic electronics through microcomputers, operational amplifiers, and active filters. Test experiments and examples have been provided in each book. We are strong believers in the use of detailed experiments and examples to reinforce basic concepts. This series originally started as our Bugbook series and many titles are now being translated into Chinese, Japanese, German and Italian.

—We have pioneered the use of small self-contained computers in hands-on courses aimed at microcomputer users. The solderless breadboarding modules developed for use in circuit design and development make it easy for people to set up and test digital circuits and computer interfaces. Some of our technical products are marketed by Group Technology, Ltd., Check, VA 24072, USA. (703) 651-3153.

—Our short course programs have been presented throughout the world, covering digital electronics through TRS-80 computer interfacing. Programs are offered through the Blacksburg Group and the Virginia Tech Extension Division. Each course offers a mix of lectures and hands-on laboratory sessions. Courses are presented on a regular basis in Blacksburg, and at various times to open groups, companies, schools, and other sponsors.

For additional information about course offerings, we encourage you to write or call Dr. Chris Titus at The Blacksburg Group, Box 242, Blacksburg, VA 24060, (703) 951-9030, or Dr. Linda Leffer at the Center for Continuing Education, Virginia Tech, Blacksburg, VA 24061, (703) 961-5241.

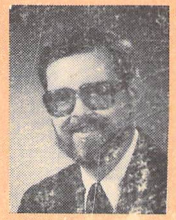
Mr. David Larsen is on the faculty of the Department of Chemistry at Virginia Polytechnic Institute and State University. Dr. Jonathan Titus and Dr. Christopher Titus are with The Blacksburg Group, Inc., all of Blacksburg, Virginia.

6502 SOFTWARE DESIGN

The 6502 integrated circuit is a very popular microprocessor that is currently being used in general-purpose microcomputers, video games, and personal computers. Chapter 1 discusses the characteristics of the 6502 integrated circuit and the AIM 65 microcomputer.

Although the AIM 65, a 6502-based microcomputer manufactured by Rockwell International, has been used to generate and test the program examples given in this book, most of the programs listed can be used on all 6502-based microcomputers. However, they may have to be slightly altered to reflect the memory and I/O devices that are wired to your microcomputer.

Chapters 2 and 3 discuss subroutines and present descriptions of the instructions that the 6502 microprocessor can execute. Then, Chapters 4 through 6 present techniques needed to process lists and tables, perform mathematical operations, and convert data. Finally, a description of ways to transfer information between the microprocessor and the input/output devices by using the 6502 instructions is given in Chapters 7 through 9.



Leo J. Scanlon is Documentation Manager for the Microelectronic Devices business segment of Rockwell International, in Anaheim, CA. He received his bachelor of Science degree in Aeronautical Engineering from St. Louis University. He has done graduate studies in Electrical Engineering and Computer Science at the University of California, in Berkeley, CA.

Leo's experience includes technical writing in the minicomputer and microcomputer industries, and engineering programming in the aerospace industry. Before joining Rockwell International, he served as Technical Publications Manager with Computer Automation, Inc., in Irvine, CA.

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

$10.50/21656

ISBN: 0-672-21656-8