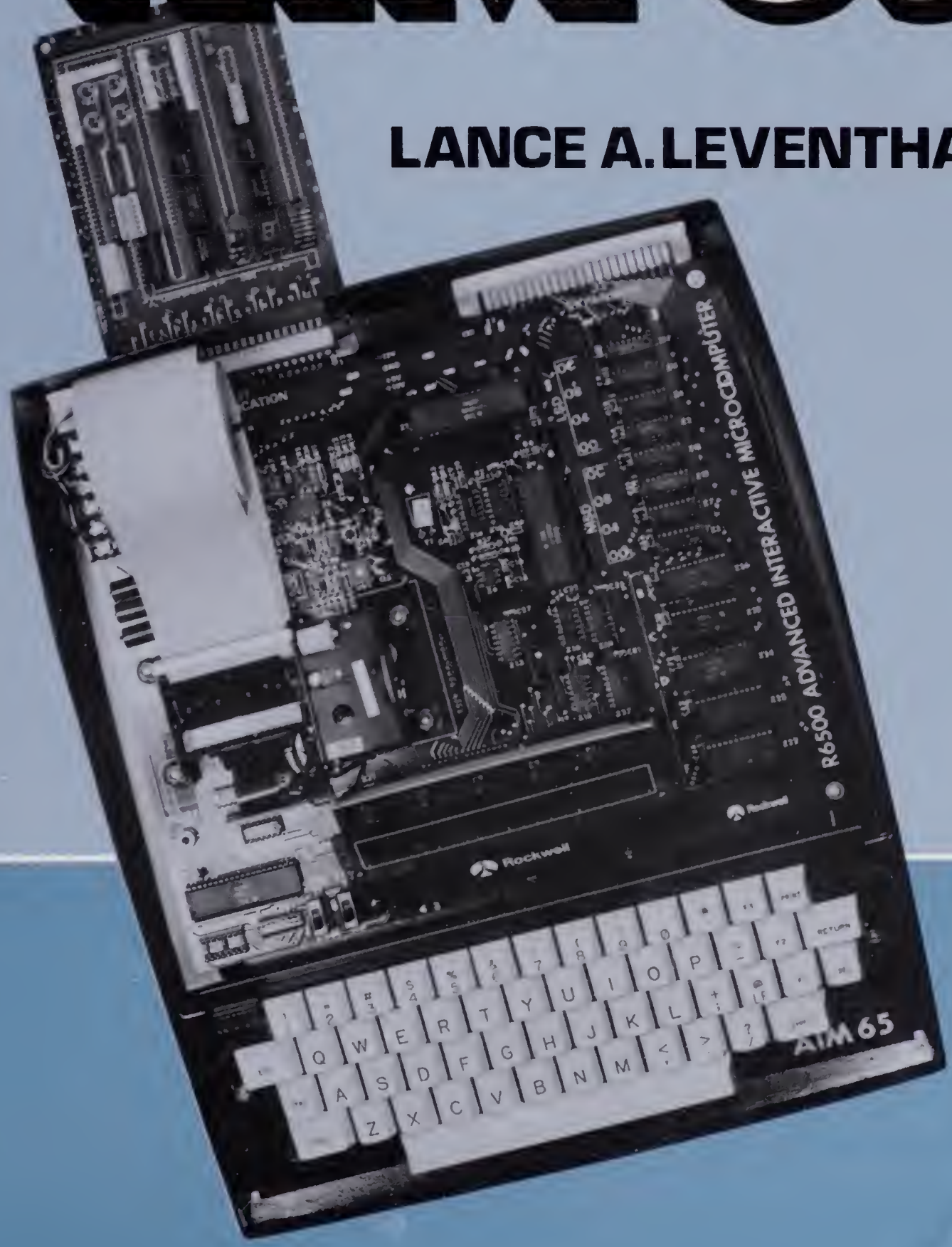
MICROCOMPUTER EXPERIMENTATION WITH THE AIM 65

LANCE A. LEVENTHAL



MICROCOMPUTER EXPERIMENTATION WITH THE AIM 65

LANCE A. LEVENTHAL
Emulative Systems Company
San Diego, California

PRENTICE-HALL, INC., *Englewood, Cliffs, New Jersey 07632*

Library of Congress Cataloging-in-Publication Data

LEVENTHAL, LANCE A., 1945-
Microcomputer experimentation with the AIM 65.

Includes index.
1. AIM 65 (Computer)—Programming—Laboratory manuals. 2. Automatic control—Laboratory manuals. 3. 6502 (Microprocessor)—Programming—Laboratory manuals. I. Title.
TK7889.A37L48 1987 629.8'95 86-12277
ISBN 0-13-580283-0

Editorial/production supervision and
interior design: *Cheryl Smith/Mary Jo Stanley*
Cover design: *Photo Plus Art*
Manufacturing buyer: *Rhett Conklin*

This book is dedicated to Stan Rogers, in appreciation of all he taught me about technical writing.

AIM 65®—Courtesy of Dynatem under license from Rockwell International.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-580283-0 025

PRENTICE-HALL INTERNATIONAL (UK) LIMITED, *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*

CONTENTS

PREFACE

The purpose of this manual is to provide experimental training on microcomputers for people in the fields of engineering, engineering technology, computer science, the physical sciences, electronics, and related fields. The emphasis is on the design of controllers for industrial and laboratory use. The experiments, examples, and problems were adapted from applications in instrumentation, test equipment, communications, computers and peripherals, industrial control, process control, business equipment, aerospace and military systems, and consumer products. The manual illustrates the use of microcomputers in tasks that are essential to all these applications, responding to switches, controlling displays, encoding and decoding data, collecting and processing data, executing arithmetic functions, interfacing simple handshaking peripherals (such as terminals and printers), timing and scheduling operations, and implementing serial communications.

First, the manual describes how to operate the microcomputer. It then introduces assembly language programming, shows how to perform simple controller functions, discusses hardware-software trade-offs, describes how to design and develop programs, illustrates alternative approaches to input/output and timing, presents the advantages and uses of programmable LSI devices, and describes serial communications. The final experiment provides a brief introduction to hardware design and development. The manual includes numerous examples drawn from actual applications, but simplified to avoid requiring extensive background, special equipment (beyond the microcomputer itself), or long setup times. Because the manual is self-contained, it can be used by people with a variety of interests and backgrounds.

The manual is based on the popular AIM 65 microcomputer (formerly manufactured by Rockwell International and now licensed to Dynatem of Irvine, California) because of its low cost, wide availability, completeness, and ease of use. The AIM does not require

expensive peripherals (such as a terminal), has excellent documentation, requires no assembly, has expansion capabilities, and contains all the components of typical microcomputers. The AIM's ROM-based monitor makes it easy to explain and use, and it has enough memory and input/output lines for many interesting and relevant examples.

This manual emphasizes the control of systems with software. This control is illustrated with simple examples using switches, single displays, and the on-board peripherals. The intent here is to provide realistic examples that require little additional hardware and can be performed in short time periods. Numerous programs are included as starting points for students and as references for examination and comparison.

The standard format used in this book conforms with other textbooks, manuals, and reference materials. I have used the notation from the 6502 assembler provided by MOS Technology (now a division of Commodore Business Machines). I have tried to make all programs clear, simple, well structured, and well documented. I have avoided programming tricks, even when they would make programs somewhat shorter and faster. Good programming practices are essential to microcomputer users, so I have tried to provide sound, fully tested examples for readers to follow. Besides, few things give readers more pleasure than writing a program that is shorter, faster, and more ingenious than the one in the book.

This manual does not describe the 6502 microprocessor in detail. Nor does it offer a complete discussion of 6502 assembly language programming. I therefore provide extensive references to appropriate textbooks, 6502 manuals, and programming books. Because the manual is self-contained, it can be used independently of the reference materials. I have also compiled an extensive index so that casual users can find programs of specific interest to them and students can readily find material for review, reference, or further information.

Each experiment in the manual is itself self-contained. Each includes a list of goals, definition of new terms, references (with page numbers), descriptions of instructions that are being introduced, a list of required equipment (with diagrams), and a key point summary. Each laboratory exercise contains numerous problems that are linked closely to the discussion. The problems illustrate key points, relate current material to previous experiments, and provide examples from actual applications; there are no "make-work" problems or rote tasks. I have tested all the problems and provide sample data, hints, and discussions.

Many people contributed to the writing of this manual. Irvin Stafford of Burroughs Corporation constructed the hardware, checked the examples and problems, and suggested many improvements and corrections. Carter Stafford provided the photographs of the AIM and the other laboratory apparatus. Ralph Reccia of Rockwell International was most helpful in answering questions and providing materials and encouragement. Dennis Starbuck of Dynatem also provided current material describing his company's product line. Professor Wilson Turner of California State University, San Bernardino, used a preliminary version of the manual in his classes and offered many criticisms and suggestions. Cheryl Smith and Mary Jo Stanley of Prentice-Hall, along with copy editor Bruce Emmer, did an outstanding production job. All remaining errors are, of course, my fault. Cheers!

LANCE A. LEVENTHAL

San Diego, California

BASIC OPERATIONS

PURPOSE

To learn how to operate the microcomputer.

REFERENCE MATERIALS

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 11–12 (hexadecimal number system), 70–72 (types of semiconductor memories), 92–94 (basic computer system organization).

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, Chapters 1 and 2.

WHAT YOU SHOULD LEARN

1. How to reset the computer.
2. How to examine a memory location.
3. How to change a memory location.
4. How to enter and run a program.

TERMS

Byte a unit of 8 bits, may be described as consisting of two hexadecimal digits (the four most significant bits and the four least significant bits).

Central processing unit (CPU) the part of the computer that controls its operations and processes data.

Cursor a movable indicator of position on a display.

Hexadecimal (hex) number system with base 16. The digits are the numbers 0 through 9, followed by the letters A through F (see Table 0–1).

Microcomputer a computer with a microprocessor as its CPU.

Microprocessor a single-chip CPU.

Monitor a program that allows the computer user to enter programs and data, run programs, examine memory and registers, and use peripherals.

Nonvolatile memory a memory that retains its contents when power is lost.

Read-only memory (ROM) a memory that can be read but not changed in normal operation.

Read/write (random-access) memory (RAM) a memory that can be both read and changed (written) in normal operation.

Reset a signal that puts the computer in a known initial state.

Scratchpad a memory area that is especially convenient to use for temporary storage.

Word the basic grouping of bits that a computer can process at one time. The 6502 microprocessor has an 8-bit word.

6502 INSTRUCTIONS

BRK (00 hex) force break; on the AIM 65, BRK transfers control to the monitor.

OVERVIEW

The AIM 65 (Figure 0-1) is an inexpensive microcomputer. Sections 1.6 through 1.8 of

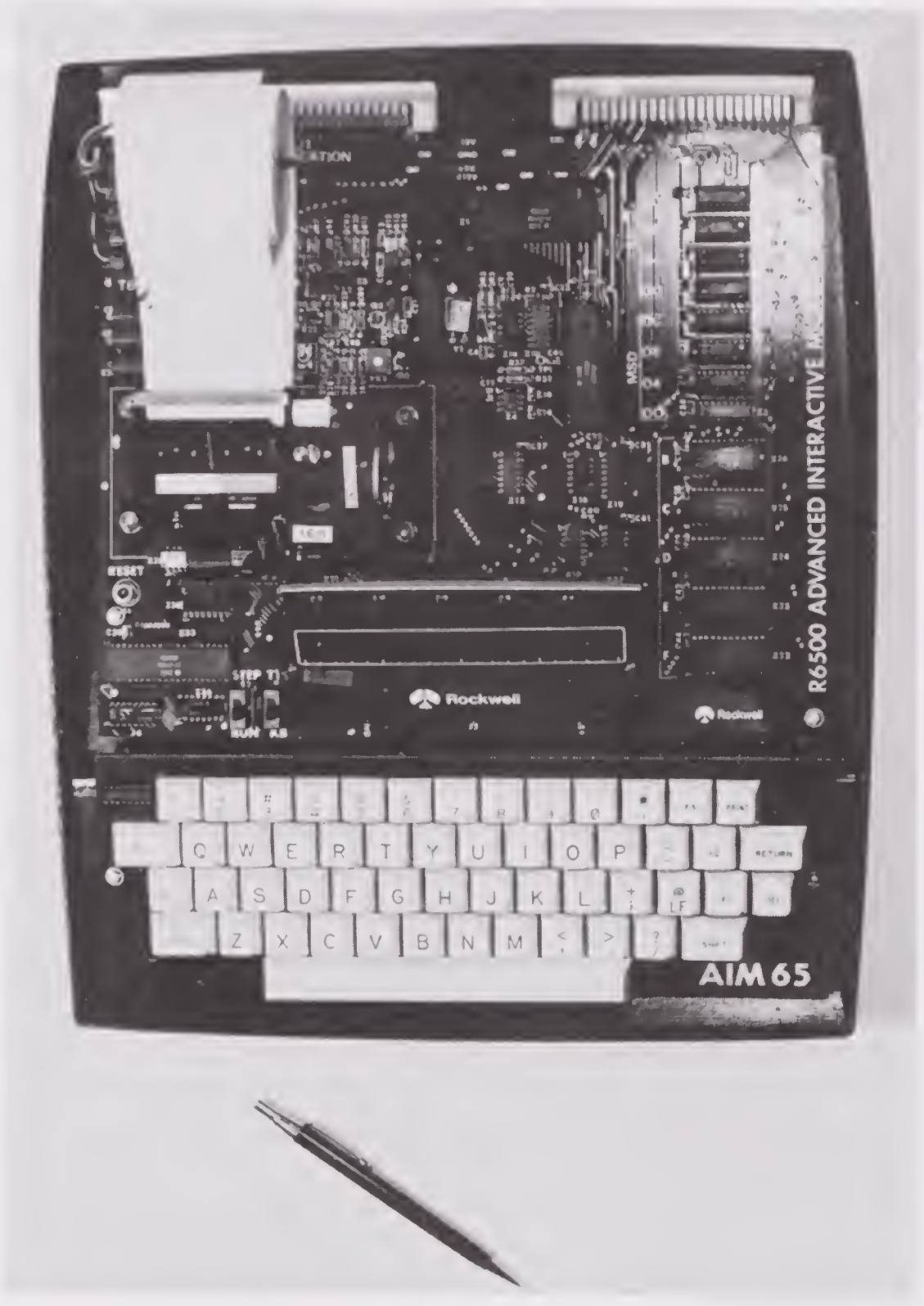


FIGURE 0–1. The AIM 65 microcomputer. (Courtesy of Carter Stafford.)

the *AIM 65 User's Guide* describe how to set it up and attach a power supply. It has the following components:

- A 6502 microprocessor, the central processing unit or "brain."
- Read-only memory, or ROM (two 2332 devices that contain a monitor). Each 2332 ROM consists of 4K 8-bit units (*bytes*). $1K = 2^{10} = 1{,}024$.
- Read/write memory, or RAM (two 2114 devices into which the user can enter data and programs). Each 2114 RAM consists of 1K 4-bit units. The two 2114s connected in parallel provide 1K bytes of memory. There are sockets for six more 2114s, thus expanding RAM to 4K bytes.

- Two 6522 Versatile Interface Adapters (VIAs). Each VIA contains two 8-bit input/output (I/O) ports.
- A 54-key keyboard. Most keys are located just as on a standard typewriter; note, however, that there are no lowercase letters.
- A 20-character alphanumeric display.
- A 20-column thermal printer.
- A cassette interface that lets you transfer programs and data to and from audio cassettes (see Sections 2.9 and 2.10 of the *AIM 65 User's Guide*). We will not discuss cassettes again; you can use them to save programs and thus avoid repetitive keyboard input.

Appendix 3 contains descriptions of the 6502 and 6522 devices. For those unfamiliar with the hexadecimal (base 16) number system, there is a brief explanation in the textbook by Tocci and Laskowski, as well as in many other books. Hexadecimal notation is merely a convenience to avoid references to long binary numbers.

RESETTING THE COMPUTER

Before you start working with the AIM, you should reset it. The RESET button is located near the left edge of the computer just below the printer. Press and release it. The AIM should flash a message (ROCKWELL AIM 65) on its display and then leave a prompt (∠) at the far left.

The AIM is now executing the monitor stored in the 2332 ROMs. This program allows you to control the AIM from its keyboard. You can place programs and data in read/write memory, run programs, examine and change memory and registers, and do other simple operations. For more details, see the booklet entitled *AIM 65 Monitor Program Listing*.

EXAMINING MEMORY

The basic AIM 65 has 1K bytes of read/write memory (RAM) in addresses 0000 through 03FF hexadecimal. Some of these addresses are special (see Appendix 5 for a complete map of AIM 65 memory). In particular, note the following:

- The monitor uses 00DF through 00FF for its own purposes. We will therefore not use those locations.
- The 6502 microprocessor uses 0100 through 01FF for its stack. We will discuss the stack in Laboratory A, but for now we will simply not use those locations either.
- Addresses 0000 through 00FF (the monitor uses some of these) serve as a scratchpad. We will see why in Laboratory I. We will use 0000 through 00DE for data only; we will not put any programs there.

Overall, we will use the following addresses:

- 0000 through 00DE for data.
- 0100 through 01FF for the stack.
- 0200 through 03FF for programs and data.

Note that each location has a 16-bit address (four hexadecimal digits) and contains 8 bits of data (two hexadecimal digits). Table 0-1 is a list of the hexadecimal digits and their binary and decimal equivalents. Use this table if you need help converting numbers from one base to another.

TABLE 0–1 HEXADECIMAL-TO-DECIMAL CONVERSION TABLE

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

Before you continue, be sure the AIM is in the following state:

1. Printer off. You can toggle the printer (that is, turn it off if it is on and vice versa) by pressing CTRL (leftmost key in the middle row) and PRINT (rightmost key in the top row) together. The display indicates whether the printer is on or off. You should turn it off initially to avoid wasting paper.
2. RUN/STEP and KB/TTY switches both forward (toward the keyboard). These switches are located just left of the display. Moving them forward puts RUN/STEP in the RUN position and KB/TTY in the KB position.

To examine memory, first press the M key. The display will show

∠M⌵ = ˆ

The ^ acts as a cursor; it indicates where you are working on the line. You must now enter the address you want to examine, starting with the most significant digit. You can omit leading zeros. Remember that the digits are hexadecimal (see Table 0–1). If you get lost, start over by pressing RESET or the ESC (escape) key (leftmost key in the next-to-top row).

For example, enter the four-digit address 0200. New digits appear at the right as you would expect. Now press RETURN (rightmost key in the next-to-top row). Remember that the display is hexadecimal and that memory addresses are four digits long, whereas data entries are two digits long.

If you have done everything correctly, the AIM will display the address at the left and the contents of that address and the next three higher addresses at the right. A typical example is

```
∠M⌴ = 0200   1F  2F  BE  9D
```

This means

1. Address 0200 contains 1F.
2. Address 0201 contains 2F.
3. Address 0202 contains BE.
4. Address 0203 contains 9D.

Use Table 0–1 if you need help with hexadecimal digits.

Since 0200 through 0203 are in RAM, their values are arbitrary. RAM loses its contents when power is removed and could start in any state whatsoever. Such a memory is said to be *volatile*. Anything you put in RAM will be lost when you turn the AIM off.

Thus you can examine a memory location as follows:

1. Press ESC or RESET to obtain the prompt (∠).
2. Press M.
3. Enter its address as four hexadecimal digits starting with the most significant digit. You may omit leading zeros.
4. Press RETURN.

The contents of the location and the next three higher addresses will appear on the display. If you enter the address incorrectly, simply press ESC to start over. If you get totally confused, press RESET.

PROBLEM 0–1

Examine address 0038 (hex).

PROBLEM 0–2

Examine address ECA2 (hex). Its contents should be BF. We know what it contains, since it is in the nonvolatile read-only memory. You can look up the contents of ROM addresses in the *AIM 65 Monitor Program Listing*.

Note the following special features of the AIM display:

1. B and D have tails at the top and bottom, so you can tell them from 8 and 0, respectively.
2. 6 has a bar at the top and 9 has one at the bottom to make them easier to read.
3. Letters are generally wider than digits. In particular, the letters O and S are twice as wide as the digits 0 and 5, respectively. Thus you can tell them apart without much trouble.
4. I is a capital letter, so you can easily distinguish it from 1 (one).
5. Many letters (particularly B, C, D, K, P, Q, R, S, and V) look strange. There are no curved lines, and letters vary in size and spacing. The overall effect is primitive but consistent, as if someone were using a machine to imitate a child's printing.

As with calculator displays, the AIM's odd-looking letters and numbers will become familiar with practice.

Once you have examined a set of four memory addresses, you can move on to the next four higher addresses by pressing the space bar. Try this and see what happens. The M disappears, but everything else looks the same, except that the address is 4 larger. Examine memory locations 0200 through 0220. You can move up through memory but not down. To stop examining memory, press ESC.

CHANGING MEMORY

You can also change the locations you are examining. First, press the / key (next-to-last key in the row just above the space bar). For example, assume that you have just examined locations 0200 through 0203. You can change 0200 to 75 by pressing

```
/   (the display shows ∠/∆0200^)
7   (the display shows ∠/∆0200 7^)
5   (the display shows ∠/∆0200 75^)
RETURN
```

The display goes blank except for the prompt. To verify the change, reexamine 0200. Note that you can change all four locations just by entering more data as the cursor moves right. To leave a location unchanged, press the space bar, and the cursor will skip over it. When you have changed or skipped the fourth location, the display will go blank.

However, you can continue at the next higher address by simply pressing the / key again.

So you can change a memory location (after examining it) as follows:

6. Press /.
7. Enter the data as two hexadecimal digits, starting with the more significant digit.

If you enter the data incorrectly, press ESC and / to start over. The address at the left will be the same as before.

Note that the monitor does not actually change memory until you enter the second digit. Check this by entering F2 into location 0200; verifying that it is there; pressing 0, ESC; and then examining 0200 again. It should still contain F2; the 0 was never actually stored in memory.

PROBLEM 0–3

Enter the following data into 0200 through 0202:

Memory Address (hex)	Memory Contents (hex)
0200	A9
0201	6A
0202	00

Verify the values after you enter them.

PROBLEM 0–4

Try changing address ECA2 to A4. What do you find when you examine it again? Remember that ECA2 is in read-only memory, not in read/write memory.

RUNNING A PROGRAM

To run a program, press ESC or RESET to restore the prompt. You must first tell the AIM where the program begins by pressing * (second key from the right in the top row). Note that * is an uppercase character, so you must press SHIFT to type it, just as if you wanted a capital letter on a typewriter. Enter the starting address as four hexadecimal digits, omitting leading zeros, and press RETURN. If you now press G and RETURN, the AIM will run your program.

A simple example is the single instruction BRK (FORCE BREAK), which makes the AIM return control to the monitor. We can enter and run this program as follows.

1. Press ESC or RESET to get the prompt.
2. Press M to examine memory.

3. Press 2,0,0, RETURN to examine address 0200. We will start (and finish) our program there. Note that we have omitted the leading zero.
4. Press /,0,0, RETURN to put 00 in 0200. 00 is the hexadecimal version of BRK; you can look it up on your programming card or in Table A1-1 of this book.
5. Press *,2,0,0, RETURN to make the starting address 0200.
6. Press G, RETURN to run the program.

What happens? All the computer does is display 0201 and some other information that we will discuss later. We do not know whether anything happened, except that the AIM did not wander off (aimlessly!). We will present more impressive programs with actual results in Laboratory 1.

PROBLEM 0–5

Enter and run BRK in 022A. What does the display show after you run it?

PROBLEM 0–6

Enter and run BRK in ECA2. What happens and why? Was BRK ever stored in memory?

KEY POINT SUMMARY

1. The AIM 65 has a monitor program stored in read-only memory (ROM). This memory is nonvolatile, and you cannot change it.
2. You can initialize the AIM and transfer control to its monitor by pressing the RESET button.
3. The AIM has read/write memory (RAM) in addresses 0000 through 03FF. This memory is volatile (its contents change when power is lost), and you can change it.
4. Some read/write memory is reserved, either by the monitor or by the microprocessor. We will use addresses 0200 through 03FF for programs and data and 0000 through 00DE for data only.
5. Each memory location has a 16-bit address (four hexadecimal digits); its contents are an 8-bit number (two hexadecimal digits).
6. You can examine four successive memory addresses by pressing the M key, entering the lowest address, and pressing the RETURN key. The contents of the four locations appear on the display from left to right. You can then change them by pressing / and entering new data (or spaces if you want to leave locations unchanged). You can continue examining successively higher addresses by pressing the space bar, and you can continue changing memory by pressing /. These procedures let you examine memory and put programs and data in it.
7. You can make the AIM run a program by entering its starting address (pressing *, address, RETURN) and then pressing G and RETURN.

WRITING AND RUNNING

SIMPLE PROGRAMS

PURPOSE

To learn how to write, load, and run simple programs.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 40–63, 72–104.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 4.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 95–99 (computer words), 99–101 (instruction words), 112–119 (hardware and software), 163–164 (program counter), 165–166 (accumulator), 310–311 (microprocessor instruction sets),

311–316 (6502 registers), 317–322 (6502 instruction set and addressing modes), 341 (memory-register transfers).

W. J. WELLER, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapters 2–5.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, Chapter 2.

WHAT YOU SHOULD LEARN

1. How to load programs into memory.
2. How to determine the length of instructions.
3. How to place addresses in instructions.
4. How to examine the results of programs.
5. How to make the 6502 do simple arithmetic and logic.
6. How to examine registers.
7. How to change registers.

TERMS

Absolute addressing an addressing mode in which the instruction contains the actual address required to execute it. In 6502 terminology, absolute addressing refers to direct addressing with 16-bit addresses.

Accumulator a register that holds one operand and, subsequently, the result in most arithmetic and logical operations.

Addressing modes the methods for specifying the addresses used to execute an instruction. Common modes include direct, immediate, indexed, and relative.

Assembler a program that converts assembly language programs into a form (machine language) that a computer can execute directly. The assembler translates mnemonic operation codes and names into their numerical equivalents and assigns locations in memory to data and instructions.

Assembly language a computer language that allows the programmer to use mnemonic operation codes, labels, and names to refer to their numerical equivalents.

Comment part of a program that has no purpose other than documentation. Comments are neither translated nor executed; they are simply copied into the program listing.

Direct addressing an addressing mode in which the instruction contains the actual address required to execute it. The 6502 has two types of direct addressing: zero-page (requiring only an 8-bit address on page 0) and absolute (requiring a full 16-bit address).

Logical shift a shift that fills vacated bits with zeros.

Machine language the programming language that the computer can execute directly with no translation other than numerical conversions.

Mnemonic name that suggests something's purpose or function.

Operation code (op code) part of an instruction that tells what operation it performs.

Page a subdivision of memory. In 6502 terminology, a page consists of 256 bytes with the same 8 most significant address bits; for example, page C6 consists of addresses C600 through C6FF.

Page number the identifier for a memory page; in 6502 terminology, the 8 most significant bits of a memory address.

Program counter a register that contains the address of the next instruction to be fetched from memory.

Register a storage location inside the CPU.

Zero-page addressing a type of direct addressing in which the instruction contains only an 8-bit address on page 0. That is, page 0 is implied.

6502 INSTRUCTIONS

AND logically AND the accumulator with a memory location.

ASL arithmetic shift left; shift the accumulator or a memory location left 1 bit and clear the least significant bit (see Figure 1–1).

EOR logically EXCLUSIVE OR the accumulator with a memory location.

LDA load accumulator from a memory location.

LDX(Y) load index register X(Y) from a memory location.

LSR logical shift right; shift the accumulator or a memory location right 1 bit and clear the most significant bit (see Figure 1–1).

ORA logically (INCLUSIVE) OR the accumulator with a memory location.

STA store accumulator in a memory location.

STX(Y) store index register X(Y) in a memory location.

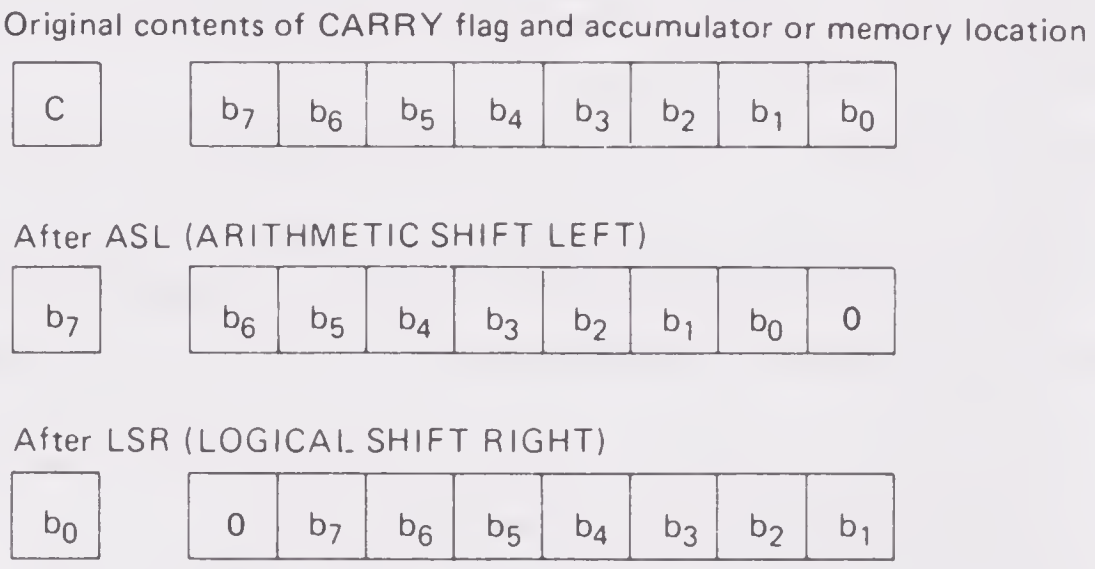


FIGURE 1–1. 6502 shift instructions ASL and LSR.

DATA TRANSFER PROGRAM

Our first program simply moves the contents of memory location 0040 (hex) to 0041. The computer here provides the equivalent of an electrical connection between the two locations.

The program is

```
LDA  $40  ;GET DATA
STA  $41  ;MOVE DATA
BRK       ;RETURN TO MONITOR
```

We are using the format of the AIM 65 assembler (see Figure 1–2); $ in front of a number means "hexadecimal." The comments (preceded by semicolons) act only as documentation and do not affect the program. Figure 1–3 is a programming model of the 6502 microprocessor.

Before a number:

$ - hexadecimal
% - binary
@ - octal

The default case (i.e., unmarked) is decimal.

Other symbols:

- immediate addressing
, - between a base address and the designation of an index register (X or Y)
' - before an ASCII character
; - before a comment

A space is required after a label and after an operation code. Parentheses around an address indicate that it is to be used indirectly (i.e., it contains an address rather than the actual data). The default addressing mode is absolute (direct).

FIGURE 1–2. Format for the AIM 65 assembler.

Let us now look at each instruction in detail:

1. LDA $40 loads the accumulator with the contents of memory location 0040. The $ means "hexadecimal," and the leading zeros can be omitted as in common practice. Remember, the address is four hexadecimal digits (16 bits) long, but the data stored there is two digits (8 bits) long.
2. STA $41 stores the accumulator in location 0041. Here again, the address is four digits long, whereas the data is two digits long.
3. BRK returns control to the AIM's monitor. You should put it at the end of every program.

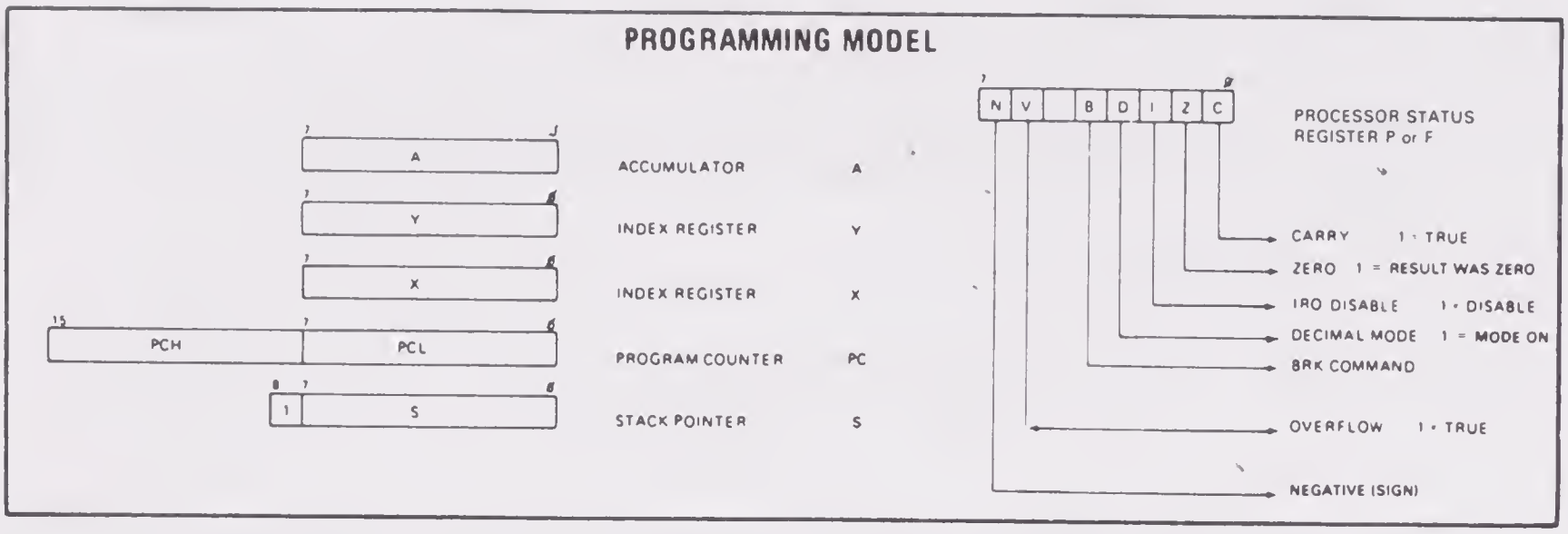


FIGURE 1–3. Programming model of the 6502 microprocessor.

One way to load a program into memory is to look up the hexadecimal operation codes on the Programming Reference Card (see Table A1–1). We could then enter the program in hexadecimal as we did in Laboratory 0. Note that most instructions have different operation (op) codes for different addressing modes. Each operation code is followed by two numbers: the one under n indicates how many clock cycles the instruction takes to execute, while the one under # indicates how many bytes of memory it occupies. We will explain the addressing modes as we use them.

A handy feature of the AIM is that it will translate mnemonic operation codes into hexadecimal numbers. The monitor's Instruction Mnemonic Entry (I) command lets us enter programs in assembly language (i.e., referring to the instructions by name) rather than in machine language (i.e., numbers only). Assembly language is obviously much easier to read and remember than machine language, since it is based on meaningful names (*mnemonics*) rather than on arbitrary numbers.

You can enter programs using mnemonics as follows:

1. Press I. The AIM will respond with the current starting address. To change it, type * (uppercase) followed by the address you want and RETURN.
2. Once you have the correct starting address, enter the three-letter operation code for the first instruction. If an operand is needed, enter it in hexadecimal according to a format from Table 1–1. When you complete the instruction and press the space bar, the AIM will first display the program counter, hexadecimal operation code, and instruction. It will then display the program counter again, this time along with the machine language version of the instruction. Finally, it will display the next available address, and you may continue entering instructions or exit by pressing ESC.

Mnemonic entry works well only if you turn the printer on (by pressing CTRL and PRINT simultaneously). With the printer off, the output appears only briefly on the display; this is difficult to follow unless you have quick reactions and a photographic memory. Remember to turn the printer off when you no longer need it.

If an instruction (such as BRK) does not require an operand, the AIM will take control as soon as you enter the third letter. It will automatically display the program counter, hexadecimal operation code, and instruction. You may then proceed as usual.

The AIM indicates undefined operation codes or improper operands with a simple ERROR message. This usually means that you made a typing mistake or one of the common errors we will describe later. If you make a mistake before completing an operation code or address, you can correct it immediately. Pressing DEL (rightmost key in the middle row) erases the last character and moves the cursor left. Note that DEL works only if you have not completed an entry; it will not, for example, move the cursor back if you have finished a three-letter operation code (say, typed LDX instead of LDY). Section 3.5.1 of the *AIM 65 User's Guide* describes several ways to correct mistakes.

TABLE 1–1 MNEMONIC ENTRY FORMATS FOR ADDRESSING MODES

Addressing Mode	Operand Format
Immediate	#HH[1]
Zero-page	HH
Zero-page, X	HH, X
Zero-page, Y	HH, Y
Accumulator	A
Absolute	HHHH
Absolute, X	HHHH, X
Absolute, Y	HHHH, Y
Relative[2]	HH *or* HHHH
Indexed indirect	(HH, X)
Indirect indexed	(HH), Y
Absolute indirect	(HHHH)

[1]H represents a hexadecimal digit; a leading zero may not be omitted.

[2]For conditional branches, you may enter the displacement from the program counter as a two-digit relative offset or as a four-digit absolute address. In the second alternative, the monitor computes the relative offset automatically.

Program 1–1 contains both hexadecimal (machine language) and mnemonic-entry versions of the data transfer program. Mnemonic entry differs from AIM 65 assembly language as follows:

1. You need not indicate that a number is hexadecimal. In fact, no other number systems are allowed. Thus, for example, you must enter address 0040 as 40, not $40. Entering $40 will result in an error message.
2. You cannot omit leading zeros except when the result is a valid format. Thus you must enter 0 as 00 and 200 as 0200. You can, however, omit the page number from 0040 and enter it as 40. Entries such as #0 and 20F will cause error messages.

PROGRAM 1–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A5	LDA 40
0201	40	
0202	85	STA 41
0203	41	
0204	00	BRK

Notc thc following in Program 1–1:

1. Wc havc used zcro-page (direct) addressing for both LDA and STA; the bytc of memory following the operation code contains thc addrcss. This address is actually 16 bits long, but if its 8 most significant bits are all zeros, we can omit them and use zcro-pagc addressing. This is like the common practice of referring to an interval of less than a minute as, for example, "20 seconds" rather than as "zcro minutes and 20 seconds." If thc 8 most significant bits of thc addrcss are not all zeros, we cannot omit them and must use absolutc (direct) addressing.
2. In zero-pagc addressing, the second bytc of thc instruction contains an address. Notc thc LDA $40 means "load the accumulator from addrcss 0040." That location could contain any 8-bit number; it need not contain 40.
3. Thc instructions vary in length. LDA $40 and STA $41 rcquirc two bytes of mcmory, whereas BRK requires only onc.

Let us now run Program 1–1 with 6C in 0040. The answer should be 6C in 0041. You should clear 0041 before running thc program to prove that the computcr is actually doing something.

ENTERING AND RUNNING THE DATA TRANSFER PROGRAM

To enter Program 1–1 and the test data, and then run thc program, proceed as follows:

Enter Program

1. Prcss RESET or ESC if the prompt is not showing. If the printer is off, turn it on by prcssing CTRL and PRINT simultaneously.
2. Start mnemonic entry at address 0200 with thc key sequence

```
I
*
2
0
0
RETURN
```

3. Enter the program in mnemonic form with the key sequence

```
L
D
A
4
0
SPACE
S
T
A
4
1
SPACE
B
R
K
ESC
```

You can verify the entries by comparing the printed machine language instructions with the memory contents in Program 1–1.

Enter Data

1. Examine 0040 with the key sequence

```
M
4
0
RETURN
```

2. Enter the data (6C) and clear 0041 with the key sequence

```
/
6
C
0
0
ESC
```

Note that you must enter the test data into memory in addition to the program.

Run Program

You can now execute the program as follows:

1. Establish the starting address with the key sequence

```
*
2
0
0
RETURN
```

2. Make the computer run the program by pressing

```
G
RETURN
```

Remember, the program starts in 0200. The final RETURN transfers control to it; control returns to the monitor when the computer executes BRK.

Examine Results

Finally, you can examine the data and the result (after running the program) with the key sequence

```
M
4
0
RETURN
```

Remember, the program stores the result in 0041. The computer does not tell you the answer (regardless of what some fiction writers think). All the computer does is execute the program (which takes about 6 microseconds) and return control to the monitor (since you put BRK at the end).

PROBLEM 1–1

Run Program 1–1 with the following data:

a. F0
b. 28

PROBLEM 1–2

Revise Program 1–1 to do the following:

a. Store the result in 0042.
b. Move the contents of 0041 to 0040.

PROBLEM 1–3

Revise Program 1–1 to use index register X instead of the accumulator. How would you make it use register Y?

PROBLEM 1–4

Revise Program 1–1 to transfer data from 0380 (hex) to 0381 (hex). Now you must use absolute addressing instead of zero-page addressing. Note in the machine language output that 2-byte absolute addresses are loaded upside down; the order in a 3-byte instruction is operation code, less significant address byte, and more significant address byte. LDA $0380, for example, is translated into AD (LDA with absolute addressing), 80 (less significant address byte), and 03 (more significant address byte).

PROBLEM 1–5

Write and run a program that moves the contents of location 0040 to 0042 and the contents of 0041 to 0043.

Sample Problem

Data: (0040) = C6
(0041) = 5E
Result: (0042) = C6
(0043) = 5E

How much longer is the program if the addresses are 0380 through 0383 instead of 0040 through 0043?

PROCESSING DATA

Of course, we usually want to process the data rather than just move it around in memory. For example, the following program shifts each data bit left one position and clears the

least significant bit before storing the result in 0041. The computer here mimics an 8-bit shift register. The program is

```
LDA  $40  ;GET DATA
ASL  A    ;SHIFT DATA LEFT
STA  $41  ;STORE RESULT
BRK       ;RETURN TO MONITOR
```

The only new instruction is ASL A, which shifts the accumulator left 1 bit and clears the least significant bit. Note that ASL A is a 1-byte instruction; it does not require an address, since the data is in the accumulator.

Run this program (Program 1–2 contains hexadecimal and mnemonic-entry versions) with the data 01 (00000001 binary) in 0040. The result should be 02 (00000010 binary) in 0041. Why? Use Table 0–1 to convert hexadecimal numbers to binary, and vice versa. For example, 01 hex is 00000001 binary since 0 hex is 0000 binary and 1 hex is 0001 binary. Going the other way, 00000010 binary is 02 hex since 0000 binary is 0 hex and 0010 binary is 2 hex. You must split the byte down the middle to form two hexadecimal digits.

PROGRAM 1–2

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A5	LDA 40
0201	40	
0202	0A	ASL A
0203	85	STA 41
0204	41	
0205	00	BRK

PROBLEM 1–6

Run Program 1–2 with the following data:

a. 40. The result should be 80.
b. C7. The result should be 8E. What happens to the 1 that is originally at the far left?

PROBLEM 1–7

Can you revise Program 1–2 to use an index register? Are there instructions that shift the index registers? The accumulator is the only register we can use for data processing.

PROBLEM 1–8

Make Program 1–2 shift the data right instead of left. (*Hint:* Replace ASL A with LSR A.)

Sample Problems

The parentheses around a memory address indicate "contents of."

a. (0040) = 02
   Result: (0041) = 01
b. (0040) = C7
   Result: (0041) = 63

What happens to the 1 that is originally at the far right?

PROBLEM 1–9

Revise Program 1–2 to shift the contents of two successive locations and store the results in the next two locations. That is, the new program should store the shifted version of 0040 in 0042 and the shifted version of 0041 in 0043.

Sample Problem

Data: (0040) = 01
      (0041) = 08
Result: (0042) = 02
        (0043) = 10

LOGICALLY ANDING TWO VALUES

We can easily convert the left-shift program into a logical AND program. The task now is to logically AND the contents of 0040 and 0041 and place the result in 0042. Here the computer mimics two 7408 quadruple two-input TTL AND gates. Note, however, that a 6502 can perform many different arithmetic and logical functions. The logical AND program is

```
LDA  $40     ;GET FIRST OPERAND
AND  $41     ;LOGICALLY AND SECOND OPERAND
STA  $42     ;STORE RESULT
BRK
```

Program 1–3 is the mnemonic-entry version. We have started it in 0210, so we can leave Program 1–2 in memory for later use. Remember that AND works as follows on each bit:

Input 1	Input 2	Input 1 AND Input 2
0	0	0
0	1	0
1	0	0
1	1	1

PROGRAM 1–3

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0210	A5	LDA 40
0211	40	
0212	25	AND 41
0213	41	
0214	85	STA 42
0215	42	
0216	00	BRK

Enter Program 1–3 into memory and run it for the following sample cases. Remember to start at 0210, *not* at 0200. Use Table 0–1 to convert hexadecimal values into binary to check your results.

a. Data: (0040) = 23
(0041) = 34
Result: (0042) = 20

b. Data: (0040) = F0
(0041) = 3B
Result: (0042) = 30

c. Data: (0040) = 0C
(0041) = 77
Result: (0042) = 04

PROBLEM 1–10

Make Program 1–3 logically OR two locations and store the result. How would you get a logical EXCLUSIVE OR? Refer to Table 3–3 if you cannot remember how logical functions work.

Sample Problem

Data: (0040) = 23
(0041) = 34
Result: (0042) = 37 for logical OR
(0042) = 17 for logical EXCLUSIVE OR

EXAMINING REGISTERS

One way to determine what a program is doing is to examine the processor's registers. On the AIM, you can examine registers by pressing the R key. With the printer on, you will see the following line of identification codes:

```
****   PS   AA   XX   YY   SS
```

You will then see the current register values in the following order from left to right:

```
PC   P   A   X   Y   S
```

PC is the program counter (a 16-bit register), P is the processor status register, and S is the stack pointer.

If the printer is off, the identification line disappears and you must use Table 1–2 to figure out what the values mean. The program counter is obvious, since it is the only four-digit register. Taping a copy of the identification line to the AIM will help you identify the other registers. You can exit from the register display by pressing ESC.

We have not yet discussed some registers. We will describe the status register in Laboratory 2 and the stack pointer in Laboratory A. The program counter contains the address of the next instruction that the CPU will fetch from memory. Each time the CPU uses the program counter, it adds 1 to its contents. Thus the computer will execute instructions sequentially unless it is specifically told to do otherwise. Since the program counter is 16 bits long, it can hold a complete memory address.

TABLE 1–2 ORDER IN REGISTER DISPLAY (LEFT TO RIGHT)

Designation	Meaning
****	Program counter (four digits)
PS	Status (P) register (flags)
AA	Accumulator
XX	Index register X
YY	Index register Y
SS	Stack pointer

CHANGING REGISTERS

You can change a register by entering either two or four (program counter only) hexadecimal digits after pressing one of the following keys:

1. * for program counter (remember, * is uppercase). You must type RETURN after completing the entry.
2. A for accumulator.
3. X for index register X.
4. Y for index register Y.
5. S for stack pointer.
6. P for processor status register.

For example, say we want to put 4C in index register Y. All that we must do is press Y, 4, C. The AIM will load 4C into Y; we can verify this by pressing R.

There are two things to watch here. First, note that you cannot change the register display directly (that is, by pressing /), nor can you change more than one register with a single command. Second, the program counter is special. It can accept a four-digit entry, and you can omit leading zeros. However, you must press RETURN after changing it. With the other registers, you cannot omit a leading zero, but you need not press RETURN. The AIM takes control as soon as you enter the second digit.

Run Program 1–2 with (0040) = C7. What are the final contents of the accumulator and program counter? Does it matter if you clear the accumulator initially (i.e., load it with 00)?

PROBLEM 1–11

Run Program 1–3 with (0040) = 23 and (0041) = 34. What are the final contents of the accumulator and program counter? Does it matter if you clear the accumulator initially? What happens to values you load into the index registers before running Program 1–3? Try different values (e.g., 00, FF, AA, 55) and see if the results vary. What happens if you reset the computer after loading the index registers? What happens if you enter 6F into the stack pointer (S register) and then press RESET?

COMMON OPERATING ERRORS

By now you have undoubtedly discovered the following common errors that plague the AIM user:

1. Forgetting to enter operands properly in mnemonic entry. You must enter operands as two or four hexadecimal digits with no leading $; you cannot omit leading zeros. Typical problem cases are LDA #0 (you must enter 00, not just 0) and LDA $206 (you must enter 0206, not 206 or $206).

2. Forgetting to press M before examining memory or / before changing it. If you begin with the address or data, you will confuse the AIM completely and start a whole series of problems. The AIM will think the first digit is a command.
3. Trying to change the register display. You must press a command key to change a register, and you can change only one at a time.
4. Entering a new command before completing the current one. The AIM is remarkably tolerant of this and will often proceed correctly with the new command. Sometimes, however, the AIM cannot figure out what you want. If, for example, you try to examine the registers during mnemonic entry, the AIM will probably think R is the first letter of an operation code. To avoid this problem, always press ESC and restore the AIM's prompt before entering a command.
5. Misinterpreting data as instructions. For example, you may set the starting address to 0040 instead of 0200. This mistake causes the AIM to execute the data as if it consisted of instructions. The way to avoid this is to keep programs and data clearly separated in your mind and in all your work.
6. Forgetting to enter a starting address. You must enter a starting address before running a program or loading one using mnemonic entry.
7. Forgetting to run the program—that is, entering the program and the data and waiting for something to happen. This is like entering data into a calculator and waiting for it to produce a result. Neither a computer nor a calculator will do anything until directed to run a program.
8. Starting program execution at the wrong address. The computer will execute whatever it finds at the address you specify. This is a common error if you have several programs in memory or if you vary the starting address. The way to ensure that you have the correct address is to mark it on each program listing.

Errors will often make the AIM lose its way and never return to the monitor. If this happens, press RESET. Always check your program and data before proceeding; the computer will probably have changed them (unless you are much luckier than I am).

KEY POINT SUMMARY

1. Most simple 6502 programs use the accumulator as the center of operations. They begin by loading the accumulator from memory and end by storing the result (from the accumulator) in memory.
2. The most straightforward addressing mode is direct addressing, in which the instruction contains the address it needs. This address follows the operation code in memory.
3. If the 8 most significant bits of a direct address are all zeros, we can omit them and use zero-page addressing. This makes page 0 special, since instructions that use it occupy less memory and execute faster than instructions that use other pages.

4. If the 8 most significant bits of a direct address are not all zeros, we cannot omit them and must use absolute addressing. An absolute address occupies 2 bytes of memory, with the 8 most significant bits in the second byte. That is, the address is stored upside down.
5. To make the computer do something useful, you must enter the program and data into memory, run the program, and examine the results. The I (mnemonic entry) command lets you enter programs in simplified assembly language rather than as hexadecimal numbers (machine language).
6. The 6502's accumulator is special, because it is the only register that can be used in most arithmetic and logical operations.
7. You can examine the 6502's registers by pressing the R key. The AIM displays identification codes and register values in the following order from left to right: program counter (four digits), status (P) register, accumulator, index register X, index register Y, stack pointer.
8. You can change a register by pressing one of the following keys: * for the program counter, A for the accumulator, X for index register X, Y for index register Y, S for the stack pointer, and P for the processor status register. The program counter requires an address (four digits, but leading zeros can be omitted), followed by RETURN. The other registers require two-digit entries; a leading zero cannot be omitted, but no RETURN is necessary.
9. Common errors in operating the AIM include using mnemonic entry incorrectly, forgetting to press M or / before examining or changing memory, attempting to change the register display, starting one command in the middle of another, failing to enter a starting address, misinterpreting data as instructions, forgetting to run the program, and starting program execution at the wrong address.

SIMPLE INPUT

PURPOSE

To learn how to use the computer's input ports.

PARTS REQUIRED

Eight switches attached to the Application Connector as shown in Figure 2–1. Table 2–1 contains the pin assignments.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 72–104, 369–370.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11–1 to 11–12, 11–39 to 11–49.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 26 (hexadecimal arithmetic), 31–32 (AND function), 62–63 (arithmetic circuits), 169–171 (status register), 174–180 (arithmetic-logic unit), 203–206 (simple input port), 312–316 (6502 flags), 319 (immediate addressing), 338–340 (compare and bit test instructions), 346–355 (conditional branch instructions).

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, pp. 6–1 to 6–19.

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1979, Chapters 2–5.

WHAT YOU SHOULD LEARN

1. How the 6502 identifies I/O ports and which instructions are commonly used for I/O.
2. How to determine whether a switch is open or closed.
3. Which bit positions can be accessed most easily.
4. How to examine the flags.
5. How to handle a series of switch closures.
6. How to recognize a starting (synchronization) character.

TABLE 2–1 APPLICATION CONNECTOR PIN ASSIGNMENTS FOR PORT A OF THE USER VIA

Assignment	Pin
Bit 0 (PA0)	14
Bit 1 (PA1)	4
Bit 2 (PA2)	3
Bit 3 (PA3)	2
Bit 4 (PA4)	5
Bit 5 (PA5)	6
Bit 6 (PA6)	7
Bit 7 (PA7)	8

TERMS

Branch instruction see *Jump instruction*.

Carry flag a flag that is 1 if the last operation generated a carry from the most significant bit and 0 if it did not.

Flag a bit that indicates a condition within the computer, often used to choose between alternative instruction sequences.

Immediate addressing an addressing method in which the instruction contains the data it requires, usually immediately following the operation code in memory.

Jump instruction (or **branch instruction**) an instruction that places a new value in the program counter, thus departing from the normal instruction sequence. A conditional jump instruction places the new value in the program counter only if a condition is true.

Label a name attached to an instruction or statement in a program. The name takes on the value of the starting address in memory of the resulting machine language or assignment.

Masking singling out 1 or more bits from a group of bits.

Memory-mapped input/output assigning I/O ports to memory addresses rather than giving them their own set of addresses.

Negative flag (or **sign flag**) a flag that contains the most significant bit of the result of the previous operation.

Port the basic addressable unit of the computer's input/output section.

Relative addressing an addressing method in which the instruction contains the offset from a base address.

Relative offset the difference between the address to be used in an instruction and the current program counter.

Relocatable able to be placed anywhere in memory without changes; that is, capable of occupying any set of consecutive memory addresses.

Sign flag see *Negative flag*.

Status register a register that contains bits (flags) describing the current state of the computer.

Synchronization (sync) character a character that is used only to synchronize the transmitter and the receiver. The character does not contain any actual information.

Zero flag a flag that is 1 if the last operation produced a zero result and 0 if it did not.

6502 INSTRUCTIONS

The following branch instructions all jump over the specified number of memory locations if the specified condition is true; otherwise, they proceed to the next instruction in sequence.

BCC branch if carry clear.
BCS branch if carry set.
BEQ branch if equal to zero (ZERO flag = 1).
BMI branch if minus (NEGATIVE flag = 1).
BNE branch if not equal to zero (ZERO flag = 0).

BPL branch if plus (NEGATIVE flag = 0).
BVC branch if overflow clear.
BVS branch if overflow set.

BIT bit test; logically AND the accumulator with a memory location but leave the accumulator unchanged. This instruction affects only the flags. BIT sets the NEGATIVE flag from bit 7 of the memory location and the OVERFLOW flag from bit 6 without considering the accumulator. The only flag that depends on the logical AND is the ZERO flag. BIT allows only zero-page and absolute (direct) addressing.

CMP compare memory and accumulator; subtract a memory location from the accumulator but leave the accumulator unchanged. This instruction affects only the flags.

6502 INPUT/OUTPUT INSTRUCTIONS

The 6502 has no specific input/output (I/O) instructions. Instead, it treats I/O ports as memory locations. In this approach, called *memory-mapped input/output*, any instruction that transfers data to or from memory can perform I/O. The 6502 instructions most often used for I/O are:

- LDA loads the accumulator from an input port.
- STA stores the accumulator in an output port.
- BIT (BIT TEST) sets the flags as if the data from an input port had been logically ANDed with the accumulator. The accumulator does not change.
- CMP (COMPARE MEMORY AND ACCUMULATOR) sets the flags as if the data from an input port had been subtracted from the accumulator. The accumulator does not change.

SIMPLE INPUT

The AIM 65 has I/O ports in its on-board 6522 Versatile Interface Adapters (VIAs). Throughout this book, we will employ user VIA port A (address A001) for input and port B (address A000) for output. Starting from reset, the following program loads the accumulator from port A:

```
LDA  $A001
BRK
```

The mnemonic-entry version is Program 2–1. Open all the switches attached to port A, reset the AIM, and execute Program 2–1. What does the accumulator contain afterward?

Close the switch attached to bit 5 and execute Program 2–1 again. Now what is in the accumulator?

PROGRAM 2–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	AD	LDA A001
0201	01	
0202	A0	
0203	00	BRK

PROBLEM 2–1

The computer interprets an open switch as ____ and a closed switch as ____ if the connections are as shown in Figure 2–1.

FIGURE 2–1. Attachment of switches to the Application Connector. The user VIA is device Z1 in the AIM schematics.

PROBLEM 2–2

Determine what value Program 2–1 places in the accumulator if:

a. The switch attached to bit 2 of port A is closed.
b. Switches attached to bits 2 and 5 are closed.
c. Switches attached to bits 0,6, and 7 are closed.

Assume that all other switches are open.

PROBLEM 2–3

What happens if you replace LDA $A001 with LDA $A000? Does opening or closing switches affect the input? Explain the result.

Remember the following:

1. The standard in the computer industry is to number bit positions starting with 0 at the far right. Thus the bits in a byte are numbered 0 through 7 from right to left; bit 0 is least significant and bit 7 most significant. Figure 2–2 (shown later in this chapter) is an example of the standard numbering. Be careful—switches and other I/O devices often use other conventions (e.g., 1 to 8 or left to right).
2. Since A001 is not on page 0, we must use absolute addressing to refer to it. As you can see in Program 2–1, the 16-bit address occupies 2 bytes of memory, with its less significant bits first.

FLAGS AND CONDITIONAL BRANCHES

To have the computer determine if a switch is open or closed, we must use the flags and conditional branch instructions. Instructions that move or process data also affect the flags. A conditional branch instruction lets the computer use a flag to choose between alternative paths through a program.

The major 6502 flags are:

C (CARRY) 1 if the last arithmetic or shift instruction produced a carry, 0 if it did not.
N (NEGATIVE or SIGN) 1 if the result of the last instruction had a 1 in its most significant bit, 0 if it did not.
Z (ZERO) 1 if the result of the last instruction was zero, 0 if it was not zero.

Conditional branch instructions place a new value in the program counter if the specified flag has the specified value. Otherwise, they leave the program counter unchanged and the processor simply continues its normal sequence. Conditional branches make a computer "smart," that is, capable of making decisions based on current

information. The computer thus becomes an intelligent controller. Table 2–2 lists the 6502's conditional branch instructions.

TABLE 2–2 6502 CONDITIONAL BRANCH INSTRUCTIONS

Instruction	Flag Used	Value on Which Branch Occurs
BCC	CARRY	0
BCS	CARRY	1
BNE	ZERO	0
BEQ	ZERO	1
BPL	NEGATIVE (SIGN)	0
BMI	NEGATIVE (SIGN)	1
BVC	OVERFLOW	0
BVS	OVERFLOW	1

WAITING FOR A SWITCH TO CLOSE

Let us now concentrate on the switch attached to bit 5 of port A (switch 5, for short). The following program waits for you to close that switch; it then returns control to the monitor. Remember that an open switch is a 1 and a closed switch is a 0 (see Figure 2–1). Program 2–2 is the mnemonic-entry version.

```
WAITC   LDA  $A001          ; GET INPUT DATA
        AND  #%00100000     ; IS SWITCH 5 CLOSED?
        BNE  WAITC          ; NO, WAIT
        BRK
```

Let us now look at each instruction:

1. LDA $A001 loads the accumulator from port A. WAITC is our name for the memory address in which LDA $A001 begins. Such a name is called a *label*; its sole purpose is to make the program easier for a reader to follow. However, since the mnemonic-entry mode does not allow labels, we must replace them with the actual addresses to which they refer. For example, if Program 2–2 starts at 0200, WAITC is 0200, and BNE WAITC must produce a conditional branch to 0200. The name WAITC is arbitrary; we chose it because it suggests *wait*ing for a closure.
2. AND #%00100000 logically ANDs the accumulator with the binary number 00100000. % means "binary" and # means "immediate" (i.e., the data is in the next byte of memory). The result is 0 if the switch is closed and 00100000 if the switch is open. (Verify this!) Singling out part of a group of bits is called *masking*.
3. BNE WAITC makes the processor execute the instruction in address WAITC next if the ZERO flag is 0. Otherwise, the processor continues sequentially (to BRK in this case). Note that the ZERO flag is 1 if the last result *was zero*.

PROGRAM 2–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	AD	WAITC	LDA A001
0201	01		
0202	A0		
0203	29		AND #20
0204	20		
0205	D0		BNE 0200
0206	F9		
0207	00		BRK

Note the following features of Program 2–2:

1. LDA $A001 uses absolute addressing. The 16-bit address is stored upside down following the operation code.
2. AND #%00100000 uses immediate addressing. The data (00100000 binary = 20 hex) follows the operation code. An instruction with immediate addressing contains the actual data, not its address.

   Although we have written the data in binary to make its purpose clearer, we must enter it in hexadecimal. To convert, we split the binary data in half and use Table 0–1 to convert the halves: 0010 is 2 hex and 0000 is 0 hex.

   Be careful when you enter AND #20. You must type # to indicate "immediate"; it is not like $, which you must omit. Entering AND 20 will not cause an error message but will result in an entirely different instruction. What does AND 20 mean? By the way, # looks strange on the display because the bottom parts of the vertical lines are missing.
3. BNE requires an 8-bit relative offset following the operation code. This offset tells the computer how many locations to jump over from the end of the instruction (address 0207 in this case). A positive offset (most significant bit = 0) is added to the final address (e.g., an offset of 02 would be added to 0207 to make the destination 0209); the maximum positive offset is 7F, or +127 decimal. A negative offset (most significant bit = 1) tells the computer how many locations down to go (down one is FF, down two is FE, etc.). You can calculate the offset by subtracting the address just after the branch from the destination address; in Program 2–2 the subtraction is

$$\begin{array}{rl} & 0200 \text{ (destination address)} \\ - & \underline{0207} \text{ (address immediately following BNE)} \\ & FFF9 \end{array}$$

Only the F9 is significant; the largest negative offset is 80 hex, or −128 decimal.

Hexadecimal subtraction is a nuisance unless you have either a hexadecimal calculator (such as the Texas Instruments Programmer) or 16 fingers. Fortunately, the AIM will calculate the relative offset if you simply enter the destination address after the branch's operation code. For example, you would enter BNE WAITC in Program 2–2 as BNE 0200.

But what if you don't know the destination address? If the branch goes backward, you can see the destination in the printed output. But if the branch goes forward, you must determine the address using the instruction lengths in Table A1–1 (under n). Of course, you can always guess and correct the value later after the AIM prints the actual address.

Enter and run Program 2–2. What happens if you leave switch 5 open? What happens if you close other switches?

PROBLEM 2–4

Make Program 2–2 wait for you to close switch 4 (i.e., the switch attached to bit 4 of A001). Next try switch 2 and then switch 6. How difficult would these changes be to make it TTL logic?

PROBLEM 2–5

Make Program 2–2 start at address 0210. A program that you can put anywhere in memory without changes is called *relocatable*. Is Program 2–2 relocatable? Explain why the relative offset in BNE is important. Would the program be relocatable if BNE contained the actual destination address? Why would you want a program to be relocatable?

SPECIAL BIT POSITIONS

Some instructions and flags make certain bit positions more accessible than others. For example:

1. ASL (see Figure 1–1) shifts each bit left. Bit 6 ends up in the NEGATIVE flag, where it can be used as a branch condition for BMI or BPL.
2. LSR (see Figure 1–1) similarly moves bit 0 to the CARRY, where it can be used as a branch condition for BCC or BCS.
3. LDA places bit 7 in the NEGATIVE flag, where it can be used as a branch condition for BMI or BPL.

So the following program waits for you to close switch 7:

```
WAITC   LDA  $A001    ; GET INPUT DATA
        BMI  WAITC    ; WAIT UNTIL SWITCH 7 IS CLOSED
        BRK
```

No AND is necessary. Program 2–3 is the mnemonic-entry version.

PROBLEM 2–6

Write two programs that wait for you to close switch 0, one using AND and one using LSR. Which program is shorter? Which takes less time to examine the switch and branch?

PROBLEM 2–7

BIT has the odd feature that it sets the NEGATIVE and OVERFLOW flags from bits 7 and 6, respectively, of the memory location without even considering the accumulator. Write a program that uses BIT to wait for you to close switch 6.

PROGRAM 2–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	AD	WAITC	LDA A001
0201	01		
0202	A0		
0203	30		BMI 0200
0204	FB		
0205	00		BRK

If you have only one or two switches (or other serial inputs) to attach to a port, which bit positions should you use for the ones that are read most frequently?

EXAMINING FLAGS

The current values of the flags are in the processor status (P) register (second register from the left in the AIM's register display; see Table 1–2). Figure 2–2 shows the organization of the P register. We will describe the DECIMAL MODE and INTERRUPT DISABLE flags later. Here the hexadecimal display is a nuisance, because only the binary values are meaningful. You can use Table 0–1 to convert hexadecimal to binary.

You can see how an instruction affects the flags by initializing the P register and the operands, letting the computer execute the instruction, and then examining the P register afterward. The result will depend on the instruction and the operands.

If, for example, we start with (A) = 80 hex and (P) = 04 (making the CARRY, NEGATIVE, and ZERO flags all 0), executing AND #$80 makes (P) = B4 hex = 10110100 binary (see Table 0–1). The major flags are:

NEGATIVE (SIGN) = 1 (bit 7 of P)

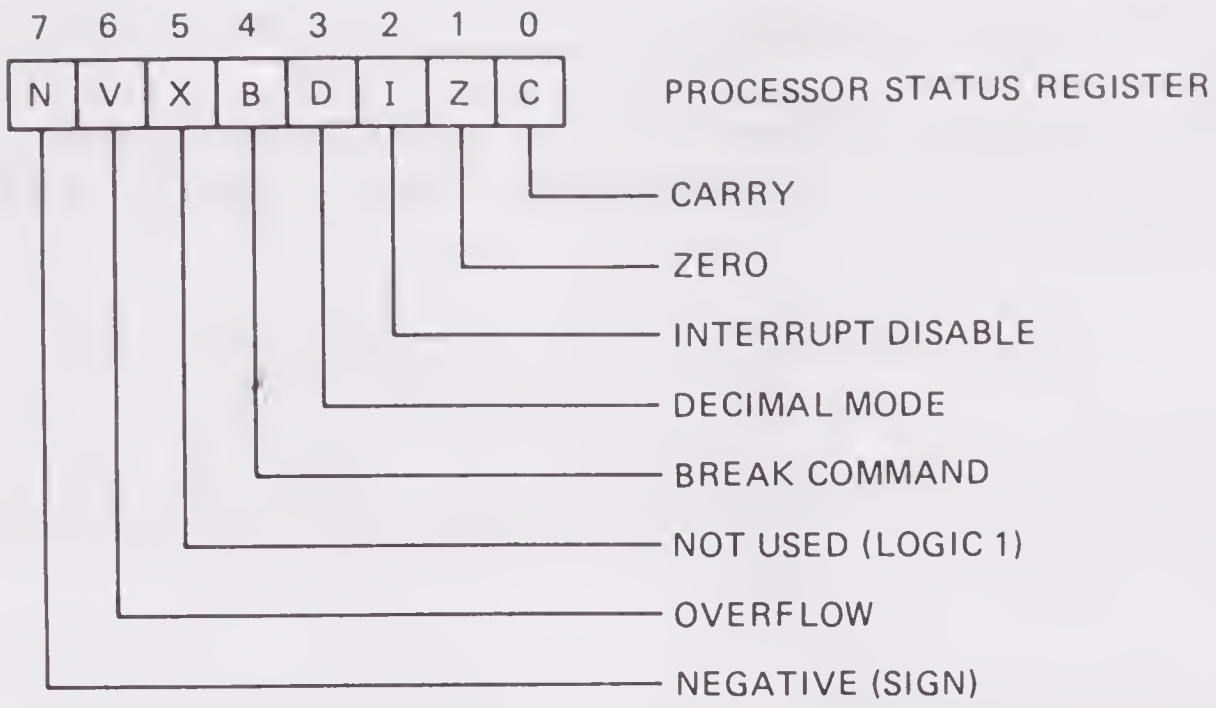


FIGURE 2–2. Organization of the 6502 processor status (P) register. Bit 5 is not used and is always 1.

ZERO = 0 (bit 1 of P)
CARRY = 0 (bit 0 of P) (This is unchanged.)

The result of logically ANDing 80 hex (10000000 binary) with itself is 80 hex. Thus the operation clears the ZERO flag since the result is not 0, and sets the NEGATIVE flag since bit 7 of the result is 1. Logical operations do not affect the CARRY. Remember, you must use the A and P keys (see Laboratory 1) to initialize the accumulator and status register.

PROBLEM 2–8

What are the NEGATIVE, ZERO, and CARRY flags after the processor executes AND #$80 for the following initial conditions?

a. (P) = FF
   (A) = 80
b. (P) = 04
   (A) = 7F
c. (P) = FF
   (A) = 7F

Hint: Use the program:

```
AND #$80
BRK
```

Do the final flag values depend on the initial values?

WAITING FOR TWO CLOSURES

We can easily extend Program 2–2 to wait for two closures. The following program waits for switches 2 and 5 to be closed in that order; assuming that you start with all switches open.

```
WAIT1   LDA  $A001          ; GET INPUT DATA
        AND  #%00000100     ; IS SWITCH 2 CLOSED?
        BNE  WAIT1          ; NO, WAIT
WAIT2   LDA  $A001          ; GET INPUT DATA
        AND  #%00100000     ; IS SWITCH 5 CLOSED?
        BNE  WAIT2          ; NO, WAIT
        BRK
```

PROGRAM 2–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	AD	WAIT1	LDA A001
0201	01		
0202	A0		
0203	29		AND #04
0204	04		
0205	D0		BNE 0200
0206	F9		
0207	AD	WAIT2	LDA A001
0208	01		
0209	A0		
020A	29		AND #20
020B	20		
020C	D0		BNE 0207
020D	F9		
020E	00		BRK

Enter and run this program; the mnemonic-entry version is Program 2–4. What happens if you close switch 2 and then switch 5? What happens if you reverse the order? Explain the result. Does it change if you allow only one switch to be closed at a time?

PROBLEM 2–9

Make Program 2–4 wait for you to close switch 3 followed by switch 1. What happens if you leave one switch closed all the time?

Write a program that waits for a particular sequence of switch closures, and let someone try to guess the sequence. What happens if the other person simply closes all the

switches? You can defeat this strategy by using CMP instead of AND. CMP (COMPARE MEMORY AND ACCUMULATOR) sets the flags as if it had subtracted a memory location from the accumulator. Thus CMP sets the ZERO flag only if its operands are equal. Since the outcome depends on all eight switches, a subsequent BNE will force a branch unless they are all set correctly. For example, after

```
LDA   $A001                ; GET INPUT DATA
CMP   #%00100000
```

the ZERO flag will be 1 only if all eight switches are in the positions specified by CMP's operand (0 = closed, 1 = open).

PROBLEM 2–10

Write a program that waits for you to close switch 0 and then switch 7. Write one version that ignores the other switches and one that works only if all other switches are open. What happens in the second version if you reverse the order (i.e., close switch 7 first and then switch 0) or leave either switch 0 or switch 7 closed all the time?

PROBLEM 2–11

Write a program that waits for you to close switches 2 and 5 at the same time and then switches 0 and 7 at the same time.

PROBLEM 2–12

Write a program that waits for you to close either switch 2 followed by switch 5 or switch 5 followed by switch 2. Allow only one switch to be closed at a time.

SEARCHING FOR A STARTING CHARACTER

In communications applications, the input data is the latest character received. Of course, if the transmitter is inactive, that character will be noise. Assume that the transmitter starts every message with 7F (a synchronization, or sync, character since it is not part of the actual information).

PROBLEM 2–13

Write a program that waits for 7F to appear in A001. An easy way to produce 7F is first to open all switches (producing FF) and then close switch 7.

If the input data is random noise, how often will the computer think it has found a message? That is, what is the probability of the random value being 7F? How often would the computer find a message erroneously if the synchronizing pattern were two 7F characters? How about three 7F characters?

Clearly, a longer synchronizing pattern results in fewer false messages. On the other hand, noise could cause a 7F to be received as something else, and the computer would then miss a real message.

PROBLEM 2–14

Write a program that accepts the input as 7F regardless of the value of bit 2. How often will this program find a message erroneously (i.e., what is its probability of finding a 7F in random data)?

KEY POINT SUMMARY

1. The 6502 microprocessor has no specific I/O instructions. Instead, it addresses I/O ports as memory locations (memory-mapped I/O), and any instruction that transfers data to or from memory can also perform I/O.
2. The AIM has two free I/O ports in its user 6522 VIA. We will use port A (address A001) for input and port B (address A000) for output.
3. The 6502 has three major flags (CARRY, ZERO, and NEGATIVE or SIGN), which are set from the results of certain instructions. Almost all instructions affect the ZERO and NEGATIVE flags, whereas only arithmetic and shift instructions affect the CARRY flag.
4. A conditional branch instruction changes the program counter if the specified condition is true. If the condition is false, the processor continues its normal sequence. Conditional branch instructions are the key to computer decision making.
5. The processor can determine the value of a bit in a register or memory location by logically ANDing the contents with a mask. The mask has a 1 in the specified bit position and 0s elsewhere. The result is 0 if and only if the bit is 0. Bit positions at either end of a byte can be handled by using the SIGN or CARRY flag and the load, shift, or bit test instructions.
6. The processor can determine whether a register or memory location contains a specified value by subtracting the value from the contents. The result is 0 only if the register or memory location contains the value.
7. The processor performs logical operations (AND, OR, EXCLUSIVE OR, NOT) bit by bit, 8 bits at a time. However, arithmetic operations (ADD, SUBTRACT) involve carries or borrows, so the bit positions are not independent.

SIMPLE OUTPUT

PURPOSE

To learn how to use the computer's output ports.

PARTS REQUIRED

Eight LEDs (light-emitting diodes) attached to the Application Connector as shown in Figure 3–1. Table 3–1 contains the pin assignments.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 72–104, 376–377, 413, 414.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, pp. 11–1 to 11–12, 11–39 to 11–42, 11–61 to 11–64.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 31–33 (logic gates), 33–34 (logic equivalences), 200–202 (LED displays), 329–334 (logical instructions), 337–338 (decrement and increment instructions), 368–373 (timing loops).

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, sec. 8.1 to 8.3.

R6500 Microcomputer System Hardware Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, pp. 6–5 to 6–7.

WHAT YOU SHOULD LEARN

1. How to make 6522 I/O lines into inputs or outputs.
2. How to turn LEDs on and off.
3. How to make the computer wait between operations.
4. How to manipulate single bits of data.
5. How to complement (invert) data.
6. How to operate LED displays at a specific duty cycle.
7. How to use software to control displays.

TERMS

Anode positive terminal.

Cathode negative terminal.

Complement see *One's complement*.

Data direction register a register that determines whether I/O lines are inputs or outputs.

Duty cycle the period of time during which a device is active as part of a total period of continuous operation.

Light-emitting diode (LED) a semiconductor device that emits light when its cathode is sufficiently more negative than its anode.

Nesting constructing programs hierarchically with one level contained inside another.

One's complement bit-by-bit inversion, replacing each 0 with a 1 and each 1 with a 0.

Software delay a program that does nothing except waste time.

6502 INSTRUCTIONS

DEC subtract 1 from a memory location. DEC cannot be applied to the accumulator.

DEX(Y) subtract 1 from index register X(Y).

INC add 1 to a memory location. INC cannot be applied to the accumulator.

INX(Y) add 1 to index register X(Y).

JMP jump (transfer control) to a memory address. JMP allows only absolute (direct) or indirect addressing.

LED CONNECTIONS

Attach eight LEDs to user VIA port B as described in Table 3–1 and Figure 3–1. An LED lights when its cathode is sufficiently more negative than its anode. The computer can therefore light an LED either by grounding its cathode or by applying +5V to its anode. Since we have connected the output port to the cathodes, a 0 from the computer lights an LED.

TABLE 3–1 APPLICATION CONNECTOR PIN ASSIGNMENTS FOR USER VIA PORT B

Assignment	Pin
Bit 0 (PB0)	9
Bit 1 (PB1)	10
Bit 2 (PB2)	11
Bit 3 (PB3)	12
Bit 4 (PB4)	13
Bit 5 (PB5)	16
Bit 6 (PB6)	17
Bit 7 (PB7)	15

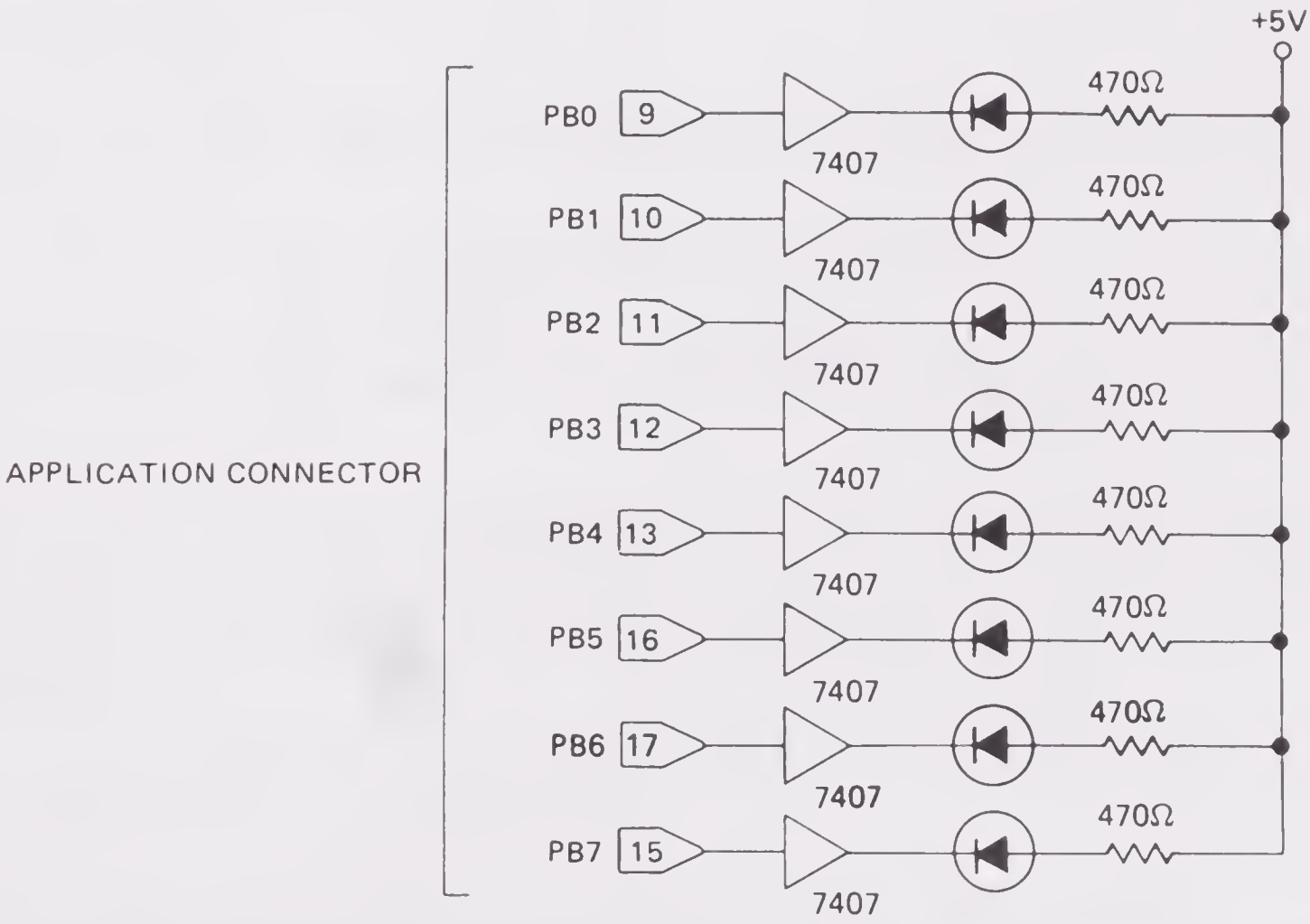


FIGURE 3–1. Attachment of LEDs to the Application Connector. The user VIA is device Z1 in the AIM schematics.

ASSIGNING DIRECTIONS TO VIA I/O LINES

In Laboratory 2, we used a 6522 VIA port for input. In fact, the programmer can make each bit of a VIA port either an input or an output by placing either a 0 (input) or a 1 (output) in the corresponding bit position of the port's data direction register.

Thus, the data direction registers control which way data flows; they act like directional arrows on a highway or railroad. The data direction registers themselves occupy memory addresses (see Table 3–2).

TABLE 3–2 MEMORY ADDRESSES FOR THE I/O PORTS IN THE USER VIA

Address (Hex)	Function
A000	Port B
A001	Port A
A002	Data direction register for port B
A003	Data direction register for port A

Typical examples of making bits inputs or outputs are:

1. Storing 0 in A003 makes port A all inputs.

```
LDA  #0
STA  $A003
```

2. Storing FF hex in A002 makes port B all outputs.

```
LDA  #$FF
STA  $A002
```

3. Storing 0F hex in A002 makes bits 4 through 7 of port B inputs and 0 through 3 outputs.

```
LDA  #$0F
STA  $A002
```

4. Storing AA hex (10101010 binary) in A003 makes bits 1, 3, 5, and 7 of port A outputs and 0,2,4, and 6 inputs.

```
LDA  #$AA
STA  $A003
```

Of course, specifying the data direction register in binary makes it easier to see which bits are inputs and which are outputs. A hexadecimal value is difficult to interpret, since only the individual bits matter.

The 6522 VIA has the following key features:

1. RESET clears the data direction registers, thus making all I/O lines inputs. You can check this by resetting the AIM and examining A002 and A003. This fact allowed us to ignore the data direction registers in Laboratory 2, as long as we started from reset.
2. The I/O ports can consist of any combination of inputs and outputs. Thus AIM users can assign I/O lines rather than making designs conform to a fixed scheme.
3. In an application, the initialization routine (starting from reset) must make bits inputs or outputs. The main program rarely changes the assignments.

PROBLEM 3–1

Write a program that makes port A of the user VIA input and port B output. How would you test this program? What happens when you run it after resetting the computer? What happens if you then change A002 to 00? Change A002 to AA (10101010 binary) and see what happens.

PROBLEM 3–2

Write a program that makes bit 0 of port B an output and the rest of port B inputs. What happens when you reset the computer and run the program?

LIGHTING AN LED

The following program lights the LED attached to bit 3 of user VIA port B (LED 3, for short).

```
LDA  #$FF            ; MAKE PORT B OUTPUT
STA  $A002
LDA  #%11110111      ; LIGHT LED 3
STA  $A000
BRK
```

Since the port is connected to the cathodes, 0 turns an LED on and 1 turns it off. We must make B an output port before any LEDs will light. Program 3–1 is the mnemonic-entry version; enter and run it. What happens if you then reset the AIM?

PROBLEM 3–3

Change Program 3–1 so that it lights only LED 4. How would you make it light only LEDs 2 and 5?

PROBLEM 3–4

Write a program that displays the data from port A on the LEDs attached to port B. Does a switch have to be open or closed for the corresponding LED to light?

PROGRAM 3–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A9	LDA #FF
0201	FF	
0202	8D	STA A002
0203	02	
0204	A0	
0205	A9	LDA #F7
0206	F7	
0207	8D	STA A000
0208	00	
0209	A0	
020A	00	BRK

PRODUCING A TIME DELAY

Of course, in real applications we do not want to leave an output in the same state forever. Instead, we typically want to leave it on or off for a specific amount of time. The microprocessor can wait by performing a simple time-wasting procedure such as:

1. Load a register with a value.
2. Decrement the register until it contains 0.

The program that does this using register X is

```
      LDX  #COUNT
DLY   DEX
      BNE  DLY
```

This works like a countdown before a missile launch or the detonation of explosives.

We can determine how much time is wasted from the following information:

Instruction	Number of Times Executed	Clock Cycles Per Execution
LDX # (IMMEDIATE)	1	2
DEX	COUNT	2
BNE	COUNT	2 if no branch, 3 if a branch occurs

BNE's execution time depends on whether a branch occurs; it takes three cycles if the ZERO flag is 0 and the program branches, and two cycles otherwise.

Table A1–1 contains the execution times for 6502 instructions. The execution time for the delay program is

$$[2 + 5 \times (\text{COUNT} - 1) + 4] \times t_C$$

where t_C is the AIM's clock period. The constants are 2 for the initial LDX #COUNT, 5 for DEX (2) and BNE with a branch (3), and 4 for the last DEX, BNE sequence in which no branch occurs and hence BNE takes only two cycles.

Since the AIM has a 1-MHz clock (see pp. 7–5 to 7–6 of the *AIM 65 User's Guide*), $t_C = 1$ μs. If, for example, COUNT = 10, the amount of time wasted is

$$[5 \times (10 - 1) + 6] \times 1 \text{ μs} = 51 \text{ μs}$$

The most time this program can waste is

$$[5 \times (256 - 1) + 6] \times 1 \text{ μs} = 1{,}281 \text{ μs, or } 1.28 \text{ ms}$$

What value of COUNT produces the longest delay? Note that the program decrements register X before branching.

You can add a delay to Program 3–1 as follows:

```
        LDA   #$FF              ; MAKE PORT B OUTPUT
        STA   $A002
        LDA   #%11110111        ; LIGHT LED 3
        STA   $A000
        LDX   #COUNT            ; DELAY
DLY     DEX
        BNE   DLY
        LDA   #%11111111        ; TURN OFF LED 3
        STA   $A000
        BRK
```

Program 3–2 is the mnemonic-entry version. The first four instructions are the same as in Program 3–1.

Be careful when you enter and run Program 3–2. Do not type DLY or COUNT; entering either will result in an error message. Omit DLY (it simply indicates where the branch goes), and replace COUNT with a number.

A good value for COUNT is 00, since it produces the longest delay. (Why?) However, 1.28 ms is still short, and you should focus your eyes and use a dark background to see the LED light. To obtain the clearest view, first enter the starting address (0200) and press G. Then put a finger on the RETURN key and stare directly at the LED. Finally, press RETURN. Be careful not to hypnotize yourself during this intellectually challenging exercise!

PROGRAM 3–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A9		LDA #FF
0201	FF		
0202	8D		STA A002
0203	02		
0204	A0		
0205	A9		LDA #F7
0206	F7		
0207	8D		STA A000
0208	00		
0209	A0		
020A	A2		LDX #COUNT
020B	00		
020C	CA	DLY	DEX
020D	D0		BNE 020C
020E	FD		
020F	A9		LDA #FF
0210	FF		
0211	BD		STA A000
0212	00		
0213	A0		
0214	00		BRK

PROBLEM 3–5

Run Program 3–2 repeatedly, dividing COUNT (address 020B) in half after each execution; use the sequence 00, 80, 40, 20, 10, 08, 04, 02, 01. What is the smallest value of COUNT for which you can see the LED light?

LENGTHENING THE DELAY

You can lengthen the delay by placing one time-wasting routine inside another (called *nesting*); that is,

```
        LDY  #CT1      ; SET MULTIPLYING FACTOR
DLY1    LDX  #CT2      ; SET DELAY FACTOR
DLY2    DEX
        BNE  DLY2
        DEY
        BNE  DLY1
```

CT1 determines how many times the CT2 loop is executed. Program 3–3 is a revision of Program 3–2 with a nested delay.

PROGRAM 3–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A9		LDA #FF
0201	FF		
0202	8D		STA A002
0203	02		
0204	A0		
0205	A9		LDA #F7
0206	F7		
0207	8D		STA A000
0208	00		
0209	A0		
020A	A0		LDY #CT1
020B	CT1		
020C	A2	DLY1	LDX #CT2
020D	CT2		
020E	CA	DLY2	DEX
020F	D0		BNE 020E
0210	FD		
0211	88		DEY
0212	D0		BNE 020C
0213	F8		
0214	A9		LDA #FF
0215	FF		
0216	8D		STA A000
0217	00		
0218	A0		
0219	00		BRK

PROBLEM 3–6

If you set CT1 (020B) to 200 (C8 hex), what value of CT2 (020D) produces a 10-ms delay? What value of CT2 produces a 100-ms delay?

PROBLEM 3–7

Revise the delay routine to count down location 0040. What is the execution time as a function of 0040's initial contents?

BIT MANIPULATION

Often, we want to change one LED without affecting others attached to the same port. We can do this by using the following effects of the logical functions (see Table 3–3):

1. Logically ANDing a bit with 0 clears it, while logically ANDing it with 1 leaves it unchanged.
2. Logically ORing a bit with 1 sets it (to 1), while logically ORing it with 0 leaves it unchanged.
3. Logically EXCLUSIVE ORing a bit with 1 complements (inverts) it, while logically EXCLUSIVE ORing it with 0 leaves it unchanged.

Thus you can change a particular bit of the accumulator (bit 5, for example) as follows:

1. Make it 1 with ORA #%00100000.
2. Make it 0 with AND #%11011111.
3. Complement (invert) it with EOR #%00100000.

In Program 3–2, for example, we could set bit 3 of the accumulator and thus affect only LED 3 by using ORA #%00001000; that is,

```
020F        09          ORA  #08
0210        08
```

Make this change and run the revised program. Make a similar change in Program 3–3.

PROBLEM 3–8

Write a program that turns LED 4 off, waits for a while, and then turns LED 4 on without affecting any other displays.

TABLE 3–3 EFFECTS OF LOGICAL INSTRUCTIONS

Original Value	Mask Value	AND	OR	Exclusive OR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

PROBLEM 3–9

Write a program that obtains the data for the LEDs from 0040, turns LEDs 2 and 5

on, waits for a while, turns LED 5 off, waits again, and finally turns LED 2 off and LED 5 on without affecting any other LEDs.

Obviously, we can change several bits at once by using the appropriate mask. For example, AND #%11101011 clears bits 2 and 4 of the accumulator. EOR complements (inverts) the entire accumulator if its mask is FF (the all 1's byte); that is, EOR #$FF replaces each 0 with a 1 and each 1 with a 0.

PROBLEM 3–10

Write a program that displays the contents of 0040 on the LEDs attached to port B of the user VIA. Make the data appear in the form an observer would expect—that is, an LED should be lit to indicate a 1 and off to indicate a 0.

DUTY CYCLE

The computer can operate an LED at a specific duty cycle by simply turning it on and then off for periods of time. The following program uses two delay routines to do the job:

```
            LDA  #$FF              ; MAKE PORT B OUTPUT
            STA  $A002
CYCLE       LDA  #%11110111        ; LIGHT LED 3
            STA  $A000
            LDY  #CT1              ; DELAY WHILE LED IS ON
DLY1        LDX  #CT2
DLY2        DEX
            BNE  DLY2
            DEY
            BNE  DLY1
            LDA  #%11111111        ; TURN OFF LED 3
            STA  $A000
            LDY  #CT3              ; DELAY WHILE LED IS OFF
DLY3        LDX  #CT4
DLY4        DEX
            BNE  DLY4
            DEY
            BNE  DLY3
            JMP  CYCLE             ; START OVER
```

Program 3–4 is the mnemonic-entry version. Enter and run it with CT1 = CT2 = CT3 = CT4 = 00.

PROBLEM 3–11

Set CT2 (020D) = CT4 (021C) = 00. Start with CT1 (020B) = CT3 (021A) = 00 and run Program 3–4. Then try the following sequence of hexadecimal values for CT1

PROGRAM 3–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A9		LDA #FF
0201	FF		
0202	8D		STA A002
0203	02		
0204	A0		
0205	A9	CYCLE	LDA #F7
0206	F7		
0207	8D		STA A000
0208	00		
0209	A0		
020A	A0		LDY #CT1
020B	CT1		
020C	A2	DLY1	LDX #CT2
020D	CT2		
020E	CA	DLY2	DEX
020F	D0		BNE 020E
0210	FD		
0211	88		DEY
0212	D0		BNE 020C
0213	F8		
0214	A9		LDA #FF
0215	FF		
0216	8D		STA A000
0217	00		
0218	A0		
0219	A0		LDY #CT3
021A	CT3		
021B	A2	DLY3	LDX #CT4
021C	CT4		
021D	CA	DLY4	DEX
021E	D0		BNE 021D
021F	FD		
0220	88		DEY
0221	D0		BNE 021B
0222	F8		
0223	4C		JMP 0205
0224	05		
0225	02		

and CT3: 80, 40, 20, 10, 08, 04, 02, 01. What is the smallest value for which you can see the LED flicker? How many times per second is the LED being turned on and off at this value?

PROBLEM 3–12

Set CT2 = CT4 = 0. Start with CT1 = CT3 = 10 hex and run Program 3–4. Try the following hexadecimal values for CT1 and CT3:

a. CT1 = 1C, CT3 = 04
b. CT1 = 18, CT3 = 08
c. CT1 = 08, CT3 = 18
d. CT1 = 04, CT3 = 1C

Deseribe how different values affeet the brightness and eontinuity of the LEDs. Compare the effeets to those you saw in Problem 3–11.

PROBLEM 3–13

Set CT2 = CT4 = 0 and CT1 = CT3 = 20 hex. Write a program that flashes the LED on and off for 5 s. Use 0040 as an overall eounter and load it initially from the keyboard before exeeuting the program.

KEY POINT SUMMARY

1. The I/O ports in the 6522 VIA ean be either inputs or outputs. Eaeh bit is made an input or an output by storing a value (0 for input, 1 for output) in the eorresponding bit position of the data direetion register. The data direetion registers themselves oeeupy memory addresses; the user must remember, however, that they are aetually loeated inside the VIA and are not eonneeted to peripherals.
2. By storing the appropriate values in the data direetion registers, the user ean vary the numbers and arrangements of inputs and outputs for different applieations. In most applieations, the initialization routine assigns the direetions and the rest of the program simply uses the ports.
3. The eomputer ean wait by eounting down a register or memory loeation. The length of the wait depends on the number of instruetions in the eountdown program and their exeeution times. Nested eountdown programs ean produee longer waits.
4. A bit ean be eleared, set, or eomplemented by means of logieal operations with appropriate masks. The entire aeeumulator ean be inverted by EXCLUSIVE ORing it with the all 1's byte.
5. The eomputer ean establish a duty eyele by waiting after turning a peripheral on and off.
6. You ean easily ehange the timing for a peripheral if it is implemented in software. Replaeing a few numbers ean ehange the operating speed or duty eyele.

LABORATORY 4

PROCESSING DATA INPUTS

PURPOSE

To learn how to process data inputs.

PARTS REQUIRED

- Eight switches attached through an encoder to user VIA port A as shown in Figure 4–1. This add-on can employ the same switches as the add-on in Figure 2–1, since the two are not needed at the same time.
- A 74148 priority encoder (see Table 4–2 and Figure 4–4 for a description).

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 369–376.

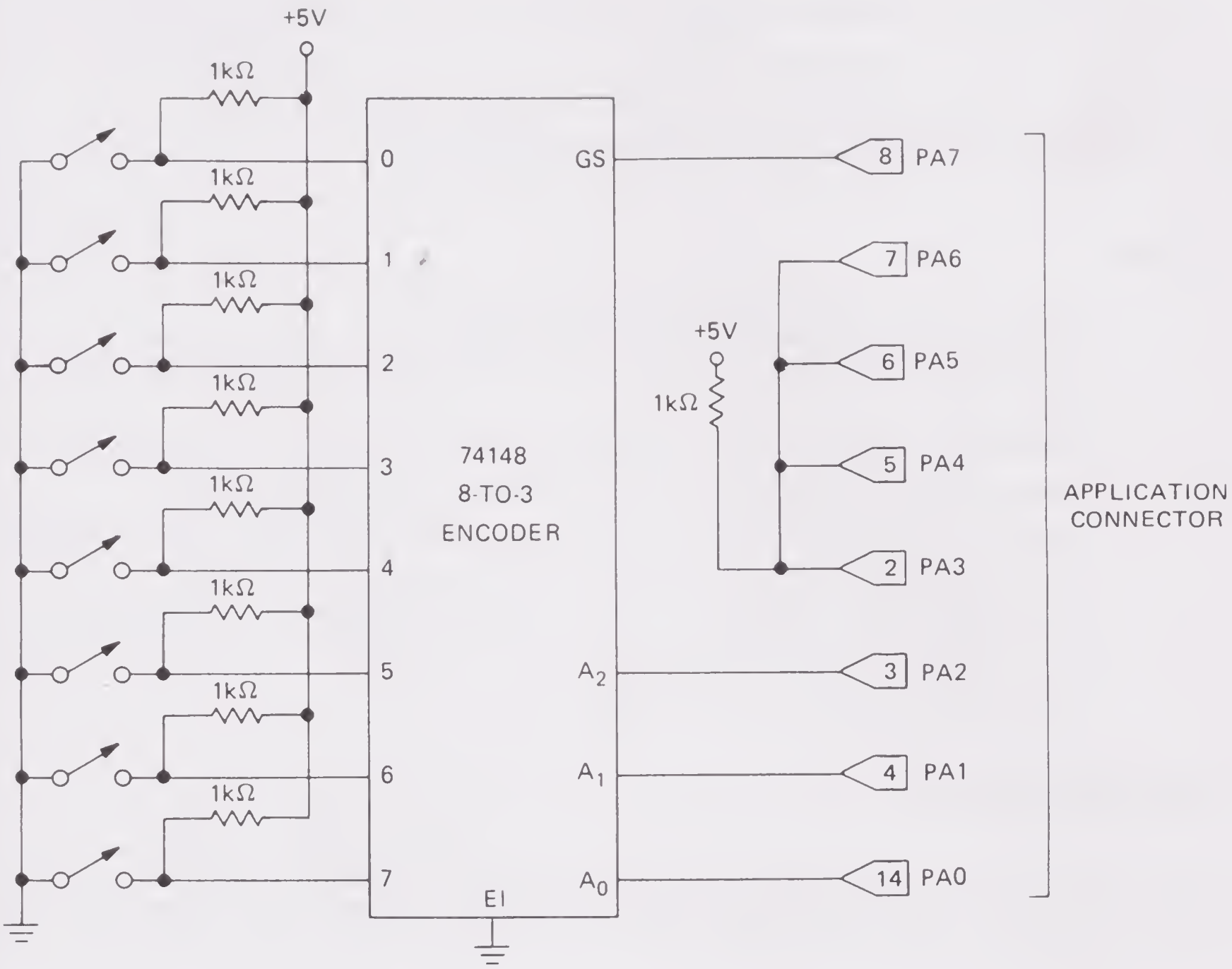


FIGURE 4–1. Attachment of switches and an encoder to user VIA port A.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11–8 to 11–12, 11–39 to 11–60.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 60 (encoders), 274–279 (keyboard input devices), 335–337 (shift and rotate instructions), 346 (unconditional jump instruction).

W. J. WELLER, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapter 11.

The TTL Data Book for Design Engineers, Texas Instruments Inc., Dallas, TX, 1976, pp. 7–151 to 7–156 (74148 encoder).

WHAT YOU SHOULD LEARN

1. How to wait for a switch to open or close.

2. How to debounce a switch.
3. How to count switch closures.
4. How to determine the bit position of a switch closure.
5. How to make simple hardware/software tradeoffs.

TERMS

Bounce move back and forth before settling down.

Cross-coupled describing two devices that each has its output fed back into the other's input.

Debounce convert the output from a contact with bounce into a clean transition.

Enable allow an activity to proceed or a device to produce data outputs.

Encoder a device that produces coded outputs from unencoded inputs. A *priority encoder* accepts only the highest-priority input if more than one is active.

Group Select (GS) a signal that indicates whether any signals in a group are active. It can be used to control function common to the entire group.

Negative logic active state is 0, rather than 1.

6502 INSTRUCTIONS

TAX(Y) transfer accumulator to index register X(Y). The accumulator does not change.

TX(Y)A transfer index register X(Y) to accumulator. The index register does not change.

HANDLING MORE COMPLEX INPUTS

We generally want a microprocessor to do more than just determine if a binary input is 0 or 1. Rather, we want it to deal with a series of inputs and perform such tasks as smoothing data and accounting for the rates at which peripherals operate. These tasks can be performed entirely in software or partly in hardware. Designers must make tradeoffs based on per-unit cost, development time and cost, reliability, compatibility with other applications, power dissipation, board space, and availability of parts that perform specific functions.

WAITING FOR ANY SWITCH TO CLOSE

Table 4–1 lists the inputs produced by closing one of eight switches. If all eight are open,

the input is all 1s (FF hex). So the following program will wait for you to close any switch attached to user VIA port A:

```
WAITC   LDA  $A001      ; GET INPUT DATA
        CMP  #$FF       ; ARE ANY SWITCHES CLOSED?
        BEQ  WAITC      ; NO, WAIT
        BRK
```

TABLE 4–1 INPUTS RESULTING FROM THE CLOSURE OF INDIVIDUAL SWITCHES

Bit Position Of Closed Switch	Input	
	Binary	Hex
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F

Program 4–1 is the mnemonic-entry version. CMP #$FF subtracts FF from the accumulator and sets the flags but does not save the result. Thus the data from port A remains in the accumulator.

Enter and run Program 4–1. Show that it exits if you close any switch. What happens if you close several switches at once? What happens if you close switches before running the program? Note the final contents of the accumulator in each case.

PROGRAM 4–1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	AD	WAITC	LDA A001
0201	01		
0202	A0		
0203	C9		CMP #FF
0204	FF		
0205	F0		BEQ 0200
0206	F9		
0207	00		BRK

We can easily add a section that waits until all the switches are open again. The new section simply branches on the opposite condition.

```
WAIT0   LDA   $A001      ; GET INPUT DATA
        CMP   #$FF       ; ARE ANY SWITCHES CLOSED?
        BNE   WAIT0      ; YES, WAIT
```

Program 4–2 contains the mnemonic-entry additions to Program 4–1. Enter them into memory and run the combined program several times. Does it always wait for you to open the switch?

PROGRAM 4–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0207	AD	WAIT0	LDA A001
0208	01		
0209	A0		
020A	C9		CMP #FF
020B	FF		
020C	D0		BNE 0207
020D	F9		
020E	00		BRK

PROBLEM 4–1

Write a program that waits for you to close and open switch 5, regardless of the other switches.

PROBLEM 4–2

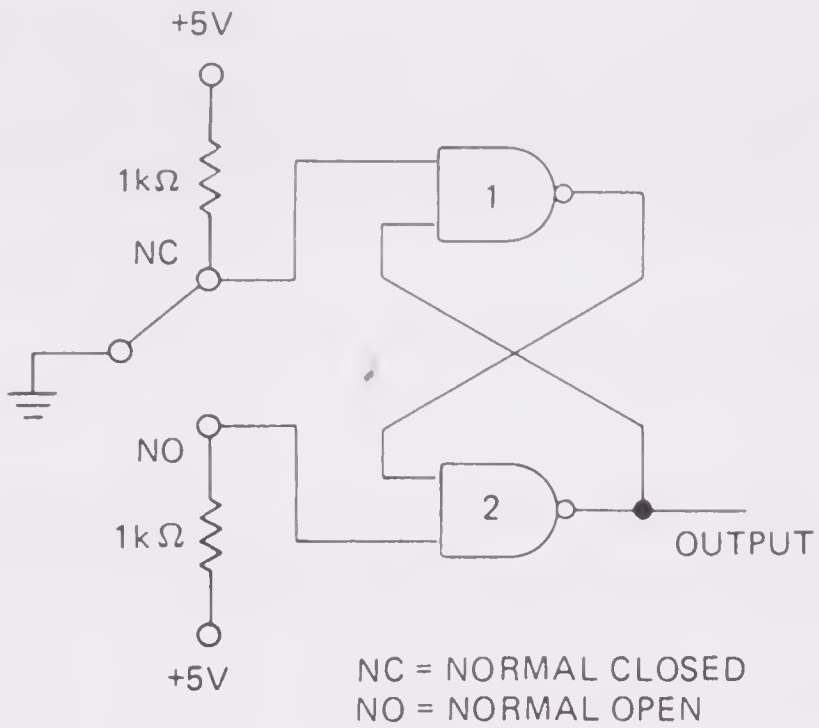
Write a program that waits for you to close switch 5, open it, and then close it again, regardless of the other switches.

DEBOUNCING A SWITCH

If you run Programs 4–1 and 4–2 many times, you will probably find that the computer often exits before you open the switch. This occurs because a mechanical switch does not open or close cleanly. Instead, it bounces for a while before settling into its final position. Thus opening or closing a switch typically causes several transitions, just as though it had been opened and closed repeatedly.

We can eliminate the extra transitions by debouncing the switch. This can be done in hardware with cross-coupled NAND gates (see Figure 4–2) or in software with a delay that waits until the switch stops bouncing. Since the bounce usually lasts less than 1 ms, the following program will do the job:

```
WAITC   LDA   $A001      ; GET INPUT DATA
        CMP   #$FF       ; ARE ANY SWITCHES CLOSED?
```



NC = NORMAL CLOSED
NO = NORMAL OPEN

FIGURE 4–2. Debouncing a switch with cross-coupled NAND gates.

```
              BEQ   WAITC          ; NO, WAIT
              LDX   #$C8           ; DELAY 1 MS TO DEBOUNCE
DLY           DEX
              BNE   DLY
WAITO         LDA   $A001          ; GET INPUT DATA
              CMP   #$FF           ; ARE ANY SWITCHES CLOSED?
              BNE   WAITO          ; YES, WAIT
              BRK
```

Program 4–3 contains the mnemonic-entry additions to Program 4–1. We obtained C8 from the timing equation in Laboratory 3. Many switches require a larger value such as 0FA0(20 ms).

PROGRAM 4–3

Address (Hex)	Contents (Hex)		Instruction (Mnemonic)
0207	A2		LDX #C8
0208	C8		
0209	CA	DLY	DEX
020A	D0		BNE 0209
020B	FD		
020C	AD	WAITO	LDA A001
020D	01		
020E	A0		
020F	C9		CMP #FF
0210	FF		
0211	D0		BNE 020C
0212	F9		
0213	00		BRK

Debouncing is an example of a tradeoff between hardware and software. The software delay costs very little, since the program is simple and takes only a few bytes of memory. On the other hand, it ties up the processor, preventing it from doing other work. Hardware debouncing frees the processor but requires an additional part and more connections.

COUNTING CLOSURES

We can keep a running count in 0040 of the number of switches closed and then reopened as follows, assuming that we close and then reopen only one switch at a time:

1. Add the instructions

```
        INC   $40          ; INCREMENT NUMBER OF CLOSURES
        LDX   #$C8         ; DELAY 1 MS TO DEBOUNCE OPENING
DLY1    DEX
        BNE   DLY1
        JMP   WAITC        ; WAIT FOR NEXT CLOSURE
```

   to the end of Program 4–3, as shown in Program 4–4.
2. Clear 0040 from the keyboard before executing the program.

Since the program runs forever, you must reset the AIM to examine 0040. If the count is erratic, lengthen the waiting period by using the nested delay in Program 3–3. Remember to debounce both the opening and the closing of the switch.

PROGRAM 4–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0213	E6		INC 40
0214	40		
0215	A2		LDX #C8
0216	C8		
0217	CA	DLY1	DEX
0218	D0		BNE 0217
0219	FD		
021A	4C		JMP 0200
021B	00		
021C	02		

PROBLEM 4–3

Write a program that returns control to the monitor after counting the number of switch closures in 0040. Assume that only one switch is ever closed at a time.

PROBLEM 4–4

Write a program that counts how many times switch 5 is closed. Use 0041 for the counter.

PROBLEM 4–5

Write a program that counts how many times switches 2 and 5 are closed. Use 0040 as the counter for switch 2 and 0041 for switch 5. Assume that only one switch is ever closed at a time. The program can then simply wait for all switches to be open rather than waiting specifically for the opening of the switch that was closed.

IDENTIFYING THE SWITCH

In Table 4–1, the bit that is 0 tells you which switch is closed. That is, bit 0 is 0 if switch 0 is closed, bit 1 is 0 if switch 1 is closed, and so on. A simple way to find out which bit is 0 is the following (see Figure 4–3 for a flowchart):

1. SWITCH NUMBER = 0
   DATA = input from switches
2. Shift DATA right 1 bit. If CARRY is 0, the program is finished.

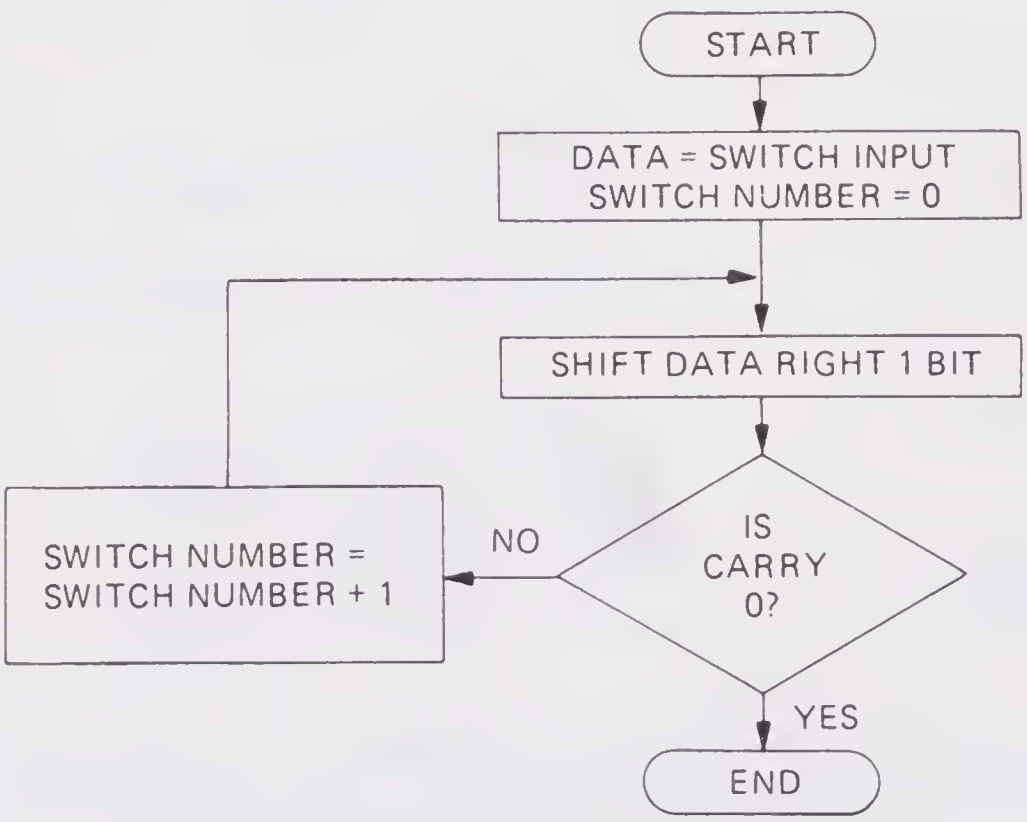


FIGURE 4–3. Flowchart for switch identification.

3. SWITCH NUMBER = SWITCH NUMBER + 1
   Go to step 2.

A program that does this is

```
        LDY   #0          ; SWITCH NUMBER = ZERO
SRCHS   LSR   A           ; IS NEXT SWITCH CLOSED?
        BCC   DONE        ; YES, DONE
        INY               ; NO, ADD 1 TO SWITCH NUMBER
        JMP   SRCHS
DONE    BRK
```

The switch number ends up in index register Y.

If we use a different starting point, we can omit JMP.

```
        LDY   #$FF        ; SWITCH NUMBER = -1
SRCHS   INY               ; ADD 1 TO SWITCH NUMBER
        LSR   A           ; IS NEXT SWITCH CLOSED?
        BCS   SRCHS       ; NO, KEEP LOOKING
        BRK
```

Which program do you prefer, and why?

A switch identification program must do the following:

1. Wait for any switch to be closed.
2. Wait 1 ms to debounce the switch.
3. Identify the switch by shifting the input and counting until CARRY becomes 0.

A complete assembly language program is

```
WAITC   LDA   $A001       ; GET INPUT DATA
        CMP   #$FF        ; ARE ANY SWITCHES CLOSED?
        BEQ   WAITC       ; NO, WAIT
        LDX   #$C8        ; YES, DELAY 1 MS TO DEBOUNCE
DLY     DEX
        BNE   DLY
        LDY   #$FF        ; SWITCH NUMBER = -1
SRCHS   INY               ; ADD 1 TO SWITCH NUMBER
        LSR   A           ; IS NEXT SWITCH CLOSED?
        BCS   SRCHS       ; NO, KEEP LOOKING
        STY   $41         ; YES, SAVE SWITCH NUMBER
        BRK
```

The switch number ends up both in 0041 and in register Y. Program 4–5 is a mnemonic-entry version, but note that addresses 0200 through 020B are the same as in Programs 4–1 and 4–3.

Enter Program 4–5 into memory and test it on each switch individually. What happens if you close more than one switch before executing the program? Which closure does it detect and why?

PROGRAM 4–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	AD	WAITC	LDA	A001
0201	01			
0202	A0			
0203	C9		CMP	#FF
0204	FF			
0205	F0		BEQ	0200
0206	F9			
0207	A2		LDX	#C8
0208	C8			
0209	CA	DLY	DEX	
020A	D0		BNE	0209
020B	FD			
020C	A0		LDY	#FF
020D	FF			
020E	C8	SRCHS	INY	
020F	4A		LSR	A
0210	B0		BCS	020E
0211	FC			
0212	84		STY	41
0213	41			
0214	00		BRK	

PROBLEM 4–6

Write a program that always finds the highest-numbered switch that is closed. (*Hint:* Shift the data left and decrement Y, but remember to initialize Y correctly.)

PROBLEM 4–7

Revise Program 4–5 so that it checks the switches only once. If it finds none closed, it places FF in 0041. What happens if this program checks the switches while one is bouncing? How could you solve this problem? (*Hint:* If the program finds all switches open, have it wait 1 ms and examine them again.)

Write a general program that accepts the input from the switches only if it remains the same after a 1-ms delay. That is, the program should keep checking the switches until two readings taken 1 ms apart have the same value.

USING A HARDWARE ENCODER

The 74148 priority encoder produces a 3-bit output in negative logic that identifies the highest-priority active (low) input. Table 4–2 is a function table for the device, and Figure 4–4 contains its pin assignments. Note the following:

1. The data outputs (As) are the logical complement of the highest-priority active input. For example, the outputs are 0,1,0 if #5 (101 binary) is the highest-priority active input.
2. The ENABLE IN (EI) input and the ENABLE OUT (EO) output are used to combine encoders to handle more than eight inputs. If EI is high (indicating activity at a level higher than the entire encoder), all outputs are high. If EI is low (indicating no higher activity) but the encoder has no active input, EO is low, thus enabling encoders of lower priority.

TABLE 4–2 FUNCTION TABLE FOR 74148 ENCODER*

Inputs									Outputs				
EI	0	1	2	3	4	5	6	7	A_2	A_1	A_0	GS	EO
H	—	—	—	—	—	—	—	—	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	—	—	—	—	—	—	—	L	L	L	L	L	H
L	—	—	—	—	—	—	L	H	L	L	H	L	H
L	—	—	—	—	—	L	H	H	L	H	L	L	H
L	—	—	—	—	L	H	H	H	L	H	H	L	H
L	—	—	—	L	H	H	H	H	H	L	L	L	H
L	—	—	L	H	H	H	H	H	H	L	H	L	H
L	—	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

*H = high (1), L = low (0).

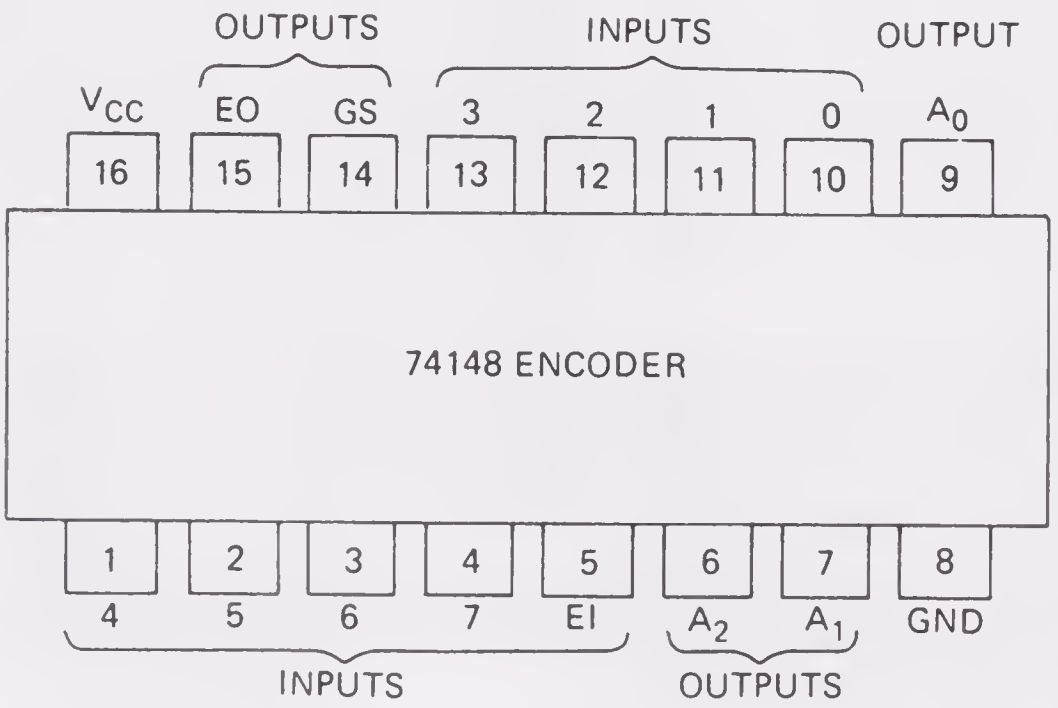


FIGURE 4–4. Pin assignments for the 74148 encoder.

3. The GROUP SELECT (GS) output is low if the encoder is enabled and has an active input. GS thus indicates activity at this encoder; it is not disabled (top line of Table 4–2) or inactive (second line of Table 4–2), even if all data outputs (As) are 1s.

Connect the encoder as described in Table 4–3. The following program identifies the highest-numbered switch that is closed before the program is executed. The switch number ends up in the accumulator and in 0041.

TABLE 4–3 CONNECTIONS FOR 74148 ENCODER

Pin Number	Designation	Connection
1	Input 4	Switch 4
2	Input 5	Switch 5
3	Input 6	Switch 6
4	Input 7	Switch 7
5 (EI)	Enable in	Ground
6 (A2)	Output 2	User VIA pin PA2
7 (A1)	Output 1	User VIA pin PA1
8	Ground	Ground
9 (A0)	Output 0	User VIA pin PA0
10	Input 0	Switch 0
11	Input 1	Switch 1
12	Input 2	Switch 2
13	Input 3	Switch 3
14 (GS)	Group select	User VIA pin PA7
15 (EO)	Enable out	No connection
16	V_{CC}	+5 V

```
LDA   $A001                ; GET SWITCH DATA
EOR   #$FF                 ; INVERT LOGIC
AND   #%00000111           ; MASK SWITCH BITS
STA   $41                  ; SAVE SWITCH NUMBER
BRK
```

EOR #$FF inverts the data. This makes up for the fact that the 74148 encoder, like many TTL devices, uses negative logic. Program 4–6 is a mnemonic-entry version; enter it and test it on several different switch closures. What happens if more than one switch is closed?

PROBLEM 4–8

To detect whether any switches are closed, you must test the encoder's Group Select (GS) output. Write a program that tests GS and stores either the switch number or FF (if no switches are closed) in 0041. Note that we have grounded EI (see Table 4–3), so the encoder is always enabled.

PROGRAM 4–6

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0200	AD	LDA	A001
0201	01		
0202	A0		
0203	49	EOR	#FF
0204	FF		
0205	29	AND	#07
0206	07		
0207	85	STA	41
0208	41		
0209	00	BRK	

PROBLEM 4–9

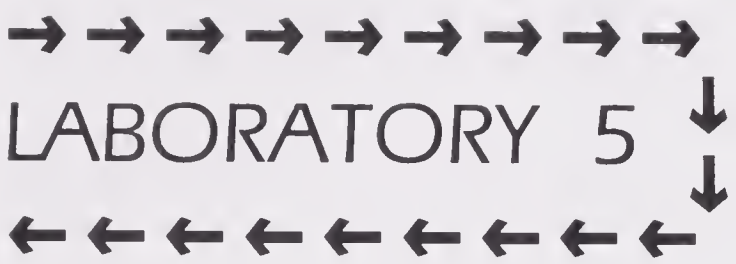
What values will the processor read from port A if you invert the switch connections (i.e., connect switch 7 to encoder input 0, and so on)? Write a program that places the switch number in 0041 in this case. How does the inversion affect the priority of the switches?

Obviously, an encoder makes the software simpler and faster and saves input lines (since it uses 4 rather than 8). On the other hand, it increases the parts count, dissipates power, requires extra connections (which reduce reliability), and uses board space. In low-volume applications, you can surely afford extra hardware if it simplifies the software. In high-volume applications, you must minimize hardware, since it adds to the cost of each system produced.

KEY POINT SUMMARY

1. A mechanical switch requires a relatively long time to settle into a new position. You can either introduce a delay during which the processor does not examine the switch, or you can add hardware that smooths the transition. Mechanical components generally take much longer to change states than do electrical components. Either hardware or software must account for this difference.
2. Inputs must usually be converted into a convenient form before they can be processed. Either hardware or software can perform this conversion.
3. Timing and code conversion are common functions that either hardware or software can perform. Extra hardware can result in shorter, simpler programs. This usually makes system development easier, particularly if the designer is more familiar with hardware than with software. Doing everything in software reduces parts count, saves board space, and increases reliability.

4. Many factors affect tradeoffs between software and hardware. Among these are the cost and availability of parts, designer experience, product volume, amount of memory available, amount of board space, and performance requirements. Remember the following considerations:
    a. Software costs are incurred only once, whereas hardware costs are repeated for each system produced. Thus high-volume products should have more software and less hardware than low-volume products.
    b. A single processor can do many tasks in software, particularly if they involve slow mechanical components. External hardware, on the other hand, is more difficult to share, even among similar tasks.
    c. Certain tasks, such as switch and keyboard encoding, display decoding, and serial/parallel interfacing, are so common that inexpensive circuits are readily available to perform them. Circuits for similar but less common tasks are generally far more expensive. The obvious reason is that manufacturers must recover their design and development costs over a smaller number of units.



PROCESSING DATA OUTPUTS

PURPOSE

To learn how to process data outputs.

REFERENCE MATERIALS

R. C. CAMP et al., *Microprocessor System Engineering*, Matrix Publishers, Portland, OR, 1979, pp. 438–445.

L. A. LEVENTHAL, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 205–208, 377–378.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 7–4 to 7–6, 11–13 to 11–22 (6520 Peripheral Interface Adapter), 11–65 to 11–75.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 58–59 (decoders), 168–170 (index register), 230–237 (practical interfacing considerations), 338–340 (compare instructions), 360–367 (indexed addressing).

W. J. WELLER, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapter 10.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, pp. 7–24 to 7–27.

WHAT YOU SHOULD LEARN

1. How the AIM display is organized and connected.
2. How to activate characters and send them data.
3. How to convert numbers to ASCII.
4. How and when to use lookup tables.
5. How indexed addressing works and how it is used.
6. How to count on the display.
7. How to move a character across the display.

TERMS

Absolute indexed addressing indexed addressing with a 16-bit base address.

Alphanumeric display display that can form both letters and digits.

ASCII American Standard Code for Information Interchange, a 7-bit character code widely used in computers and communications. Appendix 2 contains an ASCII table.

Base address (or **base**) memory address at which a table begins.

Character one of a set of elementary symbols, usually including controls and delimiters as well as representations of letters, digits, punctuation marks, and other symbols.

Effective address actual address used by an instruction to perform its overall function.

Endless loop (or **jump-to-self**) **instruction** an instruction that transfers control to itself.

Index data item used to select an element from a set of data.

Index register register that can be used to modify memory addresses.

Indexed addressing addressing method in which the address in the instruction is modified by an index register to determine the effective address.

Lookup table set of data organized so that the answer to a problem may be determined merely by selecting the correct entry (without any calculations).

Zero-page indexed addressing indexed addressing with an 8-bit address on page 0.

6502 INSTRUCTIONS

ADC add a memory location and the CARRY flag to the accumulator.

CLC clear the CARRY flag (make it 0).

CPX(Y) compare memory and index register X(Y); subtract a memory location from index register X(Y) but leave the index register unchanged. This instruction affects only the flags. The addressing modes allowed are immediate, absolute (direct), and zero-page (direct).

HANDLING MORE COMPLEX OUTPUTS

In real applications, the microprocessor must do more than merely turn a binary output on or off. Rather, it must produce a sequence of outputs and convert the data into the forms peripherals require. The processor should also time the outputs properly.

As with inputs, we can use either hardware or software to process outputs. The designer must make tradeoffs suited to the application. Furthermore, the designer can often make tradeoffs between execution time and memory usage. One way to perform a calculation is to use a table that contains all possible results. Now the program must simply select the correct entry, much as one might use a book of tables to obtain values needed in surveying, navigation, or finance. This method (called *table lookup*) is fast and easy to implement but usually requires more memory than an explicit calculation.

USING THE ON-BOARD DISPLAY

The AIM's on-board display is a simple example of an output device that requires parallel data, timing, and code conversion. Figure 5–1 shows the display interface. The 20 characters are divided into five modules, each with four alphanumeric displays. The *AIM 65 User's Guide* refers to the modules as DS1 through DS5 and the characters as 1 through 20 from left to right (see Table 5–1). Within each module, the characters are also numbered as 0 to 3 *from right to left* (see Figure 5–2). Be careful of this confusing distinction.

The display interface, a 6520 PIA, is like a VIA. All we need to know about it is

1. Address AC02 is used to send data to the displays. Bit 7 should always be 1.
2. Address AC00 is used to select a display and control data storage.

Figure 5–3 describes the organization of AC00. Bits 0 and 1 select a character from a module. This is where we need the odd right-to-left numbering. Zeros in bits 2 through 6 select modules as shown in Table 5–2. The end result is that bits 0 through 6 of AC00 select a character as given in Table 5–3.

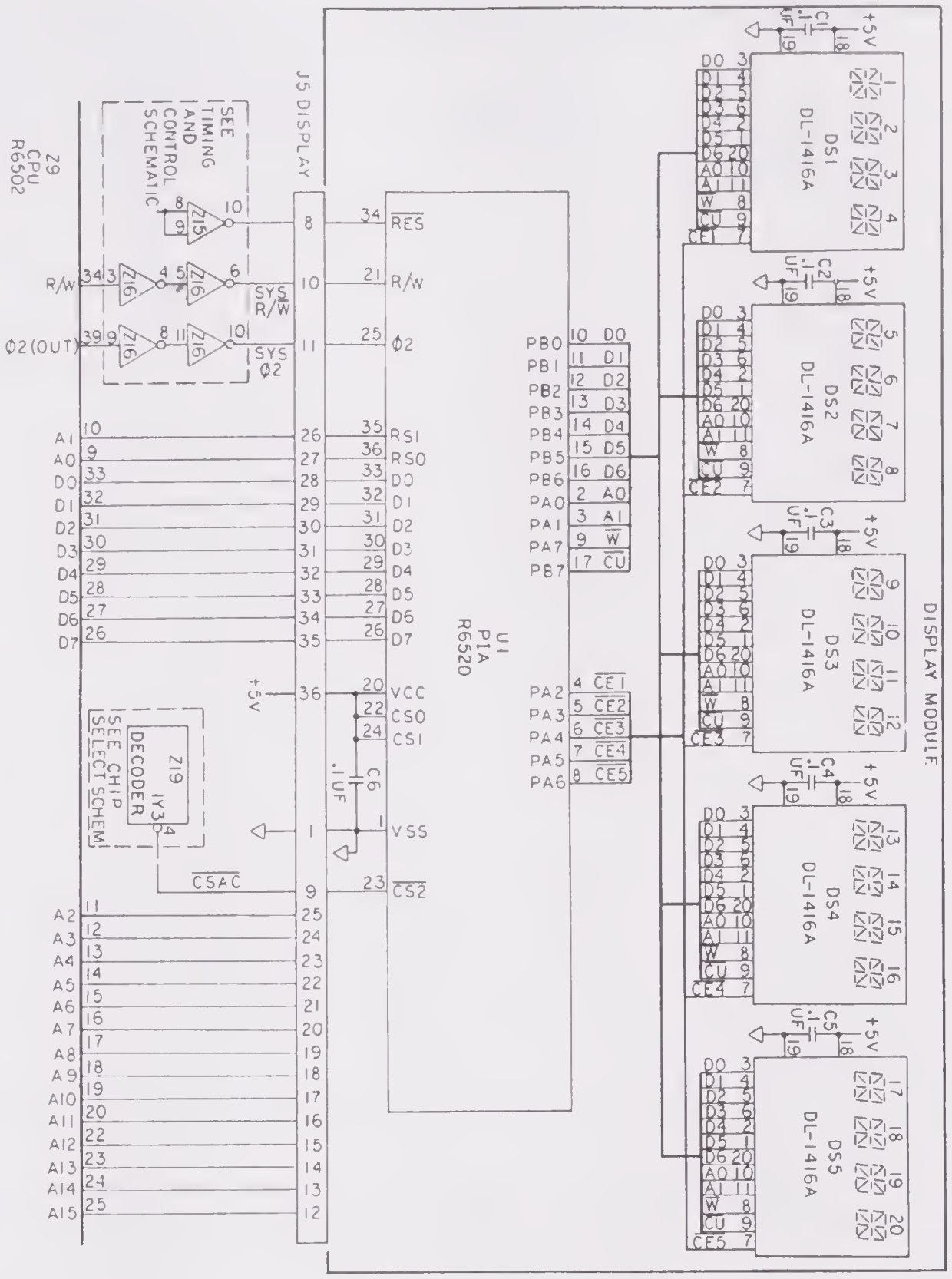


FIGURE 5–1. Schematic for the on-board display. (Courtesy of Dynatem, Irvine, Calif.)

Bit 7 controls data storage; 0 stores the current data in the selected character, whereas 1 retains the old data. This allows us to change selection codes without transient effects.

Thus the following program activates the character defined by ACTIVE and stores the value DATA there.

```
LDA  #DATA          ; SEND DATA TO DISPLAY
STA  $AC02
LDA  #ACTIVE        ; ACTIVATE A CHARACTER
STA  $AC00
```

TABLE 5–1 NUMBERING OF CHARACTERS IN ON-BOARD DISPLAY

Position	Module Designation	Overall Character Designation	Character Designation Within Module
LEFTMOST	DS1	1	3
	DS1	2	2
	DS1	3	1
	DS1	4	0
	DS2	5	3
	DS2	6	2
	DS2	7	1
	DS2	8	0
	DS3	9	3
	DS3	10	2
	DS3	11	1
	DS3	12	0
	DS4	13	3
	DS4	14	2
	DS4	15	1
	DS4	16	0
	DS5	17	3
	DS5	18	2
	DS5	19	1
RIGHTMOST	DS5	20	0

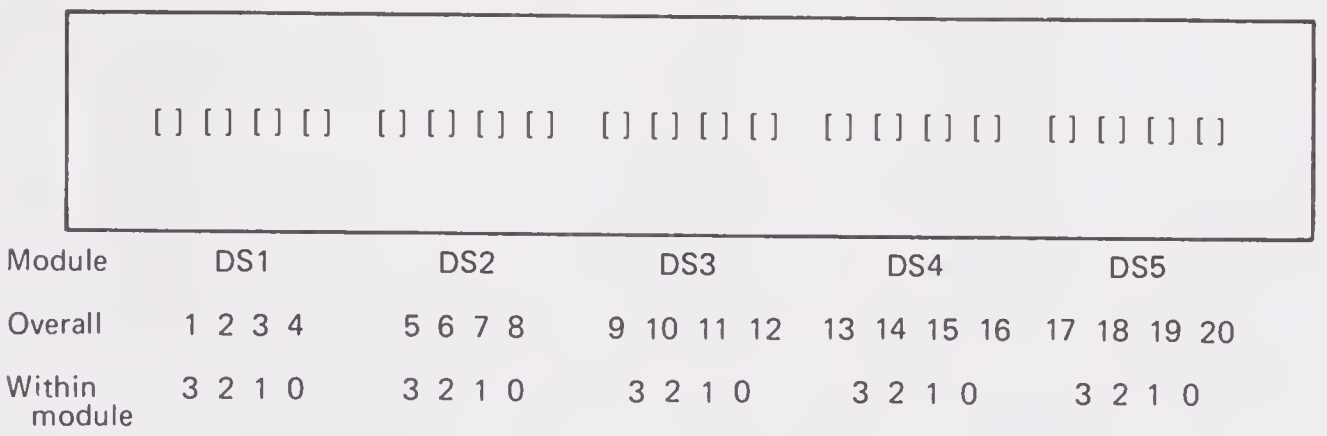


FIGURE 5–2. Numbering of characters in the on-board display.

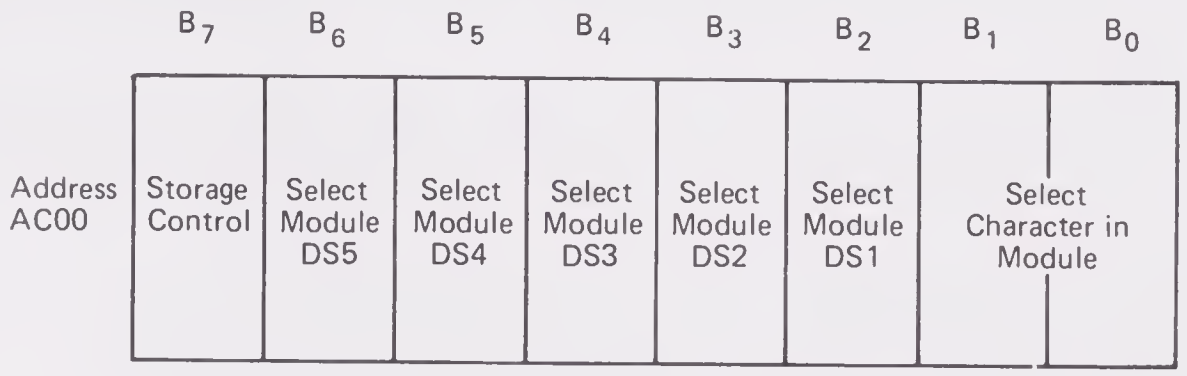


FIGURE 5–3. Control and selection for the on-board display.

TABLE 5–2 OUTPUTS FOR ACTIVATING MODULES

Module	D_6	D_5	D_4	D_3	D_2
DS1	1	1	1	1	0
DS2	1	1	1	0	1
DS3	1	1	0	1	1
DS4	1	0	1	1	1
DS5	0	1	1	1	1

TABLE 5–3 OUTPUTS FOR ACTIVATING CHARACTERS

Overall Character Designation	Output	
	Binary	Hex
1	01111011	7B
2	01111010	7A
3	01111001	79
4	01111000	78
5	01110111	77
6	01110110	76
7	01110101	75
8	01110100	74
9	01101111	6F
10	01101110	6E
11	01101101	6D
12	01101100	6C
13	01011111	5F
14	01011110	5E
15	01011101	5D
16	01011100	5C
17	00111111	3F
18	00111110	3E
19	00111101	3D
20	00111100	3C

We obtain the value ACTIVE from Table 5–3. To retain that data while changing another character, we set bit 7 of AC00 and deactivate all modules with

```
LDA  #$FF                ; RETAIN DATA, DEACTIVATE EVERYTHING
STA  $AC00
```

The AIM display requires data in a form called the American Standard Code for Information Interchange, or ASCII (pronounced "ass-kee"). Table 5–4 lists the ASCII decimal digits with bit 7 set to 1. Appendix 3 contains a complete ASCII table with bit 7 cleared. Remember that data sent to the AIM display must have bit 7 set.

The following program displays 0 on the leftmost character. The display selection code comes from Table 5–3 and the data value from Table 5–4.

```
LDA  #$B0              ; DATA = ASCII 0 WITH BIT 7 SET
STA  $AC02
LDA  #$7B              ; SELECT LEFTMOST CHARACTER
STA  $AC00
BRK
```

TABLE 5–4 DECIMAL-TO-ASCII CONVERSION TABLE

Decimal Digit	ASCII Digit with Bit 7 Set (Hex)
0	B0
1	B1
2	B2
3	B3
4	B4
5	B5
6	B6
7	B7
8	B8
9	B9

PROGRAM 5–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0200	A9	LDA	#B0
0201	B0		
0202	8D	STA	AC02
0203	02		
0204	AC		
0205	A9	LDA	#7B
0206	7B		
0207	8D	STA	AC00
0208	00		
0209	AC		
020A	00	BRK	

Enter and run Program 5–1. What happens? The problem is that the monitor takes over the displays as soon as the proeessor exeeutes BRK. Thus all we see is the next value of the program eounter (020B) and the instruetion at that address. Program 5–1 ean retain eontrol of the display if you replaee BRK with an endless loop (i.e., an instruetion that jumps to itself).

020A	4C	HERE	JMP	020A
020B	0A			
020C	02			

The revised program runs forever, so you must reset the AIM to regain control.

PROBLEM 5–1

Write a program that displays 6 on the leftmost character.

PROBLEM 5–2

Write a program that displays 6 on character 12.

PROBLEM 5–3

Write a program that displays 8 on character 4 and 2 on character 5. Be sure to retain the data on the first character and deactivate all modules before changing the activation code. What happens if you omit this step? Can you explain the result?

ADDING A DELAY

The following delay routine leaves the display on for a while before returning control to the monitor.

```
        LDY  #CT1          ; SET MULTIPLYING FACTOR
DLY1    LDX  #CT2          ; SET DELAY FACTOR
DLY2    DEX
        BNE  DLY2
        DEY
        BNE  DLY1
```

Program 5–2 is the mnemonic-entry version of a delay at the end of Program 5–1. Enter the changes, setting CT1 = CT2 = 00, and run the program.

PROGRAM 5–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020A	A0		LDY #CT1
020B	CT1		
020C	A2	DLY1	LDX #CT2
020D	CT2		
020E	CA	DLY2	DEX
020F	D0		BNE 020E
0210	FD		
0211	88		DEY
0212	D0		BNE 020C
0213	F8		
0214	00		BRK

PROBLEM 5–4

Leaving CT2 = 00, run Program 5–2 with the following values for CT1 (address 020B): 80, 40, 20, 10, 08, 04, 02, 01, 00. What is the smallest value for which you can see the zero appear?

PROBLEM 5–5

What happens if you change the display data (address 0201) to A4? Explain the result. Try the following data values and see how they look: A7, AF, BF, C2, DB.

DECIMAL-TO-ASCII CONVERSION

We can easily make the program convert decimal digits to ASCII and character numbers to activation codes. Let us first implement decimal-to-ASCII conversion.

To convert a decimal digit to ASCII with bit 7 set, we need only add B0 (see Table 5–4). Thus the following program converts a decimal digit in 0040 to an ASCII digit in 0041. Program 5–3 is the mnemonic-entry version.

```
LDA   $40          ; GET DECIMAL DIGIT
CLC                ; CONVERT DECIMAL TO ASCII
ADC   #$B0
STA   $41          ; SAVE ASCII DIGIT
BRK
```

This is more complicated than you might expect, since the 6502's only add instruction is ADC (add with carry). ADC's result is

$$(A) = (A) + (M) + \text{CARRY}$$

where M is a memory location. To keep CARRY from interfering with the conversion, we must clear it before adding. The sequence

```
CLC
ADC   #$B0
```

produces the result

$$\begin{aligned}(A) &= (A) + \text{B0 hex} + \text{CARRY} \\ &= (A) + \text{B0 hex}\end{aligned}$$

Enter Program 5–3 into memory and run it for the following test cases:

1. Data: (0040) = 00
   Result: (0041) = B0
2. Data: (0040) = 07
   Result: (0041) = B7

PROGRAM 5–3

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A5	LDA 40
0201	40	
0202	18	CLC
0203	69	ADC #B0
0204	B0	
0205	85	STA 41
0206	41	
0207	00	BRK

The next program shows the decimal digit from 0040 on the leftmost character of the display. Program 5–4 is the mnemonic-entry version.

```
LDA   #$7B                ; ACTIVATE LEFTMOST CHARACTER
STA   $AC00
LDA   $40                 ; GET DECIMAL DIGIT
CLC                       ; CONVERT DIGIT TO ASCII
ADC   #$B0
STA   $AC02               ; SEND ASCII DATA TO DISPLAY
HERE  JMP  HERE           ; WAIT FOREVER
```

PROGRAM 5–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A9		LDA	#7B
0201	7B			
0202	8D		STA	AC00
0203	00			
0204	AC			
0205	A5		LDA	40
0206	40			
0207	18		CLC	
0208	69		ADC	#B0
0209	B0			
020A	8D		STA	AC02
020B	02			
020C	AC			
020D	4C	HERE	JMP	020D
020E	0D			
020F	02			

PROBLEM 5–6

Change Program 5–4 to show the digit on eharaeter 15.

PROBLEM 5–7

Sinee adding B0 hex to a deeimal digit simply sets bits 4, 5, and 7, you ean replaee CLC and ADC #$B0 with a logieal OR. Make the ehange and run the revised program. This approaeh is shorter but not as easy to explain.

HEXADECIMAL-TO-ASCII CONVERSION

We may extend the ASCII eonversion to hexadeeimal digits using Table 5–5. Note, however, that this table has a gap. Instead of ASCII A being BA hex, it is C1 hex. After the gap, the eodes are eonseeutive again. To bridge the gap, we must add an extra 7 (C1-BA) for digits that are 10 (0A hex) or larger.

TABLE 5–5 HEXADECIMAL-TO-ASCII CONVERSION TABLE

Hexadecimal Digit	ASCII Digit with Bit 7 Set (Hex)
0	B0
1	B1
2	B2
3	B3
4	B4
5	B5
6	B6
7	B7
8	B8
9	B9
A	C1
B	C2
C	C3
D	C4
E	C5
F	C6

The following program eonverts a hexadeeimal digit in 0040 into an ASCII digit in 0041. Program 5–5 is the mnemonie-entry version.

```
        LDA   $40       ; GET HEXADECIMAL DIGIT
        CMP   #10       ; IS DIGIT DECIMAL?
        BCC   CNVRT
        ADC   #6        ; NO, ADD EXTRA 7 (6 PLUS CARRY)
CNVRT   CLC             ; CONVERT DIGIT TO ASCII
```

```
        ADC   #$B0
        STA   $41                    ; SAVE ASCII DIGIT
        BRK
```

PROGRAM 5–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A5		LDA	40
0201	40			
0202	C9		CMP	#0A
0203	0A			
0204	90		BCC	0208
0205	02			
0206	69		ADC	#06
0207	06			
0208	18	CNVRT	CLC	
0209	69		ADC	#B0
020A	B0			
020B	85		STA	41
020C	41			
020D	00		BRK	

Comparing values is also more difficult than one might expect. CMP affects CARRY as follows:

- CARRY = 1 if the subtraction does not require a borrow, that is, if the accumulator is greater than or equal to the number being subtracted from it.
- CARRY = 0 if the subtraction requires a borrow, that is, if the accumulator is less than the number being subtracted from it.

If this seems backward to you, you are surely in the majority. Most other microprocessors (e.g., the 6809, 68000, Z-80, and 8086 or 8088) work the opposite way—they set the CARRY if a borrow is necessary and clear it otherwise.

In our program, we want the computer to branch if the digit is less than 10. In that case, subtracting 10 will require a borrow and thus clear CARRY. The branch should therefore be BCC.

If BCC does not cause a branch, we know CARRY is 1. ADC #6 will therefore produce the result

$$\begin{aligned}(A) &= (A) + 6 + \text{CARRY} \\ &= (A) + 6 + 1 \\ &= (A) + 7\end{aligned}$$

Work through the program by hand if you find it confusing. Unfortunately, computer designers are not required to be logical or sensible.

Enter Program 5–5 into memory and try it on the following sample cases:

1. Data: (0040) = 08
   Result: (0041) = B8
2. Data: (0040) = 0C
   Result: (0041) = C3

PROBLEM 5–8

Extend Program 5–5 to show the hexadecimal digit on the leftmost character of the display.

COUNTING ON THE DISPLAYS

We can use either conversion routine to count on the displays. The following program counts up from 0 to 9 in the leftmost character. Program 5–6 is the mnemonic-entry version.

```
        LDA   #$B0          ; INITIAL DATA = ASCII ZERO
        STA   $AC02
        LDA   #$7B          ; ACTIVATE LEFTMOST CHARACTER
        STA   $AC00
COUNT   LDY   #CT1          ; WAIT A WHILE
DLY1    LDX   #CT2
DLY2    DEX
        BNE   DLY2
        DEY
        BNE   DLY1
        LDA   $AC02         ; HAS COUNT REACHED 9?
        CMP   #$B9
        BEQ   DONE          ; YES, DONE
        INC   $AC02         ; NO, ADD 1 TO COUNT
        JMP   COUNT
DONE    BRK
```

Enter Program 5–6 into memory with CT1 = CT2 = 00 and run it. Note that we can add 1 to the data simply by incrementing the output port with INC $AC02.

PROGRAM 5–6

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A9	LDA #B0
0201	B0	

PROGRAM 5–6 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0202	8D		STA AC02
0203	02		
0204	AC		
0205	A9		LDA #7B
0206	7B		
0207	8D		STA AC00
0208	00		
0209	AC		
020A	A0	COUNT	LDY #CT1
020B	CT1		
020C	A2	DLY1	LDX #CT2
020D	CT2		
020E	CA	DLY2	DEX
020F	D0		BNE 020E
0210	FD		
0211	88		DEY
0212	D0		BNE 020C
0213	F8		
0214	AD		LDA AC02
0215	02		
0216	AC		
0217	C9		CMP #B9
0218	B9		
0219	F0		BEQ 0221
021A	06		
021B	EE		INC AC02
021C	02		
021D	AC		
021E	4C		JMP 020A
021F	0A		
0220	02		
0221	00	DONE	BRK

PROBLEM 5–9

Make Program 5–6 count down from 9 to 0 on the rightmost character.

PROBLEM 5–10

Write a program that counts up from 0 to F on the leftmost character. How would you make the program start over at 0 after displaying F?

CHARACTER SELECTION BY LOOKUP TABLE

The next task is to make the computer select the character position. Unfortunately, Table 5–3 is not as simple as Tables 5–4 and 5–5. Since it has several gaps, the conversion program would involve many comparisons.

An alternative is to simply put Table 5–3 in memory and use it as a lookup table. A program can then perform the conversion as follows:

1. Calculate the address of the desired activation code by adding the starting (*base*) address of the table to the character number (*index*).
2. Obtain the code by loading it from the calculated address.

We can combine these steps into a single LDA with indexed addressing. In that mode, the processor adds an index register to the address in the instruction. It then uses the sum as the address from which to load the accumulator. We refer to the address in the instruction as the *base address*, the index register as the *index*, and the sum (the address actually used to perform the operation) as the *effective address*.

The following program uses indexed addressing to convert a character number in 0040 into an activation code in 0041.

```
LDX  $40          ; GET CHARACTER NUMBER
LDA  $0380,X      ; GET ACTIVATION CODE FROM TABLE
STA  $41          ; SAVE ACTIVATION CODE
BRK
```

This program assumes that the activation code table is in memory starting at 0381. The table thus does not interfere with our programs. There is an offset of 1 here, since we have numbered the characters from 1 to 20, rather than from 0 to 19. Note that you must enter both Program 5–7 (starting at 0200) and Table 5–3 (starting at 0381) into memory.

PROGRAM 5–7

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A6	LDX 40
0201	40	
0202	BD	LDA 0380,X
0203	80	
0204	03	
0205	85	STA 41
0206	41	
0207	00	BRK
0381	7B	
0382	7A	

PROGRAM 5–7 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0383	79	
0384	78	
0385	77	
0386	76	
0387	75	
0388	74	
0389	6F	
038A	6E	
038B	6D	
038C	6C	
038D	5F	
038E	5E	
038F	5D	
0390	5C	
0391	3F	
0392	3E	
0393	3D	
0394	3C	

We use LDA with absolute indexed addressing, since the table is not on page 0. When the processor executes an indexed LDA, it both calculates the required address and loads the data.

Program 5–7 works as follows (assuming that 0040 contains 06):

1. LDX $40 loads register X with the character number (06).
2. LDA $0380,X first calculates the effective address by adding register X (06) to the base address (0380). The sum is 0380 + 06 = 0386. The processor then loads the accumulator from 0386; 0386 contains 76, the activation code for character 6 in the on-board display.

To produce a visible result, use the following program, which shows 0 on the activated display.

```
        LDA  #$B0          ; DATA = ASCII ZERO
        STA  AC02
        LDX  $40           ; GET CHARACTER NUMBER
        LDA  $0380,X       ; GET ACTIVATION CODE FROM TABLE
        STA  AC00          ; ACTIVATE CHARACTER
HERE    JMP  HERE          ; WAIT FOREVER
```

Program 5–8 is the mnemonic-entry version.

PROGRAM 5–8

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)*
0200	A9		LDA #B0
0201	B0		
0202	8D		STA AC02
0203	02		
0204	AC		
0205	A6		LDX 40
0206	40		
0207	BD		LDA 0380,X
0208	80		
0209	03		
020A	8D		STA AC00
020B	00		
020C	AC		
020D	4C	HERE	JMP 020D
020E	0D		
020F	02		

MOVING A CHARACTER ACROSS THE DISPLAY

We can use Table 5–3 to move a character across the display. The following program moves a 0 from left to right. Since we are changing activation codes, we must deactivate all displays by storing FF in AC00 before proceeding to the next character.

```
                LDA   #0          ; CHARACTER NUMBER = ZERO
                STA   $40
                LDA   #$B0        ; DATA = ASCII ZERO
                STA   $AC02
DSPLC           INC   $40         ; MOVE TO NEXT POSITION
                LDX   $40         ; HAS DATA REACHED FAR RIGHT?
                CPX   #21
                BEQ   DONE        ; YES, DONE
                LDA   $0380,X     ; NO, GET NEXT ACTIVATION CODE
                STA   $AC00
                LDY   #CT1        ; WASTE SOME TIME
DLY1            LDX   #CT2
DLY2            DEX
                BNE   DLY2
                DEY
                BNE   DLY1
                LDA   #$FF        ; DEACTIVATE ALL DISPLAYS
                STA   $AC00
```

```
                JMP   DSPLC              ; CONTINUE MOVING THE ZERO
DONE            BRK
```

Program 5–9 is the mnemonic-entry version. Remember to place Table 5–3 in 0381 through 0394. CPX #21 subtracts 21 decimal (15 hex) from register X; it sets the flags but does not change X. Note that the comparison must be with 21, not with 20, since INC precedes it. We must save the count temporarily in 0040 because the delay routine uses register X.

Enter Program 5–9 into memory, setting CT1 = CT2 = 00, and run it. Note that the line of zeros extends to the right as the program proceeds; the previous characters are not erased.

PROGRAM 5–9

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A9		LDA #00
0201	00		
0202	85		STA 40
0203	40		
0204	A9		LDA #B0
0205	B0		
0206	8D		STA AC02
0207	02		
0208	AC		
0209	E6	DSPLC	INC 40
020A	40		
020B	A6		LDX 40
020C	40		
020D	E0		CPX #15
020E	15		
020F	F0		BEQ 0229
0210	18		
0211	BD		LDA 0380,X
0212	80		
0213	03		
0214	8D		STA AC00
0215	00		
0216	AC		
0217	A0		LDY #CT1
0218	CT1		
0219	A2	DLY1	LDX #CT2
021A	CT2		
021B	CA	DLY2	DEX
021C	D0		BNE 021B
021D	FD		
021E	88		DEY

PROGRAM 5–9 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
021F	D0		BNE 0219
0220	F8		
0221	A9		LDA #FF
0222	FF		
0223	8D		STA AC00
0224	00		
0225	AC		
0226	4C		JMP 0209
0227	09		
0228	02		
0229	00	DONE	BRK

PROBLEM 5–11

Write a program that moves a zero left across the display.

PROBLEM 5–12

Write a program that changes the digit as it moves. Make the digit increase from 0 to 9 as it moves across the leftmost nine characters. That is, the program should show 0123456789 at the left end of the display just before it returns control to the monitor.

PROBLEM 5–13

Make the program blank the current character (by storing A0 hex in it) before proceeding to the next character. A0 hex is an ASCII space (see Appendix 2) with bit 7 set. How would you make your program move the 0 continuously around the display (that is, make it appear at the left end after it disappears from the right end)?

KEY POINT SUMMARY

1. Most output devices (and observers) require data to be available for a long time by processor standards. The processor must not change the data too frequently.
2. Outputs must usually be converted into the forms required by peripherals.
3. Output transfers generally involve control signals as well as data. These control signals may select peripherals or control their operations.
4. Lookup tables often simplify code conversions. Such tables simply contain all codes organized in a convenient manner. They are easy and quick to use but may occupy a large amount of memory.

5. In indexed addressing, the processor calculates the actual (effective) address to be used in executing the instruction. The calculation involves adding an index register to the address included in the instruction. Indexed addressing lets the programmer implement lookup tables and access successive elements in a table.

LABORATORY 6

PROCESSING DATA ARRAYS

PURPOSE

To learn how to process data arrays.

REFERENCE MATERIALS

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 5.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, 1982, pp. 29–34, 39–40, 204–229, 382–414.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 168–170 (index register), 338–340 (compare instructions), 367–368 (indirect addressing).

W. J. WELLER, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapters 5, 10.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, pp. 6–20 to 6–29.

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1979, Chapters 6, 7, Appendix G.

WHAT YOU SHOULD LEARN

1. What identifies elements of an array.
2. The most efficient way to process arrays using the 6502.
3. How to perform a summation.
4. How to use a terminator.
5. How to determine whether numbers are within limits.
6. How to display a message.
7. How to use indirect indexed (postindexed) addressing.

TERMS

Array collection of related data items.

Borrow bit that is set (1) if the result of a subtraction is negative and cleared (0) if it is positive or 0. Borrows are used to subtract numbers that are too long for a single operation.

Checksum logical sum used to guard against errors.

Indirect address address that contains the address of the data, as opposed to a direct address that contains the actual data.

Indirect indexed addressing addressing mode in which the effective address is determined by first obtaining the base address indirectly and then indexing from it. Also known as *postindexing*, since the indexing is performed after the indirection.

Inverted borrow bit that is cleared (0) if the result of a subtraction is negative and set (1) if it is positive or 0.

Limit checking determining if data is within limits, that is, below an upper threshold and above a lower threshold. This procedure can be used to discard invalid data resulting from operator or communications errors. Typical examples of such data are a transaction dated February 30 and a room temperature setting of 70°C (instead of F).

Logical sum binary sum with no carries between bit positions.

Postindexing *see Indirect indexed addressing.*

Terminator item that marks the end of an array.

DATA ARRAYS

Most computing tasks involve applying the same instructions to collections of related data, or *arrays*. Typical array operations are calculating averages and other statistics, finding the

largest element for sealing, organizing data for storage on tape or disk, editing, sorting, arranging sequenees of operations, and scareling for commands.

The elements of arrays are usually stored in eonseeutive memory addresses. Two items are then neeled to reaeh a particular element:

1. The array's starting (*base*) address.
2. The element number, or *index*.

We often refer mathematically to an element as A_i, where A identifies the entire array (i.e., base address) and *i* identifies the partieular element (i.e., index).

Flexible addressing modes are the keys to processing arrays. One sequence of instruetions should be able to proeess any element. Otherwise, minor changes in the locations or lengths of the arrays will require major revisions in the program. A flexible addressing mode sueh as indexing allows an instruetion to use different effeetive addresses at different times.

PROBLEM 6–1

Which of these instruetions could handle any element of an array? Why?

a. LDA $40
b. LDX #$A3
c. LDA $0340,X
d. LDX $40

Which instruction can load data from different memory locations at different times even if the program is stored in read-only memory?

PROBLEM 6–2

If an array starts at address BASE and eaeh element occupies one location, which address eontains the second element? Assume that BASE eontains the "zeroth" element. Whieh address contains the *j*th element?

PROBLEM 6–3

How do the answers to Problem 6–2 change if each element oeeupies two memory loeations? What if eaeh element oeeupies *k* loeations?

PROBLEM 6–4

We ean storc a two-dimensional array eithcr by row or by eolumn. For example, we can storc an array A with *m* rows and *n* eolumns by row, starting with row 1. Denoting the element in row *i* and eolumn *j* as A_{ij}, the order in memory is: A_{11}, A_{12}, A_{13}, . . ., A_{1n},

A_{21}, A_{22}, A_{23}, . . ., A_{m1}, A_{m2}, A_{m3}, . . ., A_{mm}. If A_{11} is in address B, which address contains A_{23}? Which address contains element A_{ij}? What is the lowest address occupied by A_{ij} if each element occupies k memory locations?

PROBLEM 6–5

Assume that an array contains the angles at which a vehicle should move (0 to 359 degrees) and the number of minutes (0 to 59) for which it should travel at each angle. Each entry consists of 3 bytes; the first 2 contain the angle and the third the travel time at that angle. If the first angle is in addresses BASE and BASE + 1, where would you find:

a. The third angle?
b. The travel time at the fifth angle?
c. The sixth angle?

PROCESSING ARRAYS WITH THE 6502 MICROPROCESSOR

The fastest way to process arrays with the 6502 is as follows (see Figure 6–1 for a flowchart):

1. Load an index register with the array's length and work backward (i.e., from the highest address down). This is more efficient than working forward because the loop can end when it decrements the index register to 0. No comparison instruction is necessary.
2. Refer to an element by indexing from a base one less than the lowest occupied address. A typical instruction is ADC START − 1,X where START is the lowest occupied address. The − 1 is necessary because the loop ends when it decrements the index register to 0. Thus 1 is the smallest index ever used.
3. Access other elements either by using different bases or by changing the index register. For example, ADC START + 5,X adds to the accumulator an element located 6 bytes ahead of where the processor is working.
4. Use DEX or DEY to proceed to the next element.

This approach assumes a fixed base address. We will show later how to remove this restriction by using postindexed (indirect indexed) addressing. The computer can operate on any element simply by using the appropriate base; for example:

LDA START − 1,X loads the accumulator with the element from address START − 1 + (X).
EOR START + 9,X logically EXCLUSIVE ORs the accumulator with the element from address START + 9 + (X).

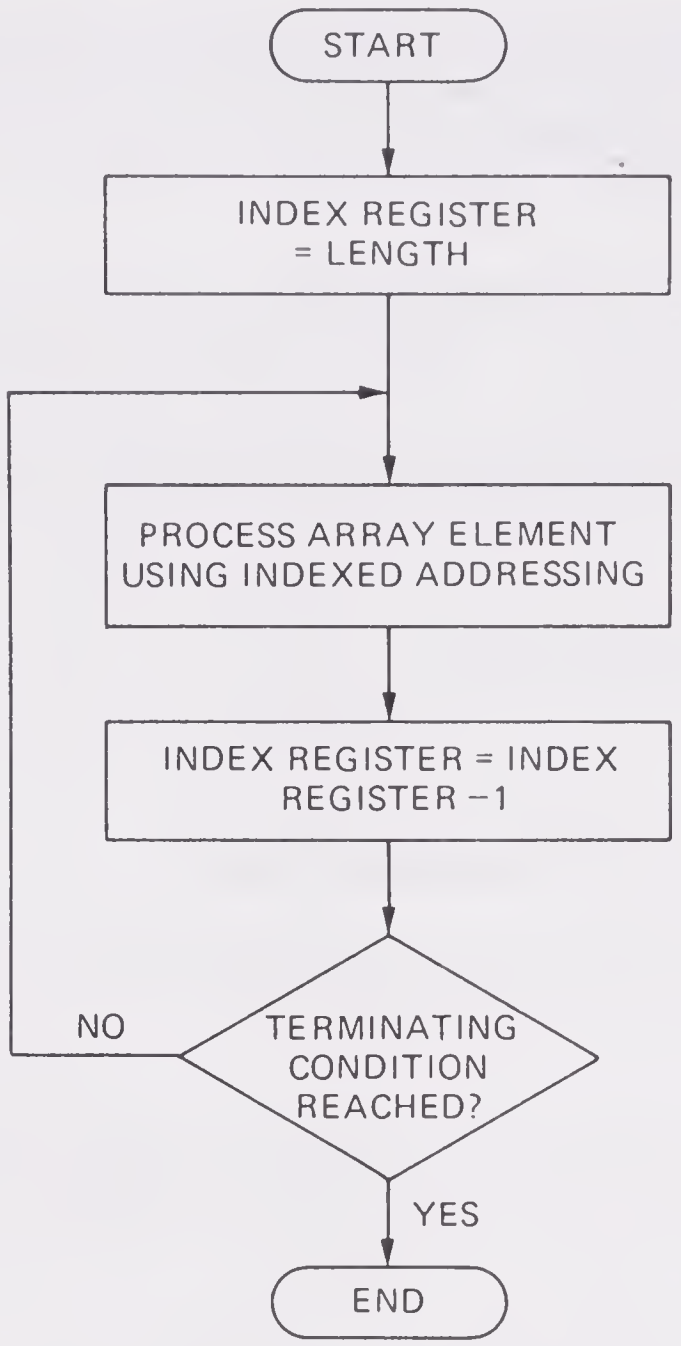


FIGURE 6–1. Array processing with the 6502 microprocessor.

PROBLEM 6–6

Write a program that logically ANDs location START + 8 + (X) with START − 1 + (X) and stores the result in START + 8 + (X). START is a fixed address.

Example: START = 0340
(X) = 06
Result: (034E) = (034E) AND (0345)

Remember that the parentheses mean "contents of."

SUM OF DATA

A simple example of array processing is summing the elements. This is an essential step in calculating averages, variances, or numerical integrals. The following program assumes an array consisting of four elements in 0340 through 0343 (see Figure 6–2 for a flowchart):

```
        LDX  #4              ; INDEX = LENGTH
        LDA  #0              ; CLEAR SUM INITIALLY
ADDELM  CLC                  ; CARRY = 0 ALWAYS
        ADC  $033F,X         ; ADD ELEMENT TO SUM
        DEX
        BNE  ADDELM          ; CONTINUE THROUGH ALL ELEMENTS
        STA  $40             ; SAVE SUM
        BRK
```

Program 6–1 is the mnemonic-entry version. Run it with the following data:

(0340) = 07
(0341) = 23
(0342) = 31
(0343) = 20
Result: (0040) = 7B

Remember that the numbers are hexadecimal. Change 0342 to F1 and run the program again. What is the result, and why? Note that we must clear CARRY in each iteration, so it does not affect the addition.

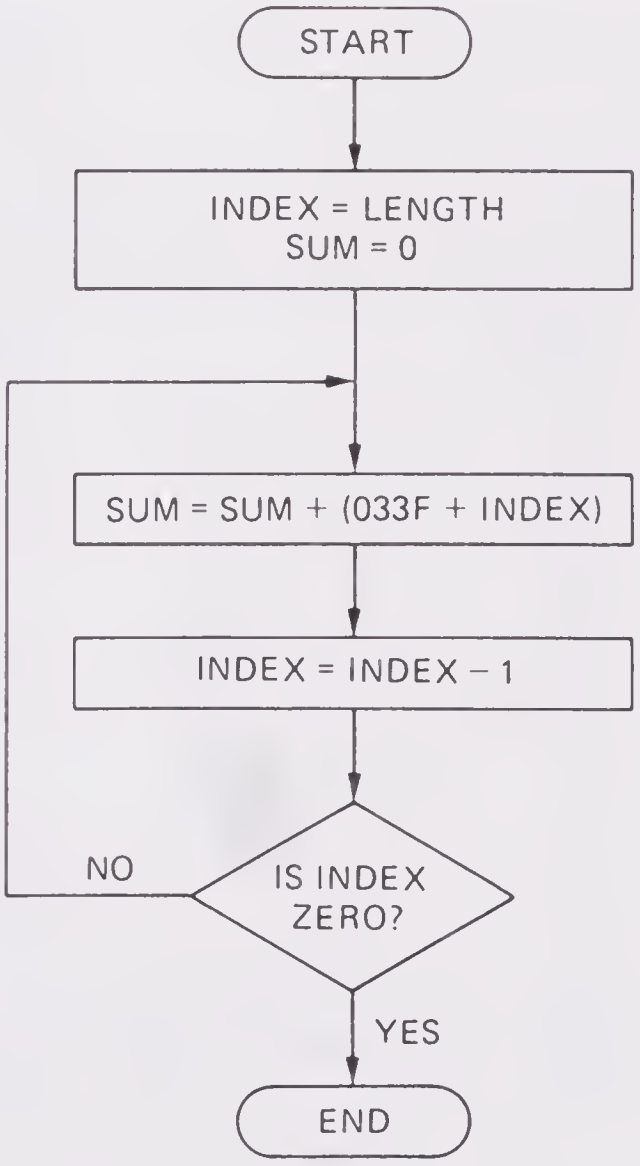


FIGURE 6–2. Flowchart for summation program.

PROBLEM 6–7

Write a program that sums six elements starting at 0340.

Sample Problem

(0340) = 07
(0341) = 23
(0342) = 31
(0343) = 20
(0344) = 16
(0345) = 38
Result: (0040) = C9

PROBLEM 6–8

Make Program 6–1 EXCLUSIVE OR the elements together instead of adding them. The result, called a *logical sum* or *checksum*, is often used to detect errors in tape or disk records.

Sample Problem

(four elements starting at 0340, result in 0040)

PROGRAM 6–1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #04
0201	04		
0202	A9		LDA #00
0203	00		
0204	18	ADDELM	CLC
0205	7D		ADC 033F,X
0206	3F		
0207	03		
0208	CA		DEX
0209	D0		BNE 0204
020A	F9		
020B	85		STA 40
020C	40		
020D	00		BRK

(0340) = 07
(0341) = 23
(0342) = 31
(0343) = 20
Result: (0040) = 35

PROBLEM 6–9

Extend Program 6–1 to save the carries and store the 16-bit sum in 0040 and 0041 (more significant byte in 0041).

Sample Problem

(0340) = F7
(0341) = 23
(0342) = 31
(0343) = 20
(0344) = 16
Result: (0040) = 81 (less significant byte of sum)
(0041) = 01 (more significant byte of sum)

USING A TERMINATOR

If you are not sure how long the array is (or do not want to count the elements), you can follow the data with a special marker called a *terminator*. It must have a value that cannot be a real data item. In a summation, 0 is a good choice because it does not affect the sum. The program using 0 as a terminator is (see Figure 6–3 for a flowchart):

```
            LDX  #0              ; INDEX = ZERO
            TXA                  ; SUM = ZERO
ADDELM      LDY  $0340,X         ; IS ELEMENT ZERO?
            BEQ  DONE            ; YES, DONE
            CLC                  ; NO, ADD ELEMENT TO SUM
            ADC  $0340,X
            INX
            JMP  ADDELM
DONE        STA  $40             ; SAVE SUM
            BRK
```

Program 6–2 is the mnemonic-entry version. Here we must work forward, since the terminator is at the end. Of course, we could put the terminator at the beginning and work backward.

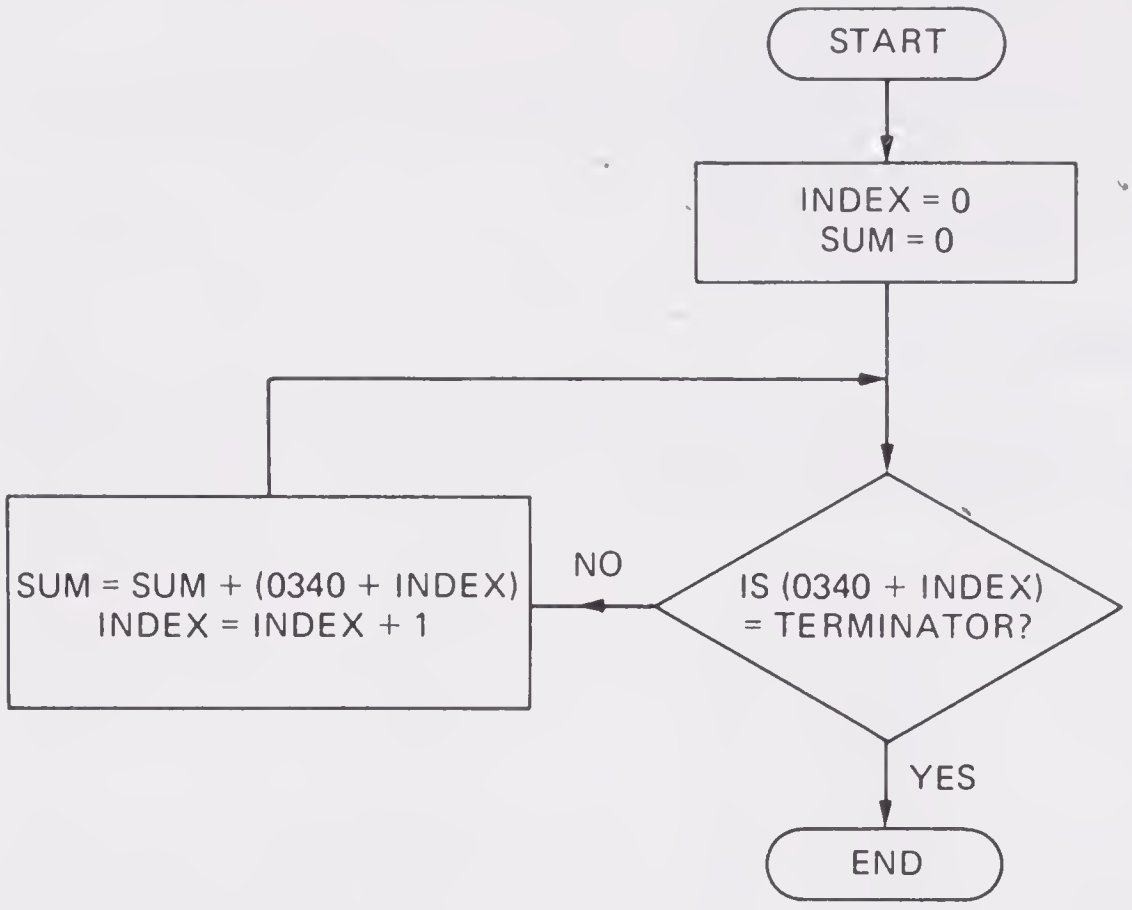


FIGURE 6–3. Flowchart for summation program with terminator.

PROGRAM 6–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #00
0201	00		
0202	8A		TXA
0203	BC	ADDELM	LDY 0340,X
0204	40		
0205	03		
0206	F0		BEQ 0210
0207	08		
0208	18		CLC
0209	7D		ADC 0340,X
020A	40		
020B	03		
020C	E8		INX
020D	4C		JMP 0203
020E	03		
020F	02		
0210	85	DONE	STA 40
0211	40		
0212	00		BRK

Run Program 6–2 with the following data:

(0340) = 07
(0341) = 23
(0342) = 31
(0343) = 20
(0344) = 16
(0345) = 38
(0346) = 00
Result: (0040) = C9

What happens if you set (0343) = 00? What are the advantages and disadvantages of using a terminator as compared to counting the elements? Which approach results in faster programs? Which makes data entry simpler?

PROBLEM 6–10

If 0 is an acceptable data value, you must use something else as the terminator. Revise Program 6–2 to use FF as a terminator. Which terminator should you use if the data values are the numbers of characters received from a 10-cps (characters per second) teletypewriter in 1 s?

LIMIT CHECKING

We often want a computer to test the validity of data, that is, whether it is within certain limits, below a threshold, or has an allowed value. Determining if data is within limits is called *limit checking*. The key instruction here is a comparison (CMP, CPX, or CPY) that subtracts a memory location from a register. Comparisons affect the flags but do not save the result.

As noted in Laboratory 5, CARRY indicates which operand is larger after a comparison involving unsigned numbers. We know that

$$\text{CARRY} = 1 \text{ if } (\text{REG}) \geq (\text{M})$$

$$\text{CARRY} = 0 \text{ if } (\text{REG}) < (\text{M})$$

where REG refers to the accumulator or index register and M to the memory location. We refer to CARRY as an *inverted borrow*, since it is cleared (0) if the subtraction requires a borrow and set (1) if it does not.

The following program sums six elements but ignores ones that are 80 hex or above (i.e., have a most significant bit of 1).

```
LDX   #6                ; INDEX = LENGTH
LDA   #0                ; CLEAR SUM INITIALLY
```

```
ADDELM      LDY   $033F,X        ; COMPARE ELEMENT TO THRESHOLD
            CPY   #$80
            BCS   COUNT          ; IGNORE IF VALUE ABOVE THRESHOLD
            ADC   $033F,X        ; ADD ELEMENT (CARRY IS 0)
COUNT       DEX
            BNE   ADDELM
            STA   $40
            BRK
```

Note that CARRY must be 0 if the program reaches ADC $033F,X, since otherwise BCS would have branched. To make the threshold itself valid (i.e., ignore elements above 80 hex instead of 80 hex or above), simply replace CPY #$80 with CPY #$81. Program 6–3 is the mnemonic-entry version; run it with the following data:

(0340) = 07
(0341) = 20
(0342) = F1
(0343) = 3C
(0344) = 80
(0345) = 73
Result: (0040) = D6

PROGRAM 6–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #06
0201	06		
0202	A9		LDA #00
0203	00		
0204	BC	ADDELM	LDY 033F,X
0205	3F		
0206	03		
0207	C0		CPY #80
0208	80		
0209	B0		BCS 020E
020A	03		
020B	7D		ADC 033F,X
020C	3F		
020D	03		
020E	CA	COUNT	DEX
020F	D0		BNE 0204
0210	F3		
0211	85		STA 40
0212	40		
0213	00		BRK

PROBLEM 6–11

Make Program 6–3 ignore elements that are 80 hex or above or 20 hex or below.

Sample Problem

(0340) = 07
(0341) = 20
(0342) = F1
(0343) = 3C
(0344) = 80
(0345) = 73
Result: (0040) = AF

Limit checking is often used to reject measurements that are far away from the majority (and therefore suspect). Many data analysis routines discard the highest and lowest values before averaging or performing other functions to eliminate readings that may be the result of noise, equipment malfunction, or human error. Limit checking also guards against errors such as an operator specifying a nonexistent time (e.g., 8:80 instead of the intended 8:00) or an unrealistic or impossible parameter value (e.g., an automobile speed of 1000 km/hr instead of the intended 100 km/hr).

DISPLAYING A MESSAGE

We can use array methods to show a message on the on-board display. We need a data array starting at 0341 and Table 5–3 (the character activation codes) starting at 0381. The following program (see Figure 6–4 for a flowchart) also assumes that 0040 contains the message length.

```
        LDX  $40          ; STARTING INDEX = MESSAGE LENGTH
DSPLC   LDA  $0340,X      ; GET A CHARACTER FROM MESSAGE
        STA  $AC02        ; SEND CHARACTER TO DISPLAY
        LDA  $0380,X      ; GET AN ACTIVATION CODE
        STA  $AC00        ; ACTIVATE A CHARACTER
        LDA  #$FF         ; HOLD DATA, DEACTIVATE DISPLAY
        STA  $AC00
        DEX
        BNE  DSPLC
HERE    JMP  HERE         ; WAIT FOREVER
```

Program 6–4 is the mnemonic-entry version of the message display program. Run it with the following data:

(0040) = 09 (number of characters in message)
(0341) = C3 (leftmost character)

(0342) = D0
(0343) = D5
(0344) = A0
(0345) = C9
(0346) = D3
(0347) = A0
(0348) = CF
(0349) = CE (rightmost character)

You can use the ASCII table in Appendix 2 to create messages, but remember to set bit 7 of each character.

PROBLEM 6–12

Place a nested delay routine inside the loop in Program 6–4. Set CT1 (outer loop constant) = CT2 (inner loop constant) = 00 and run the program. Be sure to save the character number (Program 6–4 leaves it in X). Describe what you see.

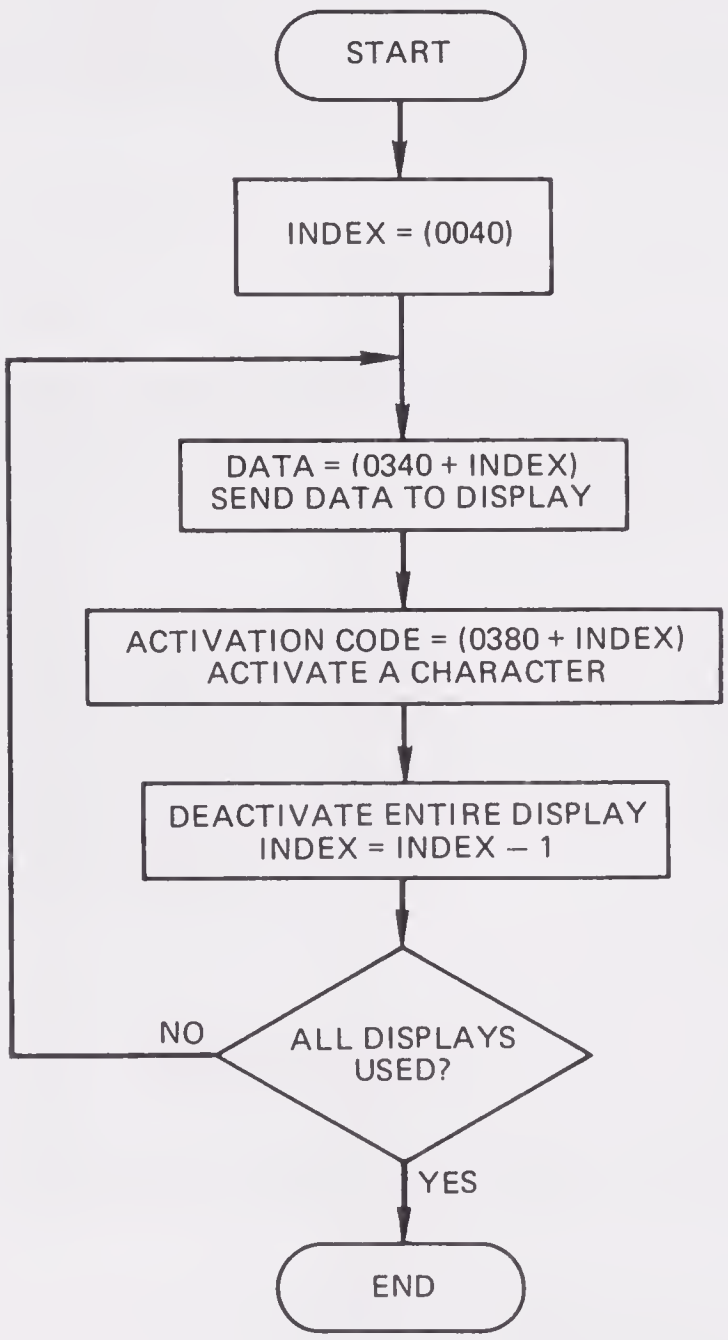


FIGURE 6–4. Flowchart for displaying a message.

PROGRAM 6–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A6		LDX 40
0201	40		
0202	BD	DSPLC	LDA 0340,X
0203	40		
0204	03		
0205	8D		STA AC02
0206	02		
0207	AC		
0208	BD		LDA 0380,X
0209	80		
020A	03		
020B	8D		STA AC00
020C	00		
020D	AC		
020E	A9		LDA #FF
020F	FF		
0210	8D		STA AC00
0211	00		
0212	AC		
0213	CA		DEX
0214	D0		BNE 0202
0215	EC		
0216	4C	HERE	JMP 0216
0217	16		
0218	02		

Now run the program repeatedly with the following series of values for CT1: 80, 40, 20, 10, 08, 04, 02, 01. Explain what happens. Obviously, it does not take much computing power to make a display look continuous to a human observer.

PROBLEM 6–13

Write a program that produces a "newspanel" or "Times Square" display in which the message appears to move from right to left. Your program should:

1. Start by placing ASCII spaces (A0 hex) on all characters.
2. After a delay, put the first element of the message on the rightmost character and ASCII spaces everywhere else.
3. Continue this process until the message has moved all the way across the display. Then start over.

Your data should consist of a set of ASCII spaces, the message, and another set of ASCII spaces. You must save a starting index that tells the computer where to find the display data for a particular iteration.

VARYING THE BASE ADDRESS

So far, we have assumed a fixed base address. Programs that make this assumption clearly lack generality, since they always work on data at a fixed place in memory. If the base address were a variable, we could then tell the program where the data is. This would make it unnecessary to move the data or change the program for a computer with a different arrangement of memory addresses.

The 6502's indirect indexed addressing (postindexing) provides the required capability. In this mode, the processor obtains the base address from two successive memory locations on page 0. It then adds register Y to the base address to determine the effective address. For example, the instruction

```
LDA  ($40),Y
```

loads the accumulator from the effective address obtained by adding register Y to the base address in 0040 and 0041. If (0040) = 80, (0041) = 03, and (Y) = 3C, the base address is 0380 and the effective address is 0380 + 3C = 03BC. Thus 0040 and 0041 indicate where the array starts.

We can easily change Program 6–1 to use indirect indexed addressing. We must use register Y instead of X and ADC with indirect indexed addressing instead of absolute indexed addressing. The revised program is

```
        LDY  #4          ; INDEX = LENGTH
        LDA  #0          ; CLEAR SUM INITIALLY
ADDELM  CLC              ; CARRY = 0 ALWAYS
        ADC  ($40),Y     ; ADD ELEMENT TO SUM
        DEY
        BNE  ADDELM
        STA  $42         ; SAVE SUM
        BRK
```

Program 6–5 is the mnemonic-entry version using indirect indexed addressing. To have this program work on a particular array, we simply place its starting address minus 1 in 0040 and 0041. Enter and run Program 6–5 for the same sample cases we used with Program 6–1. Remember to load the starting address minus 1 (033F) upside down into 0040 and 0041 before execution.

PROBLEM 6–14

Make Program 6–2 use indirect indexed addressing and store the sum in 0042.

PROBLEM 6–15

Write a program that logically ANDs memory location START + 8 + (Y) with location START − 1 + (Y) and stores the result in START + 8 + (Y). Assume that address START is stored in 0040 and 0041. Be sure that your program works even if the operands are on different pages. For the sample cases, assume that (0040) = 00 and (0041) = 02 (i.e., START is 0200).

Sample Problems

a. Data: (Y) = 80
(027F) = 23
(0288) = 34
Result: (0288) = (0288) AND (027F) = 20

b. Data: (Y) = FE
(02FD) = C7
(0306) = 6D
Result: (0306) = (0306) AND (02FD) = 45

PROGRAM 6–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A0		LDY #04
0201	04		
0202	A9		LDA #00
0203	00		
0204	18	ADDELM	CLC
0205	71		ADC (40),Y
0206	40		
0207	88		DEY
0208	D0		BNE 0204
0209	FA		
020A	85		STA 42
020B	42		
020C	00		BRK

KEY POINT SUMMARY

1. Arrays are collections of elements with similar meanings or purposes. Each element is characterized by its position or index; the entire array is characterized by its base address. Thus to reach a particular element of an array, you must know the base address and the index.

2. The keys to processing arrays are:
   a. An index that determines which element is being processed.
   b. A flexible addressing mode that allows a single set of instructions to handle any element.
   c. A counter or terminator that can be used to determine the length of the array.
3. To process arrays with the 6502, you can use an index register to hold the index, indexed addressing to reach the data in memory, and another register or a memory location to hold the counter or terminator. An efficient approach is to use the starting address minus 1 as the base and work backward. You can then use the setting of the ZERO flag as an exit condition, since the program counts the index down to 0.
4. Comparison instructions can determine if an element is within limits. If the comparison's operands are unsigned, CARRY indicates which is larger. In the 6502, CARRY is an inverted borrow; it is set if no borrow is necessary and cleared if a borrow is required.
5. Indirect indexed addressing allows the 6502 to obtain a variable base address from two memory locations on page 0. This mode assumes the use of index register Y. The locations on page 0 thus indicate where the actual array starts.

LABORATORY 7

FORMING DATA ARRAYS

PURPOSE

To learn how to form data arrays.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 179–198.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 3–9 to 3–10, Chapter 5 (particularly pp. 5–20 to 5–22), pp. 11–123.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, pp. 12, 32–34, 51–52, 193–229.

W. J. WELLER, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapters 5, 10.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, pp. 5–31 to 5–32.

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, Chapters 6–7, Appendix G.

WHAT YOU SHOULD LEARN

1. How to use indexed addressing to form arrays.
2. How to fill an area of memory.
3. How to enter input data into an array.
4. How to access a specific element of an array.
5. How to keep counts or running totals in an array.
6. How to differentiate between logical and physical devices.
7. How to handle large arrays.

TERMS

Arithmetic shift a shift that does not change the sign (most significant) bit. A right arithmetic shift copies the sign bit into the positions to the right (called *sign extension*).

Clear set to 0.

Indexed indirect addressing an addressing mode in which the effective address is determined by indexing from the base address and then using the indexed address indirectly. Also called *preindexing*, since the indexing is performed before the indirection. Of course, the array starting at the base address must contain indirect addresses.

I/O device table a table that assigns actual (physical) devices or I/O subroutines to the device numbers (logical devices) to which programs refer.

Logical device the I/O device to which a program refers. The physical device is determined from an I/O device table.

Physical device an actual I/O device, as opposed to a logical device.

Preindexing *see Indexed indirect addressing.*

Rotate a shift that works as if the data were arranged in a circle, that is, as if the most significant and least significant bits were connected.

6502 INSTRUCTIONS

ROL(R) rotate left (right); shift the accumulator or a memory location left (right) one bit as if bits 0 and 7 were connected through the CARRY flag (see Figure E–2).

STANDARD PROCEDURE FOR FORMING ARRAYS

The arrays in Laboratory 6 do not, of course, appear magically in the computer's memory. In applications, the program must form the array before processing it; as with processing, this requires a base address and an index.

The standard procedure for forming an array is as follows (see Figure 7–1):

1. Initialization.

```
BASE = STARTING ADDRESS OF ARRAY
INDEX = 0
LENGTH = LENGTH OF ARRAY (if known)
```

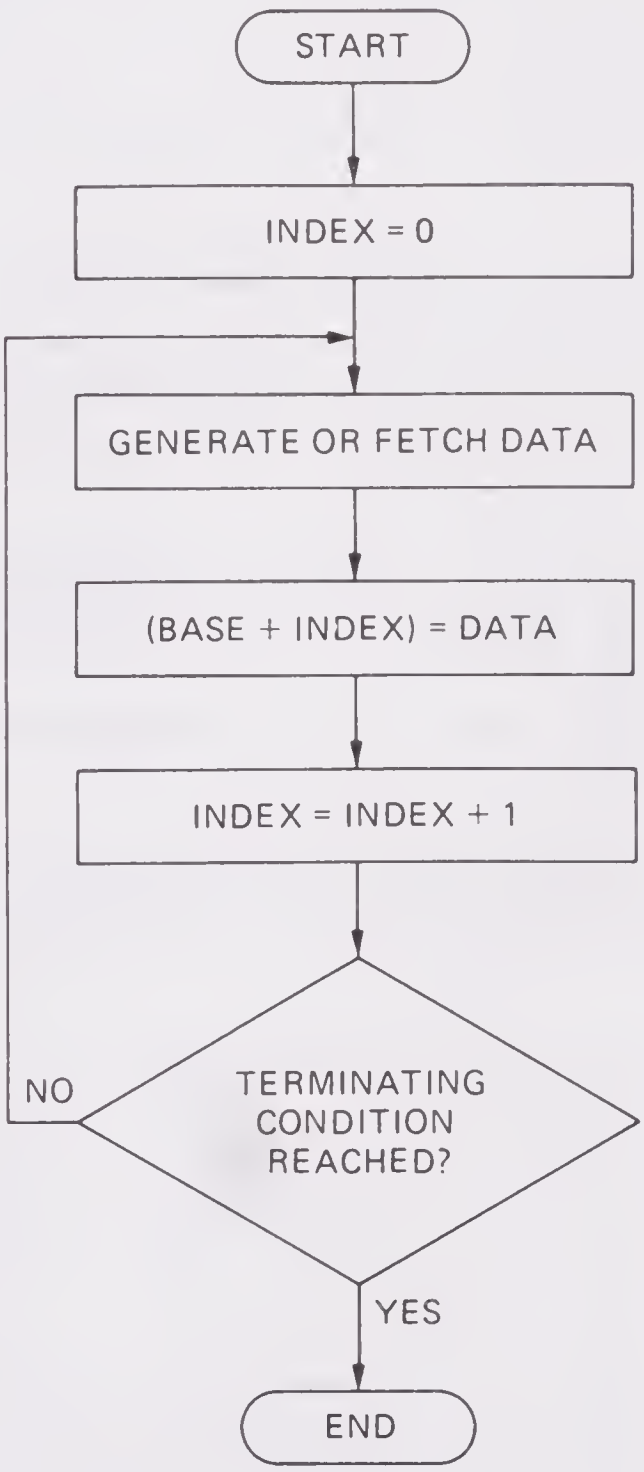


FIGURE 7–1. Flowchart for array formation.

2. Entering an element.

```
(BASE+INDEX) = DATA
INDEX = INDEX + 1
```

The data may be a constant, the result of a calculation, or an external input.

3. Conclusion.
   a. Maximum length: If INDEX = LENGTH, then DONE; otherwise, return to step 2.
   b. Terminator: If DATA = TERMINATOR, then DONE; otherwise, return to step 2.

Remember that on the 6502, we often find it more convenient to work through the array backward rather than forward. We can then count the index down to 0.

CLEARING AN ARRAY

A simple way to initialize an array is to clear its elements. This is a natural starting point for accumulating totals or test results. Note that you cannot assume that an unused RAM location contains 0; it could start with any value whatsoever when power is applied. The following program clears 0340 through 0347:

```
        LDA  #0              ; DATA = ZERO
        LDX  #8              ; NUMBER OF BYTES = 8
CLR1    STA  $033F,X         ; CLEAR A BYTE
        DEX
        BNE  CLR1            ; COUNT BYTES
        BRK
```

Program 7–1 is the mnemonic-entry version. Enter and run it.

PROGRAM 7–1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A9		LDA #00
0201	00		
0202	A2		LDX #08
0203	08		
0204	9D	CLR1	STA 033F,X
0205	3F		
0206	03		

PROGRAM 7–1 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0207	CA	DEX
0208	D0	BNE 0204
0209	FA	
020A	00	BRK

PROBLEM 7–1

Make Program 7–1 clear 0350 through 035F.

PROBLEM 7–2

Make Program 7–1 place (0040) in 0340 through a number of locations given by (0041). Does your program work properly if (0041) = 00?

Example: (0040) = 3F (value)
(0041) = 03 (number of locations)
Result: (0340) = 3F
(0341) = 3F
(0342) = 3F

The program should do nothing if (0041) = 00.

PLACING VALUES IN AN ARRAY

The next step is to place different values in different elements. The following program places the element numbers (1 through 8) in the corresponding positions (see Figure 7–2 for a flowchart). Program 7–2 is the mnemonic-entry version.

```
        LDX  #8              ; NUMBER OF BYTES = 8
LDIND   TXA                  ; ELEMENT = INDEX
        STA  $033F,X
        DEX
        BNE  LDIND
        BRK
```

Enter and run Program 7–2. It has practical value, since it creates an array of identification numbers. For example, assume that you have a set of pressure readings taken at different points in a chemical process. You could sort that set into descending order and use the identification numbers to keep track of where the readings were taken.

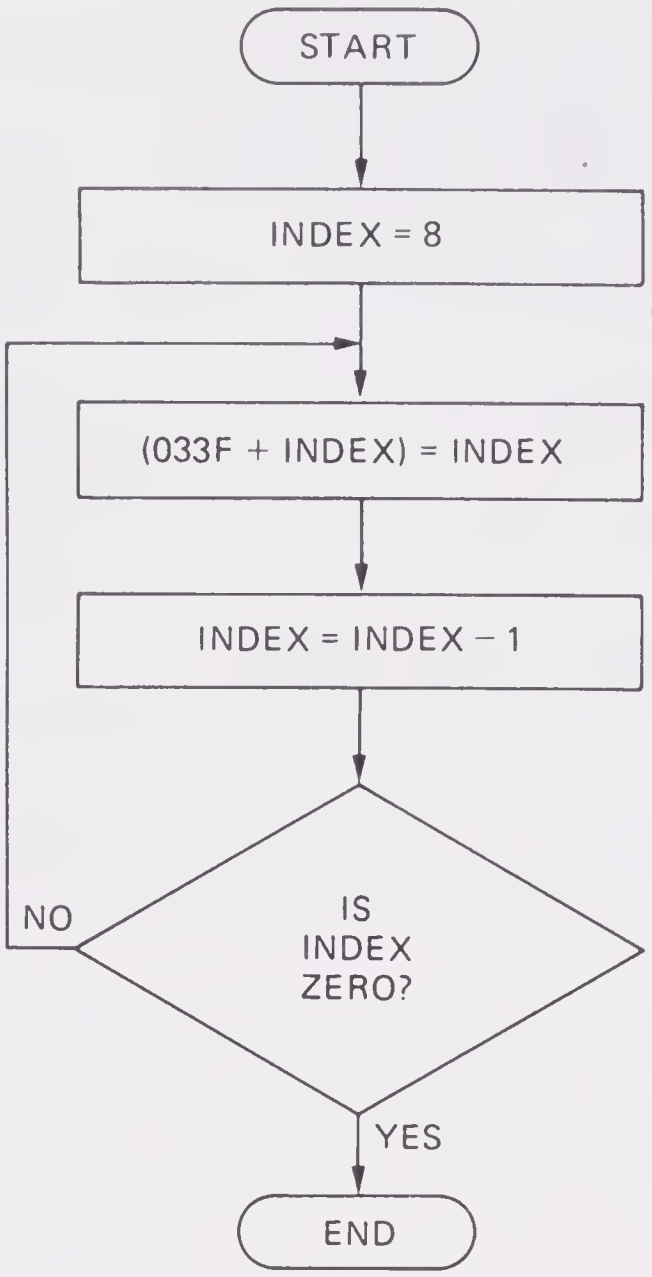


FIGURE 7–2. Flowchart for placing element numbers in an array.

The top line of the results would then show the highest value and where it occurred. For instance, you could start with

Position	Pressure
1	40
2	27
3	66
4	59

and end with the pressures arranged in descending order as

Position	Pressure
3	66
4	59
1	40
2	27

Without the array of identification numbers, we would not know where the highest pressure occurred.

PROGRAM 7–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #08
0201	08		
0202	8A	LDIND	TXA
0203	9D		STA 033F,X
0204	3F		
0205	03		
0206	CA		DEX
0207	D0		BNE 0202
0208	F9		
0209	00		BRK

PROBLEM 7–3

Start with 1 and make each subsequent element twice its predecessor; that is,

(0340) = 01
(0341) = 02
(0342) = 04
(0343) = 08
(0344) = 10
(0345) = 20
(0346) = 40
(0347) = 80

PROBLEM 7–4

Create the following sequence:

(0340) = 80 (10000000 binary)
(0341) = C0 (11000000 binary)
(0342) = E0 (11100000 binary)
(0343) = F0 (11110000 binary)
(0344) = F8 (11111000 binary)
(0345) = FC(11111100 binary)
(0346) = FE(11111110 binary)
(0347) = FF (11111111 binary)

What are these numbers if they are in two's complement form? An element can be obtained from its predecessor by means of a right *arithmetic shift*, since the sign (most significant) bit does not change. A right arithmetic shift requires an extra copy of bit 7, which you can produce with the following instructions:

```
TAY             ; SAVE ACCUMULATOR
ASL  A          ; MOVE BIT 7 TO CARRY
TYA             ; RESTORE ACCUMULATOR
ROR  A          ; SHIFT RIGHT WITH COPY OF BIT 7
```

ENTERING INPUT DATA INTO AN ARRAY

The next task is to form an array from input data entered on the switches attached to user VIA port A. The steps are as follows (see Figure 7–3):

1. Initialize the index and base address:

   BASE = STARTING ADDRESS − 1
   INDEX = LENGTH OF ARRAY

2. Wait for a switch to be closed.
3. Debounce the switch closure.
4. Identify the switch.
5. Place the switch number in the array:

   (BASE + INDEX) = SWITCH NUMBER

6. Update the index for the next entry:

   INDEX = INDEX − 1

7. Wait for all switches to be open.
8. Debounce the switch opening.
9. If INDEX is not 0, return to step 2.

The following program forms an array starting in 0340 from 4 switch closures (all switches must be opened between closures).

```
;
;    SET UP USER VIA PORT A FOR INPUT
;
                LDA   #0              ; MAKE PORT A INPUT
                STA   $A003
```

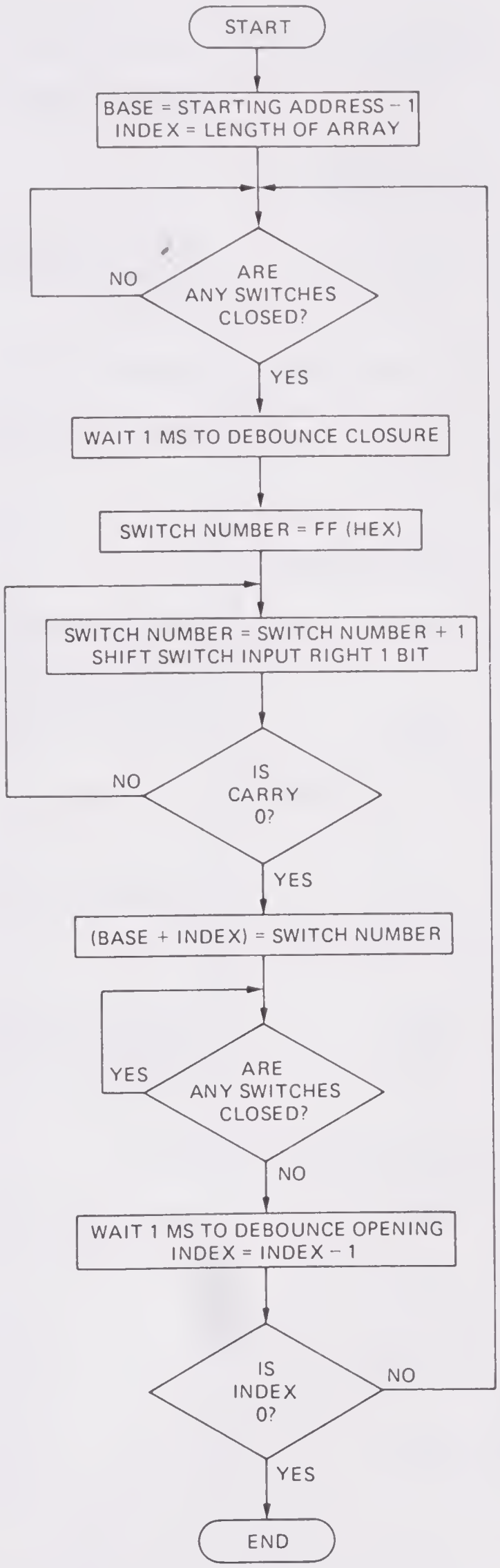


FIGURE 7–3. Flowchart for forming an array from switch inputs.

```
;
;     INITIALIZE INDEX TO LENGTH OF ARRAY
;
                LDX   #4              ; INDEX = ARRAY LENGTH
;
;     WAIT FOR SWITCH TO BE CLOSED
;
WAITC           LDA   $A001           ; READ DATA FROM SWITCHES
                CMP   #$FF            ; ARE ANY SWITCHES CLOSED?
                BEQ   WAITC           ; NO, WAIT
;
;     DEBOUNCE SWITCH CLOSURE WITH 1 MS DELAY
;
                LDY   #$C8            ; DELAY 1 MS AFTER CLOSURE
DLYC            DEY
                BNE   DLYC
;
;     IDENTIFY SWITCH BY SHIFTING INPUT
;
                LDY   #$FF            ; SWITCH NUMBER = -1
SRCHS           INY                   ; ADD 1 TO SWITCH NUMBER
                LSR   A               ; IS NEXT SWITCH CLOSED?
                BCS   SRCHS           ; NO, KEEP LOOKING
;
;     ENTER SWITCH NUMBER INTO ARRAY
;
                TYA
                STA   $033F,X         ; PUT SWITCH NUMBER IN ARRAY
;
;     WAIT FOR ALL SWITCHES TO OPEN
;
WAITO           LDA   $A001           ; READ DATA FROM SWITCHES
                CMP   #$FF            ; ARE ANY SWITCHES CLOSED?
                BNE   WAITO           ; YES, WAIT
;
;     DEBOUNCE SWITCH OPENING WITH 1 MS DELAY
;
                LDY   #$C8            ; DELAY 1 MS AFTER OPENING
DLYO            DEY
                BNE   DLYO
;
;     COUNT SWITCH CLOSURES
;
                DEX
                BNE   WAITC
                BRK
```

Program 7–3 is the mnemonic-entry version; enter and run it. Use the following sequence of switch closures: 5, 7, 0, 3. Remember to open all switches after each closure. The result should be:

(0340) = 03
(0341) = 00
(0342) = 07
(0343) = 05

PROGRAM 7–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A9		LDA	#00
0201	00			
0202	8D		STA	A003
0203	03			
0204	A0			
0205	A2		LDX	#04
0206	04			
0207	AD	WAITC	LDA	A001
0208	01			
0209	A0			
020A	C9		CMP	#FF
020B	FF			
020C	F0		BEQ	0207
020D	F9			
020E	A0		LDY	#C8
020F	C8			
0210	88	DLYC	DEY	
0211	D0		BNE	0210
0212	FD			
0213	A0		LDY	#FF
0214	FF			
0215	C8	SRCHS	INY	
0216	4A		LSR	A
0217	B0		BCS	0215
0218	FC			
0219	98		TYA	
021A	9D		STA	033F,X
021B	3F			
021C	03			
021D	AD	WAITO	LDA	A001
021E	01			

PROGRAM 7–3 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
021F	A0			
0220	C9		CMP	#FF
0221	FF			
0222	D0		BNE	021D
0223	F9			
0224	A0		LDY	#C8
0225	C8			
0226	88	DLY0	DEY	
0227	D0		BNE	0226
0228	FD			
0229	CA		DEX	
022A	D0		BNE	0207
022B	DB			
022C	00		BRK	

PROBLEM 7–5

Revise Program 7–3 to exit when you close switch 0. Can you ever get a data entry of 0?

PROBLEM 7–6

Extend Program 7–3 to combine the four entries in 0340 through 0343 into two 2-digit numbers in 0061 and 0062. Load 0061 from 0340 (4 LSBs) and 0341 (4 MSBs); load 0062 from 0342 (4 LSBs) and 0343 (4 MSBs).

Example: Switches closed are 7, 3, 4, 2.
(0340) = 02
(0341) = 04
(0342) = 03
(0343) = 07
Result: (0061) = 42
(0062) = 73

Note how similar this process is to the entry of a 4-digit hexadecimal address from a keyboard. Remember that keys are simply binary switches.

ACCESSING SPECIFIC ELEMENTS

Still another problem is how to find a specific element of an array. This is essential when a program must count events (number of transactions of a particular type or number of

activations of a particular sensor) or must accumulate data properly (e.g., total for a particular account, test point, or station). For example, the following program clears one element of an array starting at 0340. 0041 contains the element number.

Examples

1. Data: (0041) = 02
   Result: [0340 + (0041)] = (0342) = 00

2. Data: (0041) = 07
   Result: [0340 + (0041)] = (0347) = 00

```
LDX  $41              ; GET INDEX
LDA  #0               ; GET DATA
STA  $0340,X          ; CLEAR INDEXED ELEMENT
BRK
```

Program 7–4 is the mnemonic-entry version; enter it and run the two examples. Note how similar Program 7–4 is to the code conversion in Program 5–7.

PROGRAM 7–4

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A6	LDX 41
0201	41	
0202	A9	LDA #00
0203	00	
0204	9D	STA 0340,X
0305	40	
0206	03	
0207	00	BRK

PROBLEM 7–7

Revise Program 7–4 to add 1 to the element.

Example

(0041) = 04 (index)
(0344) = CF (original value)
Result: [0340 + (0041)] = (0344) = (0344) + 1 = D0

PROBLEM 7–8

Revise Program 7–4 to put (0042) in the element.

Example

(0041) = 06 (index)
(0042) = 3F (value)
Result: [0340 + (0041)] = (0346) = (0042) = 3F

How would you make your program replace the old value only if the new one is larger? Assume that the numbers are unsigned. This procedure would be necessary if the elements were the worst cases for a set of tests or scaling values for a set of plots.

PROBLEM 7–9

Revise Program 7–4 to clear a 2-byte element.

Example

(0041) = 03
Result: [0340 + 2 × (0041)] = (0346) = 00
[0340 + 2 × (0041) + 1] = (0347) = 00

(*Hint:* Use ASL to double the element number. Assume that (0041) is less than 128, so doubling it cannot produce a carry. Remember to clear both bytes of the element.)

COUNTING SWITCH CLOSURES

PROBLEM 7–10

Write a program that counts how many times each switch attached to user VIA port A is closed. Consider only single closures and assume that all switches must be opened between closures. The steps required are (see Figure 7–4):

1. Initialize the array of counts by clearing all elements.
2. Wait until a switch is closed.
3. Debounce the switch closure.
4. Identify the switch.
5. Add 1 to the count for that switch.
6. Wait until all switches are open.
7. Debounce the switch opening.
8. Return to step 2.

Use 0340 through 0347 for the array.

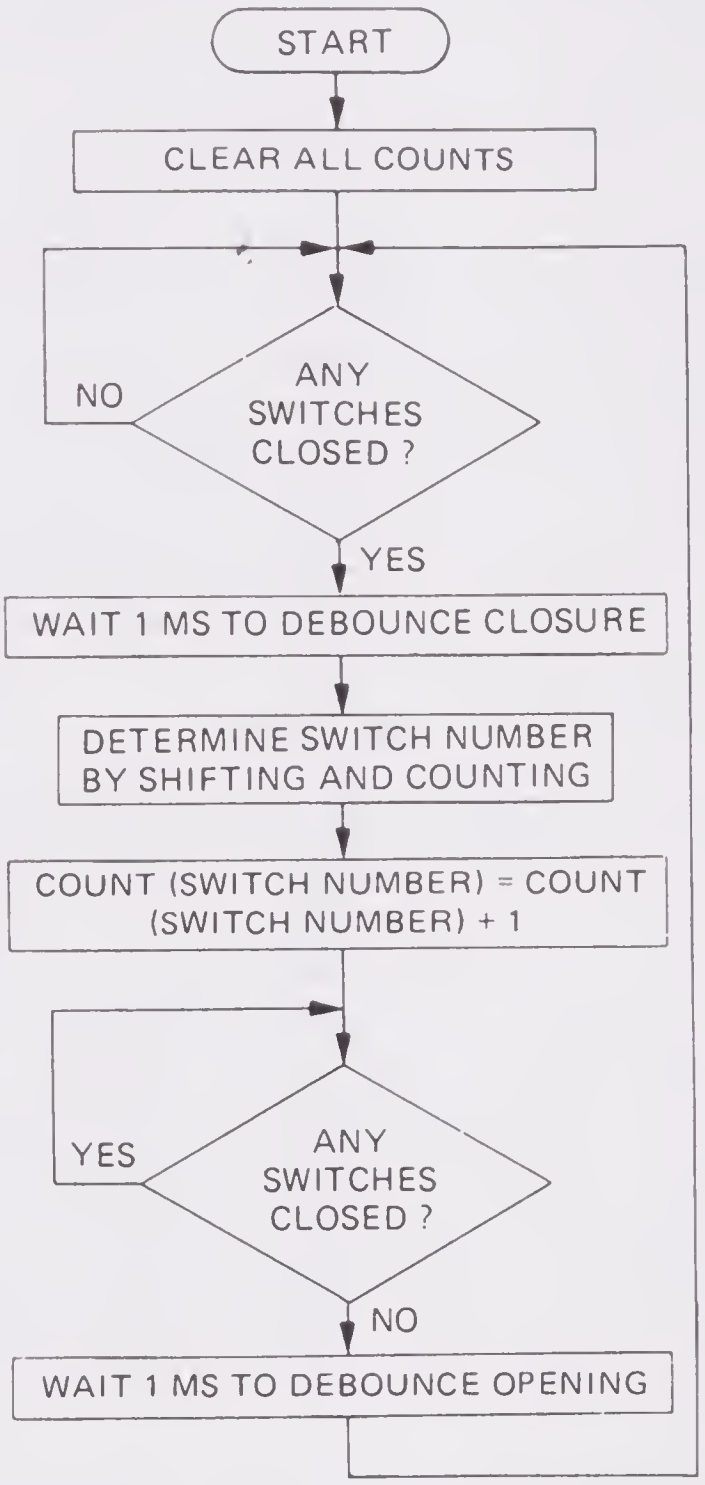


FIGURE 7–4. Flowchart for cumulative counts program.

ARRAYS OF ADDRESSES

Arrays may consist of addresses rather than data. By choosing a particular element from such an array, we can choose an address to use in data transfers. Thus we want a combination of indexing and indirection, just as in indirect indexed addressing, but here we want the indexing done first.

This combination, called *indexed indirect addressing* or *preindexing*, always uses register X and an address on page 0. A typical example is LDA ($40,X), which loads the accumulator from the address obtained indirectly by adding (X) to 0040. The result is

$$(A) = [(0040 + (X) + 1)\,(0040 + (X))]$$

where the indirect address, as usual, occupies two locations. If, for example, (X) = 04, the indirect address is in 0044 and 0045. If (0044) = 86, (0045) = 03, and (0386) = E4, the result of LDA ($40,X) is

$$(A) = [(0040 + 04 + 1)(0040 + 04)]$$
$$[(0045)(0044)] = (0386) = E4$$

Be careful with preindexing; the array on page 0 must contain addresses, and you can refer only to even-numbered elements. (Why?) The programmer must observe these constraints; the 6502 does not warn you about errors.

Indexed indirect addressing can be used to assign device numbers to I/O addresses or the starting addresses of I/O routines for a particular system. The programmer can then refer to I/O devices by number (e.g., "print on device 2" or "read data from device 3"). An *I/O device table* contains the actual I/O addresses corresponding to the numbers. We call the device number to which programs refer a *logical device* and the actual I/O device a *physical device*.

The advantages of maintaining this distinction are:

1. Programmers need not deal with actual I/O addresses. They can refer to devices by number without worrying about differences resulting from updates, model changes, or optional accessories.
2. Programs can be written using device numbers and can be made to work on a particular system by constructing an I/O device table. Such programs can be modified easily to work on computers with different peripherals.
3. A programmer can change the actual I/O addresses by modifying the device table. For example, you might want the results of a test run or a minor change shown on a CRT display rather than printed. Similarly, you could make a terminal simulate I/O devices that are unavailable or malfunctioning. Implementing these changes is much like switching the output of a stereo system from the front speaker to the back speaker.

The following program sends the data from 0040 to either device 0 (the LEDs attached to port B of the user VIA) or device 1 (the leftmost character of the on-board display). The device table is in 0050 through 0053.

```
        LDA  #$FF
        STA  $A002          ; MAKE DISPLAY PORT OUTPUT
        STA  $A000          ; TURN OFF LEDS AT PORT B
        LDA  #$7B           ; ACTIVATE LEFTMOST CHARACTER
        STA  $AC00
        LDA  $41            ; GET DEVICE NUMBER
        ASL  A              ; DOUBLE NUMBER FOR INDEXING
        TAX
        LDA  $40            ; GET DATA
        STA  ($50,X)        ; SEND DATA TO PHYSICAL DEVICE
HERE    JMP  HERE
```

Program 7–5 is the mnemonic-entry version, along with the device table. We must double the device number, since each address occupies 2 bytes. Remember that 0 outputs light the LEDs attached to the user VIA (see Laboratory 3). Run Program 7–5 with (0040) = C3 and (0041) = 00. What happens if you change 0041 to 01 and run the program again?

PROGRAM 7–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A9		LDA #FF
0201	FF		
0202	8D		STA A002
0203	02		
0204	A0		
0205	8D		STA A000
0206	00		
0207	A0		
0208	A9		LDA #7B
0209	7B		
020A	8D		STA AC00
020B	00		
020C	AC		
020D	A5		LDA 41
020E	41		
020F	0A		ASL A
0210	AA		TAX
0211	A5		LDA 40
0212	40		
0213	81		STA (50,X)
0214	50		
0215	4C	HERE	JMP 0215
0216	15		
0217	02		
0050	00	DEVICE 0	(ADDRESS A000)
0051	A0		
0052	02	DEVICE 1	(ADDRESS AC02)
0053	AC		

PROBLEM 7–11

Change Program 7–5 to send (0040) to the device number in 0042 and (0041) to the other output device.

Example

(0040) = C0 (data for primary device)

(0041) = BF (data for sceondary device)
(0042) = 00 (number of primary deviee)
Result: (A000) = C0 (device 0 gets primary data)
(AC02) = BF (deviee 1 gets secondary data)

What happens if you change (0042) to 01? What happens if you reverse the order in the device table?

LONG ARRAYS

We discussed indireet indexed addressing in Laboratory 6. In this mode, the proeessor obtains a base address from two memory locations on page 0. Thus we ean change that base address freely, sinee it ean be in RAM even when the program is in ROM.

Indireet indexed addressing has another important use in 6502 programming. It lets us handle arrays that oeeupy more than 256 bytes. Ordinary indexed addressing eannot do this beeause the index registers are only 8 bits long.

The following program (see Program 7-6 for a mnemonic-entry version) elears a seetion of memory. The section starts at the address in 0040 and 0041; its length is given by the contents of 0042 and 0043 (in complemented form).

```
        LDA  #0              ; DATA = ZERO
        TAY                  ; STARTING INDEX = ZERO
CLEAR   STA  ($40),Y         ; CLEAR A BYTE
        INY                  ; MOVE TO NEXT BYTE
        BNE  COUNT
        INC  $41             ; AND TO NEXT PAGE IF NEEDED
COUNT   INC  $42             ; COUNT BYTES
        BNE  CLEAR
        INC  $43             ; WITH CARRY TO MSB
        BNE  CLEAR
        BRK
```

Ineremenling a 16-bit number is not easy. Sinee neither INY nor INC affects CARRY, it is diffieult to determine when a carry is neeessary. The only way to tell is to test the ZERO flag. When INY makes Y zero, we move to the next page by ineremenling the more signifieant byte of the base address (0041). Similarly, when INC $42 makes the less signifieant byte of the eount zero, we use INC $43 to earry to the more signifieant byte. Run Program 7–6 with the following sample data:

(0040) = 80 (LSBs of initial base address)
(0041) = 02 (MSBs of initial base address)
(0042) = C0 (LSBs of eomplemented eount)
(0043) = FE (MSBs of eomplemented eount)

Which memory loeations are eleared? We eount up here rather than down since inere-

PROGRAM 7–6

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A9		LDA #00
0201	00		
0202	A8		TAY
0203	91	CLEAR	STA (40),Y
0204	40		
0205	C8		INY
0206	D0		BNE 020A
0207	02		
0208	E6		INC 41
0209	41		
020A	E6	COUNT	INC 42
020B	42		
020C	D0		BNE 0203
020D	F5		
020E	E6		INC 43
020F	43		
0210	D0		BNE 0203
0211	F1		
0212	00		BRK

menting a 16-bit counter is much simpler than decrementing one. Why? Counting up requires us to calculate the two's complement of the number of locations to be cleared.

PROBLEM 7–12

What values must you place in 0040 through 0043 to make Program 7–5 clear 024C through 03EF inclusive?

PROBLEM 7–13

Extend Program 7–4 to clear one element of a long array. Assume that the base address is in 0040 and 0041 and that the 16-bit index is in 0042 and 0043.

Example

(0040) = 00 (LSBs of base)
(0041) = 02 (MSBs of base)
(0042) = 80 (LSBs of index)
(0043) = 01 (MSBs of index)
Result: (0380) = 00, since BASE + INDEX = 0200 + 0180 = 0380.

(*Hint:* Add the more significant bytes in the accumulator and use the sum as part of an indirect address.)

KEY POINT SUMMARY

1. Arrays can be formed by using an index to determine which element is being filled. Either a maximum length or a terminator can conclude the formation.
2. On the 6502, an index register can hold the element number. Then you can use indexed addressing to access the element. The simplest procedure is to start the index register at the number of elements and use the lowest address of the array minus 1 as the base.
3. To reach an element, you must know the array's base address and the element's index. Indexed addressing then allows the computer to access the element easily.
4. You can handle an array with multibyte elements by multiplying the index times the size of an element and adding the product to the base address. Multiplication by a small integer can be implemented as a series of additions. An arithmetic left shift is equivalent to multiplication by 2.
5. Indexed indirect addressing allows the processor to select one of a set of indirect addresses to use in transferring data. This mode can convert the logical device numbers to which a program refers into the physical I/O addresses for a particular computer.
6. Using logical I/O devices allows one to write programs that can run on many computers. To tailor them to a particular computer, one must construct an I/O device table that converts logical device numbers into physical I/O addresses. Changing the table lets the programmer vary I/O addresses without changing the program. This makes it easy to direct test results to a console, choose whether outputs should be displayed only or printed for permanent records, or switch between local and remote control.
7. The 6502's 8-bit index registers make arrays longer than 256 bytes awkward to handle. One way to process long arrays is to use indirect indexed addressing. The program must increment the more significant byte of the indirect address after processing each 256-byte section. This approach also requires a 16-bit counter in two memory locations. The simplest way to count is to count up from the two's complement of the length of the array. Incrementing the less significant byte of the counter sets the ZERO flag to 1 if a carry occurs. The ZERO flag can then be used to decide when to increment the more significant byte.

LABORATORY 8

DESIGNING AND DEBUGGING PROGRAMS

PURPOSE

To learn the fundamental approaches to program design and debugging.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, Chapter 6.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapters 13–15.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, 1982, pp. 133–135, 389–396.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 377–378 (address vectors, BRK instruction), 378–387 (program writing).

AIM 65 User's Guide, Dynatcm, Irvinc, CA, 1979, pp. 3–22 to 3–43.

WHAT YOU SHOULD LEARN

1. Stages of softwarc devclopmcnt.
2. Standard flowcharting symbols.
3. How to usc flowcharts to design programs.
4. Common debugging tools.
5. How to insert and use breakpoints.
6. How to use the single-step (STEP) mode.
7. How to trace instructions and rcgisters.
8. How to debug simple programs systematically.
9. Common errors in 6502 assembly language programs.

TERMS

Breakpoint a condition specified by the user under which program execution is to end temporarily, used as a debugging tool. We refer to specifying conditions as *setting breakpoints* and to deactivating conditions as *clearing breakpoints*.

Coding writing computer instructions.

Debugger a program that helps locate and correct program errors.

Debugging locating and correcting errors in a program.

Disassembler a program that converts machine language programs back into assembly language (the opposite of an *assembler*).

Dump a facility that displays the contents of an entire section of memory or group of registers on an output device.

Editor a program that lets a user enter, correct, revise, load, and save text material.

File a collection of related information that is stored and retrieved as a unit.

Flowchart a graphic representation of a computer program.

Modular programming a programming method that involves dividing the overall program into logically separate sections, or *modules*.

Murphy's Law the maxim "Whatever can go wrong, will." No one has ever doubted that it applies to computer programming.

No operation (no-op) an instruction that does nothing except increment the program counter.

Problem definition the determination of exactly what requirements a system must meet.

Program design the design of a computer program to meet the requirements of the problem definition.

Single step a facility that allows a program to be executed one step at a time.

Structured programming a programming method that involves constructing all programs from a few logical forms or structures, each of which has a single entry and a single exit.

Testing ensuring that a system meets the requirements of the problem definition.

Text file a file consisting of symbolic characters rather than numbers (a *data file*) or computer instructions (a *program file*).

Toggle a switch that turns something on if it was off and off if it was on.

Top-down design a design method in which the overall program is designed first and parts of it are later defined in more detail.

Trace a facility that displays the status of a program during execution.

Unsigned number a number in which all bits represent magnitude.

STAGES OF SOFTWARE DEVELOPMENT

So far, our programs have been short, and we have started with initial versions. The programming of real applications is, of course, more difficult. We cannot deal with all its aspects here, but we will discuss the design and debugging of small and medium-sized programs.

Software development consists of a series of stages:

1. *Problem definition*, in which you determine exactly what requirements the program must meet.
2. *Program design*, in which you provide a "blueprint" for the program.
3. *Coding*, in which you translate the design into computer instructions. Note that writing instructions is only one of many stages.
4. *Debugging*, in which you locate and correct errors in the program.
5. *Testing*, in which you ensure that the program meets its requirements.
6. *Documentation*, in which you describe the program so that it can be used, maintained, and extended.
7. *Maintenance*, in which you correct and upgrade the program to handle problems found in field use.
8. *Extension and redesign*, in which you upgrade the program to handle new requirements or new tasks.

A computer program thus goes through the same stages as a hardware project. Definition, design, debugging, testing, documentation, and maintenance typically require far more time and effort than does the writing of a program (or the construction of a hardware prototype). As with any project, you should allocate enough time for definition and design and proceed cautiously and systematically through debugging and testing.

We will coneentrate here on simple problems in which

1. The requirements have already been determined.
2. The program ean be designed with a flowchart.
3. Debugging and testing are virtually the same.
4. The later stages (e.g., documentation, maintenance) can be ignored. This is eertainly not the ease in praetiee; maintenanee is often the most time-consuming and costly stage of all.

FLOWCHARTING

Flowcharting is the traditional method for designing programs. Its advantages are its graphie form, set of standard symbols (see Figure 8–1), and wide recognition and acceptance.

We strongly recommend the following approach to flowcharting:

1. First draw a rough flowchart. Don't worry about how artistic or how complete it is.
2. Check the flowchart for obvious errors and improvements. Be sure that all branches lead somewhere, all variables are initialized or derived, and all decisions make sense.
3. Next, revise the flowchart. Again, do not worry about details or appearance. It is now time to write an initial program.
4. When you finish coding, debugging, and testing the program, draw a eurrent flowchart as part of the final documentation.

Don't let the flowehart beeome a burden. There is no systematic way to debug a flowchart or to eode from it. You might as well work on the actual program as keep revising the flowchart. If the program logic is complex, flowcharting alone is not an adequate design method. You must then consider such methods as modular programming, struetured programming, and top-down design (These are described in Chapter 13 of L. A. Leventhal, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979.).

FLOWCHARTING EXAMPLE 1: COUNTING ZEROS

Purpose

Count the number of zeros in 0340 through 0347 and place the result in 0040.

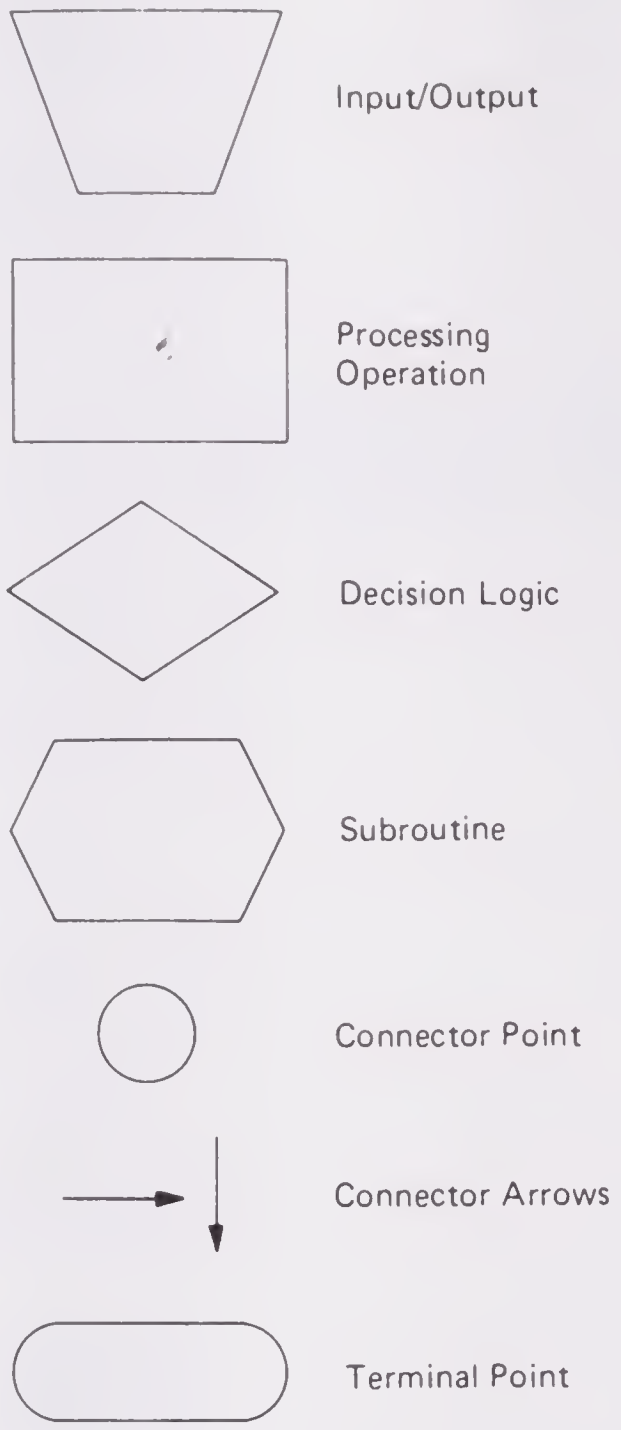


FIGURE 8–1. Standard flowchart symbols.

Sample Case

(0340) = 37
(0341) = 40
(0342) = 00
(0343) = 5E
(0344) = 00
(0345) = D1
(0346) = 39
(0347) = 00

Result: (0040) = 03, since 0342, 0344, and 0347 contain zero.

Our initial flowchart is Figure 8–2. A hand check shows that we forgot to initialize NZERO and that we reversed the branches after testing the memory location for zero.

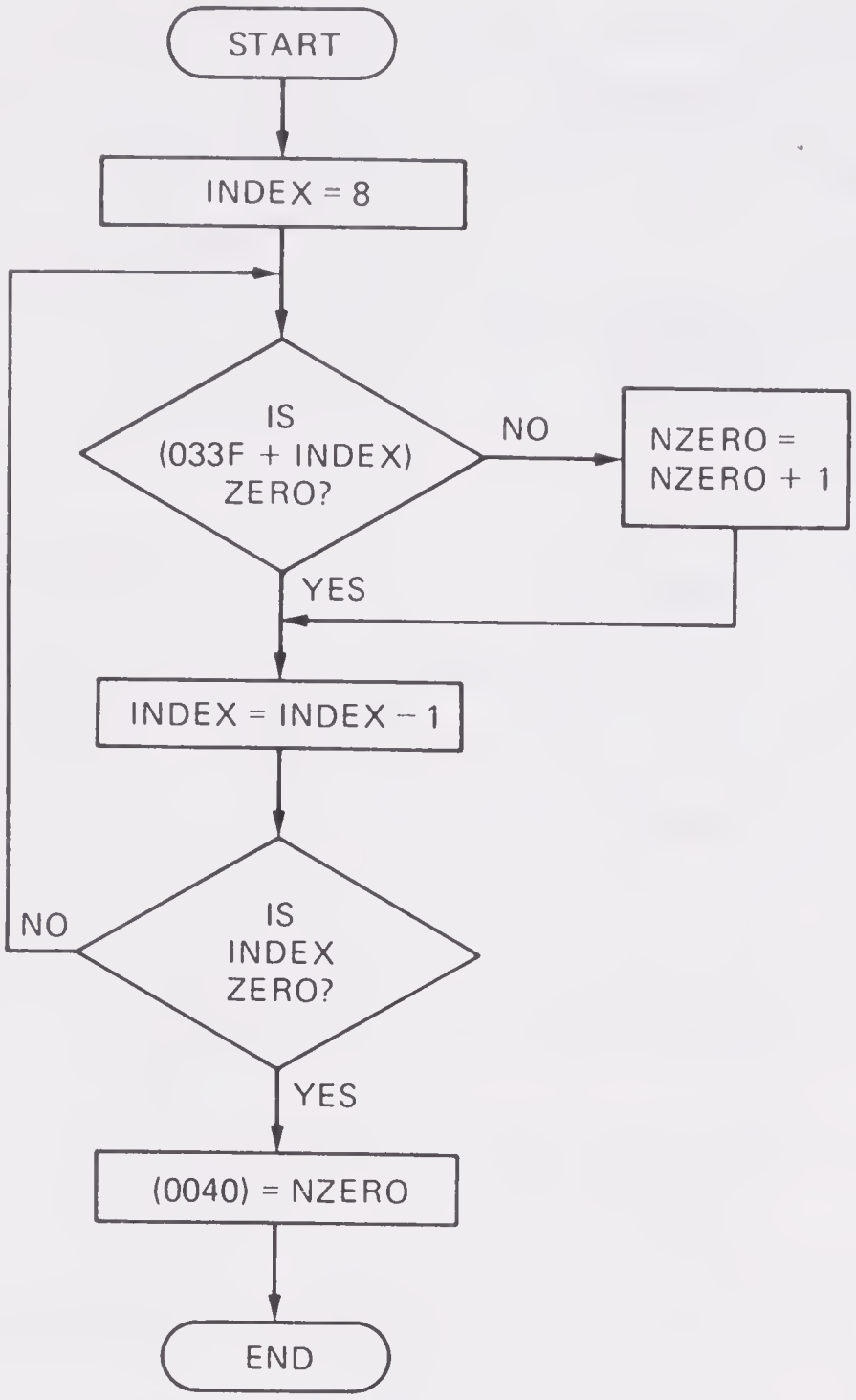


FIGURE 8–2. Initial flowchart for the zero-counting program.

Figure 8–3 shows the revised flowchart. We have not checked it in detail; we will describe how to debug the actual program later.

PROBLEM 8–1

Draw a flowchart for a program that counts the number of values in 0340 through 0347 that exceed (0041). Place the result in 0040. Assume that all values are unsigned.

Example

(0041) = 67 (threshold)
(0340) = 35 (first value)
(0341) = 4A
(0342) = A9
(0343) = 67
(0344) = B3

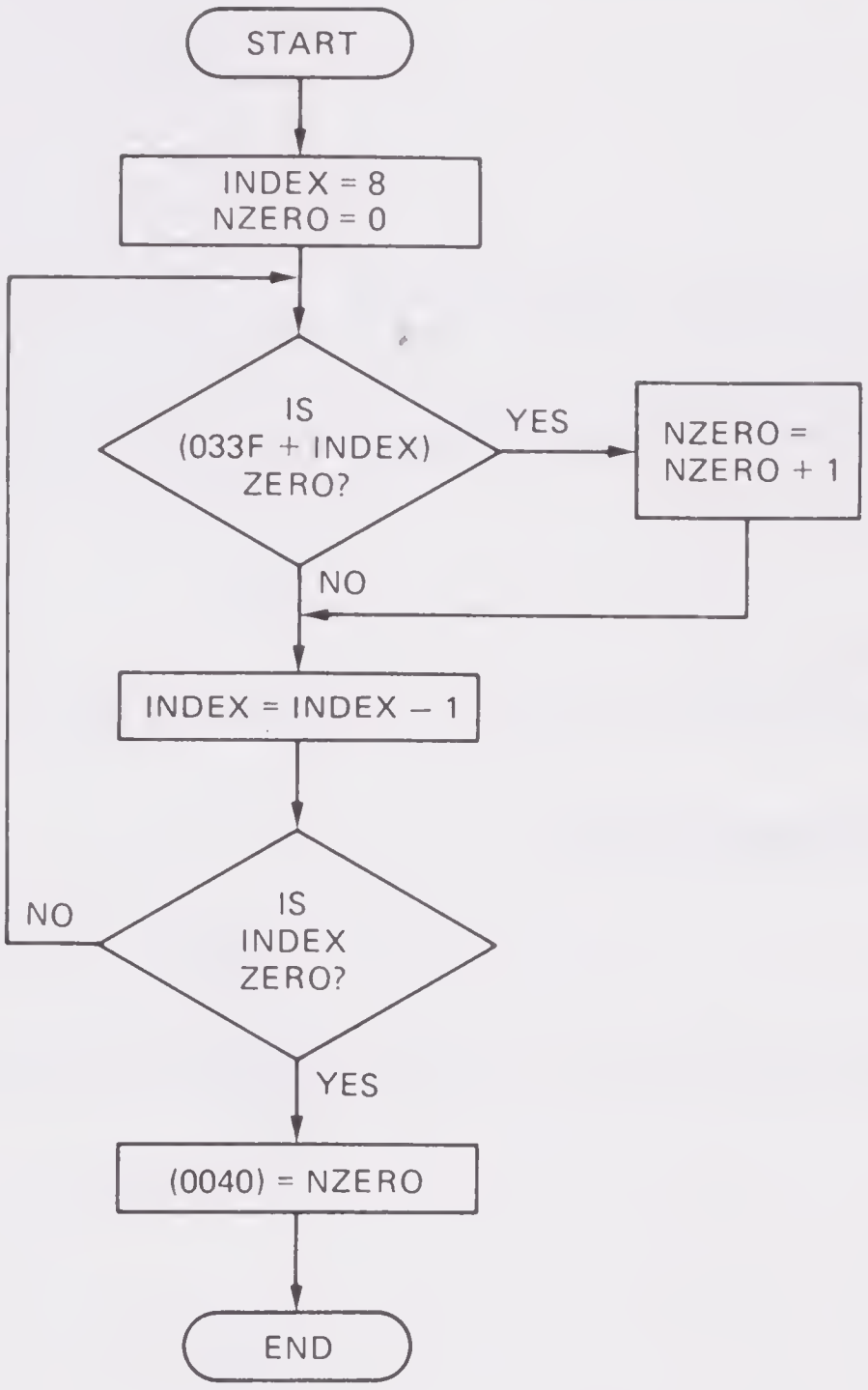


FIGURE 8–3. Revised flowchart for the zero-counting program.

(0345) = 69
(0346) = 14
(0347) = 33 (last value)

Result: (0040) = 03, since 0342, 0344, and 0345 contain values larger than (0041).

PROBLEM 8–2

Draw a flowchart for a program that searches 0340 through 0347 for a nonzero value. If it finds one, it stops the search, places the value in 0041, and clears the location from which it took the value. If all values are zero, the program clears 0041.

Example a:

(0340) = 07
(0341) = 04
(0342) = 12

(0343) = 00
(0344) = 13
(0345) = 06
(0346) = 00
(0347) = 00
Result: (0041) = 06, the first nonzero value encountered.
(0345) = 00, since the element removed from the array is then cleared.

Note that we are working backward through the array as usual on the 6502.

Example b:

(0340) through (0347) = 00
Result: (0041) = 00, since all elements are zero.

FLOWCHARTING EXAMPLE 2: MAXIMUM VALUE

Purpose

Find the largest unsigned binary number in 0340 through 0347 and store it in 0040.

Sample Case

(0340) = 37
(0341) = 40
(0342) = 88
(0343) = 5E
(0344) = 2B
(0345) = D1
(0346) = 39
(0347) = AE
Result: (0040) = D1

Our initial flowchart is Figure 8–4. A simple hand check shows that we forgot to initialize MAX and that we forgot to save the new maximum. In fact, as you will probably see if you implement the program, the revised flowchart of Figure 8–5 is still far from optimal.

PROBLEM 8–3

Draw a flowchart that finds the largest unsigned 16-bit binary number in 0340 through 0347 and stores it in 0040 and 0041. All numbers are stored in the standard 6502 format with the less significant byte first.

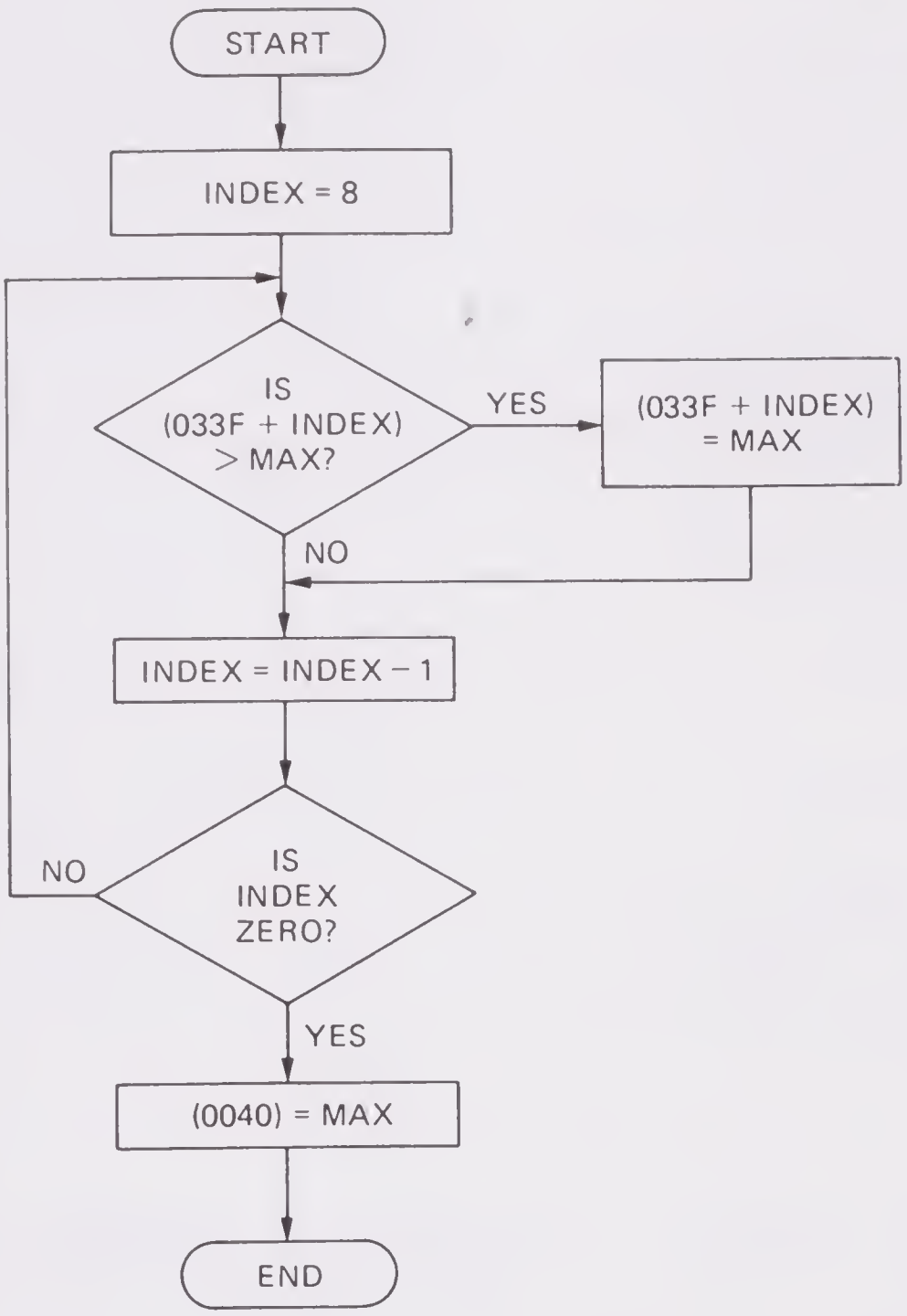


FIGURE 8–4. Initial flowchart for the maximum program.

Example

	(0340)	= 40	(LSBs of first number)
	(0341)	= 88	(MSBs of first number)
	(0342)	= 5E	(LSBs of second number)
	(0343)	= 2B	(MSBs of second number)
	(0344)	= D1	(LSBs of third number)
	(0345)	= 39	(MSBs of third number)
	(0346)	= AE	(LSBs of fourth number)
	(0347)	= A6	(MSBs of fourth number)
Result:	(0040)	= AE	(LSBs of maximum)
	(0041)	= A6	(MSBs of maximum)

since A6AE is larger than 39D1, 2B5E, or 8840.

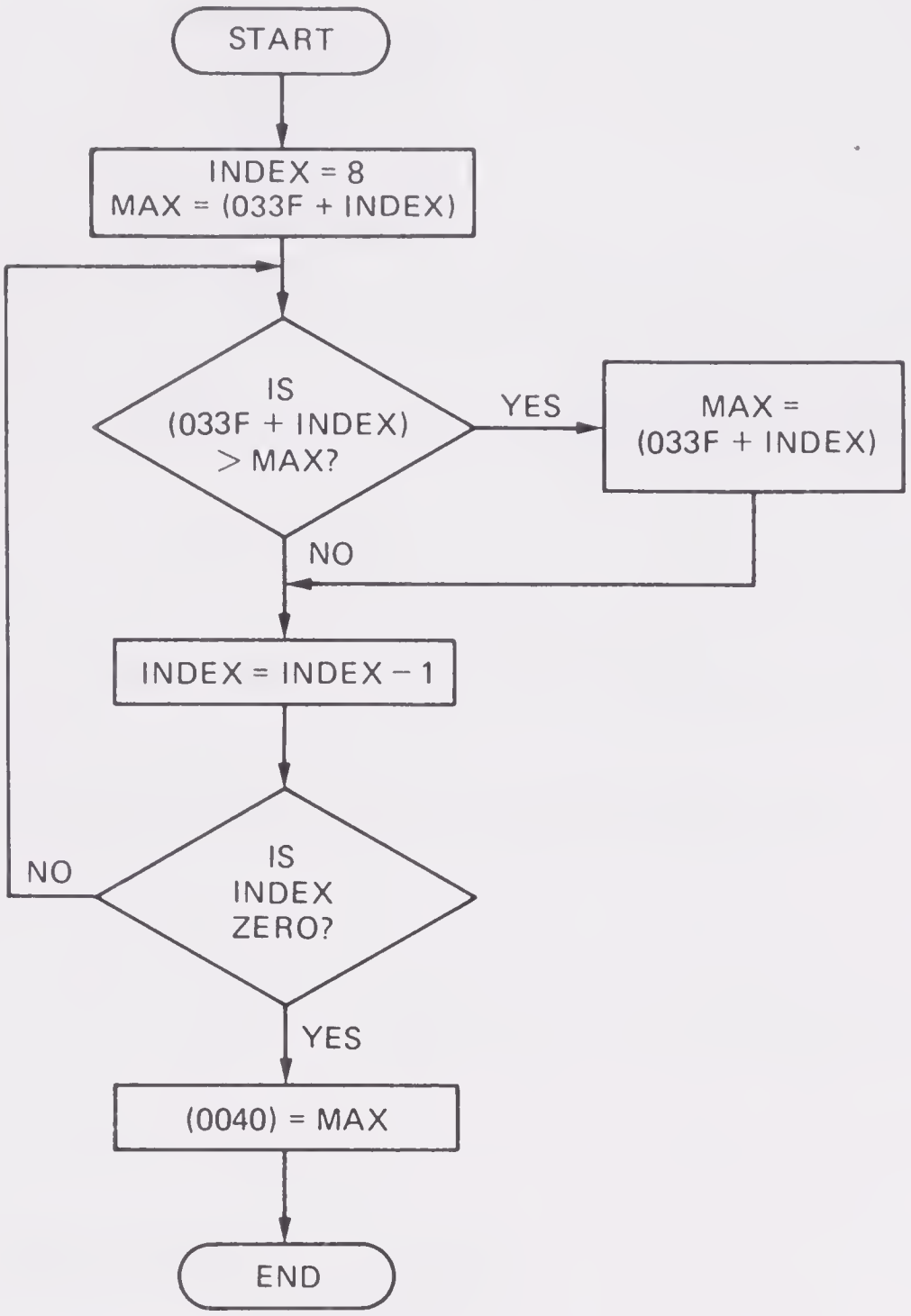


FIGURE 8–5. Revised flowchart for the maximum program.

FLOWCHARTING EXAMPLE 3: VARIABLE DELAY

Purpose

A switch attached to bit 7 of user VIA port A acts as a DELAY switch. When it is closed, the processor waits for the number of seconds (0 through 63) specified by the switches attached to bits 0 through 5 of port A.

Sample Case

The switches attached to bits 0 through 5 of port A produce a reading of 011110 (1 = open, 0 = closed). When switch 7 is closed, the processor waits for 30 s (011110 binary = 1E hex = 30 decimal). Figure 8–6 contains the initial flowchart. A check

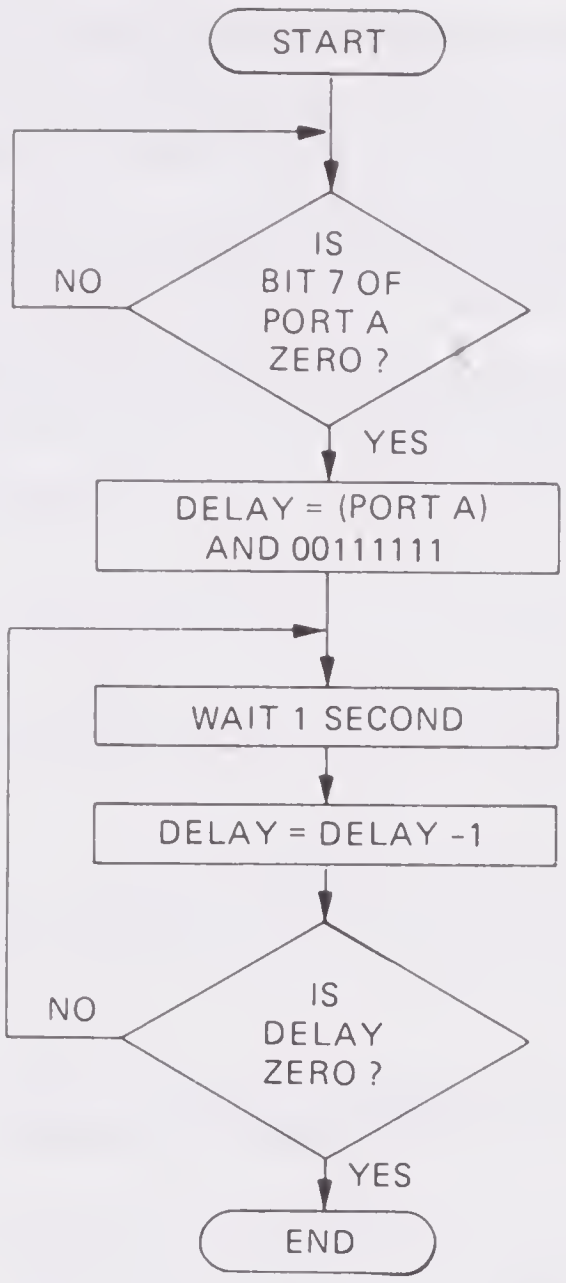


FIGURE 8–6. Initial flowchart for the variable delay program.

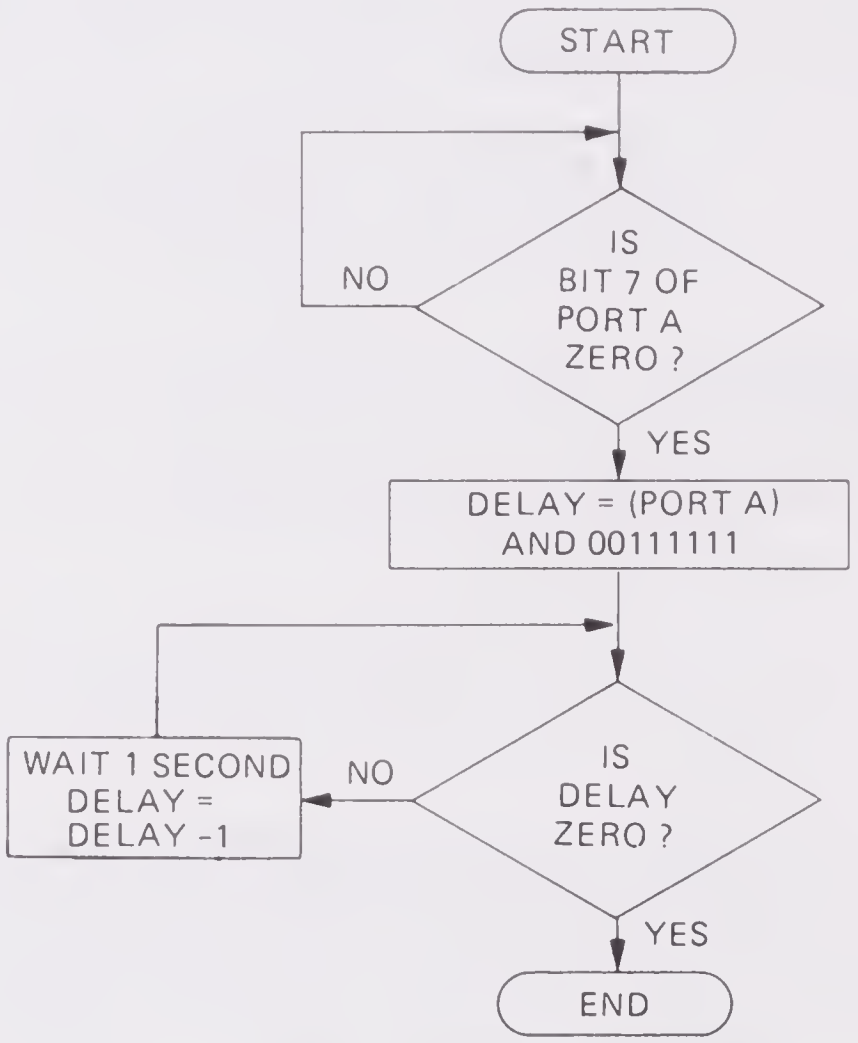


FIGURE 8–7. Revised flowchart for the variable delay program.

shows that it is wrong if the delay has zero length. (Why?) Figure 8–7 contains the revised flowchart.

PROBLEM 8–4

Draw a flowchart for an extended program that uses switch 6 to determine if the delay is in seconds (switch open) or milliseconds (switch closed).

DEBUGGING TOOLS

The AIM 65 monitor provides many useful debugging tools, including:

1. *Breakpoints*, which let the user stop the program and examine its current status. Breakpoints help you localize an error within a section of a program and pass through sections that you know are correct.

2. A *single-step* facility, which lets the user execute the program one step at a time. In the AIM, we call this the STEP mode.
3. A *trace*, which displays registers and memory locations while the program is executing. Traces provide a detailed accounting of program execution.
4. A *dump*, which displays an entire section of memory on an output device (usually the printer).
5. A *disassembler*, which converts machine language programs back into mnemonics. The disassembler can help you see if the program has been entered incorrectly, changed improperly, or affected by its own execution.

BREAKPOINTS

The following AIM 65 commands allow you to set, clear, enable, disable, and display up to four breakpoints:

B Set or clear a breakpoint. To set a breakpoint, press B and enter its number (0 through 3) and its hexadecimal address. To clear a breakpoint, do the same but enter 0 as the address.

4 Enable or disable breakpoints. The 4 command enables breakpoints if they were previously disabled and disables them if they were previously enabled. One generally uses B to set breakpoints and then 4 to enable them.

? Display breakpoints. To see where the breakpoints are, press the ? key. The AIM will display their addresses or 0000 for ones that are not set.

Clear all breakpoints. This clears all breakpoints at once. Since RESET does not clear breakpoints, you should press # after power-on and after finishing a debugging session.

The AIM's breakpointing facilities are handy. You can set or clear individual breakpoints and clear all breakpoints with a single keystroke. Furthermore, you need not worry about changing the underlying program or placing the breakpoint properly within an instruction. The AIM does not replace instructions; it simply checks whether the program has reached or passed through a breakpoint address. Thus you can resume the program with no difficulty, and you can set breakpoints anywhere, not just at addresses containing operation codes.

Microcomputer development systems usually have even more extensive breakpointing facilities than the AIM has. Useful features include the ability to set breakpoints on such conditions as:

1. Whenever a particular operation code is executed. The usual ones selected are those that perform input or output.
2. Whenever a particular memory address is accessed.

3. Whenever a particular sequence of instructions is executed.
4. Whenever a particular signal or combination of signals occurs. This is strictly a hardware breakpoint.

Still more advanced facilities combine simpler features and count occurrences. Setting breakpoints thus becomes similar to specifying triggering events on an oscilloscope.

STEP MODE

To put the AIM in the STEP mode, move the RUN/STEP switch (left of the display and the KB/TTY switch) to the STEP position (back). Now when you initialize the program counter and type G, you can tell the AIM how many instructions to execute as follows:

- 01 to 99 (two decimal digits) indicate that many instructions.
- RETURN indicates 1 instruction (same as 01).
- . or SPACE indicates continuous execution.

The AIM will continue until it executes the specified number of instructions, reaches a breakpoint (if breakpoints are enabled), or executes BRK. Note that breakpoints apply only in the STEP mode. When the AIM stops, it displays or prints the address of the next instruction (*not* the one it just executed), the operation code at that address, and the disassembled mnemonic instruction.

When the AIM returns to the monitor, you can resume execution without reinitializing the program counter. You must, however, tell the AIM how many instructions to execute each time.

TRACES

The AIM monitor has three tracing commands:

Z Toggle instruction trace. Pressing Z turns the instruction trace on if it is off and off if it is on. This trace makes the AIM disassemble and print the next instruction after each one it executes. The odd feature here is that the AIM does not disassemble the first instruction it executes; furthermore, it disassembles the instruction where it would resume the program.

V Toggle register trace. Pressing V turns the register trace on if it is off and off if it is on. This trace makes the AIM print the register contents in the usual order (PC, P, A, X, Y, and S, from left to right) after executing each instruction. However, it does not print an identification line.

H Trace program counter history. Pressing H displays the addresses of the last four instructions executed and the address of the next one to be executed. This applies only when the AIM returns control to the monitor after executing instructions in the STEP mode.

The instruction and register traces are handy if used carefully. The register trace, of course, shows you exactly what the computer is doing. The instruction trace tells you where the computer is in the program and keeps you from having to refer continually to a listing. It also alerts you to valid but incorrect entries (such as LDX instead of LDY or LDA 00 instead of LDA #00).

The problem is that traces are repetitive and slow. After all, instructions change at most one user register (A, X, or Y), the stack pointer rarely changes, and the program counter and status register are seldom of interest.

The solution is to be selective. Trace only short sections of a program (say, at most ten instructions). Be sure you know what to look for, and turn the traces off as soon as you finish with them.

We suggest the following approach to the AIM's trace commands:

1. Use breakpoints first to shorten the section to be traced. Decide what data you will use and what results you will examine.
2. Turn both traces on. The register trace generally provides the useful information, but the instruction trace lets you determine how far the computer has gone through the program.
3. Press R initially to list the starting register values and print an identification line. Remember, you can press R to print another identification line later.
4. The instruction trace is unnecessary if you are single-stepping through a program, since the AIM always dissassembles and prints one instruction.

DEBUGGING EXAMPLE: COUNTING ZEROS

From the flowchart in Figure 8–3, we write the following program for counting zeros. (Program 8–1 is the mnemonic-entry version.)

```
                LDX  8
                LDY  0
                LDA  $0340,X
CNTZ            BEQ  CHCNT
                INY
CHCNT           DEC  $0340,X
                BNE  CNTZ
                LDY  $40
                BRK
```

PROGRAM 8–1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A6		LDX 08
0201	08		
0202	A4		LDY 00
0203	00		
0204	BD		LDA 0340,X
0205	40		
0206	03		
0207	F0	CNTZ	BEQ 020C
0208	03		
0209	C8		INY
020A	DE	CHCNT	DEC 0340,X
020B	40		
020C	03		
020D	D0		BNE 0207
020E	F8		
020F	A4		LDY 40
0210	40		
0211	00		BRK

Enter this program but *don't run it*. (*Important rule*: Never just let a program run the first time. It may—and probably will—write over itself or cause other problems. Expect errors and plan for them.)

We will start by checking the initialization instructions (LDX 08, LDY 00). Since this section is short, we will trace it immediately. If it were longer, we would probably either step through it or divide it further to minimize the amount of tracing.

To trace the instructions, we must:

1. Move the RUN/STEP switch back to the STEP position.
2. Press Z and V to activate the instruction and register traces. Note that these are toggles, and the AIM always tells you whether the traces are on afterward.
3. Initialize the program counter to 0200.
4. Press R to print the initial register values and an identification line. Be sure the printer is on!
5. Press G, 0, 2, RETURN to make the AIM execute two instructions. Note that you must enter 02 after G, not just 2.

The result after the first instruction should be

(X) = 08 (initial value of index)

What do you find in X? We get

(X) = 82

Your result may be different since the program is way off base.

Obviously, the instruction is wrong. The disassembly tells us nothing, since the AIM does not disassemble the first instruction. So why is LDX 08 wrong? The simplest alternative would be LDX #08. This is, in fact, what we want, since we intend to load X with the number 8, not with the contents of address 0008. This is a common mistake—confusing a data value with an address, particularly an 8-bit address on page 0.

So we must replace LDX 08 with LDX #08. We can either do this in the mnemonic-entry mode or we can change 0200 from A6 to A2 manually. Of course, you must look up the codes on your programming card or in Table A1–1.

The second instruction should produce the result

(Y) = 00 (number of zeros found)

Instead, the register trace shows

(Y) = 10

Here the AIM at least disassembles the instruction. This tells us that we entered it correctly but obviously not whether we selected it correctly. We immediately suspect the same error as in the first instruction; we want LDY immediate, not LDY zero-page. So we must replace LDY 00 with LDY #00 at address 0202.

Making this change and repeating the run produces the correct results:

(X) = 08 (initial value of index)

(Y) = 00 (number of zeros found)

Note the key points of this debugging exercise:

1. A single-step mode (particularly if it allows you to trace the registers) can show you precisely what is wrong with a program.
2. Although tracing is helpful, it produces a lot of repetitive and irrelevant information. Note how little of the trace we actually used.
3. To debug a program effectively, you must concentrate on short sections and decide what you are looking for before proceeding. Otherwise, you will end up with useless information and poorly documented changes.
4. Most programmers make the same mistakes consistently. Knowing your own favorites will help you correct programs. Obviously, it is easier to list one's favorite errors than to change one's habits.

USING BREAKPOINTS

Since we now know that the initialization instructions are correct, we can proceed to debug the loop. Rather than count instructions this time, we simply place breakpoints at the beginning and end. First, clear all breakpoints by pressing # (remember, # is *not* a toggle); this removes stray or leftover breakpoints that could cause problems. To set breakpoint 0 in address 0204, press

1. B (set or clear breakpoints).
2. 0 (breakpoint number).
3. 2, 0, 4, RETURN (location of breakpoint 0).

To set breakpoint 1 in address 020D, press

1. B.
2. 1 (breakpoint number).
3. 2, 0, D, RETURN (location of breakpoint 1).

You should also press ? to display the breakpoints and verify the entries. Finally, press 4 to enable breakpoints.

At this point, we need some data. The choices are to make 0347 zero or nonzero. Let us first try

(0347) = 00

Since this section has branches, we will single-step through it rather than using the traces. First, reach the section by starting the program at 0200 and letting the computer run through the initialization. To do this, simply press G, SPACE. Breakpoint 0 lets us repeat the initialization quickly if necessary.

Now press R to get an initial register display and G, RETURN to execute one instruction. LDA 0340,X should produce

(A) = 00 (the element loaded from 0347)

Instead we find

(A) = E0

An obvious pitfall in indexed addressing is having the base address off by 1. Let us examine 0348 to see if that might be the problem here. Sure enough, we find

(0348) = E0

So the error is that we are going past the end of the array. Note that a register trace would not display the incorrect effective address, since it is never placed in a register. To see the effective address, we would need test equipment to monitor the processor's input and output signals. Otherwise, all we can see is the contents of the registers.

Our correction here is to replace LDA 0340,X with LDA 033F,X in 0204. We then go back to 0204 and press G, RETURN to execute the revised instruction. The result is correct now, but the next instruction is completely wrong. In the first place, the program branches when it shouldn't (the accumulator does contain 0). In the second place, the branch sends the processor to address 020C, which does not even contain an instruction. The instruction disassembly makes this error obvious; since 03 is not a valid operation code, the computer prints and displays ???.

We can correct the first problem by replacing BEQ with BNE and the second one by changing the destination from 020C to 020A. Thus, in the mnemonic-entry mode, we must put BNE 020A in 0207. Here we have managed to make two common errors: inverting decision logic and directing a forward branch to the wrong place.

Let us now try an entire iteration. We start back at 0200 and press G, SPACE once to reach 0204, and again to reach 020D. Now we press R to display the registers. The results at the breakpoint are

(A) = 00

(X) = 08

(Y) = 01

Everything is fine except that X has not been decremented. A quick check shows that DEC 0340,X is incorrect since it decrements a memory location, not register X. What we want is simply DEX. But this correction leaves two extra bytes of memory that previously held DEC's base address. What should we put there? The answer is two NOPs. They do not affect program execution, and we can always delete them later. Thus, in the mnemonic-entry mode, we enter DEX, NOP, NOP, starting at 020A. Now the program works properly through the breakpoint.

Let us try two iterations with

(0346) = 00

(0347) = 01

Execute the program starting at 0200. It will reach breakpoint 1 with the correct values:

(A) = 01 (the element loaded from 0347)

(X) = 07 (the initial index reduced by 1)

(Y) = 00 (no zero element has been found)

To run the second iteration, simply press G and SPACE. When the computer reaches breakpoint 1 again, the results should be

(A) = 00 (the element loaded from 0346)

(X) = 06 (the initial index reduced by 2)

(Y) = 01 (one zero element has been found)

The actual results are

(A) = 01

(X) = 06

(Y) = 00

The program is not loading the second value from memory. A hand check shows that label CNTZ is misplaced. It should be one instruction earlier (i.e., attached to LDA $033F,X) to make the program load a new element before checking the ZERO flag. The corrected instruction is BNE 0204.

Now try the program on some test data, such as

1. (0340) through (0347) = 00.
2. Same except (0340) = 01.
3. Same except (0347) = 01.

The final version is

```
            LDX  #8          ; NUMBER OF ELEMENTS = 8
            LDY  #0          ; NUMBER OF ZEROS FOUND = 0
CNTZ        LDA  $033F,X     ; IS NEXT ELEMENT 0?
            BNE  CHCNT
            INY              ; YES, INCREMENT NUMBER OF ZEROS
CHCNT       DEX
            BNE  CHCNT
            LDY  $40         ; SAVE NUMBER OF ZEROS FOUND
            BRK
```

Program 8–2 is a mnemonic-entry version with the corrections.

Note the following key points from this exercise:

1. A breakpoint can tell you whether an entire section of a program is correct. However, if it is not correct, the breakpoint alone does not tell you where the error is.

2. Breakpoints help you pass quickly through a section that you know is correct.
3. Different debugging tools are complementary rather than competitive. Breakpoints help restrict a search to a small section of a program, while a single-step mode and a trace provide the detailed information that usually helps you spot the error.
4. Debugging tools cannot correct programs by themselves. All they do is provide information; you must figure out what it means. Debugging is not a simple or routine task; it requires organization, caution, patience, common sense, experience, and insight.

PROBLEM 8–5

What errors still remain in Program 8–2? Correct them and run the final version for the three test cases we just described.

PROGRAM 8–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #08
0201	08		
0202	A0		LDY #00
0203	00		
0204	BD	CNTZ	LDA 033F,X
0205	3F		
0206	03		
0207	D0		BNE 020A
0208	01		
0109	C8		INY
020A	CA	CHCNT	DEX
020B	D0		BNE 0204
020C	F7		
020D	A4		LDY 40
020E	40		
020F	00		BRK

PROBLEM 8–6

Revise Program 8–2 to count the number of positive elements in 0340 through 0347. An element is positive if its most significant bit (bit 7) is zero, but its value is not zero.

Example

(0340) = 01
(0341) = 80

(0342) = 7F
(0343) = FF
(0344) = 00
(0345) = 00
(0346) = 00
(0347) = 00

Result: (0040) = 02, since 0340 and 0342 contain positive elements.

PROBLEM 8–7

Code, debug, and test Flowcharting Example 2, the maximum-value program.

PROBLEM 8–8

Code, debug, and test Flowcharting Example 3, the variable delay. The following routine uses location 0040 and the index registers to produce a 1-s wait:

```
                LDA  #5          ; WAIT 1 SECOND
                STA  $40
DLY1            LDY  #$C8
DLY2            LDX  #$C8
DLY3            DEX
                BNE  DLY3
                DEY
                BNE  DLY2
                DEC  $40
                BNE  DLY1
```

You should verify the delay constants.

DUMP AND DISASSEMBLER

We have not described the following useful debugging tools:

D The D command lets you dump a section of memory on the printer. You must enter the starting address (after the FROM = prompt), the ending address (after the TO = prompt), and the output device (type P after the OUT = prompt). The D command is more convenient than the M command for large memory areas.

K The K command disassembles a series of instructions. You must enter the starting address (after the * = prompt) and the number of instructions as two decimal digits (after the / prompt). As with G in the STEP mode, you can also enter

RETURN (1 instruction).
. or SPACE (continuous disassembly).

To suspend disassembly, press SPACE. To resume, press any key. Note that the AIM will print question marks if it finds invalid operation codes.

Disassembly is a convenient way to obtain a clean copy of a program in mnemonic form. This is useful if you have made a lot of changes while debugging. Dissassembly can also show whether you made a change properly or whether the program has somehow managed to change itself.

WHY THE EDITOR/ASSEMBLER IS USEFUL

As we have seen, deleting bytes from machine language programs is simple. All you must do is fill the unused locations with NOPs. On the other hand, inserting bytes is difficult because you must move all subsequent instructions to make room. If you accidently omit a line near the beginning of a program, you might as well reenter it. Of course, Murphy's Law guarantees that omissions always occur near the beginning of a program, rather than near the end.

Obviously, we cannot handle a long program this way, since reloading thousands of locations is impractical. A common alternative is to prepare the assembly language program using an *editor* that lets us make insertions, deletions, replacements, and other changes. Finally, the editor lets us save the completed program as a *text file* (in memory, on cassette, or on disk), which can then be assembled. If we find errors in the assembly or execution of the program, we can correct them by returning to the editor, revising the text file, and reassembling the program. The AIM 65 has an optional editor/assembler that fits in a ROM socket.

COMMON PROGRAMMING ERRORS

Watch for the following common errors in AIM 65 mnemonic-entry programs:

1. Confusing data and addresses. Remember the difference between immediate and direct addressing; immediate addressing means that the instruction contains the data, whereas direct addressing means that it contains the data's address. Remember also that the value in a memory location is not related to the effective address, base address, or index.
2. Using the CARRY incorrectly. Remember that comparisons and subtractions set the CARRY if they do not require a borrow. The CARRY flag is an inverted borrow, not a true borrow as on most other microprocessors. Note also that addition and subtraction instructions (ADC and SBC) always include the CARRY. You must explicitly clear it before addition or set it before subtraction to keep it from affecting the result.
3. Inverting the logic of conditional branch instructions (e.g., using BCC instead of BCS or BNE instead of BEQ). Be particularly careful after a comparison.

4. Jumping to the wrong address. This often results in repeating or omitting initialization instructions or instructions that update indexes or indirect addresses.

When debugging programs on the AIM, the best way to make corrections is to return to the mnemonic-entry mode. If, however, you choose to simply change memory directly, watch for the following common errors:

1. Inverting the order of the bytes in 2-byte addresses. Remember that the 6502 expects the less significant byte first.
2. Copying operation codes incorrectly. You should verify programs before executing them.
3. Calculating relative offsets incorrectly. Either use a hexadecimal calculator or double-check your results.
4. Omitting addresses, offsets, or data. Watch for instructions such as JMP, which requires a full 16-bit address in the next 2 bytes of memory. Remember that absolute addressing modes always require 2 bytes of memory and zero-page modes (including preindexing and postindexing) always require 1 byte.

Other common errors in 6502 programs are:

1. Failing to initialize counters, indexes, and indirect addresses.
2. Branching incorrectly when operands are equal. Note that comparing equal values *sets* the CARRY flag.
3. Overlooking trivial cases such as zero or one element in an array or table or no inputs.
4. Using the flags incorrectly. Typical examples are trying to use a flag that an instruction does not affect and overlooking changes caused by intermediate instructions. The only way to be sure of how an instruction affects the flags is to look it up in Table A1–1. Among 6502 instructions that often cause problems are loads (they affect the ZERO and NEGATIVE flags), stores (they affect no flags at all), increment and decrement (they do not affect the CARRY flag), and BIT (it sets the OVERFLOW and NEGATIVE flags from bits 6 and 7 of the addressed memory location, regardless of the accumulator's contents).
5. Using a register for several purposes without saving and restoring the different values. Use page 0 of memory (addresses 0000 through 00FF hex) as extra scratchpad registers, since it can be accessed quickly with zero-page addressing modes.
6. Using the wrong base address in indexing. As noted in Laboratories 6 and 7, you often must subtract 1 from the actual starting address to process an array efficiently. It is easy to be off by one location and end up either misaligned or beyond the bounds of the array.

You will undoubtedly make errors not mentioned here, but these lists should help or at least suggest some possibilities. Unfortunately, debugging computer programs is more of an art than a science.

KEY POINT SUMMARY

1. The writing of software, like the building of hardware, consists of many stages. Writing the actual computer instructions (or *coding*) is one of the easiest stages.
2. Flowcharting is a simple graphic technique for designing and documenting programs. A set of standard flowchart symbols is in widespread use.
3. A flowchart is a good starting place for writing a program, but it should not become a burden in and of itself.
4. Breakpoints are stopping places in programs that you can use to determine whether sections are correct and to pass through sections that you know are correct. The AIM's #, B, ?, and 4 commands let you set, clear, display, enable, and disable up to four breakpoints.
5. The STEP mode makes the AIM stop after it executes one or a few instructions. The register and instruction traces provide detailed information about the program during execution. You can use these tools to pinpoint an error after using breakpoints to narrow the search to a short section. You should limit your traces, however, since tracing is slow and produces a lot of useless or redundant information.
6. Common programming errors include confusing data and addresses, inverting logic or reversing the direction of operations, failing to initialize variables or save results, omitting operands, forgetting how instructions affect flags, ignoring trivial cases, and branching incorrectly.
7. Program debugging is a difficult, time-consuming job. You must know which tools to use and when to use them, what to look for, and which errors are likely. Debugging requires caution, organization, and imagination; in fact, it requires the same skills needed to work a crossword puzzle, play chess or bridge, or solve a maze, riddle, or murder mystery.

ARITHMETIC

PURPOSE

To learn to perform arithmetic calculations using the 6502 microprocessor.

REFERENCE MATERIALS

M. ANDREWS, "Mathematical Microprocessor Software: A Square Root Comparison," *IEEE Micro*, May 1982, pp. 63–79.

K. HWANG, *Computer Arithmetic*, Wiley, New York, 1979.

U. W. KULISCH and W. L. MIRANKAR, *Computer Arithmetic in Theory and Practice*, Academic Press, Orlando, FL, 1980.

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 198–210.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 4–13 through 4–15, Chapter 8.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Berkeley, CA, 1982, pp. 230–305, 382–388.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 12–14 (BCD code), 17–24 (binary arithmetic), 24–26 (BCD arithmetic), 174–180 (arithmetic/logic unit), 180–187 (comparison of microprocessors), 322–324 (status control instructions), 324–329 (arithmetic instructions), 375–377 (multibyte arithmetic operations).

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, Section 5.8 (assembler directives).

"Binary Floating-Point Arithmetic, Standard for (IEEE Std 754–1985)," IEEE, Piscataway, NJ, 1985.

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, pp. 2–4 to 2–18, 6–11 to 6–13.

WHAT YOU SHOULD LEARN

1. The standard BCD representation.
2. How to use the 6502's decimal mode.
3. How to add and subtract multiple-precision numbers.
4. How to use lookup tables to do arithmetic.

TERMS

BCD (binary-coded-decimal) a representation of decimal numbers in which each digit is coded separately in binary.

Carry a bit that is 1 if an addition overflows into the succeeding digit position.

Half-carry the carry from the less significant 4 bits in an 8-bit operation.

Interpolation estimating a function at points between those where its values are known.

Linearization approximating a function by a straight line between two points where its values are known.

Pseudo-operation an assembly language operation code that directs the assembler to do something but does not generate a machine language instruction.

6502 INSTRUCTIONS

CLD clear DECIMAL MODE (D) flag (make it 0). The processor will perform subsequent ADC and SBC instructions in binary.

SBC subtract with carry; subtract a memory location and the complemented CARRY flag from the accumulator. The result is (A) = (A) − (M) − (1 − CARRY), where M is a memory address.

SEC set CARRY flag to 1.

SED set DECIMAL MODE (D) flag to 1. The processor will perform subsequent ADC and SBC instructions in BCD.

6502 ASSEMBLER PSEUDO-OPERATIONS

.BYTE form byte-length data; place 8-bit data items (separated by commas) in the next available memory locations. .BYTE loads memory with fixed data (such as tables, messages, and numerical constants) needed for program execution.

.WORD form double-byte-length data; place 16-bit data items (separated by commas) in the next available memory locations. .WORD loads memory with 16-bit fixed data or addresses stored with the bytes in the standard 6502 order.

***=** set origin; place the machine language generated from the subsequent statements in memory addresses starting with the one specified. This pseudo-operation lets the programmer determine where programs and data are placed in memory.

APPLICATIONS OF ARITHMETIC

The processing of data almost always involves arithmetic. Typical operations are averaging, scaling, linearizing inputs, calculating numerical integrals and derivatives, determining frequency responses, performing statistical analysis, and preparing plots. Simple applications require only binary or decimal addition and subtraction. Decimal arithmetic is necessary in calculators, business equipment, terminals, instruments, appliances, and games.

This laboratory starts with 8-bit binary arithmetic programs from Laboratory 6. It then covers decimal arithmetic, multibyte arithmetic, and the use of lookup tables.

8-BIT BINARY SUM

The following program adds 8-bit unsigned binary numbers from 0340 and 0341 and stores the sum in 0040.

```
LDA   $0340          ; GET FIRST NUMBER
CLC                  ; CLEAR CARRY
ADC   $0341          ; ADD SECOND NUMBER
STA   $40            ; SAVE SUM
BRK
```

We must clear CARRY before adding, since ADC always includes it. The mnemonic-entry version of this program is Program 9–1; we have inserted two NOPs in anticipation of a BCD version.

PROGRAM 9–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	EA	NOP
0201	AD	LDA 0340
0202	40	
0203	03	
0204	18	CLC
0205	6D	ADC 0341
0206	41	
0207	03	
0208	85	STA 40
0209	40	
020A	EA	NOP
020B	00	BRK

Enter Program 9–1 and run it for the following cases:

1. (0340) = 32
   (0341) = 25
   Result: (0040) = 57
2. (0340) = 38
   (0341) = 25
   Result: (0040) = 5D

PROBLEM 9–1

Make Program 9–1 save the carry in 0041. Try the following cases:

1. (0340) = 38
   (0341) = 25
   Result: (0040) = 5D
   (0041) = 00
2. (0340) = 98
   (0341) = 89
   Result: (0040) = 21
   (0041) = 01

PROBLEM 9–2

Make Program 9–1 subtract instead of adding. How do you keep CARRY from affecting the result? Try the following cases:

1. (0340) = 32
   (0341) = 25
   Result: (0040) = 0D
2. (0340) = 32
   (0341) = 58
   Result: (0040) = DA

BINARY-CODED-DECIMAL (BCD) REPRESENTATION

A BCD code is the simplest way to represent decimal numbers in a computer, since it does not require multiplications or divisions by 10. In the standard BCD code (see Table 9–1), 0 through 9 are the same as in binary. However, numbers above 9 are different (see Table 9–2 for some examples). Note the following:

1. Each decimal digit is coded separately in BCD. This is not true in binary, since 10 is not an integral power of 2.
2. The BCD representation requires more memory than the binary representation. For example, 8 bits can represent a binary number as large as 255 but only 99 in BCD. The number 999 requires three BCD digits (12 bits) but only 10 bits in binary (since $2^{10} = 1{,}024$).
3. Some binary numbers are invalid in BCD. In standard BCD, no digit can exceed 9.

TABLE 9–1 STANDARD BCD REPRESENTATION

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

TABLE 9–2 STANDARD BCD REPRESENTATIONS OF SOME DECIMAL NUMBERS

Decimal Number	BCD Representation	Binary Representation
10	0001 0000	00001010
11	0001 0001	00001011
12	0001 0010	00001100
13	0001 0011	00001101
16	0001 0110	00010000
25	0010 0101	00011001
50	0101 0000	00110010
66	0110 0110	01000010
83	1000 0011	01010011

One problem with BCD numbers is the difficulty of processing them in binary arithmetic units. The reason is that BCD 10 (00010000) is not one larger than BCD 9 (00001001)—it is, in fact, seven larger. (Try subtracting!) Thus to obtain a BCD sum using a binary adder, you must add an extra 6 whenever the sum of two digits exceeds 9.

Example 1

$$\begin{array}{rrcl} & 33\ (\text{BCD}) & = & 00110011 \\ + & 25\ (\text{BCD}) & = & \underline{00100101} \\ & & & 01011000 = 58\ (\text{BCD}) \end{array}$$

There is no problem here, since neither digit is more than 9.

Example 2

$$\begin{array}{rrcl} & 38\ (\text{BCD}) & = & 00111000 \\ + & 25\ (\text{BCD}) & = & \underline{00100101} \\ & & & 01011101 = 5\text{D} \end{array}$$

Here an extra 6 is necessary since 8 + 5 produces a decimal carry.

$$\begin{array}{rr} & 5\text{D} \\ + & \underline{06} \\ & 63 \end{array}$$

Example 3

$$\begin{array}{rrcl} & 98\ (\text{BCD}) & = & 10011000 \\ + & 25\ (\text{BCD}) & = & \underline{00100101} \\ & & & 10111101 = \text{BD} \end{array}$$

Here an extra 6 is neeessary in both positions.

$$\begin{array}{rr} & \text{BD} \\ + & 66 \\ \hline & 123 \end{array}$$

Obviously, deeiding when to add 6 is not simple, sinee you must eheek eaeh digit. Sinee deeimal arithmetie is essential in eommon applications, most proeessors have speeial instruetions for it. The 6502 has a deeimal mode in whieh it adds and subtraets in BCD. The proeessor enters this mode by exeeuting SED (SET DECIMAL MODE), thus setting the D (DECIMAL MODE) flag. It leaves the mode by executing CLD (CLEAR DECIMAL MODE). When the D flag is set, ADC and SBC produee deeimal results; ineraments and deeraments, however, still produee binary results. You ean determine the proeessor's mode by examining the D flag (bit 3 of the status register).

8-BIT DECIMAL SUM

The following program adds BCD numbers from 0340 and 0341 and stores the sum in 0040.

```
SED                    ; ENTER DECIMAL MODE
LDA  $0340             ; GET FIRST NUMBER
CLC                    ; CLEAR CARRY
ADC  $0341             ; ADD SECOND NUMBER
STA  $40               ; SAVE SUM
CLD                    ; LEAVE DECIMAL MODE
BRK
```

The only changes from Program 9–1 are SED and CLD instead of NOPs.

Run the BCD version of Program 9–1 with the following data:

1. (0340) = 32
   (0341) = 25
   Result: (0040) = 57
2. (0340) = 38
   (0341) = 25
   Result: (0040) = 63

The second sum differs from the binary result.

PROBLEM 9–3

What is in the aeeumulator, CARRY, and half-earry (i.e., the earry from bit 3) after ADC $0341 for the following examples in the binary mode?

a. (0340) = 38
   (0341) = 25
b. (0340) = 98
   (0341) = 25
c. (0340) = 98
   (0341) = 89
d. (0340) = 90
   (0341) = 91

Why is the half-carry necessary in the decimal mode? (*Hint:* Examine the results of examples c and d. You can observe the accumulator and the CARRY—(bit 0 of register P)—but you must calculate the half-carry since it is not saved in a flag.)

PROBLEM 9–4

Make the decimal program subtract instead of add. Try the examples in Problem 9–3. What is CARRY at the end of each example? What does CARRY mean at the end of this program?

Adding and subtracting BCD numbers in the decimal mode is straightforward. However, the effects on the flags and other instructions (compares, increments, decrements, etc.) are confusing. Remember the following:

1. Increments and decrements produce binary results, regardless of the mode.
2. The NEGATIVE flag always reflects the binary result, not the BCD result. Thus it is meaningless after ADC or SBC instructions executed in the decimal mode. For example, subtracting 60 hex from 30 hex in the decimal mode sets the NEGATIVE flag (since the binary result is 11010000 or D0 hex). It does not matter that the decimal result (01110000 binary or 70 hex) has a most significant bit of 0.
3. Comparisons executed in the decimal mode set the ZERO and CARRY flags properly (the mode does not matter—why?). The NEGATIVE flag, as usual, reflects only the binary result.
4. RESET does not initialize the D flag. Thus the programmer must initialize it (usually with CLD) before the program executes any ADC or SBC instructions.
5. As you might expect, the 6502 does not warn you if your program uses the decimal mode improperly. If, for example, your program adds or subtracts nondecimal numbers (such as EC hex) in the decimal mode, the processor will simply execute the instructions and produce meaningless results.
6. Be careful not to set the D flag inadvertently or forget to clear it. This can happen when you are revising or debugging programs or when you load the status register by hand. If a program with ADC or SBC instructions produces odd results for no apparent reason, check the D flag.

DECIMAL SUMMATION

The following version of Program 6–1 adds an array of unsigned binary numbers starting at 0340; it places the sum in 0040. The length of the array is in 0041.

```
            NOP
            LDX  $41          ; INDEX = ARRAY LENGTH
            LDA  #0           ; SUM = ZERO INITIALLY
ADDELM      CLC               ; CLEAR CARRY
            ADC  $033F,X      ; ADD ELEMENT TO SUM
            DEX
            BNE  ADDELM
            STA  $40          ; SAVE SUM
            NOP
            BRK
```

Program 9–2 is the mnemonic-entry version.

PROGRAM 9–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	EA		NOP
0201	A6		LDX 41
0202	41		
0203	A9		LDA #00
0204	00		
0205	18	ADDELM	CLC
0206	7D		ADC 033F,X
0207	3F		
0208	03		
0209	CA		DEX
020A	D0		BNE 0205
020B	F9		
020C	85		STA 40
020D	40		
020E	EA		NOP
020F	00		BRK

We have inserted NOPs again in anticipation of a BCD version. Enter Program 9–2 and run it with the following data:

(0041) = 03 (number of elements)
(0340) = 35 (elements)

(0341) = 47
(0342) = 28
Result: (0040) = A4

Write a decimal version and run it with the same data. The answer should be (0040) = 10.

16-BIT ARITHMETIC

We can extend Program 9–2 to handle 16-bit numbers. However, we must now deal with carries between the bytes. Here ADC becomes really useful, since it results in

$$(A) = (A) + (M) + (CARRY)$$

where A is the accumulator and M is a memory location. So all we must do to perform 16-bit addition is

1. Clear CARRY initially.
2. Add the less significant bytes.
3. Add the more significant bytes.

ADC automatically includes the carry in the second addition.

The following program adds an array of 16-bit numbers starting at 0340; it places the sum in 0040 and 0041. Each number occupies 2 bytes, with the less significant part first. The length of the array (how many 16-bit numbers there are) is in 0042.

```
                NOP             ; NOP FOR DECIMAL VERSION
                LDY  $42        ; COUNT = ARRAY LENGTH
                LDA  #0         ; SUM = ZERO INITIALLY
                STA  $40
                STA  $41
                TAX             ; INDEX = ZERO INITIALLY
ADDELM          LDA  $40        ; ADD LSB OF ELEMENT
                CLC
                ADC  $0340,X
                STA  $40
                INX
                LDA  $41        ; ADD MSB OF ELEMENT
                ADC  $0340,X
                STA  $41
                INX
                DEY             ; COUNT ELEMENTS
                BNE  ADDELM
                NOP             ; NOP FOR DECIMAL VERSION
                BRK
```

PROGRAM 9–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	EA		NOP
0201	A4		LDY 42
0202	42		
0203	A9		LDA #00
0204	00		
0205	85		STA 40
0206	40		
0207	85		STA 41
0208	41		
0209	AA		TAX
020A	A5	ADDELM	LDA 40
020B	40		
020C	18		CLC
020D	7D		ADC 0340,X
020E	40		
020F	03		
0210	85		STA 40
0211	40		
0212	E8		INX
0213	A5		LDA 41
0214	41		
0215	7D		ADC 0340,X
0216	40		
0217	03		
0218	85		STA 41
0219	41		
021A	E8		INX
021B	88		DEY
021C	D0		BNE 020A
021D	EC		
021E	EA		NOP
021F	00		BRK

Program 9–3 is the mnemonic-entry version; enter and run it with the following data:

(0042) = 02 (number of 16-bit elements)
(0340) = 3E (LSBs of first element)
(0341) = 47 (MSBs of first element)
(0342) = F5 (LSBs of second element)
(0343) = 2A (MSBs of second element)
Result: (0040) = 33 (LSBs of sum)
(0041) = 72 (MSBs of sum)

That is,

$$\begin{array}{rr} & 473E \\ + & 2AF5 \\ \hline & 7233 \end{array}$$

PROBLEM 9–5

Make Program 9–3 perform decimal (BCD) addition. Use the following data:

	(0042)	= 02	(number of four-digit elements)
	(0340)	= 36	(LSDs of first element)
	(0341)	= 21	(MSDs of first element)
	(0342)	= 97	(LSDs of second element)
	(0343)	= 18	(MSDs of second element)
Result:	(0040)	= 33	(LSDs of sum)
	(0041)	= 40	(MSDs of sum)

That is,

$$\begin{array}{rr} & 2136 \\ + & 1897 \\ \hline & 4033 \end{array}$$

PROBLEM 9–6

Extend the answer to Problem 9–5 so that it places the carries in 0042. Use 0043 for the number of elements. Try the following data:

	(0043)	= 02	(number of four-digit elements)
	(0340)	= 36	(LSDs of first element)
	(0341)	= 21	(MSDs of first element)
	(0342)	= 97	(LSDs of second element)
	(0343)	= 98	(MSDs of second element)
Result:	(0040)	= 33	(LSDs of sum)
	(0041)	= 20	(middle digits of sum)
	(0042)	= 01	(MSDs of sum = carries)

That is,

$$\begin{array}{rr} & 2136 \\ + & 9897 \\ \hline & 12033 \end{array}$$

Be sure to keep the carries as a decimal number.

MULTIPLE-PRECISION ARITHMETIC

We can extend Programs 9–1 through 9–3 to handle numbers of any length. The procedure (see Figure 9–1) is as follows:

1. Initialization.

INDEX = LENGTH OF NUMBERS (IN BYTES)

CARRY = 0, since there is never a carry into the least significant bytes.

2. Add 8 bits.

(BASE1 − 1 + INDEX) = (BASE1 − 1 + INDEX) + (BASE2 − 1 + INDEX) + CARRY

This step produces a new CARRY.

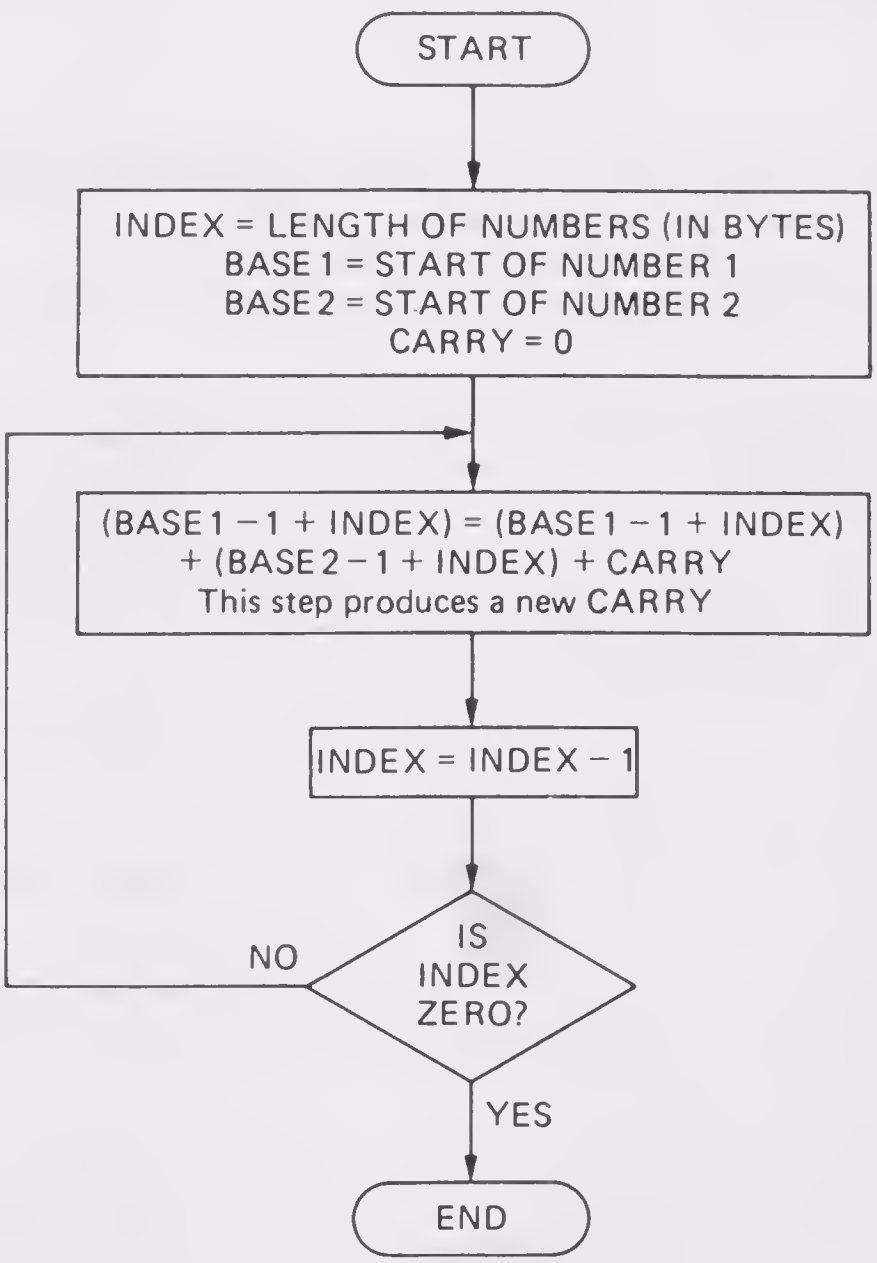


FIGURE 9–1. Flowchart for multiple-precision addition.

3. Update index.

INDEX = INDEX − 1

If INDEX is not 0, return to step 2.

Here the numbers are in the reverse of the usual order, that is, with their least significant bytes at the highest addresses.

If the length of the numbers is in 0040, the numbers start (most significant bytes first) in 0340 and 0360, and the sum replaces the number starting in 0340, the program is

```
            LDX  $40          ; INDEX = LENGTH OF NUMBERS
            CLC               ; CLEAR CARRY INITIALLY
ADBYTE      LDA  $033F,X      ; GET BYTE OF FIRST NUMBER
            ADC  $035F,X      ; ADD BYTE OF SECOND NUMBER
            STA  $033F,X      ; STORE SUM AS FIRST NUMBER
            DEX
            BNE  ADBYTE
            BRK
```

Program 9–4 is the mnemonic-entry version. A key factor is that DEX does not affect CARRY, so its value can be used in the next iteration.

PROGRAM 9–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A6		LDX 40
0201	40		
0202	18		CLC
0203	BD	ADBYTE	LDA 033F,X
0204	3F		
0205	03		
0206	7D		ADC 035F,X
0207	5F		
0208	03		
0209	9D		STA 033F,X
020A	3F		
020B	03		
020C	CA		DEX
020D	D0		BNE 0203
020E	F4		
020F	00		BRK

Try Program 9–4 on the following problem:

```
        (0040) = 04   (length of numbers in bytes)
        (0340) = 29   (MSBs of first number)
        (0341) = 3E
        (0342) = AB
        (0343) = F0   (LSBs of first number)
        (0360) = 19   (MSBs of second number)
        (0361) = D0
        (0362) = 28
        (0363) = A1   (LSBs of second number)
Result: (0340) = 43   (MSBs of sum)
        (0341) = 0E
        (0342) = D4
        (0343) = 91   (LSBs of sum)
```

That is,

```
   293EABF0
+  19D028A1
  ---------
   430ED491
```

PROBLEM 9–7

Write a program that adds decimal numbers of arbitrary length. Assume the same conditions as in Program 9–4. Use the following sample case:

```
        (0040) = 04   (length of numbers in bytes)
        (0340) = 29   (MSDs of first number)
        (0341) = 34
        (0342) = 71
        (0343) = 60   (LSDs of first number)
        (0360) = 19   (MSDs of second number)
        (0361) = 60
        (0362) = 28
        (0363) = 81   (LSDs of second number)
Result: (0340) = 48   (MSDs of sum)
        (0341) = 95
        (0342) = 00
        (0343) = 41   (LSDs of sum)
```

That is,

```
   29347160
+  19602881
  ---------
   48950041
```

PROBLEM 9–8

Write a program that subtracts decimal numbers of arbitrary length. Assume the same conditions as in Program 9–4. Subtract the number starting in 0360 from the one starting in 0340. Try the program on the following example. CARRY, as usual, is an inverted borrow.

```
        (0040) = 04   (length of numbers in bytes)
        (0340) = 29   (MSDs of minuend)
        (0341) = 34
        (0342) = 71
        (0343) = 60   (LSDs of minuend)
        (0360) = 19   (MSDs of subtrahend)
        (0361) = 60
        (0362) = 28
        (0363) = 81   (LSDs of subtrahend)
Result: (0340) = 09   (MSDs of difference)
        (0341) = 74
        (0342) = 42
        (0343) = 79   (LSDs of difference)
```

That is,

```
  29347160
- 19602881
  09744279
```

ARITHMETIC WITH LOOKUP TABLES

One way to do complex arithmetic is by using lookup tables that contain all possible results. The program must then locate the desired result, just as in the code conversion of Program 5–7.

For example, suppose we form a table of the squares of the decimal digits. The following program uses it to square the digit in 0041; it places the square in 0042.

```
LDX  $41                ; GET DATA
LDA  $0340,X            ; GET SQUARE FROM TABLE
STA  $42
BRK

*=$0340                 ; SQUARES OF DECIMAL DIGITS
.BYTE 0,1,4,9,16,25,36,49,64,81
```

*= ("set origin") is an assembler directive (called a *pseudo-operation*); it indicates where the machine language generated from subsequent statements is to be placed in memory.

The directive .BYTE (Form Byte-Length Data) places a list of 8-bit data items in the next available locations. Program 9–5 is the mnemonic-entry version.

PROGRAM 9–5

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A6	LDA 41
0201	41	
0202	BD	LDA 0340,X
0203	40	
0204	03	
0205	85	STA 42
0206	42	
0207	00	BRK
0340	00	.BYTE 0
0341	01	1
0342	04	4
0343	09	9
0344	10	16
0345	19	25
0346	24	36
0347	31	49
0348	40	64
0349	51	81

Run Program 9–5 with the following sample data:

1. (0041) = 04
   Result: (0042) = 10
2. (0041) = 07
   Result: (0042) = 31

PROBLEM 9–9

Write a program that uses the table in Program 9–5 to add the squares of 0040 and 0041. The sum should end up in 0042.

Example

(0040) = 03
(0041) = 06
Result: (0042) = 2D (hex), since 2D = 09 (3^2) + 24 (6^2).

PROBLEM 9–10

Write a program that uses a table to cube a decimal digit. Allow 2 bytes for each entry, since some cubes are larger than 256. Assume that the data is in 0041, and place the result in 0042 and 0043 (MSBs in 0043).

Examples

1. (0041) = 03
   Result: (0042) = 1B
   (0043) = 00
2. (0041) = 07
   Result: (0042) = 57
   (0043) = 01

The results are hexadecimal numbers.

PROBLEM 9–11

Write a program that takes the four-digit square root of a decimal digit. The digit is in 0041, and the square root ends up in 0042 and 0043 (most significant digits in 0043). Use the following table; enter it in memory starting at 0340 and indicate it in your program with a .WORD (Form Double-Byte-Length Data) pseudo-operation.

Value	Square Root
0	00.00
1	01.00
2	01.41
3	01.73
4	02.00
5	02.24
6	02.45
7	02.65
8	02.83
9	03.00

Examples

1. (0041) = 03
   Result: (0042) = 73
   (0043) = 01
2. (0041) = 07
   Result: (0042) = 65
   (0043) = 02

PROBLEM 9–12

Extend the answer to Problem 9–11 to produce a six-digit square root in 0042, 0043, and 0044 (most significant digits in 0044). Use the following table:

Value	Square Root
0	00.0000
1	01.0000
2	01.4142
3	01.7321
4	02.0000
5	02.2361
6	02.4495
7	02.6458
8	02.8284
9	03.0000

Examples

1. (0041) = 02
   Result: (0042) = 42
   (0043) = 41
   (0044) = 01
2. (0041) = 06
   Result: (0042) = 95
   (0043) = 44
   (0044) = 02

If the table is long, you could consider keeping only some entries in memory. You could, for example, keep every tenth entry and interpolate to obtain intermediate values.

KEY POINT SUMMARY

1. BCD is a convenient way to represent decimal numbers, since each digit is coded separately. However, BCD requires more memory and processing instructions than the binary representation.
2. The 6502 processor has a special decimal mode in which additions and subtractions produce BCD results automatically. The processor enters and leaves the decimal mode by executing SED and CLD, respectively. The programmer must initialize the D (DECIMAL MODE) flag and keep track of its value. Increments and decrements always produce binary results.
3. Multiple-precision arithmetic requires a series of 8-bit operations. The CARRY flag transfers carries or borrows between them.

4. Lookup tables provide a simple way to perform complex arithmetic. The lookup procedure depends only on the organization of the table and the length of the elements; it does not depend on the data values or the function involved.

SUBROUTINES AND THE STACK

PURPOSE

To learn how to write and use subroutines.

REFERENCE MATERIALS

R. C. CAMP et al., *Microprocessor Systems Engineering*, Matrix Publishers, Portland, OR, 1979, Chapter 8 (particularly pp. 499–505), Appendix A.

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 57–60, 97–100, 113–115, 120, 220–229.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 9–16 through 9–17, Chapters 10, 15.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, 1982.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 172–174 (stack pointer), 342–345 (stack transfers), 355–360 (subroutines), 367–368 (indirect addressing), 368–375 (timing loops).

AIM 65 Monitor Program Listing, Dynatem, Irvine, CA, 1979.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1978, pp. 6–34 to 6–36, 7–37 to 7–93.

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, Chapter 8.

WHAT YOU SHOULD LEARN

1. How to transfer control to and from subroutines.
2. How to use the stack.
3. How to use monitor subroutines.
4. How to transfer control to one of a set of subroutines.

TERMS

Indirect jump a jump to an address stored in a register or in memory.

Overflow exceeding the amount of memory allocated to a stack.

Parameter an item that a subroutine needs to execute properly.

Passing parameters making parameters available to a subroutine.

Pop (or **pull**) remove an operand from a stack.

Push store an operand in a stack.

Stack a section of memory that can be accessed in a last-in, first-out manner. That is, data can be added to or removed from it through its top; new data is placed above the old data, and the removal of a data item makes the item below it the new top.

Stack pointer a register that contains the address of the top of a stack.

Subroutine a subprogram that can be called by another program.

Subroutine call the process whereby a computer transfers control to a subroutine while retaining the information required to resume the current program.

Subroutine linkage the mechanism whereby a computer retains the information required to resume its current program after executing a subroutine.

6502 INSTRUCTIONS

JSR jump to subroutine; jump to a memory address and save the return address (the address of JSR's third byte) in the stack.

PHA(P) store the accumulator (status register) at the top of the stack and decrement the stack pointer by 1.

PLA(P) increment the stack pointer by 1 and load the accumulator (status register) from the top of the stack.

RTS return from subroutine; load the program counter from the top two stack locations and then add 1 to it. The result is a jump to the address one larger than the contents of the top two stack locations.

TSX transfer stack pointer to index register X. This is the only way to save the stack pointer.

TXS transfer index register X to stack pointer. This is the only way to load the stack pointer.

Note: In JSR, PHA, PHP, PLA, PLP, and RTS, the top of the stack is at address 01*ss* hex, where *ss* is the contents of the stack pointer.

RATIONALE AND TERMINOLOGY

Most tasks described so far occur many times in real applications. For example, actual programs often include many time delays, code conversions, and arithmetic functions. Clearly, repeating sequences of instructions each time would waste memory and programming time. We would prefer to use one copy of each common sequence.

To allow this, we need a way for the processor to suspend its current program, execute a sequence, and resume the current program where it left off. Then the processor could use the same sequence from many different points in its overall program. The sequence could even be part of the monitor; in that case, the programmer would not have to code it or load it into memory.

We use the following terminology in describing common sequences of instructions:

- The sequence is a *subroutine*, since it is subordinate to the calling program.
- The process of transferring control to the subroutine is a *subroutine call*.
- A piece of data or an address that a subroutine needs is a *parameter*.
- The process of providing the subroutine with parameters is *passing parameters*.
- The method whereby the computer transfers control to the subroutine and back to the calling program is a *subroutine linkage*.

CALL AND RETURN INSTRUCTIONS

The JSR and RTS instructions are essential for implementing subroutines; they work as follows:

- JSR (JUMP TO SUBROUTINE) saves the address of its own third byte in the stack before placing a new value in the program counter. It allows only absolute (direct) addressing.

- RTS (RETURN FROM SUBROUTINE) loads the program counter from the top two locations in the staek and adds 1 to it.

A JSR in the ealling program transfers control to the subroutine that starts at the spccified absolute address. An RTS at the end of the subroutine returns eontrol to the ealling program just after the JSR. The subroutine linkage is thus in the staek—JSR saves the return address there and RTS retrieves it.

6502 STACK AND STACK POINTER

To explain JSR and RTS, we must deseribe how the 6502 transfers data to and from its staek. This works as follows (see Figures A–1 and A–2):

1. To save data (ealled a *push*), it first stores the data at the top of the staek and then subtraets 1 from the staek pointer.
2. To remove data (ealled a *pop* or *pull*), it first adds 1 to the stack pointer and then loads the data from the top of the staek.

The staek is always on page 1 of memory. Its top (the next empty loeation) is at address 01*ss* hex, where *ss* is the eontents of the staek pointer.

Note the following:

1. The stack is just an ordinary area of memory. The proeessor moves the top of the stack up or down by deereasing or inereasing the stack pointer (examine Figures A–1 and A–2 earefully).

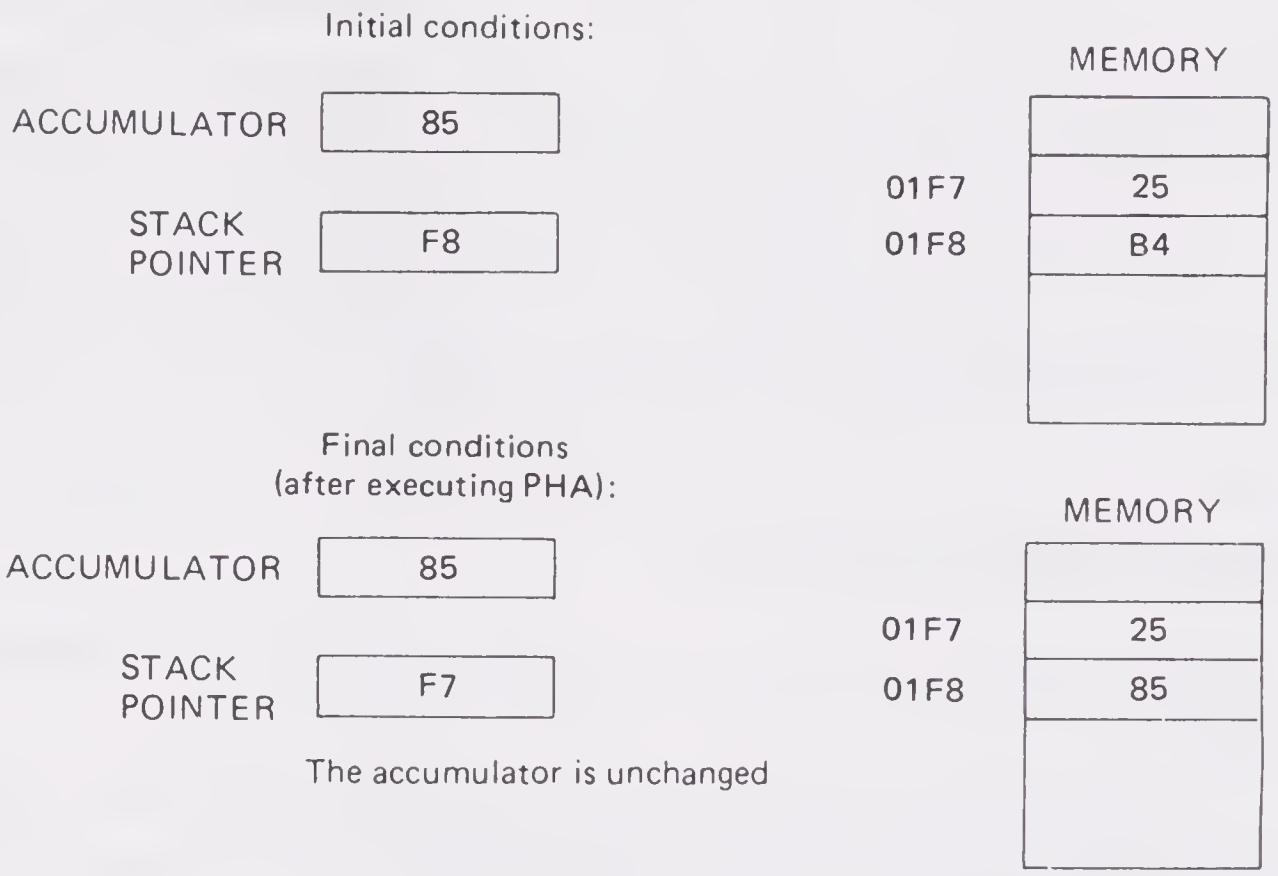


FIGURE A–1. Entering data into the stack (a *push*).

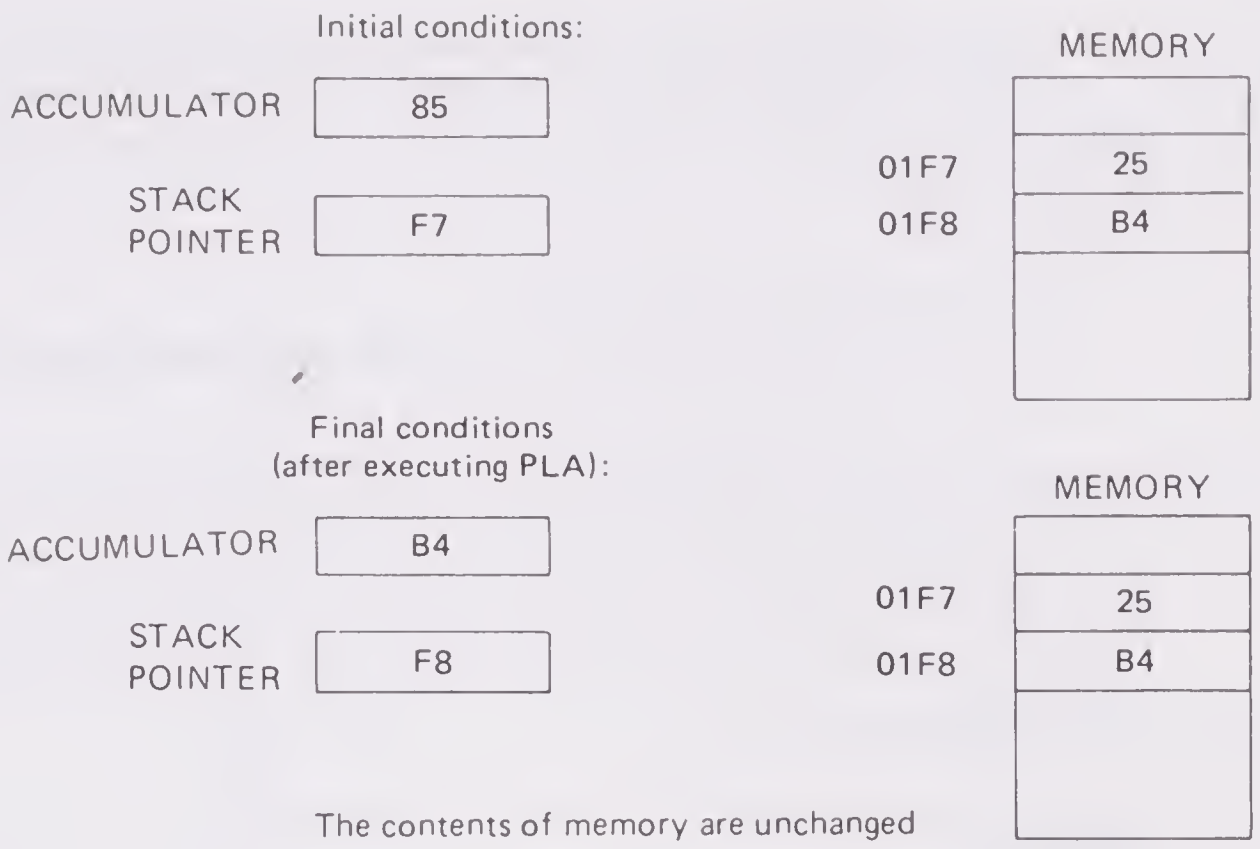


FIGURE A–2. Removing data from the stack (a *pop* or *pull*).

2. The programmer (or the monitor program) selects where the stack begins (on page 1) by initializing the stack pointer. The sequence LDX, TXS is the only way to do this. Since the AIM 65 monitor starts its stack at 01FF, we will start ours at 017F to avoid confusion. Programs seldom change the stack pointer explicitly after it has been initialized.
3. The stack grows down (i.e., toward lower addresses). If this makes you uneasy, stand on your head and everything will be all right.
4. The stack pointer always contains the next available (empty) stack address. The lowest address actually occupied by the stack is 1 larger.
5. JSR and RTS transfer 16-bit addresses to and from the stack. The less significant byte is obtained first and stored last in accordance with the usual 6502 method for storing addresses. Be careful: The less significant byte is stored last, but it ends up at the lower address because the stack is growing down. Note the strange offset of 1; JSR saves the address of its own third byte in the stack, and RTS adds 1 to the address it obtains from the stack. This apparently makes JSR execute faster, but it is also another quirk for the programmer to remember.

Examples

a. (S) = 61
(PC) = 021C
After the processor executes JSR $0238 (occupying addresses 021C through 021E),
(S) = (S) − 2 = 5F, since a 2-byte address has been saved in the stack.
(PC) = 0238, the starting address of the subroutine.

(0160) = 1E, the LSBs of the address of JSR's third byte.
(0161) = 02, the MSBs of the address of JSR's third byte.

b. (S) = 7C
(017D) = 28
(017E) = 02
After the processor executes RTS,
(S) = (S) + 2 = 7E, since a 2-byte address has been removed from the stack.
(PC) = (017E) (017F) = 0228 + 1 = 0229.

6. PLA (load accumulator from stack) and PLP (load status register from stack) load a register from the top of the stack. PHA (store accumulator in stack) and PHP (store status register in stack) store a register at the top of the stack. There is no direct path between the stack and an index register; the data must move through the accumulator.

Examples

a. (S) = 67
(A) = F2
After the processor executes PHA,
(S) = 66
(0167) = (A) = F2
The accumulator does not change.

b. (S) = 6F
(0170) = 3B
After the processor executes PLA,
(S) = 70
(A) = (01*ss*) = (0170) = 3B
Location 0170 does not change, but it is no longer part of the stack. The stack expands and contracts like ocean waves which alternately cover and uncover parts of the shoreline.

GUIDELINES FOR STACK MANAGEMENT

Most beginners find the stack confusing and even a little frightening. However, it is easy to manage if you follow these guidelines.

1. Load the stack pointer during system initialization. Start the stack at the highest available address on page 1.
2. Always pair stack operations. Pair each JSR with an RTS and each PHA or PHP with a PLA or PLP. This is just like pairing left and right parentheses in arithmetic or quotation marks in sentences.

3. Don't be faney. Leave the stack and stack pointer alone exeept for JSR, RTS, and push and pull instructions. Simple programs rarely need more than 20 bytes for the stack. Leave lots of room so the stack never overflows; many 6502 programmers simply assign all of page 1 to the stack.

SUBROUTINE LINKAGES IN THE STACK

Let us see how JSR and RTS work in a simple situation. Enter the following program into memory:

```
STARTING AT $0200

        LDX  #$7F          ; INITIALIZE USER STACK POINTER
        TXS
        JSR  $0260         ; GO TO SUBROUTINE
        BRK

STARTING AT $0260

        TSX                ; SAVE STACK POINTER
        STX  $40
        BRK
```

Program A–1 is the mnemonic-entry version. Note that we need the sequenee TSX, STX to save the stack pointer in memory.

PROGRAM A–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A2	LDX #7F
0201	7F	
0202	9A	TXS
0203	20	JSR 0260
0204	60	
0205	02	
0206	00	BRK
0260	BA	TSX
0261	86	STX 40
0262	40	
0263	00	BRK

PROBLEM A–1

What is in the stack pointer and locations 017E, 017F, and 0040 after you run Program A–1? Explain why 017E and 017F do not contain the address of the next instruction after JSR. Do not reset the AIM after running Program A–1. What happens to the stack pointer if you do? Do 017E and 017F change?

PROBLEM A–2

What are the final values of the stack pointer and locations 017E and 017F if you replace the BRK in 0263 with RTS? Explain the changes.

PROBLEM A–3

What are the final values of the stack pointer and locations 017C through 017F if you put the following instructions in memory? Remember to execute the main program starting at 0200.

```
0260   BA     TSX
0261   86     STX  40
0262   40
0263   20     JSR  0280
0264   80
0265   02
0266   60     RTS

0280   BA     TSX
0281   86     STX  41
0282   41
0283   00     BRK
```

What do 0040 and 0041 contain? What happens if you revise the program as follows?

```
0260   BA     TSX
0261   86     STX  40
0262   40
0263   20     JSR  0280
0264   80
0265   02
0266   BA     TSX
0267   86     STX  42
0268   42
0269   00     BRK

0280   BA     TSX
0281   86     STX  41
0282   41
0283   60     RTS
```

SAVING REGISTERS IN THE STACK

If you save the registers in the stack before a call, you need not worry about whether the subroutine uses them. Remember the following:

1. You can save and restore the accumulator with PHA and PLA.
2. You can save and restore the status register (Figure 2–2) with PHP and PLP.
3. You can save and restore an index register only via the accumulator. The sequences are:

```
TX(Y)A                  ; SAVE INDEX REGISTER IN STACK
PHA
```

and

```
PLA                     ; RESTORE INDEX REGISTER FROM STACK
TAX(Y)
```

Since these sequences use the accumulator, you must save it first and restore it afterward.

4. You must restore registers in the opposite order of that in which you saved them. If you save them with

```
PHP                     ; SAVE STATUS
PHA                     ; SAVE ACCUMULATOR
TXA                     ; SAVE X
PHA
TYA                     ; SAVE Y
PHA
```

you must restore them with

```
PLA                     ; RESTORE Y
TAY
PLA                     ; RESTORE X
TAX
PLA                     ; RESTORE ACCUMULATOR
PLP                     ; RESTORE STATUS
```

DELAY SUBROUTINE

The following subroutine derived from Program 4–3 produces a 1-ms delay:

```
DLYMS   LDX  #$C8       ; DELAY 1 MS
DLY     DEX
        BNE  DLY
        RTS
```

We can use it in a main program as follows:

```
LDX  #$7F          ; INITIALIZE USER STACK POINTER
TXS
JSR  DLYMS         ; DELAY 1 MS
BRK
```

Program A–2 is the mnemonic-entry version of the main program and subroutine. Enter it into memory and run it.

PROGRAM A–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #7F
0201	7F		
0202	9A		TXS
0203	20		JSR 0260
0204	60		
0205	02		
0206	00		BRK
0260	A2	DLYMS	LDX #C8
0261	C8		
0262	CA	DLY	DEX
0263	D0		BNE 0262
0264	FD		
0265	60		RTS

PROBLEM A–4

Make the subroutine preserve the status register, accumulator, and index register X. How much do the added instructions increase the execution time?

PROBLEM A–5

Revise the subroutine to produce a delay in seconds rather than in milliseconds. Have the subroutine preserve all registers. Use 0041 for the count in seconds and use the 1-s delay program from Problem 8–8.

Example:
(0041) = 05 results in a delay of 5 s.

INPUT SUBROUTINE

The following subroutine (derived from Program 4–5) encodes a switch closure. It assumes that the accumulator contains the data from an input port attached to eight switches.

```
IDSW    LDY  #$FF        ; SWITCH NUMBER = -1
SRCHS   INY              ; INCREMENT SWITCH NUMBER
        LSR  A           ; IS NEXT SWITCH CLOSED?
        BCS  SRCHS       ; NO, KEEP LOOKING
        RTS
```

Program A-3 is the mnemonic-entry version. We started at 0270 to avoid interfering with the 1-ms delay (Program A–2). The following program waits until a switch is closed at port A of the user VIA and then uses Program A–3 to identify it:

PROGRAM A–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0270	A0	IDSW	LDY #FF
0271	FF		
0272	C8	SRCHS	INY
0273	4A		LSR A
0274	B0		BCS 0272
0275	FC		
0276	60		RTS

```
        LDX  #$7F        ; INITIALIZE USER STACK POINTER
        TXS
WAITC   LDA  $A001       ; GET DATA FROM SWITCHES
        CMP  #$FF        ; ARE ANY SWITCHES CLOSED?
        BEQ  WAITC       ; NO, WAIT
        JSR  IDSW        ; YES, IDENTIFY CLOSED SWITCH
        STY  $40         ; SAVE SWITCH NUMBER
        BRK
```

Program A–4 is the mnemonic-entry version of the main program. Enter and run it. Where does the subroutine put the switch number?

PROBLEM A–6

How could you make Program A–4 examine the switches once and conclude with either the switch number or FF (if no switches are closed) in 0040?

PROGRAM A–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #7F
0201	7F		
0202	9A		TXS
0203	AD	WAITC	LDA A001
0204	01		
0205	A0		
0206	C9		CMP #FF
0207	FF		
0208	F0		BEQ 0203
0209	F9		
020A	20		JSR 0270
020B	70		
020C	02		
020D	84		STY 40
020E	40		
020F	00		BRK

PROBLEM A–7

Modify Program A–4 to wait for the number of switch closures specified in 0042 and save the identification numbers starting at 0340. Use subroutine DLYMS to provide a 1-ms delay for debouncing.

Example:
If (0042) = 03 and you close switches 0, 6, and 5 in that order, the result should be

(0340) = 00
(0341) = 06
(0342) = 05

Assume that you must open all switches between closures.

OUTPUT SUBROUTINE

The next subroutine (derived from Programs 5–1 and 5–3) converts a decimal digit in the accumulator to ASCII and shows it in the rightmost character of the on-board display. Program A–5 is the mnemonic-entry version. The subroutine does not send anything to the display if the accumulator does not contain a decimal digit.

```
DSP1    CMP  #10        ; IS DATA A DECIMAL DIGIT?
        BCS  DONE       ; NO, EXIT
        LDX  #$7B       ; YES, ACTIVATE LEFTMOST CHARACTER
        STX  $AC00
        CLC             ; CONVERT DATA TO ASCII
        ADC  #$B0
        STA  $AC02      ; SEND ASCII DATA TO DISPLAY
DONE    RTS
```

PROGRAM A–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
02A0	C9	DSP1	CMP #0A
02A1	0A		
02A2	B0		BCS 02AF
02A3	0B		
02A4	A2		LDX #7B
02A5	7B		
02A6	8E		STX AC00
02A7	00		
02A8	AC		
02A9	18		CLC
02AA	69		ADC #B0
02AB	B0		
02AC	8D		STA AC02
02AD	02		
02AE	AC		
02AF	60	DONE	RTS

PROBLEM A–8

Write a main program that uses Program A–5 and the 1-s delay routine (Problem 8–8) to show (0042) on the leftmost character for 1 s.

USING MONITOR SUBROUTINES

JSR also lets us use subroutines from the AIM monitor. Among these (see Table A–1) are routines that handle input and output, perform code conversions, and generate time delays.

An easy monitor routine to use is the time-delay DELAY, which starts in address EC0F. The length of the delay depends on the contents of A417 and A418 (more significant byte in A418). We can use DELAY as follows:

```
LDX  #$7F          ; INITIALIZE USER STACK POINTER
TXS
LDA  #0            ; DELAY CONSTANT = 8000 HEX
STA  $A417
LDA  #$80
STA  $A418
JSR  DELAY         ; WAIT A WHILE
BRK
```

PROGRAM A–6

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A2	LDX #7F
0201	7F	
0202	9A	TXS
0203	A9	LDA #00
0204	00	
0205	8D	STA A417
0206	17	
0207	A4	
0208	A9	LDA #80
0209	80	
020A	8D	STA A418
020B	18	
020C	A4	
020D	20	JSR EC0F
020E	0F	
020F	EC	
0210	00	BRK

Program A–6 is the mnemonic-entry version; enter and run it. Try the following values for the more significant byte of the delay constant (address 0209): 40, 20, 10, 08, 04, 02, 01. When can you no longer see the delay? Using the LEDs attached to port B of the user VIA will make the time lapse more obvious. Make the port output and turn all the LEDs on (with 0 bits) before calling DELAY and off (with 1 bits) afterward.

One problem with using the monitor routines is that you cannot single-step through them. The AIM disables the STEP mode when it is executing instructions in its monitor to avoid conflict with normal operating functions. So if the AIM executes a monitor subroutine in the STEP mode, it will finish the whole thing and stop after the first instruction it executes from user memory.

Even though JSR does not change any registers or flags, the subroutine may. In general, you must preserve any values you need by saving them in the stack before calling the subroutine. Note the importance of knowing which registers a subroutine affects.

PROBLEM A–9

Determine which of the following registers and flags DELAY affects by experimenting with their values in the STEP mode.

a. Accumulator.
b. Index register Y.
c. Index register X.
d. ZERO flag (bit 1 of register P).
e. CARRY flag (bit 0 of register P).

Does DELAY change A+17 and A+18?

PROBLEM A–10

Use subroutine DELAY to write a program that flashes 0 on the leftmost character of the on-board display. Experiment with the delay constant until the flashing is readily visible.

USING THE OUTPUT SUBROUTINES

Subroutine OUTDIS (starting address EF05) sends an ASCII character to the AIM display. It also adds 1 to the display pointer, so the next character will be placed 1 position to the right. The following program (Program A–7 is the mnemonic-entry version) shows the contents of 0380 through 0387 on the display. The jump-to-self at the end keeps the AIM from overwriting the message before we can see it.

```
        LDX  #$7F          ; INITIALIZE USER STACK POINTER
        TXS
        LDX  #0            ; START WITH LEFTMOST CHARACTER
DISPC   LDA  $0380,X       ; GET A CHARACTER
        JSR  OUTDIS        ; DISPLAY IT
        INX                ; MOVE ON TO NEXT CHARACTER
        CPX  #8            ; DONE WITH 8 CHARACTERS?
        BNE  DISPC         ; NO, KEEP DISPLAYING MORE
HERE    JMP  HERE          ; YES, WAIT FOREVER
```

Enter Program A–7 into memory and run it with the following data. Note that we are not setting bit 7 of the ASCII characters here. OUTDIS uses bit 7 to decide whether to clear the display to the right. If bit 7 = 0, it clears all characters to the right of where it is working.

(0380) = 43 (leftmost character)
(0381) = 4F
(0382) = 4D
(0383) = 50
(0384) = 55
(0385) = 54
(0386) = 45
(0387) = 52 (rightmost character)

To form your own messages, use the ASCII table in Appendix 2. A final BRK is unnecessary, since Program A–7 never returns control to the monitor.

PROGRAM A–7

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #7F
0201	7F		
0202	9A		TXS
0203	A2		LDX #00
0204	00		
0205	BD	DISPC	LDA 0380,X
0206	80		
0207	03		
0208	20		JSR EF05
0209	05		
020A	EF		
020B	E8		INX
020C	E0		CPX #08
020D	08		
020E	D0		BNE 0205
020F	F5		
0210	4C	HERE	JMP 0210
0211	10		
0212	02		

PROBLEM A–11

Expand the message to THE COMPUTER IS ON and show it on the display. What happens if you expand it to THE COMPUTER IS NOT WORKING?

TABLE A–1 AIM 65 MONITOR SUBROUTINES

Subroutine Name	Starting Address	Registers Affected*	Description
BLANK	E83E	A	Sends a space character (20 hex) to display or printer.

TABLE A–1 (continued) AIM 65 MONITOR SUBROUTINES

Subroutine Name	Starting Address	Registers Affected*	Description
BLANK2	E83B	A	Sends two spaee eharaeters (20 hex) to display or printer.
CLR	EB44	A	Clears display and printer pointers.
CRCK	EA24	A	Sends eontents of print buffer to printer if print pointer is not clear.
CRLF	E9F0	A	Sends ASCII CR, LF to the aetive output deviee.
CRLOW	EA13		Sends ASCII CR, LF to the display or printer.
DEBKI	ED2C	A	Generates a 5-ms delay.
DEHALF	EC2C	A	Delays 1/2 bit time aeeording to the eontents of CNTH30 (A417) and CNTL30 (A418).
DELAY	EC0F	A	Delays 1 bit time aeeording to the eontents of CNTH30 (A417) and CNTL30 (A418).
DISASM	F46C		Sends the disassembly of the eurrent instruetion pointed to by SAVPC (A425) to the aetive output deviee.
DUMPTA	E56F		Opens an audio-tape output file.
DUTI	E50A		Closes an autio-tape bloek.
EQUAL	E7D8	A	Sends an ASCII = to the display or printer.
FROM	E7A3	A,X,Y	Sends "FROM=" to the display or printer and puts the entered address in ADDR (A41C) and ADDR+1 (A41D).
GETTAP	EE29	A,Y	Reads a eharaeter from audio tape into A.
GETTY	EBDB	A,Y	Reads a eharaeter from TTY into A.
HEX	EA7D	A	Converts an ASCII hexadeeimal digit in A to hex. Puts the result in the LSD of A and elears the MSD of A. Sets CARRY to 1 if A does not eontain a hex digit.
INALL	E993	A	Reads an ASCII eharaeter from the aetive input deviee into A.
INLOW	E8F8	A	Puts ASCII CR in INFLG (A412) to indieate keyboard input.
LL	E8FE	A	Puts ASCII CR in INFLG (A412) and OUTFLG (A413) to indieate input from the keyboard and output to the display or printer.
LOADTA	E32F		Searehes for an audio-tape file with the name speeified in NAME (A42E).

TABLE A–1 (continued) AIM 65 MONITOR SUBROUTINES

Subroutine Name	Starting Address	Registers Affected*	Description
NOUT	EA51	A	Converts bits 0–3 of A to ASCII and sends them to the active output device.
NUMA	EA46	A	Converts A to two ASCII hex digits and sends them to the active output device, MSD first.
OUTALL	E9BC		Sends an ASCII character in A to the active output device.
OUTDIS	EF05		Sends an ASCII character in A to the display. Clears display to the right if bit 7 of A is 0. Scrolls the display left if more than 20 characters but less than 60 have been displayed since last CR.
OUTDP	EEFC		Sends an ASCII character to the display and the print buffer. Links to OUTDIS indirectly through DILINK (A406).
OUTLOW	E901	A	Puts ASCII CR in OUTFLG (A413) to indicate output to the display or printer.
OUTPRI	F000		Sends an ASCII character in A to the print buffer. Prints a line if the character is ASCII CR or if 20 characters are in the buffer.
OUTPUT	E97A		Sends an ASCII character from A to the display or printer.
OUTTAP	F24A	Y	Sends a character from A to tape.
PACK	EA84	A	Converts an ASCII hex number in A to hex and puts the result alternately in the MSD or LSD of A.
PHXY	EB9E		Pushes X and Y onto the stack.
PLXY	EBAC	X,Y	Pulls X and Y from the stack.
PSLI	E8E7	A	Sends ASCII / to the display or printer.
QM	E7D4	A	Sends ASCII ? to the display or printer.
RBYTE	E3FD	A	Reads two characters from the active input device. Converts them to hex and packs them into 1 byte if they are hex digits.
RCHEK	E907	A,X,Y	Scans the keyboard. Returns to caller if no keys are depressed. Returns to monitor if ESC is depressed. Waits for another key to be depressed if the space bar is depressed.

TABLE A–1 (continued) AIM 65 MONITOR SUBROUTINES

Subroutine Name	Starting Address	Registers Affeeted*	Deseription
RDRUB	E95F	A,Y	Reads a eharaeter from the keyboard and eehoes it to the display or printer. Allows RUBOUT to delete the eharaeter.
READ	E93C	A	Reads a eharaeter from the keyboard into A.
REDOUT	E973	A	Reads a eharaeter from the keyboard and echoes it to the display or printer if it is not a earriage return.
SEMI	E9BA	A	Sends an ASCII ; to the aetive output deviee.
TAISET	EDEA		Cheeks for SYN (16 hex) eharaeter on tape and returns to ealling routine if it detects five eonsecutive SYNs.
TAOSET	F21D		Sends the number of SYNs given by GAP (A409) times 4 to the tape.
TIBYTE	ED3B	A	Loads an input eharaeter from the audio-tape buffer into A. Reads a block of data from the recorder if tape buffer is empty.
TIBY1	ED53		Loads 80 bytes from audio tape into the tape buffer when BLK (0115) is 0.
TO	E7A7	A,X,Y	Sends "TO=" to the display or printer and puts the entered address in ADDR (A41C) and ADDR+1 (A41D).
TOBYTE	F18B	A	Stores a eharaeter in the audio-tape buffer. Sends a bloek of data to the recorder if tape buffer is full.
WHEREI	E848	A,X,Y	Determines and sets up the active input deviee from the answer to "IN=" and puts the device code in INFLG (A412).
WHEREO	E871	A,X,Y	Determines and sets up the aetive output device from the answer to "OUT=" and puts the device code in OUTFLG (A413).

*All routines change the P register.

CALLING VARIABLE ADDRESSES

Since JSR allows only absolute addressing, we need special techniques to let a system choose among subroutines. For example, many interaetive systems ask the operator what

to do next (e.g., continue, start over, change parameters, repeat, report results, or stop). Many systems also have function keys that the operator uses to perform common procedures such as mathematical or statistical functions, loading of programs or data, or graphics operations. Whatever the case, the system does not know which subroutine to execute until the operator selects one.

Assume that we have a table of starting addresses. For example, in a calculator, these addresses might be entry points for the sine, cosine, exponential, logarithm, reciprocal, and other routines. In a piece of test equipment, they might be entry points for the self-test, initial condition setting, parameter selection, and data analysis routines. To transfer control to a particular routine, all we need is the table's base address and the entry number.

The way to overcome JSR's limitations is to transfer control to an intermediate routine. This routine can use JMP with indirect addressing to transfer control to an address stored in memory. It must do the following:

1. Use indexed addressing to obtain the starting address of the routine from the table. We must remember to double the entry number before indexing, since each address occupies 2 bytes.
2. Store the starting address in memory so that it can be used indirectly. We will store it in 0040 and 0041.
3. Jump indirectly. We indicate an indirect jump in assembly language or mnemonic entry by placing parentheses around the address, for example, JMP ($0040) transfers control to the address stored in 0040 and 0041.

The following program (see Program A–8 for a mnemonic-entry version) performs these steps. We have assumed that the entry number is in 0042 and that the table of addresses starts at 0340.

```
JCALC   LDA  $42        ; GET ENTRY NUMBER
        ASL  A          ; DOUBLE IT FOR 2-BYTE ENTRIES
        TAX
        LDA  $0340,X    ; GET LSB OF ENTRY
        STA  $40
        LDA  $0341,X    ; GET MSB OF ENTRY
        STA  $41
        JMP  ($0040)    ; TRANSFER CONTROL TO ENTRY
```

We do not need a final BRK, since JMP ($0040) transfers control. However, to run the program, we will need a table of addresses and a BRK instruction at each destination for testing pruposes. For example, if the table is

```
* = $0340
.WORD    $0260, $0280, $02A0, 02C0
```

we must place BRK in 0260, 0280, 02A0, and 02C0.

Enter and run Program A–8 with this four-entry table. Show that it works properly for (0042) = 00, 01, 02, and 03.

PROGRAM A–8

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A5	JCALC	LDA	42
0201	42			
0202	0A		ASL	A
0203	AA		TAX	
0204	BD		LDA	0340,X
0205	40			
0206	03			
0207	85		STA	40
0208	40			
0209	BD		LDA	0341,X
020A	41			
020B	03			
020C	85		STA	41
020D	41			
020E	6C		JMP	(0040)
020F	40			
0210	00			
0260	00		BRK	
0280	00		BRK	
02A0	00		BRK	
02C0	00		BRK	
0340	60		WORD	0260,
0341	02			
0342	80			0280,
0343	02			
0344	A0			02A0,
0345	02			
0346	C0			02C0
0347	02			

PROBLEM A–12

Write a main program that simply initializes the stack pointer to 7F and then calls Program A–8. What are the final contents of the stack pointer and locations 017E and

017F? What happens if you replace the BRKs in 0260, 0280, 02A0, and 02C0 with RTSs?

PROBLEM A–13

Revise Program A–8 to exit immediately if 0042 contains an invalid value (i.e., larger than 3). Another approach to error handling is to end the table with an error exit and replace all invalid entries with its length. Revise Program A–8 to use this approach. (*Hint:* For example, put an error exit in 0348 and 0349. The program would replace any entry above 4 with 4, thus causing a jump to the error exit.)

An alternative approach is to put the starting address of the subroutine in the stack and use RTS to transfer control. It seems strange to use RTS to call a subroutine, but it works. After all, RTS jumps to the address at the top of the stack. If the stack contains a starting address rather than a return address, RTS will jump to a subroutine rather than back to a calling program. Since this approach is confusing, the documentation should explain what is happening.

PROBLEM A–14

Revise Program A–8 to store the starting address in the stack and use RTS to jump. What changes must you make in the table? Remember that RTS adds 1 to the address it obtains from the stack.

KEY POINT SUMMARY

1. You can make a single copy of a sequence of instructions available from anywhere in a program by making it into a subroutine. The process of transferring control to the subroutine is referred to as a subroutine call, and the items the subroutine needs for proper execution are called parameters.
2. On the 6502, a JUMP TO SUBROUTINE (JSR) instruction in the calling program transfers control to a subroutine and saves the address of its own last byte in the stack. A RETURN FROM SUBROUTINE (RTS) instruction at the end of the subroutine restores control to the calling program by loading the program counter from the top of the stack and adding 1 to it.
3. The stack is just an ordinary area of read/write memory on page 1. The stack pointer contains the less significant byte of the address of the next available stack location. All that happens as the stack expands or contracts is that the stack pointer decreases or increases. The stack grows down (i.e., toward lower addresses).
4. The programmer must initialize the stack pointer (using LDX, TXS) before calling subroutines or using the stack for other purposes.
5. You can use the stack for temporary storage. This is convenient since the stack is ordered and easy to expand.

6. You can use monitor subroutines just like ones you have written, but you must determine what parameters they require, which registers they use, and where they put their results. The monitor subroutines are not necessarily either general or useful.
7. A program can choose among subroutines by calling a routine that obtains the proper starting address and stores it in memory. Either JMP with indirect addressing or RTS can then transfer control to the subroutine. RTS at the end of the subroutine will return control to the original calling point. JSR itself allows only absolute (direct) addressing.

LABORATORY B

INPUT/OUTPUT USING HANDSHAKES

PURPOSE

To learn how to perform input and output using handshake status and control signals.

PARTS REQUIRED

- Two switches attached to CA1 and CB1 of the user VIA as shown in Figure B–1. Table B–1 gives the pin assignments. These switches should be debounced with cross-coupled NAND gates.
- Two LEDs and two switches attached to CA2 and CB2 of the user VIA as shown in Figures B–2 and B–3. Table B–1 gives the pin assignments. The switches should be debounced with cross-coupled NAND gates. The LEDs should be attached by their cathodes. Jumper wires can select between the LEDs and the switches to avoid damage to the gates.

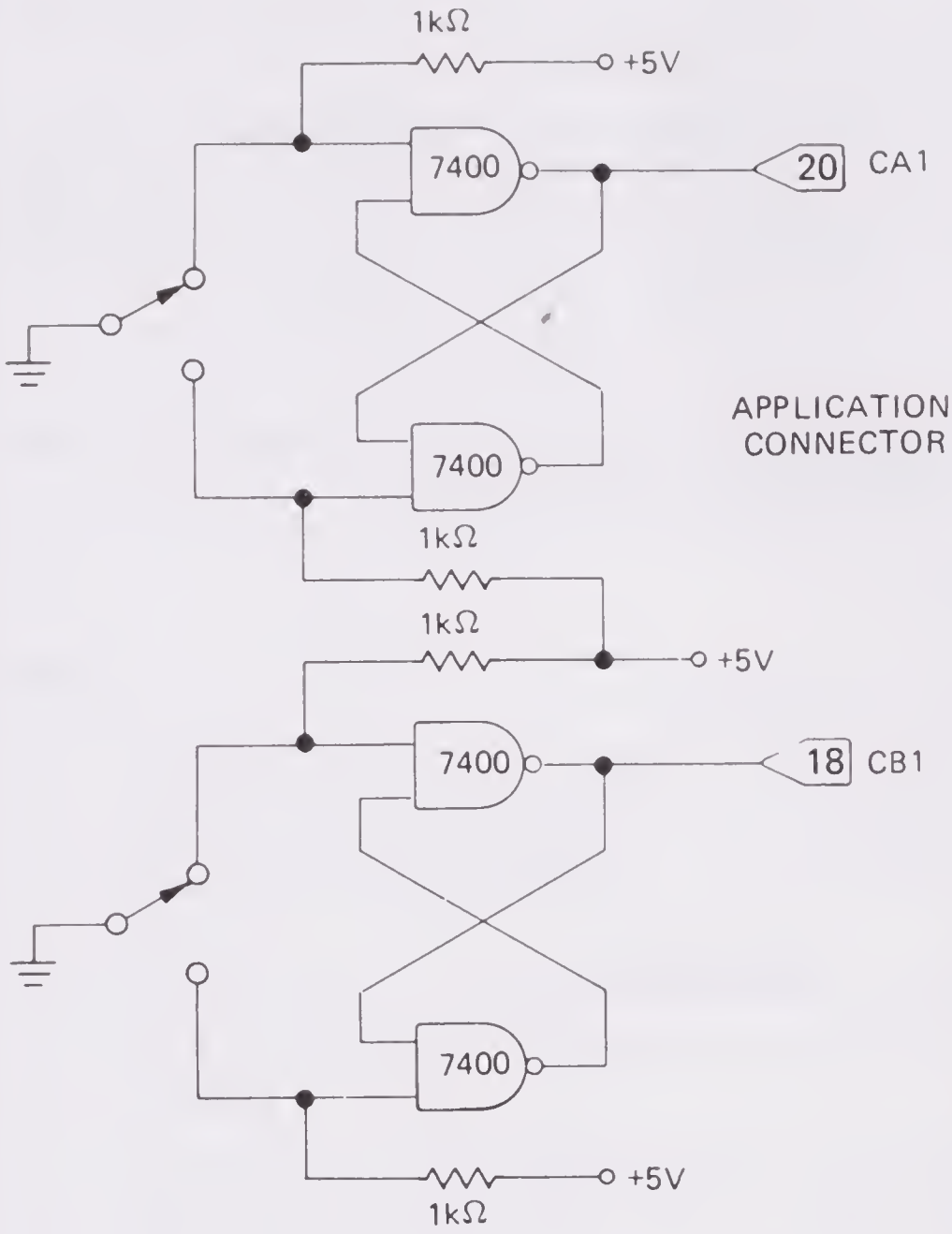


FIGURE B–1. Attachment of switches to user VIA control lines CA1 and CB1.

TABLE B–1 APPLICATION CONNECTOR PIN ASSIGNMENTS FOR USER VIA CONTROL LINES

Assignment	Pin
CA1 (Control line 1, port A)	20
CA2 (Control line 2, port A)	21
CB1 (Control line 1, port B)	18
CB2 (Control line 2, port B)	19

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 337–369, 405–427.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 11.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, 1982, Chapter 10.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 36–43 (flip-flops), 43–46 (counters), 46–53 (registers), 119–120 (I/O), 196–197 (I/O terms), 197–199 (I/O examples), 199–200 (I/O methods), 206–212 (polled I/O), 259–265 (6520 PIA).

AIM 65 User's Guide, Dynatem, Irvine, CA, 1979, pp. 7–31, 8–1 to 8–25.

R6500 Microcomputer System Hardware Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, Section 6.

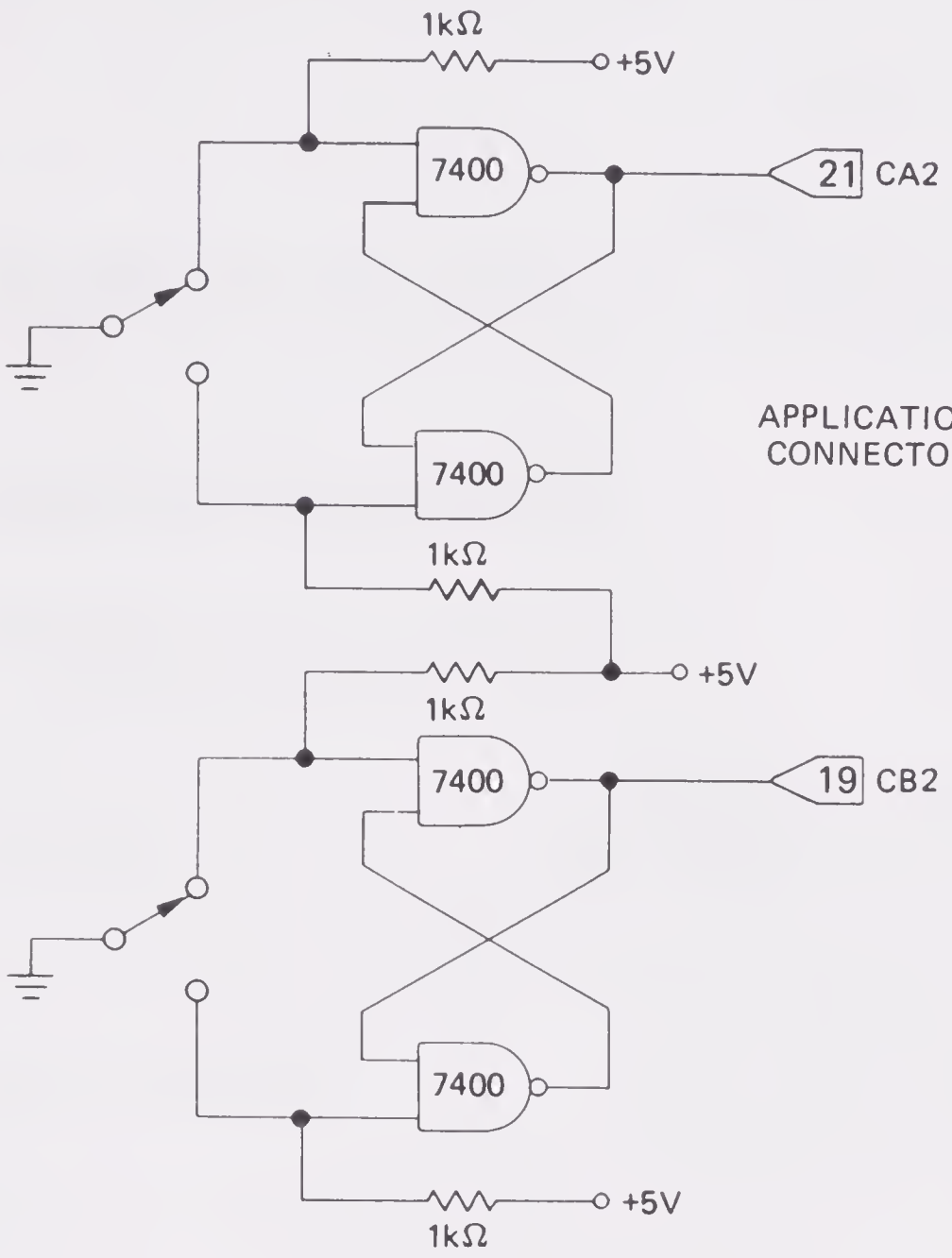


FIGURE B–2. Attachment of switches to user VIA control lines CA2 and CB2.

WHAT YOU SHOULD LEARN

1. How to perform synchronous and asynchronous I/O.
2. How to handle handshake status inputs and control outputs in software.
3. The features of a 6522 Versatile Interface Adapter (VIA).

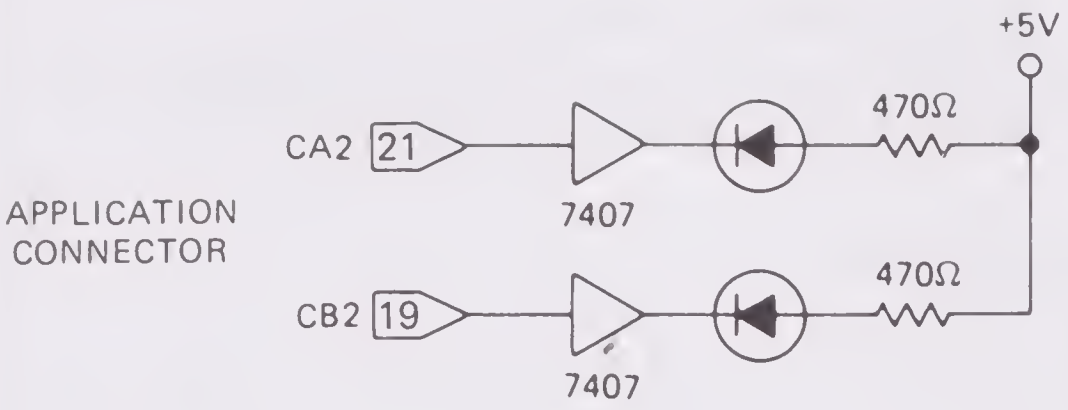


FIGURE B–3. Attachment of LEDs to user VIA control lines CA2 and CB2.

4. How to determine a VIA's operating mode.
5. How to implement handshaking with a VIA.

TERMS

Asynchronous operating without a clock, that is, at irregular intervals.

Clock a regular series of pulses that control transitions in a system.

Control (or command) register a register whose contents determine how a device operates.

Control signal a signal that directs an I/O transfer.

Data accepted a signal indicating whether the latest data has been transferred successfully .

Data ready a signal indicating whether new data is available; same as *valid data*.

Handshake the exchange of signals by sender and receiver to establish synchronization and manage a data transfer.

Interrupt flag in a VIA, a bit that indicates the occurrence of an event such as an active transition on a control line.

Interrupt request (IRQ) a signal indicating that a peripheral is requesting service.

Latch a storage device that retains its current contents until they are explicitly changed.

Peripheral port in a VIA, the actual input or output port.

Peripheral ready a signal indicating whether a peripheral can accept more data.

Polling determining which I/O devices are ready by examining the status of one at a time.

Programmable I/O device an I/O device that has its operating mode determined by a program.

Ready for data a signal indicating whether the receiver can accept more data.

Status register a register whose contents indicate the state of a transfer or the operating mode of a device.

Status signal a signal that describes another set of signals and can be used to control a buffer or latch.

Synchronous operating according to a clock, that is, at regular intervals.

Valid data a signal indicating that new data is available.

Versatile Interface Adapter (6522 VIA) a device consisting of two 8-bit bidirectional I/O ports, four status and control lines, two timers, and a shift register.

SYNCHRONOUS AND ASYNCHRONOUS I/O

So far we have dealt only with simple, low-speed I/O devices such as switches and displays. Our only problems have been ignoring meaningless changes in inputs and making outputs last long enough to satisfy a peripheral or an observer. Factors that we have not considered include:

1. Whether the peripheral is ready.
2. Whether new data is available.
3. Whether the data has been transferred successfully.

These matters become important for medium-speed peripherals such as terminals, printers, modems, plotters, and card readers.

To transfer data successfully, the following conditions must hold:

1. The receiver must be ready.
2. The data must be available (or *valid*).
3. The receiver must accept the data before it changes.

So the sender must know whether the receiver is ready and whether it has accepted the data. The receiver must know whether new data is available.

One approach is to use a clock (i.e., a regular series of pulses) as a reference. The receiver must then be ready, and the data must be available and accepted at particular points in the clock cycle (e.g., 100 ns after the rising edge of a pulse). This method, called *synchronous transfer*, requires synchronization (i.e., alignment) of the receiver and the transmitter with the clock.

The major problem with this approach is its rigidity. The only way to change the data rate is by changing the clock. Thus synchronous transfer cannot easily handle peripherals that operate at varying data rates or provide data irregularly.

An alternative is to use status and control signals to manage the transfer. Typical signals are

READY FOR DATA active when the receiver can accept more data.
VALID DATA active when new data is available.
DATA ACCEPTED active when the receiver has accepted the latest data.

The sender must provide VALID DATA; the receiver must provide READY FOR DATA and DATA ACCEPTED. This method, called *asynchronous transfer*, requires no clock and allows any data rate.

The advantages of this approach are flexibility (since the devices determine the timing) and simplicity (no clock or synchronization is necessary). The disadvantages are the increased number of signals and reduced maximum data rates (since signals must overlap properly).

The terms *polling* and *handshaking* are commonly used to describe asynchronous transfers. *Polling* is examining each peripheral's status to determine whether it is ready for a data transfer. *Handshaking* is exchanging status and control signals to manage a data transfer; the handshake validates the transfer much as a human handshake validates a contract.

TREATING STATUS AND CONTROL SIGNALS AS DATA

One way to handle status and control signals is to treat them as data. They then act like the binary inputs and outputs considered in Laboratories 2 and 3. In our examples, we will use bit 7 of port A of the user VIA as a status input and bit 7 of port B as a control output.

The following program (Program B–1 is the mnemonic-entry version) assigns directions to the user VIA's ports and turns the LEDs off. We will use it for initialization throughout this laboratory.

```
LDA  #0                    ; MAKE PORT A INPUT
STA  $A003
LDA  #$FF
STA  $A002                 ; MAKE PORT B OUTPUT
STA  $A000                 ; TURN OFF THE LEDS
```

PROGRAM B–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A9	LDA #00
0201	00	
0202	8D	STA A003
0203	03	
0204	A0	
0205	A9	LDA #FF
0206	FF	
0207	8D	STA A002
0208	02	
0209	A0	
020A	8D	STA A000
020B	00	
020C	A0	

USING DATA LINES FOR STATUS

Let us now use bit 7 of port A for status. During input, this signal typically indicates whether new data is available (e.g., whether an operator has pressed a key on a keyboard). The following program waits for the switch attached to bit 7 (switch 7) to close. It then reads the data from port A and shows it on the LEDs at port B.

```
WAITR   LDA  $A001     ; IS DATA READY?
        BMI  WAITR     ; NO, WAIT
        EOR  #$FF      ; YES, ACCEPT DATA
        STA  $A000     ; AND SHOW IT ON THE LEDS
        BRK
```

Program B–2 is the mnemonic-entry version. Open switch 7 and run Program B–2. What happens? Does opening or closing other switches have any effect? Open all other switches and then close switch 7. What happens? The processor accepts only the data that is present when the status signal becomes active. Changes that occur while the status signal is inactive are ignored.

PROGRAM B–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	AD	WAITR	LDA A001
020E	01		
020F	A0		
0210	30		BMI 020D
0211	FB		
0212	49		EOR #FF
0213	FF		
0214	8D		STA A000
0215	00		
0216	A0		
0217	00		BRK

PROBLEM B–1

Change Program B–2 to make the status signal active-high. How would you make the program wait for the status signal to go low and then back high? Remember to debounce the switch.

PROBLEM B–2

Extend Program B–2 to read data into an array starting at 0340. It should read an item each time the status signal goes low. Remember to debounce the switch.

During output, the status signal indicates whether the peripheral is ready for more data (e.g., whether a printer has finished with the last character). The following program waits for switch 7 to close before sending data from 0340. (Program B–3 is the mnemonic-entry version.)

```
WAITR   LDA  $A001     ; IS PERIPHERAL READY?
        BMI  WAITR     ; NO, WAIT
        LDA  $0340     ; YES, SEND DATA
        EOR  #$FF      ; WITH REVERSED POLARITY
        STA  $A000
        BRK
```

PROGRAM B–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	AD	WAITR	LDA A001
020E	01		
020F	A0		
0210	30		BMI 020D
0211	FB		
0212	AD		LDA 0340
0213	40		
0214	03		
0215	49		EOR #FF
0216	FF		
0217	8D		STA A000
0218	00		
0219	A0		
021A	00		BRK

Enter Program B–3 into memory and run it with (0340) = FF. What happens before you close switch 7? Does it matter what 0340 contains? What happens when you close switch 7? The old data persists until the peripheral specifically requests new data or informs the computer that it is ready.

PROBLEM B–3

Make Program B–3 send data from an array starting at 0340. It should send a new item each time the status signal goes low. Remember to debounce the switch.

Example array (single light moves left, starting from bit 7):

(0340) = 80
(0341) = 40
(0342) = 20

(0343) = 10
(0344) = 08
(0345) = 04
(0346) = 02
(0347) = 01

PROBLEM B–4

Make your answer to Problem B–3 exit after sending eight items. How would you make it stop afer it sends a zero value?

USING DATA LINES FOR CONTROL

We can also use data lines as control signals. A control signal for an input device could indicate that the computer has read the latest data. The following program loads 0340 with the data from port A and then lights the control LED by clearing bit 7 of port B.

```
LDA   $A001          ; GET DATA FROM INPUT PORT
STA   $0340          ; SAVE DATA IN MEMORY
LDA   #%01111111     ; TURN CONTROL LIGHT ON
STA   $A000
BRK
```

PROGRAM B–4

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
020D	AD	LDA A001
020E	01	
020F	A0	
0210	8D	STA 0340
0211	40	
0212	03	
0213	A9	LDA #7F
0214	7F	
0215	8D	STA A000
0216	00	
0217	A0	
0218	00	BRK

Enter and run Program B–4, the mnemonic-entry version. Here the light indicates that the computer has accepted the latest data and is ready for more. Such a signal may be called READY FOR DATA, DATA ACCEPTED, or DATA BUFFER EMPTY.

Combining Programs B–2 and B–4 (see Program B–5) makes the computer wait for the status signal to become active before accepting the data. It then sets the control signal to mark the acceptance. Here we have a complete handshake (see Figure B–4); the sender indicates that new data is available, and the receiver, in response, reads the data and acknowledges it.

```
WAITR   LDA  $A001          ; IS DATA READY?
        BMI  WAITR          ; NO, WAIT
        STA  $0340          ; YES, ACCEPT DATA
        LDA  #%01111111     ; INDICATE DATA ACCEPTED
        STA  $A000
        BRK
```

PROGRAM B–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	AD	WAITR	LDA A001
020E	01		
020F	A0		
0210	30		BMI 020D
0211	FB		
0212	8D		STA 0340
0213	40		
0214	03		
0215	A9		LDA #7F
0216	7F		
0217	8D		STA A000
0218	00		
0219	A0		
021A	00		BRK

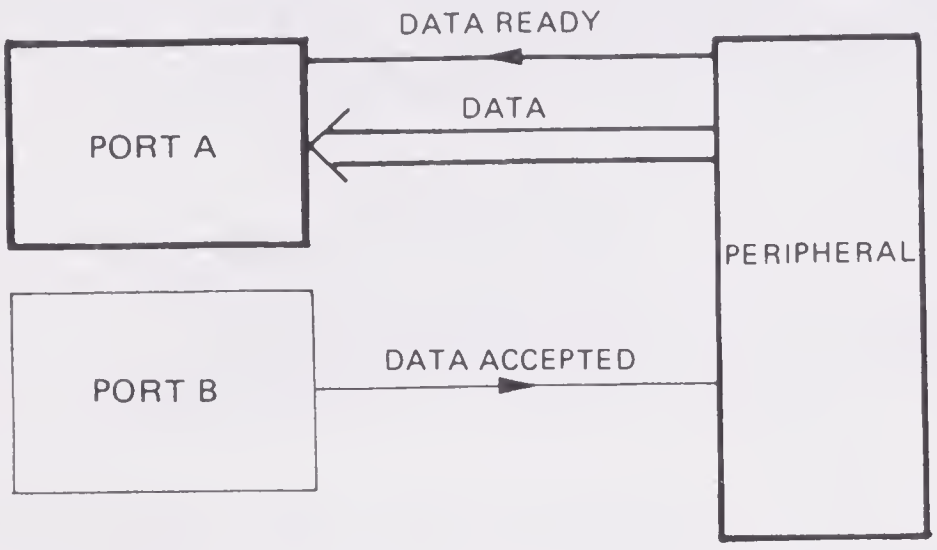
Enter and run Program B–5. An obvious problem is that the light stays on. Clearly, it should go off before the next transfer. The control light may, for example,

1. Remain active only briefly, thus producing a pulse that can be counted or latched.
2. Go off when the status signal becomes active again to begin the next transfer. The control signal then indicates whether the processor has accepted the latest data (i.e., it is a BUFFER EMPTY signal).
3. Remain active for an amount of time determined by the program.

As we shall see later, the 6522 VIA contains circuits for all these alternatives. The user simply selects an operating mode by storing a value in a VIA control register.

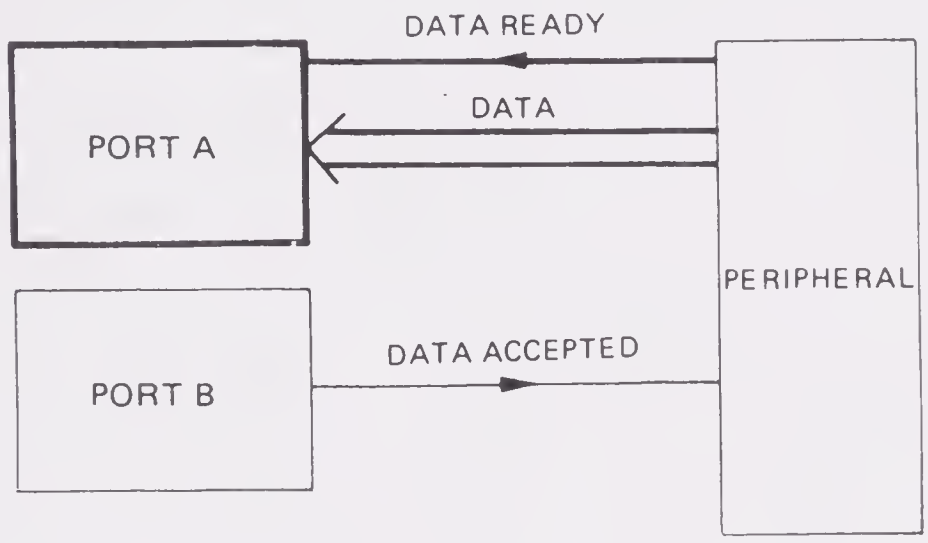
Combining Programs B–3 and B–4 (see Program B–6) makes the computer wait for the status signal to become active before sending the data. It then sets the control signal to

STEP 1
PERIPHERAL PROVIDES DATA AND ACTIVATES DATA READY.

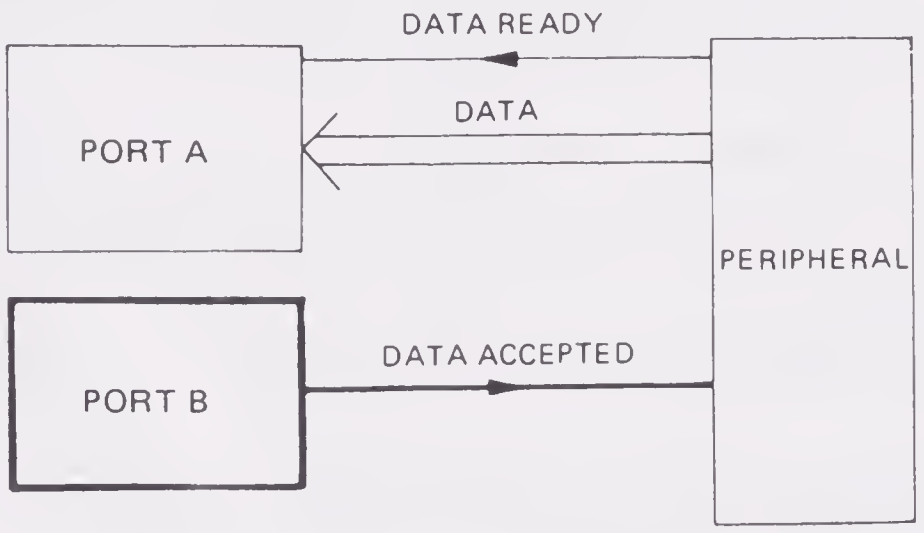


The peripheral provides both the data and an active DATA READY signal.

STEP 2
CPU RECOGNIZES THAT DATA READY IS ACTIVE AND READS THE DATA, THUS PERFORMING THE ACTUAL DATA TRANSFER.



STEP 3
CPU ACTIVATES DATA ACCEPTED, INDICATING THE SUCCESSFUL COMPLETION OF THE TRANSFER.



The peripheral can examine DATA ACCEPTED to determine when it can send more data.

FIGURE B–4. Procedure for a complete input handshake.

mark the transmission. Here again we have a complete handshake (see Figure B–5), although the order and meaning of the signals differ from the input case. The receiver indicates that it is ready to accept data; in response, the sender provides the data and indicates its availability.

```
WAITR   LDA  $A001          ; IS PERIPHERAL READY?
        BMI  WAITR          ; NO, WAIT
        LDA  $0340          ; YES, SEND DATA
        EOR  #$FF
        AND  #%01111111     ; INDICATE DATA AVAILABLE
        STA  $A000
        BRK
```

PROGRAM B–6

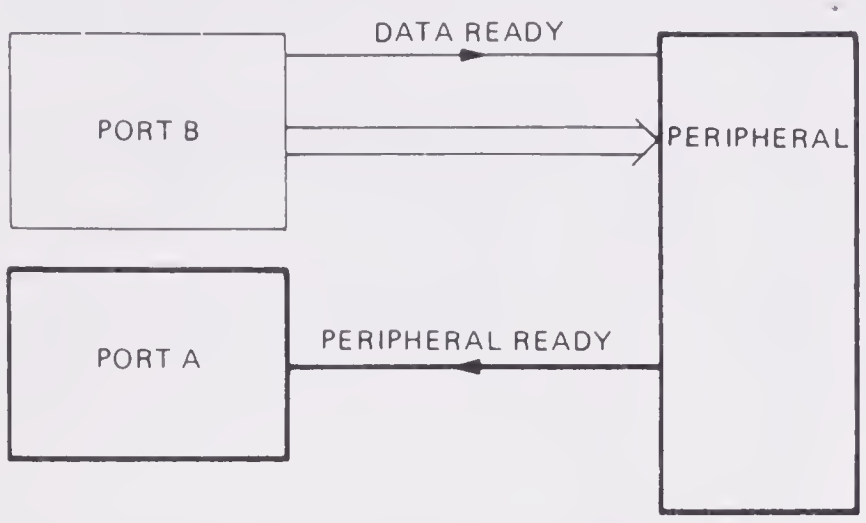
Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	AD	WAITR	LDA A001
020E	01		
020F	A0		
0210	30		BMI 020D
0211	FB		
0212	AD		LDA 0340
0213	40		
0214	03		
0215	49		EOR #FF
0216	FF		
0217	29		AND #7F
0218	7F		
0219	8D		STA A000
021A	00		
021B	A0		
021C	00		BRK

6522 VERSATILE INTERFACE ADAPTER

LSI I/O devices can greatly simplify interfacing and I/O programming. Furthermore, they are smaller, cheaper, more reliable, and use less power than circuits made from TTL parts.

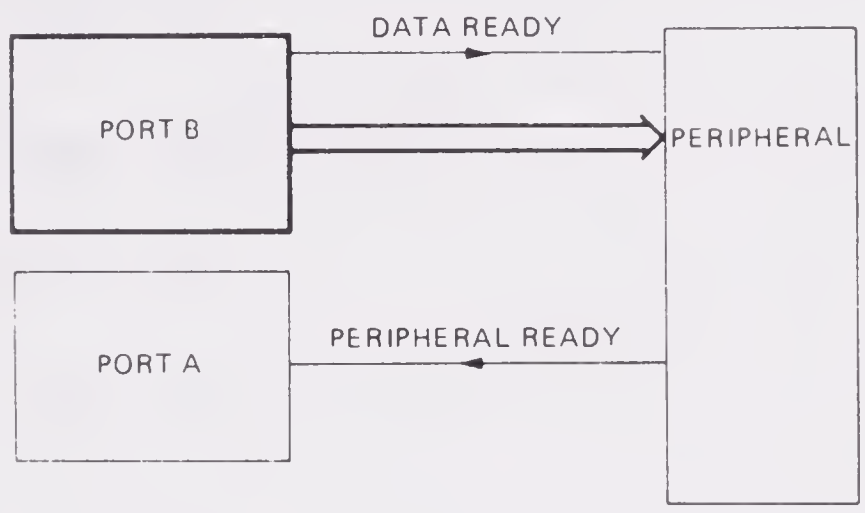
As an example, consider the 6522 Versatile Interface Adapter (VIA). This generalized I/O device can operate in many useful ways; the programmer selects the operating mode for each port by storing data in control registers. This data activates circuits in the VIA, much as an instruction does in the CPU. The circuits in the VIA, however, are much simpler than those in the CPU and are designed to perform common I/O functions such as handshaking.

STEP 1
PERIPHERAL ACTIVATES PERIPHERAL READY, INDICATING THAT IT IS ABLE TO ACCEPT DATA.

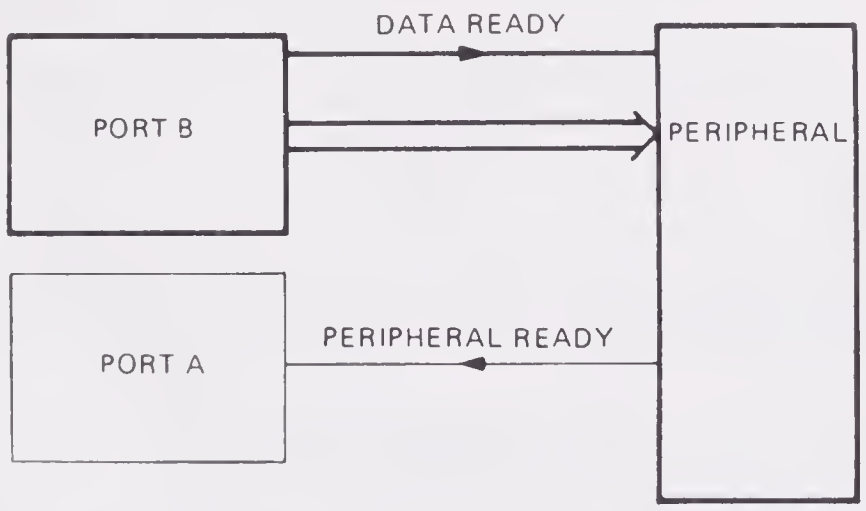


The output peripheral must provide the input status signal PERIPHERAL READY.

STEP 2
CPU RECOGNIZES THAT PERIPHERAL READY IS ACTIVE AND SENDS THE DATA, THUS PERFORMING THE ACTUAL DATA TRANSFER.



STEP 3
CPU ACTIVATES DATA READY, THUS INFORMING THE PERIPHERAL THAT NEW DATA IS AVAILABLE.



The peripheral can examine DATA READY to determine when new data is available.

FIGURE B–5. Procedure for a complete output handshake.

Each VIA port contains:

- An input register, used to hold data read from an input device.
- An output register (or latch), used to hold data sent to an output device.
- A data direction register that determines whether the I/O lines are inputs or outputs. This register is inside the VIA and is not connected to peripherals.
- Two control lines that can be used for status and control signals as governed by the control registers. These lines, like the data lines, are connected to peripherals.

The peripheral control register (Figure B–6) and the auxiliary control register determine how a VIA operates. We will discuss the peripheral control register here and the auxiliary control register in Laboratory D.

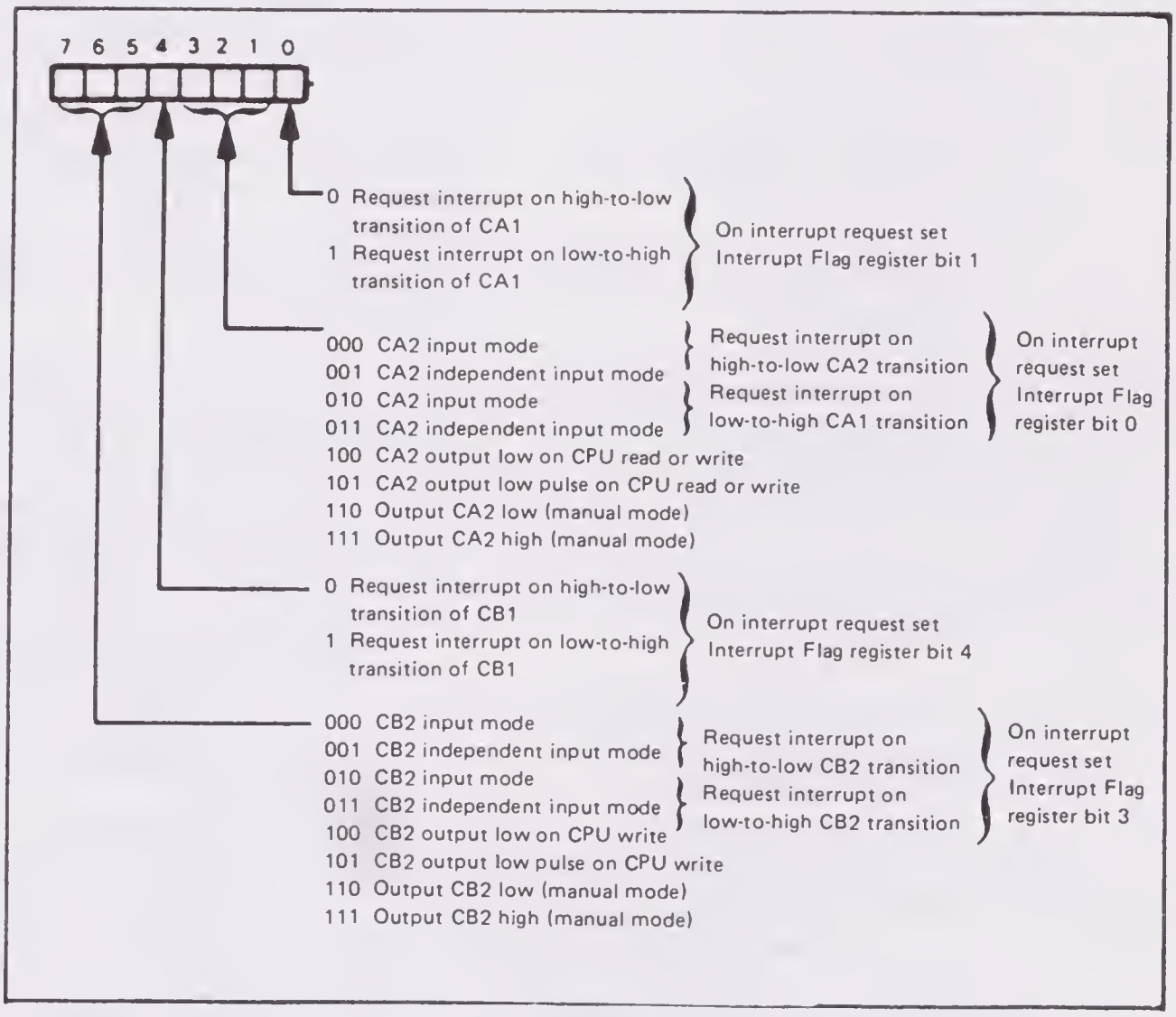


FIGURE B–6. The 6522 VIA's peripheral control register (PCR). (Reprinted from *Osborne 4- and 8-Bit Microprocessor Handbook* by permission of Osborne/McGraw-Hill.)

VIA STATUS INPUTS

The VIA's status inputs work as follows:

- Bits 0, 1, 3, and 4 of the interrupt flag register (Figure B–7) are set to 1 whenever an active transition occurs on control line CA2 (bit 0), CA1 (bit 1), CB2 (bit 3), or CB1 (bit 4).

7	6	5	4	3	2	1	0
IRQ	T1	T2	CB1	CB2	SR	CA1	CA2

Bit No.	Set By	Cleared By
0	Active transition of the signal on the CA2 pin	Reading or writing the A Port Output register (ORA) using address 0001
1	Active transition of the signal on the CA1 pin	Reading or writing the A Port Output register (ORA) using address 0001
2	Completion of eight shifts	Reading or writing the Shift register
3	Active transition of the signal on the CB2 pin	Reading or writing the B Port Output register
4	Active transition of the signal on the CB1 pin	Reading or writing the B Port Output register
5	Timeout of timer 2	Reading T2 low-order counter or writing T2 high-order counter
6	Timeout of timer 1	Reading T1 low-order counter or writing T1 high-order latch
7	Any active and enabled interrupt condition	Action which clears interrupt condition

Bits 0, 1, 3, and 4 are the I/O handshake signals. Bit 7 (IRQ) is 1 if any of the interrupts is both active and enabled.

FIGURE B–7. The 6522 VIA's interrupt flag register (IFR). (Reprinted from *Osborne 4- and 8-Bit Microprocessor Handbook* by permission of Osborne/McGraw-Hill.)

- Bits in the peripheral control register (Figure B–6) determine whether the active transition is positive (a change from 0 to 1) or negative (1 to 0). Bit 0 governs CA1; bit 2, CA2; bit 4, CB1; and bit 6, CB2. A 0 value makes negative edges active, while a 1 makes positive edges active.

Thus we can provide a status bit without using a data line. Furthermore, the VIA latches (holds) the status bit, even if the input signal is only a brief pulse. All VIA status bits are readily available for testing in the interrupt flag register.

The VIA also clears the status automatically when the I/O port is read or written. That is, reading data from the I/O port or storing data into it clears the corresponding interrupt flags (bits 0 and 1 for port A, bits 3 and 4 for port B). No additional hardware or software is necessary.

TABLE B–2 REGISTER ADDRESSES IN THE USER VIA

Address In User VIA	Register Designation	Function
A000	ORB/IRB	Input/output register B
A001	ORA/IRA	I/O register A (handshake)
A002	DDRB	Data direction register B
A003	DDRA	Data direction register A
A00C	PCR	Peripheral control register
A00D	IFR	Interrupt flag register

The following program (see Program B–7) waits until the switch attached to CA1 (the CA1 switch, for short) closes. It then reads the data from port A and displays it on the LEDs. Table B–2 contains the addresses for the user VIA's peripheral control and interrupt flag registers. Note that a high-to-low transition on CA1 (i.e., closing the CA1 switch) sets bit 1 of the interrupt flag register. We must therefore test for a 1 even though the active state on CA1 is 0.

```
        LDA  #0                 ; ACTIVE TRANSITION IS NEGATIVE
        STA  $A00C
        LDA  #%00000010         ; MASK FOR CA1 INTERRUPT FLAG
WAITR   BIT  $A00D              ; IS DATA READY?
        BEQ  WAITR              ; NO, WAIT
        LDA  $A001              ; YES, FETCH DATA
        EOR  #$FF               ; AND SHOW IT ON THE LEDS
        STA  $A000
        BRK
```

PROGRAM B–7

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	A9		LDA #00
020E	00		
020F	8D		STA A00C
0210	0C		
0211	A0		
0212	A9		LDA #02
0213	02		
0214	2C	WAITR	BIT A00D
0215	0D		
0216	A0		
0217	F0		BEQ 0214
0218	FB		
0219	AD		LDA A001
021A	01		
021B	A0		
021C	49		EOR #FF
021D	FF		
021E	8D		STA A000
021F	00		
0220	A0		
0221	00		BRK

Enter and run Program B-7. What is the final value in A00D? Explain what happened. Remember that reading port A (address A001) automatically clears bit 1 of the interrupt flag register. Show that this actually happens by executing the program in the STEP mode and examining A00D before and after LDA $A001. Does it matter if you replace LDA $A001 with STA $A001?

Sometimes the processor may exit from Program B–7 before you close the CA1 switch. The usual reason is that the interrupt flag was already set when the program started. To keep this from happening, examine A001 (port A) or press RESET before executing the program.

PROBLEM B–5

Any instruction that reads or writes port A clears the CA1 interrupt flag. Try the following: ASL $A001, LSR $A000, STA $A001, STA $A000, CMP $A001, INC $A001, LDA $A003. How does each affect the interrupt flag? Note that reading or writing the data direction register or port B does not do the job.

PROBLEM B–6

Make Program B–7 respond to the opening of the CA1 switch. Remember that the switch is debounced.

PROBLEM B–7

Make Program B–7 load data into an array starting at 0340. The revised program should load an item each time you close the CA1 switch.

You can use CA2 just like CA1. Now active transactions set bit 0 of the interrupt flag register instead of bit 1, and the edge control is bit 2 of the peripheral control register instead of bit 0.

PROBLEM B–8

Make Program B–7 respond to the opening of the CA2 switch.

Similarly, we can use CB1 or CB2 as a PERIPHERAL READY input. The revised version of Program B–3 using CB1 is as follows (see Program B–8 for a mnemonic-entry version).

```
          LDA   #0                ; MAKE ACTIVE TRANSITION NEGATIVE
          STA   $A00C
          LDA   #%00010000        ; GET MASK FOR CB1 INTERRUPT FLAG
WAITR     BIT   $A00D             ; IS PERIPHERAL READY?
          BEQ   WAITR             ; NO, WAIT
          LDA   $0340             ; YES, SEND DATA
          EOR   #$FF              ; WITH REVERSED POLARITY
          STA   $A000
          BRK
```

PROBLEM B–9

Make Program B–8 send data from an array starting at 0340. The program should send an item each time the status signal goes low. How would you make it respond to the status signal going high? How would you make it respond to CB2 instead of CB1?

PROBLEM B–10

Make your answer to Problem B–9 respond to CA2 instead of CB1. Be careful; you must send the data to port A, but you must also clear the interrupt flag by reading or writing port B. Note that you can use BIT or CMP to read port B without changing the accumulator. You can use control lines from other ports as long as you manage the status and control signals properly.

PROGRAM B–8

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	A9		LDA #00
020E	00		
020F	8D		STA A00C
0210	0C		
0211	A0		
0212	A9		LDA #10
0213	10		
0214	2C	WAITR	BIT A00D
0215	0D		
0216	A0		
0217	F0		BEQ 0214
0218	FB		
0219	AD		LDA 0340
021A	40		
021B	03		
021C	49		EOR #FF
021D	FF		
021E	8D		STA A000
021F	00		
0220	A0		
0221	00		BRK

Remember that a high-to-low transition on CB1 (i.e., closing the CB1 switch) sets bit 4 of the interrupt flag register. Enter Program B–8 into memory and run it with (0340) = FF.

VIA CONTROL OUTPUTS

So far, we have used VIA control lines as DATA READY or PERIPHERAL READY signals. This approaeh does not use any data lines. Furthermore, the status is latched, and the interrupt flags are cleared automatically.

We can, however, do even more with the VIA's control lines. We can also use CA2 or CB2 as an output control signal. The peripheral eontrol register bits (see Figure B–6) required do this are:

1. Bit 3 determines whether CA2 is input (0) or output (1). Bit 7 does the same for CB2. Note that we have cleared these bits in the previous examples.
2. If CA2 is output, bit 2 determines whether it is pulsed automatically after input or output (0) or left at a fixed level (1). Bit 6 does the same for CB2. These alternatives

are called the *automatic mode* and the *manual mode*, respectively. In the automatic mode, control line 2 is normally 1 and becomes 0 during a pulse.

3. If CA2 is in the automatic mode, bit 1 determines whether the pulse lasts one clock cycle (1) or until the next active transition on CA1 (0). Bit 5 does the same for CB2. The first alternative can produce a brief BYTE OUT pulse for multiplexing displays. The second alternative can produce an acknowledgment to an intelligent peripheral such as a terminal with its own microprocessor. The acknowledgment indicates that the computer has accepted the latest input data or has produced new output data that the peripheral has not yet accepted.
4. If CA2 (CB2) is in the manual mode, bit 1 (bit 5) is its value. The program can then change control line 2 by setting or clearing a bit in the peripheral control register.

Let us now revise Program B–4 to use CA2 in the manual mode as the data accepted signal. We must load the peripheral control register with 00001110 binary (0E hex), where

bit 3 = 1 to make CA2 an output
bit 2 = 1 to operate CA2 manually
bit 1 = 1 to make CA2 initially 1 (thus turning the LED off)

The revised program is as follows (see Program B–9 for a mnemonic-entry version):

```
LDA  #%00001110          ; TURN CONTROL LIGHT OFF
STA  $A00C
LDA  $A001               ; GET DATA FROM INPUT PORT
EOR  #$FF
STA  $A000               ; SHOW DATA ON LEDS
LDA  #%00001100          ; TURN CONTROL LIGHT ON
STA  $A00C
BRK
```

All the LEDs are available for data, since CA2 acts as the control signal. Enter and run Program B–9; use the STEP mode to see the data appear on the LEDs and the control light go off and on.

PROGRAM B–9

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
020D	A9	LDA #0E
020E	0E	
020F	8D	STA A00C
0210	0C	
0211	A0	
0212	AD	LDA A001

PROGRAM B–9 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0213	01	
0214	A0	
0215	49	EOR #FF
0216	FF	
0217	8D	STA A000
0218	00	
0219	A0	
021A	A9	LDA #0C
021B	0C	
021C	8D	STA A00C
021D	0C	
021E	A0	
021F	00	BRK

We can also revise Program B–5 (handshake input) to use CA2. Here CA1 indicates whether new data is available, and CA2 indicates whether the data has been accepted (see Figure B–4). Program B–10 is the mnemonic-entry version.

```
                LDA   #%00001110        ; TURN CONTROL LIGHT OFF
                STA   $A00C
                LDA   #%00000010        ; GET MASK FOR CA1 INTERRUPT FLAG
WAITR           BIT   $A00D             ; IS DATA READY?
                BEQ   WAITR             ; NO, WAIT
                LDA   $A001             ; YES, GET DATA FROM INPUT PORT
                EOR   #$FF              ; SHOW DATA ON LEDS
                STA   $A000
                LDA   #%00001100        ; TURN CONTROL LIGHT ON
                STA   $A00C
                BRK
```

One problem with the manual mode is avoiding changes in other bits in the peripheral control register. We can use logical functions to manipulate a single bit as follows:

1. To make control line 2 a 1, logically OR bit 1 (for CA2) or bit 5 (for CB2) with 1.

```
LDA   $A00C
ORA   #%00000010          ; BRING CA2 HIGH
STA   $A00C
```

PROGRAM B–10

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	A9		LDA #0E
020E	0E		
020F	8D		STA A00C
0210	0C		
0211	A0		
0212	A9		LDA #02
0213	02		
0214	2C	WAITR	BIT A00D
0215	0D		
0216	A0		
0217	F0		BEQ 0214
0218	FB		
0219	AD		LDA A000
021A	00		
021B	A0		
021C	49		EOR #FF
021D	FF		
021E	8D		STA A000
021F	00		
0220	A0		
0221	A9		LDA #0C
0222	0C		
0223	8D		STA A00C
0224	0C		
0225	A0		
0226	00		BRK

```
LDA   $A00C
ORA   #%00100000          ; BRING CB2 HIGH
STA   $A00C
```

2. To make control line 2 a 0, logically AND bit 1 (for CA2) or bit 5 (for CB2) with 0.

```
LDA   $A00C
AND   #%11111101          ; BRING CA2 LOW
STA   $A00C

LDA   $A00C
AND   #%11011111          ; BRING CB2 LOW
STA   $A00C
```

PROBLEM B–11

Write a program that uses logical functions to bring CB2 high, low, and then high again.

VIA AUTOMATIC CONTROL MODES

We can simplify Programs B–9 and B–10 further by using the automatic mode, in which the VIA generates a pulse on control line 2. For example, let us store the binary value 00001000 (08 hex) in the peripheral control register, where

bit 3 = 1 to make CA2 an output
bit 2 = 0 to generate a pulse on CA2
bit 1 = 0 to make CA2 go low after the I/O port is read and remain low until the next active transition on CA1 (CA2 is thus an INPUT BUFFER EMPTY signal)

The following program works the same as Program B–8; the processor waits for an active transition on CA1, reads the data, and then brings CA2 low to indicate that the data has been accepted. Program B–11 is the mnemonic-entry version.

```
        LDA  #%00001000        ; PUT CA2 IN AUTOMATIC MODE
        STA  $A00C
        LDA  #%00000010        ; GET CA1 INTERRUPT MASK
WAITR   BIT  $A00D             ; IS DATA READY?
        BEQ  WAITR             ; NO, WAIT
        LDA  $A001             ; YES, GET DATA FROM INPUT PORT
        EOR  #$FF              ; SHOW DATA ON LEDS
        STA  $A000
        BRK
```

Enter Program B–11 into memory and run it in the STEP mode. The LED attached to CA2 (the CA2 LED, for short) should be off until the processor executes LDA $A001. Then it should light and remain lit until you open and close the CA1 switch, thus producing the next active transition. Note that we need not change the peripheral control register after initializing it; the VIA sends the control line low and high automatically.

Obviously, the automatic mode requires less programming than the manual mode. However, the manual mode gives the programmer complete control over the pulse's length and polarity. As you might expect, the automatic mode often does not fit a particular situation; in such cases the manual mode is handy.

PROBLEM B–12

Make Program B–6 use CB2 in the automatic mode as the control output.

PROGRAM B–11

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
020D	A9		LDA #08
020E	08		
020F	8D		STA A00C
0210	0C		
0211	A0		
0212	A9		LDA #02
0213	02		
0214	2C	WAITR	BIT A00D
0215	0D		
0216	A0		
0217	F0		BEQ 0214
0218	FB		
0219	AD		LDA A001
021A	01		
021B	A0		
021C	49		EOR #FF
021D	FF		
021E	8D		STA A000
021F	00		
0220	A0		
0221	00		BRK

PROBLEM B–13

Make the answer to Problem B–12 send data from an array starting at 0340. The program should send the next item each time the status input (CB1) goes low. Single-step through the program to see the control light go on and off.

The other automatic mode (peripheral control register bit 1 or bit 5 = 1) generates a brief pulse lasting one CPU clock cycle. Besides multiplexing displays, this mode can clock I/O devices such as A/D and D/A converters. Change location 020E to 0A and run Program B–11 again. You will not be able to see the control light come on because the pulse is too brief. To verify that the pulse occurred, tie CA2 to CB1 and test bit 4 of the interrupt flag register after running the program.

We should note that port A and port B have different automatic modes. Port A produces automatic pulses after the I/O port is either read or written, whereas port B produces automatic pulses only after the I/O port is written. Of course, we can always fool the VIA by including an unnecessary store instruction in an input program:

```
LDA   VIAORB        ; GET DATA FROM PORT B
STA   VIAORB        ; PRODUCE AUTOMATIC PULSE
```

PROGRAMMABLE I/O DEVICES

The VIA has many operating modes (see Figures B–6 and B–7). A program can put it in a specific mode by storing appropriate values in its control registers. The advantages of such a device are that you can use the same hardware in many different applications and you can make changes or corrections in software. The disadvantages are the extra programming required and the lack of standards. The device's manufacturer determines arbitrarily what modes are available and how they are selected. For example, the functions and positions of the bits in the VIA's control registers are arbitrary; similar devices from other manufacturers would have completely different registers.

However, the following features are typical of all programmable I/O devices:

1. Control or command registers that determine how the device operates.
2. Status registers that describe the current state of the device and the data transfer. The VIA's interrupt flag register serves as its status register.
3. Separate data and status or control inputs and outputs.

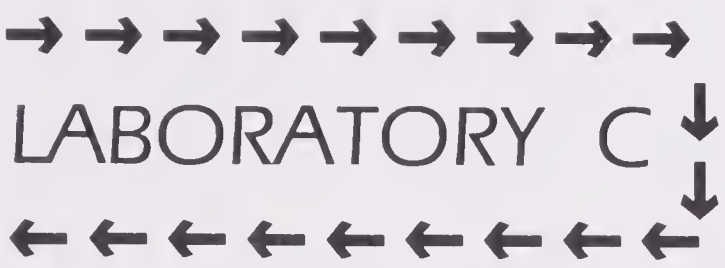
Most or all bits in the control registers are set during initialization to implement a particular interface. The main program does not change them. When using the VIA, for example, most applications programs do not change the arrangement of input and output lines or the operating mode.

Programmable I/O devices require careful documentation. The instructions that determine their operating modes and use them are arbitrary and are seldom described well in manuals.

KEY POINT SUMMARY

1. Input and output can proceed properly only if there is a way to determine when the receiver is ready, when new data is available, and when the receiver has accepted the data.
2. Synchronous transfers use a clock, whereas asynchronous transfers require a handshake in which sender and receiver exchange status and control signals.
3. Status and control signals can be implemented using data ports. Such implementations are simple in theory but require a lot of software and hardware in practice.
4. If status and control signals are treated as data, determining the status of a peripheral (*polling*) is much like determining the state of a switch. Managing a control output is much like turning an LED on and off. A handshake requires a series of input and output operations.
5. The 6522 Versatile Interface Adapter (VIA) is an LSI device that simplifies polling and handshaking. A VIA contains two bidirectional I/O ports, two status inputs, two bidirectional status or control lines, and latches (interrupt flags) that are set by transitions on the status lines.

6. Bits 0, 1, 3, and 4 of the VIA's interrupt flag register are set by active transitions on the input control lines (bit 0 by CA2, bit 1 by CA1, bit 3 by CB2, and bit 4 by CB1). Bits 0 (CA1), 2 (CA2), 4 (CB1), and 6 (CB2) of the peripheral control register determine whether positive or negative transitions are active (0 means negative, 1 positive). The interrupt flags are cleared automatically when the CPU reads or writes the associated I/O port, thus preparing the VIA for the next transfer.
7. Control line 2 of each VIA port can be an output signal. Bit 3 (port A) or bit 7 (port B) of the peripheral control register determines whether line 2 is input (0) or output (1). If it is an output, there are several operating modes. In the automatic modes (bit 2 or bit 6 = 0), the control line is pulsed automatically after each I/O operation (output only on port B). The pulse lasts either one clock cycle or until the next active transition on control line 1; this choice depends on bit 1 (bit 5) of the peripheral control register (0 makes the pulse last until the next transition, while 1 makes it last one clock cycle). In the manual mode (bit 2 or bit 6 = 1), the control line takes the value of peripheral control register bit 1 or bit 5. The manual mode requires more programming than the automatic mode but can provide pulses of any length and polarity.
8. Programmable I/O devices simplify hardware design. However, the lack of standards for them makes careful program documentation essential. Each programmable device has its own set of operating modes, ways to select those modes, and special features.



INTERRUPTS

PURPOSE

To learn when and how to use interrupts.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, Chapter 9.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 12, pp. 14–1 to 14–5.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, 1982, pp. 63–68, 153–155, 464–489.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 199–200 (I/O

alternatives), 212–222 (interrupts), 343–345 (saving registers during interrupts), 377–378 (address vectors).

AIM 65 Monitor Program Listing, Dynatem, Irvine, CA, 1979, pp. 9–11.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1978, pp. 7–89 to 7–91, 8–26 to 8–30.

R6500 Microcomputer System Hardware Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, pp. 1–17 to 1–25, 2–10 to 2–13, 3–18 to 3–23, 6–20 to 6–23, 6–29 to 6–30.

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, Chapter 9.

WHAT YOU SHOULD LEARN

1. The features of the 6502 interrupt system.
2. How to use the 6522 VIA with interrupts.
3. How to use interrupts in handshake I/O.
4. How to buffer interrupt-driven I/O.
5. How to manage systems with multiple interrupts.

TERMS

Disable stop an activity from proceeding.

Enable allow an activity to proceed.

Interrupt a signal that temporarily suspends the computer's normal sequence of operations and transfers control to a special routine.

Interrupt-driven dependent on interrupts for its operation.

Interrupt mask (or **interrupt enable**) a bit that determines whether interrupts will be recognized. A mask or disable bit must be cleared to allow interrupts, whereas an enable bit must be set.

Interrupt service routine a program that responds to an interrupt.

Interrupt vector a pointer that directs the CPU to an interrupt service routine.

Maskable interrupt an interrupt that the CPU can disable.

Nonmaskable interrupt an interrupt that the CPU cannot disable.

Polling interrupt system an interrupt system in which a program determines the cause of an interrupt by examining the possibilities one at a time.

Priority interrupt system an interrupt system in which some interrupts take precedence over others—that is, are serviced first or can interrupt the others' service routines.

Reentrant able to be executed correctly while the same routine is being interrupted.

Transparent routine a routine that operates without interfering with other routines.

Vectored interrupt an interrupt that produees an identifieation eode (or *vector*) that the CPU ean use to transfer eontrol to the serviee routine.

6502 INSTRUCTIONS

CLI clear interrupt disable (enable interrupts); elear the INTERRUPT DISABLE (I) flag, thus enabling the maskable interrupt ($\overline{\text{IRQ}}$ input).

RTI return from interrupt; load the program eounter and the status register from the staek, assuming that they were saved as shown in Figure C–1.

SEI set interrupt disable (disable interrupts); set the INTERRUPT DISABLE (I) flag, thus disabling the maskable interrupt ($\overline{\text{IRQ}}$ input).

OVERVIEW OF INTERRUPTS

Interrupts go direetly into the CPU. They inform it of an event mueh as the ringing of a telephone tells you that someone is ealling. The program then need not test status inputs. Instead, interrupts foree the CPU to suspend its normal operations and respond immediately.

While interrupts result in a quiek, direet response to external events, they introduee new problems. They make testing and debugging diffieult, sinee they ean oeeur at any time. Extra hardware is often neeessary to eontrol them and simplify their identifieation (this hardware aets like a telephone switehboard). Furthermore, the designer must deeide when to allow them and how to transfer data between the main program and the serviee routines.

6502 INTERRUPT SYSTEM

The 6502 has two interrupt inputs:

1. $\overline{\text{NMI}}$ (nonmaskable interrupt) is eommonly used to inform the proeessor of an impending loss of power. The input is edge-sensitive, so that it will not interrupt its own serviee routine. *Nonmaskable* means that the interrupt eannot be shut out (*disabled*). In many applieations, a drop in the supply voltage below a speeified level eauses a nonmaskable interrupt. In response, the proeessor saves essential data in a low-power memory attaehed to a battery. Obviously, loss of power should have top priority, sinee it will quiekly shut everything down anyway.
2. $\overline{\text{IRQ}}$ is a maskable interrupt generally used for I/O and other normal system funetions. The input is level-sensitive. The I (INTERRUPT DISABLE) flag determines whether the proeessor reeognizes $\overline{\text{IRQ}}$ interrupts. Setting the I flag disables $\overline{\text{IRQ}}$, whereas elearing it enables $\overline{\text{IRQ}}$.

The 6502 responds to an interrupt by completing its current instruction and then fetching a new program counter value from a fixed pair of memory locations (see Table C–1). The processor saves the old program counter (the address of the next instruction) and status register in the stack automatically, as shown in Figure C–1. RTI at the end of the service routine restores the registers saved during the response.

TABLE C–1 MEMORY MAP FOR 6502 INTERRUPT VECTORS

Vector Address		
MS	LS	Input or Instruction
FFFF	FFFE	$\overline{\text{IRQ}}$ input
FFFB	FFFA	$\overline{\text{NMI}}$ input

RESET sets the I flag, disabling the maskable interrupt and allowing the program to initialize the system before any interrupts are accepted. Accepting an interrupt also sets the I flag and thus disables the maskable interrupt. An interrupt will therefore not disturb its own service routine. RTI normally reenables maskable interrupts automatically, since it restores the old I flag.

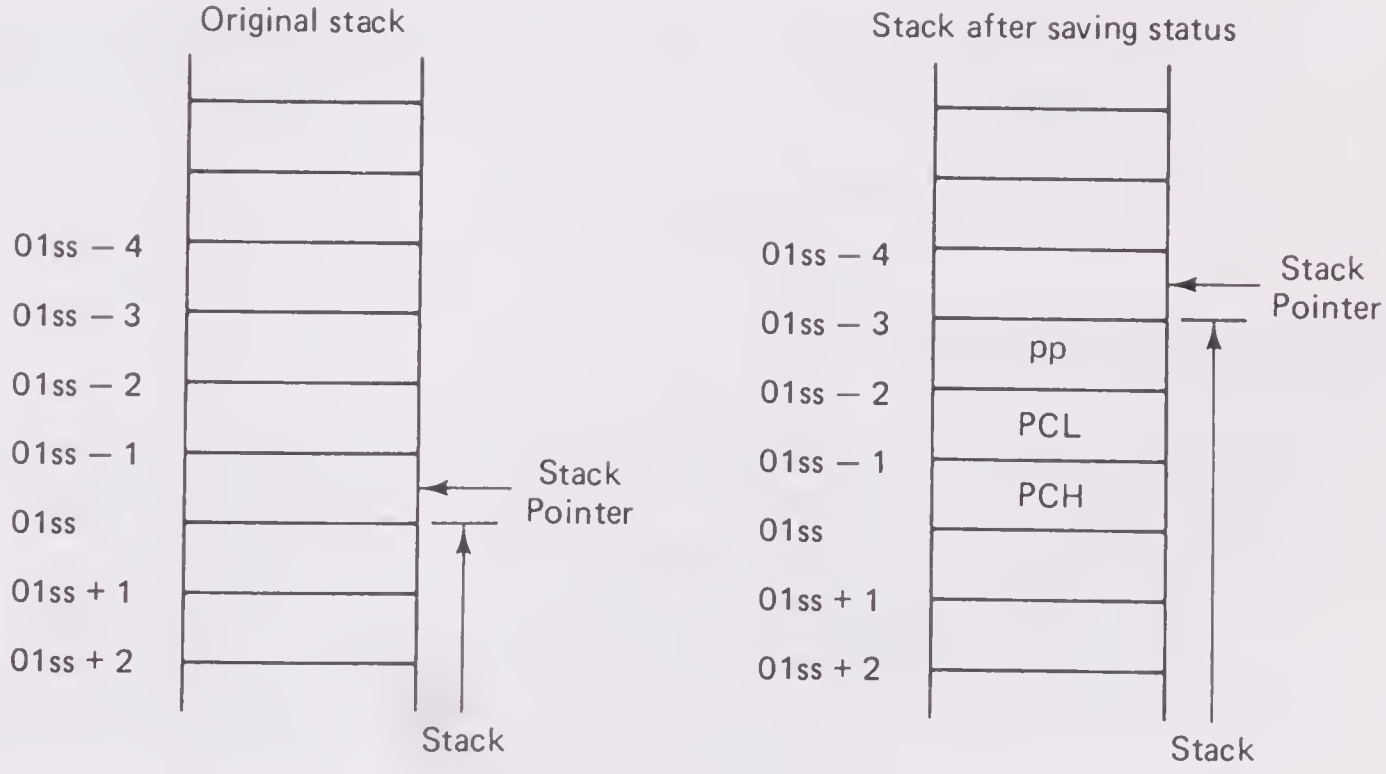


FIGURE C–1. Saving the 6502's status in the stack.

AIM INTERRUPTS

Table C–2 lists the interrupt addresses used by the AIM monitor. It directs (*vectors*) $\overline{\text{NMI}}$ through A402 and A403. The default value there is E07B, the starting address of the STEP routine. In the STEP mode, the AIM produces a nonmaskable interrupt during each instruction executed from user memory.

The monitor vectors $\overline{\text{IRQ}}$ interrupts through A400 and A401. These locations have no default value, so the user must load them.

TABLE C–2 AIM INTERRUPT ADDRESSES

Input	Function	Location of Service Routine	Default Value (Hex)
$\overline{\text{NMI}}$	Nonmaskable interrupt	Address in A402 and A403	E07B
$\overline{\text{IRQ}}$	Maskable interrupt	Address in A400 and A401	None

Before the AIM can respond properly to interrupts, your program must:

1. Initialize the stack pointer, since the interrupt response uses the stack.
2. Put the starting address of the service routine in A400 and A401 ($\overline{\text{IRQ}}$) or A402 and A403 ($\overline{\text{NMI}}$).
3. Enable the maskable interrupt with CLI.

NONMASKABLE INTERRUPTS

The easiest interrupt to use is the nonmaskable one produced by moving the RUN/STEP switch to the STEP position. All your program must do is initialize the stack pointer, since the monitor provides the interrupt vector and the service routine.

The following program (Program C–1 is the mnemonic-entry version) waits for you to move the RUN/STEP switch to the STEP position. It then returns control to the monitor. Obviously, you cannot use the STEP mode to debug this routine.

```
        LDX   #$7F          ; INITIALIZE USER STACK POINTER
        TXS
HERE    JMP   HERE          ; WAIT FOR INTERRUPT
```

Here the interrupt is always enabled; it normally stops program execution and returns control to the monitor. Enter and run Program C–1; be sure to start in the RUN mode.

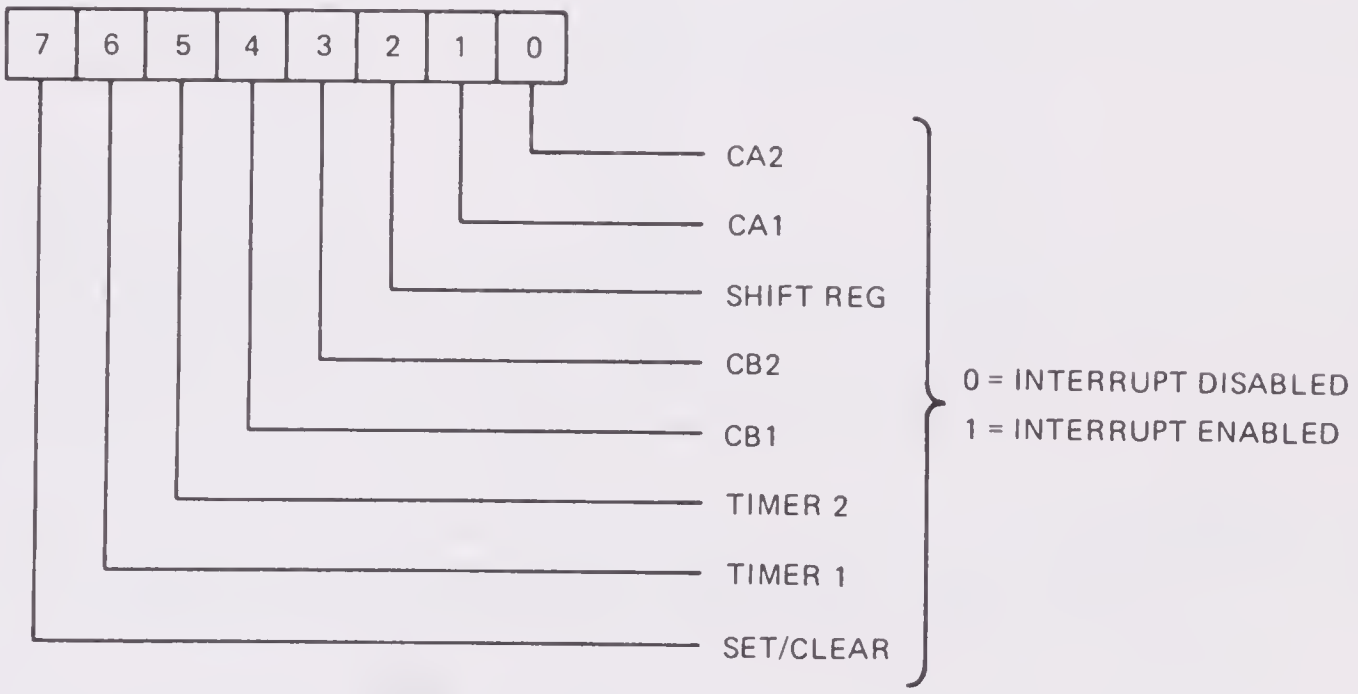
PROGRAM C–1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)
0200	A2		LDX #7F
0201	7F		
0202	9A		TXS
0203	4C	HERE	JMP 0203
0204	03		
0205	02		

6522 VIA INTERRUPTS

A VIA can cause interrupts. Its interrupt flag register (Figure B–7) indicates which ones have occurred, while its interrupt enable register (Figure C–2) indicates which ones are active. Loading the interrupt enable register is tricky since bit 7 is a set or clear control; that is,

1. Bit 7 must be 1 to enable interrupts or 0 to disable them.



NOTES:

1. If bit 7 is a 0, each 1 in bits 0-6 disables the corresponding interrupt.
2. If bit 7 is a 1, each 1 in bits 0-6 enables the corresponding interrupt.
3. If this register is read, bit 7 will be 0, and bits 0-6 will reflect their enable/disable state.

FIGURE C–2. The 6522 VIA's interrupt enable register (IER). (Reprinted courtesy of Rockwell International, Semiconductor Products Division, Newport Beach, CA.)

2. The other bits determine which interrupts are affected. 1s affect the enabling bits, whereas 0s leave them unchanged. We can enable or disable several interrupts at once, but we cannot enable some and disable others in a single operation.

Here are some examples for the user VIA (the interrupt enable register is address A00E; see Table C–3):

1. Enable the CA1 interrupt.

```
LDA   #%10000010          ; ENABLE CA1 INTERRUPT
STA   $A00E
```

Bit 7 = 1 to enable interrupts and bit 1 = 1 to affect specifically the CA1 interrupt.

2. Disable the CB2 interrupt.

```
LDA   #%00001000          ; DISABLE CB2 INTERRUPT
STA   $A00E
```

Bit 7 = 0 to disable interrupts and bit 3 = 1 to affect specifically the CB2 interrupt.

TABLE C–3 REGISTER ADDRESSES IN THE USER VIA

Address In User VIA	Register Designation	Function
A000	ORB/IRB	I/O register B
A001	ORA/IRA	I/O register A
A002	DDRB	Data direction register B
A003	DDRA	Data direction register A
A00C	PCR	Peripheral control register
A00D	IFR	Interrupt flag register
A00E	IER	Interrupt enable register

PROBLEM C–1

Write a program that enables both CA1 and CB1 interrupts.

PROBLEM C–2

Write a program that enables CA1 and CB2 interrupts and disables CA2 and CB1 interrupts.

Let us now make the user VIA cause interrupts. Its interrupt output is tied to the maskable interrupt ($\overline{\text{IRQ}}$), which is serviced at the address in A400 and A401. Initially, we will use the monitor's BRK handling routine (address E163). The main program must:

1. Select the active transition by loading the VIA's peripheral control register (Figure B–6).

2. Load the interrupt veetor; that is, put 63 in A400 and E1 in A401.
3. Enable the VIA interrupts by storing a value with bit 7 set in the interrupt enable register (Figure C–2).
4. Enable the CPU interrupt by elearing the I flag.

The following program (Program C–2 is a mnemonie-entry version) responds to a high-to-low transition on CA1.

```
        LDX   #$7F              ; INITIALIZE USER STACK POINTER
        TXS
        LDA   #0                ; CA1 ACTIVE HIGH-TO-LOW
        STA   $A00C
        LDA   #$63              ; LOAD INTERRUPT VECTOR
        STA   $A400
        LDA   #$E1
        STA   $A401
        LDA   #%10000010        ; ENABLE CA1 INTERRUPT
        STA   $A00E
        CLI                     ; ENABLE CPU INTERRUPT
HERE    JMP   HERE              ; WAIT FOR INTERRUPT
```

Enter and run Program C–2. Closing the CA1 switeh will produee an interrupt. Be sure to elear it afterward by examining A001 (I/O port A).

PROBLEM C–3

Make Program C–2 allow interrupts on high-to-low transitions on CA2. What does the interrupt flag register (address A00D) eontain after the program runs?

How would you make Program C–2 use CB1 instead of CA1 as the interrupt souree? What memory loeation must you examine to elear the interrupt flag? What ehanges must you make to use CB2?

When ehanging interrupts, press RESET to clear the entire interrupt enable register. Otherwise, interrupts from earlier programs will remain enabled and may eause eonfusion. A sequenee that disables all user VIA interrupts is

```
LDA   #%01111111          ; DISABLE ALL INTERRUPTS
STA   $A00E
```

What happens if you replaee LDA #%01111111 with LDA #%11111111?

Note that you eannot use the STEP mode to debug a serviee routine. The STEP mode depends on nonmaskable interrupts, whieh take priority over maskable interrupts. The proeessor will therefore not respond to maskable interrupts when the AIM is in the STEP mode.

PROGRAM C–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A2		LDX	#7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#00
0204	00			
0205	8D		STA	A00C
0206	0C			
0207	A0			
0208	A9		LDA	#7F
0209	7F			
020A	8D		STA	A400
020B	00			
020C	A4			
020D	A9		LDA	#E1
020E	E1			
020F	8D		STA	A401
0210	01			
0211	A4			
0212	A9		LDA	#82
0213	82			
0214	8D		STA	A00E
0215	0E			
0216	A0			
0217	58		CLI	
0218	4C	HERE	JMP	0218
0219	18			
021A	02			

To employ our own service routine instead of the monitor's, we must

1. Change the address Program C–2 loads into A400 and A401. For example, we could make that value 0280 (80 in A400 and 02 in A401).
2. Load our service routine into RAM, beginning at that address. For example, we could place BRK (00) in 0280. Make these changes and run Program C–2 again. How does the display change?

HANDSHAKING WITH INTERRUPTS

Interrupts are often used to perform handshake I/O (see Figures B–4 and B–5). To experiment with this, we need the following startup routine (see Program C–3 for a

mnemonic-entry version). It loads the stack pointer, the VIA's data direction registers, and the AIM's interrupt vector; it also turns all the LEDs off. (We leave the initialization of the peripheral control and interrupt enable registers for later, since their values will vary.)

```
LDX    #$7F          ; INITIALIZE USER STACK POINTER
TXS
LDA    #0            ; MAKE PORT A INPUT
STA    $A003
LDA    #$FF
STA    $A002         ; MAKE PORT B OUTPUT
STA    $A000         ; TURN LEDS OFF
LDA    #$80          ; LOAD INTERRUPT VECTOR
STA    $A400
LDA    #$02
STA    $A401
```

PROGRAM C–3

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A2	LDX #7F
0201	7F	
0202	9A	TXS
0203	A9	LDA #00
0204	00	
0205	8D	STA A003
0206	03	
0207	A0	
0208	A9	LDA #FF
0209	FF	
020A	8D	STA A002
020B	02	
020C	A0	
020D	8D	STA A000
020E	00	
020F	A0	
0210	A9	LDA #80
0211	80	
0212	8D	STA A400
0213	00	
0214	A4	
0215	A9	LDA #02
0216	02	
0217	8D	STA A401
0218	01	
0219	A4	

The following program is an interrupt-driven version of Program B–2. (Program C–4 is a mnemonic-entry version.) It loads data from port A when you close the CA1 switch. The main program clears the DATA READY flag (0040) and waits for the service routine to set it. In response to the interrupt, the service routine loads the data from the input port and sets the DATA READY flag.

Note the following features of Program C–4:

1. We enable the VIA and CPU interrupts last, after initializing all system parameters. Otherwise, an early interrupt could cause problems if it found the VIA in the wrong operating mode.
2. We must set the bits in the VIA's interrupt enable register to allow interrupts, but we must clear the CPU's INTERRUPT DISABLE flag.
3. The final SEI (SET INTERRUPT DISABLE FLAG) disables the CPU interrupt before returning control to the monitor. This precaution avoids conflict with monitor functions or later user programs.
4. The LDA $A001 instruction in the service routine clears the interrupt flag and reads the data from port A.

```
        LDA   #0              ; CA1 ACTIVE HIGH-TO-LOW
        STA   $A00C
        LDA   #%10000010      ; ENABLE CA1 INTERRUPT
        STA   $A00E
        LDA   #0              ; CLEAR READY FLAG
        STA   $40
        CLI                   ; ENABLE CPU INTERRUPT
WTRDY   LDA   $40             ; IS DATA READY?
        BEQ   WTRDY           ; NO, WAIT
        SEI                   ; YES, DISABLE CPU INTERRUPT
        BRK

        *=$0280               ; INTERRUPT SERVICE ROUTINE
        PHA                   ; SAVE ACCUMULATOR
        INC   $40             ; SET READY FLAG
        LDA   $A001           ; GET DATA, CLEAR INTERRUPT
        STA   $41
        PLA                   ; RESTORE ACCUMULATOR
        RTI
```

The service routine need only save and restore the accumulator, since the interrupt response saves the status register automatically. Try running Program C–4 for various inputs at port A. What do you find in 0040 and 0041?

PROBLEM C–4

Make Program C–4 display the data on the LEDs at port B after each closure.

PROGRAM C–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
021A	A9		LDA	#00
021B	00			
021C	8D		STA	A00C
021D	0C			
021E	A0			
021F	A9		LDA	#82
0220	82			
0221	8D		STA	A00E
0222	0E			
0223	A0			
0224	A9		LDA	#00
0225	00			
0226	85		STA	40
0227	40			
0228	58		CLI	
0229	A5	WTRDY	LDA	40
022A	40			
022B	F0		BEQ	0229
022C	FC			
022D	78		SEI	
022E	00		BRK	
0280	48		PHA	
0281	E6		INC	40
0282	40			
0283	AD		LDA	A001
0284	01			
0285	A0			
0286	85		STA	41
0287	41			
0288	68		PLA	
0289	40		RTI	

PROBLEM C–5

Make Program C–4 save input data values starting at 0340. That is, each time you close the CA1 switch, the program should read the data from port A and store it in the next available location. Use 0040 for the buffer index, and make the service routine save and restore any registers it uses.

PROBLEM C–6

Make the main part of Program C–4 wait for a 7F input, the synchronization

character discussed in Laboratory 2. If the input is not 7F, the main program should simply clear the DATA READY flag and wait.

PROBLEM C–7

Write an interrupt-driven version of Program B–3. The main program should clear 0040 to indicate that data is available for output. When a high-to-low transition occurs on CB1, the service routine should send the data from 0041 and set 0040 to 1, indicating that the data has been sent.

PROBLEM C–8

Change the answer to Problem C–7 so that the service routine sends the data only if it is a synchronization character (7F hex). Otherwise, the service routine simply sets 0040 to 1 and clears the interrupt flag. All the LEDs will remain off unless (0041) = 7F.

COMMUNICATING WITH INTERRUPT SERVICE ROUTINES

We cannot use registers to transfer data to or from an interrupt service routine, since the main program generally needs them to do its own work. What we need is an approach that makes the service routines transparent to the main program, so that either can be changed without affecting the other.

One method is to use memory locations (called a *mailbox*) for data transfers. They serve the same purpose as the drops in spy movies. The agent places orders, requests, and payments in the drop; the informant picks up the mail and puts the information in the drop. The agent and the informant never talk or meet. Either can be replaced without affecting the flow of information.

For example, in Program C–4, the main program clears 0040 and waits for the service routine to change it. When that happens, the main program exits. Here the interrupt acts like a RUN command, causing the main program to proceed.

PROBLEM C–9

Make Program C–4 use bit 7 of 0040 as the READY flag. The data is then in bits 0 to 6, as one would expect if it consists of 7-bit ASCII characters.

PROBLEM C–10

Make Program C–4 wait until 0040 contains 10. How would you make it wait until 0040 contains the same value as 0020? This approach is useful when the computer must count external events, such as clock pulses or activations of a sensor.

Obviously, the receiver must check the mailbox often enough to avoid missing messages. One drawback is that other programs may use the mailbox. Imagine an informant hiding valuable papers in a trash container. Unfortunately, before the agent can retrieve them, the sanitation department collects the trash.

BUFFERING INTERRUPTS

Program C–4 handles the input data one character at a time. Clearly, this is awkward and time-consuming if the data rate is high or if only sequences of data are meaningful (as is generally true when the inputs are from a terminal or communications line). The obvious solution is to buffer the data in memory. Then the service routine can fill a buffer, and the main program need not be concerned with individual characters. The buffer serves the same purpose as a buffer memory in a printer or terminal. The computer can send the peripheral a block of data, and the peripheral can then handle individual items at its own speed. In our case, the service routine plays the role of the peripheral.

In the following program, the main program waits until the count in 0040 reaches 4. It also stores the inputs in successive locations starting at 0340. (Program C–5 is a mnemonic-entry version.)

```
        LDA   #0              ; MAKE CA1 ACTIVE HIGH-TO-LOW
        STA   $A00C
        LDA   #%10000010      ; ENABLE CA1 INTERRUPT
        STA   $A00E
        LDA   #0              ; CLEAR COUNT TO START
        STA   $40
        CLI                   ; ENABLE CPU INTERRUPT
        LDA   #4              ; GET TARGET COUNT
WTCNT   CMP   $40             ; ENOUGH INPUTS RECEIVED?
        BNE   WTCNT           ; NO, CONTINUE
        SEI                   ; YES, DISABLE CPU INTERRUPT, EXIT
        BRK

        *=0280
        PHA                   ; SAVE ACCUMULATOR, X REGISTER
        TXA
        PHA
        LDX   $40             ; BUFFER INDEX = COUNT
        LDA   $A001           ; GET INPUT DATA
        STA   $0340,X         ; STORE DATA IN BUFFER
        INC   $40             ; INCREMENT BUFFER INDEX
        PLA                   ; RESTORE X REGISTER, ACCUMULATOR
        RTI
```

This service routine must save and restore both A and X. Remember that the processor saves only the program counter and status register automatically.

Enter and run Program C–5. Set the switches to form the following array:

(0340) = F0 (11110000 binary)
(0341) = 0F (00001111 binary)
(0342) = AA (10101010 binary)
(0343) = 55 (01010101 binary)

PROGRAM C–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
021A	A9		LDA	#00
021B	00			
021C	8D		STA	A00C
021D	0C			
021E	A0			
021F	A9		LDA	#82
0220	82			
0221	8D		STA	A00E
0222	0E			
0223	A0			
0224	A9		LDA	#00
0225	00			
0226	85		STA	40
0227	40			
0228	58		CLI	
0229	A9		LDA	#04
022A	04			
022B	C5	WTCNT	CMP	40
022C	40			
022D	D0		BNE	022B
022E	FC			
022F	78		SEI	
0230	00		BRK	
0280	48		PHA	
0281	8A		TXA	
0282	48		PHA	
0283	A6		LDX	40
0284	40			
0285	AD		LDA	A001
0286	01			
0287	A0			
0288	9D		STA	0340,X
0289	40			
028A	03			
028B	E6		INC	40
028C	40			
028D	68		PLA	
028E	AA		TAX	
028F	68		PLA	
0290	40		RTI	

PROBLEM C–11

Make Program C–5 fill the buffer until it receives an input of 0D, an ASCII carriage return character. Use 0041 as an END OF LINE flag. The main program should clear the flag initially and then wait for it to be set. The service routine should set the flag when it receives a 0D input. A program like this handles input from a terminal one line at a time.

PROBLEM C–12

Make Program C–5 fill the buffer with a message that starts with an ASCII STX (Start of Text) character (02) and ends with an ASCII ETX (End of Text) character (03). All inputs before the STX are simply ignored, and the STX and ETX characters themselves are not placed in the buffer. Such control characters are often used for synchronization.

Example

If the inputs are (in order of receipt)

```
67
B2
02          ASCII STX
47          ASCII G
5F          ASCII O (letter)
0D          ASCII Carriage Return
03          ASCII ETX
```

the final buffer contents are

```
(0340) = 47     ASCII G
(0341) = 5F     ASCII O (letter)
(0342) = 0D     ASCII Carriage Return
```

The two inputs preceding the STX are ignored. The STX and ETX characters do not appear in the buffer. (*Hint:* Use 0042 as a TRANSMISSION IN PROGRESS flag. The main program should clear the flag initially, and the service routine should set the flag when it receives an STX input.)

PROBLEM C–13

A common practice, called *double buffering*, allows the service routine to fill one buffer while the main program processes another. Extend Program C–5 so that it first fills the buffer starting at 0340 with four inputs and then fills the buffer starting at 0360. Use

0041 as a flag that indicates whether the first buffer is full (0 means empty, 1 full), and use 0042 as a similar flag for the second buffer. Use 0043 and 0044 to hold the address of the buffer the service routine is currently using. Be sure to disable CPU interrupts while changing buffers. (Why?)

Example

Initially, 0041 and 0042 are both cleared. If the inputs are (in order of receipt)

FE (all switches open except #0)
FD (all switches open except #1)
FB (all switches open except #2)
F7 (all switches open except #3)
EF (all switches open except #4)
DF (all switches open except #5)
BF (all switches open except #6)
7F (all switches open except #7)

the first four values end up in the first buffer and the second four in the second buffer. 0041 is set to 1 after the first four inputs have been received; 0042 is set to 1 after the second four inputs have been received. The values in the first buffer are

(0340) = FE
(0341) = FD
(0342) = FB
(0343) = F7

The values in the second buffer are

(0360) = EF
(0361) = DF
(0362) = BF
(0363) = 7F

PROBLEM C–14

Write an interrupt-driven output routine that transmits data from a buffer starting at 0340 until it finds a 0D value, the ASCII carriage return character.

Example

(0340) = 80
(0341) = 40

```
(0342) = 20
(0343) = 10
(0344) = 0D
```

The output should be a single light that moves right one position after each interrupt. The program should exit with the displays showing 0D.

MULTIPLE SOURCES OF INTERRUPTS

So far, we have assumed a single source of interrupts. Real applications normally have several sources, such as input devices, output devices, alarms, timers, control panels, and remote stations. The problem is how to determine which source caused an interrupt. Once that is done, the processor must execute the appropriate service routine.

The least expensive approach is to examine the status of one source at a time. This is like answering a telephone connected to several lines by trying lines successively until you find the caller. The first active source is serviced, and the others are handled in the order of examination. This approach (called *polling*) is particularly simple for a VIA since its status is readily available in its interrupt flag register.

The next program waits for an interrupt on either CA1 or CB1 of the user VIA. The service routine examines the interrupt flag register and services the first interrupt it finds active. If the input (CA1) interrupt is active, the routine increments 0040 and loads the input data into 0041. If the output (CB1) interrupt is active, the routine increments 0042 and sends the output data from 0043.

The main service routine disables the VIA interrupts so you can see the effects of priority. Otherwise, the second interrupt would be serviced as soon as the first service routine was completed. The disabling keeps the lower-priority interrupt from being serviced, hence its interrupt flag remains set. You can check the interrupt flags by examining A00D, but be careful not to clear them by examining the I/O ports.

In the general case with many VIAs, the polling routine need initially examine only bit 7 of each interrupt flag register. This is the IRQ (interrupt request) flag (see Figure B–7); it is 1 if any interrupt condition is both active and enabled. A polling routine can therefore use BIT to examine many VIAs successively. It need only examine the rest of the interrupt flag register after it narrows its search to a particular VIA.

```
        LDA   #0              ; INTERRUPTS ACTIVE HIGH-TO-LOW
        STA   $A00C
        LDA   #%10010010      ; ENABLE CA1, CB1 INTERRUPTS
        STA   $A00E
        LDA   #0              ; CLEAR READY FLAGS
        STA   $40
        STA   $42
        CLI                   ; ENABLE CPU INTERRUPT
WTINT   CMP   $40             ; INPUT DATA AVAILABLE?
        BNE   DONE            ; YES, DONE
```

```
                CMP   $42          ; OUTPUT BUFFER EMPTY/
                BEQ   WTINT        ; NO, WAIT
DONE            SEI                ; EXIT BACK TO MONITOR
                BRK

                *=$0280            ; POLLING ROUTINE
                PHA                ; SAVE ACCUMULATOR
                LDA   #%00010010   ; DISABLE FURTHER VIA INTERRUPTS
                STA   $A00E
                LDA   #%00000010   ; INPUT INTERRUPT ACTIVE?
                BIT   $A00D
                BNE   SRVIN        ; YES, SERVICE INPUT
                LDA   #%00010000   ; OUTPUT INTERRUPT ACTIVE?
                BIT   $A00D
                BNE   SRVOUT       ; YES, SERVICE OUTPUT
                PLA                ; RETURN IF NEITHER ACTIVE
                RTI

                *=$02C0            ; INPUT (CA1) INTERRUPT SERVICE
SRVIN           INC   $40          ; SET INPUT READY FLAG
                LDA   $A001        ; FETCH DATA FROM INPUT PORT
                STA   $41          ; SAVE DATA
                PLA                ; RESTORE ACCUMULATOR
                RTI

                *=$02E0            ; OUTPUT (CB1) INTERRUPT SERVICE
SRVOUT          INC   $42          ; SET OUTPUT READY FLAG
                LDA   $43          ; GET OUTPUT DATA
                EOR   #$FF         ; INVERT POLARITY
                STA   $A000        ; SEND DATA TO LEDS
                PLA                ; RESTORE ACCUMULATOR
                RTI
```

You can test the priority scheme by setting both interrupt flags before executing Program C–6. Only the input interrupt will be serviced, and the output interrupt (CB1) flag will remain set. You must, however, remove the STA $A000 instruction from Program C–3 (replace it with three NOPs) since it will clear the CB1 interrupt flag.

PROBLEM C–15

If an input interrupt and an output interrupt occur simultaneously, which will the processor service? Why? How could you change the polling routine to invert the priority?

PROBLEM C–16

Some interrupt systems may ignore low-priority interrupts indefinitely if there are many high-priority interrupts. The situation is like that of a caller who is left "on hold" forever. One way to ensure that all interrupts get serviced is to rotate the priorities. Make

the polling routine invert the order in which it examines interrupts as part of each execution. Use 0044 as a flag indicating the order of examination (00 means "input interrupt first" and FF means "output interrupt first").

PROGRAM C–6

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
021A	A9		LDA	#00
021B	00			
021C	8D		STA	A00C
021D	0C			
021E	A0			
021F	A9		LDA	#92
0220	92			
0221	8D		STA	A00E
0222	0E			
0223	A0			
0224	A9		LDA	#00
0225	00			
0226	85		STA	40
0227	40			
0228	85		STA	42
0229	42			
022A	58		CLI	
022B	C5	WTINT	CMP	40
022C	40			
022D	D0		BNE	0233
022E	04			
022F	C5		CMP	42
0230	42			
0231	F0		BEQ	022B
0232	F8			
0233	78	DONE	SEI	
0234	00		BRK	
0280	48		PHA	
0281	A9		LDA	#12
0282	12			
0283	8D		STA	A00E
0284	0E			
0285	A0			
0286	A9		LDA	#02
0287	02			
0288	2C		BIT	A00D
0289	0D			
028A	A0			

PROGRAM C–6 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
028B	D0		BNE	02C0
028C	33			
028D	A9		LDA	#10
028E	10			
028F	2C		BIT	A00D
0290	0D			
0291	A0			
0292	D0		BNE	02E0
0293	4C			
0294	68		PLA	
0295	40		RTI	
02C0	E6	SRVIN	INC	40
02C1	40			
02C2	AD		LDA	A001
02C3	01			
02C4	A0			
02C5	85		STA	41
02C6	41			
02C7	68		PLA	
02C8	40		RTI	
02E0	E6	SRVOUT	INC	42
02E1	42			
02E2	A5		LDA	43
02E3	43			
02E4	49		EOR	#FF
02E5	FF			
02E6	8D		STA	A000
02E7	00			
02E8	A0			
02E9	68		PLA	
02EA	40		RTI	

PROBLEM C–17

Write a program for a complete interrupt-driven I/O system. The program should initially enable only the input interrupt and wait for input data. When it receives data, it should disable the input interrupt, enable the output interrupt, and wait for the output device to become ready. When the output device is ready, the program should send the input data, disable the output interrupt, and complete the cycle by enabling the input interrupt. Note that the VIA latches transitions that occur while its interrupt outputs are disabled.

Do not service a disabled interrupt. Remember that its interrupt flag can still be set. No interrupt will occur, but a polling routine will find the flag set. Thus if some VIA interrupts are disabled, your polling routine should not check their flags. You can determine if an interrupt is enabled by testing the interrupt enable register.

Polling is adequate if there are only a few sources. As the number increases, however, it becomes slow and cumbersome. The alternative is a vectored system in which each interrupt directs the CPU to its service routine. For example, we could connect the interrupts to an encoder like the one discussed in Laboratory 4. The processor could then read the encoded value from an input port and use it to select a service routine.

GUIDELINES FOR PROGRAMMING WITH INTERRUPTS

In designing interrupt-based systems, the programmer should consider the following guidelines.

1. Initialize all parameters before enabling interrupts.
2. Make all service routines transparent to the programs they can interrupt.
3. Provide a well-defined method for transferring data between the main program and the service routines. This method should be flexible and program-independent.

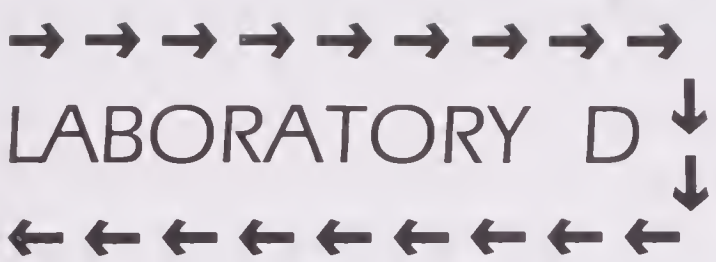
There are many aspects of programming with interrupts that we have not discussed. Among them are reentrant programs that can be interrupted and resumed even if the interrupt service executes them. Such programs must use the registers and the stack for temporary storage, not specific memory addresses that would be overwritten.

Still another problem is the need to disable interrupts during activities that could not be resumed properly, such as delay loops, command sequences, and updating of multiple-byte results that must be used during the interrupt service. If a program is changing multibyte data, it must finish if the interrupt service routine uses the data. Otherwise, the service routine could find the data only partially changed.

KEY POINT SUMMARY

1. Interrupts allow a computer to respond rapidly and directly to external events such as changes in the status of peripherals. The program need not check whether events have occurred, since the occurrences change hardware inputs.
2. The 6502 microprocessor has two interrupts: maskable ($\overline{\text{IRQ}}$) and nonmaskable ($\overline{\text{NMI}}$). In response to them, the processor saves the program counter and status register in the stack, disables the maskable interrupt, and fetches a new program counter value from a specified pair of memory locations. An RTI instruction at the end of the service routine restores the old program counter and status register from the stack. The service routine must save and restore the accumulator and index registers if it uses them.

3. Before enabling interrupts, the main program must load the stack pointer and initialize any parameters that the service routines use. It must also determine the operating modes for VIAs and other I/O devices. RESET disables the CPU interrupt and the VIA interrupts.
4. The main program and the service routines cannot communicate through the registers, because each needs them for its own use. A simple way to communicate is through memory locations, which act like a mailbox. One program can place information there for the other to read.
5. Buffering allows the main program and the service routine to communicate less often. The main program need only concern itself with an entire buffer's worth of data. Either a large buffer or multiple buffers (double buffering) gives the main program more time to do its work without missing data or ignoring requests for service.
6. A VIA can be used in an interrupt-driven mode by setting bits in its interrupt enable register. Transitions on the control lines then cause interrupts as well as set the interrupt flags.
7. If there are many sources of interrupts, the program must have a way of identifying them. Polling means that the processor examines the status of successive sources until it finds one that is active. Vectoring means that each source provides its own identification.
8. In polling interrupt systems, the priority of the sources depends on the order of examination. This order can be changed or varied if necessary. Since the time required to identify a source increases linearly with how many there are, polling is reasonable only in systems with just a few sources.



TIMING METHODS

PURPOSE

To learn how to time input and output operations.

PARTS REQUIRED

- A low-frequency clock input (5 to 200 Hz), such as one generated from a 555 timer (see Figure D–1). The clock should be tied with a jumper to pin PA5 of the user VIA (pin 6 of the Application Connector) as shown in Figure D–2. Jumpers will let you choose between the clock input and the switch in Figure 2–1.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 343–345, 485–486.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11–8 to 11–11, 11–36 to 11–42, 12–23 to 12–31.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, 1982, pp. 460–463, 490–503.

A. OSBORNE and G. KANE, *Osborne 4- and 8-Bit Microprocessor Handbook*, Osborne/McGraw-Hill, Berkeley, CA, 1981, pp. 10–35 to 10–55.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 43–46 (counters), 88–89 (process control), 368–375 (timing loops).

AIM 65 Monitor Program Listing, Dynatem, Irvine, CA, 1978, pp. 39, 55.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1978, pp. 8–30 to 8–40 (6522 timers), Appendix K (AIM 65 user submonitor with 24-hour clock).

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, pp. 6–8 to 6–14, 6–30 to 6–33.

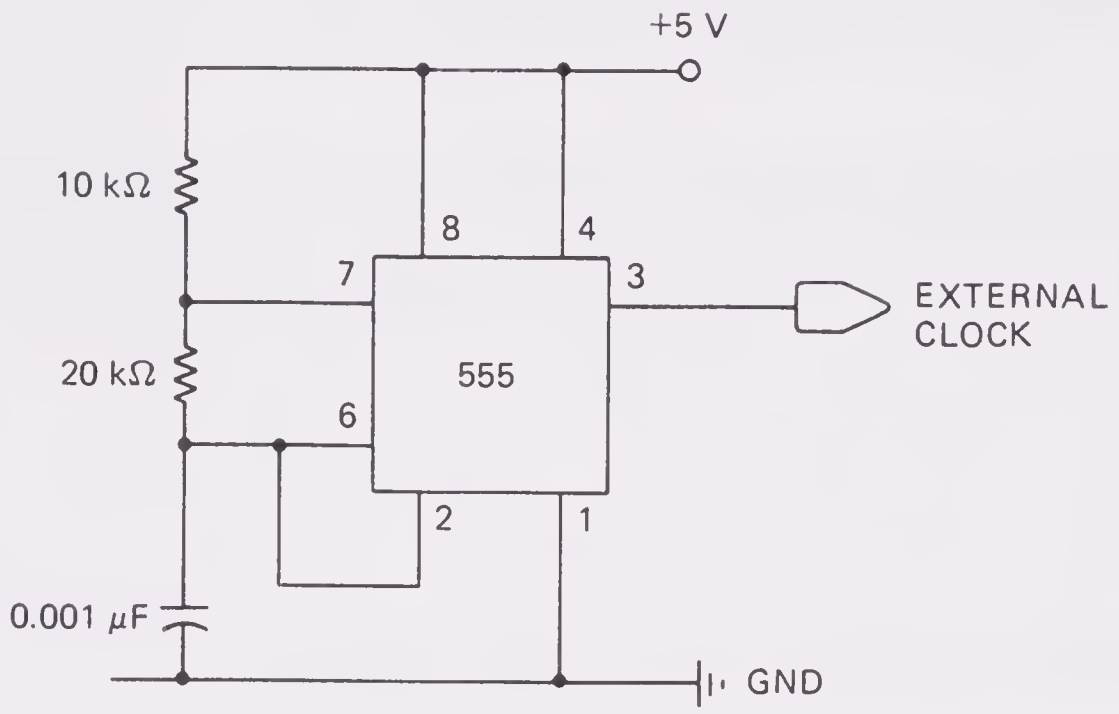


FIGURE D–1. Simple circuit for generating a low-frequency clock from a 555 timer. The frequency is approximately 83 Hz.

FIGURE D–2. Connection of the external clock to bit 5 of port A of the user VIA. (*Note:* Jumper wires can be used to select either this input or the switch input in Figure 2–1.)

WHAT YOU SHOULD LEARN

1. How to synchronize with an external clock.
2. How to determine the period of an external clock.
3. How to program the timers in the 6522 VIA.
4. How to use an elapsed time interrupt.
5. How to produce and use a real-time clock.

TERMS

Multitasking executing many tasks at once, usually by working on the highest-priority task that is currently active and suspending tasks that must wait for I/O, the completion of other tasks, or external events.

One-shot a device that produces a single pulse of known length in response to a pulse input.

Operating system (OS) a computer program that controls the overall operations of a computer and performs such functions as assigning places in memory to programs and data, scheduling the execution of programs, processing interrupts, and controlling the overall I/O system. Also known as a *monitor* or *executive*.

Programmable timer a device that can perform many timing functions, such as generating delays, under program control.

Real-time in synchronization with the actual occurrence of events.

Real-time clock a device that interrupts a CPU at regular time intervals.

Real-time operating system an operating system that can control programs with real-time requirements.

Scheduler a program that decides when to start and terminate other programs.

Supervisor a program that controls the loading and execution of other programs.

Suspend halt execution and preserve the status of a task.

Task a self-contained program that forms part of a system under the control of a supervisor.

Task status the parameters that specify a task's current state.

Utility a program that performs a common overhead operation such as sorting, converting data from one format to another, or copying a file.

TIMING REQUIREMENTS AND METHODS

Timing is a continual problem in microprocessor applications. Systems must handle inputs and outputs at the proper rates and perform their work on schedule. Delay programs, such as those described in Laboratory 3, can meet simple timing requirements.

However, since they occupy the processor completely, they are inadequate for applications with complex, varying timing needs.

Many applications, particularly in process and industrial control, involve real-time constraints; the microcomputer must take measurements and control operations at specific times. Some applications, such as energy management systems, navigation systems, and security systems, must actually maintain a time-of-day clock and a calendar.

We will explore the following methods of handling timing:

1. Adapting to the frequencies of external clocks.
2. Using a programmable timer.
3. Using a real-time clock.

These methods provide more flexibility than fixed delay routines; timers also reduce the burden on the processor.

WAITING FOR A CLOCK TRANSITION

Many systems use either hardware (such as synchronizing circuits) or parameter values stored in ROM to determine when time intervals begin and how long they last. This approach is simple and compatible with systems that are not computer-based, but it is inflexible. You cannot readily modify the systems to handle different or improved peripherals. This is a major handicap currently, since peripheral (e.g., printer, terminal, tape, and disk) technology is undergoing rapid change. A system that cannot be upgraded easily soon becomes outdated and uncompetitive.

An alternate approach is for the program to determine the time constants for its I/O devices. For example, the system could synchronize itself to a clock input. Attach a low-frequency (5- to 200-Hz) clock to bit 5 of port A of the user VIA as shown in Figure D–2. The following program (see Program D–1 and Figure D–3) waits for a low-to-high clock transition.

```
WAITL   LDA   $A001          ; IS CLOCK LINE LOW?
        AND   #%00100000
        BNE   WAITL           ; NO, WAIT
WAITH   LDA   $A001          ; IS CLOCK LINE HIGH?
        AND   #%00100000
        BEQ   WAITH           ; NO,WAIT
        BRK
```

Enter and run Program D–1. Vary the clock rate. How would you make the program wait for a low-to-high transition? Using a debounced switch as the clock input will give you complete control for testing purposes.

PROGRAM D–1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	AD	WAITL	LDA	A001
0201	01			
0202	A0			
0203	29		AND	#20
0204	20			
0205	D0		BNE	0200
0206	F9			
0207	AD	WAITH	LDA	A001
0208	01			
0209	A0			
020A	29		AND	#20
020B	20			
020C	F0		BEQ	0207
020D	F9			
020E	00		BRK	

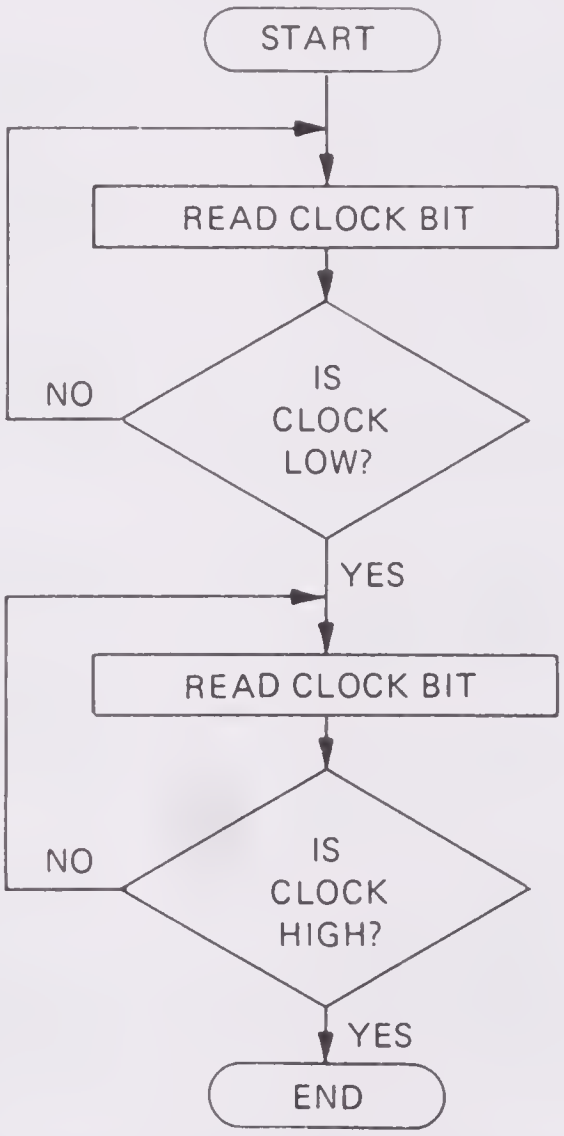


FIGURE D–3. Flowchart of the clock synchronization program.

PROBLEM D–1

Make Program D–1 wait for a full clock pulse (i.e., a low-to-high transition followed by a high-to-low transition).

PROBLEM D–2

Make Program D–1 wait for ten low-to-high transitions.

MEASURING THE CLOCK PERIOD

We can extend Program D–1 to make the processor measure the clock's period. This involves:

1. Waiting for a transition.
2. Counting time intervals until the next similar transition. Obviously, the period must last many CPU clock cycles for this method to work.

The following program (see Figure D–4 for a flowchart) waits for a low-to-high clock transition and then counts milliseconds until another one occurs:

```
        LDY     #0                      ; CLOCK COUNT = ZERO
WTL1    LDA     $A001                   ; IS CLOCK LINE LOW?
        AND     #%00100000
        BNE     WTL1                    ; NO, WAIT
WTH1    LDA     $A001                   ; IS CLOCK LINE HIGH?
        AND     #%00100000
        BEQ     WTH1                    ; NO, WAIT
WTL2    INY                             ; INCREMENT CLOCK COUNT
        LDX     #$C8                    ; WAIT 1 MS
DLYL    DEX
        BNE     DLYL
        LDA     $A001                   ; IS CLOCK LINE LOW?
        AND     #%00100000
        BNE     WTL2                    ; NO, WAIT
WTH2    INY                             ; INCREMENT CLOCK COUNT
        LDX     #$C8                    ; WAIT 1 MS
DLYH    DEX
        BNE     DLYH
        LDA     $A001                   ; IS CLOCK LINE HIGH?
        AND     #%00100000
        BEQ     WTH2                    ; NO, WAIT
        STY     $40                     ; SAVE CLOCK COUNT
        BRK
```

Program D–2 is the mnemonic-entry version; enter it into memory and run it. Check how accurately it measures some low-frequency clocks.

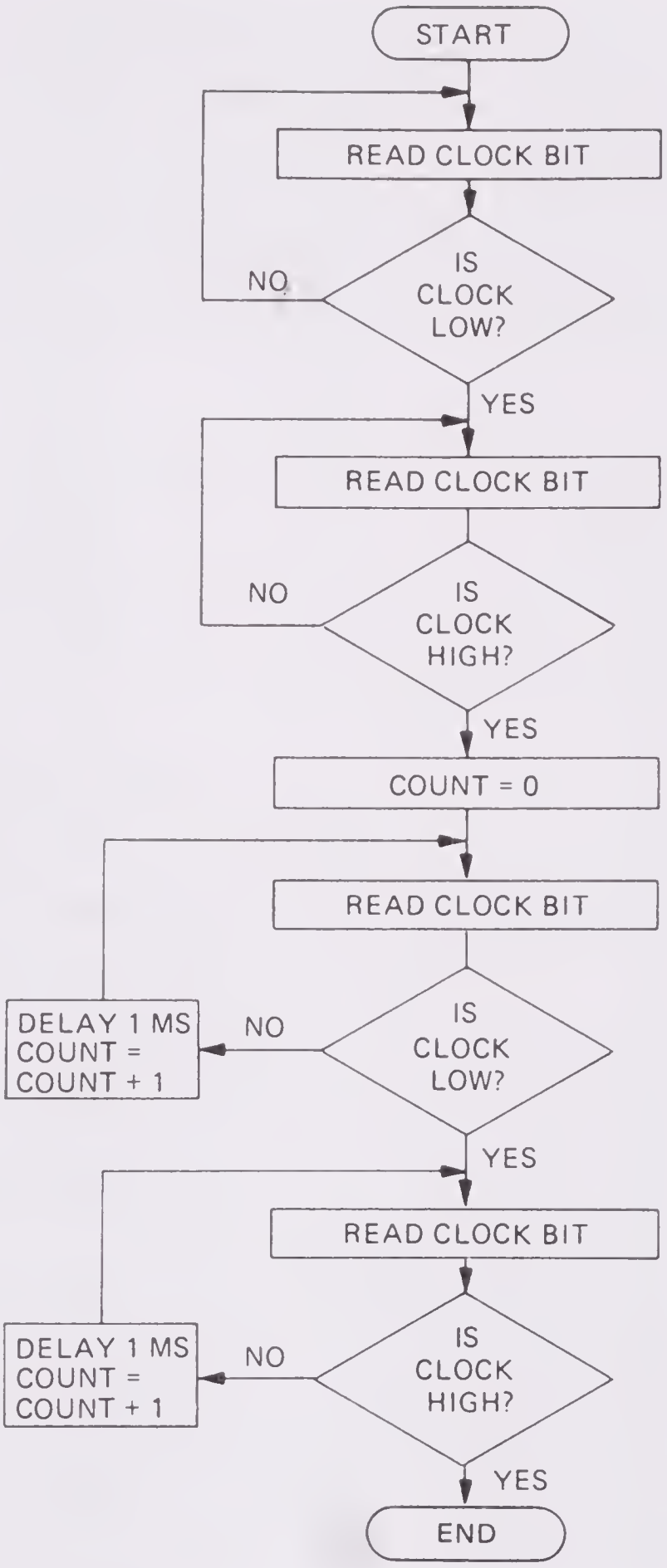


FIGURE D–4. Flowchart of the program that measures a clock period.

PROBLEM D–3

Make Program D–2 measure the high phase of the clock.

PROBLEM D–4

Make Program D–2's resolution 100 μs instead of 1 ms. Measure the period of your clock using both resolutions.

By using the measured clock period to time input and output, a program can operate at different rates. For example, it could handle serial I/O with a terminal at any common data rate (10 or 30 characters per second for low-speed units, 1,200 to 19,200 bits per second for higher-speed units).

The AIM monitor uses this approach to determine the data rate of a teletypewriter. All you must do is connect the teletypewriter, move the KB/TTY switch to the TTY position, reset the AIM, and type a RUBOUT (ASCII RUBOUT = 7F hex = 01111111 binary). The AIM measures the time between bits and saves the corresponding values in locations A417 and A418.

PROGRAM D–2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A0		LDY	#00
0201	00			
0202	AD	WTL1	LDA	A001
0203	01			
0204	A0			
0205	29		AND	#20
0206	20			
0207	D0		BNE	0202
0208	F9			
0209	AD	WTH1	LDA	A001
020A	01			
020B	A0			
020C	29		AND	#20
020D	20			
020E	F0		BNE	0209
020F	F9			
0210	C8	WTL2	INY	
0211	A2		LDX	#C8
0212	C8			
0213	CA	DLYL	DEX	
0214	D0		BNE	0213
0215	FD			
0216	AD		LDA	A001
0217	01			
0218	A0			
0219	29		AND	#20
021A	20			
021B	D0		BNE	0210
021C	F3			
021D	C8	WTH2	INY	
021E	A2		LDX	#C8
021F	C8			
0220	CA	DLYH	DEX	

PROGRAM D–2 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0221	D0	BNE 0220
0222	FD	
0223	AD	LDA A001
0224	01	
0225	A0	
0226	29	AND #20
0227	20	
0228	F0	BEQ 021D
0229	F3	
022A	84	STY 40
022B	40	
022C	00	BRK

PROGRAMMABLE TIMERS

The previous methods still depend on the proeessor generating time intervals with delay routines. An alternative is to use a hardware timer under eomputer eontrol. The proeessor then only has to determine how the timer will operate, start it, and wait for it to indieate the end of the time interval.

The simplest hardware timer is a one-shot that produees a single pulse of fixed length in reponse to a pulse input. More complex timers eontain counters and lateh es; input eontrols may determine how many stages are used.

Programmable timers are like the programmable input/output ports deseribed in Laboratory B. These deviees have many operating modes; the program seleets one by loading eontrol registers. The timer's eurrent state may be determined by examining its status registers. Typieal timer options are binary or deeimal (BCD) eounts, the shape of output pulses (e.g., square wave or brief pulse on terminal eount), whether an interrupt is produeed, whether the eloek is divided, and whether the timer operates continuously (i.e., reloads its eounters with their initial values after eounting them down to zero).

Like programmable interfaees, programmable timers simplify hardware design, save parts, and allow the use of standard boards in many applications. On the other hand, they are diffieult to use and doeument beeause of their arbitrary features and unique programming requirements.

6522 INTERVAL TIMERS

The 6522 VIA contains two 16-bit timers. Eaeh has two 8-bit eounter registers; timer 1 also has separate lateh es that ean be loaded without affeeting its eurrent operation. A program determines the starting eount by storing it in the timer with two 8-bit operations.

Programs should always load the less significant byte first, since loading the more significant byte transfers the entire 16-bit value to the actual counter and starts the timer. When the count reaches zero, the assigned bit (#5 for timer 2, #6 for timer 1) in the VIA's interrupt flag register (see Figure D–5) is set to 1. The processor may clear the interrupt flag by either reading the less significant byte of the counter or loading the more significant byte.

TABLE D–1 USER VIA TIMER ADDRESSES

Address (Hex)	Register Designation	Function
A004	T1C-L	Timer 1 counter low-order byte
A005	T1C-H	Timer 1 counter high-order byte
A006	T1L-L	Timer 1 latch low-order byte
A007	T1L-H	Timer 1 latch high-order byte
A008	T2C-L	Timer 2 low-order byte
A009	T2C-H	Timer 2 high-order byte
A00B	ACR	Auxiliary control register
A00D	IFR	Interrupt flag register
A00E	IER	Interrupt enable register

Note: Timer 1 has two pairs of addresses, A004-A005 and A006-A007. The difference is that loading A006 and A007 does not load the actual counter or clear the interrupt flag. Thus A006 and A007 can be changed without affecting the timer's current operation; such changes will, however, affect later operations and can be used to create complex, varying waveforms.

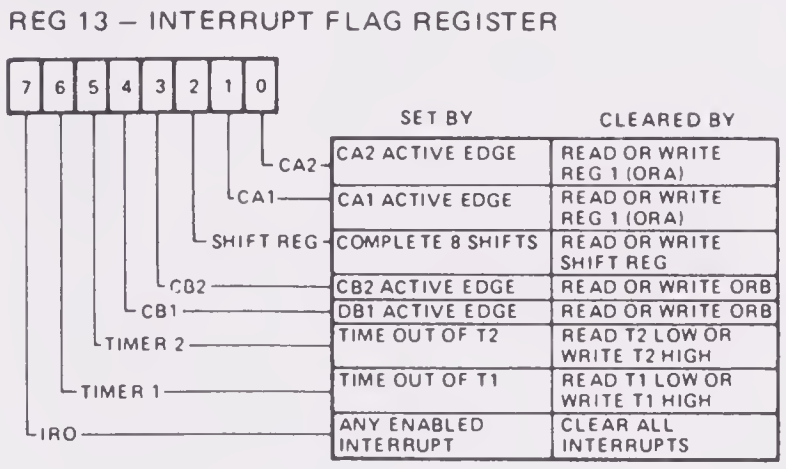


FIGURE D–5. VIA interrupt flag register (IFR). (Reprinted courtesy of Rockwell International, Semiconductor Products Division, Newport Beach, CA.)

The auxiliary control register (see Figure D–6) determines how the timers operate. We will be concerned only with the following options:

- Bit 5 = 0 to make timer 2 simply count down its initial value using the 6502's clock. This is called a *one-shot mode*, since it creates a single pulse of known duration.
- Bits 6 and 7 = 0 to make timer 1 operate continuously (*free-running mode*), reloading its counters from the latches after each interval.

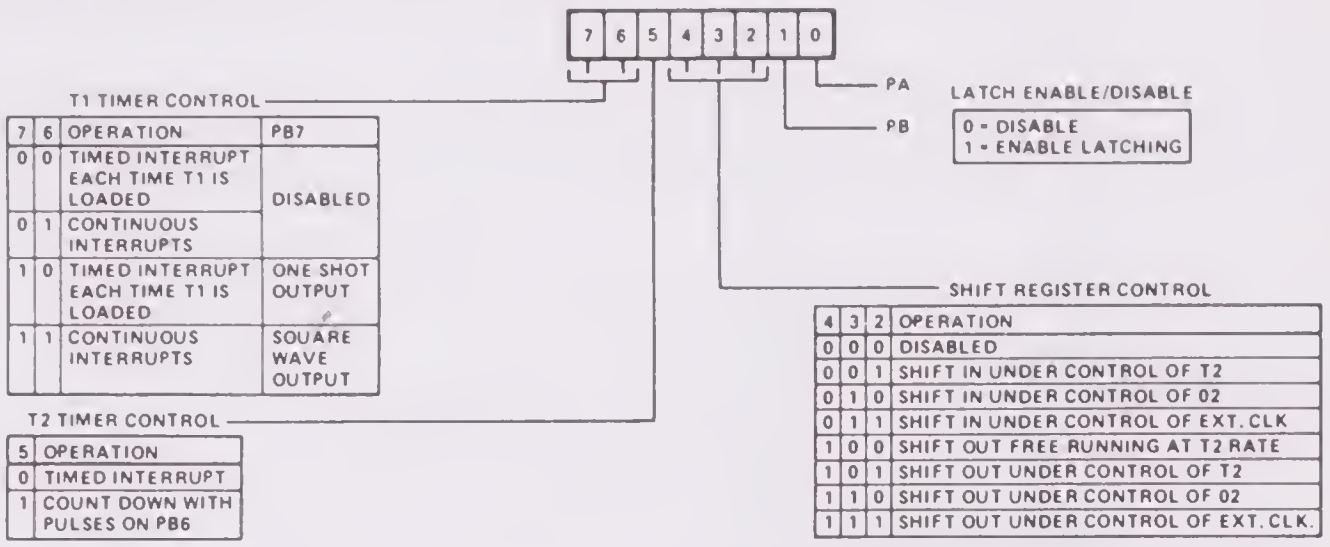


FIGURE D–6. VIA auxiliary control register (ACR). (Reprinted courtesy of Rockwell International, Semiconductor Products Division, Newport Beach, CA.)

To use a timer, a program must do the following:

1. Determine the operating mode by loading the auxiliary control register.
2. Load the timer's counter, less significant byte first. Loading the more significant byte actually loads the 16-bit internal counter and starts the timer.
3. Wait for the timer interrupt flag to be set.
4. Clear the timer interrupt flag by reading the counter's less significant byte or by loading its more significant byte. Of course, loading its more significant byte also starts a new countdown.

The following program (Program D–3 is the mnemonic-entry version) produces a delay of 50,000 clock cycles (50 ms) by storing C350 in timer 1's counter in the user VIA. The program clears the auxiliary control register, thus making timer 1 produce a single countdown. It then waits for bit 6 of the interrupt flag register to be set (indicating that the count has reached zero). Finally, it clears the bit by reading the less significant byte of timer 1's counter. Note (see Table D–1) that the user VIA's timers occupy addresses A004 and A005 (timer 1) and A008 and A009 (timer 2) and that its auxiliary control register occupies address A00B. We are ignoring an overhead factor (about two clock cycles) that occurs in each 6522 timing operation (see Figures 6–7 and 6–8 in the *R6500 Microcomputer System Hardware Manual*).

```
        LDA   #0
        STA   $A00B      ; TIMER 1 IN ONE-SHOT MODE, DISABLE PB7
        LDA   #$50       ; START COUNT AT 50,000 (C350 HEX)
        STA   $A004
        LDA   #$C3
        STA   $A005
WTTIM   BIT   $A00D      ; IS INTERVAL OVER?
        BVC   WTTIM      ; NO, WAIT
        LDA   $A004      ; YES, CLEAR TIMER 1 INTERRUPT FLAG
        BRK
```

PROGRAM D–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A9		LDA	#00
0201	00			
0202	8D		STA	A00B
0203	0B			
0204	A0			
0205	A9		LDA	#50
0206	50			
0207	8D		STA	A004
0208	04			
0209	A0			
020A	A9		LDA	#C3
020B	C3			
020C	8D		STA	A005
020D	05			
020E	A0			
020F	2C	WTTIM	BIT	A00D
0210	0D			
0211	A0			
0212	50		BVC	020F
0213	FB			
0214	AD		LDA	A004
0215	04			
0216	A0			
0217	00		BRK	

PROBLEM D–5

Make Program D–3 use timer 2. Which timer is easier to use? Why?

PROBLEM D–6

Make Program D–3 wait for ten intervals given by a count of C350 hex. How would you make the program obtain the count from 0040 and 0041 (more significant byte in 0041) and the number of intervals from 0042?

Sample Case

(0040) = 20 (less significant byte of count)
(0041) = 4E (more significant byte of count)
(0042) = 32 (number of intervals)

Result: The program waits for 32 hex (50 decimal) intervals given by a count of 4E20 hex (20,000 decimal). The total delay should therefore last approximately 1 s.

ELAPSED TIME INTERRUPTS

Obviously, we have not yet solved the problem of using the processor efficiently. Program D–3 still forces it to wait idly for the time interval to end. However, we can eliminate that requirement by having the timer produce an interrupt. Now the processor can perform other tasks while the timer counts.

To produce an interrupt from timer 1, all we must do is set bit 6 (see Figure D–7) of the interrupt enable register. Program D–4 is an initialization routine that we will use throughout the rest of this laboratory. It loads the stack pointer, the data direction registers, timer 1's counters, the interrupt enable register, and the AIM's interrupt vector. Program D–5 produces an interrupt after 10 ms (10,000 clock cycles: 2710 hex); note that we must enable and disable the CPU interrupt and clear the timer interrupt flag.

```
;
; GENERALIZED INITIALIZATION ROUTINE
;
        LDX   #$7F              ; INITIALIZE USER STACK POINTER
        TXS
        LDA   #0                ; MAKE PORT A INPUT
        STA   $A003
        LDA   #$FF
        STA   $A002             ; MAKE PORT B OUTPUT
        STA   $A000             ; TURN OFF LEDS INITIALLY
        LDA   #$10              ; SET TIMER FOR 10 MS
        STA   $A004
        LDA   #$27
        STA   $A005
        LDA   #%11000000        ; ENABLE TIMER 1 INTERRUPT
        STA   $A00E
        LDA   #$80              ; LOAD INTERRUPT VECTOR
        STA   $A400
        LDA   #$02
        STA   $A401
```

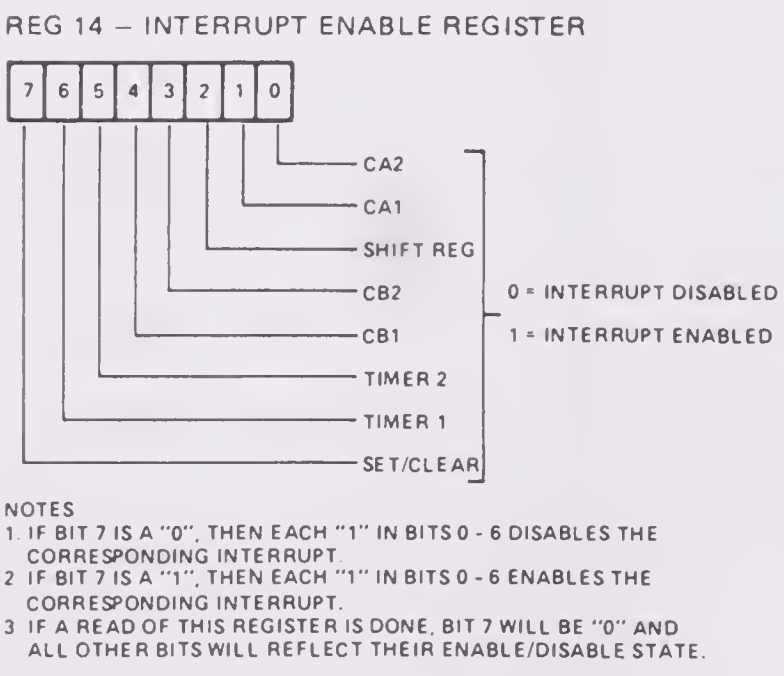


FIGURE D–7. VIA interrupt enable register (IER). (Reprinted courtesy of Rockwell International, Semiconductor Products Division, Newport Beach, CA.)

PROGRAM D–4

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0200	A2	LDX	#7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#00
0204	00		
0205	8D	STA	A003
0206	03		
0207	A0		
0208	A9	LDA	#FF
0209	FF		
020A	8D	STA	A002
020B	02		
020C	A0		
020D	8D	STA	A000
020E	00		
020F	A0		
0210	A9	LDA	#10
0211	10		
0212	8D	STA	A004
0213	04		
0214	A0		
0215	A9	LDA	#27
0216	27		
0217	8D	STA	A005
0218	05		
0219	A0		
021A	A9	LDA	#C0
021B	C0		
021C	8D	STA	A00E
021D	0E		
021E	A0		
021F	A9	LDA	#80
0220	80		
0221	8D	STA	A400
0222	00		
0223	A4		
0224	A9	LDA	#02
0225	02		
0226	8D	STA	A401
0227	01		
0228	A4		

PROGRAM D–5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0229	A9		LDA	#00
022A	00			
022B	8D		STA	A00B
022C	0B			
022D	A0			
022E	A9		LDA	#00
022F	00			
0230	85		STA	40
0231	40			
0232	58		CLI	
0233	A5	WTTIM	LDA	40
0234	40			
0235	F0		BEQ	0233
0236	FC			
0237	78		SEI	
0238	00		BRK	
0280	E6		INC	40
0281	40			
0282	2C		BIT	A004
0283	04			
0284	A0			
0285	40		RTI	

```
;
; INTERRUPT AFTER 10 MS (10,000 CLOCK CYCLES = 2710 HEX)
;
            LDA   #0
            STA   $A00B       ; TIMER 1 IN ONE-SHOT MODE
            LDA   #0          ; CLEAR READY FLAG
            STA   $40
            CLI               ; ENABLE CPU INTERRUPT
WTTIM       LDA   $40         ; IS INTERVAL OVER?
            BEQ   WTTIM       ; NO, WAIT
            SEI               ; YES, DISABLE CPU INTERRUPT
            BRK

            *=$0280           ; TIMER SERVICE ROUTINE
            INC   $40         ; SET READY FLAG
            BIT   $A004       ; CLEAR TIMER INTERRUPT FLAG
            RTI
```

BIT clears the timer interrupt flag without changing the accumulator. It affects only the status register, which RTI restores automatically.

PROBLEM D–7

Write a program that waits for the number of interrupts in 0030 and then lights the LEDs at port B for the number of interrupts in 0031.

Example:

(0030) = 50
(0031) = 30

Result: The program waits for 50 hex (80 decimal) interrupts and then lights the LEDs for 30 hex (48 decimal) interrupts. Finally, it turns the LEDs off. Thus the LEDs are off for 0.5 s and then on for 0.3 s.

REAL-TIME CLOCK

A real-time clock simply produces interrupts continuously. The computer can keep time by counting them. To create a real-time clock, we operate timer 1 in the free-running mode by setting bit 6 of the auxiliary control register. Program D–5 keeps a clock count in 0040 if we put an endless loop at the end of its main part. Now 0040 indicates how many hundredths of a second have elapsed. Run the revised program a few times (Program D–6 is the mnemonic-entry version) and see what values you find in 0040 when you reset the computer.

```
        LDA   #%01000000      ; TIMER 1 IN FREE-RUNNING MODE
        STA   $A00B
        LDA   #0              ; CLEAR TIMER COUNT TO START
        STA   $40
        CLI                   ; ENABLE CPU INTERRUPT
HERE    JMP   HERE            ; WAIT AND COUNT
```

You should disable the CPU interrupt and clear the timer before returning control to the monitor. Otherwise, the timer will run indefinitely.

Other programs can use the clock count to measure time, much as a person uses a watch. If your watch now reads 2:33, for example, you can wait 15 minutes by adding 15 to 2:33 and waiting for your watch to read 2:48. Similarly, a program can produce a delay by reading the clock count, adding the required number of clock periods, and waiting until the sum and the count are equal. The following instructions at the end of Program D–6 make the computer wait 50 ms (five clock periods). Here the initial count is zero, so no addition is necessary.

PROGRAM D–6

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0229	A9		LDA	#40
022A	40			
022B	8D		STA	A00B
022C	0B			
022D	A0			
022E	A9		LDA	#00
022F	00			
0230	85		STA	40
0231	40			
0232	58		CLI	
0233	4C	HERE	JMP	0233
0234	33			
0235	02			
0280	E6		INC	40
0281	40			
0282	2C		BIT	A004
0283	04			
0284	A0			
0285	40		RTI	

```
                LDA     #5
WAIT5           CMP     $40             ; HAS CLOCK COUNT REACHED 5?
                BNE     WAIT5           ; NO, WAIT
                SEI                     ; YES, DISABLE CPU INTERRUPT
                BRK
```

The mnemonic-entry version of the changes is

0233	A9		LDA	#05
0234	05			
0235	C5	WAIT5	CMP	40
0236	40			
0237	D0		BNE	0235
0238	FC			
0239	78		SEI	
023A	00		BRK	

Enter and run the modified program. Make it wait ten clock periods.

PROBLEM D–8

Make the modified program wait five clock periods and then light the LEDs for ten clock periods. Change it to produce the following on-off periods:

a. OFF–10
   ON–5
b. OFF–1
   ON–1

PROBLEM D–9

Make the answer to Problem D–8 operate continuously, turning the LEDs on and off according to the duty cycle in 0030 and 0031. To determine when a time interval ends, add its length to 0040. Does it matter if this produces a carry?

Run the program for the following test cases and describe what happens.

a. OFF–(0030) = 01
   ON–(0031) = 01
b. OFF–(0030) = 04
   ON–(0031) = 1C
c. OFF–(0030) = 10 (hex)
   ON–(0031) = 10 (hex)
d. OFF–(0030) = 1C
   ON–(0031) = 04

PROBLEM D–10

Make the program from Problem D–9 turn the LEDs on and off according to the following duty cycle for each iteration:

OFF – 10 (hex)
ON – 20 (hex)
OFF – 40 (hex)
ON – 80 (hex)

Industrial and process controllers often have complex duty cycles with many variations in length and amplitude.

To make the program shorter and more general, put the ON–OFF values in a table. For example, you could use 0380 through 0383 as follows:

(0380) = 10 (first OFF period)
(0381) = 20 (first ON period)
(0382) = 40 (second OFF period)
(0383) = 80 (second ON period)

LONGER TIME INTERVALS

We can handle longer time intervals by using more memory locations for the clock count. The following service routine uses 0040 and 0041 (MSBs in 0041). The main program should clear both locations before enabling interrupts.

```
            *=$0280
            PHA                 ; SAVE ACCUMULATOR
            INC   $40           ; INCREMENT LSB OF CLOCK COUNT
            BNE   CLRINT
            INC   $41           ; AND MSB IF NECESSARY
CLRINT      BIT   $A004         ; CLEAR TIMER INTERRUPT FLAG
            PLA                 ; RESTORE ACCUMULATOR
            RTI
```

Enter and run this program, placing an endless loop at the end of the main program. Program D–7 is the mnemonic-entry version. Let it run for a while and see what you find in 0040 and 0041 when you reset the computer.

PROGRAM D–7

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0280	48		PHA	
0281	E6		INC	40
0282	40			
0283	D0		BNE	0287
0284	02			
0285	E6		INC	41
0286	41			
0287	2C	CLRINT	BIT	A004
0288	04			
0289	A0			
028A	68		PLA	
028B	40		RTI	

PROBLEM D–11

Write a main program that uses Program D–7 to turn all the LEDs at port B off for 300 (012C) clock periods and then on for 400 (0190) clock periods. Change your program to produce the following on-off periods.

a. OFF–400 (0190 hex)
   ON–300 (012C hex)
b. OFF–300 (012C hex)
   ON–300 (012C hex)

PROBLEM D–12

Make the program from Problem D–11 operate continuously, turning the LEDs on and off according to 0030 and 0031 (OFF period) and 0032 and 0033 (ON period). Try the following test cases:

a. (0030) = 2C (012C hex = 300 decimal)
   (0031) = 01
   (0032) = 58 (0258 hex = 600 decimal)
   (0033) = 02
b. (0030) = F4 (01F4 hex = 500 decimal)
   (0031) = 01
   (0032) = F4 (01F4 hex = 500 decimal)
   (0033) = 01

KEEPING TIME IN STANDARD UNITS

We can easily write a service routine (Program D–8 is the mnemonic-entry version) that keeps time in seconds and minutes. As before, the main program must clear the counter locations initially.

PROGRAM D–8

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0280	48	PHA
0281	E6	INC 40
0282	40	
0283	A5	LDA 40
0284	40	
0285	38	SEC
0286	E9	SBC #64
0287	64	
0288	D0	BNE 02A0
0289	16	
028A	85	STA 40
028B	40	
028C	E6	INC 41
028D	41	
028E	A5	LDA 41
028F	41	
0290	E9	SBC #3C
0291	3C	
0292	D0	BNE 02A0
0293	0C	
0294	85	STA 41

PROGRAM D–8 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0295	41			
0296	E6		INC	42
0297	42			
0298	A5		LDA	42
0299	42			
029A	E9		SBC	#3C
029B	3C			
029C	D0		BNE	02A0
029D	02			
029E	85		STA	42
029F	42			
02A0	2C	ENDINT	BIT	A004
02A1	04			
02A2	A0			
02A3	68		PLA	
02A4	40		RTI	

```
          * = $0280
          PHA             ; SAVE ACCUMULATOR
          INC   $40       ; UPDATE HUNDREDTHS OF SECONDS
          LDA   $40
          SEC             ; IS THERE A CARRY TO SECONDS?
          SBC   #100
          BNE   ENDINT    ; NO, DONE (C=1 IF NO BRANCH)
          STA   $40       ; YES, MAKE HUNDREDTHS ZERO
          INC   $41       ; UPDATE SECONDS
          LDA   $41
          SBC   #60       ; IS THERE A CARRY TO MINUTES?
          BNE   ENDINT    ; NO, DONE (C=1 IF NO BRANCH)
          STA   $41       ; YES, MAKE SECONDS ZERO
          INC   $42       ; UPDATE MINUTES
          LDA   $42
          SBC   #60       ; IS THERE A CARRY TO HOURS?
          BNE   ENDINT    ; NO, DONE
          STA   $42       ; YES, MAKE MINUTES ZERO
ENDINT    BIT   $A004     ; CLEAR TIMER INTERRUPT FLAG
          PLA             ; RESTORE ACCUMULATOR
          RTI
```

PROBLEM D–13

Make Program D–8 keep time as pairs of decimal digits in 0040, 0041, and 0042. Use the decimal mode, but remember that INC and DEC always produce binary results.

PROBLEM D–14

Write a main program that uses Program D–8 to turn all the LEDs off for 1 min and 30 s and then on for 1 min and 15 s.

PROBLEM D–15

We can use tables to produce complex timing sequences. Write a program that turns the LEDs at port B on and off according to the following schedule. Each entry is the length of a period in seconds, and the final zero is a terminator.

```
(0380) = 0A    (first OFF period, 10 s)
(0381) = 0F    (first ON period, 15 s)
(0382) = 14    (second OFF period, 20 s)
(0383) = 0F    (second ON period, 15 s)
(0384) = 0A    (third OFF period, 10 s)
(0385) = 14    (third ON period, 20 s)
(0386) = 00    (terminator)
```

The LEDs should be off for 10 s (0A hex), on for 15 s, off for 20 s, on for 15 s, off for 10 s, and on for 20 s.

PROBLEM D–16

We can extend Problem D–15 to put the LEDs in other states besides all on and all off. Write a program that operates the LEDs according to the following table. Each entry consists of a length in seconds followed by a data value for the LEDs. The final zero is a terminator. Turn all the LEDs off before concluding.

```
(0380) = 14    (first period, 20 s)
(0381) = 00    (all LEDs on)
(0382) = 1E    (second period, 20 s)
(0383) = 01    (all LEDs on except #0)
(0384) = 14    (third period, 20 s)
(0385) = 03    (all LEDs on except #0 and #1)
(0386) = 00    (terminator)
```

PROBLEM D–17

With Program D–8, what percentage of the processor's time is spent servicing the real-time clock? Determine both the maximum and the average percentages. Note that the 6502 takes seven clock cycles to respond to an interrupt. Furthermore, the AIM monitor takes 29 cycles to vector the interrupt through A400 and A401. How much would the percentages increase if the clock interrupted every 1 ms? What is the maximum clock frequency if we limit the service routine to 10% of the processor's time on the average? What if we limit it to 10% maximum during any clock period?

REAL-TIME OPERATING SYSTEMS

A real-time clock can satisfy many timing requirements. Tasks can be scheduled or suspended, delays can be produced, and real-time inputs and outputs can be handled. The programmer must, however, determine the order and priority of tasks and specify exactly how they use the clock.

A typical example application is a real-time monitoring system for process or industrial control. The system must collect data periodically (e.g., readings from sensors located at various points in a pipeline, tank, or reactor), respond immediately to alarms, and report status to a central computer. The times when alarms occur must be printed for permanent records. This system has many tasks to perform: data logging, alarm recognition, alarm recording, printing, and communications with the central computer. The priority of the tasks is critical. For example, if an alarm occurs while a record is being printed, the system must suspend the printing, mark when the alarm occurred, prepare a new record for later printing, and then resume the suspended task. A similar procedure is necessary if the central computer requests a report while the system is busy. The programmer must manage the computer's resources so that it does its work without missing any data, alarms, or requests for reports.

A real-time operating system removes much of the burden of task management from the programmer. This piece of packaged software schedules tasks, handles communications between them (e.g., its routine would let the alarm recording task send information about the alarm to the printer task), generates time intervals, and provides real-time interrupt control for I/O devices. The programmer must learn only how to use the operating system. The obvious advantages of such systems are that they can be purchased rather than written and that they provide standard procedures and formats.

To use a real-time operating system in the monitoring application, you would have to write programs in the proper form to handle data logging, alarm monitoring, printing, and status reporting. Each program (or *task*) would call subroutines (sometimes called *utilities*) from the operating system. The user tasks and the operating system would thus together control the monitoring system. Note that you could change one task (e.g., attach a new printer, allow more alarms, or add a local data buffer) without changing the other tasks. We would like to thank Bill Renwick of Kadak Products Ltd. (Vancouver, B.C., Canada) for suggesting this example and describing how it would run under Kadak's AMX Operating System.

KEY POINT SUMMARY

1. Programs can be made more flexible by allowing them to determine timing parameters from system inputs. The same program can then handle peripherals operating at different data rates.
2. A program can easily examine a clock line, synchronize with it, and measure its period as long as its frequency is much lower than the CPU clock frequency.
3. A programmable timer can replace a delay routine. It simply indicates when a starting value loaded into it has been counted down to zero. Programmable timers

make systems more flexible because they can operate in different modes under program control. However, these timers require careful use and documentation because there are no standards for their options or programming.

4. The 6522 VIA has two 16-bit timers. A program can start a timer at a specific count by loading data into its registers, less significant byte first. The timer indicates the exhaustion of the count by setting a bit in the interrupt flag register. Either timer can be used in an interrupt-driven mode by setting a bit in the interrupt enable register. The auxiliary control register determines the timers' operating modes.
5. Interrupts are a convenient way to handle timing. A real-time clock is a regular source of interrupts that can be counted for timing and scheduling. Time is specified in counts.
6. A real-time operating system performs scheduling, coordination, and communications on a real-time basis. It provides a standard supervisor for applications with real-time needs.

SERIAL INPUT/OUTPUT

PURPOSE

To learn how to use the microcomputer to send and receive serial data.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 360–363, 385–388, 420–427, 489–492.

L. A. LEVENTHAL, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11–103 to 11–122, 12–32 to 12–36.

L. A. LEVENTHAL and W. SAVILLE, *6502 Assembly Language Subroutines*, Osborne/McGraw-Hill, Berkeley, CA, 1982, pp. 428–439, 464–471, 480–489.

J. E. MCNAMARA, *Technical Aspects of Data Communications*, 2nd ed., Educational Services Department, Digital Equipment Corp., Maynard, MA, 1982, Chapters 1–3, 5, 15.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 14–16 (alphanumeric codes), 16–17 (parity), 48–49 and 51–52 (shift registers), 237–240 (asynchronous serial I/O), 240–244 (UARTs), 245–254 (6850 device), 255–259 (serial transmission standards), 279–281 (teletypewriters), 335–337 (shift instructions).

AIM 65 Monitor Program Listing, Dynatem, Irvine, CA, 1979, pp. 39, 46.

AIM 65 User's Guide, Dynatem, Irvine, CA, 1978, pp. 7–35, 9–28 to 9–35.

R6500 Microcomputer System Programming Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, Chapter 10.

WHAT YOU SHOULD LEARN

1. How to convert data between serial and parallel forms.
2. How to provide timing for serial communications.
3. How to generate and recognize start and stop bits.
4. How to detect false start bits using majority logic.
5. How to generate and check parity.

TERMS

ASCII American Standard Code for Information Interchange, a 7-bit character code widely used in computers and communications. Appendix 2 contains an ASCII table.

Error-correcting code a code that the receiver can use to correct errors in messages.

Error-detecting code a code that the receiver can use to detect errors in messages.

False start bit a start bit that does not last the minimum required amount of time.

Majority logic a logic function that is true when more than half its inputs are true.

Parallel more than one bit at a time.

Parity a 1-bit code that makes the total number of 1 bits in a word, including the parity bit, odd (odd parity) or even (even parity).

Protocol a set of conventions governing the format and timing of data transfers.

Serial one bit at a time.

Start bit a bit indicating the start of data transmission by an asynchronous device.

Stop bit a bit indicating the end of data transmission by an asynchronous device.

Universal asynchronous receiver/transmitter (UART) an LSI device that interfaces parallel systems to asynchronous serial peripherals.

6502 INSTRUCTIONS

ROL rotate left; shift the accumulator or a memory location left one bit as if bits 0 and 7 were connected through the CARRY (see Figure E–1).

ROR rotate right; shift the accumulator or a memory location right one bit as if bits 0 and 7 were connected through the CARRY (see Figure E–1).

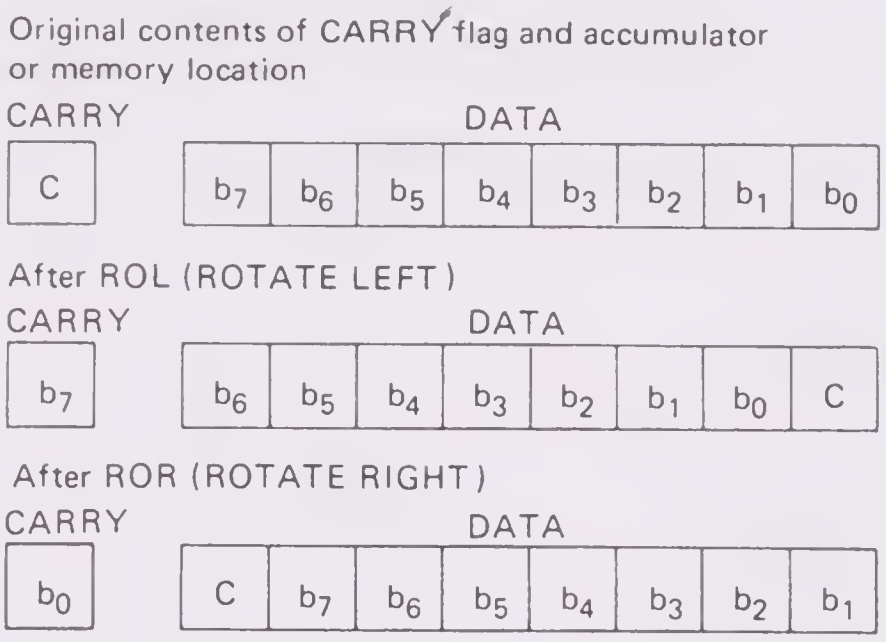


FIGURE E–1. 6502 shift instructions ROL and ROR.

SERIAL INTERFACING

Most communications equipment and many other peripherals transfer data 1 bit at a time (*serially*) rather than in larger units (i.e., in *parallel*). The serial approach is cheaper to implement since it requires only one data line. However, special interfaces are needed to connect serial peripherals to a computer that handles data in parallel.

This laboratory describes how to interface serial peripherals using software. We will show how to convert data between serial and parallel forms, provide timing, add and detect start and stop bits, and check and generate parity.

Serial interfaces allow many tradeoffs between hardware and software. A common alternative to the software interface is the universal asynchronous receiver/transmitter (UART). UARTs, which cost only a few dollars, do all the tasks we just mentioned. They are thus an attractive choice in all except the most cost-sensitive applications.

SERIAL/PARALLEL CONVERSION

Shift instructions are the keys to programs that convert data between parallel and serial forms. Since serial data transmission generally starts with bit 0, the following program places bit 0 of location 0060 on the LED at bit 7 of user VIA port B (LED 7, for short). Program E–1 is the mnemonic-entry version. LSR $60 moves a data bit from 0060 to the CARRY. ROR $A000 then sends it on to the output port. We have complemented

the data, so it appears on the LEDs in positive logic. Program E–1 turns all the LEDs off initially by storing FF in A000.

```
LDX    #$7F          ; INITIALIZE USER STACK POINTER
TXS
LDA    #$FF
STA    $A002         ; MAKE PORT B OUTPUT
STA    $A000         ; TURN OFF THE LEDS
EOR    $60           ; COMPLEMENT DATA (A IS FF)
STA    $60
LSR    $60           ; GET 1 BIT OF PARALLEL DATA
ROR    $A000         ; MOVE BIT TO SERIAL OUTPUT PORT
BRK
```

PROGRAM E–1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A2	LDX #7F
0201	7F	
0202	9A	TXS
0203	A9	LDA #FF
0204	FF	
0205	8D	STA A002
0206	02	
0207	A0	
0208	8D	STA A000
0209	00	
020A	A0	
020B	45	EOR 60
020C	60	
020D	85	STA 60
020E	60	
020F	46	LSR 60
0210	60	
0211	6E	ROR A000
0212	00	
0213	A0	
0214	00	BRK

Program E–1 transmits 1 bit. Execute it eight times starting with (0060) = AA (10101010 binary). After the first time, start at 020F to skip the initialization instructions. LED 7 should alternate between off and on, since the data consists of alternating 0 and 1 bits, starting with a 0 in bit 0. ROR $A000 shifts the previous serial outputs right so you can see all the bits.

PROBLEM E–1

Make Program E–1 use LED 0 as the serial output. The data now appears on the LEDs in reverse order, starting with bit 0 at the far left.

Converting inputs from serial to parallel is also simple. The following program (Program E–2 is a mnemonic-entry version) fetches a serial input from bit 7 of port A and combines it with the data in 0061. We assume that bit 0 is received first. Executing Program E–2 repeatedly makes the data bits move right on the LEDs and end up in their normal order.

Clear 0061 initially and execute Program E–2 eight times to assemble a data byte. After the first time, start at 020B to skip the initialization. Vary the switch position to end up with (0061) = AA.

```
LDX   #$7F            ; INITIALIZE USER STACK POINTER
TXS
LDA   #$FF
STA   $A002           ; MAKE PORT B OUTPUT
STA   $A000           ; TURN OFF THE LEDS
ASL   $A001           ; MOVE SERIAL INPUT TO CARRY
ROR   $61             ; AND COMBINE WITH PREVIOUS INPUTS
LDA   $61             ; SHOW DATA ON LEDS
EOR   #$FF            ; IN POSITIVE LOGIC
STA   $A000
BRK
```

PROGRAM E–2

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A2	LDX #7F
0201	7F	
0202	9A	TXS
0203	A9	LDA #FF
0204	FF	
0205	8D	STA A002
0206	02	
0207	A0	
0208	8D	STA A000
0209	00	
020A	A0	
020B	0E	ASL A001
020C	01	
020D	A0	
020E	66	ROR 61
020F	61	

PROGRAM E–2 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0210	A5	LDA 61
0211	61	
0212	49	EOR #FF
0213	FF	
0214	8D	STA A000
0215	00	
0216	A0	
0217	00	BRK

PROBLEM E–2

Make Program E–2 start with data bit 7 and use switch 0 as the serial input.

GENERATING BIT RATES

In real applications, the computer must wait between bits. One way to do this is with a delay routine. The next program (see Program E–3 for a mnemonic-entry version) uses a routine that waits for one-eighth of a second times the value in register Y. No registers change except the status. Run the program with (0060) = AA and with (0060) = 55.

```
        LDX   #$7F      ; INITIALIZE USER STACK POINTER
        TXS
        LDA   #$FF
        STA   $A002     ; MAKE PORT B OUTPUT
        STA   $A000     ; TURN OFF THE LEDS
        EOR   $60       ; COMPLEMENT DATA (A IS FF)
        STA   $60
        LDY   #16       ; BIT TIME = 2 SECONDS
        LDX   #8        ; NUMBER OF BITS = 8
OUTB    LSR   $60       ; MOVE SERIAL OUTPUT TO CARRY
        ROR   $A000     ; AND ON TO LEDS
        JSR   DLYE      ; WAIT A BIT TIME
        DEX
        BNE   OUTB      ; COUNT BITS
        BRK

        *=$0300         ; DELAY 1/8 S TIMES (Y)
DLYE    PHA             ; SAVE ACCUMULATOR
        TXA             ; SAVE X REGISTER
        PHA
```

```
                TYA                 ; SAVE Y REGISTER
                PHA                 ; Y COUNT ENDS UP IN A
DLY1            LDY     #$7D        ; DELAY 1/8 SECOND
DLY2            LDX     #$C8
DLY3            DEX
                BNE     DLY3
                DEY
                BNE     DLY2
                SEC                 ; COUNT 1/8THS OF A SECOND
                SBC     #1
                BNE     DLY1
                PLA                 ; RESTORE Y REGISTER
                TAY
                PLA                 ; RESTORE X REGISTER
                TAX
                PLA                 ; RESTORE ACCUMULATOR
                RTS
```

PROGRAM E–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A2		LDX	#7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#FF
0204	FF			
0205	8D		STA	A002
0206	02			
0207	A0			
0208	8D		STA	A000
0209	00			
020A	A0			
020B	45		EOR	60
020C	60			
020D	85		STA	60
020E	60			
020F	A0		LDY	#10
0210	10			
0211	A2		LDX	#08
0212	08			
0213	46	OUTB	LSR	60
0214	60			
0215	6E		ROR	A000
0216	00			
0217	A0			

PROGRAM E–3 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0218	20		JSR	0300
0219	00			
021A	03			
021B	CA		DEX	
021C	D0		BNE	0213
021D	F5			
021E	00		BRK	
0300	48	DLYE	PHA	
0301	8A		TXA	
0302	48		PHA	
0303	98		TYA	
0304	48		PHA	
0305	A0	DLY1	LDY	#7D
0306	7D			
0307	A2	DLY2	LDX	#C8
0308	C8			
0309	CA	DLY3	DEX	
030A	D0		BNE	0309
030B	FD			
030C	88		DEY	
030D	D0		BNE	0307
030E	F8			
030F	38		SEC	
0310	E9		SBC	#01
0311	01			
0312	D0		BNE	0305
0313	F1			
0314	68		PLA	
0315	A8		TAY	
0316	68		PLA	
0317	AA		TAX	
0318	68		PLA	
0319	60		RTS	

PROBLEM E–3

Write a reception program that waits between bits using a 2-second delay. Assume that the data starts with bit 0 and comes from switch 7. Run the program, setting the switch to produce (0061) = AA. If you need more time, either increase DLYE's parameter or replace it with an RTS instruction and use the STEP mode. Don't use a breakpoint, since it will make the AIM step through the delay. This will take a very long time!

USING THE REAL-TIME CLOCK

We can also use a real-time clock to wait between bits. The following program initializes the clock and transmits a bit each time 0040 increases by 100 (i.e., at 1-s intervals, since the clock is 100 Hz). Program E–4 is the mnemonic-entry version. The clock service routine is from Program D–6.

Enter and run Program E–4 with (0060) = AA and with (0060) = 55. What happens if adding 100 to the current clock count produces a carry?

PROGRAM E–4

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A2	LDX #7F
0201	7F	
0202	9A	TXS
0203	A9	LDA #FF
0204	FF	
0205	8D	STA A002
0206	02	
0207	A0	
0208	8D	STA A000
0209	00	
020A	A0	
020B	45	EOR 60
020C	60	
020D	85	STA 60
020E	60	
020F	A9	LDA #00
0210	00	
0211	85	STA 40
0212	40	
0213	A9	LDA #40
0214	40	
0215	8D	STA A00B
0216	0B	
0217	A0	
0218	A9	LDA #10
0219	10	
021A	8D	STA A004
021B	04	
021C	A0	
021D	A9	LDA #27
021E	27	
021F	8D	STA A005
0220	05	
0221	A0	

PROGRAM E–4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0222	A9		LDA	#C0
0223	C0			
0224	8D		STA	A00E
0225	0E			
0226	A0			
0227	A9		LDA	#80
0228	80			
0229	8D		STA	A400
022A	00			
022B	A4			
022C	A9		LDA	#02
022D	02			
022E	8D		STA	A401
022F	01			
0230	A4			
0231	58		CLI	
0232	A2		LDX	#08
0233	08			
0234	46	OUTB	LSR	60
0235	60			
0236	6E		ROR	A000
0237	00			
0238	A0			
0239	A5		LDA	40
023A	40			
023B	18		CLC	
023C	69		ADC	#64
023D	64			
023E	C5	WTCLK	CMP	40
023F	40			
0240	D0		BNE	023E
0241	FC			
0242	CA		DEX	
0243	D0		BNE	0234
0244	EF			
0245	78		SEI	
0246	00		BRK	
0280	E6		INC	40
0281	40			
0282	2C		BIT	A004
0283	04			
0284	A0			
0285	40		RTI	

```
                LDX   #$7F              ; INITIALIZE USER STACK POINTER
                TXS
                LDA   #$FF
                STA   $A002             ; MAKE PORT B OUTPUT
                STA   $A000             ; TURN OFF THE LEDS
                EOR   $60
                STA   $60
                LDA   #0                ; CLEAR CLOCK COUNT INITIALLY
                STA   $40
                LDA   #%01000000        ; TIMER 1 IN FREE-RUNNING MODE
                STA   $A00B
                LDA   #$10              ; SET TIMER 1 FOR 10 MS
                STA   $A004
                LDA   #$27
                STA   $A005
                LDA   #%11000000        ; ENABLE TIMER 1 INTERRUPT
                STA   $A00E
                LDA   #$80              ; LOAD INTERRUPT VECTOR
                STA   $A400
                LDA   #$02
                STA   $A401
                CLI                     ; ENABLE CPU INTERRUPT
                LDX   #8                ; NUMBER OF BITS = 8
OUTB            LSR   $60               ; MOVE SERIAL OUTPUT TO CARRY
                ROR   $A000             ; AND ON TO LEDS
                LDA   $40               ; GET STARTING CLOCK COUNT
                CLC                     ; CALCULATE TARGET VALUE
                ADC   #100
WTCLK           CMP   $40               ; TARGET VALUE REACHED?
                BNE   WTCLK             ; NO, WAIT
                DEX                     ; YES, COUNT BITS
                BNE   OUTB
                SEI                     ; DISABLE CPU INTERRUPT
                BRK

                *=$0280                 ; CLOCK SERVICE ROUTINE
                INC   $40               ; INCREMENT CLOCK COUNT
                BIT   $A004             ; CLEAR TIMER 1 INTERRUPT
                RTI
```

PROBLEM E–4

Make the serial reception program wait for 100 clock interrupts between bits.

Sample cases

a. All inputs 1s (leave the switch open).
   Result: (0061) = FF

b. All inputs 0s (leave the switch closed).
Result: (0061) = 00

PROBLEM E–5

Make Program E–4 use Program D–8 as the service routine. Now time is kept in minutes, seconds, and hundredths of seconds. Make the time between bit outputs 1 s.

Remember that we have not stopped the clock or disabled timer 1's interrupt. You can do both by resetting the AIM, since that clears all VIA registers.

START AND STOP BITS

So far, we have assumed that reception can occur at any time. Of course, in practice, the receiver (computer or peripheral) must determine when the data starts and ends.

One approach is to put markers around the data. Figure E–2 shows a popular format with a start bit (0) ahead of each 8-bit character and two stop bits (1s) afterward. Note that the data line is normally 1.

Although this approach is simple and easy to implement, it does have disadvantages. Noise can produce false start bits; we will discuss methods for detecting them later. Adding start and stop bits reduces the actual data rate (since more bits must be transmitted) and increases the overhead for each character. An alternative is to group the characters into blocks and synchronize only on a block-by-block basis. Most protocols (see J. E. McNamara's *Technical Aspects of Data Communications*) use this approach to increase speed and reliability.

We can easily modify Program E–3 to produce a start bit. We must clear CARRY initially and shift 0060 at the end of the loop instead of at the beginning. Now the program transmits a 0 first and data bit 0 second. Of course, the bit count must be 9 instead of 8. Program E–5 is the mnemonic-entry version.

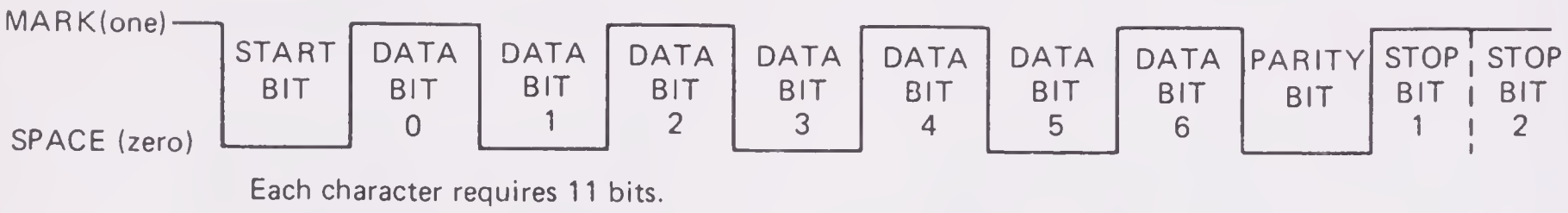


FIGURE E–2. Serial data format with a start bit and two stop bits.

```
LDX    #$7F          ; INITIALIZE USER STACK POINTER
TXS
LDA    #$FF
STA    $A002         ; MAKE PORT B OUTPUT
STA    $A000         ; TURN OFF THE LEDS
EOR    $60           ; COMPLEMENT THE DATA
STA    $60
```

PROGRAM E–5

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0200	A2	LDX	#7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#FF
0204	FF		
0205	8D	STA	A002
0206	02		
0207	A0		
0208	8D	STA	A000
0209	00		
020A	A0		
020B	45	EOR	60
020C	60		
020D	85	STA	60
020E	60		
020F	A0	LDY	#10
0210	10		
0211	A2	LDX	#09
0212	09		
0213	18	CLC	
0214	6E	ROR	A000
0215	00		
0216	A0		
0217	20	JSR	0300
0218	00		
0219	03		
021A	46	LSR	60
021B	60		
021C	CA	DEX	
021D	D0	BNE	0214
021E	F5		
021F	00	BRK	

```
                LDY   #16          ; BIT TIME = 2 SECONDS
                LDX   #9           ; NUMBER OF BITS = 9
                CLC                ; FORM START BIT
OUTB            ROR   $A000        ; MOVE SERIAL OUTPUT TO LEDS
                JSR   DLYE         ; WAIT A BIT TIME
                LSR   $60          ; MOVE NEXT SERIAL OUTPUT TO CARRY
                DEX                ; COUNT BITS
                BNE   OUTB
                BRK
```

Enter Program E–5 into memory and run it with (0060) = 00 and with (0060) = AA. The start bit appears as a light in front of the actual data, since 0 lights an LED. By making the following changes to Program E–5, we can also generate two stop bits.

1. Make the bit count 11 instead of 9.
2. Replace LSR $60 with SEC (SET CARRY), ROR $60. This sequence automatically shifts 1s in at the left as it shifts the data right. The 1s form the stop bits.

PROBLEM E–6

Write and run a transmission program that generates a start bit and two stop bits. How could you make your program produce one stop bit instead of two? Many terminals use a 10-bit format with one stop bit.

Receiving data with start and stop bits is more difficult than transmitting it. The next program (see Program E–6 for a mnemonic-entry version) first detects the 1-to-0 transition that indicates the beginning of a start bit. It then waits half a bit time to center the reception. This delay makes the computer read the data bits near the centers of the pulses rather than at the edges, thus avoiding the transition areas. Centering also makes precise bit times unnecessary, since a slight drift from the center does not matter.

```
                LDX   #$7F          ; INITIALIZE USER STACK POINTER
                TXS
WTSTB           LDA   $A001         ; WAIT FOR A START BIT
                BMI   WTSTB
                LDY   #8            ; WAIT HALF BIT TIME TO CENTER
                JSR   DLYE
                LDY   #16           ; BIT TIME = 2 SECONDS
                LDX   #8            ; NUMBER OF BITS = 8
INBIT           JSR   DLYE          ; WAIT A BIT TIME
                ASL   $A001         ; MOVE SERIAL INPUT TO CARRY
                ROR   $61           ; AND COMBINE WITH PREVIOUS DATA
                DEX                 ; COUNT BITS
                BNE   INBIT
                BRK
```

Enter and run Program E–6. The STEP mode will give you time to set the switch for each data value. The program will not leave the initial loop until you close the switch and form a start bit. Remember to replace the delay routine with an RTS instruction in 0300. Set the switch after the computer executes JSR DLYE but before it executes ASL $A001. Try the following sample cases:

1. Start with the switch closed and move it each time the computer enters DLYE.
   Result: (0061) = 55 hex
2. Close the switch to form the start bit and then immediately open it and leave it open.
   Result: (0061) = FF

PROGRAM E–6

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A2		LDX	#7F
0201	7F			
0202	9A		TXS	
0203	AD	WTSTB	LDA	A001
0204	01			
0205	A0			
0206	30		BMI	0203
0207	FB			
0208	A0		LDY	#08
0209	08			
020A	20		JSR	0300
020B	00			
020C	03			
020D	A0		LDY	#10
020E	10			
020F	A2		LDX	#08
0210	08			
0211	20	INBIT	JSR	0300
0212	00			
0213	03			
0214	0E		ASL	A001
0215	01			
0216	A0			
0217	66		ROR	61
0218	61			
0219	CA		DEX	
021A	D0		BNE	0211
021B	F5			
021C	00		BRK	

PROBLEM E–7

Make Program E–6 check if two stop bits follow the data. The revised program should set 0062 to 00 if the two stop bits are present and to FF otherwise. Lack of the proper number of stop bits is called a *framing error*.

DETECTING FALSE START BITS

Many errors occur in communications, particularly over noisy connections (such as telephone lines) or long distances. One problem is that noise may make the input briefly 0, thus producing a *false start bit*. The receiver can tell a short noise pulse from a start bit

by sampling the line several times and requiring that a majority of the samples be 0s. This approach is called *majority logic*; it works like voting—the value that occurs most often "wins."

The following program samples the data at one-quarter, one-half, and three-quarters of a bit time after the initial detection of a 0. It requires that at least two samples be 0s. Program E–7 is the mnemonic-entry version. If the computer accepts the start bit, it must wait one-quarter of a bit time to reach the beginning of data bit 0 (see Figure E–2).

Run Program E–7 in the STEP mode so that you can control the switch and trace the sampling. Remember to replace DLYE with an RTS instruction in location 0300. Try the following cases (starting with the switch closed to form the start bit):

1. Move the switch every time the computer executes JSR DLYE (reaching 0300). The sample values will be 1, 0, and 1. Since only one is 0, the computer should reject the start bit and return to address WTSTB (0205).
2. Leave the switch closed until the computer enters DLYE for the second time. Then move it after each entry. The sample values will be 0, 1, and 0. Since two are 0, the computer should accept the start bit and return to the monitor.

```
            LDX   #$7F          ; INITIALIZE USER STACK POINTER
            TXS
            LDY   #4            ; SET DELAY FOR 1/4 BIT TIME
WTSTB       LDA   $A001         ; WAIT FOR A START BIT
            BMI   WTSTB
            LDX   #0            ; CLEAR ZERO COUNT TO START
            LDA   #3            ; NUMBER OF SAMPLES = 3
            STA   $30
CHBIT       JSR   DLYE          ; WAIT 1/4 BIT TIME
            LDA   $A001         ; IS DATA ZERO?
            BMI   CSAMP
            INX                 ; YES, INCREMENT ZERO COUNT
CSAMP       DEC   $30           ; COUNT SAMPLES
            BNE   CHBIT
            CPX   #2            ; WAS MAJORITY OF SAMPLES ZERO?
            BCC   WTSTB         ; NO, FALSE START BIT
            JSR   DLYE          ; YES, WAIT 1/4 BIT TIME TO FINISH
            BRK
```

PROGRAM E–7

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
0200	A2	LDX #7F
0201	7F	
0202	9A	TXS
0203	A0	LDY #04
0204	04	

PROGRAM E–7 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0205	AD	WTSTB	LDA	A001
0206	01			
0207	A0			
0208	30		BMI	0208
0209	FB			
020A	A2		LDX	#00
020B	00			
020C	A9		LDA	#03
020D	03			
020E	85		STA	30
020F	30			
0210	20	CHBIT	JSR	0300
0211	00			
0212	03			
0213	AD		LDA	A001
0214	01			
0215	A0			
0216	30		BMI	0219
0217	01			
0218	E8		INX	
0219	C6	CSAMP	DEC	30
021A	30			
021B	D0		BNE	0210
021C	F3			
021D	E0		CPX	#02
021E	02			
021F	90		BCC	0205
0220	E4			
0221	20		JSR	0300
0222	00			
0223	03			
0224	00		BRK	

PROBLEM E–8

Revise Program E–7 to check the input at intervals of one-eighth of a bit time. At least four of the seven samples must be 0 to accept the start bit. The program should wait for the end of the start bit.

PROBLEM E–9

Write a reception program that checks each bit at one-fourth, one-half, and three-fourths of a bit time and determines the actual value by majority logic. That is, the bit

value is the value of at least two samples. Make the program start at the beginning of data bit 0, assuming that the initialization routine has detected a start bit but has not centered the reception.

GENERATING AND CHECKING PARITY

Still another way to avoid errors is to add error-detecting or -correcting codes to the data. These codes show whether the data was received correctly and, if not, where the errors were; they contain no additional information and thus reduce the rate at which actual data can be sent.

Parity is a simple error-detecting code. It is a single bit added to each character, which makes the total number of 1 bits even (if even parity) or odd (if odd parity). Note the following examples:

1. Data = 01101101: even parity = 1, since the data contains an odd number of 1 bits (5).
2. Data = 00010001: even parity = 0, since the data contains an even number of 1 bits (2).

Parity has the following features:

1. It allows the receiver to detect single but not double errors. Two erroneous bits result in the same parity as the correct data.
2. It does not allow for error correction. If the parity is wrong, the receiver knows that an error occurred but not which bit is wrong. All the receiver can do is request retransmission.

One way to generate parity is to count the 1 bits. The least significant bit of the count is 1 if the data contains an odd number of 1 bits and 0 if it contains an even number. That bit is even parity, since it makes the total number of 1 bits (including itself) even. We can readily combine the counting of 1 bits with serial transmission (Program E–3). The following program (Program E–8 is the mnemonic-entry version) generates even parity and sends it after the data from 0060. Since Program E–8 does not complement the data (to avoid confusion in generating parity), the serial outputs appear inverted (0 = light on, 1 = light off).

```
LDX    #$7F          ; INITIALIZE USER STACK POINTER
TXS
LDA    #$FF
STA    $A002         ; MAKE PORT B OUTPUT
STA    $A000         ; TURN OFF THE LEDS
LDA    #0            ; START PARITY AT ZERO
STA    $40
LDY    #16           ; BIT TIME = 2 SECONDS
```

```
            LDX   #8          ; NUMBER OF DATA BITS = 8
OUTB        LSR   $60         ; MOVE SERIAL OUTPUT TO CARRY
            BCC   SENDB       ; IS SERIAL OUTPUT 1?
            INC   $50         ; YES, ADD 1 TO PARITY
SENDB       ROR   $A000       ; SEND SERIAL OUTPUT TO LEDS
            JSR   DLYE        ; WAIT A BIT TIME
            DEX               ; COUNT DATA BITS
            BNE   OUTB
            LSR   $60         ; PARITY = LSB OF NUMBER OF 1 BITS
            ROR   $A000       ; TRANSMIT EVEN PARITY
            JSR   DLYE        ; WAIT A BIT TIME
            BRK
```

PROGRAM E–8

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A2		LDX	#7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#FF
0204	FF			
0205	8D		STA	A002
0206	02			
0207	A0			
0208	8D		STA	A000
0209	00			
020A	A0			
020B	A9		LDA	#00
020C	00			
020D	85		STA	50
020E	50			
020F	A0		LDY	#10
0210	10			
0211	A2		LDX	#08
0212	08			
0213	46	OUTB	LSR	60
0214	60			
0215	90		BCC	0219
0216	02			
0217	E6		INC	50
0218	50			
0219	6E	SENDB	ROR	A000
021A	00			
021B	A0			
021C	20		JSR	0300
021D	00			
021E	03			

PROGRAM E–8 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)
021F	CA	DEX
0220	D0	BNE 0213
0221	F1	
0222	46	LSR 50
0223	50	
0224	6E	ROR A000
0225	00	
0226	A0	
0227	20	JSR 0300
0228	00	
0229	03	
022A	00	BRK

Enter and run Program E–8 for the following examples:

1. (0060) = 41 (ASCII A)
   Result: Even parity bit = 0, since the data has two 1 bits (41 hex = 01000001 binary).
2. (0060) = 43 (ASCII C)
   Result: Even parity bit = 1, since the data has three 1 bits (43 hex = 01000011 binary).

The results of Program E–8 are confusing, since bit 0 has been lost off the end. You can restore positive logic by complementing the final data as follows:

022A	AD	LDA A000
022B	00	
022C	A0	
022D	49	EOR #FF
022E	FF	
022F	8D	STA A000
0230	00	
0231	A0	
0232	00	BRK

Regardless of whether you complement the data, the final values on the LEDs should be, starting with bit 7: even parity bit, data bit 7, data bit 6, data bit 5, data bit 4, data bit 3, data bit 2, and data bit 1. In example 1 above, the values are 0, 0, 1, 0, 0, 0, 0, 0, since 41 hex = 01000001 binary. In example 2, the values are 1, 0, 1, 0, 0, 0, 0, 1, since 43 hex = 01000011 binary.

PROBLEM E–10

Many computers and peripherals use 7-bit ASCII characters and reserve bit 7 for parity. Make Program E–8 transmit 7-bit characters followed by even parity.

Examples:

a. (0060) = 41 (ASCII A)
   Result: Transmitted data is 41, since its parity is even.
b. (0060) = 43 (ASCII C)
   Result: Transmitted data is C3, since 43 has odd parity.

PROBLEM E–11

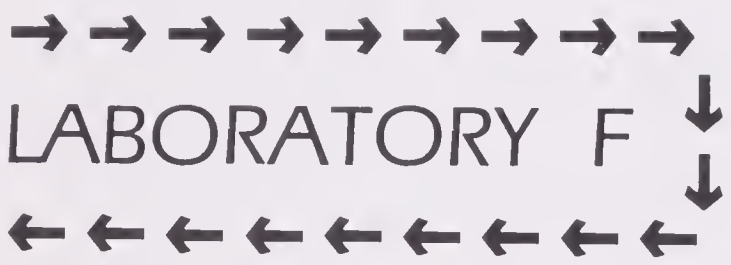
Write a serial reception program that checks parity. The program should place the parallel data in 0061 and set 0062 to 0 if the parity is even and to 1 if it is odd.

Examples:

a. Received data: 41 hex = 01000001 binary.
   Result: (0061) = 41 (parallel data)
   (0062) = 00, since 41 hex has an even number of 1 bits.
b. Received data: C1 hex = 11000001 binary.
   Result: (0061) = C1 (parallel data)
   (0062) = 01, since C1 hex has an odd number of 1 bits.

KEY POINT SUMMARY

1. Serial I/O requires such functions as parallel/serial conversion, the addition and detection of start and stop bits, clocking, and parity generation and checking. Either hardware such as UARTs or software can perform these functions.
2. Shift instructions can easily convert data between serial and parallel forms. Changes in the initial and final conditions are all that is needed to generate or detect start and stop bits.
3. Serial data can be clocked using software delay loops, programmable timers, or a real-time clock.
4. You can reduce the number of errors in serial communications by centering the reception, by sampling bits several times and using majority logic, or by including an error-detecting or -correcting code such as parity. Parity can be generated by counting the number of 1 bits in the data; even parity is the least significant bit of the count.



MICROCOMPUTER TIMING AND CONTROL

PURPOSE

To learn how the 6502 microprocessor executes instructions and how the addresses in the memory and I/O sections of 6502-based microcomputers are decoded.

PARTS REQUIRED

A dual-trace oscilloscope with a bandwidth of at least 5 MHz.

REFERENCE MATERIALS

L. A. LEVENTHAL, *Introduction to Microprocessors: Hardware, Software, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 284–316.

J. B. PEATMAN, *Digital Hardware Design*, McGraw-Hill, New York, 1980.

J. B. PEATMAN, *Microcomputer-based Design*, McGraw-Hill, New York, 1977, Chapter 3.

R. J. TOCCI and L. P. LASKOWSKI, *Microprocessors and Microcomputers: Hardware and Software*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 35–36 (tristate logic), 38–39 (clock signals), 39–50 (flip-flops), 52–53 (tristate registers), 53–58 (data bus), 58–59 (decoders), 60–62 (multiplexers and demultiplexers), 63–64 (memory devices), 65–73 (semiconductor memories), 73–79 (combining memory chips), 99–101 (instruction words), 101–105 (program example), 105–109 (operating cycles), 109–112 (instruction word formats), 127–137 (microcomputer structure), 137–142 (read and write operations), 142–146 (address allocation), 146–154 (address decoding), 160–162 (timing and control section).

AIM 65 User's Guide, Dynatem, Irvine, CA, 1978, Sections 7, 10.

R6500 Microcomputer System Hardware Manual, Rockwell International, Semiconductor Products Division, Newport Beach, CA, 1978, Sections 3, 4, Appendix A.

WHAT YOU SHOULD LEARN

1. How the 6502 executes instructions.
2. How to decode address lines to select memories and I/O devices.
3. What tradeoffs the designer can make between a computer's memory capacity and the number of parts required to decode addresses.
4. How to decode I/O addresses efficiently using linear select.

TERMS

Address bus the bus the CPU uses to select a memory location or I/O port.

Address space the total range of addresses to which a computer may refer.

Bus parallel lines that connect devices.

Bus contention more than one device trying to put data on a bus at the same time.

Decoder a device that produces unencoded outputs from coded inputs.

Instruction a group of bits that defines a computer operation.

Linear select using coded bus lines individually for selection rather than decoding them. Linear select requires no decoders but can address with n lines only n devices rather than 2^n.

Logic analyzer a piece of test equipment that detects, stores, and displays parallel digital signals.

Memory capacity the total number of memory locations that may be attached to a computer.

Tristate logic outputs with three states—high, low, and inactive (high-impedance or open-circuit). Inactive (disabled) outputs can be combined without gates.

SPECIAL PROBLEMS IN MICROCOMPUTER HARDWARE DESIGN

Describing the flow of signals in a microcomputer is not simple. Not only does data move in parallel (typically 8, 16, or 32 bits at a time), but the clock rate is high and few periodic sequences occur. The result is that microprocessor-based products are difficult to debug, maintain, and repair. In practice, engineers often buy board computers rather than designing their own, and companies often have service people replace entire circuit boards rather than trying to pinpoint a malfunction.

Of course, the designer must understand how a microcomputer operates and how its parts are connected. This laboratory can provide only a brief overview of hardware design. We assume that you have a dual-trace oscilloscope with a bandwidth of at least 5 MHz. Unfortunately, even a good oscilloscope is inadequate for design or troubleshooting. To diagnose hardware faults, you must be able to examine simultaneously the clock, data bus, address bus, and control signals. This requires a test instrument called a *logic analyzer* that can display many lines in a comprehensible form. Since logic analyzers are expensive, we will content ourselves with examining signals one at a time on an oscilloscope.

TIMING AND CONTROL FUNCTIONS

To design or understand a microcomputer, we must answer the following questions:

1. How does the processor transfer data to or from memory and I/O ports? Clearly timing is a critieal factor.
2. How does the processor decode and exceute instructions? Although this is an internal funetion, an understanding of it is important, since it governs the computer's operations.
3. How does the processor distinguish different types of cycles? The designer must use the processor's signals to control external hardware and monitor system operation.
4. How are memory addresses and I/O ports selected? Address lines and control signals must be decoded properly.
5. How can memories, I/O ports, and other deviees share system buses? Most miero-processors have a tristate data bus. Only one memory or input port is enabled at a time; the disabled ones do not affect the bus, since they are in the high-impedance state.

The microcomputer designer must, of course, eonsider economic and physical factors such as cost, speed, board size, and power consumption. Other important faetors include consistency with other applications and standards and how easy the computer is to test, expand, update, and maintain.

SYSTEM CLOCK

Let us now look at processor signals on the oscilloscope. Figure F–1 and Tables F–1 and F–2 give the pin assignments for the 6502 microprocessor and the AIM 65's Application

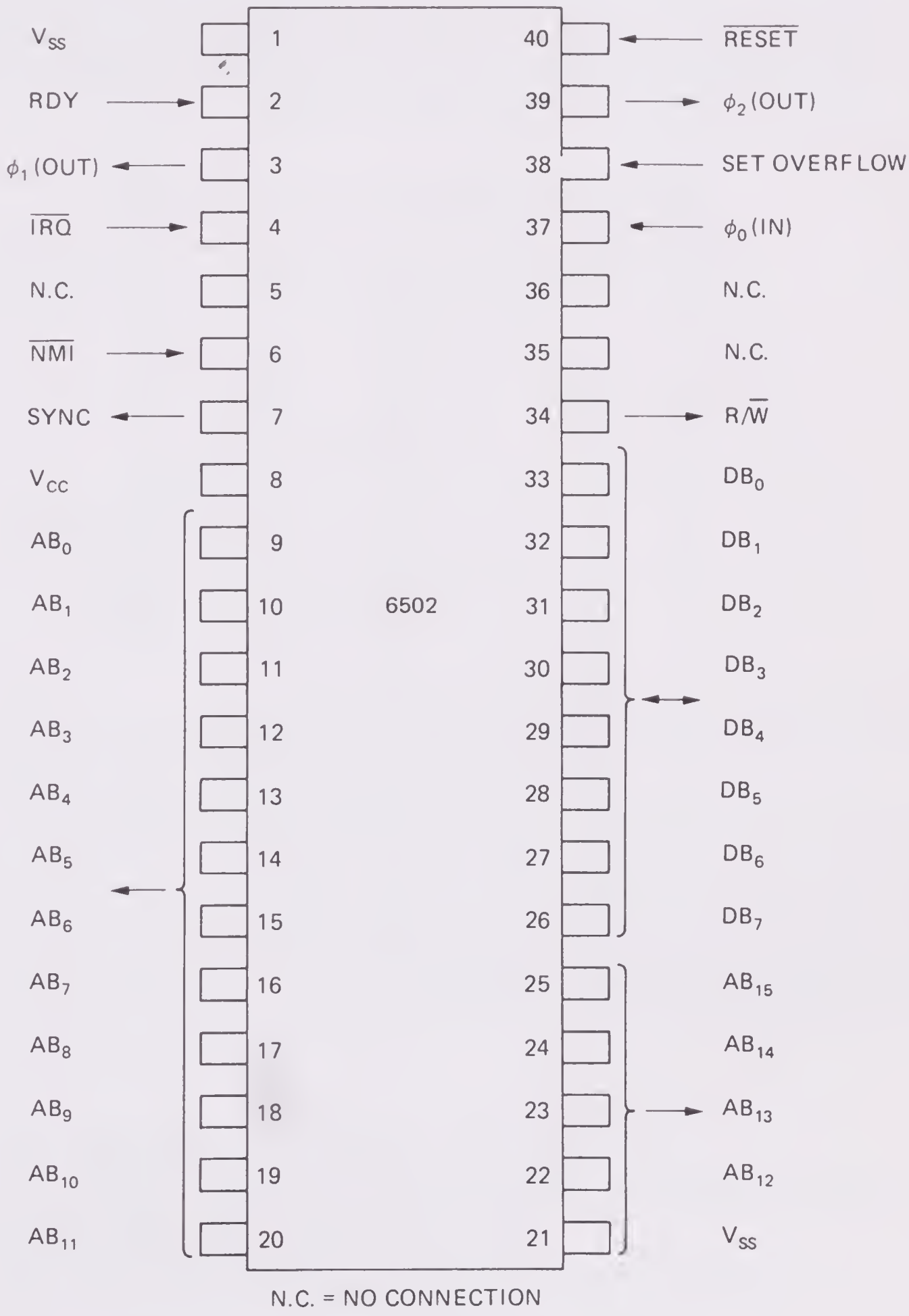


FIGURE F–1. Pin assignments for the 6502 microprocessor.

and Expansion Connectors. Attach the oscilloscope ground to pin 22 of the Expansion Connector. Put your oscilloscope in the CHOP mode so that it maintains timing relationships rather than retriggering when you switch channels; do not use the ALTERNATE mode.

Attach one probe to pin 3 (ϕ_1) of the Expansion Connector and the other to pin U (ϕ_2). These are the two phases of the system clock (see Figure F–2). During phase 1, the processor puts a new value on the address bus and determines all control signals. This is a setup period. During phase 2, the processor actually transfers data to or from memory or I/O ports; all control and address signals are stable.

TABLE F–1 AIM 65 APPLICATION (J1) CONNECTOR PIN ASSIGNMENTS

Pin	Signal	Pin	Signal
22		Z	
21	CA2	Y	SERIAL INPUT
20	CA1	X	
19	CB2	W	TAPE 1A
18	CB1	V	TAPE 2A
17	PB6	U	TTY PRINTER
16	PB5	T	TTY KEYBOARD
15	PB7	S	TTY PRINTER RETURN (+)
14	PA0	R	TTY KEYBOARD RETURN (+)
13	PB4	P	AUDIO OUT HIGH
12	PB3	N	+12V
11	PB2	M	AUDIO OUT LOW
10	PB1	L	AUDIO IN
9	PB0	K	
8	PA7	J	TAPE 2B
7	PA6	H	TAPE 2B RETURN
6	PA5	F	TAPE 1B
5	PA4	E	TAPE 1B RETURN
4	PA1	D	R/$\overline{W}$
3	PA2	C	ϕ2
2	PA3	B	
1	GND	A	+5V

Source: Dynatem, Irvine, Calif.

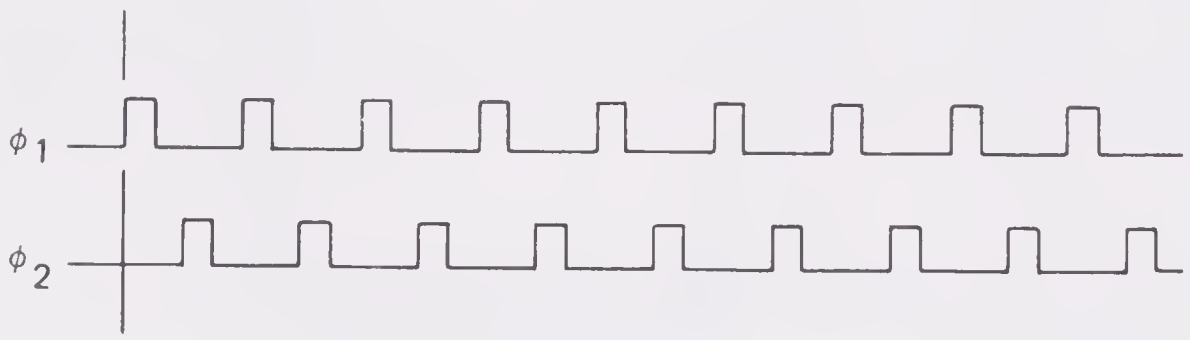


FIGURE F–2. Two-phase 6502 system clock.

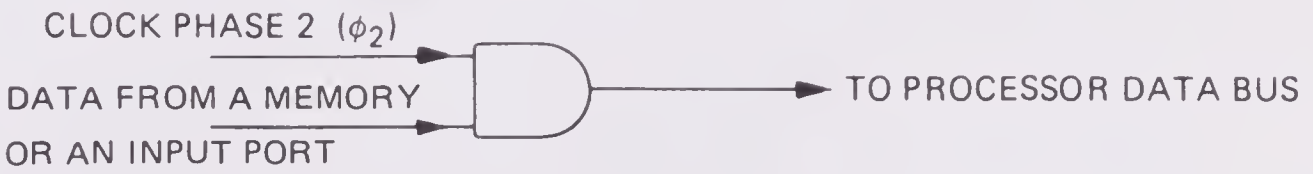
TABLE F–2 AIM 65 EXPANSION (J3) CONNECTOR ASSIGNMENTS

Pin	Signal	Pin	Signal
22	GND	Z	RAM R/$\overline{W}$
21	+5V	Y	$\overline{\phi 2}$
20	$\overline{CSA}$	X	TEST
19	$\overline{CS9}$	W	R/$\overline{W}$
18	$\overline{CS8}$	V	SYS R/$\overline{W}$
17	+12V	U	SYS $\phi 2$
16	−12V	T	A15
15	D0	S	A14
14	D1	R	A13
13	D2	P	A12
12	D3	N	A11
11	D4	M	A10
10	D5	L	A9
9	D6	K	A8
8	D7	J	A7
7	$\overline{RESET}$	H	A6
6	$\overline{NMI}$	F	A5
5	SET OVERFLOW	E	A4
4	$\overline{IRQ}$	D	A3
3	$\phi 1$	C	A2
2	RDY	B	A1
1	SYNC	A	A0

Source: Dynatem, Irvine, Calif.

Many devices, including the microprocessor, memories, input ports, and output ports, must share the data bus. Control signals must prevent *bus contention*, an attempt by more than one device to control the bus at a given time.

Bus contention can occur when the microprocessor changes its address outputs (that is, when it finishes reading one address and begins reading another). Since electronic devices have finite switching times, both the old address and the new one will be active briefly after the changeover. The designer can prevent contention during this period by gating the outputs from each memory or input port with clock phase 2 as shown in Figure F–3. Then the data bus is in use only during phase 2. Remember, however, that the processor changes its address outputs during phase 1; by the time phase 2 begins, the old address will be inactive.



(*Note:* the output of the gate is 0 unless clock phase 2 is high.)

FIGURE F–3. Gating clock phase 2 with data from memory or an input port.

TRACING INSTRUCTION EXECUTION

Put the following instruetion in locations 0200 through 0202:

```
HERE    JMP    HERE
```

This instruetion transfers eontrol to itself, thus produeing a repetitive sequenee of signals. Program F–1 is the mnemonie-entry version.

PROGRAM F–1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonie)
0200	4C	HERE	JMP 0200
0201	00		
0202	02		

Attaeh one probe to cloek phase 2 and the other to SYNC, pin 1 of the Expansion Conneetor. SYNC is aetive (high) during each cycle in whieh the proeessor is fetching an operation eode. Thus the rising edge on SYNC marks the beginning of eaeh exeeution of JMP. Note that SYNC is high about one-third of the time while Program F–1 is execut-ing. Thus the proeessor spends one-third of its time fetching JMP's operation code from memory and two-thirds fetehing the destination address and performing internal operations.

Now attaeh your second probe to address line AB_0 (pin A of the Expansion Conneetor). This line is high during the seeond eloek eyele of JMP, when the proeessor is fetching the less signifieant byte of the address from 0201. Similarly, AB_1 (pin B of the Expansion Conneetor) goes high during JMP's third eloek eyele, when the proeessor is fetching the more signifieant byte of the address from 0202.

The proeessor thus exeeutes instructions in a series of elock eyeles. Eaeh cycle eonsists of one phase used to ehange and stabilize addresses and one used to transfer data. Appendix A of the *R6500 Microcomputer System Hardware Manual* deseribes in detail the execution of all instruetions.

The 6502 executes JMP with absolute addressing in three eloek cycles as follows:

1. During the first eyele, it fetches the operation eode (4C hex). It does this by putting the program eounter (0200 hex) on the address bus and reading the data from memory. The proeessor adds 1 to the program counter after eaeh use.
2. During the seeond eycle, it fetches the less signifieant byte (00) of the destination address. Here again, the processor puts the program eounter (now 0201 hex) on the address bus and reads the data from memory. As in the first eyele, the proeessor adds 1 to the program counter, making its final value 0202.

3. During the third cycle, the CPU fetches the more significant byte (02) of the destination address. Finally, the CPU puts the address in the program counter, thus executing JMP.

Can you recognize an instruction cycle on the oscilloscope? Note that SYNC is high during the clock cycle in which the CPU fetches the operation code.

PROBLEM F–1

Determine how long AB_0 is high. How long is AB_1 high? Explain these results.

PROBLEM F–2

How long is SYNC high? Suggest some uses for SYNC.

The CPU always accepts data at the end of clock phase 2. The memory address is always stable before the end of clock phase 1. How much time does this allow for a memory access? The only way to allow more is to reduce the clock frequency.

EXECUTION OF ADDRESSING MODES

Enter Program F–2 into memory and execute it.

PROGRAM F–2

Memory Address (Hex)	Memory Contents (Hex)		Instruetion (Mnemonic)	
0200	A9	HERE	LDA	#00
0201	00			
0202	4C		JMP	0200
0203	00			
0204	02			

PROBLEM F–3

What happens to SYNC during Program F–2? Explain its appearance.

The 6502 executes instructions with immediate addressing in two clock cycles:

1. During the first cycle, it uses the program counter to fetch the operation code from memory. As usual, it adds 1 to the program counter after using it.
2. During the second cycle, it uses the program counter to fetch the data from memory. It performs the operation and adds 1 to the program counter again.

SYNC is high during the first eyele to indieate the operation eode feteh.

PROBLEM F–4

What happens to SYNC if you replaee A9 (LDA immediate) in 0200 with A5 (LDA zero-page)? Explain the result. What instruetion is the proeessor executing?

The 6502 performs all register instruetions with zero-page (direet) addressing in three eloek eyeles as follows:

1. During the first eyele, it uses the program eounter to fetch the operation eode from memory.
2. During the seeond cycle, it uses the program eounter to feteh the address on page 0 from memory. In zero-page modes, the proeessor simply elears the more signifieant byte of the memory address internally.
3. During the third eyele, it uses the memory address to transfer the data. It then performs the operation. Sinee the program eounter is not used in this eyele, it is not ehanged.

Instruetions that store data in memory must produee a signal that indicates when data is available. READ/$\overline{\text{WRITE}}$ (Expansion Connector pin V) serves this purpose. In the programs we have run so far, this line should always be 1 (READ). Examine READ/$\overline{\text{WRITE}}$ during the execution of Program F–2 and verify this.

PROBLEM F-5

What happens to SYNC and READ/$\overline{\text{WRITE}}$ if you replaee A9 in 0200 with 85? What instruetion is now in 0200 and 0201? Verify your answer by loading the aeeumulator and 0000 initially and cheeking the results.

In relative addressing, the processor adds the offset to the program eounter to obtain the destination address. The 6502 minimizes the execution time of relative braneh es by not doing the addition if the condition is false. Furthermore, it performs a 16-bit addition (taking one extra eyele) only if the branch crosses a page boundary.

Enter Program F–3 into memory. CLC simply makes the branch eondition true. Examine SYNC and determine how many cyeles BCC takes. Here the branch (from 0203 to 0201) does not cross a page boundary.

PROGRAM F–3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	18		CLC	
0201	90	HERE	BCC	0201
0202	FE			

PROBLEM F–6

What happens to SYNC if you move Program F–3 to 02FE through 0300? How many cycles does BCC take now? Explain the change.

DECODING ADDRESS LINES

The AIM 65 decodes the upper half of memory (addresses 8000 through FFFF) as described in Table F–3 using a 74138 3-to-8-line decoder (see Figure F–4 and Table F–4). Since AB_{15} enables the decoder, all its outputs are inactive unless that line is 1.

The lower half of memory is partially decoded using a 74155 dual 2-to-4 decoder (see Figure F–5 and Table F–5). Half of this decoder is enabled through two NOR gates (7433; Z15) and an inverter (7404; Z16), making it active only when the four most

TABLE F–3 AIM 65 ADDRESS DECODING FOR UPPER 32K BYTES OF MEMORY*

AB_{14}	AB_{13}	AB_{12}	Symbol	Device Activated
0	0	0	$\overline{CS8}$	Available for memory expansion
0	0	1	$\overline{CS9}$	Available for memory expansion
0	1	0	$\overline{CSA}$	I/O
0	1	1	$\overline{CSB}$	Optional on-board BASIC Z26
1	0	0	$\overline{CSC}$	Optional on-board BASIC Z25
1	0	1	$\overline{CSD}$	Optional on-board assembler
1	1	0	$\overline{CSE}$	AIM monitor ROM Z23
1	1	1	$\overline{CSF}$	AIM monitor ROM Z22

*Figure F–6 shows the decoding circuit.

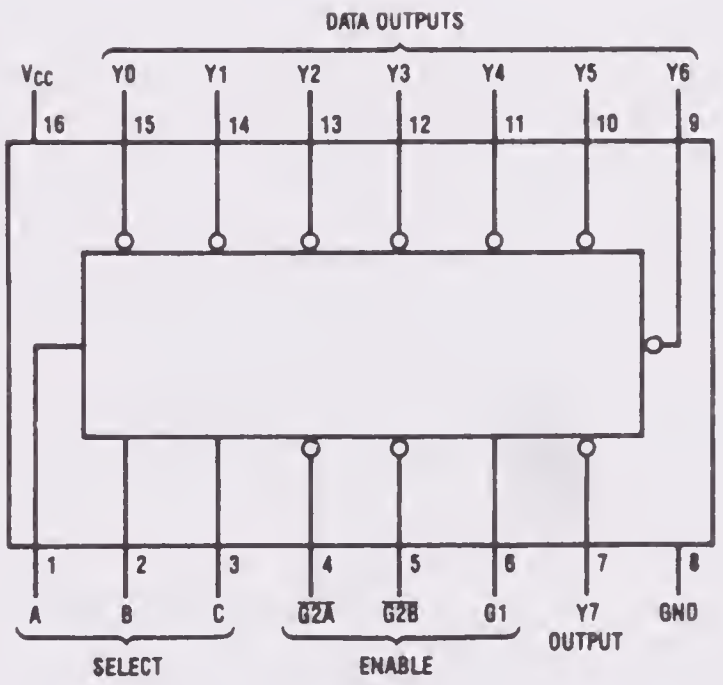


FIGURE F–4. Pin assignments for the 74138 3-to-8-line decoder.

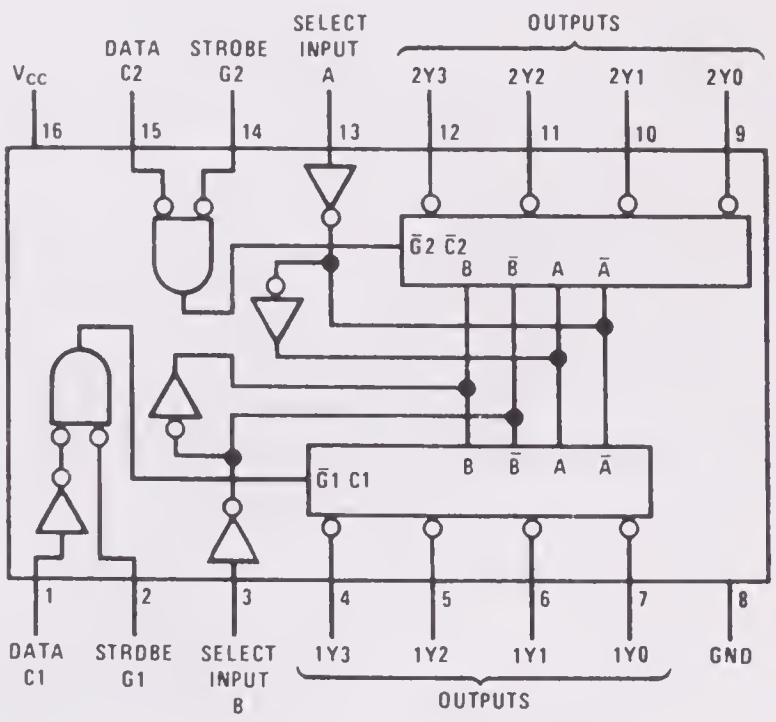


FIGURE F–5. Pin assignments for the 74155 dual 2-to-4-line decoder/demultiplexer.

significant address lines (AB_{12}, AB_{13}, AB_{14}, and AB_{15}) are all 0s. This half selects the RAM, which therefore occupies addresses 0000 through 03FF. Figure F–6 shows the entire decoding circuit.

TABLE F–4 FUNCTION TABLE FOR THE 74138 3-TO-8-LINE DECODER

Inputs					Outputs							
Enable		Select										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
–	H	–	–	–	H	H	H	H	H	H	H	H
L	–	–	–	–	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

H = high level, L = low level.
*G2 = G2A + G2B

TABLE F–5 FUNCTION TABLES FOR THE 74155 DUAL 2-TO-4-LINE DECODER/DEMULTIPLEXER

Inputs				Outputs			
Select		Strobe	Data				
B	A	1G	1C	1Y0	1Y1	1Y2	1Y3
–	–	H	–	H	H	H	H
L	L	L	H	L	H	H	H
L	H	L	H	H	L	H	H
H	L	L	H	H	H	L	H
H	H	L	H	H	H	H	L
–	–	–	L	H	H	H	H

Inputs				Outputs			
Select		Strobe	Data				
B	A	2G	2C	2Y0	2Y1	2Y2	2Y3
–	–	H	–	H	H	H	H
L	L	L	L	L	H	H	H
L	H	L	L	H	L	H	H
H	L	L	L	H	H	L	H
H	H	L	L	H	H	H	L
–	–	–	H	H	H	H	H

H = high level, L = low level.

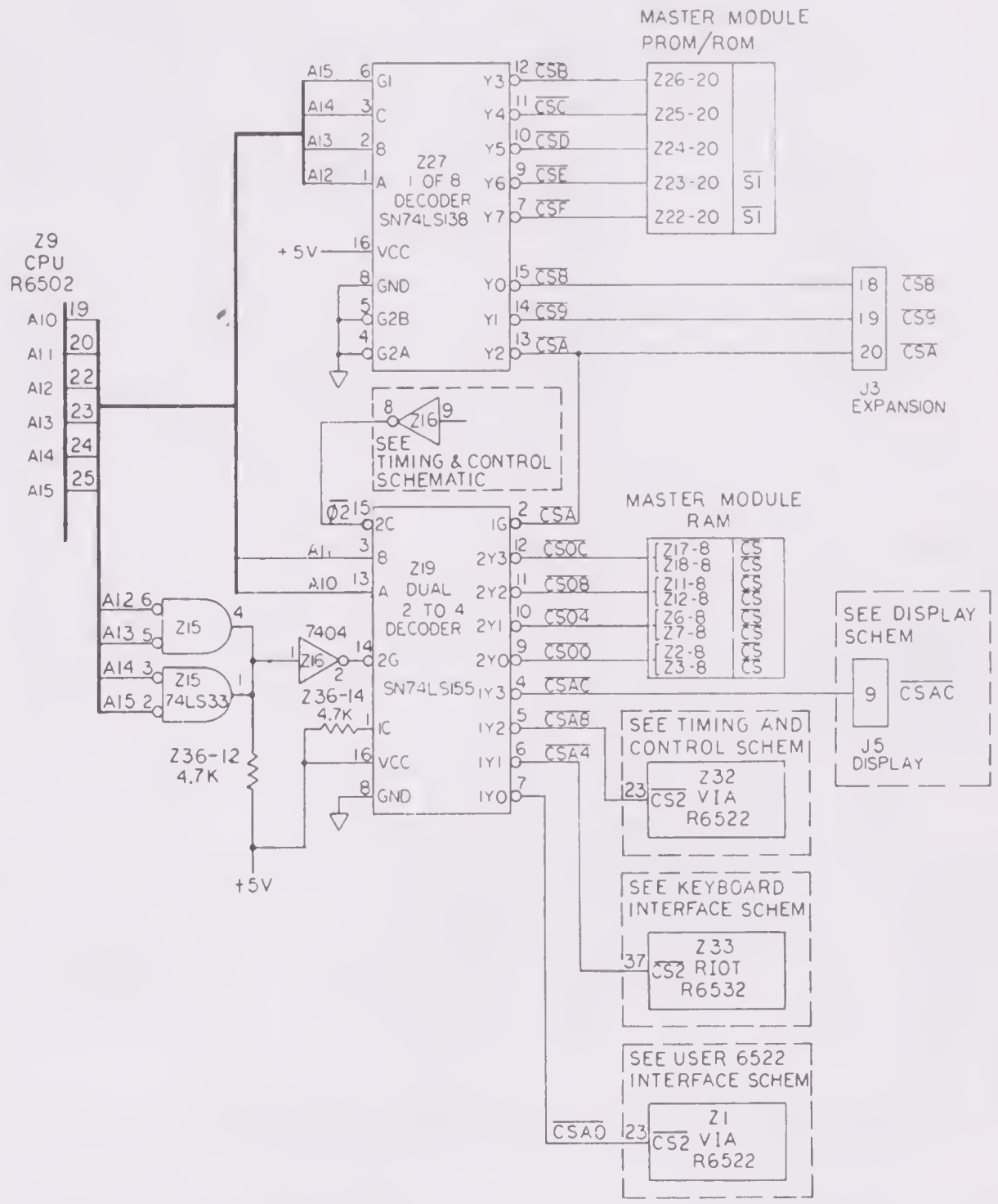


FIGURE F–6. Decoding circuit for AIM 65 memory and I/O. (Reprinted courtesy of Dynatem, Irvine, CA.)

Execute Program F–1 and examine $\overline{CS00}$ (74155 decoder pin 9). What happens to it? Note that Program F–1 is in the lowest 1K of user RAM.

The 74138 decoder divides the upper 32K of address space into eight 4K sections. $\overline{CS8}$ and $\overline{CS9}$ are uncommitted and can be used for off-board expansion; $\overline{CSA}$ is used for on-board I/O; and $\overline{CSB}$ through $\overline{CSF}$ enable the on-board ROM/PROM slots (see Figure F–7).

PROBLEM F–7

What happens to $\overline{CS00}$ if you put Program F–1 in A400 through A402? Remember to change the destination address in JMP to A400. What happens to output $\overline{CSA4}$ from the 74155 decoder? What happens to $\overline{CS00}$ and $\overline{CSA4}$ if you execute the following program?

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	AD	HERE	LDA	A400
0201	00			
0202	A4			
0203	4C		JMP	0200
0204	00			
0205	02			

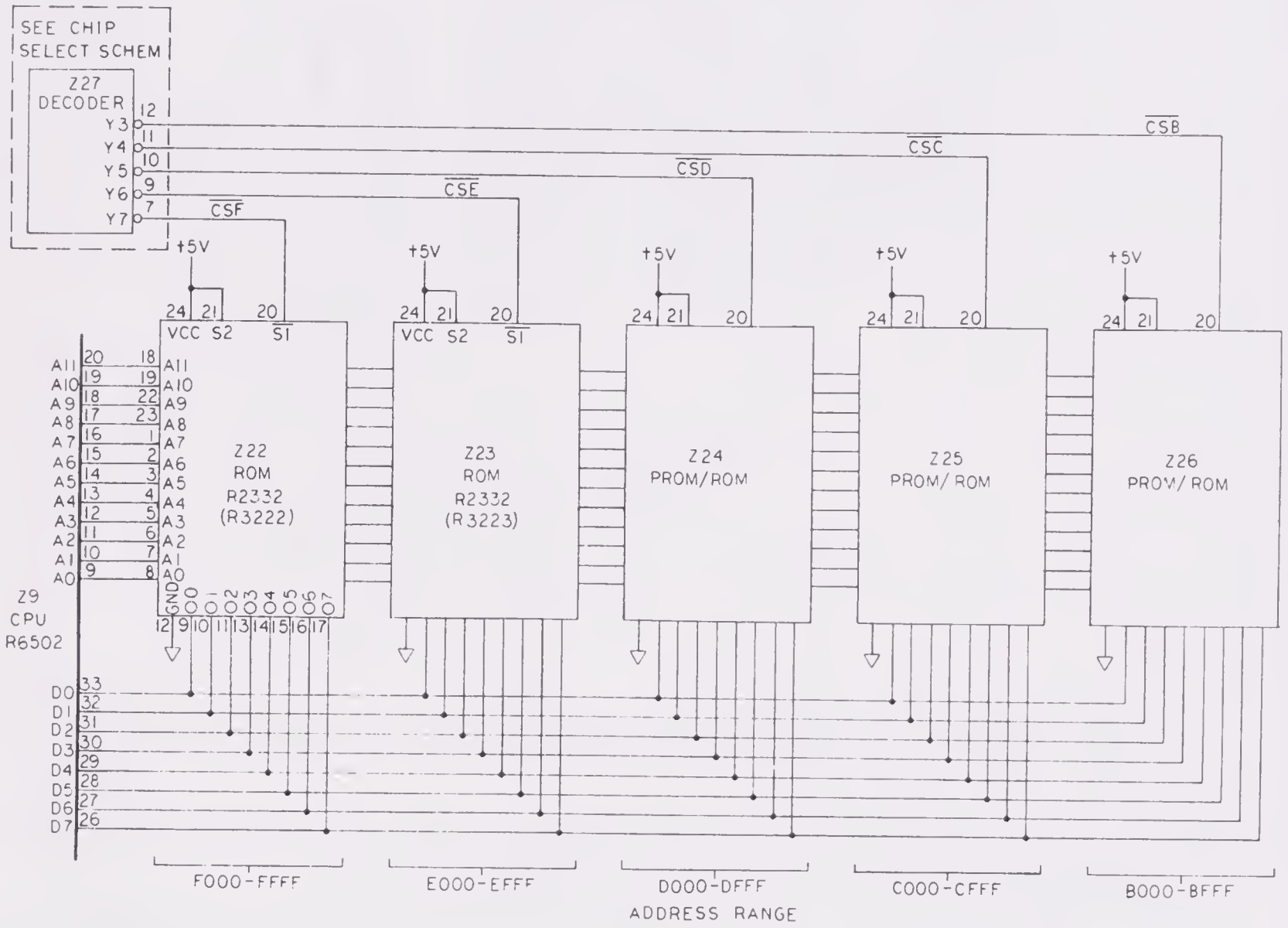


FIGURE F–7. AIM 65 PROM/ROM interface schematic. (Reprinted courtesy of Dynatem, Irvine, CA.)

PROBLEM F–8

The following program uses the printer dot-generation table in the monitor (starting in F2E1).

```
        LDX   #0          ; DIGIT = ZERO
HERE    LDA   $F2E1,X     ; CONVERT TO DOT-GENERATION CODE
        JMP   HERE
```

In mnemonic-entry form this is

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0200	A2		LDX	#00
0201	00			
0202	BD	HERE	LDA	F2E1,X
0203	E1			
0204	F2			
0205	4C		JMP	0202
0206	02			
0207	02			

Examine $\overline{\text{CSF}}$ (pin 7 of decoder Z27). This output signal activates the upper monitor ROM. How long is it active? Describe the execution of a register instruction using absolute indexed addressing. Describe the behavior of SYNC.

MULTIPLE ADDRESSES AND MEMORY EXPANSION

$\overline{\text{CS8}}$ and $\overline{\text{CS9}}$ from the 74138 decoder can be used to add 8K bytes of external RAM. When adding memory, we must ensure that one address never selects two locations. Reading that address would produce bus contention.

On the other hand, no contention occurs if many addresses select the same location. This sounds odd, but it does not affect the microcomputer's operation, any more than the practice of assigning a large building several street addresses (corresponding to different entrances) creates problems in mail delivery. For example, say we want to add an extra 1K-by-8 (1K-byte) RAM to the AIM. Rather than decode the address space further, we could simply enable the RAM with $\overline{\text{CS8}}$. The 1K RAM would then occupy 4K of address space, and each location in it would have four addresses. Since AB_{10} and AB_{11} are not connected to the RAM or to the decoder, their values would not affect selection.

The advantage of multiple addresses is obvious. To decode the 4K fully would require another 2-to-4-line decoder (decoding AB_{10} and AB_{11} in the same way as decoder Z19). This would mean an additional part, more connections, and more board space.

The disadvantage of multiple addresses is that they reduce the computer's memory capacity. In the example, 1K bytes of memory occupy an address space that could hold 4K bytes. This reduces the memory capacity by 3K bytes. The reduction does not matter, of course, if we have no further expansion plans.

The tradeoff here is clear. Decoding all address lines allows us to attach the maximum amount of memory. On the other hand, it requires additional parts (decoders). In simple applications, the extra parts are an unnecessary expense. Thus designers typically decode all address lines in large systems such as personal computers, graphics terminals, and robots. They typically leave address lines undecoded in small systems such as printers, electronic games, and simple instruments.

PROBLEM F–9

If we add a 1K-byte RAM as proposed (enabling it with $\overline{CS8}$), which addresses does it occupy? List the addresses that refer to the same memory location as 8000 hex.

ADDRESSING I/O DEVICES

The 74155 decoder in Figure F–6 divides the I/O address space into 1K sections. The I/O part of the decoder is activated by $\overline{CSA}$, so the sections start at A000. This still leaves us with the problem of addressing individual I/O devices. 6522 VIAs, for example, occupy only 16 bytes apiece. Thus 1K of address space could hold 64 VIAs, far more than most microcomputers need.

There is obviously no need to decode the I/O address space fully. After all, to decode 64 VIAs with 74138 devices fully would require eight chips. A less costly alternative is simply to use the available address lines without decoding.

This is easy to do with a VIA, since it has chip select inputs (see the pin assignments in Figure F–8). The RS (register select) lines decode the internal registers and are normally tied to AB_0 through AB_3. The CS (chip select) lines can be used for selection. For example, we could tie a decoder output from Figure F–6 (say, $\overline{CSA0}$) to the active-low chip select ($\overline{CS2}$). That VIA would then occupy the entire address space from A000 through A3FF. Address lines AB_{10} through AB_{15} are decoded in Figure F-6, AB_0 through AB_3 select internal VIA registers, and AB_4 through AB_9 are unconnected.

The advantage of this approach is obvious: We have attached a VIA without a decoder. The disadvantage is that we have used 1K of address space. Addresses differing only in AB_4 through AB_9 select the same VIA register. For example, A000, A010, A020, A040, A080, A100, and A200 all select I/O port B. We can readily observe this in practice, since the AIM 65 decodes the user VIA this way. Make port B output by loading FF into A002 and turn all the LEDs on by loading 00 into A000. Now examine A010. What happens when you place F0 in A010? Try changing A020, A040, A080, A100, and A200. If the AIM complains (with a MEM FAIL message) when you try to change VIA registers from the keyboard, use a program instead.

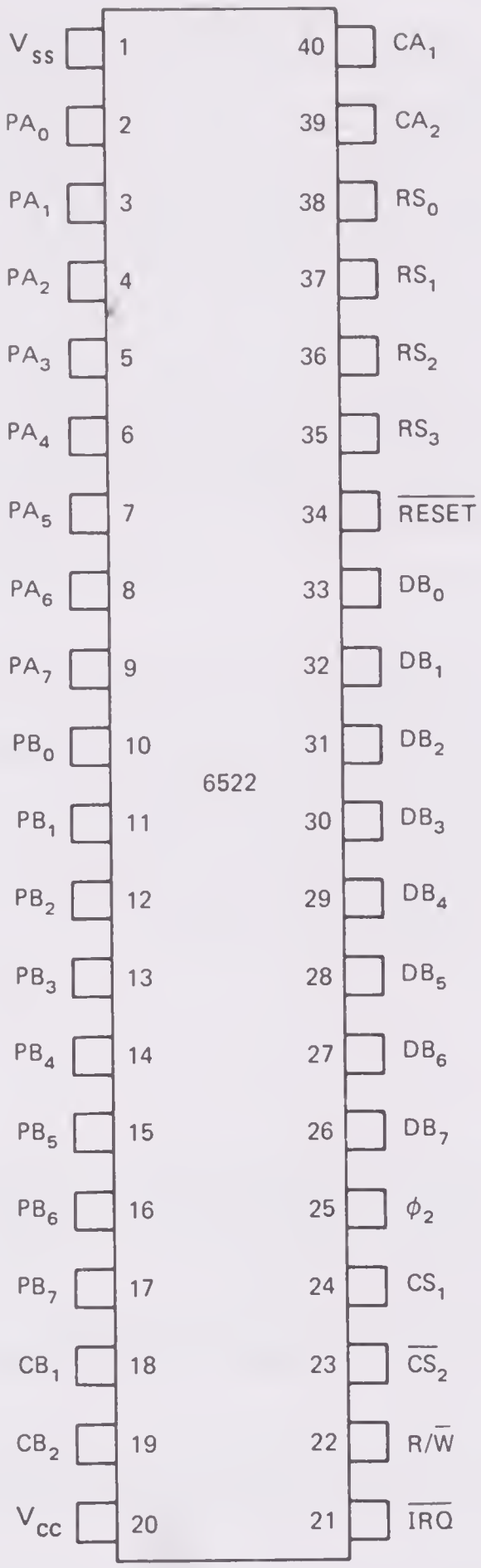


FIGURE F–8. Pin assignments for the 6522 VIA. (Reprinted courtesy of Rockwell International, Semiconductor Products Division, Newport Beach, CA.)

PROBLEM F–10

Which VIA registers occupy the following addresses?

a. A030
b. A1FB
c. A30E

We could use the address space more efficiently by partially decoding AB_4 through AB_9. For example, we could place two VIAs in 1K of memory by tying CS1 on one VIA to AB_9 and CS1 on the other to $\overline{AB}_9$.

A more clever way of decoding VIAs is to tie the chip selects directly to undecoded address lines. This approach is called *linear select*. It lets us place 6 VIAs in 1K of memory, thus providing sufficient I/O for many applications.

PROBLEM F–11

Linear select results in discontinuous addresses. Each 1 bit in the selection lines activates an I/O device, so only addresses with one 1 bit in those lines are occupied. Which addresses do VIAs occupy if we attach six of them using linear select and tie $\overline{CSA8}$ from Figure F–6 to $\overline{CS2}$ on each device?

PROBLEM F–12

One outgrowth of the discontinuous addresses in Problem F–11 is that we can actually send data to several VIAs at once. This is done by storing the data in an address with 1 bits in several selection lines. This multiple transmission is called a *broadcast*, since it is like a general broadcast on a communications network. Assuming that we have six VIAs addressed as in Problem F–11, write a program that makes all A ports input and all B ports output and stores 55 hex in all B ports. Note that port B, port A, data direction register B, and data direction register A are addresses 0, 1, 2, and 3, respectively, inside the VIA.

KEY POINT SUMMARY

1. A logic analyzer is needed for full understanding or debugging of the hardware in microprocessor-based systems. The analyzer can display many signals simultaneously in a comprehensible format.
2. The 6502 executes instructions in a series of clock cycles consisting of a setup period followed by a transfer period. All address and control signals are stable during the transfer period.
3. The 6502 differentiates between operation code fetches and other cycles with the SYNC signal. SYNC is high when the processor is fetching an operation code from memory.
4. Instruction execution takes at least two clock cycles. During the first cycle, the CPU fetches an operation code. It puts the program counter on the address bus and loads the data from memory into the instruction register. The CPU adds 1 to the program counter after each use.
5. The processor executes instructions with different addressing modes in different ways. For immediate addressing, it uses the program counter to fetch the data. For zero-page and absolute (direct) addressing, it uses the program counter to fetch the

data's address and then uses that address to aeeess memory. For indexed and relative addressing, it must first perform an addition to ealeulate the effeetive address.

6. The more signifieant address lines are usually deeoded to form enabling signals. These signals seleet a partieular memory or I/O deviee. In general, only one memory or I/O deviee ean be seleeted at a time.
7. The designer ean make tradeoffs between a eomputer's memory eapaeity and the eomplexity of its deeoding system. Full deeoding of addresses maximizes memory eapaeity but ineréases parts eount. Partial deeoding is often suffieient in small systems.
8. I/O addresses are seldom deeoded fully beeause few applieations require more than a small fraetion of the available eapaeity. Linear seleet (using an address line direetly to seleet an I/O deviee) is a eonvenient way to provide a reasonable I/O eapaeity without deeoders.

APPENDIX 1—6502 MICROPROCESSOR INSTRUCTION SET*

Table A1-1

MICROPROCESSOR OPERATION CODES AND OBJECT CODES

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND, X)		
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2
ASL	C ← 7 0 ← 0				0E	6	3	06	5	2	0A	2	1						
BCC	BRANCH ON C = 0 (2)																		
BCS	BRANCH ON C = 1 (2)																		
BEQ	BRANCH ON Z = 1 (2)																		
BIT	A∧M				2C	4	3	24	3	2									
BMI	BRANCH ON N = 1 (2)																		
BNE	BRANCH ON Z = 0 (2)																		
BPL	BRANCH ON N = 0 (2)																		
BRK	BREAK (See Fig 1)													00	7	1			
BVC	BRANCH ON V = 0 (2)																		
BVS	BRANCH ON V = 1 (2)																		
CLC	0 → C													18	2	1			
CLD	0 → D													D8	2	1			
CLI	0 → I													58	2	1			
CLV	0 → V													B8	2	1			
CMP	A - M	C9	2	2	CD	4	3	C5	3	2							C1	6	2
CPX	X - M	E0	2	2	EC	4	3	E4	3	2									
CPY	Y - M	C0	2	2	CC	4	3	C4	3	2									
DEC	M - 1 → M				CE	6	3	C6	5	2									
DEX	X - 1 → X													CA	2	1			
DEY	Y - 1 → Y													88	2	1			
EOR	A ⊻ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2
INC	M + 1 → M				EE	6	3	E6	5	2									
INX	X + 1 → X													E8	2	1			
INY	Y + 1 → Y													C8	2	1			
JMP	JUMP TO NEW LOC				4C	3	3												
JSR	JUMP SUB (See Fig A1 2)				20	6	3												
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2									
LOY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2									
LSR	0 → 7 0 → C				4E	6	3	46	5	2	4A	2	1						
NOP	NO OPERATION													EA	2	1			
ORA	A ∨ M → A	09	2	2	0D	4	3	05	3	2							01	6	2
PHA	A → Ms S - 1 → S													48	3	1			
PHP	P → Ms S - 1 → S													08	3	1			
PLA	S + 1 → S Ms → A													68	4	1			
PLP	S + 1 → S Ms → P													28	4	1			
ROL	7 0 ← C				2E	6	3	26	5	2	2A	2	1						
ROR	C → 7 0				6E	6	3	66	5	2	6A	2	1						
RTI	RTRN INT (See Fig A1 1)													40	6	1			
RTS	RTRN SUB (See Fig A1 2)													60	6	1			
SBC	A - M - C → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2
SEC	1 → C													38	2	1			
SED	1 → D													F8	2	1			
SEI	1 → I													78	2	1			
STA	A → M				8D	4	3	85	3	2							81	6	2
STX	X → M				8E	4	3	86	3	2									
STY	Y → M				8C	4	3	84	3	2									
TAX	A → X													AA	2	1			
TAY	A → Y													A8	2	1			
TSX	S → X													BA	2	1			
TXA	X → A													8A	2	1			
TXS	X → S													9A	2	1			
TYA	Y → A													98	2	1			

(IND), Y			Z PAGE X			ABS X			ABS, Y			RELATIVE			INDIRECT			Z PAGE Y			PROCESSOR STATUS CODES	MNEMONIC
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	
71	5	2	75	4	2	7D	4	3	79	4	3										N V • • • • Z C	ADC
31	5	2	35	4	2	3D	4	3	39	4	3										N • • • • • Z •	AND
			16	6	2	1E	7	3													N • • • • • Z C	ASL
												90	2	2							• • • • • • • •	BCC
												B0	2	2							• • • • • • • •	BCS
												F0	2	2							• • • • • • • •	BEQ
																					M_7 M_6 • • • • Z •	BIT
												30	2	2							• • • • • • • •	BMI
												D0	2	2							• • • • • • • •	BNE
												10	2	2							• • • • • • • •	BPL
																					• • • 1 • 1 • •	BRK
												50	2	2							• • • • • • • •	BVC
												70	2	2							• • • • • • • •	BVS
																					• • • • • • • 0	CLC
																					• • • • 0 • • •	CLD
																					• • • • • 0 • •	CLI
																					• 0 • • • • • •	CLV
D1	5	2	D5	4	2	DD	4	3	D9	4	3										N • • • • • Z C	CMP
																					N • • • • • Z C	CPX
																					N • • • • • Z C	CPY
			D6	6	2	DE	7	3													N • • • • • Z •	DEC
																					N • • • • • Z •	DEX
																					N • • • • • Z •	DEY
51	5	2	55	4	2	5D	4	3	59	4	3										N • • • • • Z •	EOR
			F6	6	2	FE	7	3													N • • • • • Z •	INC
																					N • • • • • Z •	INX
																					N • • • • • Z •	INY
															6C	5	3				• • • • • • • •	JMP
																					• • • • • • • •	JSR
B1	5	2	B5	4	2	BD	4	3	B9	4	3										N • • • • • Z •	LDA
									BE	4	3							B6	4	2	N • • • • • Z •	LDX
			B4	4	2	BC	4	3													N • • • • • Z •	LDY
			56	6	2	5E	7	3													0 • • • • • Z C	LSR
																					• • • • • • • •	NOP
11	5	2	15	4	2	1D	4	3	19	4	3										N • • • • • Z •	ORA
																					• • • • • • • •	PHA
																					• • • • • • • •	PHP
																					N • • • • • Z •	PLA
																					(RESTORED)	PLP
			36	6	2	3E	7	3													N • • • • • Z C	ROL
			76	6	2	7E	7	3													N • • • • • Z C	ROR
																					(RESTORED)	RTI
																					• • • • • • • •	RTS
F1	5	2	F5	4	2	FD	4	3	F9	4	3										N V • • • • Z (3)	SBC
																					• • • • • • • 1	SEC
																					• • • • 1 • • •	SED
																					• • • • • 1 • •	SEI
91	6	2	95	4	2	9D	5	3	99	5	3										• • • • • • • •	STA
																		96	4	2	• • • • • • • •	STX
			94	4	2																• • • • • • • •	STY
																					N • • • • • Z •	TAX
																					N • • • • • Z •	TAY
																					N • • • • • Z •	TSX
																					N • • • • • Z •	TXA
																					• • • • • • • •	TXS
																					N • • • • • Z •	TYA

(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
∨ OR
⊻ EXCLUSIVE OR
M_7 MEMORY BIT 7
M_6 MEMORY BIT 6
n NO. CYCLES
NO. BYTES

*Courtesy of Semiconductor Products Division, Rockwell International, Newport Beach, CA.

Table A1-2

MNEMONIC EQUIVALENTS OF HEXADECIMAL OPERATION CODES

LSD / MSD	0	1	2	3	4	5	6	7
0	BRK	ORA IND X				ORA Z PAGE	ASL Z PAGE	
1	BPL	ORA IND Y				ORA Z PAGE X	ASL Z PAGE X	
2	JSR	AND IND X			BIT Z Page	AND Z PAGE	ROL Z PAGE	
3	BMI	AND IND Y				AND Z PAGE X	ROL Z PAGE X	
4	RTI	EOR IND X				EOR Z PAGE	LSR Z PAGE	
5	BVC	EOR IND Y				EOR Z PAGE X	LSR Z PAGE X	
6	RTS	ADC IND X				AOC Z PAGE	ROR Z PAGE	
7	BVS	ADC IND Y				AOC Z PAGE X		
8		STA IND X			STY Z Page	STA Z PAGE	STX Z PAGE	
9	BCC	STA IND Y			STY Z Page X	STA Z PAGE X	STX Z PAGE Y	
A	LDY IMM	LDA IND X	LDX IMM		LDY Z Page	LDA Z PAGE	LDX Z PAGE	
B	BCS	LDA IND Y			LDY Z Page X	LDA Z PAGE X	LDX Z PAGE Y	
C	CPY IMM	CMP IND X			CPY Z Page	CMP Z PAGE	DEC Z PAGE	
D	BNE	CMP IND Y				CMP Z PAGE X	DEC Z PAGE X	
E	CPX IMM	SBC IND X			CPX Z Page	SBC Z PAGE	INC Z PAGE	
F	BEQ	SBC IND Y				SBC Z PAGE X	INC Z PAGE X	

8	9	A	B	C	D	E	F	LSD / MSD
PHP	ORA IMM	ASL A			ORA ABS	ASL ABS		0
CLC	ORA ABS Y				ORA ABS X	ASL ABS X		1
PLP	AND IMM	ROL A		BIT ABS	AND ABS	ROL ABS		2
SEC	AND ABS Y				AND ABS X	ROL ABS X		3
PHA	EOR IMM	LSR A		JUMP ABS	EOR ABS	LSR ABS		4
CLI	EOR ABS Y				EOR ABS X	LSR ABS X		5
PLA	ADC IMM	ROR A		JMP IND	ADC ABS	ROR ABS		6
SEI	ADC ABS Y				ADC ABS X			7
DEY		TXA		STY ABS	STA ABS	STX ABS		8
TYA	STA ABS Y	TXS			STA ABS X			9
TAY	LDA IMM	TAX		LDY ABS	LDA ABS	LDX ABS		A
CLV	LDA ABS Y	TSX		LDY ABS X	LDA ABS X	LDX ABS Y		B
INY	CMP IMM	DEX		CPY ABS	CMP ABS	DEC ABS		C
CLD	CMP ABS Y				CMP ABS X	DEC ABS X		D
INX	SBC IMM	NOP		CPX ABS	SBC ABS	INC ABS		E
SED	SBC ABS Y				SBC ABS X	INC ABS X		F

ADDRESSING MODES

IMM — IMMEDIATE ADDRESSING — The operand is contained in the second byte of the instruction.

ABS — ABSOLUTE ADDRESSING — The second byte of the instruction contains the 8 low order bits of the effective address (EA). The third byte contains the 8 high order bits of the effective address.

Z PAGE — ZERO PAGE ADDRESSING — Second byte contains the 8 low order bits of the effective address. The 8 high order bits are zero.

A — ACCUMULATOR — One byte instruction operating on the accumulator.

Z PAGE, X - Z PAGE, Y — ZERO PAGE INDEXED — The second byte of the instruction is added to the index (carry is dropped) to form the low order byte of the EA. The high order byte of the EA is zero.

ABS, X - ABS, Y — ABSOLUTE INDEXED — The effective address is formed by adding the index to the second and third byte of the instruction.

(IND, X) — INDEXED INDIRECT — The second byte of the instruction is added to the X index, discarding the carry. The result points to a location on page zero which contains the 8 low order bits of the EA. The next byte contains the 8 high order bits.

(IND), Y — INDIRECT INDEXED — The second byte of the instruction points to a location in page zero. The contents of this memory location is added to the Y index, the result being the low order eight bits of the EA. The carry from this operation is added to the contents of the next page zero location, the result being the 8 high order bits of the EA.

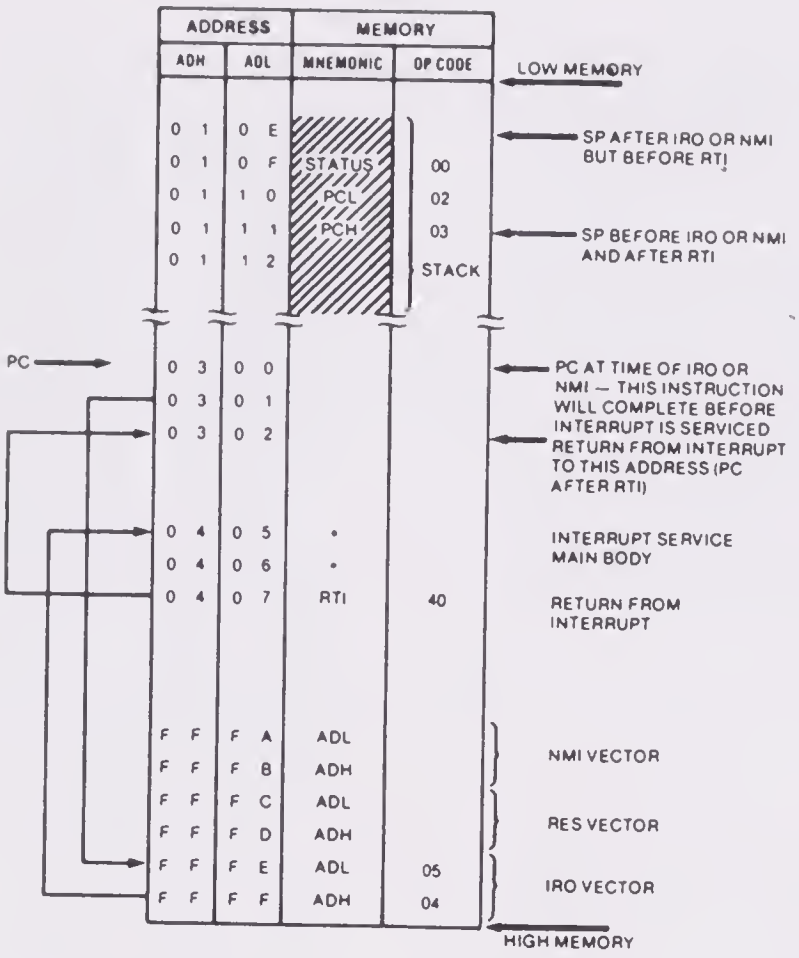


FIGURE A1–1. Response to $\overline{\text{IRQ}}$ and NMI inputs and operation of the RTI and BRK instructions.

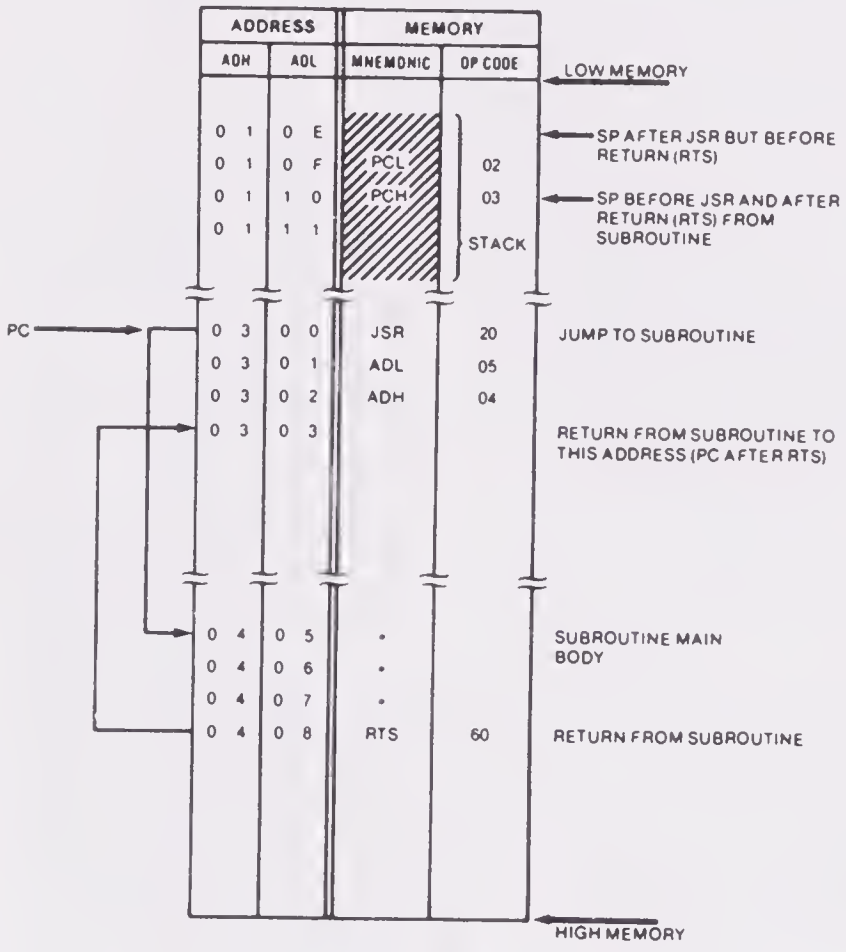


FIGURE A1–2. Operation of the JSR and RTS instructions.

Table A1-3

6502 MICROPROCESSOR INSTRUCTION SET—ALPHABETIC SEQUENCE

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

Table A1-4

SUMMARY OF 6502 ADDRESSING MODES

Accumulator addressing. This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

Immediate addressing. In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

Absolute addressing. In absolute addressing, the second byte of the instruction specifies the 8 low-order bits of the effective address while the third byte specifies the 8 high-order bits. Thus the absolute addressing mode allows access to the entire 64K bytes of addressable memory.

Zero page addressing. The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high-address byte. Careful use of the zero page can result in a significant increase in code efficiency.

Indexed zero page addressing (X, Y indexing). This form of addressing is used in conjunction with an index register and is referred to as "Zero Page, X" or "Zero Page, Y." The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally, due to the "Zero Page" addressing nature of this mode, no carry is added to the high-order 8 bits of the effective address, so it is always in page zero.

Indexed absolute addressing (X, Y indexing). This form of addressing is used in conjunction with index register X or Y and is referred to as "Absolute, X" and "Absolute, Y." The effective address is formed by adding the contents of X or Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

Implied addressing. In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

Relative addressing. Relative addressing is used only with branch instructions and establishes a destination for the conditional branch. The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower 8 bits of the program counter when the counter is set at the next instruction. The range of the offset is -128 to +127 bytes from the next instruction.

Indexed indirect addressing. In indexed indirect addressing (referred to as (Indirect,X)), the second byte of the instruction is added to the contents of index register X, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low-order 8 bits of the effective address. The next memory location in page zero contains the high-order 8 bits of the effective address. Both memory locations specifying the high- and low-order bytes of the effective address must be in page zero.

Indirect indexed addressing. In indirect indexed addressing [referred to as (Indirect), Y] the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of index register Y, the result being the low-order 8 bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high-order 8 bits of the effective address.

Absolute indirect. The second byte of the instruction contains the low-order 8 bits of a memory address. The high-order 8 bits of that memory address is contained in the third byte of the instruction. The contents of the fully specified memory location is the low-order byte of the effective address. The next memory location contains the high-order byte of the effective address, which is loaded into the 16 bits of the program counter.

APPENDIX 2—ASCII CHARACTER TABLE

HEX-ASCII TABLE

00	NUL		21	!		42	B		63	c	
01	SOH		22	"		43	C		64	d	
02	STX		23	#		44	D		65	e	
03	ETX		24	$		45	E		66	f	
04	EOT		25	%		46	F		67	g	
05	ENQ		26	&		47	G		68	h	
06	ACK		27	'		48	H		69	i	
07	BEL		28	(		49	I		6A	j	
08	BS		29	)		4A	J		6B	k	
09	HT		2A	*		4B	K		6C	l	
0A	LF		2B	+		4C	L		6D	m	
0B	VT		2C	,		4D	M		6E	n	
0C	FF		2D	-		4E	N		6F	o	
0D	CR		2E	.		4F	O		70	p	
0E	SO		2F	/		50	P		71	q	
0F	SI		30	0		51	Q		72	r	
10	DLE		31	1		52	R		73	s	
11	DC1	(X-ON)	32	2		53	S		74	t	
12	DC2	(TAPE)	33	3		54	T		75	u	
13	DC3	(X-OFF)	34	4		55	U		76	v	
14	DC4		35	5		56	V		77	w	
15	NAK		36	6		57	W		78	x	
16	SYN		37	7		58	X		79	y	
17	ETB		38	8		59	Y		7A	z	
18	CAN		39	9		5A	Z		7B	{	
19	EM		3A	:		5B	[		7C	|	
1A	SUB		3B	;		5C	\		7D	}	
1B	ESC		3C	<		5D	]		~	(ALT MODE)	
1C	FS		3D	=		5E	^	(↑)	7E		
1D	GS		3E	>		5F	—	(←)	7F	DEL	
1E	RS		3F	?		60	`			(RUBOUT)	
1F	US		40	@		61	a				
20	SP		41	A		62	b				

Rockwell

R6500 Microcomputer System DATA SHEET

R6500 MICROPROCESSORS (CPU)

SYSTEM ABSTRACT

The 8-bit R6500 microcomputer system is produced with N-Channel, Silicon Gate technology. Its performance speeds are enhanced by advanced system architecture. This innovative architecture results in smaller chips – the semiconductor threshold is cost-effectivity. System cost-effectivity is further enhanced by providing a family of 10 software-compatible microprocessor (CPU) devices, described in this document. Rockwell also provides memory and microcomputer system– as well as low-cost design aids and documentation.

R6500 MICROPROCESSOR (CPU) CONCEPT

Ten CPU devices are available. All are software-compatible. They provide options of addressable memory, interrupt input, on-chip clock oscillators and drivers. All are bus-compatible with earlier generation microprocessors like the M6800 devices.

The family includes six microprocessors with on-board clock oscillators and drivers and four microprocessors driven by external clocks. The on-chip clock versions are aimed at high performance, low cost applications where single phase inputs, crystal or RC inputs provide the time base. The external clock versions are geared for multiprocessor system applications where maximum timing control is mandatory. All R6500 microprocessors are also available in a variety of packaging (ceramic and plastic), operating frequency (1 MHz, 2 MHz and 3 MHz) and temperature (commercial and industrial) versions.

MEMBERS OF THE R6500 MICROPROCESSOR (CPU) FAMILY

Microprocessors with Internal Two Phase Clock Generator

Model	Addressable Memory
R6502	64K Bytes
R6503	4K Bytes
R6504	8K Bytes
R6505	4K Bytes
R6506	4K Bytes
R6507	8K Bytes

Microprocessors with External Two Phase Clock Input

Model	Addressable Memory
R6512	64K Bytes
R6513	4K Bytes
R6514	8K Bytes
R6515	4K Bytes

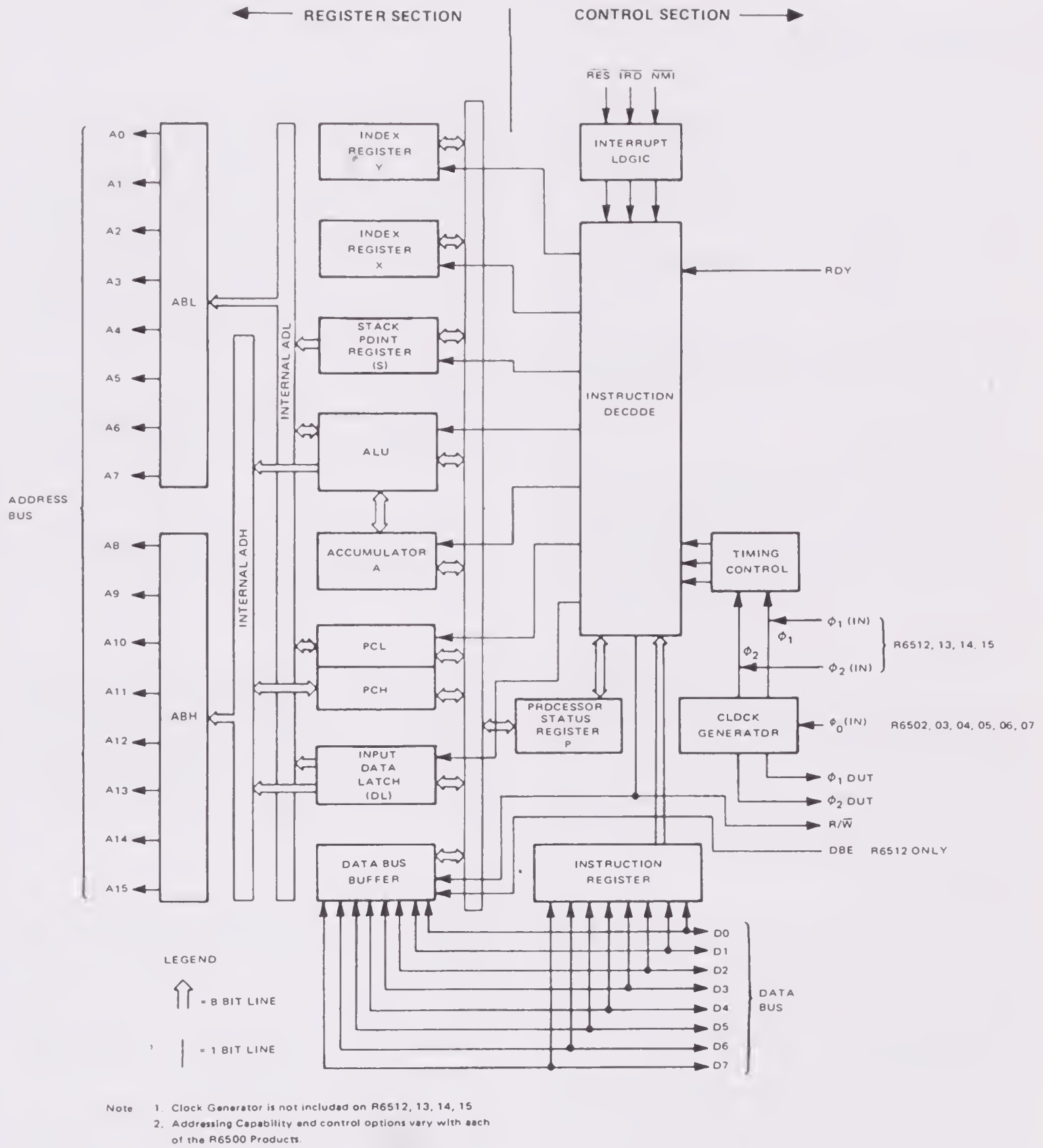
FEATURES

- Single +5V supply
- N channel, silicon gate, depletion load technology
- Eight bit parallel processing
- 56 Instructions
- Decimal and binary arithmetic
- Thirteen addressing modes
- True indexing capability
- Programmable stack pointer
- Variable length stack
- Interrupt capability
- Non-maskable interrupt
- Use with any type of speed memory
- 8-bit Bidirectional Data Bus
- Addressable memory range of up to 64K bytes
- "Ready" input
- Direct Memory Access capability
- Bus compatible with M6800
- 1 MHz, 2 MHz, and 3 MHz versions
- Choice of external or on-chip clocks
- On-the-chip clock options
  - External single clock input
  - Crystal time base input
- Commercial and industrial temperature versions
- Pipeline architecture

Ordering Information

Order Number: R65XX _ _ _

- Temperature Range:
  - No suffix = 0°C to +70°C
  - E = -40°C to +85°C (Industrial)
- Package: C = Ceramic, P = Plastic
- Frequency Range:
  - No suffix = 1 MHz
  - A = 2 MHz
  - B = 3 MHz
- Model Designator:
  - XX = 02, 03, 04, . . 15



Note 1. Clock Generator is not included on R6512, 13, 14, 15
2. Addressing Capability and control options vary with each of the R6500 Products.

R6500 Internal Architecture

The specification sheets on pages 310–323 are reprinted with the permission of Semiconductor Products Division, Rockwell International, Newport Beach, CA.

R6500 Signal Description

Clocks (ϕ_1, ϕ_2)

The R651X requires a two phase non-overlapping clock that runs at the V_{CC} voltage level.

The R650X clocks are supplied with an internal clock generator. The frequency of these clocks is externally controlled.

Address Bus (A0-A15)

These outputs are TTL compatible, capable of driving one standard TTL load and 130 pF.

Data Bus (D0-D7)

Eight pins are used for the data bus. This is a bidirectional bus, transferring data to and from the device and peripherals. The outputs are tri-state buffers capable of driving one standard TTL load and 130 pF.

Data Bus Enable (DBE)

This TTL compatible input allows external control of the tri-state data output buffers and will enable the microprocessor bus driver when in the high state. In normal operation DBE would be driven by the phase two (ϕ_2) clock, thus allowing data output from microprocessor only during ϕ_2. During the read cycle, the data bus drivers are internally disabled, becoming essentially an open circuit. To disable data bus drivers externally, DBE should be held low.

Ready (RDY)

This input signal allows the user to halt or single cycle the microprocessor on all cycles except write cycles. A negative transition to the low state during or coincident with phase one (ϕ_1) will halt the microprocessor with the output address lines reflecting the current address being fetched. If Ready is low during a write cycle, it is ignored until the following read operation. This condition will remain through a subsequent phase two (ϕ_2) in which the Ready signal is low. This feature allows microprocessor interfacing with the low speed PROMs as well as Direct Memory Access (DMA).

Interrupt Request ($\overline{IRQ}$)

This TTL level input requests that an interrupt sequence begin within the microprocessor. The microprocessor will complete the current instruction being executed before recognizing the request. At that time, the interrupt mask bit in the Status Code Register will be examined. If the interrupt mask flag is not set, the microprocessor will begin an interrupt sequence. The Program Counter and Processor Status Register are stored in the stack. The microprocessor will then set the interrupt mask flag high so that no further interrupts may occur. At the end of this cycle, the program counter low will be loaded from address FFFE, and program counter high from location FFFF, therefore transferring program control to the memory vector located at these addresses. The RDY signal must be in the high state for any interrupt to be recognized. A 3KΩ external resistor should be used for proper wire-OR operation.

Non-Maskable Interrupt ($\overline{NMI}$)

A negative going edge on this input requests that a non-maskable interrupt sequence be generated within the microprocessor.

$\overline{NMI}$ is an unconditional interrupt. Following completion of the current instruction, the sequence of operations defined for $\overline{IRQ}$ will be performed, regardless of the state interrupt mask flag. The vector address loaded into the program counter, low and high, are locations FFFA and FFFB respectively, thereby transferring program control to the memory vector located at these addresses. The instructions loaded at these locations cause the microprocessor to branch to a non-maskable interrupt routine in memory.

$\overline{NMI}$ also requires an external 3K Ω register to V_{CC} for proper wire-OR operations.

Inputs $\overline{IRQ}$ and $\overline{NMI}$ are hardware interrupts lines that are sampled during ϕ_2 (phase 2) and will begin the appropriate interrupt routine on the ϕ_1 (phase 1) following the completion of the current instruction.

Set Overflow Flag (S.O.)

A negative going edge on this input sets the overflow bit in the Status Code Register. This signal is sampled on the trailing edge of $\phi 1$ and must be externally synchronized.

SYNC

This output line is provided to identify those cycles in which the microprocessor is doing an OP CODE fetch. The SYNC line goes high during ϕ_1 of an OP CODE fetch and stays high for the remainder of that cycle. If the RDY line is pulled low during the ϕ_1 clock pulse in which SYNC went high, the processor will stop in its current state and will remain in the state until the RDY line goes high. In this manner, the SYNC signal can be used to control RDY to cause single instruction execution.

Reset

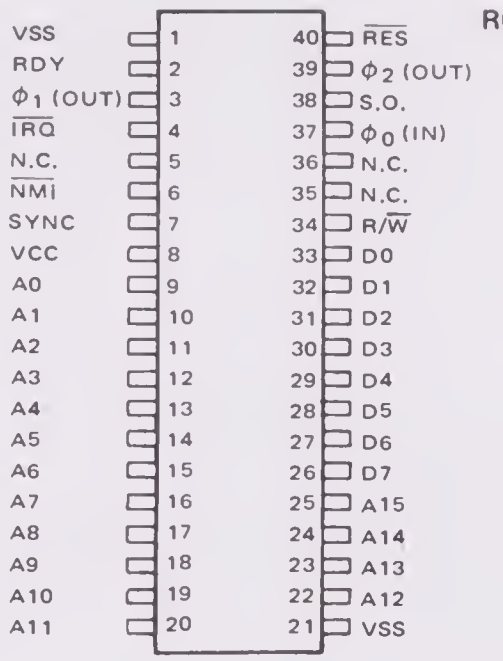
This input is used to reset or start the microprocessor from a power down condition. During the time that this line is held low, writing to or from the microprocessor is inhibited. When a positive edge is detected on the input, the microprocessor will immediately begin the reset sequence.

After a system initialization time of six clock cycles, the mask interrupt flag will be set and the microprocessor will load the program counter from the memory vector locations FFFC and FFFD. This is the start location for program control.

After V_{CC} reaches 4.75 volts in a power up routine, reset must be held low for at least two clock cycles. At this time the R/W and (SYNC) signal will become valid.

When the reset signal goes high following these two clock cycles, the microprocessor will proceed with the normal reset procedure detailed above.

R6502 — 40 Pin Package



Features of R6502

- 64K Addressable Bytes of Memory (A0-A15)
- $\overline{IRQ}$ Interrupt
- On-the-chip Clock
  - TTL Level Single Phase Input
  - RC Time Base Input
  - Crystal Time Base Input
- SYNC Signal
  (can be used for single instruction execution)
- RDY Signal
  (can be used to halt or single cycle execution)
- Two Phase Output Clock for Timing of Support Chips
- $\overline{NMI}$ Interrupt

Rockwell

R6500 Microcomputer System
DATA SHEET

VERSATILE INTERFACE ADAPTER (VIA)

- Two 8-Bit Bidirectional I/O Ports
- Two 16-Bit Programmable Timer/Counters
- Serial Data Port
- Single +5V Power Supply
- TTL Compatible
- CMOS Compatible Peripheral Control Lines
- Expanded "Handshake" Capability Allows Positive Control of Data Transfers Between Processor and Peripheral Devices
- Latched Output and Input Registers
- 1 MHz and 2 MHz Operation

The R6522 Versatile Interface Adapter (VIA) is a very flexible I/O control device. In addition, this device contains a pair of very powerful 16-bit interval timers, a serial-to-parallel/parallel-to-serial shift register and input data latching on the peripheral ports. Expanded handshaking capability allows control of bi-directional data transfers between VIA's in multiple processor systems.

Control of peripheral devices is handled primarily through two 8-bit bi-directional ports. Each line can be programmed as either an input or an output. Several peripheral I/O lines can be controlled directly from the interval timers for generating programmable frequency square waves or for counting externally generated pulses. To facilitate control of the many powerful features of this chip, an interrupt flag register, an interrupt enable register and a pair of function control registers are provided.

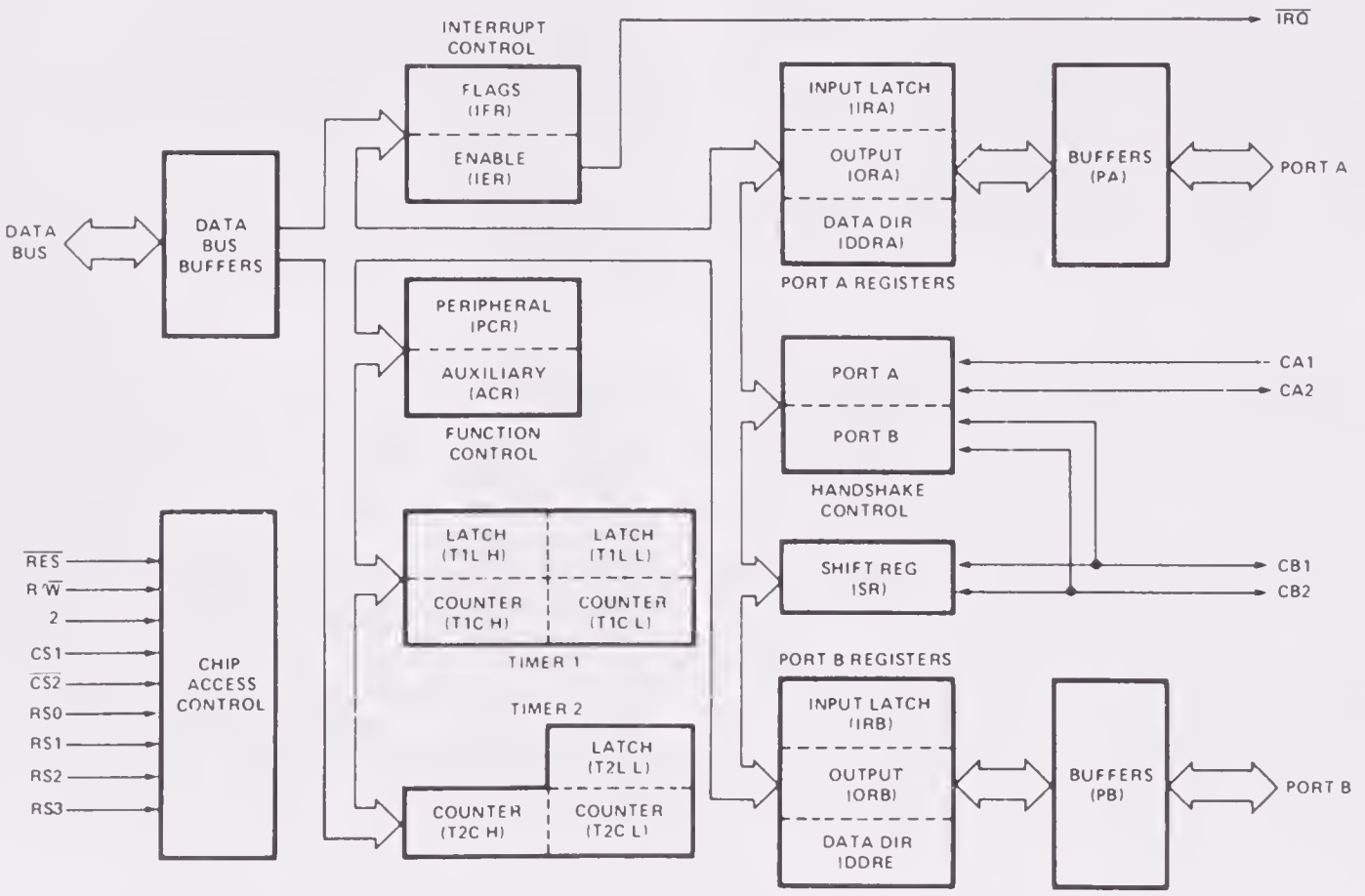


Figure 1. R6522 Block Diagram

PIN DESCRIPTIONS

$\overline{RES}$ (Reset)

The reset input clears all internal registers to logic 0 (except T1 and T2 latches and counters and the Shift Register). This places all peripheral interface lines in the input state, disables the timers, shift register, etc. and disables interrupting from the chip.

$\phi 2$ (Input Clock)

The input clock is the system $\phi 2$ clock and is used to trigger all data transfers between the system processor and the R6522.

R/$\overline{W}$ (Read/Write)

The direction of the data transfers between the R6522 and the system processor is controlled by the R/$\overline{W}$ line. If R/W is low, data will be transferred out of the processor into the selected R6522 register (write operation). If R/W is high and the chip is selected, data will be transferred out of the R6522 (read operation).

DB0–DB7 (Data Bus)

The eight bi-directional data bus lines are used to transfer data between the R6522 and the system processor. During read cycles, the contents of the selected R6522 register are placed on the data bus lines and transferred into the processor. During write cycles, these lines are high-impedance inputs and data is transferred from the processor into the selected register. When the R6522 is unselected, the data bus lines are high-impedance.

CS1, $\overline{CS2}$ (Chip Selects)

The two chip select inputs are normally connected to processor address lines either directly or through decoding. The selected R6522 register will be accessed when CS1 is high and $\overline{CS2}$ is low.

RS0–RS3 (Register Selects)

The four Register Select inputs permit the system processor to select one of the 16 internal registers of the R6522, as shown in Figure 6.

Register Number	RS Coding				Register Desig.	Description	
	RS3	RS2	RS1	RS0		Write	Read
0	0	0	0	0	ORB/IRB	Output Register "B"	Input Register "B"
1	0	0	0	1	ORA/IRA	Output Register "A"	Input Register "A"
2	0	0	1	0	DDRB	Data Direction Register "B"	
3	0	0	1	1	DDRA	Data Direction Register "A"	
4	0	1	0	0	T1C-L	T1 Low-Order Latches	T1 Low-Order Counter
5	0	1	0	1	T1C-H	T1 High-Order Counter	
6	0	1	1	0	T1L-L	T1 Low-Order Latches	
7	0	1	1	1	T1L-H	T1 High-Order Latches	
8	1	0	0	0	T2C-L	T2 Low-Order Latches	T2 Low-Order Counter
9	1	0	0	1	T2C-H	T2 High-Order Counter	
10	1	0	1	0	SR	Shift Register	
11	1	0	1	1	ACR	Auxiliary Control Register	
12	1	1	0	0	PCR	Peripheral Control Register	
13	1	1	0	1	IFR	Interrupt Flag Register	
14	1	1	1	0	IER	Interrupt Enable Register	
15	1	1	1	1	ORA/IRA	Same as Reg 1 Except No "Handshake"	

Figure 6. R6522 Internal Register Summary

IRQ (Interrupt Request)

The Interrupt Request output goes low whenever an internal interrupt flag is set and the corresponding interrupt enable bit is a logic 1. This output is "open-drain" to allow the interrupt request signal to be "wire-or'ed" with other equivalent signals in the system.

PA0-PA7 (Peripheral A Port)

The Peripheral A port consists of 8 lines which can be individually programmed to act as inputs or outputs under control of a Data Direction Register. The polarity of output pins is controlled by an Output Register and input data may be latched into an internal register under control of the CA1 line. All of these modes of operation are controlled by the system processor through the internal control registers. These lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. Figure 7 illustrates the output circuit.

CA1, CA2 (Peripheral A Control Lines)

The two Peripheral A control lines act as interrupt inputs or as handshake outputs. Each line controls an internal interrupt flag with a corresponding interrupt enable bit. In addition, CA1 controls the latching of data on Peripheral A port input lines. CA1 is a high-impedance input only while CA2 represents one standard TTL load in the input mode. CA2 will drive one standard TTL load in the output mode.

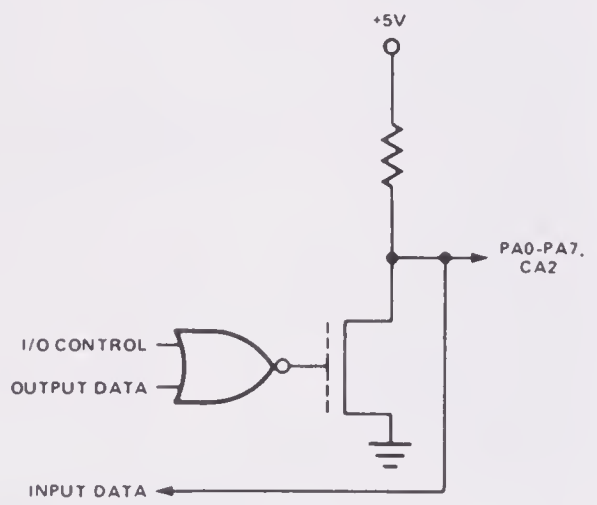


Figure 7. Peripheral A Port Output Circuit

PB0-PB7 (Peripheral B Port)

The Peripheral B port consists of eight bi-directional lines which are controlled by an output register and a data direction register in much the same manner as the PA port. In addition, the polarity of the PB7 output signal can be controlled by one of the interval timers while the second timer can be programmed to count pulses on the PB6 pin. Peripheral B lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. In addition, they are capable of sourcing 1.0mA at 1.5VDC in the output mode to allow the outputs to directly drive Darlington transistor circuits. Figure 8 is the circuit schematic.

CB1, CB2 (Peripheral B Control Lines)

The Peripheral B control lines act as interrupt inputs or as handshake outputs. As with CA1 and CA2, each line controls an interrupt flag with a corresponding interrupt enable bit. In addition, these lines act as a serial port under control of the Shift Register. These lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. Unlike PB0-PB7, CB1 and CB2 cannot drive Darlington transistor circuits.

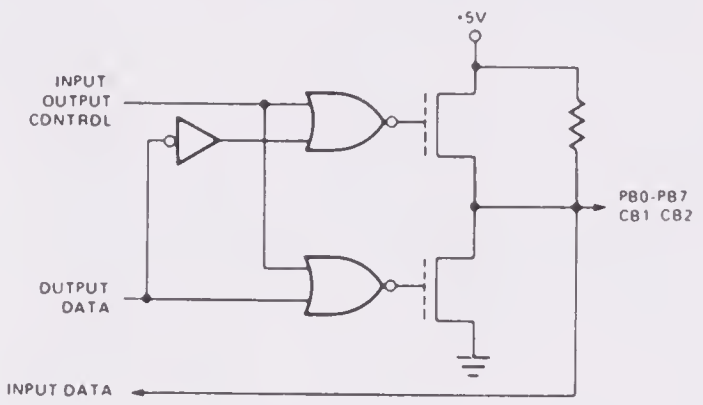


Figure 8. Peripheral B Port Output Circuit

FUNCTIONAL DESCRIPTION

Port A and Port B Operation

Each 8-bit peripheral port has a Data Direction Register (DDRA, DDRB) for specifying whether the peripheral pins are to act as inputs or outputs. A 0 in a bit of the Data Direction Register causes the corresponding peripheral pin to act as an input. A 1 causes the pin to act as an output.

Each peripheral pin is also controlled by a bit in the Output Register (ORA, ORB) and an Input Register (IRA, IRB). When the pin is programmed as an output, the voltage on the pin is controlled by the cor-

responding bit of the Output Register. A 1 in the Output Register causes the output to go high, and a "0" causes the output to go low. Data may be written into Output Register bits corresponding to pins which are programmed as inputs. In this case, however, the output signal is unaffected.

Reading a peripheral port causes the contents of the Input Register (IRA, IRB) to be transferred onto the Data Bus. With input latching disabled, IRA will always reflect the levels on the PA pins. With input latching enabled, IRA will reflect the levels on the PA pins at the time the latching occurred (via CA1).

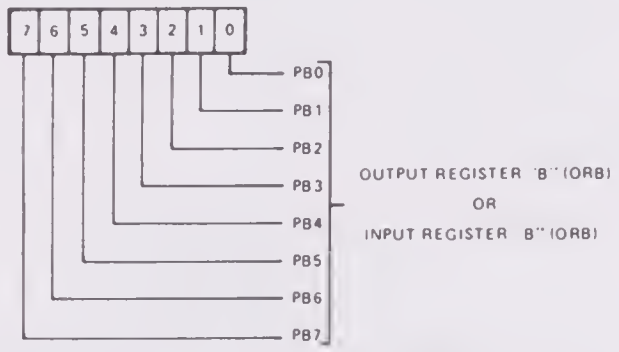
The IRB register operates similar to the IRA register. However, for pins programmed as outputs there is a difference. When reading IRA, the level on the pin determines whether a 0 or a 1 is sensed. When reading IRB, however, the bit stored in the output register, ORB, is the bit sensed. Thus, for outputs which have large loading effects and which pull an output "1" down or which pull an output "0" up, reading IRA may result in reading a "0" when a "1" was actually programmed, and reading a "1" when a "0" was programmed. Reading IRB, on the other hand, will read the "1" or "0" level actually programmed, no matter what the loading on the pin.

Figures 9, 10, and 11 illustrate the formats of the port registers. In addition, the input latching modes are selected by the Auxiliary Control Register (Figure 16.)

Handshake Control of Data Transfers

The R6522 allows positive control of data transfers between the system processor and peripheral devices through the operation of "handshake" lines. Port A lines (CA1, CA2) handshake data on both a read and a write operation while the Port B lines (CB1, CB2) handshake on a write operation only.

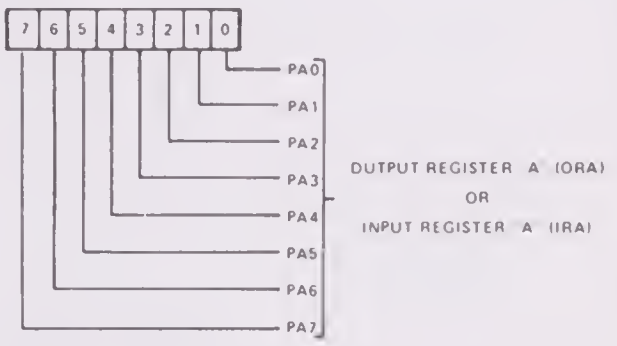
REG 0 – ORB/IRB



Pin Data Direction Selection	WRITE	READ
DORB = "1" (OUTPUT)	MPU writes Output Level (ORB)	MPU reads output register bit in ORB. Pin level has no affect
DORB = "0" (INPUT) (Input latching disabled)	MPU writes into ORB, but no effect on pin level, until DORB changed	MPU reads input level on PB pin
DDRB = "0" (INPUT) (Input latching enabled)		MPU reads IRB bit, which is the level of the PB pin at the time of the last CB1 active transition

Figure 9. Output Register B (ORB), Input Register B (IRB)

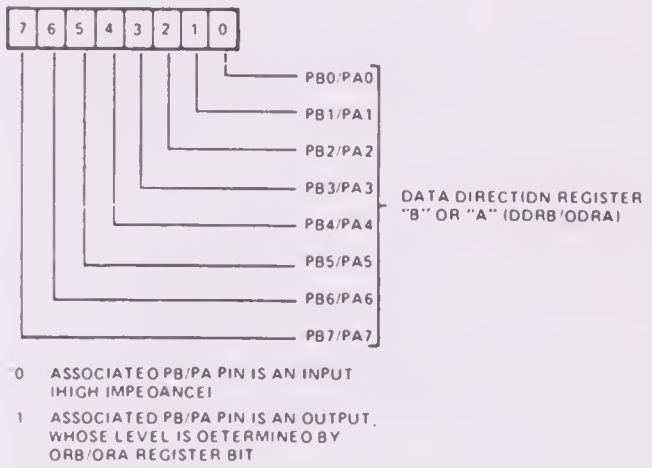
REG 1 – ORA/IRA



Pin Data Direction Selection	WRITE	READ
DORA = "1" (OUTPUT) (Input latching disabled)	MPU writes Output Level (ORA)	MPU reads level on PA pin
DORA = "1" (OUTPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition
DORA = "0" (INPUT) (Input latching disabled)	MPU writes into ORA, but no effect on pin level until DORA changed	MPU reads level on PA pin
DDRA = "0" (INPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition

Figure 10. Output Register A (ORA), Input Register A (IRA)

REG 2 (DDRB) AND REG 3 (DDRA)



0 ASSOCIATED PB/PA PIN IS AN INPUT (HIGH IMPEDANCE)

1 ASSOCIATED PB/PA PIN IS AN OUTPUT, WHOSE LEVEL IS DETERMINED BY ORB/ORA REGISTER BIT

Figure 11. Data Direction Registers (DDRB, DDRA)

Read Handshake

Positive control of data transfers from peripheral devices into the system processor can be accomplished very effectively using Read Handshaking. In this case, the peripheral device must generate the equivalent of a "Data Ready" signal to the processor signifying that valid data is present on the peripheral port. This signal normally interrupts the processor, which then reads the data, causing generation of a "Data Taken" signal. The peripheral device responds by making new data available. This process continues until the data transfer is complete.

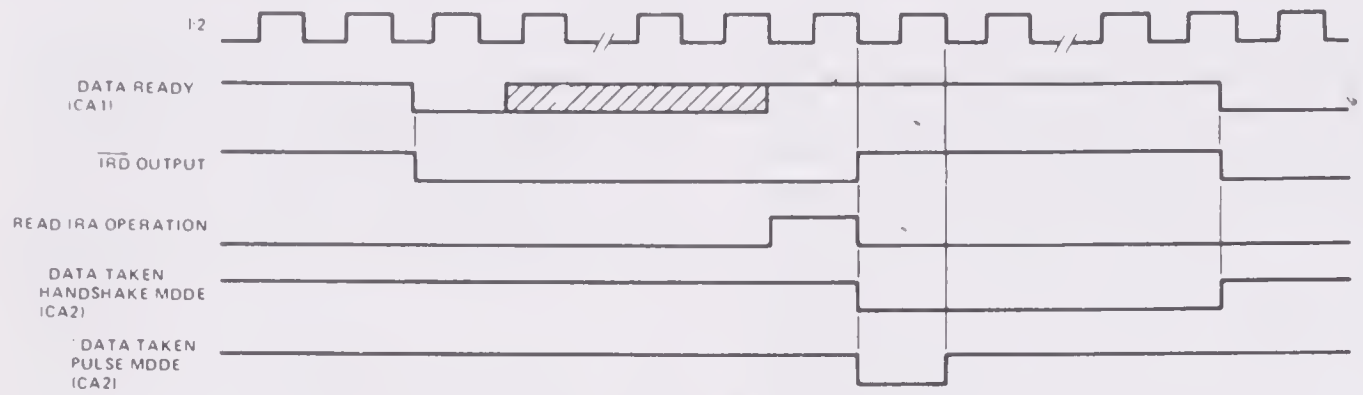


Figure 12. Read Handshake Timing (Port A, Only)

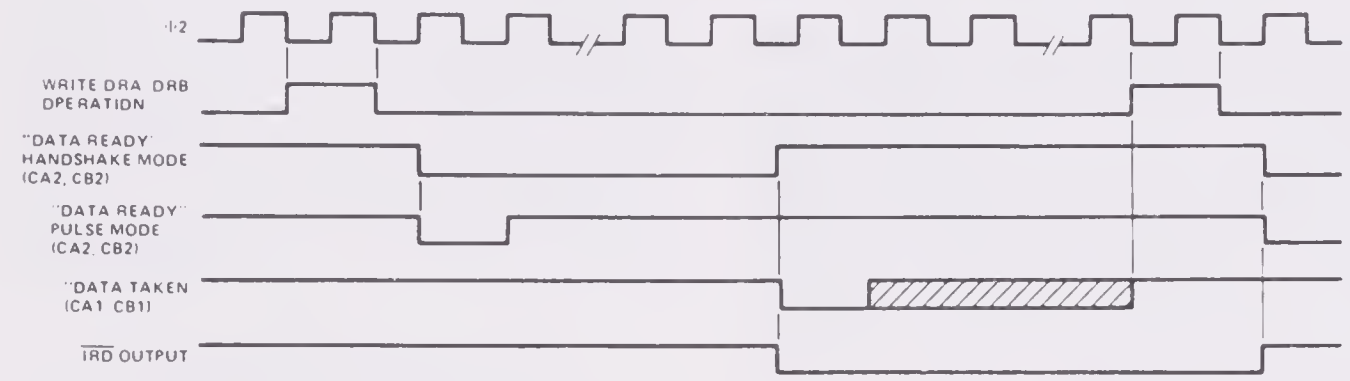


Figure 13. Write Handshake Timing

In the R6522, automatic "Read" Handshaking is possible on the Peripheral A port only. The CA1 interrupt input pin accepts the "Data Ready" signal and CA2 generates the "Data Taken" signal. The "Data Ready" signal will set an internal flag which may interrupt the processor or which may be polled under program control. The "Data Taken" signal can either be a pulse or a level which is set low by the system processor and is cleared by the "Data Ready" signal. These options are shown in Figure 12 which illustrates the normal Read Handshaking sequence.

Write Handshake

The sequence of operations which allows handshaking data from the system processor to a peripheral device is very similar to that described for Read Handshaking. However, for Write Handshaking, the R6522 generates the "Data Ready" signal and the peripheral device must respond with the "Data Taken" signal. This can be accomplished on both the PA port and the PB port on the R6522. CA2 or CB2 act as a "Data Ready." output in either the handshake mode or pulse mode and CA1 or CB1 accept the "Data Taken" signal from the peripheral device, setting the interrupt flag and cleaning the "Data Ready" output. This sequence is shown in Figure 13.

Selection of operating modes for CA1, CA2, CB1, and CB2 is accomplished by the Peripheral Control Register (Figure 14).

Timer Operation

Interval Timer T1 consists of two 8-bit latches and a 16-bit counter. The latches are used to store data which is to be loaded into the counter. After loading, the counter decrements at Φ2 clock rate. Upon reaching zero, an interrupt flag will be set, and $\overline{IRQ}$ will go low if the interrupt is enabled. The timer will then disable any further interrupts, or will automatically transfer the contents of the latches into the counter and will continue to decrement. In addition, the timer may be programmed to invert the output signal on a peripheral pin each time it "times-out". Each of these modes is discussed separately below.

The T1 counter is depicted in Figure 15 and the latches in Figure 16.

REG 12 – PERIPHERAL CONTROL REGISTER

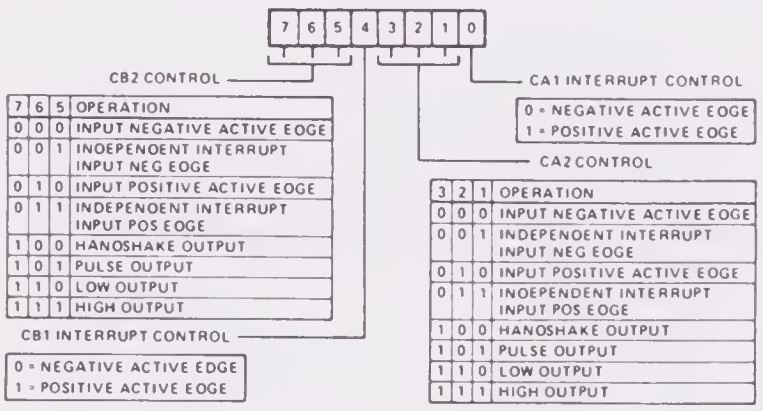
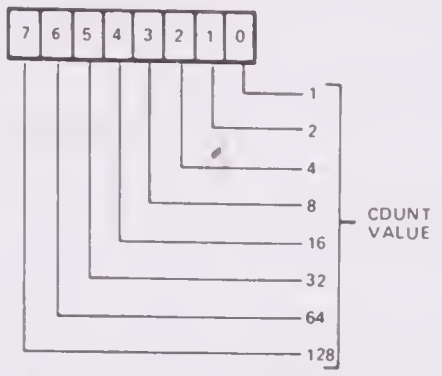


Figure 14. CA1, CA2, CB1, CB2 Control

Two bits are provided in the Auxiliary Control Register (bits 6 and 7) to allow selection of the T1 operating modes. The four possible modes are depicted in Figure 17.

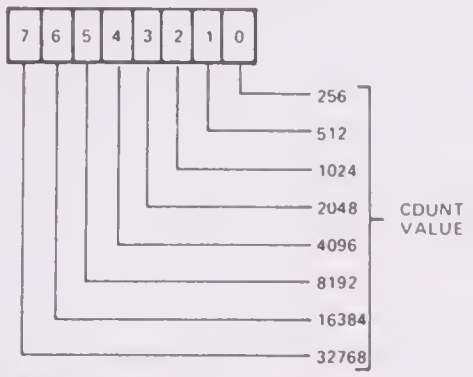
REG 4 – TIMER 1 LOW-ORDER COUNTER



WRITE – 8 BITS LDAOED INTO T1 LDW ORDER LATCHES LATCH CONTENTS ARE TRANSFERRED INTO LDW ORDER COUNTER AT THE TIME THE HIGH OROER COUNTER IS LOAOED (REG 5)

READ – 8 BITS FROM T1 LOW-DRDER COUNTER TRANSFERRED TO MPU IN ADOITION T1 INTERRUPT FLAG IS RESET (BIT 6 IN INTERRUPT FLAG REGISTER)

REG 5 – TIMER 1 HIGH-ORDER COUNTER



WRITE – 8 BITS LOADED INTO T1 HIGH ORDER LATCHES ALSO, AT THIS TIME BDTH HIGH AND LDW DRDER LATCHES TRANSFERRED INTO T1 COUNTER T1 INTERRUPT FLAG ALSO IS RESET

READ – 8 BITS FROM T1 HIGH ORDER COUNTER TRANSFERRED TO MPU

Figure 15. T1 Counter Registers

REG 6 – TIMER 1 LOW-ORDER LATCHES

WRITE – 8 BITS LDADEO INTD T1 LOW ORDER LATCHES THIS OPERATIDN IS NO DIFFERENT THAT A WRITE INTO REG 4

READ – 8 BITS FROM T1 LOW ORDER LATCHES TRANSFERREO TO MPU UNLIKE REG 4 OPERATION, THIS DOES NDT CAUSE RESET OF T1 INTERRUPT FLAG

REG 7 – TIMER 1 HIGH-ORDER LATCHES

WRITE – 8 BITS LOAOED INTD T1 HIGH ORDER LATCHES UNLIKE REG 4 OPERATION NO LATCH TO COUNTER TRANSFERS TAKE PLACE

READ – 8 BITS FROM T1 HIGH DRDER LATCHES TRANSFERRED TO MPU

Figure 16. T1 Latch Registers

REG 11 – AUXILIARY CONTROL REGISTER

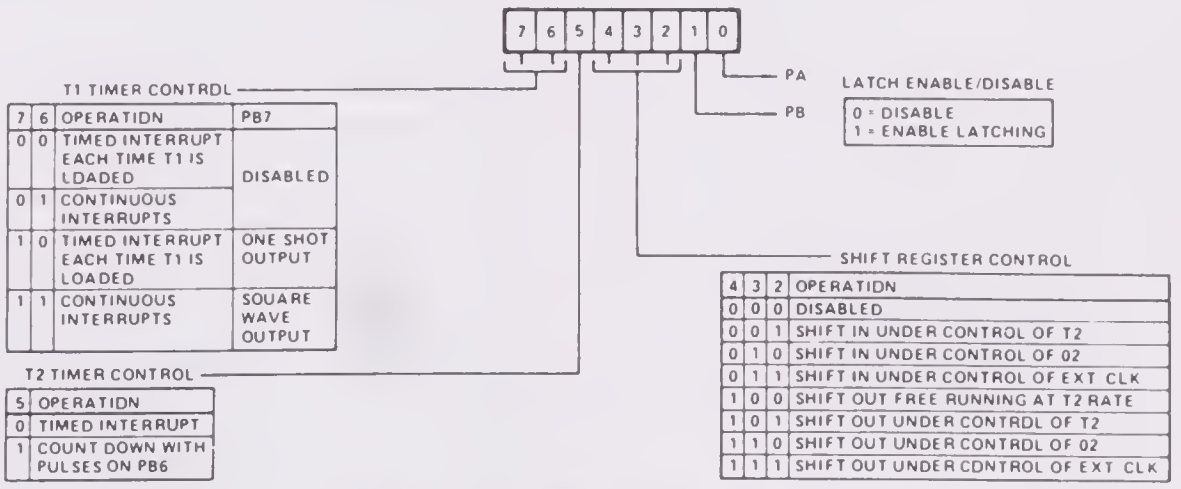


Figure 17. Auxiliary Control Register

Note: The processor does not write directly into the low order counter (T1C-L). Instead, this half of the counter is loaded automatically from the low order latch when the processor writes into the high order counter. In fact, it may not be necessary to write to the low order counter in some applications since the timing operation is triggered by writing to the high order counter.

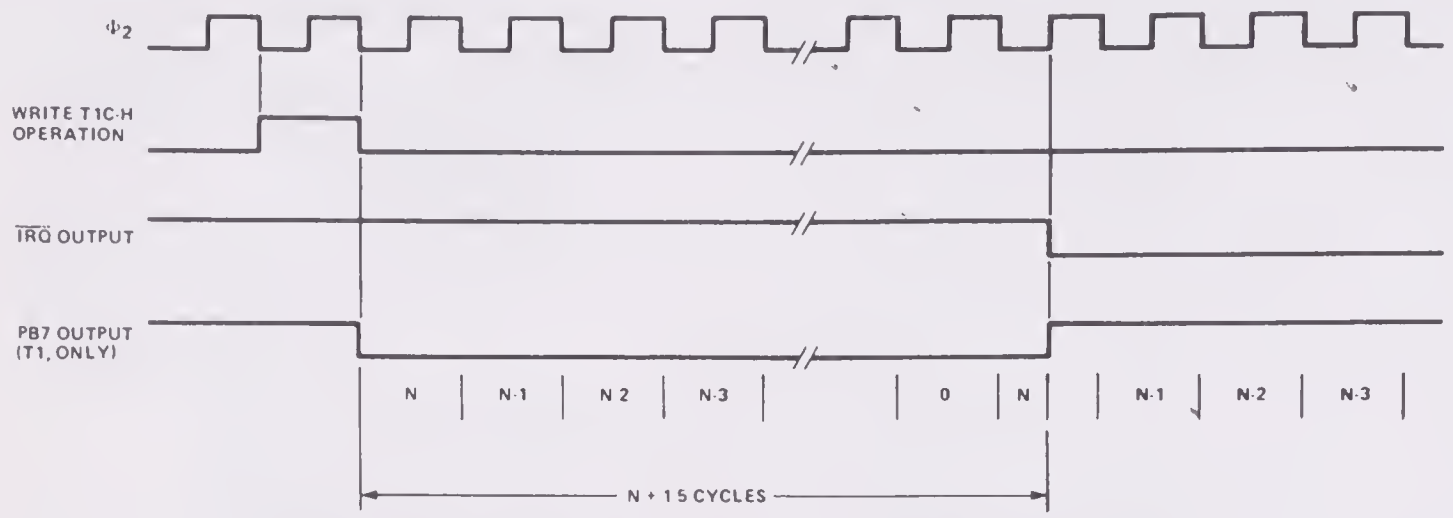


Figure 18. Timer 1 and Timer 2 One-Shot Mode Timing

Timer 1 One-Shot Mode

The interval timer one-shot mode allows generation of a single interrupt for each timer load operation. As with any interval timer, the delay between the "write T1C-H" operation and generation of the processor interrupt is a direct function of the data loaded into the timing counter. In addition to generating a single interrupt, Timer 1 can be programmed to produce a single negative pulse on the PB7 peripheral pin. With the output enabled (ACR7=1) a "write T1C-H" operation will cause PB7 to go low. PB7 will return high when Timer 1 times out. The result is a single programmable width pulse.

In the one-shot mode, writing into the high order latch has no effect on the operation of Timer 1. However, it will be necessary to assure that the low order latch contains the proper data before initiating the count-down with a "write T1C-H" operation. When the processor writes into the high order counter, the T1 interrupt flag will be cleared, the contents of the low order latch will be transferred into the low order counter, and the timer will begin to decrement at system clock rate. If the PB7 output is enabled, this signal will go low on the phase two following the write operation. When the counter reaches zero, the T1 interrupt flag will be set, the $\overline{IRQ}$ pin will go low (interrupt enabled), and the signal on PB7 will go high. At this time the counter will continue to decrement at system clock rate. This allows the system processor to read the contents of the counter to determine the time since interrupt. However, the T1 interrupt flag cannot be set again unless it has been cleared as described in this specification.

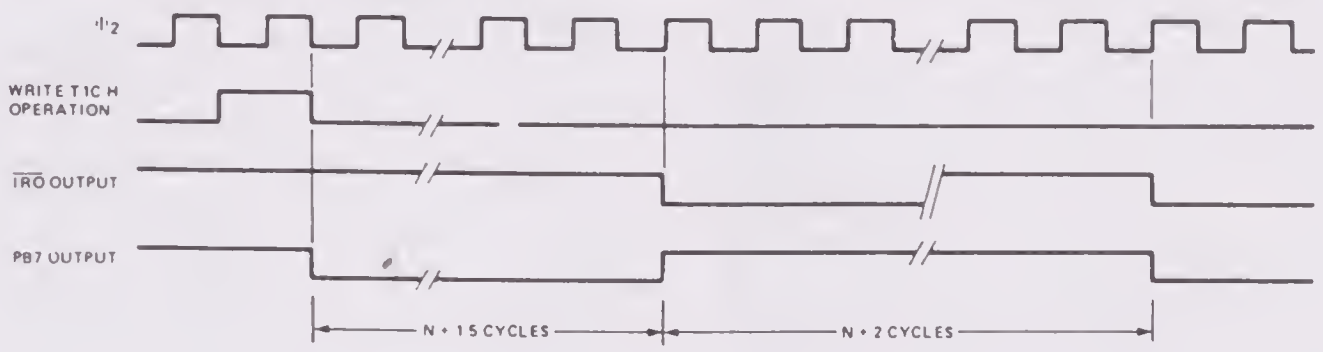
Timing for the R6522 interval timer one-shot modes is shown in Figure 18.

Timer 1 Free-Run Mode

The most important advantage associated with the latches in T1 is the ability to produce a continuous series of evenly spaced interrupts and the ability to produce a square wave on PB7 whose frequency is not affected by variations in the processor interrupt response time. This is accomplished in the "free-running" mode.

In the free-running mode, the interrupt flag is set and the signal on PB7 is inverted each time the counter reaches zero. However, instead of continuing to decrement from zero after a time-out, the timer automatically transfers the contents of the latch into the counter (16 bits) and continues to decrement from there. The interrupt flag can be cleared by writing T1C-H, by reading T1C-L, or by writing directly into the flag as described later. However, it is not necessary to rewrite the timer to enable setting the interrupt flag on the next time-out.

All interval timers in the R6522 are "re-triggerable". Rewriting the counter will always re-initialize the time-out period. In fact, the time-out can be prevented completely if the processor continues to rewrite the timer before it reaches zero. Timer 1 will operate in this manner if the processor writes into the high order counter (T1C-H). However, by loading the latches only, the processor can access the timer during each down-counting operation without affecting the time-out in process. Instead, the data loaded into the latches will determine the length of the next time-out period. This capability is particularly valuable in the free-running mode with the output enabled. In this mode, the signal on PB7 is inverted and the interrupt flag is set with each time-out. By responding to the interrupts with new data for the latches, the processor can determine the period of the next half cycle during each half cycle of the output signal on PB7. In this manner, very complex waveforms can be generated. Timing for the free-running mode is shown in Figure 19.



Note: A precaution to take in the use of PB7 as the timer output concerns the Data Direction Register contents for PB7. Both DDRB bit 7 and ACR bit 7 must be 1 for PB7 to function as the timer output. If one is 1 and the other is 0, then PB7 functions as a normal output pin, controlled by ORB bit 7.

Figure 19. Timer 1 Free-Run Mode Timing

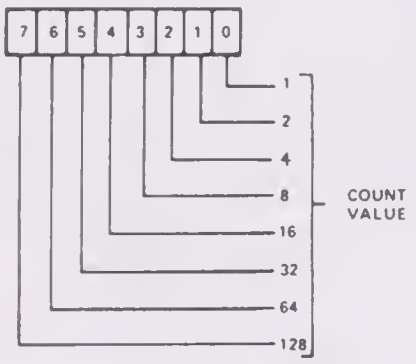
Timer 2 Operation

Timer 2 operates as an interval timer (in the "one-slot" mode only), or as a counter for counting negative pulses on the PB6 peripheral pin. A single control bit is provided in the Auxiliary Control Register to select between these two modes. This timer is comprised of a "write-only" low-order latch (T2L-L), a "read-only" low-order counter and a read/write high order counter. The counter registers act as a 16-bit counter which decrements at Φ2 rate. Figure 20 illustrates the T2 Counter Registers.

Timer 2 One-Shot Mode

As an interval timer, T2 operates in the "one-shot" mode similar to Timer 1. In this mode, T2 provides a single interrupt for each "write T2C-H" operation. After timing out, the counter will continue to decrement. However, setting of the interrupt flag will be disabled after initial time-out so that it will not be set by the counter continuing to decrement through zero. The processor must rewrite T2C-H to enable setting of the interrupt flag. The interrupt flag is cleared by reading T2C-L or by writing T2C-H. Timing for this operation is shown in Figure 18.

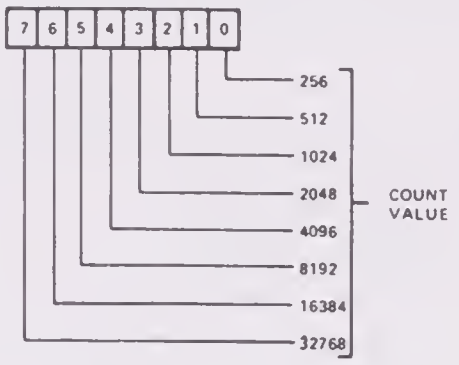
REG 8 – TIMER 2 LOW-ORDER COUNTER



WRITE – 8 BITS LOADED INTO T2 LOW-ORDER LATCHES

READ – 8 BITS FROM T2 LOW-ORDER COUNTER TRANSFERRED TO MPU T2 INTERRUPT FLAG IS RESET

REG 9 – TIMER 2 HIGH-ORDER COUNTER



WRITE – 8 BITS LOADED INTO T2 HIGH-ORDER COUNTER. ALSO, LOW ORDER LATCHES TRANSFERRED TO LOW ORDER COUNTER IN ADDITION, T2 INTERRUPT FLAG IS RESET

READ – 8 BITS FROM T2 HIGH ORDER COUNTER TRANSFERRED TO MPU

Figure 20. T2 Counter Registers

Timer 2 Pulse Counting Mode

In the pulse counting mode, T2 serves primarily to count a predetermined number of negative-going pulses on PB6. This is accomplished by first loading a number into T2. Writing into T2C-H clears the interrupt flag and allows the counter to decrement each time a pulse is applied to PB6. The interrupt flag will be set when T2 counts down past zero. At this time the counter will continue to decrement with each pulse on PB6. However, it is necessary to rewrite T2C-H to allow the interrupt flag to set on subsequent down-counting operations. Timing for this mode is shown in Figure 21. The pulse must be low on the leading edge of $\phi 2$.

Shift Register Operation

The Shift Register (SR) performs serial data transfers into and out of the CB2 pin under control of an internal modulo-8 counter. Shift pulses can be applied to the CB1 pin from an external source or, with the proper mode selection, shift pulses generated internally will appear on the CB1 pin for controlling external devices.

The control bits which select the various shift register operating modes are located in the Auxiliary Control Register. Figure 22 illustrates the configuration of the SR data bits and the SR control bits of the ACR.

Figures 23 and 24 illustrate the operation of the various shift register modes.

Interrupt Operation

Controlling interrupts within the R6522 involves three principal operations. These are flagging the interrupts, enabling interrupts and signaling to the processor that an active interrupt exists within the chip. Interrupt flags are set by interrupting conditions which exist within the chip or on inputs to the chip. These flags normally remain set until the interrupt has been serviced. To determine the source of an interrupt, the microprocessor must examine these flags in order from highest to lowest priority. This is accomplished by reading the flag register into the processor accumulator, shifting this register either right or left and then using conditional branch instructions to detect an active interrupt.

Associated with each interrupt flag is an interrupt enable bit. This can be set or cleared by the processor to enable interrupting the processor from the corresponding interrupt flag. If an interrupt flag is set to a logic 1 by an interrupting condition, and the corresponding interrupt enable bit is set to a 1, the Interrupt Request Output ($\overline{IRQ}$) will go low. $\overline{IRQ}$ is an "open-collector" output which can be "wire-or'ed" with other devices in the system to interrupt the processor.

In the R6522, all the interrupt flags are contained in one register. In addition, bit 7 of this register will be read as a logic 1 when an interrupt exists within the chip. This allows very convenient polling of several devices within a system to locate the source of an interrupt.

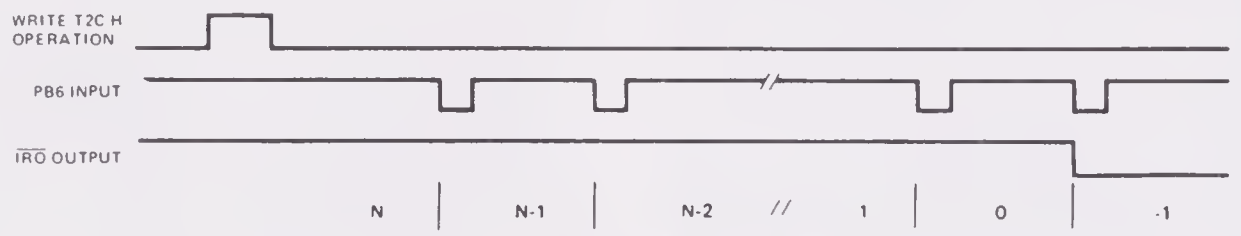
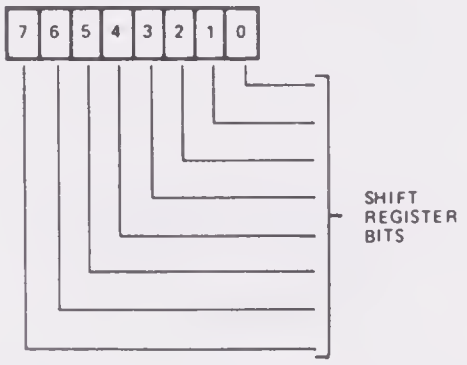


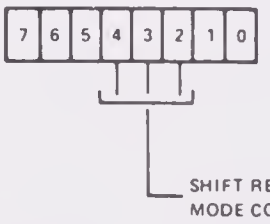
Figure 21. Timer 2 Pulse Counting Mode

REG 10 – SHIFT REGISTER



NOTES
1. WHEN SHIFTING OUT, BIT 7 IS THE FIRST BIT OUT AND SIMULTANEOUSLY IS ROTATED BACK INTO BIT 0.
2. WHEN SHIFTING IN, BITS INITIALLY ENTER BIT 0 AND ARE SHIFTED TOWARDS BIT 7.

REG 11 – AUXILIARY CONTROL REGISTER



4	3	2	OPERATION
0	0	0	DISABLED
0	0	1	SHIFT IN UNDER CONTROL OF T2
0	1	0	SHIFT IN UNDER CONTROL OF $\phi 2$
0	1	1	SHIFT IN UNDER CONTROL OF EXT CLK
1	0	0	SHIFT OUT FREE RUNNING AT T2 RATE
1	0	1	SHIFT OUT UNDER CONTROL OF T2
1	1	0	SHIFT OUT UNDER CONTROL OF $\phi 2$
1	1	1	SHIFT OUT UNDER CONTROL OF EXT CLK

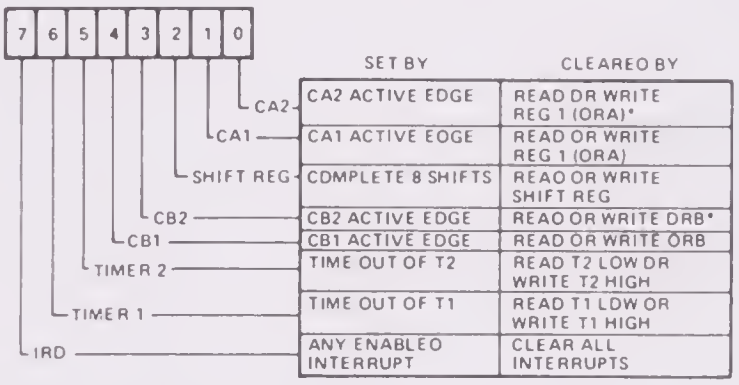
Figure 22. SR and ACR Control Bits

The Interrupt Flag Register (IFR) and Interrupt Enable Register (IER) are depicted in Figures 25 and 26, respectively.

The IFR may be read directly by the processor. In addition, individual flag bits may be cleared by writing a "1" into the appropriate bit of the IFR. When the proper chip select and register signals are applied to the chip, the contents of this register are placed on the data bus. Bit 7 indicates the status of the IRQ output. This bit corresponds to the logic function: IRQ = IFR6 x IER6 + IFR5 x IER5 + IFR4 x IER4 + IFR3 x IER3 + IFR2 x IER2 + IFR1 x IER1 + IFR0 x IER0. Note: X = logic AND, + = Logic OR.

The IFR bit 7 is not a flag. Therefore, this bit is not directly cleared by writing a logic 1 into it. It can only be cleared by clearing all the flags in the register or by disabling all the active interrupts as discussed in the next section.

REG 13 – INTERRUPT FLAG REGISTER



* IF THE CA2 CB2 CONTROL IN THE PCR IS SELECTEO AS "INOEPENDENT" INTERRUPT INPUT THEN READING OR WRITING THE DUTPUT REGISTER ORA/DRB WILL NOT CLEAR THE FLAG BIT INSTEAD, THE BIT MUST BE CLEARED BY WRITING INTO THE IFR, AS DESCRIBEO PREVIOUSLY

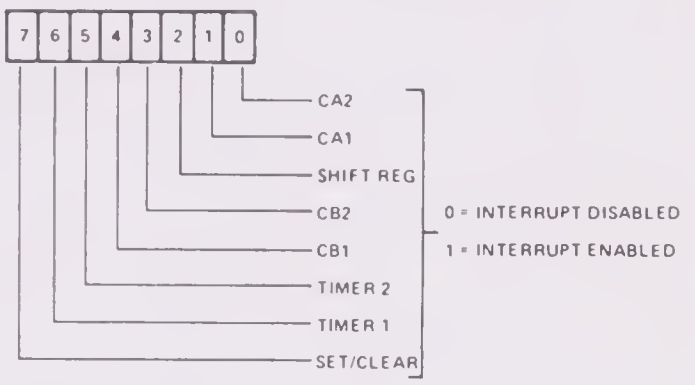
Figure 25. Interrupt Flag Register (IFR)

For each interrupt flag in IFR, there is a corresponding bit in the Interrupt Enable Register. The system processor can be set or clear selected bits in this register to facilitate controlling individual interrupts without affecting others. This is accomplished by writing to address 1110 (IER address). If bit 7 of the data placed on the system data bus during this write operation is a 0, each 1 in bits 6 through 0 clears the corresponding bit in the Interrupt Enable Register. For each zero in bits 6 through 0, the corresponding bit is unaffected.

Setting selected bits in the Interrupt Enable Register is accomplished by writing to the same address with bit 7 in the data word set to a logic 1. In this case, each 1 in bits 6 through 0 will set the corresponding bit. For each zero, the corresponding bit will be unaffected. This individual control of the setting and clearing operations allows very convenient control of the interrupts during system operation.

In addition to setting and clearing IER bits, the processor can read the contents of this register by placing the proper address on the register select and chip select inputs with the R/W line high. Bit 7 will be read as a logic 1.

REG 14 – INTERRUPT ENABLE REGISTER



NOTES
1 IF BIT 7 IS A "0", THEN EACH "1" IN BITS 0 - 6 DISABLES THE CORRESPONDING INTERRUPT
2 IF BIT 7 IS A "1", THEN EACH "1" IN BITS 0 - 6 ENABLES THE CORRESPONDING INTERRUPT
3 IF A READ OF THIS REGISTER IS OONE, BIT 7 WILL BE "1" ANO ALL OTHER BITS WILL REFLECT THEIR ENABLE/DISABLE STATE

Figure 26. Interrupt Enable Register (IER)

APPENDIX 4—LABORATORY INTERFACES AND PARTS LISTS

These are the interfaees required to perform the experiments in this manual. Explanations of the funetions and operations of the individual interfaees are eontained in the experiments. Table F–1 eontains the pin assignments for the AIM's Application (J1) Connector.

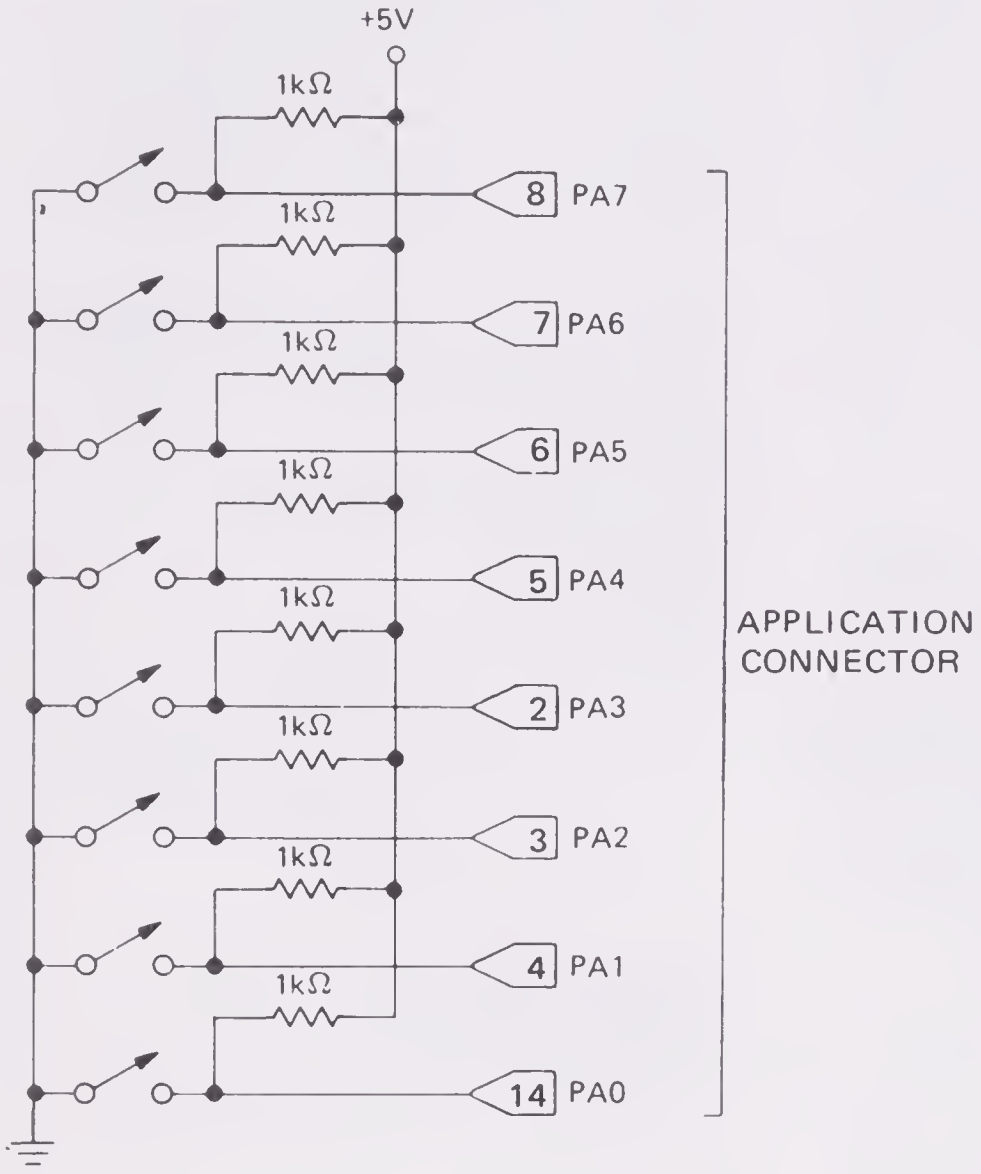


FIGURE A4–1. Attachment of switches to the Application Connector.

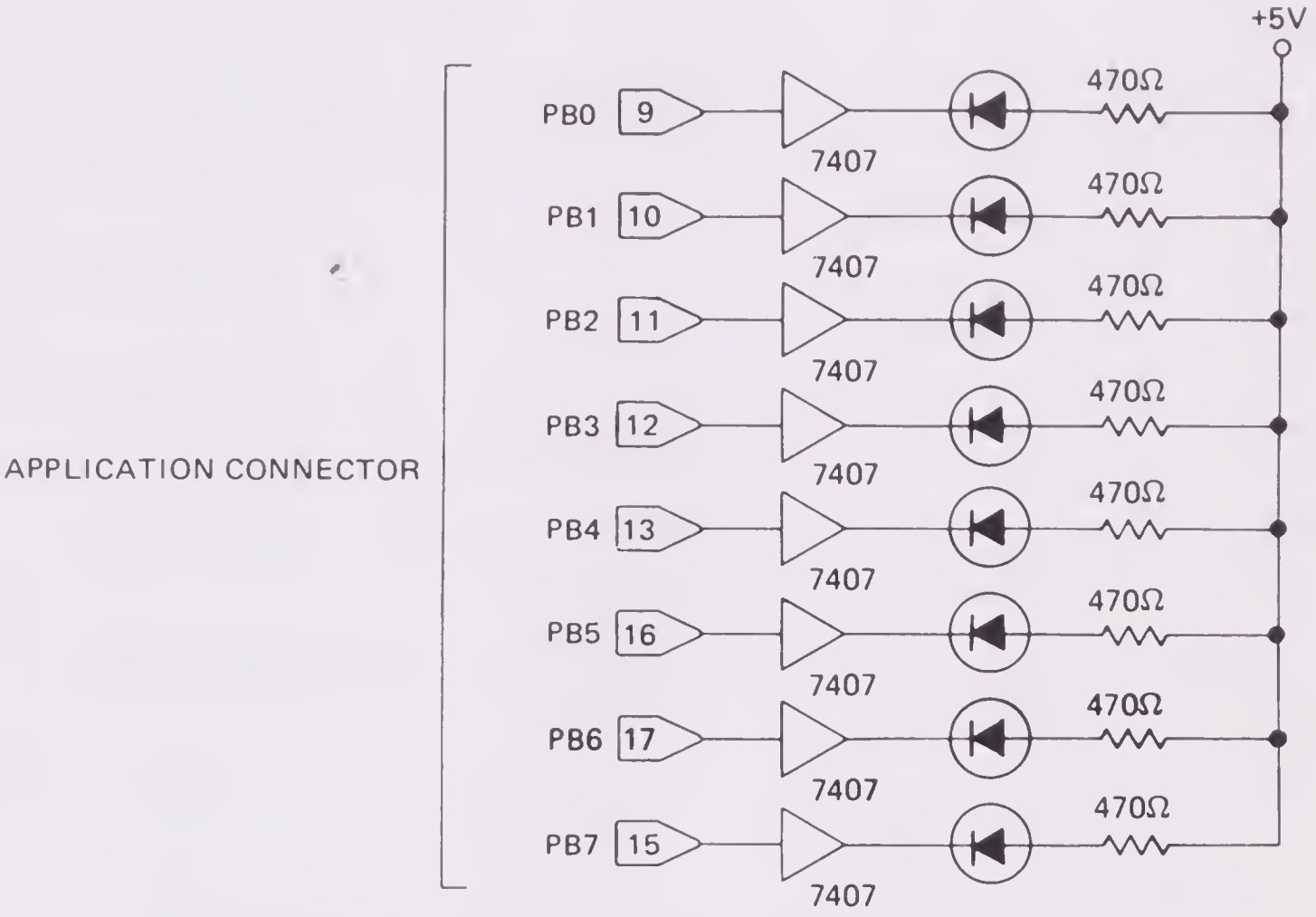


FIGURE A4–2. Attachment of LEDs to the Application Connector.

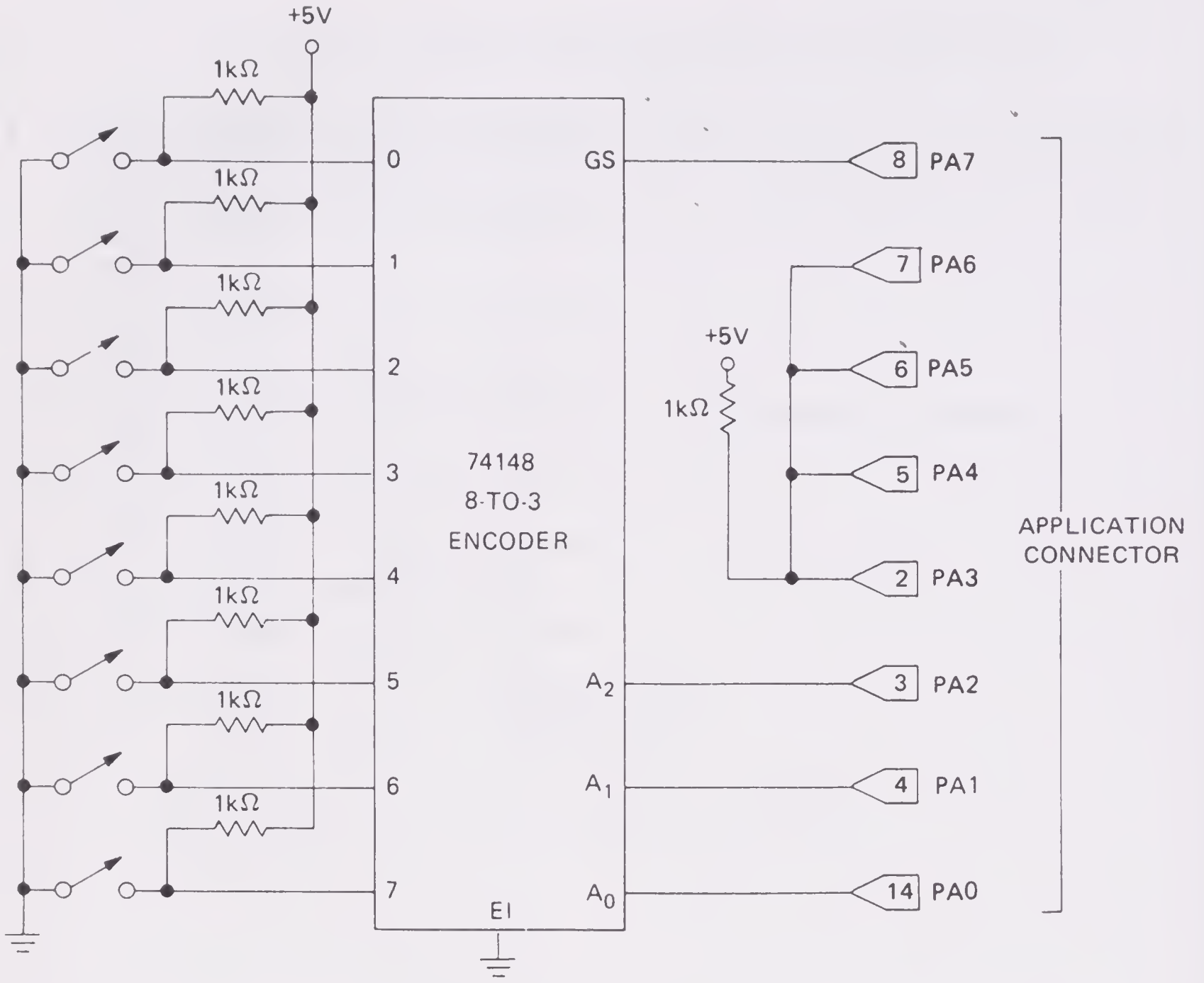


FIGURE A4–3. Attachment of switches and encoder to port A of the user VIA.

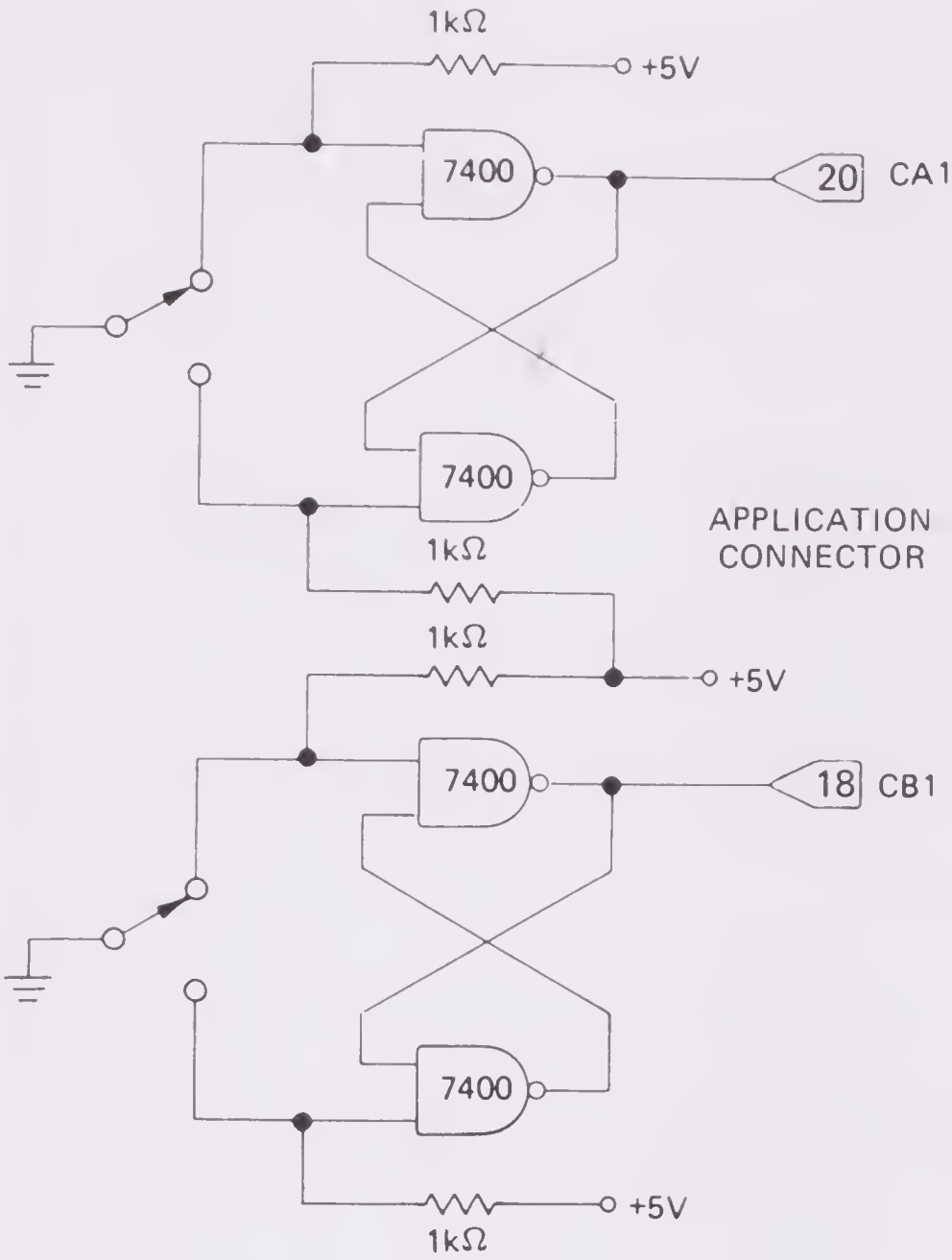


FIGURE A4–4. Attachment of switches to user VIA control lines CA1 and CB1.

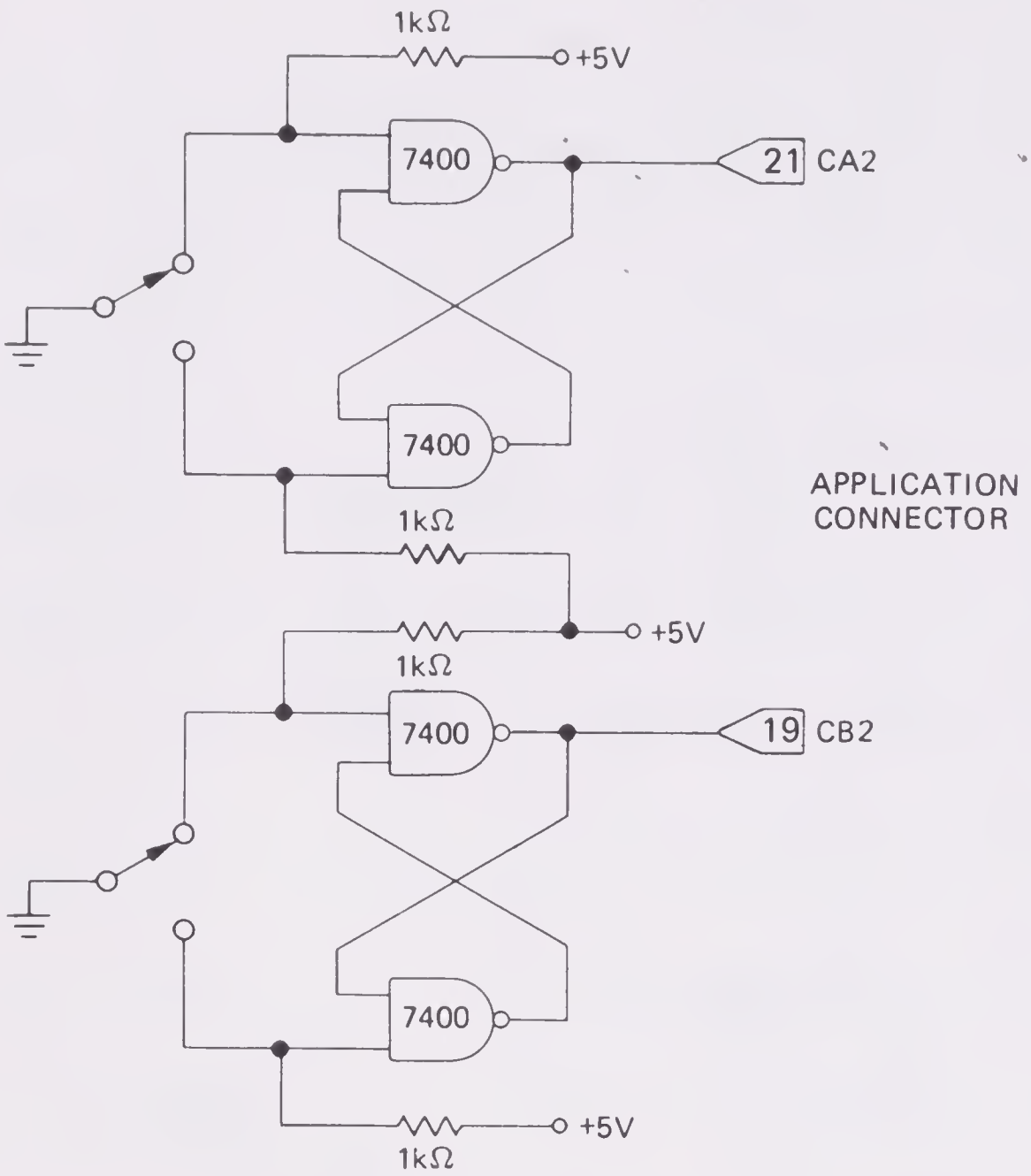


FIGURE A4–5. Attachment of switches to user VIA control lines CA2 and CB2.

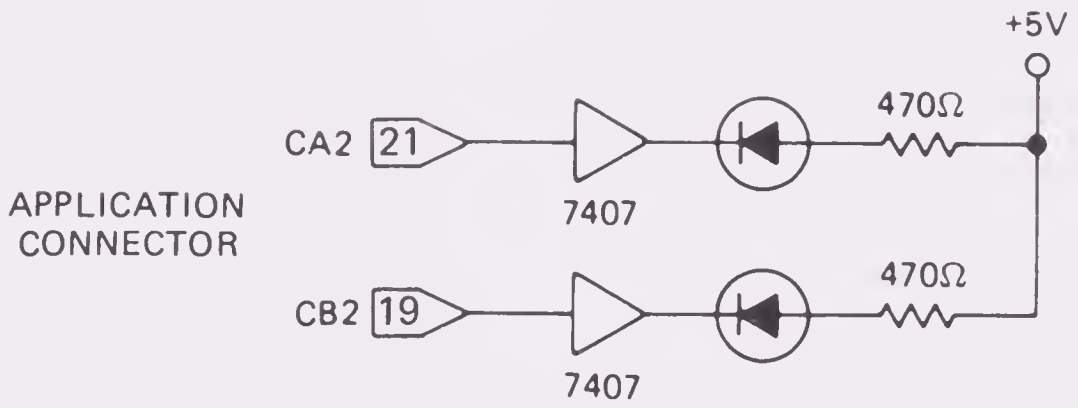


FIGURE A4–6. Attachment of LEDs to user VIA control lines CA2 and CB2. (*Note:* Jumper wires are an easy way to select between the circuits in Figures A4–5 and A4–6; if you are not careful, using CA2 and CB2 as outputs could damage the AND gates in Figure A4–5.)

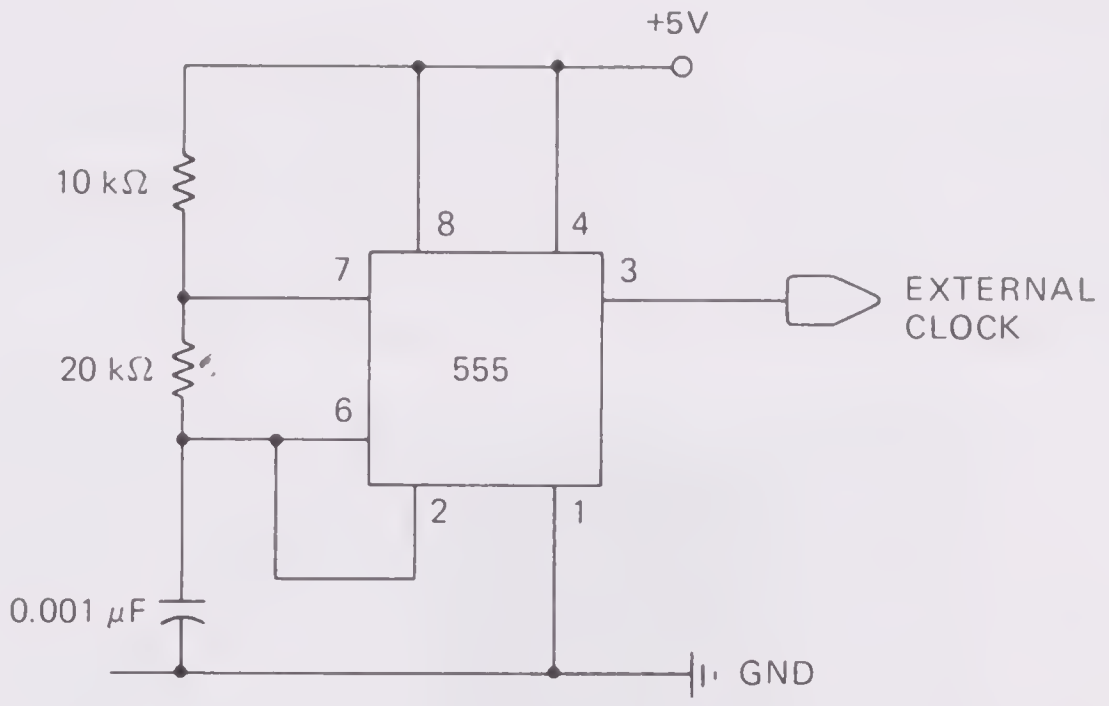


FIGURE A4–7. Simple circuit for generating a low-frequency clock from a 555 timer chip. The frequency is approximately 83 Hz.

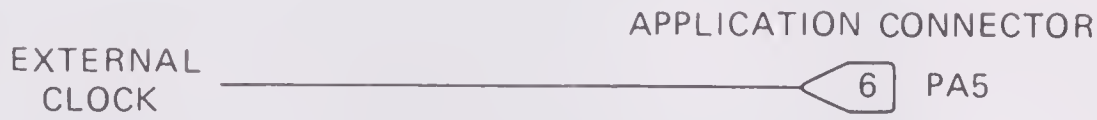


FIGURE A4–8. Connection of the external clock to bit 5 of port A of the user VIA. (*Note:* Jumper wires can be used to select either this input or the switch input shown in Figure A4–1.)

EXAMPLE LABORATORY SETUP

The example laboratory setup was constructed on a Vector 3677-2 prototyping board. Figure A4–9 shows the prototyping board and its connection to the AIM. Figures A4–1 through A4–8 and Table A4–1 describe the circuitry on the board in detail. A complete laboratory interface board (MICROLAB™) is also available from Cambridge Development Laboratory, 36 Pleasant St., Watertown, MA 02172.

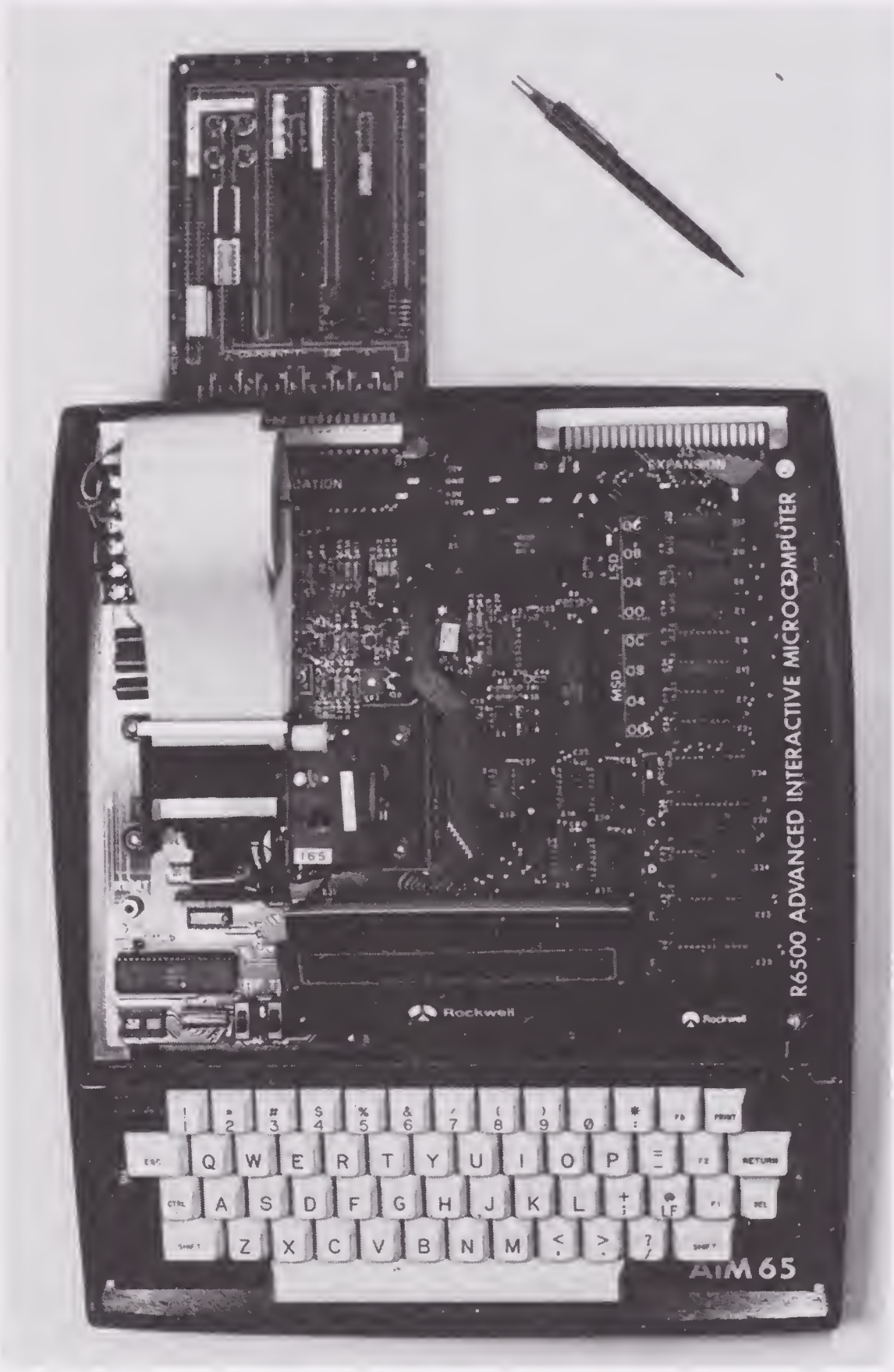


FIGURE A4–9. The AIM 65 microcomputer connected to the laboratory experiment board. (Courtesy of Carter Stafford.)

TABLE A4–1 PARTS LIST FOR LABORATORY EXERCISES

Item	Description	Quantity	Laboratories	User VIA Port
SPDT switch	Alco TT 11DG-WW-2T	8	2,7,B,C,E	A
1-kΩ resistor pack	Bourns 898-1-R1K	1	2,7,B,C,E	A
or 1-kΩ resistors		8	2,7,B,C,E	A
LED display	Red	7	3,C,D,E	B
500-Ω resistor pack	Bourns 898-1-500	1	3,C,D,E	B
or 500-Ω resistors		7		
7407 IC	Hex Buffer/Driver	2	3,C,D,E	B
Decimal switch		1	4	A
74148 IC	Priority Encoder	1	4	A
1-kΩ resistor pack	Bourns 898-1-R1K	1	4	A
SPDT switch	Alco TT 11DG-WW-2T	4	B,C	CA1,CA2,CB1,CB2
1-kΩ resistor pack	Bourns 898-1-R1K	1	B,C	CA1,CA2,CB1,CB2
or 1-kΩ resistors		4		
7400 IC	Quad NAND	1	B,C	CA1,CA2,CB1,CB2
LED display	Red	2	B,C	CA2,CB2
500-kΩ resistor		2	B,C	CA2,CB2
7407 IC	Hex Buffer/Driver	1	B,C	CA2,CB2
10-kΩ resistor		1	D	
20-kΩ resistor		1	D	
0.001-μF capacitor		1	D	
555 timer IC		1	D	A (bit 5)
Miscellaneous:				
Vector prototyping board 3677-2				
44-pin connector		2		
8-pin wire-wrap sockets		1		
14-pin wire-wrap sockets		12		
16-pin wire-wrap sockets		3		

APPENDIX 5—SUMMARY OF THE AIM 65 MONITOR

The following descriptions are taken from the *AIM 65 Summary Card* and are reprinted here with the permission of Dynatem, Irvine, CA.

AIM 65 MONITOR COMMANDS

MAJOR FUNCTION ENTRY COMMANDS

[RESET] — Enter and Initialize Monitor
ROCKWELL AIM 65

E — Enter and Initialize Editor
<E>

T — Re-enter Text Editor at Top of Text
<T>
TOP LINE OF TEXT

N — Enter Assembler
<N>

5 — Enter and Initialize BASIC Interpreter
<5>

6 — Re-enter BASIC Interpreter
<6>

INSTRUCTION ENTRY AND DISASSEMBLY COMMANDS

I — Enter Mnemonic Instruction Entry Mode
<I>
AAAA[*] = [ADDRESS]
AAAA XX [OPCODE][HEX OPERAND]
AAAA XX XX XX

K — Disassemble Memory
<K> * = [ADDRESS]
/[DECIMAL NUMBER]
AAAA XX OPCODE HEX OPERAND

DISPLAY/ALTER REGISTER COMMANDS

* — Alter Program Counter
<*> = [ADDRESS]

A — Alter Accumulator
<A> = [BYTE]

X — Alter X Register
<X> = [BYTE]

Y — Alter Y Register
<Y> = [BYTE]

P — Alter Processor Status
<P> = [BYTE]

S — Alter Stack Pointer
<S> = [BYTE]

R — Display Register Values
<R>
* * * * PS AA XX YY SS
0200 00 00 01 02 FF

DISPLAY/ALTER MEMORY CONTENTS

M — Display Specified Memory Locations
<M> = [ADDRESS] XX XX XX XX

SPACE — Display Next 4 Memory Locations
< > AAAA XX XX XX XX

/ — Alter Current Memory Locations
</> AAAA XX XX XX XX

LOAD/DUMP MEMORY COMMANDS

L — Load Object Code into Memory
<L> IN = [INPUT DEVICE]

D — Dump Memory
<D>
FROM = [ADDRESS] TO = [ADDRESS]
OUT = [OUTPUT DEVICE]
MORE? [Y, N]

BREAKPOINT MANIPULATION COMMANDS

— Clear All Breakpoints
<#> OFF

4 — Toggle Breakpoint Enable
<4> OFF/ON

B — Set/Clear Breakpoint Address
<B> BRK/[0, 1, 2, 3] = [ADDRESS]

? — Display Breakpoint Addresses
<?>
AAAA AAAA AAAA AAAA

AIM 65 MONITOR COMMANDS (Continued)

EXECUTION/TRACE CONTROL COMMANDS

G — Start Execution of User's Program
<G>/[DECIMAL NUMBER]

Z — Toggle Instruction Trace Mode
<Z> ON/OFF

V — Toggle Register Trace Mode
<V> ON/OFF

H — Trace Program Counter History
<H>
AAAA
⋮
AAAA

CONTROL PERIPHERAL DEVICES

CTRL PRINT — Toggle Printer On/Off
<CTRL><PRINT>

PRINT — Print Display Contents
<PRINT>

LF — Advance Printer Paper
<LF>

1 — Toggle Tape 1 Control On/Off
<1>

2 — Toggle Tape 2 Control On/Off
<2>

3 — Tape Verify Block Checksum
<3> IN = [T] F = [FILE NAME] T = [1, 2]

USER FUNCTION COMMANDS

F1 — Call User Function 1 (through loc. $010C)
<F1>

F2 — Call User Function 2 (through loc. $010F)
<F2>

F3 — Call User Function 3 (through loc. $0112)
<F3>

AIM 65 COMMAND DEFINITIONS

[ADDRESS] Hexadecimal address, one to four characters

[BYTE] Two-digit hexadecimal value from 00 to FF.

[DECIMAL NUMBER] A two-digit decimal number in the range 00 to 99.

[FILE NAME] A string of 1 to 5 characters.

[HEX OPERAND] The instruction operand.

Addressing Mode	Operand Format
Accumulator	A
Immediate	#HH
Zero Page	HH
Zero Page, X	HH, X or HHX
Zero Page, Y	HH, Y or HHY
Absolute	HHHH
Absolute, X	HHHH, X or HHHHX
Absolute, Y	HHHH, Y or HHHHY
Relative	HH or HHHH
(Indirect, X)	(HH,X) or (HHX) or (HH,X or (HHX
(Indirect), Y	(HH),Y or (HH)Y
(Absolute Indirect)	(HHHH)

[INPUT DEVICE] RETURN or SPACE — AIM 65 Keyboard (S2 = KB) or TTY Keyboard (S2 = TTY)
M — Memory
T — Audio Tape, AIM 65 format
K — Audio Tape, KIM-1 format
L — TTY Paper Tape Reader
U — User-defined input device

[MNEMONIC OPCODE] A three-letter mnemonic abbreviation

[OUTPUT DEVICE] RETURN or SPACE — AIM 65 Display/Printer (S2 = KB) or TTY Printer (S2 = TTY)
P — AIM 65 Printer
X — Dummy
T — Audio Tape, AIM 65 format
K — Audio Tape, KIM-1 format
L — TTY Paper Tape Punch
U — User-defined output device

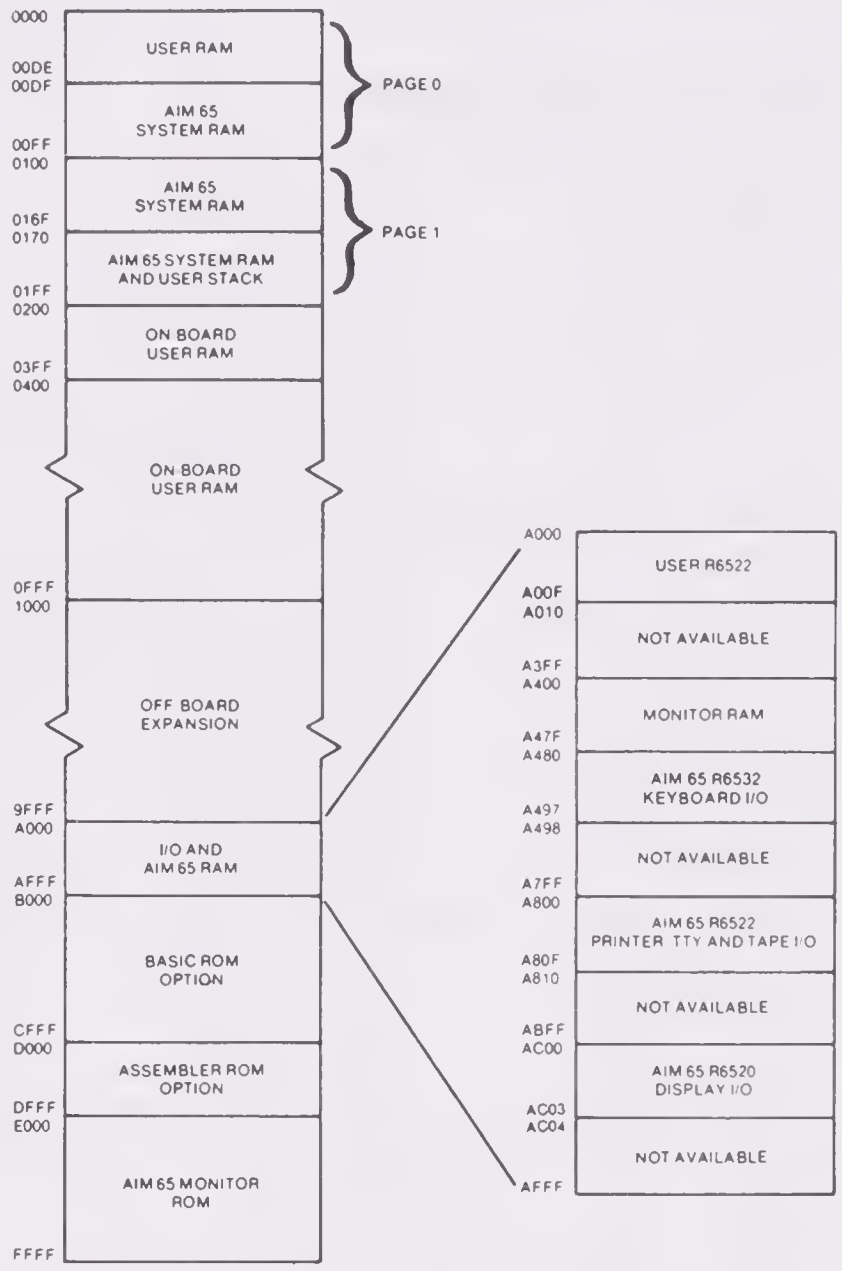
AIM 65 SUBROUTINE SUMMARY

Sub. Name	Entry Addr.	Registers Altered	Function
BLANK	E83E	A	Outputs one SP to D/P
BLANK2	E83B	A	Outputs two SP's to D/P
CLR	EB44	A	Clears D/P pointers
CRCK	EA24	A	Outputs print buffer to Printer
CRLF	E9F0	A	Outputs CR, LF & NUL to AOD
CRLOW	EA13		Outputs CR & LF to D/P
CUREAD	FE83	A	Inputs one ASCII character from KB to A, displays cursor
DEBK1	ED2C	A	Generates a five-millisecond delay
DUMPTA	E56F		Opens Audio Tape output file
EQUAL	E7D8	A	Outputs " = " to D/P
FROM	E7A3	A,X,Y	Outputs "FROM = " to D/P and enters address
GETTAP	EE29	A,Y	Inputs one character from Audio Tape
HEX	EA7D	A	Converts a hex number in A from ASCII to binary, and puts result in the LSD of A, with zero in MSD of A
INALL	E993	A	Inputs one ASCII character from AID to A
INLOW	E8F8	A	Selects KB input in INFLG
LL	E8FE	A	Selects input from KB and output to D/P
LOADTA	E32F		Searches for audio input file
NOUT	EA51	A	Converts a hex number in LSD of A from binary to ASCII, and outputs it to AOD
NUMA	EA46	A	Converts two hex numbers in A from binary to ASCII, and outputs them to AOD, MSD first
OUTALL	E9BC		Outputs ASCII character in A to AOD
OUTLOW	E901	A	Selects output to D/P
OUTPUT	E97A		Outputs ASCII character in A to D/P
PACK	EA84	A	Converts a hex number in A from ASCII to binary, and puts result in the LSD of A, with the result of the last call to PACK or HEX in the MSD of A
PHXY	EB9E		Push X and Y Registers onto Stack
PLXY	EBAC	X,Y	Pull X and Y Registers from Stack
PSL1	E837	A	Outputs "/" to D/P
QM	E7D4	A	Outputs "?" to D/P
RBYTE	E3FD	A	Inputs two ASCII characters from AID, if hex, converts to binary with result in A
RCHEK	E907	A,X,Y	Scans KB, returns to Monitor on ESC, to caller on no entry, wait on SP
RDRUB	E95F	A,Y	Inputs one ASCII character from KB to A, with echo to D/P Allows DEL, if Y ≠ 0
READ	E93C	A	Inputs one ASCII character from KB to A
RED1	FE96	A	Inputs one character from KB to A, with echo to D/P
REDOUT	E973	A	Inputs one ASCII character from KB to A, with echo to D/P, displays cursor
SEMI	E9BA	A	Outputs ";" to AOD
TAISET	EDEA		Sets up Audio Tape input, detects five SYN characters
TAOSET	F21D		Sets up Audio Tape output, issues (GAP) x 4 SYN characters
TIBY1	ED53		Loads a block of 80 bytes from Audio Tape
TO	E7A7	A,X,Y	Outputs "TO" to D/P and enters address
WHEREI	E848	A,X,Y	Sets up the AID and loads INFLG
WHEREO	E871	A,X,Y	Sets up the AOD and loads OUTFLG

ABBREVIATIONS

D/P = Display/Printer
AOD = Active Output Device
AID = Active Input Device

AIM 65 MEMORY MAP



AIM 65 USER-ALTERABLE ADDRESSES

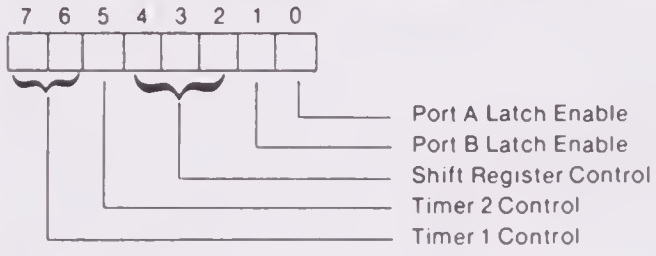
Location	Name	Bytes	Description
0108	UIN	2	Vector to User Input Handler
010A	UOUT	2	Vector to User Output Handler
010C	KEYF1	3	JMP to User Function 1
010F	KEYF2	3	JMP to User Function 2
0112	KEYF3	3	JMP to User Function 3
A400	IRQV4	2	Vector to IRQ after Monitor Interrupt Routine
A402	NMIV2	2	Vector to NMI Interrupt Routine
A404	IRQV2	2	Vector to IRQ Interrupt Routine
A406	DILINK	2	Vector to Display Routine
A408	TSPEED	1	Audio Tape Speed Default = $C7 (AIM 65) Options = $5A (KIM-1 x 1) $5B (KIM-1 x 3)
A409	GAP	1	Audio Tape Gap Default = $08 = 32 SYN characters Option = $80 for Assembler input & Editor update

USER R6522 VERSATILE INTERFACE ADAPTER (VIA)

R6522 MEMORY ASSIGNMENTS

Location	Function		
A000	Port B Output Data Register (ORB)		
A001	Port A Output Data Register (ORA)	**Controls handshake**	
A002	Port B Data Direction Register (DDRB)	0 = Input	
A003	Port A Data Direction Register (DDRA)	1 = Output	
	Timer	**R/$\overline{W}$ = L**	**R/$\overline{W}$ = H**
A004	T1	Write T1L-L	Read T1C-L Clear T1 Interrupt Flag
A005	T1	Write T1L-H & T1C-H T1L-L → T1C-L Clear T1 Interrupt Flag	Read T1C-H
A006	T1	Write T1L-L	Read T1L-L
A007	T1	Write T1L-H Clear T1 Interrupt Flag	Read T1L-H
A008	T2	Write T2L-L	Read T2C-L Clear T2 Interrupt Flag
A009	T2	Write T2C-H T2L-L → T2C-L Clear T2 Interrupt Flag	Read T2C-H
A00A	Shift Register (SR)		
A00B	Auxiliary Control Register (ACR)		
A00C	Peripheral Control Register (PCR)		
A00D	Interrupt Flag Register (IFR)		
A00E	Interrupt Enable Register (IER)		
A00F	Port A Output Data Register (ORA)	*No effect on handshake*	

R6522 AUXILIARY CONTROL REGISTER (ACR), LOC. $A00B



PORT A LATCH ENABLE

ACR0 = 1 Port A latch is enabled to latch input data when CA1 Interrupt Flag (IFR1) is set
= 0 Port A latch is disabled, reflects current data on PA pins

PORT B LATCH ENABLE

ACR1 = 1 Port B latch is enabled to latch the voltage on the pins for the input lines or the ORB contents for the output lines when CB1 Interrupt Flag (IFR4) is set
= 0 Port B latch is disabled, reflects current data on PB pins

SHIFT REGISTER CONTROL

ACR4	ACR3	ACR2	Mode
0	0	0	Shift Register Disabled
0	0	1	Shift in under control of Timer 2
0	1	0	Shift in under control of ø2.
0	1	1	Shift in under control of external clock.
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 ø2.
1	1	1	Shift out under control of external clock.

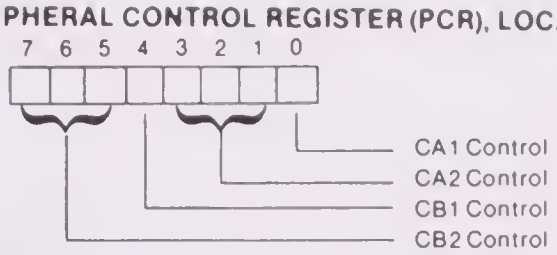
TIMER 2 CONTROL

ACR5 = 0 T2 acts as an interval timer in the one-shot mode
= 1 T2 counts a predetermined number of pulses on PB6.

TIMER 1 CONTROL

ACR7	ACR6	Mode
0	0	T1 one-shot mode — Generate a single time-out interrupt each time T1 is loaded. Output to PB7 disabled.
0	1	T1 free-running mode — Generate continuous interrupts. Output to PB7 disabled.
1	0	T1 one-shot mode — Generate a single time-out interrupt and an output pulse on PB7 each time T1 is loaded
1	1	T1 free-running mode — Generate continuous interrupts and a square wave output on PB7

R6522 PERIPHERAL CONTROL REGISTER (PCR), LOC. $A00C



CA1 CONTROL

PCR0 = 0 The CA1 Interrupt Flag (IFR1) will be set by a negative transition (high to low) on the CA1 pin
= 1 The CA1 Interrupt Flag (IFR1) will be set by a positive transition (low to high) on the CA1 pin.

CA2 CONTROL

PCR3	PCR2	PCR1	Mode
0	0	0	CA2 negative edge interrupt (IFR0/ORA clear) mode — Set CA2 interrupt flag (IFR0) on a negative transition of the CA2 input signal. Clear IFR0 on a read or write of the ORA or by writing logic 1 into IFR0
0	0	1	CA2 negative edge interrupt (IFR0 clear) mode — Set IFR0 on a negative transition of the CA2 input signal. Clear IFR0 by writing logic 1 into IFR0
0	1	0	CA2 positive edge interrupt (IFR0/ORA clear) mode — Set CA2 interrupt flag (IFR0) on a positive transition of the CA2 input signal. Clear IFR0 on a read or write of the ORA or by writing logic 1 into IFR0.
0	1	1	CA2 positive edge interrupt (IFR0 clear) mode — Set IFR0 on a positive transition of the CA2 input signal. Clear IFR0 by writing logic 1 into IFR0
1	0	0	CA2 handshake output mode — Set CA2 output low on a read or write of the Peripheral A Output Register. Reset CA2 high with an active transition on CA1
1	0	1	CA2 pulse output mode — CA2 goes low for one cycle following a read or write of the Peripheral A Output Register
1	1	0	CA2 low output mode — The CA2 output is held low in this mode
1	1	1	CA2 high output mode — The CA2 output is held high in this mode

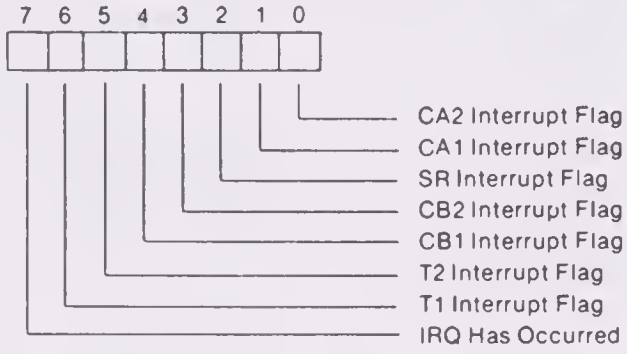
CB1 CONTROL

PCR4 = 0 The CB1 Interrupt Flag (IFR4) will be set by a negative transition (high to low) on the CB1 pin.
= 1 The CB1 Interrupt Flag (IFR4) will be set by a positive transition (low to high) on the CB1 pin

CB2 CONTROL

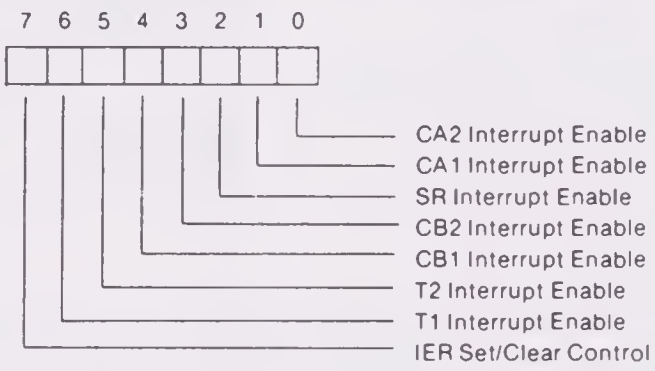
PCR7	PCR6	PCR5	Mode
0	0	0	CB2 negative edge interrupt (IFR3/ORB clear) mode — Set CB2 interrupt flag (IFR3) on a negative transition of the CB2 input signal. Clear IFR3 on a read or write of the ORB or by writing logic 1 into IFR3
0	0	1	CB2 negative edge interrupt (IFR3 clear) mode — Set IFR3 on a negative transition of the CB2 input signal. Clear IFR3 by writing logic 1 into IFR3.
0	1	0	CB2 positive edge interrupt (IFR3/ORB clear) mode — Set CB2 interrupt flag (IFR3) on a positive transition of the CB2 input signal. Clear IFR3 on a read or write of the ORB or by writing logic 1 into IFR3.
0	1	1	CB2 positive edge interrupt (IFR3 clear) mode — Set IFR3 on a positive transition of the CB2 input signal. Clear IFR3 by writing logic 1 into IFR3
1	0	0	CB2 handshake output mode — Set CB2 output low on a write of the Peripheral B Output Register. Reset CB2 high with an active transition on CB1
1	0	1	CB2 pulse output mode — CB2 goes low for one cycle following a read or write of the Peripheral B Output Register
1	1	0	CB2 low output mode — The CB2 output is held low in this mode.
1	1	1	CB2 high output mode — The CB2 output is held high in this mode.

R6522 INTERRUPT FLAG REGISTER (IFR), LOC. $A00D



IFR Bit	Set By	Cleared By
0	Active transition on CA2	Reading or writing the ORA ($A001 or $A00F)
1	Active transition on CA1	Reading or writing the ORA ($A001 or $A00F)
2	Completion of eight shifts	Reading or writing the SR ($A00A)
3	Active transition on CB2	Reading or writing the ORB ($A000)
4	Active transition on CB1	Reading or writing the ORB ($A000)
5	Time-out of Timer 2	Reading T2C-L ($A008) or writing T2C-H ($A009)
6	Time-out of Timer 1	Reading T1C-L ($A004) or writing T1L-H ($A005 or $A007)
7	Any IFR bit set with its corresponding IER bit also set	Clearing IFR0-IFR6 ($A00D) or IER0-IER6 ($A00E)

R6522 INTERRUPT ENABLE REGISTER (IER), LOC. $A00E



INTERRUPT ENABLE BITS (IER0-6)

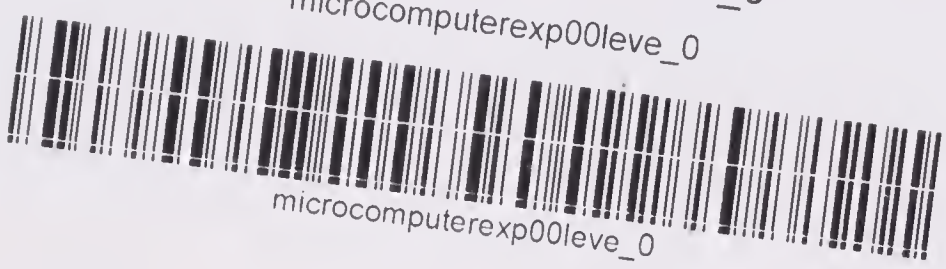
IERn = 0 Disable interrupt
= 1 Enable interrupt

IER SET/CLEAR CONTROL (IER7)

IER7 = 0 For each data bus bit set to logic 1, clear corresponding IER bit
= 1 For each data bus bit set to logic 1, set corresponding IER bit

Note: IER7 is active only when R/$\overline{W}$ = L; when R/$\overline{W}$ = H, IER7 will read logic 1.

INDEX



MICROCOMPUTER EXPERIMENTATION WITH THE AIM 65

LANCE A. LEVENTHAL

This self-contained manual provides experimental training on the AIM 65 microcomputer and utilizes clear, well-written examples and well-documented programs to do so. With this book, users from a wide variety of backgrounds—including students of engineering and engineering technology, computer science, physical sciences, and electronics—can develop solid programming practices.

Microcomputer Experimentation emphasizes the control of systems with software—specifically, the design of controllers for industrial and laboratory use. What's more, it utilizes realistic experiments that can be completed in short time periods.

Here are some of the topics this book covers:

- How to operate the AIM 65
- An introduction to assembly language programming
- How to perform simple controller functions
- Hardware/software tradeoffs
- How to design and develop programs
- Alternative approaches to input/output and timing
- Advantages and uses of programmable LSI devices
- An introduction to hardware design and development

PRENTICE-HALL, INC., Englewood Cliffs, N.J. 07632

ISBN 0-13-580283-0