6502

ASSEMBLY LANGUAGE PROGRAMMING

JUDI N. FERNANDEZ
DONNA N. TABLER
RUTH ASHLEY

Computers $14.95

Boost the power and performance of your micro with

6502 ASSEMBLY LANGUAGE PROGRAMMING
A SELF-TEACHING GUIDE

Assembly language programming is the key to writing faster, more sophisticated programs that tap the full power of your microcomputer. This dynamic guide shows you how to put the power and efficiency of this programming tool—previously available only to professionals—to work for you. The format is self-paced and self-instructional, and the crystal-clear coverage is equally applicable to Apple®, Atari®, and Commodore microcomputers—in fact to any machine based on the 6502 microprocessor chip.

From a discussion of the characteristics and features of 6502 Assembly Language, the authors take you step by step through the entire set of assembly language instructions. You'll get hundreds of opportunities to practice coding typical routines and to check and correct your errors. You'll master techniques for handling routine operations, conditional commands, assembly language logic, subroutines, numeric manipulation, and more.

Scores of sample programs demonstrate how the instructions are used in practice. Abundant summaries and self-tests reinforce the material every step of the way. And, because the book requires only minimal knowledge of programming, even computer novices can easily add the fundamentals of assembly language to their repertoire.

Judi N. Fernandez and **Ruth Ashley** are Co-Presidents of DuoTech, Inc. They are co-authors of three other bestselling Self-Teaching Guides.
Donna N. Tabler is a programmer/analyst with the Naval Air Rework Facility.

STG

More than a million people have learned to program and use microcomputers with Wiley Self-Teaching Guides. Look for them at your favorite bookshop or computer store!

JOHN WILEY & SONS, INC.
605 Third Avenue, New York, N.Y. 10158
New York • Chichester • Brisbane
Toronto • Singapore

Apple® is a registered trademark of Apple Computer, Inc.
Atari® is a registered trademark of Atari, Inc.

ISBN 0 471 86120-0

Cover Photography: Shiah Grumet

6502 ASSEMBLY LANGUAGE PROGRAMMING

Judi N. Fernandez
Donna N. Tabler
Ruth Ashley

JOHN WILEY & SONS, INC.

New York • Chichester • Brisbane • Toronto • Singapore

How To Use This Book

This Self-Teaching Guide consists of 12 chapters that have been carefully designed to introduce you to 6502 Assembly Language and to help you develop useful programming skills. We have made every effort to organize the material in the best possible sequence to learn as quickly as possible. You will first learn to code easy programs, with increasingly more complex programs following, until you have mastered the language.

Each chapter begins with a short introduction followed by objectives that outline what you can expect to learn. A Self-Test at the end allows you to measure your learning and practice what you have studied. Each chapter also contains a review that summarizes the material in the chapter.

The body of each chapter is divided into frames—short numbered sections in which information is presented or reviewed, followed by questions that ask you to apply the material. The correct answers are given immediately after the questions. As you work through the book, use a folded paper or a card to cover the correct answer until you have written yours. And be sure you actually write each response, especially when the activity involves coding Assembly Language instructions. Only by actually writing out the instructions and checking them carefully can you get the most from this Self-Teaching Guide.

Contents

	How To Use This Book	vii
Chapter 1	Introduction	1
Chapter 2	Number Systems and Data Representation	19
Chapter 3	Instruction Format	47
Chapter 4	Operand Formats	61
Chapter 5	Elementary Instruction Set	83
Chapter 6	Assembler Directives	107
Chapter 7	Conditional Instructions	141
Chapter 8	Logical Operations	173
Chapter 9	The Stack	193
Chapter 10	Subroutines	203
Chapter 11	Numeric Manipulation	231
Chapter 12	Additional Instructions	263
	Appendix A	269
	Appendix B	270
	Appendix C	272
	Appendix D	274
	Index	275

CHAPTER ONE

INTRODUCTION

Before you can understand and code instructions, you should know some basic concepts of the language. You should also know how the microcomputer works and what it is.

In this chapter, we'll discuss 6502 Assembly Language and what types of programs it can be used for. We'll also compare it to other computer languages. We'll talk about the 6502 microprocessor itself—what makes up the microprocessor and how data is stored in the computer in bits and bytes. You'll also learn what the registers are and what they are used for in a 6502 microcomputer.

When you complete this chapter, you'll be able to:

- classify 6502 Assembly Language's level and use;
- identify the size and characteristics of bits and bytes;
- name the 6502 registers and their functions.

1. 6502 Assembly Language is used to program computers that contain 6502 microprocessors made by MOS Technology, Inc. and others, or any computer that has a 6502 assembler. An assembler is a program that translates Assembly Language into the correct machine language for the processor. If the processor is a 6502, then the assembler would translate 6502 Assembly Language into 6502 machine language.

Computer languages are usually categorized as *low-level* or ***high-level***. A low-level language is very machine-oriented; the lowest level language is the machine's own language, which is comprised entirely of digits. A high-level language is oriented more to humans; the instructions use English words or abbreviations and English-like syntax.

Assembly Languages are always low-level languages. The language of an assembler is very close to the actual machine language, but uses alphanumeric codes instead of digits. (Alphanumeric code is made up of letters, numbers, and symbols such as *.)

(a) Which of the following describe 6502 Assembly Language? (More than one answer is correct.)

_____ low-level

_____ high-level

_____ uses only digits

_____ uses alphanumeric codes

_____ uses English-like words and phrases

(b) Which of the following types of computers can be programmed using 6502 Assembly Language?

_____ any computer

_____ any computer that has a 6502 microprocessor and an assembler program

_____ any microcomputer

(c) See if you can identify the following instructions as machine language, Assembly Language, or high-level language.

ADD 1 TO COUNTER _______________

000 000 100 011 000 110 _______________

ADC #1 _______________

— — — — — — — — — —

(a) low-level, uses alphanumeric codes; (b) any computer that has a 6502 microprocessor and an assembler program; (c) high-level, machine, Assembly

2. You've probably heard of such high-level languages as COBOL, FORTRAN, and BASIC. These languages have the advantages of being easy to learn and use; but there are disadvantages. A high-level program must be translated into machine language before it can be used. The translation is done by a program called an interpreter or a compiler. This compilation step is time-consuming. Also, the machine-language program produced by the compiler is never the most efficient program possible. One high-level instruction, such as ADD, may be translated into ten or more machine-language instructions.

When you use Assembly Language, you have to think less like a human and more like a computer. For example, to add two numbers you must:

1. move the first number to a special place (the accumulator);
2. add the second number to it;
3. make sure the result isn't too large for the accumulator (check for overflow);
4. do something about overflow if it occurs;
5. check and set the sign (positive or negative) of the result;
6. store the result in memory.

Assembly Language programs are also translated, but it's a much simpler process called assembling. Assembling is a one-to-one translation of the alphanumeric instructions into their machine language counterparts. So when you code in Assembly Language, you're virtually coding in machine language. The alphanumeric codes save you the bother of using the digital form of the instructions.

By coding at the machine level, you can code a much more efficient program than a compiler can produce. This is one advantage of using Assembly Language. Another advantage is that you have more control over the computer. Instructions exist at the Assembly-Language level that have no high-level equivalents. They allow you to access and control computer functions at a very minute level. For example, only in Assembly Language can you directly address a 6502 register, such as the accumulator that is used for arithmetic.

Match the languages with their characteristics.

_____ (a) Assembly		1. more efficient
		2. less efficient
_____ (b) high-level (COBOL, FORTRAN, BASIC, etc.)		3. compiled
		4. assembled
		5. more control
		6. easier to learn and use
		7. interpreted

— — — — — — — — — —

(a) 1,4,5; (b) 2,3,6,7

3. When do you use Assembly Language? Some people use it all the time. but most people use it when they're writing *system* programs as opposed to *application* programs.

An application program is a program that solves some sort of problem for a user. If your computer is in a business setting, then typical applications might be payroll and inventory. If your computer is in a scientific setting, then typical applications might be statistical analysis and graph plotting.

In contrast, a system program is one that solves a problem for the computer system itself. System programs are frequently used by programmers, computer operators, and other computer programs. (An application program will call a system program to read data from a terminal, for example.) Typical system programs are:

- input/output (I/O) routines that transfer data between peripheral devices and main storage. (A peripheral device is a device that is attached by cable to the main part of the computer; terminals, printers, disk, and tape units may be peripheral devices.)
- interpreters and compilers that translate high-level code into machine language.
- librarians that organize disk files and keep up-to-date directories.

An application program may be used once a week or even once a day. A system program may be used several times an hour. Some of the more important system programs, such as the I/O routines, are used several times a second. It's critical that a system program be as efficient as possible. And that's one reason we use Assembly Language to code system programs, even when we have high-level languages available to us. Another reason is to take advantage of that extra measure of control that's available with Assembly Language and not with high-level languages. System programs frequently require us to make full use of the computer's capabilities.

Match the two types of programs with their characteristics.

_____ (a) system programs

_____ (b) application programs

1. typically used by non-computer staff such as accounting department, research staff
2. typically used by computer programmers and operators
3. frequently used by other programs
4. solves computer problems
5. solves user problems
6. typically coded in Assembly Language
7. typically coded in high-level language
8. commonly used on daily, weekly, or monthly basis
9. may be used several times per minute

(c) List two reasons that we usually use Assembly Language for system programs.

- - - - - - - - - -

(a) 2,3,4,6,9; (b) 1,5,7,8; (c) efficiency and control

Any low-level language is involved with the physical structure (architecture) of the system it programs. Before you can begin learning to code Assembly Language instructions, you need to know more about the microprocessor itself. In the next section of this chapter, we'll explore the critical details of the 6502 chip.

BITS AND BYTES

4. A *microprocessor* is an integrated circuit inside the microcomputer that contains the logic that makes the rest of the computer work. The microprocessor contains some control circuits and special storage areas called registers. Main storage (also called internal memory, main memory, internal storage, or core memory) is located on separate chips external to the microprocessor itself. The registers and main storage are both used to store data while it's being worked on.

Data is stored in *bytes* (pronounced "bites"). A byte is the amount of space it takes to store one alphanumeric character such as the letter 'A,' the number '5,' or the symbol '&,' or the space to store a value up to 255. The size of a storage area is usually given in terms of the number of bytes it can hold. 1K stands for 1024 bytes, or 2^{10} bytes.

(a) Which of the following are considered part of the microprocessor?

_____ main storage

_____ registers

_____ peripheral devices

_____ logic circuits

(b) If your computer has 20K bytes of main storage, how many characters can it hold? (K stands for 1024.)_____________________

(c) If a register holds one byte, how many characters can it hold?_____________

- - - - - - - - - -

(a) registers and logic circuits; (b) 20,480 characters; (c) one

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

FIGURE 1. Binary Equivalents

5. To store a character in a byte, it must be encoded as a binary number. In this frame, we'll explain what binary numbers are and why we have to use them.

The binary number system has a base of two, instead of ten as the decimal number system has. It has only two digits—zero and one. Figure 1 shows the binary equivalents for the first ten decimal numbers. You'll be learning a great deal more about the binary number system in the next chapter. For now, the important things to remember are that only two digits are involved and that binary numbers are generally longer than decimal numbers.

Because binary numbers have only two digits, we can represent them electronically. For example, a high voltage in a circuit can represent a one and a low voltage can represent a zero. This is why we use binary numbers rather than decimal numbers inside the computer. The input/output devices, such as the terminal, translate data between decimal and binary.

(a) The binary number system has a base of ______

(b) The digits of the decimal number system are 0,1,2,3,4,5,6,7,8,9. Write the digits of the binary number system. ______

(c) Using Figure 1, what is the binary equivalent of the decimal number 5? ______

(d) Which is easier to represent electronically—a binary number or a decimal number? ______

(e) In order to use a computer, you have to translate all your data into binary numbers before you can type them on the terminal or punch them on cards. True or false? ______

— — — — — — — — — —

(a) two; (b) 0,1; (c) 101; (d) a binary number; (e) false—the I/O devices do the translating

6. One binary digit is called a *bit*. ("Bit" is an acronym for "*BI*nary digi*T*" or maybe "*B*inary dig*IT*.") A bit is either a one or a zero. There are two basic types of data: numeric and alphanumeric. Numeric data can be converted directly to binary and stored in memory.

We use a code system to translate alphanumeric data into binary numbers. It takes several bits to represent one character. The number of bits depends on the code system we use. For example, one popular system is called ASCII (American Standard Code for Information Interchange). It requires seven bits per character. The letter A is encoded as 1000001. The number 5 is encoded as 0110101. The symbol '&' is encoded as 0100110.

Another popular code system is EBCDIC (Extended Binary-Coded Decimal Interchange Code). It uses eight bits per character. The letter A is encoded as 11000001. The number 5 is encoded as 11110101. The symbol '&' is encoded as 01010000.

Remember that a byte holds one character. So a byte holds several bits. In the 6502 microprocessor, one byte contains eight bits. (This is becoming the standard

byte size throughout the computer industry.) So a byte in the 6502 chip is large enough to use either ASCII or EBCDIC code. For numeric data, it can hold a value up to 255.

ASCII is usually pronounced ask'-ey and EBCDIC is usually pronounced ebb'-see-dick.

(a) A binary digit is also called a ______

(b) A bit can have one of two values. What are they? ______

(c) Suppose your microcomputer has 20K bytes of main storage. How many bits does it have? ______

— — — — — — — — — —

(a) bit; (b) 0,1; (c) 160K or 163,840

7. Let's review what you have learned about bits and bytes. Match each term with its characteristics.

______ (a) bit
______ (b) byte

1. holds one character
2. holds a zero or a one
3. holds eight zeros and/or ones
4. the smaller storage area
5. the larger storage area
6. memory size is given in terms of this

— — — — — — — — — —

(a) 2,4; (b) 1,3,5,6

MEMORY

8. Your computer will have some chips that are used for storage of information. They are connected to the microprocessor. These chips comprise your computer's memory. The size of memory depends on the number and types of memory chips your computer has.

Where is your computer's memory?

______ (a) on the microprocessor chip
______ (b) on separate chips connected to the microprocessor
______ (c) on tape or disk

— — — — — — — — — —

(b)

9. Every byte in memory has an *address*. The address is simply a number that refers to that byte. For example, the first byte has the address 0.

The address for the byte does not tell us what is in it, any more than your street address indicates who lives in your house.

(a) What is the address of the second byte in memory?__________

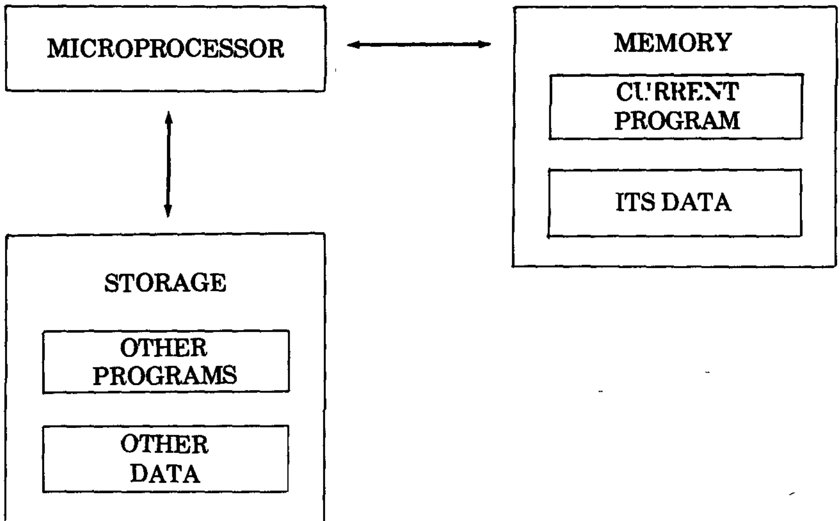
(b) Suppose your computer has 65,536 bytes (64K) of memory. If the first address is 0, what's the address of the last byte?__________

— — — — — — — — — —

(a) 1; (b) 65535

We usually show memory addresses as four-digit hexadecimal numbers, so we would normally write the above two addresses as $0001 and $FFFF. You'll learn how to use hexadecimal numbers in Chapter 2.

10. Memory is used for the storage of programs and data while they're being worked on. The diagram below depicts the relationship of the microprocessor, storage, and memory.



(a) Where does the program currently being executed get stored?

____

(b) Where do programs not currently being executed get stored?

____

(c) What is the main purpose of memory? ______________________________

____

____

— — — — — — — — — —

(a) memory; (b) storage (tapes, disks, and so forth); (c) to hold the current program and its data

REGISTERS

The 6502 microprocessor has several registers. Some have special purposes, and some are available for the programmer to use (general purpose). In this section, we'll discuss a few of the special-purpose registers and all of the general-purpose registers.

11. A register is a very small storage area. Most of the registers store only one byte. A couple of them are two bytes long. The 6502 registers we will study are: A, X, Y, stack pointer, program counter, and status.

The A register is also called the *accumulator* because you use it for addition and subtraction. To add two values, move one value into the A register, then add the second value to it.

The A register contains one byte. It is called a general-purpose register. This means that you, the programmer, can control the contents of the register. The system never changes the contents of the A register unless you tell it to.

(a) How large are most of the registers? ______________________________

(b) The A register is also called the ______________________________

(c) Which of the following is the intended purpose of the A register?

______ to hold the sum of an addition

______ to hold a byte to be output to a terminal

______ to receive input characters from a terminal

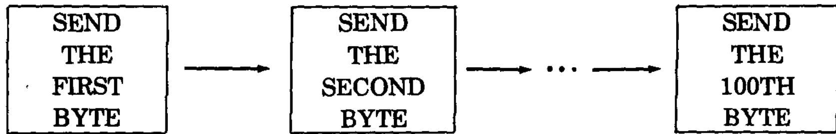
(d) The A register is one byte. What is the maximum value it can hold? (Refer to frame 6 if you don't remember.) ______________________________

— — — — — — — — — —

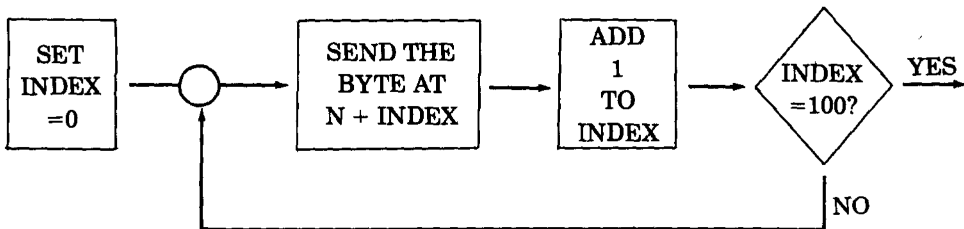
(a) one byte; (b) accumulator; (c) to hold the sum of an addition; (d) 255

12. X and Y are also one-byte, general-purpose registers; they are called the index registers. Their intended purpose needs a bit of explaining.

Suppose you want to access 100 bytes in memory; for example, you want to send a 100-byte message to the terminal. The most direct, but longest and most tedious, method is diagrammed below.



It would take an awful lot of instructions to do it this way; more than 100 instructions. A more economical way is diagrammed below.



We start by initializing an index register to zero. N is a label that addresses the first byte of the message in memory. So, the first time we do the loop, we send the N + 0 byte, or the first byte. Then we add one to the index register. It doesn't equal 100, so we repeat the loop, causing the second byte to be sent. And on it goes until we send the byte at N + 99, which is the 100th byte. After that, the index is incremented to 100 and we leave the loop.

This second technique looks more complicated, but it takes only a few instructions to accomplish. So the use of the index register saves a lot of time and bother.

The 6502 microprocessor, with its X and Y registers, is perfectly designed to use indexing techniques for accessing data in memory. You'll be using these two registers a lot, because you'll rarely want to access just one byte in memory.

(a) Name the 6502 index registers. ______________________

(b) Which of the following best explains the purpose of the 6502 index registers?

_____ to provide additional arithmetic storage space for problems with more than two factors.

_____ to allow you to read or write two bytes at a time.

_____ to provide a means of addressing successive memory bytes without having to code separate instructions.

(c) How many bytes does the X register contain? ______________________

- - - - - - - - - - -

(a) X and Y; (b) to provide a means of addressing successive memory bytes without having to code separate instructions; (c) one

A, X, and Y are the three general-purpose registers of the 6502. Now let's turn our attention to the special-purpose registers.

13. One special-purpose register is called the *stack pointer* (SP). Some Assembly Language instructions allow you to store data in a memory stack and retrieve it again. It's called a stack because of the way it behaves. Imagine a stack of plates. When you add one more, it goes on top. When you remove one, you get the top plate—that is, the one that was stacked last. This is frequently referred to as "last in, first out" or LIFO. This is how a 6502 memory stack works. It is a handy tool in Assembly Language programming, as you will see in Chapter 9.

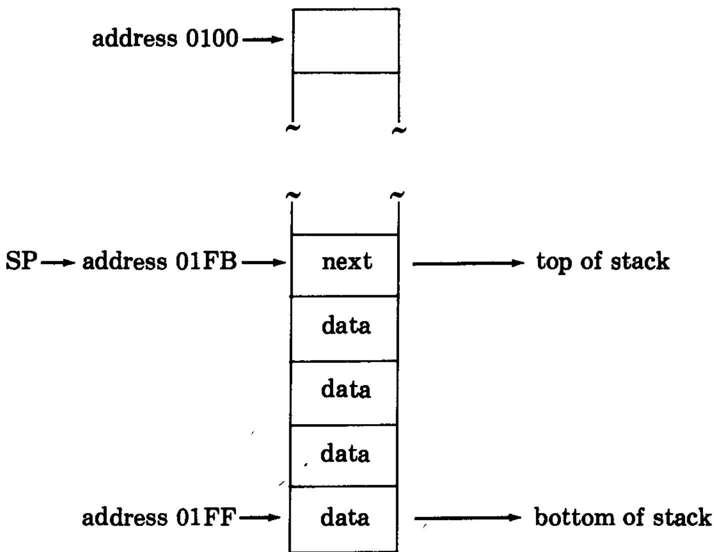


FIGURE 2. A Data Stack in Memory

The stack pointer is a register that always keeps track of, or points to, the current top of the stack in memory. (See Figure 2. Notice that the "top" of the stack is the *lowest* address and the "bottom" of the stack is the *highest* address.) Whenever you add something to the stack, the stack pointer is decreased, or decremented, after the data is stored. Whenever you remove something from the stack, the stack pointer is increased, or incremented. Then, the item is copied from the stack. So the SP is always pointing to the top of the stack.

In the 6502, memory addresses are all two bytes long. Since the stack pointer holds the address of the top of the stack, you'd think it would be a two-byte register. But it isn't. The stack must be in the part of memory where the addresses begin with 01.

The stack pointer holds only one byte—the second byte of the address. The first byte is presumed to be 01. So if the stack pointer says 25, the top of the stack is at 0125.

(a) Where is the stack in memory? ____________

(b) Where does the stack pointer point; the top or bottom of the stack? ______

(c) When you remove something from the stack, what do you get?

______ The first thing that was put in.

______ The last thing that was put in.

(d) How long is a 6502 address? ____________

(e) How large is the stack pointer register? ____________

— — — — — — — — — —

(a) at addresses starting with 01; (b) top; (c) the last thing that was put in; (d) two bytes; (e) one byte

14. The *program counter* register is a double (two-byte) register that tells the computer what to do next.

A computer program is made up of a series of instructions. They are stored in main memory when the program is executed. The instructions are executed one at a time. As each instruction is picked up from memory for execution, the memory address of the first byte of the *next* instruction is stored in the program counter register. When an instruction has finished executing, the computer uses the program counter to find out where to pick up the next instruction.

The programmer has Assembly Language instructions to change the address in the program counter register. These are called jump instructions because they cause the computer to jump to another memory location instead of executing the program in sequence. You will learn to use the jump instructions in this book.

(a) (*review*) How long is a memory address in the 6502 microcomputer? ______

(b) Which register holds the address of the next instruction? ____________

How long is it? ____________

(c) The programmer can change the value in the program counter register with a

__________ instruction.

— — — — — — — — — —

(a) two bytes; (b) program counter, two bytes; (c) jump;

15. Now we'll talk about the status register. The status register is treated as eight separate bits. Seven of the bits are used as flags or indicators. If a flag bit contains a 1, the flag is on or "set." If it contains a 0, the flag is off or "cleared." The flags are set or cleared as a result of operations such as addition, subtraction, or any changes to a register. They tell you about the result of the operation. They tell you

such things as whether the result is positive or negative, whether it overflowed the register, and so forth. There are many Assembly Language instructions that access the values of the flags.

Two of the flags tell something about the result of a data movement or operation.

> The *zero flag* shows whether the result is zero.
>
> The *sign flag* or negative result flag reflects the eighth (high-order) bit of the result. This is useful when you are using signed numbers. You'll learn how to deal with signed numbers in Chapter 11.

Two other flags also indicate something about the result of an operation.

> The *carry flag* shows whether an arithmetic operation needed to carry or borrow outside of the result byte.
>
> The *overflow* flag reflects the seventh bit of the result. Like the sign flag. it is useful when dealing with signed numbers.

The next two flags are used in processing interrupts. You'll learn about these in Chapter 12.

> The *interrupt disable* flag tells the microprocessor whether it's all right to process an interrupt immediately.
>
> The *break* flag shows whether an interrupt is caused by an external event or by a program command.

The final flag you will also learn more about in Chapter 11.

> The *decimal* flag tells the microprocessor whether it is doing binary or decimal arithmetic.

(a) How large is the status register? ____________

(b) How many flags are in the status register? ____________

(c) Which flag tells you if an arithmetic operation resulted in a carry (or a borrow)? ____________

(d) Which flag tells the processor whether you are using binary or decimal numbers? ____________

(e) Which flag tells you about the seventh bit of a result? ____________

(f) Which flag tells you about the source of an interrupt? ____________

(g) Which flag tells you about the high-order bit of a result? ____________

(h) Which flag tells you if an operation resulted in zero? ____________

(i) Which flag tells the processor whether to allow an interrupt? ____________

— — — — — — — — — —

(a) one byte; (b) seven; (c) carry; (d) decimal; (e) overflow; (f) break; (g) sign; (h) zero; (i) interrupt disable

16. Figure 3 depicts the directions in which data may be transferred between registers. You will learn the instructions to cause these data transfers later in this book.

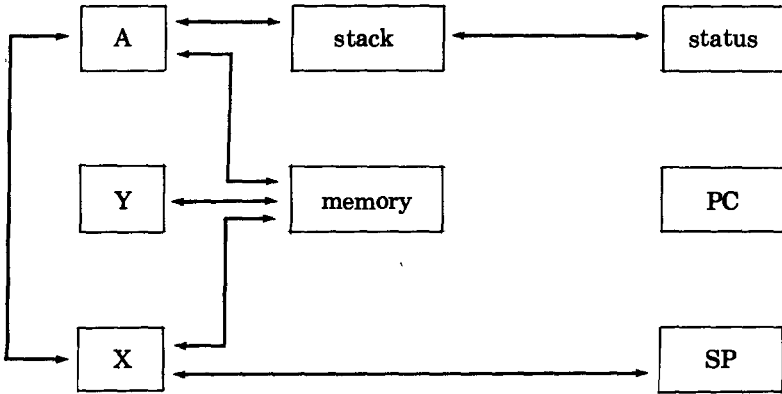


FIGURE 3. 6502 Data Flow

Which of the following transfers are possible in 6502 Assembly Language?

_____ (a) A to memory

_____ (b) X to Y

_____ (c) SP to X

_____ (d) memory to Y

_____ (e) X to A

— — — — — — — — — —

(a), (c), (d), (e)

REVIEW

Here's what you have learned in this chapter:

- Computer languages can be classified as high-level and low-level. A high-level language is more "humanized"—it uses English words and syntax. It is easier to learn and use, but it is less efficient and gives the programmer less control. A low-level language is more like the machine's internal language. It uses alphanumeric codes, and it gives the programmer more control and more efficient programs.

- 6502 Assembly Language is a low-level language used to program microcomputers containing the 6502 microprocessor. It is generally used for writing system programs as opposed to application programs. An application program solves a user problem such as payroll. A system program solves a computer problem such as input/output. System programs are quite heavily used and must be efficient; they must also make full use of the system's capabilities.
- A microprocessor is an integrated circuit containing control circuits, registers, and main storage. A byte is the amount of storage space necessary to hold one character or a number up to 255. In the 6502, a byte contains eight bits. A bit is a binary digit. The binary number system has only two digits, zero and one. Computers use the binary number system because the two digits can be represented electrically. All data is translated into binary codes as it enters the system. Two popular coding systems are ASCII and EBCDIC.
- The 6502 microprocessor has three general purpose registers.
  - The A register, or accumulator, is used for arithmetic operations.
  - The X and Y registers are used for indexed addressing.
- Some of the special-purpose registers are:

  The stack pointer, which holds the address of the top of the stack, a LIFO storage area in the part of memory with addresses starting with 01.

  The program counter, which keeps track of the next instruction in memory. Assembly Language jump instructions are used to change this address.

  The status register, which contains seven on/off flags that reflect the status of certain operations such as addition and subtraction.

  - The carry flag shows a carry or borrow.
  - The zero flag indicates a zero result.
  - The sign flag shows the eighth bit.
  - The overflow flag shows the seventh bit of the result.
  - The interrupt disable flag tells the processor whether to allow interrupts.
  - The break flag indicates the source of a break.
  - The decimal flag tells the processor what kind of arithmetic to use.

Now complete the Self-Test to practice what you have learned.

CHAPTER 1 SELF-TEST

1. Is 6502 Assembly Language a low-level or high-level language? ____________

2. Which of the following are characteristics of system programs?

   ___ a. used by non-computer staff

   ___ b. used by programmers

   ___ c. used by programs

   ___ d. solve business or scientific problems

   ___ e. solve computer system problems

   ___ f. high usage rate

   ___ g. low usage rate

3. The microprocessor contains ____________ and ____________

4. Memory size is stated in terms of ____________

5. In the 6502, how many bits are in a byte? ____________

6. One byte can hold:

   ___ a. one alphanumeric character

   ___ b. a value up to 255

   ___ c. eight characters

   ___ d. a zero or a one only

7. One bit can hold:

   ___ a. one alphanumeric character

   ___ b. a value up to 255

   ___ c. eight characters

   ___ d. a zero or a one only

8. What is memory used for? ____________

   ____

9. What is the address of the first byte of memory? ____________

10. Name the registers described below.
    a. general-purpose registers ____________
    b. holds seven status flags ____________
    c. the accumulator ____________
    d. index registers ____________
    e. holds the next instruction address ____________
    f. points to the stack ____________

11. Name the two-byte register. __________

12. Match the flag names with their descriptions.

______	carry flag	a.	shows carry or borrow status
______	overflow flag	b.	shows whether interrupts are allowed
______	break flag	c.	shows source of an interrupt
______	sign flag	d.	shows the eighth bit of a result
______	zero flag	e.	shows whether a value is zero
______	interrupt disable flag	f.	shows type of arithmetic
______	decimal flag	g.	shows the seventh bit of a result

Self-Test Answer Key

1. low-level
2. b, c, e, f
3. registers and logic circuits
4. bytes
5. eight
6. a and b
7. d
8. to hold the current program and its data
9. 0 (or $0000)

10. a. A, X, Y
    b. status
    c. A
    d. X and Y
    e. program counter
    f. stack pointer

11. program counter

12. carry flag - a
    overflow flag - g
    break flag - c
    sign flag - d
    zero flag - e
    interrupt disable flag - b
    decimal flag - f

If you missed any of these, you may want to review the appropriate frames before going on to Chapter 2.

CHAPTER TWO

NUMBER SYSTEMS AND DATA REPRESENTATION

The binary number system was briefly introduced in Chapter 1. In the study of Assembly Language programming, number systems are so important that they warrant a chapter to themselves. You must become comfortable not only with binary, but also hexadecimal (base 16). numbers.

Of the two code systems we introduced in Chapter 1, ASCII and EBCDIC, your microcomputer probably uses ASCII. EBCDIC is used mainly by larger IBM computers. Therefore, we'll also introduce you to ASCII code in this chapter.

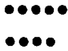
By the time you have finished this chapter, you will be able to:

- add and subtract binary numbers;
- add and subtract hexadecimal numbers;
- convert numbers among binary, decimal, and hexadecimal;
- interpret ASCII codes using a chart.

The Decimal Number System

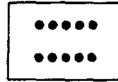
Let's start by reviewing some basic facts about the system you've used every day since the first grade—the decimal number system. This review will give you the concepts and terminology you need to learn about other number systems.

1. The decimal number system has a base of ten. What does that mean? Essentially, it means that we count in groups of ten.

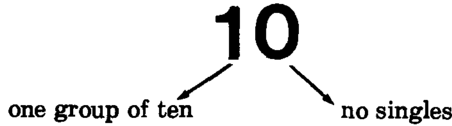


This many objects we call 9.

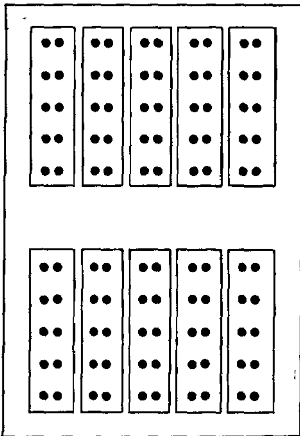
If we add one more, we make one group, which we call 10.

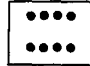


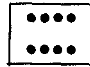
The number 10 means:



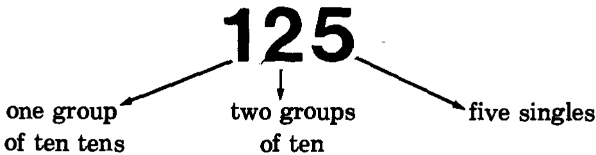
When we get ten groups of ten, we make a larger group.







is written 125.



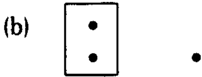
The same principle holds true for all number systems.

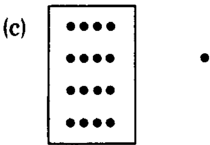
(a) What is the base of the decimal system?________________________

(b) The binary number system has a base of two. Draw the number of objects represented by the binary number 11.

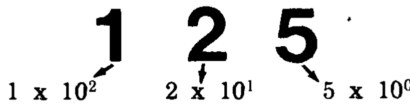
(c) The hexadecimal number system has a base of 16. Draw the number of objects represented by the hexadecimal number 11.

(a) ten

(b)  (one group of two; one single — three objects altogether)

(c)  (one group of sixteen: one single — seventeen objects altogether)

2. Because the decimal system has a base of ten, each column represents a power of ten. The singles, or units, column represents 10^0.



Any number (except 0) to the zero power equals 1. $10^0 = 1$. $5^0 = 1$. $45^0 = 1$. $154387269416333.61527^0 = 1$. $X^0 = 1$, unless $X = 0$.

The second column from the right, the group-of-ten column, represents 10^1. Any number to the first power equals itself. $10^1 = 10$. $5^1 = 5$. $45^1 = 45$. $154387269416333.61527^1 = 154387269416333.61527$. $X^1 = X$.

The third column from the right represents 10^2. The fourth column 10^3, etc. Here is how we break down a decimal number:

10523 =	1×10^4 =	1 x 10000 =	10000
	0×10^3 =	0 x 1000 =	0
	5×10^2 =	5 x 100 =	500
	2×10^1 =	2 x 10 =	20
	3×10^0 =	3 x 1 =	3
			10523

Use the framework below to break down the decimal value 1984.

$1984 =$ ________ $\times 10^3 =$ ________ x ________ = ________

________ $\times 10^2 =$ ________ x ________ = ________

________ $\times 10^1 =$ ________ x ________ = ________

________ $\times 10^0 =$ ________ x ________ = ________

1984

— — — — — — — — — —

$1 \times 10^3 = 1 \times 1000 = 1000$
$9 \times 10^2 = 9 \times 100 = 900$
$8 \times 10^1 = 8 \times 10 = 80$
$4 \times 10^0 = 4 \times 1 = 4$

1984

3. The above shows how you would convert a decimal number to the decimal number system. The result is the same as the original because we didn't change number systems. In other number systems, you use the same method.

The binary value 101 is converted to decimal like this:

$101 = 1 \times 2^2 = 1 \times 4 = 4$
$0 \times 2^1 = 0 \times 2 = 0$
$1 \times 2^0 = 1 \times 1 = 1$

5

The hexadecimal value 11 is converted to decimal like this:

$11 = 1 \times 16^1 = 1 \times 16 = 16$
$1 \times 16^0 = 1 \times 1 = 1$

17

(a) Convert the hexadecimal number 106 to decimal.

(b) Convert the binary number 1101 to decimal.

— — — — — — — — — —

(a) hexadecimal 106 = $1 \times 16^2 = 1 \times 256 = 256$

$0 \times 16^1 = 0 \times 16 = 0$

$6 \times 16^0 = 6 \times 1 = 6$

262

(b) binary 1101 = $1 \times 2^3 = 1 \times 8 = 8$

$1 \times 2^2 = 1 \times 4 = 4$

$0 \times 2^1 = 0 \times 1 = 0$

$1 \times 2^0 = 1 \times 1 = 1$

13

4. Now let's talk about the individual digits in the decimal number system. Decimal is based on ten and so there are ten digits: 0,1,2,3,4,5,6,7,8,9. When you add 1 to 9, you get a group of ten, so you move to the left one column.

$$9 + 1 = 10$$

0 is the lowest value digit and 9 is the highest value digit.

(a) The binary number system has a base of two. How many digits does it need? ________ What are they? ________ What's the lowest value digit? ________ What's the highest value digit? ________

(b) The hexadecimal number system has a base of sixteen. How many digits does it need? ________ Hexadecimal uses letters when it runs out of decimal digits. What do you think the hexadecimal digits are? ________________ What's the lowest value digit? ________ From the digits you used, what's the highest value digit? ________

— — — — — — — — — —

(a) two; 0,1; 0; 1; (b) sixteen; the standard hexadecimal digits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F; 0; F

5. Now you've reviewed the decimal number system and learned quite a bit about binary and hexadecimal systems. These questions will help you to practice what you've learned.

(a) What is the decimal base? ________________

(b) What is the binary base? ________________

(c) What is the hexadecimal base? ________________

Convert the following numbers to decimal.

(d) binary 110 =

(e) hexadecimal 21 =

Give the digits for these number systems.

(f) Decimal ______

(g) Binary ______

(h) Hexadecimal ______

(i) What is the highest digit in decimal? ______

(j) What is the highest digit in binary? ______

(k) What is the highest digit in hexadecimal? ______

— — — — — — — — — —

(a) ten; (b) two; (c) sixteen;

(d) binary 110 = $1 \times 2^2 = 1 \times 4 = 4$
$1 \times 2^1 = 1 \times 2 = 2$
$0 \times 2^0 = 0 \times 1 = \underline{0}$
6

(e) hexadecimal 21 = $2 \times 16^1 = 2 \times 16 = 32$
$1 \times 16^0 = 1 \times 1 = \underline{1}$
33

(f) 0,1,2,3,4,5,6,7,8,9; (g) 0,1; (h) 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F; (i) 9; (j) 1; (k) F

THE HEXADECIMAL NUMBER SYSTEM

You may be saying to yourself, "I see why I might need to understand binary since the computer uses it. But why do I have to learn the hexadecimal number system?"

6. Sometimes we want to communicate with the computer in binary numbers instead of decimal. For example, memory addresses are usually not translated into decimal. But binary numbers are long and awkward and it's easy to make mistakes when reading, writing, and typing them. So instead of using binary directly, we use hexadecimal as a go-between.

Hexadecimal and binary numbers are directly related. Four binary digits equal one hexadecimal digit. So each byte can be expressed in two digits rather than eight. This saves a lot of bother when you are coding instructions to the computer. The computer can easily convert hex to binary; all work is actually done in binary.

```
8000:            1          ORG  $8000
8000: A2 00      2          LDX  #0
8002: BD 16 80   3   MAPLOP LDA  TEXT,X     ; TRANSFER TEXT TO
8005: 9D 00 04   4          STA  $0400,X    ; THE FIRST LINE
8008: E8         5          INX             ; OF THE SCREEN
8009: E0 16      6          CPX  #22        ; MEMORY
800B: D0 F5      7          BNE  MAPLOP
800D: AC 54 C0   8   MAPOUT LDY  $C054      ; SET SCREEN TO
8010: 8C 51 C0   9          STY  $C051      ; PRIMARY TEXT PAGE
8013: 4C 13 80  10   LOOP   JMP  LOOP
8016: 50 4C 45  11   TEXT   ASC  'PLEASE TYPE YOUR NAME:'
8019: 41 53 45
801C: 20 54 59
801F: 50 45 20
8022: 59 4F 55
8025: 52 20 4E
8028: 41 42 45
802B: 3A

*** SUCCESSFUL ASSEMBLY:  NO ERRORS
```

FIGURE 4. Assembler Listing

The computer also prints some data in hexadecimal. For an example, look at Figure 4. This is a portion of an assembler listing—the report we get when a program has been assembled. We have circled the data that has been printed in hexadecimal. It includes memory address information and the machine language instructions. The computer prints them out as hex (hex is short for hexadecimal), but internally only binary is used.

In 6502 Assembly Language, we usually indicate a binary number by the prefix % and a hexadecimal number by the prefix $; a decimal number has no prefix.

(a) What number system is used inside the computer? ____________

(b) What number system is used as a shorthand for binary numbers?

____

Indicate the number system of each of the following values:

(c) %1010 ____________

(d) 10 ____________

(e) $01 ____________

— — — — — — — — — —

(a) binary; (b) hexadecimal; (c) binary; (d) decimal; (e) hexadecimal

7. The binary system is based on two and the hexadecimal system is based on sixteen. Since $2^4 = 16$, there is a direct relationship between a group of four binary digits and a hexadecimal digit.

Figure 5 is a table showing the binary values for all the hexadecimal digits. Use the table to answer the questions below.

decimal	$hexadecimal	%binary
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
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

FIGURE 5. Decimal-Hexadecimal-Binary Equivalents

Give the binary equivalents of these numbers.

(a) $3 = ____________

(b) $9 = ____________

(c) $A = ____________

(d) $C = ____________

Give the hexadecimal equivalents of these numbers.

(e) %1010 = ____________

(f) %1000 = ____________

(g) %0101 = ____________

(h) %1111 = ____________

(i) %0110 = ____________

— — — — — — — — — —

(a) %0011; (b) %1001; (c) %1010; (d) %1100; (e) $A; (f) $8; (g) $5; (h) $F; (i) $6

8. Converting between larger binary and hexadecimal values is also easy. If a binary number has more than four digits, divide it into groups of four starting from the right.

100100101

Fill in leading zeros as necessary to make groups of four digits.

000100100101

Then translate each group into hex.

%000100100101 = $125

Give the hexadecimal equivalents for each of the following numbers.

(a) %101110 = __________

(b) %1111000 = __________

(c) %10000 = __________

— — — — — — — — — —

(a) $2E; (b) $78; (c) $10

9. You can convert hex into binary one digit at a time. For example, $52B is equivalent to %0101 0010 1011. Many programmers like to write binary numbers with a space every four digits to simplify conversion.

Give the binary equivalents for each of the following numbers.

(a) $23 = __________

(b) $FB = __________

— — — — — — — — — —

(a) %0010 0011 or %10 0011; (b) %1111 1011

10. Recall that a 6502 memory address is two bytes, or sixteen bits. So it takes four hex digits to write a memory address. The first memory address is $0000.

(a) What is the second memory address? __________

(b) What is the eleventh memory address? __________

(c) What is the highest possible memory address (the largest hex value that will fit in two bytes)? __________

(d) In decimal, what's the highest possible address? __________

— — — — — — — — — —

(a) \$0001; (b) \$000A; (c) \$FFFF; (d) 65,535

$$
\begin{aligned}
15 \times 16^3 &= 15 \times 4096 = 61440 \\
15 \times 16^2 &= 15 \times 256 = 3840 \\
15 \times 16^1 &= 15 \times 16 = 240 \\
15 \times 16^0 &= 15 \times 1 = 15 \\
& \qquad\qquad\quad 65{,}535
\end{aligned}
$$

(The value 65,535 is the same as $16^4 - 1$ and $2^{16} - 1$.)

11. Here are some more binary-hexadecimal conversion problems for you.

(a) Suppose the A register contains the value \$F0. In binary, what is the value?

____

In decimal? ____________________

(b) Suppose the program counter (PC) contains \$0010. In binary, what is the value?

____

In decimal? ____________________

(c) In the flag register, the least significant (right most) bit is the carry flag. For each of the following values, convert to binary to find out whether the carry flag is on or off.

\$23: ________

\$0A: ________

\$F1: ________

\$FF: ________

— — — — — — — — — —

(a) %1111 0000; 240; (b) %0000 0000 0001 0000; 16; (c) on; off; on; on (Remember the spaces we show aren't really there. We've separated the binary digits into groups of four to make it easier for *you* to read them.)

Since one byte is eight bits, we usually represent a binary value showing all eight bits and a hex value showing two digits, including leading zeros as necessary.

DECIMAL CONVERSIONS

Now you've been introduced to the binary and hexadecimal number systems and can make these conversions: binary to decimal, binary to hexadecimal, hexadecimal to binary, and hexadecimal to decimal. In the following frames, we'll show you how to convert decimal to binary and decimal to hexadecimal.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16,384
15	32,768
16	65,536

FIGURE 6. Powers of Two

n	16^n
0	1
1	16
2	256
3	4096
4	65,536
5	1,048,576
6	16,777,216
7	268,435,456
8	4,294,967,296

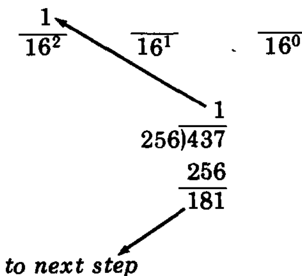
FIGURE 7. Powers of 16

12. Figures 6 and 7 show the values of some powers of two and sixteen, respectively. You'll need them to convert from decimal. To show you how it's done, we'll convert 437 to hexadecimal.

A. First, we find the largest power of 16 that will divide into 437. It's 16^2, or 256. (16^3 (4096) is too big.) From this, we know that our answer is going to have three digits, since we'll have some number times 16^2.

$\overline{16^2}$ $\quad\overline{16^1}$ $\quad\overline{16^0}$

$256\overline{)437}$



B. We divide 256 into 437. The quotient is our first digit because it tells us how many 16^2s there are in 437. We save the remainder for the next step.

C. We divide the remainder by 16^1. The quotient becomes the second digit of the answer. Note that we convert the decimal 11 to a single hexadecimal digit, B.

$$\frac{1}{16^2} \quad \frac{B}{16^1} \quad \frac{\ }{16^0}$$

```
     11
16)181
    16
    21
    16
     5
```

$$\frac{1}{16^2} \quad \frac{B}{16^1} \quad \frac{5}{16^0}$$

```
     11
16)181
    16
    21
    16
     5
```

D. Since we're down to the units column ($16^0 = 1$), the remainder becomes the last digit.

If any quotient or the final remainder comes out greater than 15, a mistake has been made somewhere and you need to recalculate it.

Now convert the following decimal numbers to hexadecimal.

(a) 25 = ________

(b) 100 = ________

(c) 241 = ________

(d) 716 = ________

(e) 4291 = ________

— — — — — — — — — —

(a) 25 = $19

```
     1
16) 25
    16
     9
```

(b) 100 = $64

```
     6
16)100
    96
     4
```

(c) 241 = $F1

```
     15
16)241
   16
    81
    80
     1
```

(d) 716 = $2CC

```
      2            12
256)716        16)204
    512            16
    204            44
                   32
                   12
```

(e) 4291 = $10C3

```
        1           0          12
4096)4291     256)195      16)195
     4096                      16
      195                      35
                               32
                                3
```

13. Decimal to binary conversions are done just the same as decimal to hex except that you use powers of two instead of powers of sixteen. For our example, we'll convert 21 to binary.

A. The largest power of two that will divide into 21 is $2^4 = 16$. This tells us that the answer has five digits.

```
 1
2^4  2^3  2^2  2^1  2^0

        1
   16)21
      16
       5
```

B. The next lower power of two, $2^3 = 8$, produces a zero quotient.

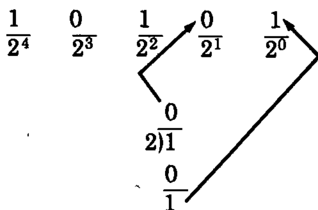
```
 1    0
2^4  2^3  2^2  2^1  2^0

      0
    8)5
      0
      5
```

C. The next lower power of two, $2^2 = 4$, will divide into 5.

```
 1    0    1
2^4  2^3  2^2  2^1  2^0

        1
      4)5
        4
        1
```



D. The next lower power of two, $2^1 = 2$, produces a zero quotient. And the final remainder is 1.

When converting to binary, each quotient will either be 1 or 0. If you get a quotient (or final remainder) larger than 1, you've made a mistake somewhere.

Convert the following decimal numbers to binary.

(a) 10 = __________

(b) 16 = __________

(c) 25 = __________

(d) 33 = __________

— — — — — — — — — —

(a) %1010; (b) %10000; (c) %11001; (d) %100001

14. Now you can convert a number from any one system to any other.

Practice by filling in the chart below.

decimal	hexadecimal	binary
210	(a)	(b)
(c)	$96	(d)
(e)	(f)	%1011 0111
(g)	$22A	(h)
49	(i)	(j)

(k) What is the largest number (in decimal) that the accumulator can hold? ______

— — — — — — — — — —

(a) $D2; (b) %1101 0010; (c) 150; (d) %1001 0110; (e) 183; (f) $B7; (g) 554; (h) %0010 0010 1010; (i) $31; (j) %0011 0001; (k) 255 ($FF or %1111 1111—This is equivalent to $16^2 - 1$ or $2^8 - 1$.)

ADDITION

In order to read assembler listings, you need to be able to do simple addition and subtraction in binary and hexadecimal. We'll cover addition first.

15. Do you remember learning how to add? If you had the usual education, you memorized the addition facts from 1 + 0 through 9 + 9. Then you learned to handle larger numbers in columns.

No, you don't have to memorize math facts in hexadecimal and binary. We'll give you some tables to use. But you should be able to figure out simple addition problems without using the tables.

Here's the hex count from $1 to $20:

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20

Here are some sample problems that you can work out using the hex count line above.

$2 + $3 = $5
$1 + $6 = $7
$A + $0 = $A
$9 + $1 = $A (A critical fact! 10 = $A)
$2 + $B = $D
$7 + $5 = $C (Count on your fingers if you have to: 7,8,9,A,B,C.)
$F + $1 = $10 (Another critical fact! 16 = $10)

Now you can solve the problems below.

(a) $10 + $B = ________

(b) $5 + $F = ________

(c) $9 + $2 = ________

(d) $5 + $5 = ________

(e) $19 + $1 = ________

(f) $1F + $1 = ________

(g) $15 + $B = ________

— — — — — — — — — — —

(a) $1B; (b) $14; (c) $B; (d) $A; (e) $1A; (f) $20; (g) $20

16. The hexadecimal addition and subtraction table is located in Appendix A. To use it for addition, find the row for one addend and the column for the other addend. The intersection gives the sum.

Use the table to solve these problems.

(a) \$5 + \$9 = ________

(b) \$A + \$B = ________

(c) \$3 + \$9 = ________

(d) \$D + \$D = ________

— — — — — — — — — —

(a) \$E; (b) \$15; (c) \$C; (d) \$1A

17. Binary addition is very simple. There are only three math facts.

```
  %0       %0       %1
+ %0     + %1     + %1
  %0       %1      %10
```

See if you can solve the problems below.

```
(a)  %101      (b)  %1000      (c)  %10
      %10              %1           %11
     ____            _____          ___
```

— — — — — — — — — —

(a) %111; (b) %1001; (c) %101

18. Now let's do some problems with carrying.

Decimal examples:

```
    1        11
   257      3265
+   28    + 2149
   285      5414
```

Hexadecimal examples:

```
   /            ///
  $2A          $39FF
+ $1B        +  $B25
  $45          $4524
```

When you carry a 1 to the second column from the right, you are actually carrying $10 or 16.

Binary examples:

```
    //              /////
  %10110          %111101
+    %10        +   %1111
  %11000          %1001100
```

Solve the problems below.

```
(a)    $2B          (b)   %11011
     + $2B              +  %1100
     _____              ________

(c)    $FFF         (d)   %111
     +   $1             +   %1
     ______             ______

(e)    %1110        (f)   $DEF
     +  %111            + $927
     _______            ______
```

— — — — — — — — — —

(a) $56; (b) %100111; (c) $1000; (d) %1000; (e) %10101; (f) $1716

SUBTRACTION

19. To use the hex tables for subtraction, find the minuend (the smaller number) across the top. Then go down that column until you find the subtrahend (the larger number). Go across to the left column to find the answer.

Examples:

```
  $1D        $10        $15
-  $E      -  $8      -  $7
  ---        ---        ---
  $F         $8         $E
```

Problems:

```
(a)   $B      (b)   $10      (c)   $13
    - $5          -  $A          -  $7
    ----          -----          -----

(d)   $17
    -  $B
    -----
```

— — — — — — — — — —

(a) $6; (b) $6; (c) $C; (d) $C

20. Now let's try some subtraction with borrowing.
Decimal examples:

```
   8          20
  4Ø5         ƷΊ6
-   9       -  28
  ---         ---
  486         288
```

Hexadecimal examples:

```
    9/          /
  $4A20        $21
-   $B7      - $1F
  -----        ---
  $4969        $2
```

Now try to work these problems.

```
(a)   $325      (b)   $116      (c)   $10A
    -   $B          -  $2F          -  $BB
    ------          ------          ------

(d)   $3211
    -  $BCD
    -------
```

```
        1              0              F
(a)   $3$\not{2}$5     (b)   $1$\not{1}$6     (c)   $1$\not{0}$A
    -   $B          -  $2F          -  $BB
     $31A            $E7             $4F

(d)   $3211
    -  $BCD
     $2644
```

21. Binary subtraction is simpler than hex subtraction. Here are the binary subtraction facts:

```
 %0      %1      %1      %10     %10
-%0     -%0     -%1     - %1    - %0
 %0      %1      %0      %1      %10
```

The important thing to remember in binary subtraction is that %10 − %1 = %1. Now solve these problems.

```
(a)   %1100     (b)   %10111     (c)   %11111111
    -  %100         -   %101         - %10101010
```

(a) %1000; (b) %10010; (c) %01010101

22. Binary subtraction often requires borrowing. Here are some examples.

```
   00 0          0
 %111010       %10101
-   %1101     -   %10
 %101101       %10011
```

Find the answers to these problems.

```
(a)  %11110      (b)  %101010
   -  %1101         -   %0101
```

— — — — — — — — — —

```
(a)       0      (b)    0
     %11110           %101010
   -  %1101         -   %0101
     %10001           %100101
```

23. Sometimes you have to borrow through zero. This is the trickiest part of subtraction. Let's look at how it works in decimal. We'll use the problem 50001 − 16.

A. We borrow from the first non-zero digit. ALL THE INTERVENING ZEROS CHANGE TO HIGHEST VALUED DIGITS. (Don't think of them as 9's. think of them as highest digits.) The borrowing column receives 10, so it becomes 11.

```
  4999
  50001
-    16
```

```
  4999
  50001
-    16
      5
```

B. The units column is then completed.

C. The remainder of the problem is subtracted in normal fashion.

```
  4999
  50001
-    16
```

Here are some hex examples:

```
   9FF1          0FFFF          5F
 $10021       $3100000B       $1601
-   $49      -       $F      - $107
 $9FFD8       $30FFFFC       $14FA
```

Here are some binary examples:

```
  01             01110            011
 %10010       %1000101         %1000
- %100       -   %111        -    %1
 %1110        %111110          %111
```

Now work these problems. Remember to use highest value digits for the appropriate number system.

```
(a)  $C0001      (b)  $2A00B      (c)  %1000
    -    $5          -   $FF          - %11
    -------          -------          -----
```

— — — — — — — — — —

```
      BFFF             9FF              011
(a)  $C0001      (b)  $2A00B      (c)  %1000
    -    $5          -   $FF          - %11
     $BFFFC           $29F0C           %101
```

24. For practice, find the sums and differences below.

```
(a) %1001 1111              (b) %1110 0000 1111
   + %10 0011                       +     %1010

(c) %1001 1111              (d) %1110 0000 1111
   - %10 0011                       -     %1010

(e)   $426A                 (f)   $60D0
    +   $B9                     + $51E2

(g)   $426A                 (h)   $60D0
    -   $B9                     - $51E2
```

— — — — — — — — — —

(a) %1100 0010; (b) %1110 0001 1001; (c) %0111 1100; (d) %1110 0000 0101; (e) $4323; (f) $B2B2; (g) $41B1; (h) $0EEE

ASCII CODE

So far, you have learned how to handle hexadecimal and binary numbers. You'll use this information frequently as you develop programs. But when a number is entered into the system from the outside, it's not translated directly into binary; it's always treated as alphanumeric data and encoded by a code system such as ASCII or EBCDIC. In this book, we'll use ASCII and assume that the first bit (the left-most or high-order bit) is zero, since ASCII uses only seven of the eight available bits. Many systems use the high-order bit for special purposes. The Apple, for example, uses it as an I/O indicator. The Pet can use it to indicate an alternative character set. The Atari uses it to reverse the colors displayed on the terminal. You'll need to become familiar with your system's use of the high-order bit.

25. Appendix B shows ASCII code. Each character shown in the grid is represented by two hex digits. These are shown on the top and left of the grid. For example, $50 is the ASCII code for the letter P. Use Appendix B to answer the questions below.

Write the ASCII code (in hex) for the following characters.

(a) * __________

(b) 3 __________

(c) A __________

(d) d __________

Give the character indicated by each of the following ASCII codes.

(e) $35 __________

(f) $4D __________

(g) $77 __________

(h) $25 __________

— — — — — — — — — —

(a) $2A; (b) $33; (c) $41; (d) $64; (e) 5; (f) M; (g) w; (h) %

26. Notice particularly how the ten decimal digits are coded in ASCII. The code always starts with $3. The second half of the byte is equal to the decimal digit. Thus, 0 is coded as $30, 1 is coded as $31, etc.

(a) If you type and enter the character '5,' what will be received by the computer? (Choose one.)

_______ %0000 0101 (binary for decimal 5)

_______ $05 (hex for decimal 5)

_______ %0011 0101 ($35)

(b) Suppose you write a program to add together registers A and X.

Register A contains the ASCII code for 2. What is it? ________

Register X contains the ASCII code for 3. What is it? ________

What will the sum be? (Give the answer in ASCII code.) ________

What is the character form of that code? ________

— — — — — — — — — —

(a) %00110101 ($35); (b) $32, $33, $65, e (That's right, the computer comes up with the wrong answer. Later in this book you'll learn how to handle numeric values so your programs don't produce erroneous arithmetic results.)

27. The codes from $00 through $1F and code $7F are special codes that control the action of the output device. These are recommended codes used by most systems. A terminal or other device may interpret them differently. (The Ataris are different.) You use them by sending (or *writing*) the appropriate byte to the device. For example, if you write the code $07, the bell on the output device will ring. (Nothing will be printed.) If the terminal does not have a bell, it won't recognize $07 as a special code. All unrecognized codes are ignored or printed as blanks. Refer to the explanations in Appendix B to answer these questions.

(a) What code will cause one character to be deleted (DEL)? ________

(b) What code will cause the output device (printer or video screen) to start a new page (FF)? ________________________________

(c) If you want to start a new line on a printer or a video terminal, you need to write a carriage return (CR) followed by a line feed (LF). What are the ASCII codes for these two characters?

____

(d) Suppose your program sends $0B, for vertical tab, to a terminal that does not have a vertical tabbing capability. What will happen?

____

\- - - - - - - - - -

(a) $7F; (b) $0C; (c) $0D and $0A; (d) it will be ignored or a space will be printed.

Don't be frightened by all those special characters. Most of them are only for special equipment and special applications. Remember, too, that your equipment may use different codes. Check your manuals if you want to use these codes.

28. When you are using ASCII values in your Assembly Language programs, you don't need to use the hexadecimal codes. You can use the letters or digits directly by enclosing them in single quotes.

For example, we can store a T in the accumulator by this instruction:

```
LDA  #$54
```

Or we can code:

```
LDA  #'T'
```

This has a special advantage because we don't have to worry about whether our system uses standard ASCII code. So one program can be used on more than one machine.

Write an instruction to load the ASCII code for + in the accumulator.

____

\- - - - - - - - - -

LDA #'+'

29. The ASCII control codes from $00 through $1F and $7F cannot be called on directly in quotes. You have to use the hexadecimal code.

You should memorize the codes for carriage return and line feed. You'll use these a lot. The others can be looked up if you ever need them.

(a) What is the ASCII code for a carriage return (CR)? ________________

(b) What is the ASCII code for a line feed (LF)? ________________

\- - - - - - - - - -

(a) $0D or $8D; (b) $0A or $8A

REVIEW

In this chapter, you have studied data representation in the computer.

- The binary number system is based on two. Each column represents a power of two. The least significant digit represents 2^0, the next left column represents 2^1, and so forth. Zero is the lowest digit and one is the highest digit.
- Binary numbers are converted to decimal by multiplying each column by the appropriate power of two and summing the results.
- Decimal numbers are converted to binary by dividing by successive powers of two, starting with the largest power that will fit into the decimal number.
- The binary math facts are:

  0 + 0 = 0; 0 + 1 or 1 + 0 = 1; 1 + 1 = 10.
- The hexadecimal (or hex) number system is based on 16. Hex numbers are used merely as shorthand for binary numbers. The computer will report numbers to you in hex and you can give it hex numbers.

  Each column represents a power of 16. The least significant column is 16^0, the next column is 16^1, etc. The digits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. F is the highest digit.
- Hex numbers are converted to decimal numbers by multiplying each column by the appropriate power of 16 and summing the results.
- Decimal numbers are converted to hexadecimal by dividing by successive powers of 16, starting with the largest power that will fit into the decimal number.
- The hex math facts are shown in Appendix A.
- Binary and hex numbers are directly related. Four binary digits equal one hex digit. Numbers can be converted between the two systems on sight if you memorize the table of equivalents, from %0 = $0 through %1111 = $F. (See Figure 5.)
- Data entering the system through a terminal is always treated as alphanumeric data and is encoded in a code system such as ASCII. Appendix B shows ASCII code. Assume that the left-most (high-order or most significant) bit is always zero.

Now take the Self-Test for this chapter.

CHAPTER 2 SELF-TEST

You may use Figures 5 through 7 and Appendices A and B for this Self-Test.

1. Convert to decimal.

   a. $21 = ____________
   b. $16 = ____________
   c. $7 = ____________
   d. $10 = ____________
   e. $100 = ____________
   f. $255 = ____________

2. Convert to decimal.

   a. %101 = ____________
   b. %10100 = ____________
   c. %1100 = ____________
   d. %1001 = ____________
   e. %1111 = ____________
   f. %11011 = ____________

3. Convert to hex.

   a. %101 = ____________
   b. %11011011 = ____________
   c. %10001 = ____________
   d. %1001 = ____________
   e. %100000 = ____________
   f. %11011 = ____________

4. Convert to binary.

   a. $16 = ____________
   b. $7 = ____________
   c. $21 = ____________
   d. $2 = ____________
   e. $10 = ____________
   f. $A = ____________

5. Convert to hex.

   a. 21 = ____________
   b. 4 = ____________
   c. 10 = ____________
   d. 16 = ____________
   e. 100 = ____________
   f. 255 = ____________

6. Convert to binary.

   a. 8 = ____________
   b. 25 = ____________
   c. 38 = ____________
   d. 85 = ____________
   e. 52 = ____________
   f. 18 = ____________

7. Add.

```
a.   %101101      b.   %11111      c.   %1000110
   +    %101         +   %110         +   %11010
```

8. Add.

a. \$64 + \$AB

b. \$FF1 + \$A9

c. \$21B3 + \$C15

9. Subtract.

a. %100101 − %1110

b. %110000 − %11

c. %100000 − %11111

10. Subtract.

a. \$100A − \$2B

b. \$A0007 − \$BF

c. \$3D09 − \$1AA

11. Write the alphanumeric character for each of these ASCII codes.

a. \$23 = ______

b. \$37 = ______

c. \$4A = ______

d. \$62 = ______

e. \$7D = ______

f. \$3F = ______

12. Write the ASCII code for each of these characters or control functions.

a. 5 = ______

b. [= ______

c. carriage return = ______

d. line feed = ______

e. space = ______

f. H = ______

Self-Test Answer Key

1.
a. 33
b. 22
c. 7
d. 16
e. 256
f. 597

2.
a. 5
b. 20
c. 12
d. 9
e. 15
f. 27

3. a. $5 b. $DB c. $11 d. $E e. $20 f. $1B

4. a. %10110 b. %111 c. %100001 d. %10 e. %10000 f. %1010

5. a. $15 b. $4 c. $A d. $10 e. $64 f. $FF

6. a. %1000 b. %11001 c. %100110 d. %1010101 e. %110100 f. %10010

7. a. %110010 b. %100101 c. %1100000

8. a. $10F b. $109A c. $2DC8

9. a. %10111 b. %101101 c. %1

10. a. $FDF b. $9FF48 c. $3B5F

11. a. # b. 7 c. J d. b e. } f. ?

12. a. '5' or $35 b. '[' or $5B c. $0D d. $0A e. ' ' or $20 f. 'H' or $48

If you missed any of these, study the appropriate frames before going on to Chapter 3.

This has been a brief look at number systems and data representation. If you would like to learn more, try the Wiley Self-Teaching Guide, *Background Math for a Computer World*, by Ruth Ashley.

CHAPTER THREE

INSTRUCTION FORMAT

In the two previous chapters you studied some necessary background information. Now you're ready to begin attacking the subject of Assembly Language itself. In this chapter, we'll look at the format of an Assembly Language instruction. You'll learn how to code all the parts of an instruction, and you'll be exposed to many instructions that will be used later in this book.

When you have finished this chapter, you will be able to:

- name the parts of an Assembly Language instruction, and identify the required part;
- recode it in correct format, given Assembly Language code in incorrect format;
- create a label for an instruction or a data area;
- identify the types of instructions that need labels;
- write comments in a program;
- identify the operands in an instruction.

1. Figure 8 shows some sample code that we'll use throughout this chapter to demonstrate the format of Assembly Language instructions. It is not a complete program, just a few lines of code.

```
; THIS ROUTINE READS A NUMBER FROM
; THE TERMINAL, ADDS TO IT, AND PUTS
; THE RESULT BACK OUT TO THE TERMINAL
; THEN THE ROUTINE GOES INTO AN ENDLESS
; LOOP UNTIL STOPPED BY OPERATOR.
;
READIN   JSR   INPUT    ; READ BYTE
         LDA   NUMBER
ADDIT    ADC   #$02     ; ADD 2
         STA   NUMBER
         JSR   OUTPUT   ; DISPLAY BYTE
DONE     JMP   DONE
```

FIGURE 8. Sample Code

The first six lines are descriptive comments. They are printed whenever the program is printed but are otherwise ignored by the computer. They are intended for human beings who are reading the program.

The first instruction, JSR (*J*ump to *S*ub*R*outine), calls a subroutine named INPUT that reads a byte from the terminal and stores it in a memory location called NUMBER. The next instruction, LDA (*LoaD A*ccumulator), copies the byte from NUMBER to the accumulator.

The third instruction, ADC (*AD*d with *C*arry) adds 2 to the contents of the accumulator, assuming that the carry flag is clear. The fourth instruction, STA (*ST*ore *A*ccumulator) copies the changed byte from the accumulator back to memory location NUMBER.

The fifth instruction, JSR, arranges to write the byte in NUMBER to the original terminal. Notice that the original byte was *not* written back—only the changed one. The final instruction (*JuMP*) sets up an endless loop, which will repeat itself until the program is interrupted.

(a) What happens if the user presses the "3" on the keyboard?

____ "3" is displayed on the screen, and then "5"

____ "5" is displayed

(b) What happens after the answer is displayed?

____ the program waits for the user to type another number

____ the program ends; the user can start to run another program or re-run this one

____ the program continues running, with no action occurring

— — — — — — — — — —

(a) "5" is displayed on the screen (the number typed never shows up on the screen); (b) the program continues running, with no action occurring, until the user interrupts it in some way (such as turning off the system)

GENERAL INSTRUCTION FORMAT

The format of an Assembly Language instruction is dictated by the program that will translate it—the assembler. Different brands of assemblers have different requirements for instruction formats. but we can find some common points. We'll discuss the common points and show you what our assembler requires. Don't forget that your assembler may be somewhat different.

2. Refer to Figure 8 again and you'll see that an Assembly Language instruction can have up to four parts:

- The *label* gives the instruction a name that we can refer to from other instructions.
- The *operation* tells the computer what to do (such as jump or load accumulator).
- The *operand* identifies the data or other program address to be used in the operation.
- Any *instruction* may have a narrative comment at the end.

In the instruction below, identify the four parts.

READIN JSR INPUT ;READ BYTE

(a) ________ (b) ________ (c) ________ (d) ________

— — — — — — — — — — — —

(a) label; (b) operation; (c) operand; (d) comment

3. Refer to Figure 8 again. Two types of statements are shown—comments and instructions. Comment statements are free-form after the semicolon (;) in column one. Any information may be contained in column 2 and later. Instructions, on the other hand, may consist of up to four parts. Name the four parts.

(a) ______________ (b) ______________

(c) ______________ (d) ______________

— — — — — — — — — — — —

(a) label; (b) operation; (c) operand; (d) comment

THE LABEL FIELD

4. A label gives a name to an instruction. You then use that name as an operand in other instructions. There are two major ways we use labels.

- Jump or branch instructions: When we want to jump or branch to an instruction, we give it a label. Then we jump or branch to that label. In Figure 8, DONE is a label. The JMP instruction jumps to itself, creating the endless loop at the end of the program.
- Data names: We assign names to data storage areas in our programs. Then our instructions refer to the data areas by name rather than numeric address. The names serve as labels for the storage areas.

Which of the following would you label?

___ (a) Every instruction in the program.

___ (b) Every fifth instruction in the program.

___ (c) Instructions that have to be jumped to.

___ (d) Jump instructions.

___ (e) Data storage areas.

___ (f) The last instruction in the program.

___ (g) The first instruction in the program.

— — — — — — — — — —

(c) and (e) are the best answers. Many programmers also label (g) to give the program a name.

5. Your assembler will have a set of rules for proper labels. In this book, we'll use the rules shown below. They're fairly common.

- A label must be one to six characters long.
- It can contain letters (A–Z) or digits (0–9). No special characters such as $, –, /.
- It must start with a letter.
- It must start in column 1.
- The following cannot be used as labels:

  —register names (A, X, Y)

  —6502 operation codes (such as JMP and STA)

According to *our* rules, which of the following are legal labels?

___ (a) START

___ (b) STA

___ (c) 3RDONE

___ (d) THIRD

___ (e) B100

___ (f) TRY#5

___ (g) EXCEPTION

___ (h) A

___ (i) 1

___ (j) K

— — — — — — — — — —

(a), (d), (e), and (j) are correct

(b) is an operation code; (c) starts with a digit; (f) contains an illegal character; (g) is too long; (h) is a register name; and (i) starts with a digit

6. Most programmers prefer to use meaningful names as labels. For example, suppose you want to give a name to the first instruction of a routine that adds two numbers. ADDER is a better name than B1 or Q.

Meaningful labels will help other people read your programs with understanding and will also help you when you return to a program you haven't worked on for a couple of months.

Which of the following labels are better?

a) For a data storage area that will hold a social security number—SSN, SOCSEC, or N9? ____________

(b) For the first instruction of a routine that reads and stores the social security number—GETSOC, RANDS, or X2T1? ____________

Now try writing some labels of your own.

(c) For the first instruction of a routine that prints a page number on a new page.

____

(d) For the first instruction of a routine that counts the time in tenths of seconds until the user pushes any key. ____________

— — — — — — — — — — —

(a) SOCSEC is best, SSN is second best; (b) GETSOC is best; (c) we would use NUMPAG or PAGE; (d) we would use TIMER (You could use any meaningful name that meets the name-forming rules. It's easier if your names are pronounceable.)

7. According to *our* rules, which of the following labels are legal?

____(a) STOPPER	____ (h) RUN-IT
____(b) GATER	____ (i) X
____(c) ENDER	____ (j) *
____(d) END#1	____ (k) 3010
____(e) ENDTWO	____ (l) RUN10
____(f) JMP	____ (m) FORCE
____(g) JONES	____ (n) T/N

Write good labels for each of the following:

(o) The first instruction of a routine that handles input errors.

____

(p) A data storage area that holds the old balance. ____________

— — — — — — — — — —

(b), (c), (e), (g), (l), and (m) are correct

(a) is too long; (d) contains an illegal character; (f) is an operation code; (h) contains an illegal character; (i) is a register name; (j) contains an illegal character; (k) doesn't start with a letter; and (n) contains an illegal character;

(o) we would use INERR or ERRIN; (p) we would use OLDBAL

THE OPERATION FIELD

8. The operation is like the verb of a sentence; it tells the computer what to do. Some typical operation codes are:

LDA for *LoaD* data into the *A*ccumulator

ADC for *AD*d with *C*arry

CMP for *C*o*MP*are to accumulator

BEQ for *B*ranch if *EQ*ual

JMP for *J*u*MP*

STA for *ST*ore data from *A*ccumulator

You don't make up your own operation codes for the standard 6502 operations. There is a standard set of 6502 codes. (Your assembler may have added some special ones to the set.)

Every operation code contains three letters. The operation is not optional; every instruction must have an operation.

(a) In the sentence SET THE TABLE, which word is most like the operation?

___ SET

___ THE

___ TABLE

(b) Can you make up your own operation codes? ______________________

(c) Two of the following are not operation codes. Can you tell which two?

___ X

___ SBC

___ ADC

___ ERASE

(d) Which instructions in Figure 8 require an operation code?

____

— — — — — — — — — —

(a) SET; (b) no; (c) X is too short and ERASE is too long; (d) all instructions; the comments aren't instructions

9. Your assembler may have some rules for coding the operation. A fairly common rule is that the operation must be preceded by at least one space. If the instruction contains a label, the space or spaces separate the label from the operation. If there is no label, the space or spaces tell the assembler that there is no label.

Most programmers code their programs so that the operation codes and the operands are lined up in columns. (See Figure 8 again.) This makes programs much easier to read. It also allows us to add labels later on if we need to. We always start the operation code in column 8 (to leave room for a six-character label plus one space). We always start the operands in column 12 (to leave room for a three-character operation code followed by a space).

A section of Assembly Language code is shown below. Some of the instructions are coded correctly and some incorrectly. Recode the entire section legibly, according to our rules. Use the coding form provided.

```
START  LDA   #10
ADC MORE
STA ANSWER
STOP JMP STOP
```

— — — — — — — — — —

```
START  LDA #10
       ADC MORE
       STA ANSWER
STOP   JMP STOP
```

THE COMMENTS FIELD

10. You've learned how to code labels and operations. We're going to skip over operands for now and discuss comments first. Then we'll finish up the chapter with operands—a very large topic.

Comments are used to document the program. They're ignored by the assembler. They're used by human beings who are reading the program. You can usually code comments after the instruction. If they're too long to fit on the screen line, they'll show up on the next line, as long as the total line doesn't exceed 80 characters.

Why do we add comments to a program? To help others understand what the program is doing. The effect or intent of a program or segment is not always clear from reading the labels, operations, and operands.

We also write comments for ourselves. Sometimes we can forget the intention of a routine by the next day. Reading an Assembly Language program written a month ago can be like reading hieroglyphics.

(a) Are comments required or optional? ________________

(b) Which is better—to limit comments or to use them freely?

____

(c) Suppose you're writing a program for your own purpose and it will never be seen by anyone else. Should you add comments or not? ________________

— — — — — — — — — —

(a) optional; (b) use comments freely; (c) yes

11. Most assemblers allow you to code separate comment lines. These are lines that do not contain any label, operation, or operand—just narrative comments. For our assembler, we indicate a separate comment line by coding a semicolon in column 1. Then we can use the rest of the line (through column 71) to code our comments. (Leaving the line blank after the semicolon provides some spacing, which makes the program easier to read.)

Figure 8 is an example of a well-commented program. Use it to answer the questions below.

(a) How many separate comment lines are there? ________________

(b) How many instructions contain comments? ________________

(c) Why did we include a line that contains only a semicolon in column 1?

____

— — — — — — — — — —

(a) 6; (b) 3; (c) for spacing

12. Below is the sample code you formatted before. Add the following comments to the coding form:

(a) On separate lines at the beginning, add: THIS ROUTINE ADDS 10 TO A MEMORY BYTE AND STORES THE RESULT. Break the line anywhere that's convenient.

(b) On the second instruction, add the comment: MORE CONTAINS 01

```
START   LDA #10
        ADC MORE
        STA ANSWER
STOP    JMP STOP
```

— — — — — — — — — —

```
; THIS ROUTINE ADDS 10 TO A MEMORY BYTE
; AND STORES THE RESULT
START   LDA #10
        ADC MORE        ; MORE CONTAINS 01
        STA ANSWER
STOP    JMP STOP
```

THE OPERAND FIELD

13. You have learned how to code the label, operation, and comments; now we'll talk about the operand.

An operand answers the question "WHERE?" In the instruction JMP DONE, the operation code JMP means "jump to . . ."; the operand, DONE, specifies WHERE to jump. In the instruction LDA NUMBER, the operation code LDA means "copy the byte from . . . into the accumulator"; the operand, NUMBER, specifies WHERE to copy from.

Sometimes an operand may have two parts, separated by a comma; for example, the operand in LDA $30,X has two parts.

(a) What is the operand in the instruction ADC C? ______________
If the operation code means "add to the accumulator . . .," what does the operand specify? ______________

(b) What is the operand in the instruction STA $90,X? ______________
If the operation code means "copy the value from the accumulator . . .," what does the operand specify? ______________

(c) What is the operand in the instruction CMP SUMOFD? ______________
If the operation code means "compare the value in the accumulator to . . .," what does the operand specify? ______________

— — — — — — — — — —

(a) C, where to find the value to be added; (b) $90,X where to copy the accumulator value to; (c) SUMOFD, where to find the value to compare it to

14. Some instructions do not require an operand. The instruction itself includes all the necessary information. For example, the instruction TAX means "transfer the byte in register A to register X"; no operand is necessary to describe WHERE the byte comes from or WHERE it goes.

In each of the following instructions, indicate whether there's an operand and if so, what it is.

(a) CMP $05 Is there an operand? ____ If so, what is it? ____________

(b) DEX Is there an operand? ____ If so, what is it? ____________

(c) DEC $20,X Is there an operand? ____ If so, what is it? ____________

(d) JMP STOP Is there an operand? ____ If so, what is it? ____________

(e) BRK Is there an operand? ____ If so, what is it? ____________

— — — — — — — — — —

(a) yes, $05; (b) no; (c) yes, $20,X; (d) yes, STOP; (e) no

15. Your assembler will have coding rules for operands. Here are some fairly common rules:

- Operands must be separated from the operation code by at least one space.
- No spaces are permitted within the operands, except inside quotation marks.

(a) If your assembler uses the format rules as presented in this chapter and you want all your instructions to line up in columns (as in Figure 8), in what column will you start the label? ____, the operation? ____, the operand? ____

(b) Code ANY as an operand in this instruction: STA ____________

(c) Code #120 as an operand in this instruction: ADC ____________

— — — — — — — — — —

(a) 1, 8, 12; (b) STA ANY; (c) ADC #120

REVIEW

Let's review what you've learned in this chapter.

- The general format of an Assembly Language instruction for the majority of assemblers is:

  [*label*] *operation* [*operand*] [;*comment*]

  Brackets indicate optional items.

- A label gives a name to the location of the instruction. The label is then used as an operand in other instructions. Every assembler will have rules for the formation of labels. Here are some common rules:

  - one to six characters
  - numbers or letters, no special symbols
  - start with a letter
  - start in column 1
  - don't use register names or operation codes

  Most programmers prefer to use meaningful names as labels.

- The operation code is a three-character code that is standard for 6502 Assembly Language. It tells the computer what to do. Most assemblers require that it be preceded and followed by at least one space.

- Comments are used to document the intention of routines and individual instructions that might not otherwise be clear. We document for ourselves as well as others. Comments are ignored by the assembler and do not appear in the machine language version of the program. In many assemblers, they are separated from the operands by at least one space and are preceded with a semicolon or other special character.

- Most assemblers allow separate comment lines by coding a semicolon or other special symbol in column 1.

- An operand specifies WHERE to find data or WHERE to transfer control. Some instructions have one operand; some have none. No spaces may appear in the operand section except within quotation marks. Some operands are written in two parts, with a comma separating the parts.

CHAPTER 3 SELF-TEST

1. What are the four possible parts of an instruction? ______________________

   ____

2. Which of the four parts of the instruction are required in every instruction?

   ____

3. Here is a series of instructions and comments. Recode them on the coding form so the parts are in the proper columns.

```
;BEGIN HERE
LDA START,X
REPEAT CMP MAX
BEQ NEXT
TAX
ADC #02
JMP REPEAT ; REPEAT UNTIL MAX REACHED
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45

4. Make up valid labels for the following items.

   a. The first instruction of a routine that reads and stores an addend.

   ____

   b. The memory area where the addend is stored.

   ____

   c. The first instruction of a routine that multiplies a value by 5.

   ____

   d. The memory area where the above result is stored.

   ____

5. Which of the following items should have labels?

____ a. Instructions that are jumped to.

____ b. Jump instructions (JMP and JSR).

____ c. Instructions without comments.

____ d. Data areas used as operands.

6. The following routine reads a byte from the terminal (JSR INPUT), displays the same byte on the terminal (JSR OUTPUT), compares the byte to 9 (CMP #9) and goes into an endless loop if the byte equals 9 (LOOP BEQ LOOP). Add appropriate comments to the program, including at least one comment line before the routine.

```
CHECK9  JSR INPUT
        JSR OUTPUT
        CMP #'9'
LOOP    BEQ LOOP
```

7. In each of the following instructions, identify the operand. If the instruction has no operand, write NONE.

   a. `STA $02` ____________________

   b. `CMP $FF` ____________________

   c. `CMP $25,X` ____________________

   d. `INX` ____________________

   e. `ADC ADDEND` ____________________

8. If ADC means "add . . . to the accumulator," what does ADC ADDEND mean?

____

Answer Key

1. label, operation, operand, comments

2. only the operation is required

3.

```
; BEGIN HERE
        LDA START,X
REPEAT  CMP MAX
        BEQ NEXT
        TAX
        ADC #Ø2
        JMP REPEAT    ;REPEAT UNTIL MAX REACHED
```

4. Some sample answers are shown below. Yours may be different but should follow the general rules for labels.

   a. GETADD , STORAD

   b. ADDEND , ADDER1 (presuming there are more than one)

   c. MULTER , MULTI5 , TIMES5

   d. PRODUC, RESULT, ANSWER, FIVEX

5. a and d

6. Our sample comments are shown below. Yours may be different but should follow the general rules for comments.

```
; THE FOLLOWING ROUTINE READS A BYTE AND
; STALLS IF IT'S 9.
;
CHECK9  JSR INPUT             ;READ THE BYTE
        JSR OUTPUT            ;DISPLAY IT
        CMP #'9'              ;IS IT 9?
LOOP    BEQ LOOP              ;STALL IF IT'S 9
```

7. a. $02

   b. $FF

   c. $25,X

   d. none

   e. ADDEND

8. Add the data stored at ADDEND to the accumulator.

If you missed any, restudy the appropriate frames before going on to the next chapter.

CHAPTER FOUR

OPERAND FORMATS

In 6502 Assembly Language, there are several different formats for operands that you'll learn how to code in this chapter. You'll see a variety of terms for the formats. There is no agreement on the number of them. We have chosen a set of terms that we feel are descriptive and clear.

Don't be disturbed if your reference manuals use different terms for these operand formats, or even if they group them differently. They're the same formats. You will learn *all* the formats here.

When you have completed studying this chapter, you will be able to:

- code the following addressing modes:
  - immediate
  - direct
  - zero-page direct
  - indexed direct
  - zero-page indexed direct
  - indirect
  - pre-indexed indirect
  - post-indexed indirect
  - relative;
- Use labels and expressions as operands.

IMMEDIATE ADDRESSING

1. Some instructions manipulate one byte of data—copying it, adding or subtracting it from the accumulator, etc. If you know when you code the instruction exactly what value this byte will have at execution time, you can write the value as the operand. This is called *immediate* addressing.

When you use immediate addressing, the question "WHERE?" is answered "RIGHT HERE!" On many 6502 assemblers. and in this book. a byte of immediate data is preceded by "#"; the value of the byte may be coded in any of the notations described in Chapter 2. For instance, #10 would be the immediate address for value 10 written in decimal notation; the same value in hexadecimal notation would be #$0A.

(a) How many bytes are in an immediate address? ______________

Code the following operands using immediate addresses:

(b) the value 25, using decimal notation ______________

(c) the value 14, using hexadecimal notation ______________

(d) the letter A, using ASCII character notation ______________

(e) the value 4, using binary notation ______________

— — — — — — — — — —

(a) one; (b) #25: (c) $0E; (d) #'A'; (e) #%00000100 (We'll continue to use two digits for hex and eight for binary. Hex addresses will have four digits.)

2. Figure 9 shows a set of operation codes that we'll use throughout this chapter.

CODE	STANDS FOR	EFFECT OF OPERATION
ADC	*AD*d with *C*arry	Adds value from operand to value in accumulator
BEQ	*B*ranch if *EQ*ual	Changes "next instruction" address in program counter if zero-flag is "on"
CMP	*C*o*MP*are	Compares value in accumulator to value in operand
JMP	*J*u*MP*	Change "next instruction" address in program counter
LDA	*L*oa*D* *A*ccumulator	Copies value from operand to accumulator
STA	*ST*ore *A*ccumulator	Copies value in accumulator to operand

FIGURE 9. Some Sample Operation Codes That Require Operands

Use the operation codes from the figure and immediate addressing to code a series of instructions that will set the accumulator to zero, add 25 to the accumulator, and compare the result to 20. Use decimal values.

— — — — — — — — — —

```
LDA #0
ADC #25
CMP #20
```

3. You will find yourself using immediate addressing quite often in a program. When you want to load a specific value in a register, immediate addressing is the most natural way.

For example, suppose we want to use index register X in a loop. We need to initialize it to zero. The easiest way is:

```
LDX   #0
```

Suppose we want to write out an asterisk on the terminal. It has to be loaded into register A first. The easiest way is:

```
LDA   #'*'
```

Another frequent use of immediate addressing is in making comparisons. For example, suppose we have just read a byte from the terminal keyboard and we want to see if it is a carriage return. We could code:

```
CMP   #$0D
```

Suppose we want to compare the value in A to decimal 25. We could code:

```
CMP   #25
```

Which of the following are frequent uses of immediate addressing?

____ (a) comparisons

____ (b) branching

____ (c) loading registers

____ (d) storing registers

— — — — — — — — — —

(a) and (c)

DIRECT ADDRESSING

4. Operands that are not immediate answer the question "WHERE?" by specifying a memory address. With some operations, this would be the address of a byte of data; for others, it would be the beginning address for the next instruction.

In the *direct* addressing mode, you just write the numeric value of the address as the operand. Addresses are usually coded in hexadecimal notation, showing all four digits.

Use operation codes from Figure 9 and direct addressing to code instructions that will:

(a) copy data from location $02FF to the accumulator ____________________

(b) execute next the instruction beginning at address $72FF ________________

— — — — — — — — — —

(a) LDA $02FF; (b) JMP $72FF

ZERO-PAGE DIRECT ADDRESSING

5. In 6502 Assembly Language, it is convenient to look at the 64K bytes of memory as though they were divided into 256 pages of 256 bytes each. The first 256 bytes, with addresses from $0000 to $00FF, are called the zero page. The next page, with addresses from $0100 to $01FF, is page one.

(a) What would you call the page with addresses from $FA00 to $FAFF?

____

(b) What addresses are on page 10? ______________________________

— — — — — — — — — —

(a) page FA; (b) $1000 to $10FF

6. When you code a direct address that is on the zero page, you do not need to code the two leading zeros. For instance, $30 represents the same address as $0030. (The Pet assembler requires an asterisk to indicate a zero-page address.) The computer will assume that an address with only one byte (two hexadecimal digits) is on the zero page. A zero-page address saves one byte of space in the length of the instruction; it can also increase the speed of executing the instruction. Data that will be referenced often should be put on the zero page.

(a) What is the unabbreviated form of this address — \$F5? ____________

(b) What is a shorter way to code the address \$0029? ____________

(c) What is the advantage of using zero-page addressing?

____

— — — — — — — — — —

(a) \$00F5; (b) \$29 (for Pet, this would be *\$29); (c) it saves space and time

7. You will probably code more direct addresses than any other type in a program. We use direct addressing whenever we want to access a known memory location. Don't forget that a label can be a direct address.

Suppose we want to transfer control to the instruction labeled OTLOOP. The most natural instruction is:

```
JMP   OTLOOP
```

Suppose we want to load register A from the byte labeled CARRET. We would code:

```
LDA   CARRET
```

Suppose we want to store A at address \$05 in memory. We would code:

```
STA   $05
```

When do you use direct addressing? (Choose one.)

____ (a) When you're not sure what memory address to use.

____ (b) Whenever you can't use immediate addressing.

____ (c) When you want to refer to a specific memory address.

— — — — — — — — — —

(c)

8. Let's review the three types of operands you have studied so far. Using operation codes from Figure 9, code instructions to:

(a) set the accumulator to 10 ______________________

(b) add the value from the seventh byte on page 3 to the accumulator

____

(c) compare the value in the accumulator to the value in the third byte of memory

____

(d) How do you distinguish between an immediate address and a direct address?

____

— — — — — — — — — —

(a) LDA #10; (b) ADC $0306; (c) CMP $02; (d) an immediate address is marked by a # in most assemblers; a direct address is not.

INDEXED DIRECT ADDRESSING

9. When an operand is written as an *indexed direct* address, the value in one of the index (X or Y) registers is added to a direct address; the result is the actual address. This is one of those two-part operands we mentioned earlier, where the two parts are separated by a comma. The first part is a direct address, and the second part is the register name (X or Y).

For instance, the instruction LDA $0218,X will load the accumulator with the byte from $021A if the value in X is $02, because $0218 + $02 = $021A. If the value in X is $03, the same instruction will load the accumulator with the byte from $021B.

(a) If the value in register X is $01, and the value in register Y is $0A, where will the comparison bytes be found in these instructions?

```
CMP $0527,X  ________
CMP $08FF,Y  ________
```

(b) The following instructions use direct addressing. Recode them using indexed direct addresses, assuming that register X = $02 and register Y = $15.

```
LDA $0713  ____________
ADC $08F8  ____________
```

— — — — — — — — — —

(a) $0528; $0909; (b) LDA $0711,X or LDA $06FE,Y; ADC $08F6,X or ADC $08E3,Y

ZERO-PAGE INDEXED DIRECT ADDRESSING

10. When the direct address portion of an indexed direct address is on the zero page—that is, between $0000 and $00FF—the two leading zeroes can be omitted. For example, the indexed direct address $0036,X is equivalent to $36,X. What is another way to code each of the following?

(a) `$00AA,X` __________

(b) `$28,X` __________

— — — — — — — — — — —

(a) $AA,X; (b) $0028,X

11. Using *zero-page indexed direct* instead of indexed direct addressing will cut one byte from the length of an instruction and increase execution speed. You need to be careful, though. If the result of the addition is larger than $FF, the page-number portion (first two hexadecimal digits) will be dropped; the result will still be a location on the zero page. For example, if X register = $02, the address resulting from $FF,X will be $01, which is the second byte on the zero page.

Are the following operands valid? If so, what is the actual address represented?

(a) $32,X (X register = $02) __________

(b) $0050,Y (Y register = $04) __________

(c) $01FF,Y (Y register = $05) __________

(d) $FF,Y (Y register = $10) __________

— — — — — — — — — — —

(a) $34; (b) $54; (c) $0204; (d) $0F (They are all valid.)

12. We usually use indexed addressing in loops where we are accessing *successive* bytes of memory. For example, suppose we want to write the message "THANK YOU" on the terminal. The message is stored in memory starting at address THANKS. Here's the routine we would use:

```
       LDX  #0          ; INITIALIZE X
OTLOOP LDA  THANKS,X    ; MOVE NEXT LETTER INTO A
       JSR  OUTPUT      ; WRITE THE LETTER
       INX              ; ADD 1 TO X
       CMP  #'U'        ; IS IT THE LAST LETTER?
       BNE  OTLOOP      ; IF NOT, GO BACK AND GET THE NEXT LETTER
```

The instruction labeled OTLOOP contains an indexed address, using the X register as the index. Before the loop starts, we initialize X to zero. The first loop loads A from THANKS+0. This puts a 'T' in register A, which is sent to the terminal by the JSR instruction. Then we increment X. On the second loop, H will be loaded from THANKS+1, and so forth until the entire message has been written.

Match the types of addresses you have studied so far with their major uses.

____	(a) immediate	1. accessing successive addresses in a loop
____	(b) direct and zero page direct	2. accessing a specific memory address
____	(c) indexed direct and indexed direct page	3. avoiding memory access and putting the data right in the instruction

---

(a) 3; (b) 2; (c) 1

13. In the previous frames, we have discussed five ways of coding operands. In *immediate* addressing, the data to be used by the operation is included in the instruction. In the four forms of direct addressing, the operand represents the address where a byte of data is to be found or the address of the next instruction to be executed. In the simplest form, a *direct* address is written as part of the instruction. A *zero-page direct* address is a direct address on the zero page; the two leading zeroes on the address are not written. An *indexed direct* address specifies a direct address and the name of an index register that contains a value to be added to the direct address. A *zero-page indexed direct* address does the same thing, using the shorter zero-page address.

To practice these addressing modes, fill in the blanks in the chart below. Assume that the X register = $12, the Y register = $20, and the accumulator = 15 (decimal) before the instruction is executed. Use operation codes from Figure 9. The first two examples have been done for you.

Address Mode	Instruction	Result
Direct	LDA $1732	A = byte at $1732
Zero-Page Indexed Direct	CMP $25,X	A compared to byte at $0037
Direct	ADC $1111	(a) A + ______
Immediate	(b) ADC ______	10 is added to A
Direct	(c) JMP ______	Next instruction executed is at $3212
Zero-Page Direct	(d) ______	A = byte at $0015
Indexed Direct	LDA $0222,Y	(e) A = byte at ______

— — — — — — — — — —

(a) byte at $1111; (b) #10; (c) $3212; (d) LDA $15; (e) $0242

These are the address types you will use most often. In the following section, we'll show some more operand styles. You won't use the following types of operands nearly as often until you begin to write advanced programs. But you'll see them in your system manuals, so you need to know what they look like and what they do.

INDIRECT ADDRESSING

14. An *indirect* address points to a memory location where the actual address to be used in the instruction has been stored. In other words, this second address is the one which would have been the operand if the instruction had been coded using a direct address. This form of addressing is useful when we need to change an operand at execution time. Instructions previously executed will store the address, which can vary depending on input data or other factors. An indirect operand is always enclosed in parentheses. ($1010) and ($0010) are indirect addresses; $1010 is a direct address. This simplest form of indirect addressing is used only with the JMP operation code.

Code JMP instructions to do the following:

(a) Jump to an address which can be found at location $2020.

____

(b) Jump to address $1510. ______________________________

(c) Jump to an address which can be found at byte $15 of page one.

____

— — — — — — — — — —

(a) JMP ($2020); (b) JMP $1510; (c) JMP ($0115)

15. An address is two bytes long, but each memory location can only hold one byte; so two consecutive memory locations are needed to hold an address. In 6502 Assembly Language, addresses are stored with the low-order byte (last two hexadecimal digits) in the *first* location, and the high-order byte (first two hexadecimal digits, or page number) in the following location. The indirect address points to the first location. The microprocessor will automatically go to the next byte to get the rest of the address.

If a partial map of memory shows:

Location	*Value*
...	
$1530	$03
$1531	$13
$1532	$FA
$1533	$02
...	

(a) What is the address of the instruction which is executed after JMP ($1530)?

____

(b) What about JMP ($1532)? ______________________________

(c) Code an instruction to jump to address $FA13 using a direct address.

____

(d) Recode the instruction using an indirect address. ______________________________

— — — — — — — — — —

(a) $1303; (b) $02FA; (c) JMP $FA13; (d) JMP ($1531)

PRE-INDEXED INDIRECT ADDRESSING

16. A *pre-indexed indirect* address is a zero-page indexed direct address, enclosed in parentheses, indicating the memory location where the address of the actual operand can be found. The actual address can refer to any memory location, but the location it is stored in must be on the zero page. Only the X register can be used in this operand.

Some valid pre-indexed indirect addresses are: ($26,X), ($30,X), and ($FF,X). ($16,Y) is invalid because it uses the Y register. Pre-indexed indirect addressing must use the X register. ($0116,X) is invalid because it references an indirect address that is not on page zero. Whenever indexing is used with indirect addresses, the indirect address must be on page zero.

This addressing mode will "wrap around" if necessary, so the resulting address is always on the zero page. If the X register contains $15, the address pointed to by ($FF,X) would be $0014.

What address is pointed to by each of these operands? (Assume the X register contains $10.)

(a) ($03,X) ____________

(b) ($75,Y) ____________

(c) ($0103,X) ____________

(d) ($17,X) ____________

(e) ($F1,X) ____________

Assume the X register = $10; code pre-indexed indirect addresses that will point to addresses stored at the following locations:

(f) byte $25 of the zero page ____________________

(g) byte $07 of the zero page ____________________

(h) byte $15 of page one ____________________

(i) byte $F2 of memory ____________________

(j) byte $0101 of memory ____________________

— — — — — — — — — —

(a) $0013; (b) none, the Y register cannot be used in this addressing mode; (c) none, only zero-page addresses can be used in this addressing mode; (d) $0027; (e) $0001; (f) ($15,X); (g) ($F7,X); (h) can't do this—pre-indexed indirect addresses always point to locations on page zero; (i) ($E2,X); (j) can't do this —byte $0101 of memory is byte $01 on page one

POST-INDEXED INDIRECT ADDRESS

17. A *post-indexed indirect* address uses an indirect address to point to a zero-page location, then adds the contents of the Y register to the address stored at that location. The result is the actual address used in the instruction. Only the Y register can be used in this addressing mode. The indirect address portion of the operand is enclosed in parentheses; the Y is not. Some valid post-indexed indirect addresses are ($13),Y and ($02),Y.

Here is a partial map of memory:

Location	*Value*
$0011	$01
$0012	$BC
$0013	$88
$0014	$E3
$0015	$00
...	...
$00FF	$E0
$0100	$FE
$0101	$02
$0102	$0A
$0103	$0F
...	...
$FF00	$08

Use this map and operation codes from Figure 9 to fill in the blanks in the examples below. Assume that the X register = $12, the Y register = $20, and the accumulator = 15 before each instruction is executed. The first two examples have been filled in for you.

Addressing Mode	Instruction	Result
Pre-indexed Indirect	STA ($02,X)	$02+$12=$14; address at $14-15 is $00E3; Byte $00E3 receives accumulator (15)
Indirect	JMP ($0102)	Next instruction at $0F0A
Pre-indexed Direct	(a) STA ($00,____)	(b) Byte ____ = 15
Indirect	(c) JMP ______	Next instruction at $00E3
Post-indexed Indirect	ADC ($14),Y	(d) ________
Indirect	(e) JMP ______	Next instruction at $02FE

— — — — — — — — — —

(a) X; (b) $88BC; (c) ($0014); (d) A = 30; (e) ($0100)

The following addressing mode is used only with branching instructions. Since it is the only form of operand allowed for branching instructions, you'll need to know it very well.

RELATIVE ADDRESSING

18. A *relative address* is a special form of address that is used only with the branch instructions. Before we can show you how it's coded in Assembly Language, we must discuss its machine language form; that is, after it has been translated by the assembler. In machine language, it is a one-byte signed number indicating the number of bytes to branch forward or backward. A relative operand of $20 would mean to branch forward 32 bytes, since $20 = +32. Because the machine language operand is limited to one byte, and the value is treated as a signed number, the maximum branches are +127 bytes (that's the largest positive signed number) and −128 bytes (that's the largest negative signed number).

(a) What kind of addressing must be used with branch instructions?

____

(b) What does a relative address of −16 tell the computer to do?

____

(c) What is the largest possible forward branch? ______________

(d) What is the largest possible backward branch? ______________

— — — — — — — — — —

(a) relative addressing; (b) branch backward 16 bytes; (c) 127 bytes; (d) 128 bytes

19. In Assembly Language, you don't code relative addresses themselves. You indicate what you want using the other forms of 6502 addresses. For example, relative addresses may be coded as direct addresses. The assembler calculates the relative address by subtracting the "current" address from the operand address. The "current" address is the address that would be in the PC when the branch instruction is processed at execution time.

The direct address form of relative addressing is not the only way to code a relative address in Assembly Language, but it is by far the easiest way. Other forms of coding relative addresses involve counting the bytes in the machine language instructions to be branched past; this can land you in a lot of hot water. Experienced programmers always use direct addresses, preferably labels, as operands of branch instructions.

(a) What is the best way to express a relative address in Assembly Language?

____

(b) If a direct address is a two-byte operand, how can it be used to express a one-byte signed operand? ______________

____

(c) In the following routine, the branch instruction, BNE, wants to transfer control to the LDA instruction, which is at address $0725. Code the operand for the BNE instruction.

```
LDX   #5
LDA   #5
JSR   OUTPUT
DEX
BNE   __________
```

(d) (Review) What is the maximum branch forward? __________, backward? __________ If all instructions contain 1, 2, or 3 bytes, will the BNE instruction above exceed the maximum backward branch? __________

— — — — — — — — — —

(a) as a direct address, preferably a label; (b) the assembler calculates the one-byte operand from the information you give it; (c) BNE $0725; (d) 127, 128, no—at *most*, you've jumped back 12 bytes (actually, it's eight)

As we described the various ways of coding addresses as operands, we assumed that you would know the address of each instruction and each data byte used in your programs. Some programmers do calculate the addresses they want to use by counting bytes; but fortunately, there is a simpler way. In the next few frames, you will learn to use *labels* and *expressions* as address operands.

LABELS AS OPERANDS

20. Look at the program in Figure 8 again. Notice that the instruction JSR INPUT has a label, READIN. Anytime the 6502 assembler sees the name READIN in the program, it will replace it with the address of the JSR INPUT instruction. If the address of JSR INPUT is $2513, the instruction JMP READIN is the same as the instruction JMP $2513.

If the instruction ADC $02 has the address $FA12, and the label ADIT, show two ways that you could code an instruction to jump to it.

— — — — — — — — — —

JMP $FA12; JMP ADIT

21. Labels as operands have the obvious advantages that you don't need to know the exact addresses and that, if you change your program, you don't need to change all the operands. This holds true whether the operand addresses an instruction or a byte of data.

If the value in address $0110 is 15, and you have assigned the label FTEEN to address $0110, the command LDA FTEEN will place the value 15 in the accumulator.

If the location $0FF0 contains the value $10, and the label FIRST has been attached to this location, what will the command ADC FIRST do?

— — — — — — — — — —

Add $10 to the accumulator

22. So far, all our examples using labels have used direct addressing. However, since the assembler replaces every label with an address, labels can be used to replace addresses in any addressing mode.

If the label TEMP has been assigned to location $00AA, and REPLAC to location $0515, recode the following instructions using labels:

(a) `JMP ($0515)` ______________________

(b) `LDA $AA,X` ______________________

(c) `ADC ($AA,X)` ______________________

(d) `CMP ($AA),Y` ______________________

(e) `STA $0515,Y` ______________________

(f) `LDA $AA` ______________________

— — — — — — — — — —

(a) JMP (REPLAC); (b) LDA TEMP,X; (c) ADC (TEMP,X); (d) CMP (TEMP),Y; (e) STA REPLAC,Y; (f) LDA TEMP

EXPRESSIONS AS OPERANDS

23. Many assemblers allow you to use expressions as operands. An expression is like the right side of an equation, as in X = Y+5. In instructions that use addresses for operands, the expression must work out to be a legitimate address. For instance, 3+5 would be okay, because it works out to 8, or $0008. But 5 − 9 would not be valid because it yields a negative number. $FFFF+$05 would not be valid because it yields a value outside of the memory range. A more common form of expression is START+5, where START is a label in the program. The result would be an address five bytes beyond the first byte of the instruction or data labeled START.

If your assembler allows expressions, it will have its own rules for how they are coded. Usually + is used for addition and − for subtraction. Other arithmetic operations may or may not be allowed. You need to be very careful when you use an expression as an operand. Remember that if lines are added or removed from the program the value at the computed location may change. However. expressions are useful when you want to refer to a data area without developing a name for every byte of data.

If the label START represents address $033F:

(a) What address does START+1 represent? ______________________

(b) What address does START−1 represent? ______________________

(c) Assuming that the X register = 3, recode the instruction STA BEGIN.X using an expression equivalent to the indexed direct address. ______________

— — — — — — — — — —

(a) $0340; (b) $033E; (c) STA BEGIN+3

REVIEW

Let's review what you've learned in this chapter.

- There are several operand formats, called addressing modes, which can be used when coding operations. We call them: immediate, direct, zero-page direct, indexed direct, zero-page indexed direct, indirect, pre-indexed indirect, post-indexed indirect, and relative.
- In the immediate addressing mode, a byte of data is coded by writing its value, preceded by #, in the operand field. Examples: #36, #$15, #'A'
- In the direct addressing mode. the numeric value of an address is written as an operand. Examples: $0136, $0278, $FF15.
- The first two hexadecimal digits of an address identify the page it is on. In zero-page direct addressing, an address on the zero page is coded *omitting* the page digits. Examples: $FF, $00. This is faster and saves space.
- In the indexed direct addressing mode, a two-part operand specifies a direct address and an index register (X or Y) that contains a value to be added to the direct address. The result is the actual address to use for the instruction. Examples: $1515,X; $FF34,Y.
- The zero-page indexed direct addressing mode is similar to the indexed direct mode. but both the direct address and the actual address are always on the zero page. Examples: $15,X; $FA,X.
- In the indirect addressing mode, the operand field contains an address enclosed in parentheses. This address points to a memory location where the low-order byte of the actual address is stored. (The computer will find the high-order, or page-number, byte in the next location.) This mode is used only with JMP (jump) instructions. Examples: ($1001); ($FAFA).
- In the pre-indexed indirect addressing mode, the operand field contains a zero-page indexed direct operand using the X register, enclosed in parentheses. Adding the value in the X register to the specified address results in another zero-page address that points to the memory location where the low-order byte of the actual address is stored. (The high-order, or page-number, byte is in the next location.) Examples: ($15,X); ($FA,X).
- In the post-indexed indirect addressing mode, a zero-page address enclosed in parentheses is followed by ".Y". The zero-page address points to the memory location where the low-order byte of another address is stored. (The high-order byte is in the next location.) The value in the Y register is added to this second address, resulting in the actual address. Examples: ($15),Y; ($FA),Y.
- In the relative addressing mode, the address is a one-byte signed value used to branch forward or backward. It is used only with branch instructions. The maximum branches are −128 to +127. We usually use direct addresses as branch operands and let the assembler calculate the relative address.

- Labels should replace addresses in operands. If the operand requires a zero-page address, the label must be attached to a zero-page address. A label in a conditional branch instruction replaces the entire operand; the assembler will calculate the difference between the address attached to the label and the one in the PC register. Therefore, this difference must be within the range −128 to +127. Examples: START; START,X; (START); (START,X); START,Y; (START),Y.
- Simple expressions such as a label plus or minus a number may be used as addresses on most operands. The assembler will interpret such an expression as an address that is that number of bytes after or before the address the label is attached to. The computed address must be a valid one for the addressing mode. Examples: START+1; (BEGIN−3,X).

CHAPTER 4 SELF-TEST

1. Identify the addressing mode of each of the following operands. If it is invalid, indicate why. (Assume that none are relative addresses.)

a. `$0719,X` ________________

b. `($17,X)` ________________

c. `$0315` ________________

d. `$E0,X` ________________

e. `#10` ________________

f. `($1930)` ________________

g. `($FA),Y` ________________

h. `#$FF13` ________________

i. `($86,X)` ________________

j. `$FA,X` ________________

k. `$FFFF` ________________

l. `$15` ________________

Examine the operands in each instruction below. Identify from the list on the right what each operand represents.

2. `CMP $5F` ______
3. `BEQ START` ______
4. `STA ($F8,X)` ______
5. `ADC $16FD` ______
6. `ADC #$14` ______
7. `JMP ($02)` ______
8. `CMP $FE,X` ______
9. `LDA #'B'` ______
10. `JMP $16F4` ______

a. immediate value

b. address of data byte or instruction

c. address that contains another address

d. value for determining a relative address

Write operands for the instructions below.

11. Add the value at address $0072 to the accumulator.
    ADC ______
12. Load the letter Z into the accumulator. LDA ______
13. Compare register A to decimal 16. CMP ______
14. If the zero flag is set, branch to location $06F0, which is labeled NEXT.
    BEQ ______
15. Jump to location $1740, which is labeled FOURTH.
    JMP ______
16. Store the value from the accumulator at memory locations $1400, which is labeled KEEP. STA ______
17. Store the value in the accumulator at the memory location that is at location $1400 (labeled KEEP) *plus* the value in register X. STA ______
18. Jump to the address currently stored in byte $1020 and 1021.
    JMP ______
19. Suppose byte $1020=$20 and byte $1021=$40. What is the actual address of the next instruction? ______
    Check your answers below.

Self-Test Answer Key

1. a. indexed direct

   b. pre-indexed indirect

   c. direct

   d. zero-page indexed direct

   e. immediate

   f. indirect

   g. post-indexed indirect

   h. this is invalid — an immediate address is one byte; $00 through $FF, or 0 through 256.

   i. pre-indexed indirect

   j. zero-page indexed direct

   k. direct

   l. zero-page direct

2. b

3. d

4. c

5. b

6. a

7. c

8. b

9. a

10. b

11. `ADC $72`

12. `LDA #'Z'`

13. `CMP #16`

14. `BEQ NEXT`

15. `JMP FOURTH (or JMP $1740)`

16. `STA KEEP (or STA $1400)`

17. `STA KEEP,X (or STA $1400,X)`

18. `LDA ($1020)`

19. `$4020`

If you missed any, restudy the appropriate frames.

CHAPTER FIVE

ELEMENTARY INSTRUCTION SET

Now you're ready to actually begin learning to use some Assembly Language instructions. In this chapter, we're going to introduce a very basic set of instructions; ones you'll need most of the time no matter what program you're writing. You'll learn enough instructions to be able to read data from the terminal, move data around from place to place inside the computer, add and subtract, write data out to the terminal, specify which instruction to process next, and stop processing.

When you have finished this chapter, you will be able to:

- code instructions to:
  - load registers from memory (LDA, LDX, LDY)
  - store data from registers in memory (STA, STX, STY)
  - transfer data between registers (TAX, TAY, TXA, TYA)
  - increment and decrement registers and memory (INC, INX, INY, DEC, DEX, DEY)
  - add and subtract using the carry flag (ADC, SBC)
  - set and clear the carry flag (SEC, CLC)
  - jump to another instruction (JMP)
  - jump to a subroutine (JSR)
- solve the following types of programming problems:
  - read data from a terminal
  - store data in memory
  - write data to a terminal
  - add and subtract
  - stop a program

LOADING REGISTERS FROM MEMORY

We'll start with some instructions that move data from a memory location to a register. This is called *loading* a register. Actually, the data is *copied*, not moved—the value at the original location is not changed.

1. The operation LDA (*LoaD Accumulator*) loads the accumulator with a byte of data from a memory location specified by an operand. The operand can be in any of these addressing modes:

- immediate
- direct
- zero-page direct
- indexed direct (using either the X or Y index register)
- zero-page indexed direct (using the X index register)
- pre-indexed indirect
- post-indexed indirect

(a) Which of the following instructions are valid?

```
_____ LDA #25

_____ LDA (START)

_____ LDA BEGIN+1

_____ LDA HERE,X

_____ LDA ($15),Y
```

(b) Code an instruction that will copy the value in location FIRST into the accumulator. ______________________

(c) Code an instruction that will set the accumulator to zero.

____

— — — — — — — — — —

(a) LDA #25, LDA BEGIN+1, LDA HERE,X, LDA ($15),Y. (LDA (START) is invalid because the operand is indirect); (b) LDA FIRST; (c) LDA #0

2. Look at the table in Appendix C. This table has a line for each operation code used in 6502 Assembly Language; it has a column for each of the nine addressing modes. As you learn about the operations, you will fill in the table so that it shows which addressing modes can be used with each operation. The table will become a handy reference document, so be sure to fill it in accurately.

LDA has been filled in for you. Notice that the columns for indexed direct and zero-page indexed direct specify which index registers can be used with the instructions. Why isn't it necessary to specify the registers which can be used in the pre-indexed and post-indexed indirect modes?

____

____

____

— — — — — — — — — —

Because the pre-indexed indirect mode always uses the X register, and the post-indexed mode always uses the Y register.

3. Data can be loaded into the X register and into the Y register using operation codes LDX (*LoaD X*) and LDY (*LoaD Y*). Each of these operations requires an operand that specifies where to find the byte to be loaded into the register. Immediate, direct, and zero-page direct operands can be used with both LDX and LDY.

Code instructions to:

(a) Set the Y register to zero. ________________

(b) Copy the data from location HITIME to the X register. ________________

(c) Load data from location $15 into the Y register. ________________

(d) Copy data from location $1212 to the accumulator. ________________

— — — — — — — — — —

(a) LDY # 0; (b) LDX HITIME; (c) LDY $15; (d) LDA $1212

4. LDX and LDY can also be coded using indexed direct and zero-page indexed direct operands. But with LDX, only the Y register can be used for indexing, and with LDY, only the X register can be used for indexing. Neither LDX nor LDY can use operands with indirect, pre-indexed or post-indexed indirect, or relative addressing.

(a) Fill in the lines for LDX and LDY in the table in Appendix C.

Which of these instructions are valid?

____ (b) `LDX ONE,X`

____ (c) `LDA (START),Y`

____ (d) `LDX HIGH`

____ (e) `LDY NEW,X`

____ (f) `LDY $15`

____ (g) `LDX (ONE)`

____ (h) `LDY (NEW,X)`

____ (i) `LDX #123`

____ (j) `LDA (ANEW,X)`

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
LDX	ok	ok	ok	Yok	Yok	--	--	--	--
LDY	ok	ok	ok	Xok	Xok	--	--	--	--

(c), (d), (e), (f), (i), and (j) are valid; (b) is invalid because LDX cannot use the X register for indexing; (g) is invalid because LDX cannot use indirect addresses; (h) is invalid because LDY cannot use a pre-indexed indirect address

5. LDA, LDX, and LDY instructions affect the zero and sign status flags. This means that if the byte moved has a value of zero, the zero flag will be set (=1); otherwise, it will be cleared (=0). The sign flag reflects the high-order bit of the byte moved. If you are using signed data, a one in this bit indicates a negative value, and a zero indicates a positive (or zero) value. (Signed numbers are discussed in Chapter 11.) Will the sign and zero flags be on or off after each of these instructions?

(a) LDY WHICH, where the location labeled WHICH contains $00.

____

(b) LDX LESS, where the location labeled LESS contains −25.

____

(c) LDA OK, where the location labeled OK contains 75. ______________

(d) LDY #130. ______________________________

\- - - - - - - - - -

(a) Zero = on, Sign = off; (b) Zero = off, Sign = on; (c) Zero = off, Sign = off; (d) Zero = off, Sign = on [the value is greater than +127, or +$7F]

Now you have learned how to move data from memory locations into the accumulator (LDA), X register (LDX) and Y register (LDY). You have learned that:

- all three of these operations can use direct, zero-page direct, indexed direct, and zero-page indexed direct addresses, with the restriction that the register being loaded cannot be used for indexing;
- the LDA operation can also use pre- or post-indexed indirect addresses;
- none of the three can use indirect or relative addressing;
- all three operations set the zero and sign flags.

In the next part of the chapter, you'll learn how to move data from these registers to memory and how to move data between these registers.

STORING DATA IN MEMORY

6. A byte of data can be copied from the accumulator, the X register. or the Y register to a memory location. This is called *storing* data in memory. The operation codes used are: STA (*ST*ore *A*ccumulator), STX (*ST*ore *X*), and STY (*ST*ore *Y*). These operations do not affect any status flags, but each of them requires an operand to specify the location where the byte of data is to be stored. Direct and zero-page direct operands can be used with any of these operations. Code instructions to:

(a) Copy the value from the accumulator to the location labeled HERE.

___

(b) Store the Y register in byte $15 on the zero page.

___

(c) Move the value in register X to byte $20 on page $17.

___

— — — — — — — — — —

(a) STA HERE; (b) STY $15; (c) STX $1720

7. In addition to direct and zero-page direct, one other addressing mode can be used with STX and STY instructions—zero-page indexed direct, using the Y index register with STX, and the X index register with STY. The STA instruction can use both indexed direct and zero-page indexed direct addresses, but only with the X register. STA can also use pre-indexed and post-indexed indirect addresses. Fill in the lines in Appendix C for the *ST*ore operations.

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
STA	--	ok	ok	Xok	Xok	--	ok	ok	--
STX	--	ok	ok	--	Yok	--	--	--	--
STY	--	ok	ok	--	Xok	--	--	--	--

8. The 6502 assembler instructions do not include any instructions that move data from one memory location to another. To copy a value from a location named FIRST to a location labeled SECOND, you need a series of two instructions, for example:

```
LDA FIRST
STA SECOND
```

Code a routine using the X register to move a byte of data from SECOND to THIRD; then use the Y register to go from THIRD to HOME. (A 'routine' is a series of instructions.)

— — — — — — — — — — —

```
LDX SECOND
STX THIRD
LDY THIRD
STY HOME
```

TRANSFERING DATA BETWEEN REGISTERS

9. Data can be moved from the accumulator to the X or Y register, and from the X or Y register to the accumulator. This process is called transferring data between registers. The operation codes used are:

TAX (*T*ransfer *A*ccumulator to *X*)
TAY (*T*ransfer *A*ccumulator to *Y*)
TXA (*T*ransfer *X* to *A*ccumulator)
TYA (*T*ransfer *Y* to *A*ccumulator)

These operations don't need operands; the operation codes specify where the data comes *from*, and where it's going *to*. (There are no instructions that transfer data directly between the X and Y registers.)

(a) Write instructions to do the following:

Transfer the value in the accumulator to the X register.

____

Copy the value from the Y register to the accumulator.

____

Move the value from the X register to the Y register.

____

(Note: You will need to use a *routine* with *two* instructions.)

(b) Fill in the lines for TAX, TAY, TXA, and TYA in Appendix C.

— — — — — — — — — —

(a) TAX; TYA; TXA followed by TAY (Note: You could also do this by moving the data to and from a memory location. For example, STX NAME1, LDY NAME1)

(b)

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
TAX	--	--	--	--	--	--	--	--	--
TAY	--	--	--	--	--	--	--	--	--
TXA	--	--	--	--	--	--	--	--	--
TYA	--	--	--	--	--	--	--	--	--

10. The transfer instructions affect the zero and sign status flags, just as the load register instructions do. In the table below, the first line shows the values of some flags, registers, and memory locations at the beginning of a routine. Fill in the table with the values after each step of the routine has been executed.

INSTRUCTION	FLAGS ZERO	SIGN	ACCUM	X	Y	HERE	THERE	WHAT
start	OFF	OFF	12	−3	137	0	2	−5
(a) LDX HERE								
(b) TYA								
(c) STA THERE								
(d) STX THERE								
(e) LDA #2								
(f) TAY								
(g) LDA WHAT								
(h) STX WHAT								

— — — — — — — — — — —

INSTRUCTION	FLAGS ZERO	SIGN	ACCUM	X	Y	HERE	THERE	WHAT
start	OFF	OFF	12	−3	137	0	2	−5
(a) LDX HERE	ON	OFF	12	0	137	0	2	−5
(b) TYA	OFF	ON	137	0	137	0	2	−5
(c) STA THERE	OFF	ON	137	0	137	0	137	−5
(d) STX THERE	OFF	ON	137	0	137	0	0	−5
(e) LDA #2	OFF	OFF	2	0	137	0	0	−5
(f) TAY	OFF	OFF	2	0	2	0	0	−5
(g) LDA WHAT	OFF	ON	−5	0	2	0	0	−5
(h) STX WHAT	OFF	ON	−5	0	2	0	0	0

INPUT AND OUTPUT SUBROUTINES

11. Almost every program needs to read some data from an input device and write some data to an output device. Unfortunately, it's very difficult to code routines to do this. There are many different types of devices available and each has its own peculiarities.

For now, we're going to avoid any details of how to code I/O routines. We're going to use two instructions, JSR INPUT and JSR OUTPUT, to handle our I/O needs.

The JSR (*J*ump to *S*ub*R*outine) instruction transfers control to a *subroutine*—a series of instructions that accomplish a function. When the subroutine ends, control is automatically returned to the instruction *after* the JSR instruction. JSR must use a direct operand.

Suppose INPUT is the label of a subroutine that reads a byte from a CRT terminal and leaves the byte in register A. Examine the following code:

```
JSR   INPUT
TAX
```

(a) What instruction is executed after JSR INPUT? ____________________

(b) What instruction is executed after control returns from the INPUT subroutine?

____

(c) What is the effect of this routine? ____________________

(d) Fill out Appendix C for JSR.

— — — — — — — — — —

(a) the first instruction of the INPUT subroutine; (b) TAX; (c) read a byte and store it in register X

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(d) JSR	--	ok	--	--	--	--	--	--	--

12. The advantage of a subroutine is that a program can call it over and over again, but it needs to be coded only once. Every time it is called, control returns to the calling location.

Examine this routine:

```
JSR  INPUT
TAX
JSR  INPUT
TAY
```

(a) What instruction is executed after control returns from INPUT the first time?

____

(b) What instruction is executed after control returns from INPUT the second time? ____________________

(c) What is the total effect of the routine? ____________________

____

____

— — — — — — — — — —

(a) TAX; (b) TAY; (c) two bytes are read—the first is stored in X and the second is stored in Y

13. Suppose OUTPUT is a subroutine that writes the byte in the accumulator to the terminal (the same terminal that INPUT reads from).

Assume that both our INPUT and OUTPUT routines return the registers in exactly the same condition that they received them. That is, if X contained 5 before the routines are called, it contains 5 afterwards. The same is true for the Y register, status registers, etc.

(a) Code a routine to write the data in register X to the terminal.

(b) Code a routine to write the data in memory location OUTBYT to the terminal.

— — — — — — — — — —

(a)
```
TXA
JSR   OUTPUT
```

(b)
```
LDA   OUTBYT
JSR   OUTPUT
```

14. Terminal input and output are separate functions as far as a duplex system is concerned. When a user types a character for input, the character does not display unless the program writes it back to the terminal.

Code a routine that reads a byte from the terminal and immediately writes that byte back to the terminal. (We call this echoing.)

— — — — — — — — — —

```
        JSR   INPUT
        JSR   OUTPUT          ; ECHO INPUT BYTE
```

15. From now on, whenever you read a byte from the terminal always echo that byte immediately so the users can see what they're typing.

Adapt the routine shown below to include appropriate echoing.

```
JSR   INPUT
TAX
JSR   INPUT
TAY
```

- - - - - - - - - -

```
JSR   INPUT
JSR   OUTPUT
TAX
JSR   INPUT
JSR   OUTPUT
TAY
```

16. INPUT and OUTPUT are not the only subroutines a program might use. You might create and use many subroutines to accomplish your program logic. For example. suppose you have a subroutine named ASCBIN that converts the byte in A from ASCII to binary code.

(a) Code a routine to convert the byte at MEMBYT from ASCII to binary and store the result at NEWBYT.

(b) Code a routine to read a byte from the terminal, convert it to binary, and store the result at BINNUM.

— — — — — — — — — — —

(a)
```
LDA   MEMBYT
JSR   ASCBIN
STA   NEWBYT
```

(b)
```
JSR   INPUT
JSR   OUTPUT        ; ECHO
JSR   ASCBIN
STA   BINNUM
```

You'll learn a lot more about coding and using subroutines later in this book. But from here on in, we'll use INPUT and OUTPUT to communicate with a terminal hooked up to the microprocessor.

The instructions you have learned so far in this chapter all copy data from one location to another. In the next section, you are going to learn some of the basic instructions for *changing* data—addition, subtraction, incrementing and decrementing registers and memory.

STATUS FLAGS IN ARITHMETIC

17. Addition and subtraction instructions always store the result in the accumulator. The result affects the sign, overflow, zero, and carry flags.

- The *zero* flag will be turned on if the result of an addition or subtraction is zero. Otherwise, it will be turned off. Notice this means if the result = zero, the zero flag = 1, and vice versa.
- The *sign* flag will reflect the high-order bit of the result; if you add or subtract signed numbers and the sign flag gets turned on, it means the result is negative.
- The *carry* flag will be turned on if an addition results in a carry from the high-order bit. It will also be turned on by a subtraction that does *not* need to borrow in order to subtract the high-order bit.
- The *overflow* flag reflects the seventh bit of a result; it is also used when you are doing signed arithmetic; it can indicate an invalid result. We won't use it at this time.

Let's look at some examples, using unsigned numbers.

DECIMAL	HEX	BINARY	ACCUM	ZERO	SIGN	CARRY
150 + 110 260	$96 $6E $104	%10010110 %01101110 %100000100	$04	OFF	OFF	ON
170 − 20 150	$AA − $14 $96	%10101010 − %00010100 %10010110	$96	OFF	ON	ON

Notice that the carry flag is *on*; this is because we didn't need to borrow to subtract the high-order bit.

```
    20          $14       %000010100
-   20       -  $14    -  %000010100
     0          $00       %000000000      $00    ON    OFF    ON
```

Now you can try some. Just fill in the flags and the accumulator.

(a) 15 + 85 ______ ______ ______ ______

(b) 150 − 20 ______ ______ ______ ______

(c) 16 − 3 ______ ______ ______ ______

(d) 12 + 244 ______ ______ ______ ______

— — — — — — — — — —

	ACCUM	ZERO	SIGN	CARRY
(a)	$64	OFF	OFF	OFF
(b)	$82	OFF	ON	ON
(c)	$0D	OFF	OFF	ON
(d)	$00	ON	OFF	ON

ADDITION

18. Addition in 6502 Assembly Language always consists of adding the value of a byte from a memory location and the carry flag bit to the byte in the accumulator. The operation code used is ADC (*AD*d with *C*arry). The memory location is specified by an operand. Valid addressing modes for ADC are the same as the ones for LDA.

(a) Fill in the line in Appendix C for ADC.

(b) Code an instruction to add the value 15 to the accumulator.

____

(c) Code an instruction to add the value in UPDATE to the accumulator.

____

(d) How many bytes can the accumulator hold? ______________

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) ADC	ok	ok	ok	XYok	Xok	--	ok	ok	–

(b) ADC #15; (c) ADC UPDATE; (d) one

19. The value in the carry flag is always included in the addition when an ADC instruction is executed. If the value in the accumulator is 15 and we execute the instruction ADC #02, the result (in the accumulator) will be 17 if the carry flag is off. What will it be if the carry flag is on?

— — — — — — — — — —

18

20. When you add large numbers—those that need more than one byte of storage—you will need to include the carry from one byte to another. You will learn more about this in a later chapter. When you add one-byte numbers, you will want to be sure that the carry flag is off before you add. The instruction CLC (*CL*ear *C*arry) turns off the carry flag, without affecting anything else. No operand is required.

(a) Fill in the line for CLC in Appendix C.

(b) Write a routine to add 15 and 17.

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) CLC	--	--	--	--	--	--	--	--	--

(b)

```
LDA #15
CLC
ADC #17
```

or

```
CLC
LDA #17
ADC #15
```

21. Look at this routine:

```
        LDA #$FA
STEP1   CLC
        ADC #$06
        STA FIRST
        LDA #$03
STEP2   CLC
        ADC #$02
        STA SECOND
```

(a) At the end of this routine, what are the values in:

the accumulator? ______

FIRST? ______

SECOND? ______

the carry flag? ______

(b) If the instruction labeled STEP2 had been left out of the routine, what would the final values have been in:

the accumulator? ______

FIRST? ______

SECOND? ______

the carry flag? ______

— — — — — — — — — —

(a) accumulator = $05, FIRST = $00, SECOND = $05, carry = 0; (b) accumulator = $06, FIRST = $00, SECOND = $06, carry = 0 (the carry flag = 1 after the first ADC, but is reset by the second)

SUBTRACTION

22. The operation code for subtraction is SBC (*SuB*tract with Carry). It requires an operand, which can be in any of the addressing modes used with ADC and LDA.

(a) Fill in the line for SBC in Appendix C.

(b) Code an instruction to subtract 25 from the accumulator. ______

(c) Code an instruction to subtract the value at location WHAT from the accumulator. ______

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) SBC	ok	ok	ok	XYok	Xok	--	ok	ok	--

(b) SBC #25; (c) SBC WHAT

23. The SBC operation causes the byte specified by the operand to be subtracted from the accumulator. The *opposite*, or *complement*, of the carry flag bit is also subtracted from the accumulator. So, if the accumulator = 15, the instruction SBC #03 would cause the accumulator to be changed to 12 if the carry flag is *on* or to 11 if the carry flag is *off*. In either case, the carry flag will be turned on by the execution

of the instruction and the zero flag turned off. The sign and overflow flags will also be affected; but these are only significant when doing signed arithmetic. Fill in the blanks in the table below; the first line has already been done for you.

	BEFORE				AFTER		
	ACCUM	ZERO	CARRY	INSTRUCTION	ACCUM	ZERO	CARRY
	25	ON	ON	SBC #12	13	OFF	ON
(a)	115	OFF	ON	SBC #23	_____	_____	_____
(b)	10	OFF	ON	ADC #8	_____	_____	_____
(c)	87	ON	OFF	SBC #86	_____	_____	_____
(d)	32	OFF	OFF	SBC #23	_____	_____	_____

— — — — — — — — — —

(a)	92	OFF	ON
(b)	19	OFF	OFF
(c)	0	ON	ON
(d)	8	OFF	ON

24. The carry flag will be useful later when you learn to subtract large numbers. When you are working with one-byte numbers, you will want to be sure that the carry flag is *on* before you subtract. The instruction that turns on the carry flag without affecting anything else is SEC (*SE*t *C*arry). No operand is required.

(a) Fill in the line for SEC in Appendix C.

(b) Write a routine to subtract 12 from 125, and store the result in NEWNUM.

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) SEC	--	--	--	--	--	--	--	--	--

(b)
```
LDA #125
SEC
SBC #12
STA NEWNUM
```

INCREMENTING AND DECREMENTING

25. The X or Y registers are often used for counting. Incrementing (adding 1 to) or decrementing (subtracting 1 from) one of these registers would take at least four steps using instructions you have learned so far—for example, incrementing the X register would take these instructions:

```
TXA
CLC
ADC   #1
TAX
```

There is a simpler way. The instruction INX (*IN*crement *X*) does the same job; no operands are required; the zero and sign flags are affected. Similarly, INY (*IN*crement *Y*), DEX (*DE*crement *X*), and DEY (*DE*crement *Y*) instructions are available.

(a) Code one instruction to add 1 to the Y register. ______________

(b) Code one instruction to subtract 1 from the X register. ______________

(c) Fill in the lines on Appendix C for INX, INY, DEX, and DEY.

— — — — — — — — — —

(a) INY; (b) DEX;

(c)	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
INX	--	--	--	--	--	--	--	--	--
INY	--	--	--	--	--	--	--	--	--
DEX	--	--	--	--	--	--	--	--	--
DEY	--	--	--	--	--	--	--	--	--

26. It is also possible to increment or decrement a memory location, using the operations INC (*INC*rement) or DEC (*DEC*rement). Each of these operations requires an operand in one of four addressing modes—direct, zero-page direct, indexed direct, or zero-page indexed direct. Both of the indexed modes can use only the X register. The zero flag and sign flag are set by these instructions.

(a) Fill in the lines for DEC and INC in Appendix C.

(b) Code an instruction to decrement the byte in the location called LESS.

____

(c) Code a routine that will add 2 to the byte in location $15 without using the accumulator.

— — — — — — — — — —

		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a)	DEC	--	ok	ok	Xok	Xok	--	--	--	--
	INC	--	ok	ok	Xok	Xok	--	--	--	--

(b)
```
DEC LESS
```

(c)
```
INC $15
INC $15
```

BRANCHING

27. Usually a program is executed in the order it is written—the first instruction to be executed is on line 1, the second on line 2, etc. But often it's necessary to jump around, or branch, in the program. The operation code JMP (*JuMP*) will change the address in the program counter (PC) register, which always contains the address of the next instruction to be executed. JMP requires an operand to specify the new address to be put in the program counter. The operand can use direct or indirect addressing but it cannot use immediate, zero-page direct, any type of indexing, or relative address mode.

(a) Fill in the line in Appendix C for JMP.

(b) Code an instruction that will cause the next instruction to be the one at memory location $1531. ____________________

(c) Code an instruction that will cause control to jump to REPEAT.

____

(d) Code an instruction to jump to the address contained in the two bytes located at NEXT and NEXT+1. ____________________

— — — — — — — — — —

		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a)	JMP	--	ok	--	--	--	ok	--	--	--

(b) JMP $1531; (c) JMP REPEAT; (d) JMP (NEXT)

28. Often a program repeats the same series of instructions over and over. The JMP instruction can be used to create a loop, so the routine needs to be coded only one time. The routine

```
        LDA  #0
        LDX  #0
ADDONE  CLC
        ADC  #1
        STA  NUMBER,X
        INX
        JMP  ADDONE
```

would store the number 1 at location NUMBER, 2 at location NUMBER+1, 3 at NUMBER+2, etc. This is an example of a closed loop; there is no way out of it except to interrupt the program. In a later chapter, you will learn instructions that will allow you to leave a loop when you are through with it.

(a) Write a closed loop that will read values from a terminal (don't forget to "echo" the input).

(b) Write a closed loop that will count the number of times the loop is executed and display the count on a terminal each time this loop is restarted. Set the counter to $30 before entering the loop. This is the equivalent of ASCII zero.

— — — — — — — — — —

```
(a) NEXT    JSR  INPUT
            JSR  OUTPUT
            JMP  NEXT

            LDA  #$30
(b) LOOP    CLC
            ADC  #1
            JSR  OUTPUT
            JMP  LOOP
```

ENDING A PROGRAM

29. One way to end a program is to write a one-instruction closed loop that will repeat itself until you interrupt the program. Some 6502 microprocessor systems provide direct methods to return control to the operating system when a program is done, such as coding a JMP or JSR instruction with a specific address, or a BRK instruction, which we will discuss in a later chapter. You should check the documentation for your computer to find the recommended way to end a program; in this book we will use closed loops for this purpose.

Code a one-line closed loop to end a program. ______________________

\- - - - - - - - - -

```
ENDIT  JMP  ENDIT
```

REVIEW

In this chapter, you have learned a very basic set of instructions that you will need for almost every program.

- The sign flag will be adjusted to match the high order bit of a byte whenever an instruction changes the value in a register and whenever an arithmetic instruction changes a value in memory.
- The carry flag is set if an addition results in a carry or if a subtraction does *not* borrow. Otherwise, an arithmetic operation clears the carry flag. The ADC instruction adds the carry flag as well as the two addends. The SBC instruction subtracts the *complement* of the carry flag as well as the minuend.
- The LDA, LDX, and LDY instructions move data from memory into the accumulator, X register, and Y register. These instructions affect the zero and sign flags.
- The STA, STX, and STY instructions move data from the accumulator, X register, and Y register to memory. They do not affect the flags.
- The TAX, TAY, TXA, and TYA instructions move data between the accumulator and the X or Y register, and they affect the zero and sign flags.
- You have learned to add one-byte numbers, using CLC to clear the carry flag and then ADC for the addition. The sign, overflow, zero, and carry flags are affected.
- You have learned to subtract one-byte numbers, using SEC to set the carry flag and then SBC for the subtraction. The sign, overflow, zero, and carry flags are affected. Other memory operations, such as STA, do not affect the flags.

- The INC, INX, and INY instructions increment (add 1 to) a memory location, the X register, and the Y register. These instructions affect the zero and sign flags.
- The DEC, DEX, and DEY instructions to decrement (subtract 1 from) a memory location, the X register, and the Y register. These instructions affect the zero and sign flags.
- The JMP instruction causes a branch to another instruction and can be used to create a loop.
- The JSR instruction calls a subroutine. We will use the INPUT and OUTPUT subroutines throughout this book.
- You have learned how to end a program using a closed loop.
- You have filled in the addressing modes permitted with each of the instructions in the table in Appendix C.

In the Self-Test for this chapter, you'll get a chance to write routines that use these instructions.

CHAPTER 5 SELF-TEST

Code the instructions described below.

1. Load the byte from address LOMEM into the accumulator. ______
2. Copy the value from Y into X. ______
3. Store the byte from the accumulator at address $2105. ______
4. Add 40 (decimal) to the accumulator. ______
5. Jump to address $2000. ______
6. Load the hexadecimal value $21 into the accumulator. ______
7. Subtract one from the memory byte at address COUNTR. ______
8. Read a byte from the terminal into the A register. ______
9. Subtract $5 from the accumulator. ______
10. Copy the value from X into A. ______
11. Subtract one from the Y register. ______
12. Clear the X register (that is, load it with a zero.) ______
13. Add one to the memory byte at address $0027. ______
14. Jump to address NEWADD. ______
15. Write a byte from the A register to the terminal. ______
16. End the program by creating a closed loop. ______

17. Store the byte from the Y register at YREGIS. ______________________

18. Clear the carry flag. ______________________

19. Add one to the X register. ______________________

20. Add the value at address $16 to the accumulator. ______________________

Code the short routines described below.

21. Read and echo a byte at the terminal. Store the byte in MEMBYE.

22. Write the message "HI" to the terminal.

23. Create a loop to read and echo bytes and store them in memory. Store the first byte at INMESG. The second byte should be stored at INMESG+1, and so forth. Use indexed addressing to store the bytes.

24. Create a loop to read and echo bytes. Add 5 to each byte and write the new sum.

25. Copy a byte from OLDADD to NEWADD. ______________________

26. Subtract 6 from the value at memory address TOTSUM. The result should be at TOTSUM. ______________________

Self-Test Answer Key

1. `LDA LOMEM`

2. `TYA, TAX`

3. `STA $2105`

4. `ADC #40`

5. `JMP $2000`

6. `LDA #$21`

7. `DEC COUNTR`

8. `JSR INPUT`

9.
```
SEC
SBC #$5
```

10. `TXA`

11. `DEY`

12. `LDX #0`

13. `INC $0027`

14. `JMP NEWADD`

15. `JSR OUTPUT`

16. `LOOP JMP LOOP`

17. `STY YREGIS`

18. `CLC`

19. `INX`

20. `ADC $16` (You might have included CLC first.)

21.
```
JSR INPUT
JSR OUTPUT
STA MEMBYE
```

22.
```
LDA  #'H'
JSR  OUTPUT
LDA  #'I'
JSR  OUTPUT
```

23.
```
      LDX  #0
LOOP  JSR  INPUT
      JSR  OUTPUT
      STA  INMESG,X
      INX
      JMP  LOOP
```

24.
```
LOOP  JSR  INPUT
      JSR  OUTPUT
      CLC
      ADC  #5
      JSR  OUTPUT
      JMP  LOOP
```

25.
```
LDA  OLDADD
STA  NEWADD
```

26.
```
LDA  TOTSUM
SEC
SBC  #6
STA  TOTSUM
```

If you missed any items, reread the appropriate frames.

Now that you can code some complete routines, it would be nice if you could try them out on your computer. If you want to try, read Appendix D, which discusses some of the processes you'll need to go through.

CHAPTER SIX

ASSEMBLER DIRECTIVES

Your system will have a set of instructions that control the assembler program rather than the computer itself. We call these the assembler directives, because they direct the assembler rather than your programs. (Another common term for them is "pseudo-operations.")

The assembler directives are not standardized; different assemblers will have different ones. The set that we present in this chapter is used on our system. Even though your assembler may be different, you will probably have directives that serve the same purposes.

When you have completed your study of this chapter, you will be able to:

- answer questions about the assembly process;
- identify the difference between an assembler directive and a machine instruction;
- code the following assembler directives: DS, ASC, DFB, ORG, and EQU;
- organize all the parts of a complete program.

In order to understand the assembler directives, you have to understand what the assembler does. And in order to understand what the assembler does, you have to understand a little bit about 6502 machine language.

```
8000:               1           ORG   $8000
8000: A2 00         2           LDX   #0
8002: BD 16 80      3   MAPLOP  LDA   TEXT,X
8005: 9D 00 04      4           STA   $0400,X
8008: E8            5           INX
8009: E0 16         6           CPX   #22
800B: D0 F5         7           BNE   MAPLOP
800D: AC 54 C0      8   MAPOUT  LDY   $C054
8010: 8C 51 C0      9           STY   $C051
8013: 4C 13 80     10   LOOP    JMP   LOOP
8016: 50 4C 45     11   TEXT    ASC   'PLEASE TYPE YOUR NAME:'
8019: 41 53 45
801C: 20 54 59
801F: 50 45 20
8022: 59 4F 55
8025: 52 20 4E
8028: 41 42 45
802B: 3A

*** SUCCESSFUL ASSEMBLY:  NO ERRORS
```

FIGURE 10. Assembler Listing

1. Figure 10 is copy of a printout from a 6502 assembler. Each line shows the beginning address of the machine language code, the code itself, the line number of the Assembly Language instruction, and then the Assembly Language instruction. The address and the machine language code are given in hex. The prefix '$' is not used—you're supposed to know that it's hex.

In this instruction:

```
8013:   4C 13 80    10    LOOP    JMP   LOOP
```

(a) What's the memory address? ____________________

(b) What's the machine code? ____________________

(c) What's the Assembly Language code? ____________________

(d) What's the line number? ____________________

— — — — — — — — — —

(a) 8013; (b) 4C 13 80; (c) LOOP JMP LOOP; (d) 10

2. A machine language instruction can be composed of one, two, or three bytes. The first byte is always the operation code. This one-byte code tells the 6502 processor exactly what to do and what addressing mode to expect in the operand. For example, to a 6502 microprocessor, $9D means 'store the byte from the accumulator at a memory location indicated by an indexed direct address'; $9C means store the byte from the Y register at the location given by a direct address'; $E8 means 'add 1 to the X register.'

In the listing, you can see all the operation codes in this column.

```
8000:                1          ORG   $8000
8000: A2 00          2          LDA   #0
F002: BD 16 80       3  MAPLOP  LDA   TEXT,X
etc.
```

(a) How long is a machine language instruction? ____________

(b) Which part of the machine language instruction is the operation code?

____

(c) In this instruction:

```
800D:   AC 54 C0    10            LDY   $C054
```

What's the machine operation code for LDY? ____________

— — — — — — — — — —

(a) 1-3 bytes; (b) the first byte; (c) $AC

3. Many machine instructions are only one byte long. They have only an operation code and no operands. Assembly Language instructions that have no operands, such as TAX, get translated into one-byte machine instructions.

(a) Which of the following types of instructions translate as one-byte machine language instructions?

___ instructions with no operands

___ instructions with direct address operands

___ instructions with zero-page direct operands

___ instructions with immediate operands

(b) Which of the following instructions would translate into one byte?

___ `LDX $15`

___ `SBC #2`

___ `TAY`

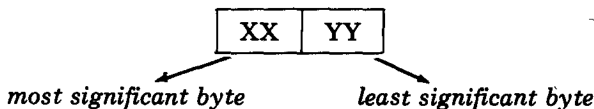
___ `STA $1575`

___ `INX`

— — — — — — — — — —

(a) instructions with no operands; (b) TAY, INX

4. Instructions with direct, indexed direct, or indirect operands get translated into three-byte machine instructions. The first byte contains the operation code. The second and third bytes contain the address specified in the operand. Remember that 6502 addresses are two bytes long.



In the machine language instruction, the least significant part of the address goes into the second instruction byte and the most significant part of the address goes into the third instruction byte. In other words, the two address bytes are reversed. The processor straightens them out when it uses that address.

For example, LDY $0555 is translated as AC 55 05. $AC is the machine operation code for LDY with a direct address, and 55 05 is the address of the operand with the bytes reversed.

The instructions LDY $0555,X and JMP ($0555) would also both have 55 05 as the second and third bytes of their machine code. The processor would know how to use these addresses by looking at the machine operation code: $B4 for LDY with an indexed direct operand; $6C for JMP with an indirect operand.

(a) Which of the following types of Assembly Language instructions have a three-byte machine language counterpart?

____ instructions with no operands

____ instructions with direct operands

____ instructions with immediate operands

____ instructions with indexed direct operands

____ instructions with indirect operands

(b) Which ones of the following instructions would translate into three-byte machine language instructions?

____ `INX`

____ `JMP ($2135)`

____ `SBC $15,X`

____ `ADC $1212`

____ `LDY #5`

____ `ADC $FEAB,Y`

(c) In the machine language instruction AC 54 C0, what address is referenced?

____

— — — — — — — — — —

(a) instructions with direct operands, instructions with indexed direct operands, instructions with indirect operands; (b) JMP ($2515), ADC $1212, ADC $FEAB,Y; (c) $C054

5. Instructions with immediate, zero-page, and relative addresses get translated into two-byte instructions. The first byte contains the machine code and the second byte contains the operand. Remember that all the pre- and post-indexed indirect operands are zero-page addresses.

Which of the following Assembly Language instructions have a two-byte machine language counterpart?

____ (a) instructions with no operands

____ (b) instructions with immediate operands

____ (c) instructions with pre-indexed indirect operands

____ (d) instructions with indirect operands

____ (e) instructions with zero-page direct operands

____ (f) instructions with post-indexed indirect operands

____ (g) instructions with indexed direct operands

____ (h) instructions with relative operands

____ (i) instructions with direct operands

____ (j) instructions with indexed zero-page operands

Which of the following instructions would translate into two-byte machine language instructions?

____ (k) `SBC $15`

____ (l) `SEC`

____ (m) `ADC ($10),Y`

____ (n) `BNE TOTAL`

____ (o) `JMP $1212`

____ (p) `LDA $FF,X`

____ (q) `ADC #241`

____ (r) `STA $0300,X`

____ (s) `LDA ($93,X)`

— — — — — — — — — —

(b), (c), (e), (f), (h); (k), (m), (j), (n), (p), (q), (s)

6. Let's review the subject of machine language instruction size. 6502 assembler instructions with no operands translate into one-byte machine language instructions. Those with operands in direct, indexed direct, or indirect addressing modes translate into three-byte machine language instructions. All others translate into two-byte machine language instructions—that includes those with operands in immediate, zero-page direct, zero-page indexed direct, pre-indexed indirect, post-indexed indirect, and relative addressing modes.

Fill in the table below to indicate whether each type of instruction has one, two, or three bytes. For each line, check the appropriate box.

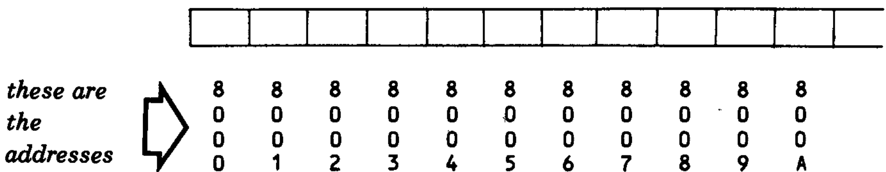
	TYPE	EXAMPLE	ONE BYTE	TWO BYTES	THREE BYTES
(a)	no operands	TAX	[ ]	[ ]	[ ]
(b)	immediate	LDA #$12	[ ]	[ ]	[ ]
(c)	direct	LDA $24B1	[ ]	[ ]	[ ]
(d)	zero-page direct	LDA $16	[ ]	[ ]	[ ]
(e)	indexed direct	LDA $24B1,X	[ ]	[ ]	[ ]
(f)	zero-page indexed direct	LDA $16,X	[ ]	[ ]	[ ]
(g)	indirect	JMP ($2416)	[ ]	[ ]	[ ]
(h)	pre-indexed indirect	LDA ($16,X)	[ ]	[ ]	[ ]
(i)	post-indexed indirect	LDA ($16),Y	[ ]	[ ]	[ ]
(j)	relative	BEQ START	[ ]	[ ]	[ ]

	TYPE	EXAMPLE		ONE BYTE	TWO BYTES	THREE BYTES
(a)	no operands	TAX		[X]	[ ]	[ ]
(b)	immediate	LDA	#$12	[ ]	[X]	[ ]
(c)	direct	LDA	$24B1	[ ]	[ ]	[X]
(d)	zero-page direct	LDA	$16	[ ]	[X]	[ ]
(e)	indexed direct	LDA	$24B1,X	[ ]	[ ]	[X]
(f)	zero-page indexed direct	LDA	$16,X	[ ]	[X]	[ ]
(g)	indirect	JMP	($2416)	[ ]	[ ]	[X]
(h)	pre-indexed direct	LDA	($16,X)	[ ]	[X]	[ ]
(i)	post-indexed indirect	LDA	($16),Y	[ ]	[X]	[ ]
(j)	relative	BEQ	START	[ ]	[X]	[ ]

Now you know how Assembly Language instructions get translated into machine language instructions. But where do the memory addresses come from? And how are labels interpreted as operands? We'll explore these things next.

7. Your program must be stored in main storage, or memory, in order to be executed. Only the machine code is stored. It's stored as a sequential series of bytes.

Imagine that the following diagram depicts the beginning of page 80 in main storage.



Each box represents the memory space to hold one byte. The numbers beneath represent the memory addresses, in hex.

If we loaded the machine code from Figure 10, it would look like this:

A2	00	BD	16	80	9D	00	04	E8	E0	16		etc.
8000	8001	8002	8003	8004	8005	8006	8007	8008	8009	800A		

The first line in Figure 10 is an assembler directive. It does not get stored, but it tells the processor to start loading the next instructions at address $8000. The program will be loaded into main storage in straight succession until another 'ORG' directive is found, or the last byte of code is loaded.

The first instruction, A2 00, goes into bytes 8000-8001. The second instruction, BD 16 80, goes into bytes 8002-8004.

(a) In Figure 10, the third machine instruction is: ______________________

Where will it be stored in memory? ______________________

(b) Suppose the program shown below was loaded into memory:

```
0000 20 50 AB   1     JSR   INPUT
0003 AA         2     TAX
0004 E8         3     INX
0005 85 12      4     STA   $12
. . .
```

Show the memory contents in the diagram below.

— — — — — — — — — —

(a) 9D 00 04, in bytes 8005-8007

(b)

20	50	AB	AA	E8	85	12	...	
0000	0001	0002	0003	0004	0005	0006	...	

8. Here is the memory diagram for Figure 10 again.

A2	00	BD	16	80	9D	00	04	E8	E0	16	D0	
8000	8001	8002	8003	8004	8005	8006	8007	8008	8009	800A	800B	

Compare it to the figure. The left-hand column of the listing gives the memory address for each instruction. It is the address of the *first byte*—the byte that contains the operation code. We call this the instruction address.

(a) Use the left-hand column in Figure 10 to locate the addresses of these instructions:

```
CPX    #22         ______________________________

MAPOUT LDY   $C054   ______________________________

STA    #0400,X     ______________________________
```

(b) What does the instruction address tell you? ______________________________

____

— — — — — — — — — —

(a) $8009, $800D, $8005; (b) the memory address of the first byte—the operation code—when the program is stored in memory

9. When a program is loaded for execution, the microprocessor will be told the starting address—for example, $8000. When the program is executed, the microprocessor begins by examining location $8000. From the operation code there, the microprocessor knows whether the first instruction is one, two, or three bytes. It picks up the instruction and advances the program counter (PC) to point to the next instruction address. If the first instruction is two bytes long, for example, the PC is set to 8002. Then the first instruction, whatever it is, is executed.

Assuming that the first instruction was not a jump or a branch, the PC is pointing at the first byte of the second instruction so that's the next instruction to receive control. The operation code is examined, the PC is incremented, and the instruction is executed. And so instructions are executed one after another straight through memory until a jump or branch is encountered.

Suppose your program looks like this:

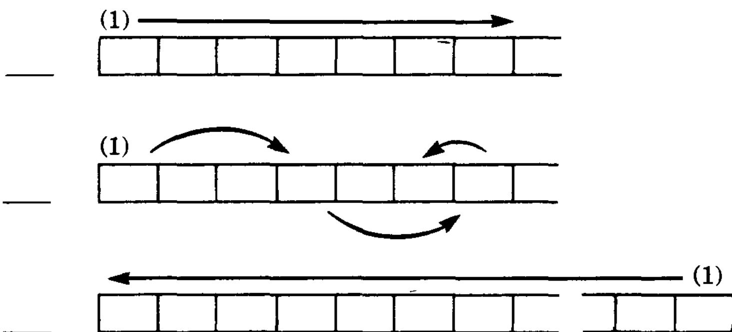
```
0000  A9 05    1          LDA   #5
0002  18       2          CLC
0003  69 0A    3          ADC   #10
0005  85 30    4          STA   $30
```

(a) What's the first instruction to be executed? (Write your answer in Assembly Language). ______________________

(b) Before it is executed, what will the PC be set to? ______________________

(c) What instruction will be executed second? ______________________

(d) Which diagram below best depicts the way in which instructions are executed as long as there are no jumps or branches?



- - - - - - - - - -

(a) LDA #5; (b) $0002; (c) CLC;

(d)

10. A jump or a branch causes the address in the PC to be replaced, thus changing the straightforward sequence of instructions.

Examine the program listing below.

```
0000  A9 00       1        LDA  #0
0002  4C 07 00    2        JMP  $0007
0005  A2 17       3        LDX  #23
0007  85 30       4        STA  $30
0009  4C 09 00    5        JMP  $0009
```

(a) When the program is loaded into memory, what is the first instruction to be executed? (Give your answer in Assembly Language.) ____________

(b) What is the second instruction to be executed? ____________

(c) What is the third instruction to be executed? ____________

(d) What is the fourth instruction to be executed? ____________

(e) When does the LDX instruction get executed? ____________

(f) (Extra thought question) What do you think would happen if the second instruction said JMP $0006 instead of JMP $0007?

____

____

____

— — — — — — — — — —

(a) LDA #0; (b) JMP $0007; (c) STA $30; (d) JMP $0009; (e) never; (f) the microprocessor would try to treat 17 as an operation code. If it is a legitimate operation code, that instruction would be picked up and executed, probably yielding a result that was not intended. If it isn't an operation code, the microprocessor will do strange things.

11. The last question in the preceding frame points out a programming problem—how do you know what address to jump to? Suppose you're writing the program shown below.

```
LDX   #0
LDA   MESAGE,X
JSR   OUTPUT
INX
JMP   ????
```

You want to jump back to the LDA instruction. But how can you know its address? Do you have to count bytes from the beginning of the program? You might as well code in machine language. No. You can use a symbolic address. You give the LDA instruction a label.

```
        LDX   #0
LOOP    LDA   MESAGE,X
        JSR   OUTPUT
        INX
        JMP   LOOP
```

And you use the label as your JMP operand. The assembler translates the label into an address.

Use Figure 10 to answer the questions below.

(a) What value did the assembler assign to the label LOOP? ________________

(b) What is the machine code for the instruction JMP LOOP? ________________

(c) How was the operand LOOP translated in the machine code?

____

— — — — — — — — — —

(a) \$8013; (b) 4C 13 80; (c) the address of the label LOOP was used as the operand, with bytes reversed. (Remember that when an address is stored in memory, the system looks for the least significant byte first, and the most significant byte following.)

12. Let's look at how the assembler handles labels. It actually processes your Assembly Language instructions in two passes. The first time through, it determines the address of each instruction and builds a table of labels. At that point, your program would look something like this:

```
8000:         1         ORG  $8000
8000:         2         LDX  #0
8002:         3  MAPLOP LDA  TEXT,X
8005:         4         STA  $0400,X
8008:         5         INX
8009:         6         CPX  #22
800B:         7         BNE  MAPLOP
800D:         8  MAPOUT LDY  $C054
8010:         9         STY  $C051
8013:        10  LOOP   JMP  LOOP
...          ...             ...

SYMBOL TABLE:
     MAPLOP = 8002
     MAPOUT = 800D
     LOOP   = 8013
```

The second time through, it translates the instructions. Whenever it encounters a label as an operand, it substitutes the appropriate address from its table. The final product looks like Figure 10.

Many assemblers are "two-pass" assemblers; they go through the program twice.

(a) What do they do on the first pass? ________________

____

(b) What do they do on the second pass? ________________

____

— — — — — — — — — —

(a) Build a table of addresses for all the symbolic labels; (b) translate the instructions, using addresses in place of labels.

You've seen how instructions are addressed. Data bytes also need addresses and a way to assign names to these addresses. Let's talk about how to set up data storage areas in memory.

DEFINING DATA AREAS

We use assembler directives to assign names to data storage areas. There are two major types of data storage—uninitialized and initialized. In this book, we'll use a DS (define storage) directive to create uninitialized data storage and ASC (ASCII) and DFB (define byte) directives to create initialized data storage areas. Your assembler may use different directives for these functions.

13. The DS directive defines uninitialized data storage. DS stands for "define storage."

This directive is used to reserve a group of bytes without having to define what is to be put in the bytes. The bytes will contain whatever values were there before; memory is not automatically cleared when a new program is started. We refer to these accidental values as "garbage."

Uninitialized space is usually used to store input values or the results of calculations. It doesn't need to be initialized because the new values will overlay the garbage values anyway.

The format for a directive is the same as that for an instruction, with the directive in place of the operation code. With the DS directive, the label and comments are optional. The operand specifies the number of bytes to be reserved. We usually code it in decimal, but you can use binary or hex in this book if you want. (Don't forget that your assembler may have a different instruction and/or different rules.)

Here's an example:

```
NAME  DS  10          ; THIS RESERVES 10 BYTES FOR A NAME
```

(a) What does uninitialized storage contain? ______________________

(b) Code a directive to save two bytes of uninitialized space called SPACE.

____

(c) Code a directive to save 20 bytes of uninitialized space called STORAG.

____

— — — — — — — — — —

(a) garbage; (b) SPACE DS 2; (c) STORAG DS 20 or STORAG DS $14

Initialized storage has values in it when the program is loaded and begun. The program defines the values to be placed there. The values might be used as *constants* (that is, values that don't change throughout the life of the program) or initial values of *variables* (that is, values that will change). An example of a constant might be a message that is written to the user, such as PLEASE ENTER YOUR USER NUMBER. An example of an initialized variable might be a page number initialized to 1.

14. The ASC directive defines storage initialized with a string of ASCII values. The operand is a string of ASCII characters, enclosed in single quotes. The label and comments are optional. Here's an example:

```
MESAGE ASC 'HI'   ; A BEGINNING MESSAGE
```

The size of the data area reserved will depend on the number of characters in the string. The example shown above will reserve two bytes.

(a) Write an ASC directive to initialize a five byte area with the value 'WHY '. Call the area QUEST. ______

(b) Look at line 11 in Figure 10. How many bytes are reserved by the ASC directive?

— — — — — — — — — —

(a) `QUEST   ASC   'WHY  '` (Note: don't forget the two spaces at the end; if you leave them out, you will only initialize a three-byte area):

(b) 22 bytes

15. The DFB directive defines storage initialized by any kind of value. DFB stands for "define byte."

The operand is a list of values, one for each byte to be defined. The values are separated by commas. Label and comments are optional. Here's an example:

```
TOMUCH  DFB  3,$15,12,7
```

The example defines four bytes. The first byte is called TOMUCH. If TOMUCH was at address 0000, the beginning of the memory map would look like this:

03	15	0C	07	. . .
0000	0001	0002	0003	

ASCII values can be defined using DFB. The example we used before could also be coded:

```
MESAGE   DFB  'H','I'
```

Notice that each character is enclosed in single quotes. The ASC directive is a little easier for most of us.

(a) Define a three-byte area initialized with the values 40, 10, 30. Call the area HOWMCH. ______________________________

(b) Define a four-byte area initialized with all binary zeros. Call the area COUNTR. ______________________________

(c) Revise the above definition so COUNTR is initialized with ASCII zeros.

____

— — — — — — — — — —

(a) `HOWMCH   DFB   40, 10, 30`

(b) `COUNTR   DFB   0,0,0,0`

(c) `COUNTR DFB '0','0','0','0'` or `COUNTR ASC '0000'`

16. A label defined by DS, ASC, or DFB has a memory address value. It can be used as an operand in Assembly Language instructions.

It can be used in place of an address in an operand, as in JMP *addr*. However, you should not jump to a data area because the computer can't interpret ordinary data as an instruction except accidentally.

Look at this section of a program:

```
OKMESS   ASC  'OK'
COUNT    DS   5
PAGNUM   DS   1
HILOW    DFB  $90,$10
```

If OKMESS is at address $2000. the beginning of page 20 looks like this:

ASCII code (bytes 2000–2001) *uninitialized* (bytes 2002–2007)

4F	4B	60	72	3F	FF	00	B1	90	10	. . .
2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	. . .

When our program refers to OKMESS, address $2000 will be referenced; when it refers to COUNT, address $2002 will be referenced.

Suppose we need to refer to address $2001 symbolically. At the end of Chapter 3, we talked about using expressions as operands. This is where they come in handy. Normally, we would reference $2001 as OKMESS+1; we could also call it COUNT−1, PAGNUM−6, or HILOW−7.

(a) What does the assembler do with the label of an ASC, DFB, or DS instruction? ______________________________

____

(b) In which instructions could you use the above labels?

___ JMP *label*

___ STA *label*

___ ADC *label*, X

(c) Write a command that would put the third byte of a storage area called LINENO into the X register. ______________________

(d) Look at Figure 10. Write a routine that will change the last character of TEXT from ':' to '>'.

— — — — — — — — — — —

(a) Converts it to an address value; (b) STA *label*, ACD *label*, X;

(c)
```
LDX   LINENO+2;
```

(d)
```
LDA   #'>'
STA   TEXT+21
```

(Note: If you coded TEXT+22, you forgot that the first byte is TEXT+0.)

THE ORG DIRECTIVE

17. The ORG (*OR*i*G*in) directive specifies the current memory address to the assembler. The directive ORG $0500 says, "No matter what memory address you're currently at, I want the next instruction to start at $0500." Subsequent instructions would follow $0500, of course.

In order to understand why ORG is important, you need to understand how your programs are ordinarily assembled and loaded into memory.

The assembler will put your first instruction at memory address $0000. In the code below:

```
START  LDX  #0
       STX  COUNTR
       ...
```

the instruction for LDX #0 will be assigned to addresses $0000 and $0001. The code for STX COUNTR will be assigned to addresses $0002, $0003, and $0004.

After the program has assembled, it can be run. When you give the command to run it, it is loaded into memory at the addresses assigned by the assembler.

(a) Where does the assembler begin assigning addresses? ____________

(b) If the assembler assigns an address of $0014 to an instruction, where will that instruction be loaded when the program is executed? ____________

— — — — — — — — — —

(a) $0000: (b) at address $0014

18. Sometimes we don't want to load our programs into address $0000 and subsequent addresses. Most of our computers have a system monitor—an executive program with its own instructions and data. We want to avoid overlaying the system monitor with our program.

Suppose your system monitor uses addresses $0000 through $0050 for its zero-page data. Its instructions are on pages $A0 through $D5 and its subroutines are on pages $F0 through $FD.

(a) Where could you put your zero-page data without interfering with the system monitor's zero page data? ____________

(b) Where could you start your instructions and subroutines? They shouldn't be on page 0 or 1 (page 1 is reserved for the stack). ____________

— — — — — — — — — —

(a) at address $0051 through $00FF; (b) at address $0200 (page 2)

19. How do you tell the assembler to put your zero-page data starting at address $0051 and your instructions starting at page $02? You use the ORG directive. Here's an example:

```
        ORG   $0051
STOREX  DS    1
STOREY  DS    1
YESMSG  ASC   'THAT IS CORRECT'
NOMSG   ASC   'NO -- TRY AGAIN'
HUHMSG  ASC   'HUH?'
COUNTR  DFB   00
FRAMNO  DFB   0,1
        ORG   $0200
START   LDX   #1
        JSR   INPUT
        ...
```

(a) What address will be assigned to STOREX? ______

(b) What address will be assigned to STOREY? ______

(c) What address will be assigned to the first byte of YESMSG? ______

(d) What address will be assigned to the LDX #1 instruction? ______

(e) When we run this program, what will be loaded into address $0000? ______

— — — — — — — — — —

(a) $51; (b) $52; (c) $53; (d) $0200; (e) nothing; this program doesn't affect address $0000

20. Add assembler directives to the following code so the data is stored on the zero page beginning at address $A0, and the instructions begin on page $05.

```
SAVEA   DS    1

SAVEX   DS    1

SAVEY   DS    1

COUNTR  DS    2

PAGENO  DFB   1

BLOKNO  DFB   1

        LDA   #$0A

        JSR   OUTPUT

        ...
```

— — — — — — — — — —

```
        ORG   $00A0
SAVEA   DS    1
SAVEX   DS    1
SAVEY   DS    1
COUNTR  DS    2
PAGENO  DFB   1
BLOKNO  DFB   1
        ORG   $0500
        LDA   #$0A
        JSR   OUTPUT
        ...
```

21. If your first instruction is not loaded at address $0000, how does the system know where to find it? The answer depends on the system.

Some systems *require* the first instruction to be stored at location $0000. Here's how we handle that:

```
        JMP   START          ; COULD SAY JMP   $0200
        ORG   $0051
SAVEX   DS    1
SAVEY   DS    1
        ...
        ORG   $0200
START   LDX   #1
        ...
```

The JMP START instruction will be assembled at addresses $0000 and $0001. It will cause a jump to our *real* first instruction, LDX at address $0200.

Some systems assume the lowest address used by the program when it is loaded contains the first instruction. Here's how we can handle that.

```
        ORG   $0051
        JMP   START
SAVEX   DS    1
SAVEY   DS    1
        ...
        ORG   $0200
START   LDX   #1
        ...
```

When the program is loaded, control is given to address $51, where it is immediately jumped to address $0200, our real first instruction.

Suppose your system expects the first instruction to be loaded at address $0000. Adapt your code from the previous frame so the system can find your *real* first instruction.

```
        ORG   $00A0
SAVEA   DS    1
SAVEX   DS    1
SAVEY   DS    1
COUNTR  DS    2
PAGENO  DFB   1
BLOKNO  DFB   1
        ORG   $0500
        LDA   #$0A
        JSR   OUTPUT
        ...
```

— — — — — — — — — —

All you need to do is add a JMP $0500 instruction at the beginning:

```
JMP   $0500
ORG   $00A0
```

This completes our discussion of the ORG directive. You will see it used occasionally throughout the remainder of this book. You'll probably need to use it with your system if you don't want to overlay the system monitor.

THE EQU DIRECTIVE

22. Let's move on now to another assembler directive. The EQU (equate) directive directly assigns a value to a label. *Any* value that is acceptable to your assembler (usually 0 ... 65535) can be assigned. For example:

```
HIVAL  EQU  $FF
LOVAL  EQU  $00
```

Now anywhere in the program that the label HIVAL is used as an operand, the assembler will substitute $FF, and $00 will be substituted for LOVAL.

Note the important difference between equates and symbolic addresses. In the above example, HIVAL and LOVAL are *not* symbolic addresses. They do not have address values; they have the value of their operands.

The EQU directive does not define a storage area. It is not translated into a machine instruction. It simply tells the assembler, "When I say *this*, I really mean *that*."

(a) Code a directive to assign the value $10 to the label TTYPRT.

___

(b) Code directives to assign the value $0D to the label CR and the value $0A to the label LF.

___

___

— — — — — — — — — —

(a) TTYPRT EQU $10; (b) CR EQU $0D; LF EQU $0A

23. An EQU label can be used for any operand where its value makes sense. For example, suppose you have this set of equates:

```
ADDIT  EQU  15
HIVAL  EQU  $FF
LOVAL  EQU  $00
MESAG  EQU  'HI'
STORIT EQU  $1517
```

MESAG and STORIT are two-byte values; the others are one-byte values.

The instruction LDA #HIVAL would be the same as LDA #$FF; LDA LOVAL would be equivalent to LDA $00, or load A from the byte at address $0000.

The instruction LDA #STORIT would not make sense since you can't have a two-byte immediate operand, but LDA STORIT would be valid.

For each of the following operand types, indicate which of the above labels could be used:

(a) immediate ___

(b) direct ___

(c) zero-page indexed direct ___

— — — — — — — — — —

(a) #ADDIT, #HIVAL, and #LOVAL could all be used as immediate operands because they all represent one byte values; (b) MESAG (which is $4849) and STORIT could both be used as direct addresses because they both represent two-byte values; (c) ADDIT, HIVAL, and LOVAL could all represent zero-page addresses

24. Why do we use equates? Why not use the values themselves directly as operands? Because equates make it easier to revise a program. Suppose you need to change the address of an output port on your system from $0210 to $0215. If you have defined that address this way:

```
OUTPRT EQU  $0210
```

and then used OUTPRT in the instructions, you have to make only *one change*. If you didn't use the equate, you'll have to search the entire program for references to the $0210 address.

A programmer spends about 25% of the time writing new programs and 75% of the time revising old programs—correcting, updating, expanding, adapting, and so forth. All new programs should be written with the thought in mind that they will be revised at least ten times before they have outlived their usefulness. Equates are one way to make the revision task easier later on.

(a) A really good program never needs to be changed.

True or false? ________________

(b) How do equates make revisions easier? ________________________________

____

— — — — — — — — — — —

(a) false—the better a program is, the more likely it is to be borrowed, adapted, expanded, etc.; (b) by cutting down the number of instructions that have to be changed

25. Equates look similar to DFBs and ASCs, but they have different effects and different uses.

Match:

_______ (a) EQU

_______ (b) DFB and ASC

1. stores values in memory
2. does not store values in memory
3. label can have any value, of any length
4. label has a two-byte address value
5. label can be used in place of an address in an operand
6. label can be used as an address, or as an immediate byte

— — — — — — — — — — —

(a) 2, 3, 6; (b) 1, 4, 5.

26. When do you use EQU to define a value and when would you use DS, DFB, or ASC? The answer lies in whether you need the value to be stored in memory. If the value is used as data, which is operated on by an instruction, it needs to be stored in memory. If the value is used as an operand, then you can use EQU to define it. We usually use EQU to define values for immediate operands.

Here are some examples:

```
LDX  #0
```

The zero is an immediate operand. It does not need to be stored in memory since it is included in the instruction. You could code it this way instead:

```
ZERO   EQU   0
       ...
       LDX   #ZERO
```

Here's another example:

```
LDA  #$0D
```

Again, the immediate byte can be equated, as in:

```
CR     EQU  $0D
       ...
       LDA  #CR
```

There are two advantages to doing this. First, it's clearer to someone reading the program that we're loading a carriage return code into A. Second, if we want to move our program to a system that has a different code for a carriage return, we have to make the change in only one place—the EQU directive.

Here's another example:

```
LDY  INDEXY
```

Here, INDEXY is a direct address. The value stored at that address is the data used in this instruction. The *value* must be stored in memory, so we would define INDEXY this way:

```
INDEXY  DS   1
        ...
        LDY  INDEXY
```

Suppose, instead, we defined it this way:

```
INDEXY  EQU  1
        ...
        LDY  INDEXY
```

The LDY instruction would translate as LDY 1. The operand would be interpreted as a zero-page address, equivalent to $0001, and the value from that address would be loaded into Y.

You *can* define addresses in EQU instructions. Here's an example:

```
INPUT     EQU   $FD0C
OUTPUT    EQU   $FDED
          ...
          JSR   INPUT
          JSR   OUTPUT
```

When the system executes the JSR INPUT example, it will jump to the subroutine beginning at address $FD0C. JSR OUTPUT will jump to the subroutine beginning at $FDED. We use this technique to reference addresses outside the domain of our program. For addresses within the program, put the labels on the routines themselves.

The code below shows some values that we would define with EQU directives and suggests the directives we would use.

```
NEWLIN    LDA   #$0D       CR EQU $ØD
          JSR   $FDED      OUTPUT EQU $FDED
          LDA   #$0A       LF EQU $ØA
          JSR   $FDED
          LDX   #0         ZERO EQU Ø
MESSAG    LDA   OUTEXT,X
          JSR   $FDED
          INX
          CPX   #24        LENG24 EQU 24
          BNE   MESSAG
          ...
OUTEXT    ASC   'I DON'T UNDERSTAND THAT!'
```

Now you try it. Indicate which operands below could be replaced by labels from EQU directives and show the directives you would use. Also show any DS, DFB, or ASC directives that are needed by the routine.

```
SAVREG    STX   SAVEX
          STY   SAVEY
          LDX   #1
          LDY   #25
          JSR   WRITIT
          LDX   SAVEX
          LDY   SAVEY
          ADC   #5
          BCC   DOLOOP
          ...
```

— — — — — — — — — —

Here's what we would do:

```
SAVREG  STX  SAVEX
        STY  SAVEY    ONE EQU 1
        LDX  #1
        LDY  #25      TWFIVE EQU 25
        JSR  WRITIT
        LDX  SAVEX
        LDY  SAVEY    FIVE EQU 5
        ADC  #5
        BCC  DOLOOP
        ...
SAVEX   DS   1
SAVEY   DS   1
```

27. Many systems have a special equate instruction that looks like this:

label EQU *

The * operand says "this address." The EQU instruction assigns the current address as the value of the label. Since the EQU instruction doesn't have its own address (it's not translated into machine language), the label becomes the address of the next Assembly Language instruction. Thus:

```
MIXER  EQU  *
       JSR  INPUT
```

is the same as:

```
MIXER  JSR  INPUT
```

Why use the equate? To make future revisions easier. Suppose you find you need to add an instruction to the beginning of the MIXER routine. It's easier to make the revision if the symbolic address has its own line.

We will use the EQU * instruction throughout the remainder of this book. Your assembler may not recognize the instruction and may have a different way of assigning labels to routines. You'll have to check your assembler manual to see what you can use.

The familiar echo routine is shown below. Revise it so the label is on a separate line.

```
ECHO    JSR   INPUT
        JSR   OUTPUT
        JMP   ECHO
```

— — — — — — — — — —

```
ECHO    EQU   *
        JSR   INPUT
        JSR   OUTPUT
        JMP   ECHO
```

28. Suppose you're writing a rather long program. Your system recommends starting all programs at $1000. Good programming practice recommends putting all data definitions after the last instruction. You want to follow these recommendations but you do have two frequently used data items that should be on the zero page, starting at address $10. (Assume your operating system uses addresses $00-$0F for its own purposes.)

```
CR      EQU $0D
LF      EQU $0A
HIVAL   EQU $FF
ZERO    EQU 0
        ORG $10
COUNTR  DS  2
PAGNUM  DFB 1
PROMPT  ASC '>'
QUESTN  ASC '?'
        ORG $1000
NEWPAG  EQU *
        LDA PAGNUM
        JSR OUTPUT
        . . .
END     JMP END
HEADNG  ASC 'DISK UTILIZATION PAGE: '
TEMPV   DS  1
NEWSIZ  DS  1
        . . .
```

FIGURE 11. Sample Program Layout

Figure 11 shows a good way to lay out the program. First code the data equates. These aren't stored, but when they appear first, the assembler can use them as it processes later instructions. Next use ORG to get to address $0010. Then define the data you want to put on the zero page, and use ORG to get to address $1000. Here you can actually begin the instructions that will get translated into machine code.

The END JMP END instruction halts the program. We've included that instruction to show that some more data definitions follow it. It's safe to put the definitions there because control won't accidentally fall through from the JMP instruction to the next byte of memory. Always make sure that control doesn't get into your data areas when writing your programs.

(a) Where do you usually put the EQU instructions that assign specific values to labels? (Not the EQU * instructions.)

____

____

(b) Rearrange the following program so that all the equates come first, all the non-ASCII data is on the zero page (starting at address $0050), and all the instructions are on page $04. The ASCII data should follow the instructions. Put an instruction at location $0000 that jumps to your first instruction at $0400.

```
ZERO    EQU 0
YRFLAG  DFB ZERO
YEAR    ASC '8586'
MESAG   ASC 'THE YEAR IS '
OUTPUT  EQU $C055
DOYEAR  EQU *
        LDX #ZERO
MSGLOP  EQU *
        LDA MESAG,X
        JSR OUTPUT
        INX
        CPX #12
        BMI MSGLOP
        LDX #ZERO
        LDA YRFLAG
        BEQ YROUT
        LDX #2
YROUT   EQU *
        LDA YEAR,X
        JSR OUTPUT
        INX
        LDA YEAR,X
        JSR OUTPUT
DONE    JMP DONE
```

— — — — — — — — — —

(a) at the beginning of the program;

(b)
```
ZERO    EQU 0
OUTPUT  EQU $C055
        ORG $0
        JMP DOYEAR
        ORG $50
YRFLAG  DFB ZERO
        ORG $0400
DOYEAR  EQU *
        LDX #ZERO
MSGLOP  EQU *
        LDA MESAG,X
        JSR OUTPUT
        INX
        CPX #12
        BMI MSGLOP
        LDX #ZERO
        LDA YRFLAG
        BEQ YROUT
        LDX #2
YROUT   EQU *
        LDA YEAR,X
        JSR OUTPUT
        INX
        LDA YEAR,X
        JSR OUTPUT
DONE    JMP DONE
YEAR    ASC '8586'
MESAG   ASC 'THE YEAR IS '
```

REVIEW

In this chapter, you have studied several assembler directives. Your assembler directives may have different names, but they should include at least the functions shown here.

- The assembler program translates Assembly Language instructions into machine language instructions. The operation code is translated into a numeric machine code. Operand types are included in the machine code. Addresses are converted into binary and their bytes are reversed. Immediate data is converted into binary. Symbolic addresses are given numeric address values. Labels defined by equates are given their equated values.

- Assembler directives speak to the assembler program. They are not translated into machine language although their effect may be seen in the machine code that is produced.

- The DS directive defines uninitialized memory space.

  Format: [*label*] DS *size* [;*comments*]

  Size gives the number of bytes to set aside. The bytes will contain garbage. The label becomes the symbolic address for the first byte of that memory area; it can be used as an operand.

- The ASC directive defines memory space initialized by ASCII data.

  Format: [*label*] ASC '*text...*' [;*comments*]

  The number of bytes set aside will depend on the number of characters in the text. The label becomes the symbolic address for the first byte of the memory area; it can be used as an operand.

- The DFB directive defines and initializes memory bytes.

  Format: [*label*] DFB *value*[,*value...*] [;*comments*]

  Each value defines one byte. The label becomes the symbolic address for the beginning of that memory area; it can be used as an operand.

- The ORG directive specifies the current memory location to assembler.

  Format: [*label*] ORG *addr* [;*comments*]

  ORG is used to skip over memory space, either because it's in use by other programs or to reserve data space without using the DS instruction. It's most often used to specify where the program starts in memory.

- The EQU instruction assigns a value to a label.

  Format: *label* EQU *value* [;*comments*]

  The assembler substitutes the value for the label wherever the label is used as an operand.

- *label* EQU * is a special instruction that assigns a symbolic address to the current memory address.

CHAPTER 6 SELF-TEST

1. Describe the function of the assembler.

   ____

2. Which of the following become part of the machine language program?

____ a. labels

____ b. operation codes

____ c. operands

____ d. comments

3. Which of the following are replaced by the assembler with address values?

____ a. labels

____ b. operation codes

____ c. operands

____ d. comments

4. Which of the following are ignored by the assembler?

____ a. labels

____ b. operation codes

____ c. operands

____ d. comments

5. Refer to Figure 12 and answer the questions below.

   a. What is the address of the INX instruction? ____________

   b. What is the value of the label INLOOP? ____________

   c. Look at the JSR instruction at address $8002. What value did the assembler substitute for the operand INPUT? ____________

   Why? ____________

   ____

```
8000:              1         ORG  $8000
8000:              2  CR     EQU  $8D
8000: A9 00        3         LDX  #0
8002:              4  INLOOP EQU  *
8002: 20 00 90     5         JSR  INPUT
8005: 20 80 90     6         JSR  OUTPUT
8008: 9D 14 80     7         STA  INTEXT,A
800B: E8           8         INX
800C: 38           9         SEC
800D: E9 8D       10         SBC  #CR
800F: D0 F1       11         BNE  INLOOP
8011: 4C 11 80    12  DONE   JMP  DONE
8014:             13  INTEXT DS   80
 . . .                       . . .
9000:             20         ORG  $9000
9000:             21  INPUT  EQU  *
 . . .                       . . .
9080:             30         ORG  $9080
9080:             31  OUTPUT EQU  *
 . . .                       . . .
```

FIGURE 12. Parts of an Assembler Listing

d. Look at the SBC instruction at address $800D. What value did the assembler substitute for CR? ______

Why? ______

______

6. Match.

______ a. assembler directive — 1. translated into machine language; controls the microprocessor.

______ b. instruction — 2. not translated; controls the assembler program.

7. Code directives for each of the following functions.

a. Define an uninitialized 10-byte data area named QUES7A.

______

b. Define a data area named QUES7B initialized to $5B.

______

c. Define a string of initialized bytes containing the digits 0 through 9 in pure binary code, not ASCII. Name the area DIGITS.

______

d. Define a string of initialized bytes containing "SELF-TEST." Name the area QUIZ.

______

e. Set the label ZEROS equal to zeros. ______

f. Set the label LIMIT equal to '**'. Don't store it in memory.

______

g. Assign the label ANSWER to the first instruction of the routine below. Use an EQU statement.

```
LDA  NOTEXT,X
INX
JSR  OUTPUT
```

h. Cause the routine below to be stored beginning at $0200.

```
LDA  NOTEXT,X
INX
JSR  OUTPUT
```

8. The following simple program reads, echos, and stores 10 bytes. Set up the program so the zero-page data, COUNTR, is at address $05. Initialize it to a value of ten. The instructions should start at the beginning of page $06. The first byte in memory, $0000, should be a jump to the first instruction.

Code the following equates: INPUT=$C054, OUTPUT=$C0A4, ZERO=0. Also, define a storage area (STORAG) to hold ten bytes. Put it at the end of the instructions instead of on the zero page.

```
       LDX #ZERO
INSTOR EQU *
       JSR INPUT
       JSR OUTPUT
       STA STORAG,X
       INX
       DEC COUNTR
       BNE INSTOR
STOP   JMP STOP
```

Self-Test Answer Key

1. translates Assembly Language instructions into machine instructions

2. b, c

3. a

4. d

5. a. $800B

   b. $8002

   c. $0090; this is the address of the instruction labeled INPUT ($9000) with the bytes reversed

   d. $8D; CR is assigned this value in an EQU directive

6. a. 2

   b. 1

7. a. `QUES7A DS    10`

   b. `QUES7B DFB   $5B`

   c. `DIGITS DFB   0,1,2,3,4,5,6,7,8,9`

   d. `QUIZ   ASC   'SELF-TEST'`

   e. `ZEROS  EQU  0`

   f. `LIMIT  EQU  '**'`

   g. `ANSWER EQU  *`

   h. `ORG   $0200`

8. Your completed program should look like this:

```
INPUT  EQU $C054
OUTPUT EQU $C0A4
ZERO   EQU 0
       JMP $0600
       ORG $05
COUNTR DFB 10
       ORG $0600
       LDX #ZERO
INSTOR EQU *
       JSR INPUT
       JSR OUTPUT
       STA STORAG,X
       INX
       DEC COUNTR
       BNE INSTOR
STOP   JMP STOP
STORAG DS  10
```

If you missed any of these, review the appropriate frames before going on to the Manual Exercise.

MANUAL EXERCISE

Before continuing, you should find out what the assembler directives are for your assembler. Look them up in your manual under "directives" or "pseudo-operations." If that fails, try looking up EQU and ORG. Many assemblers use those two directives. Use your manual to find the answers to the questions below.

1. Show the format of the directive (or directives) to define initialized space.

   ____

2. Show the format of the directive to define uninitialized space.

   ____

3. Show the format of an equate.

   ____

4. Show the format of the directive to specify an origin.

   ____

5. What other directives are available to you?

   ____

   ____

   ____

   ____

6. How does your system know where your program starts? ______________

   ____

7. What areas of memory are reserved for your operating system and not to be overlaid by your program? ______________

   ____

   ____

CHAPTER SEVEN

CONDITIONAL INSTRUCTIONS

In this chapter we're going to expand on your basic set of instructions. You will learn how to use the conditional instructions, which test the values of the status flags and take appropriate action. (That is, they take action only if a certain *condition*—as indicated by the status flags—is true.) For example, you've learned how to use the JMP instruction to create a closed loop, but you can also tell the system to jump or branch only if the zero flag is on (BEQ) or branch if the carry flag is not on (BCC).

The conditional instructions are used to create open loops and alternate program paths. You'll learn how to code both of these types of program structures.

When you have finished this chapter, you will be able to:

- Code the following instructions:
  - BEQ (*B*ranch if *EQ*ual to zero)
  - BNE (*B*ranch if *N*ot *E*qual to zero)
  - BCC (*B*ranch if *C*arry *C*lear)
  - BCS (*B*ranch if *C*arry *S*et)
  - BMI (*B*ranch if *MI*nus)
  - BPL (*B*ranch if *PL*us)
  - BVC (*B*ranch if o*V*erflow *C*lear)
  - BVS (*B*ranch if o*V*erflow *S*et)
  - CMP (*CoMP*are to accumulator)
  - CPX (*C*om*P*are to register *X*)
  - CPY (*C*om*P*are to register *Y*)
- Create the following types of program structures:
  - open loops
  - alternate paths

REVIEW OF THE FLAGS

The conditional instructions all use the status flags, so let's review those flags before we start on the instructions.

1. Seven of the eight bits of the flag register are used as status flags. The seven status flags are: sign, overflow, break, decimal mode, interrupt disable, zero, and carry. Of these, you'll only use the sign, zero, and carry flags for conditional instructions, so those are the ones we'll concentrate on in this chapter.

(a) How many status flags are there? ______________________

(b) Which ones will you use for condition testing? ______________________

(c) If a flag is one bit, what are its two possible values?

____

— — — — — — — — — —

(a) seven; (b) sign, zero, carry; (c) zero and one

2. The zero flag is turned on or "set" (equal to 1) when a value becomes zero. Otherwise, it is turned off or "cleared" (equal to 0). It is set or cleared as the result of any command that changes the value in a register or that increments or decrements a memory location. (You'll learn later that it is also set as a result of certain other commands, such as "compare.")

The zero flag and the other flags are not affected by branch or jump instructions or by moving data to memory.

Below is a list of instructions you learned in Chapter 5. Select the instructions that affect the zero flag.

____ (a) LDA ____ (d) SBC

____ (b) INC ____ (e) STX

____ (c) ADC ____ (f) JMP

— — — — — — — — — —

(a), (b), (c), (d) (Your reference summary in Appendix C shows the effect of each instruction on the status flags.)

3. Some people get confused by the setting of the zero flag when a result reaches zero. A non-zero result turns the zero flag *off* (0) and a zero result turns it *on* (1).

In the following example, suppose that the instruction SBC #5 has just been executed.

(a) If the accumulator = 0, the zero flag = ______________________

(b) If the accumulator does not = 0, the zero flag = ______________________

— — — — — — — — — —

(a) 1; (b) 0

4. In each of the following examples, show whether the zero flag will be set on (1) or off (0) after the operation.

	zero flag	*accumulator*	*instruction*	*value in zero flag*
(a)	1	00	ADC #10	____
(b)	1	10	STA OUT	____
(c)	0	10	LDX #0	____
(d)	0	02	SBC #2	____
(e)	0	00	SBC #3	____

— — — — — — — — — —

(a) 0; (b) 1; (c) 1; (d) 1; (e) 0 (Notice that the status of the zero flag is not affected by the STA instruction. It remains unchanged.)

5. The carry flag indicates whether an operation caused an overflow; that is, whether the result is too large for the receiving byte. It is set after an addition operation if a one is carried from the most significant bit (the first bit) and may be lost. It is also set if a subtraction operation did *not* need to borrow in order to subtract the most significant bit. Otherwise, it is cleared by any arithmetic operation, except increments and decrements. It's also turned off or on by the CLC and SEC commands. (You will learn later about other operations, such as "compare," that affect the carry flag.)

Indicate the setting of the carry flag after each operation below.

	carry	*receiving byte*	*instruction*	*effect*
(a)	0	A=%11000011	ADC %01100011	____
(b)	0	A=%11000111	LDA %10110011	____
(c)	1	A=%11000111	SBC %00001111	____
(d)	0	A=%10001111	ADC %11000000	____
(e)	1		CLC	____
(f)	0	X=%11111111	INX	____

— — — — — — — — — —

(a) 1; (b) 0; (c) 1; (d) 1; (e) 0; (f) 0

The carry flag tells you whether the result of an arithmetic operation has overflowed the accumulator. It's up to you to code your program to correctly handle

overflow situations and keep the end result accurate. You'll learn how to handle the four basic arithmetic functions—addition, subtraction, multiplication, and division —in Chapter 11.

6. The sign flag reflects the value of the most significant bit (MSB) in the result field. If the MSB is on, the sign flag is set, and if the MSB is off, the sign flag is cleared. Why? Most programmers like to reserve the MSB as a sign bit, limiting the value in a byte to seven bits. If the sign bit is on, the value is negative. If the sign bit is off, the value is positive. The sign flag duplicates the information and can be tested by the conditional instructions.

You'll learn how to handle negative numbers in Chapter 11.

The sign flag is affected by any command that changes the value in a register, or that increments or decrements a memory location.

Indicate the value of the sign flag after each operation below.

	sign flag	*accumulator*	*instruction*	*result on sign flag*
(a)	1	10001000	LDX #0	____
(b)	1	10000000	TAX	____
(c)	0	00001000	ADC %11000000	____

— — — — — — — — — —

(a) 0; (b) 1; (c) 1

You have reviewed the three major flags—zero, carry, and sign—and have seen that they are affected by arithmetic operations. Now let's go on to the instructions that use them. (We'll also briefly introduce the instructions that use the overflow flag.)

CONDITIONAL JUMPS

7. You have already learned how to transfer control to another point in the program using the JMP instruction. JMP is called an *unconditional jump* because the jump always takes place when the instruction is executed.

A conditional jump only happens if the specified condition is true when the instruction is executed. Otherwise, control goes on to the instruction after the conditional jump. (We say that control "falls through" to the next instruction.)

Suppose your program contains this sequence of instructions:

```
MIXER  EQU  *
       JSR  INPUT
       SEC
       SBC  #$20
       BEQ  MIXER
       CLC
       ADC #1
       ...
```

Figure 13 diagrams the logic of the routine. The diamond-shaped box is used to indicate the point at which a question is asked and a yes-no or true-false decision made. In this example, we get a value from the terminal and subtract $20 from it. We then ask the question: Does the result equal zero? (We don't really. We really ask if the zero flag is on. But the *effect* is the same.)

If the answer is yes, control is returned to the statement labeled MIXER and the loop is repeated. If the answer is no (the result is not zero; the zero flag is off), control falls through to the CLC instruction.

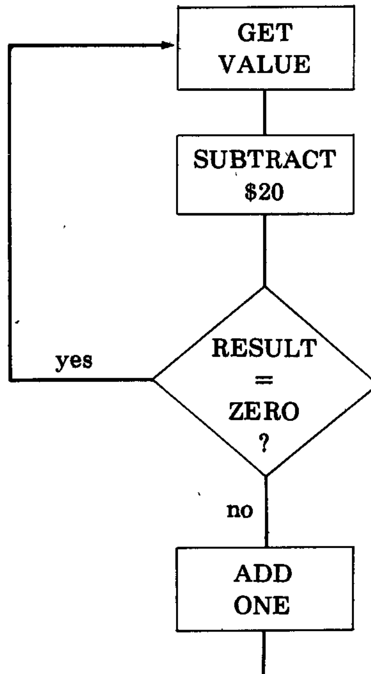


FIGURE 13. Sample Routine

The overall function of the routine is to read characters from the terminal until a non-blank character is obtained. We add one to that character, and what happens after that is not shown.

(a) Is JMP a conditional or unconditional instruction? ____________

(b) Is BEQ a conditional or unconditional instruction? ____________

(c) In the routine diagrammed in Figure 13, what happens if the user types a space?

____

____

(d) What happens if the user types a B? ____________

(e) Is the MIXER loop closed or open? ____________

— — — — — — — — — —

(a) unconditional; (b) conditional; (c) control returns to the beginning of the loop (branch to MIXER); (d) control falls through (one is added to the character); (e) open

8. These are the conditional jump instructions:

BEQ — *B*ranch if *EQ*ual: jump if the zero flag is on
BNE — *B*ranch if *N*ot *E*qual: jump if the zero flag is off
BCS — *B*ranch if *C*arry *S*et: jump if the carry flag is on
BCC — *B*ranch if *C*arry *C*lear: jump if the carry flag is off
BMI — *B*ranch if *MI*nus: jump if the sign flag is on
BPL — *B*ranch if *PL*us: jump if the sign flag is off
BVS — *B*ranch if o*V*erflow *S*et: jump if the overflow flag is on
BVC — *B*ranch if o*V*erflow *C*lear: jump if the overflow flag is off

Write the appropriate jump instructions for the following situations:

(a) Jump to ENDER if the decrement below results in a zero.

```
DEX
```

____

(b) Jump to LOOP if the subtraction below results in a nonzero value.

```
SBC   #10
```

____

(c) Jump to NEGVAL if the input value is negative.

```
JSR   INPUT
```

____

(d) Jump to OK if the subtraction below results in a positive value.

```
SBC   MIND
```

(e) Jump to TOOBIG if the addition below results in a carry.

```
ADC   $10
```

(f) Jump to CYCLE if the addition below does not overflow.

```
ADC   HALF
```

— — — — — — — — — —

(a) BEQ ENDER; (b) BNE LOOP; (c) BMI NEGVAL; (d) BPL OK; (e) BCS TOOBIG; (f) BVC CYCLE

9. Suppose we want to branch to NOT50 if the value in the accumulator is under $50. We start with a subtraction:

```
SEC
SBC #$50
```

Now which flag do we test—carry or sign?

Let's examine the effect on the flags if A is greater than $50. The subtraction results in no borrow, so the carry flag will be *set*. We don't know the effect on the sign flag because we don't know the original value in A. If it's greater than $D0, the MSB will be on and the sign flag will be set. Otherwise, it will be cleared.

If A equals $50, there will be no borrow. and the carry flag will be set. The sign flag will be cleared. (The zero flag will also be set.)

If A is less than $50, there will be a borrow, so the carry flag will be cleared. The sign flag will be set because the high order bit of the remainder will always be a one.

(a) What branch instructions should be used? ____________________

(b) Code a routine to branch to ERROR if the value in the accumulator is less than $10.

(c) Code a routine to branch to TIMOUT if the value in the accumulator is greater than 'A'.

— — — — — — — — — —

(a)
```
BCC NOT50
```

(b)
```
SEC
SBC #$10
BCC ERROR
```

(c)
```
SEC
SBC #'B'
BCS TIMOUT
```

(In this problem, it's necessary to subtract 'B' because the branch will take place if the value was greater than *or equal to* 'B.')

10. All of the branch instructions require relative address operands. When a label is used as an operand in a branch (for example, BEQ LOOPER), the assembler will compare the number of bytes from the current address to the label and use the appropriate one-byte relative address for the operand. This means that the range of a branch must be from −128 to +127 (−$80 to +$7F) bytes, which is usually up to 40 instructions in either direction.

(a) Fill in the table in Appendix C for the branch instructions.

(b) What is wrong with this routine?

```
        ORG  $0150
START   EQU  *
          . . .
          . . .
        ORG  $2150
NEXT    EQU  *
        SBC  #10
        BEQ  START
```

____

____

— — — — — — — — — —

		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a)	BCC	--	--	--	--	--	--	--	--	ok
	BCS	--	--	--	--	--	--	--	--	ok
	BEQ	--	--	--	--	--	--	--	--	ok
	BMI	--	--	--	--	--	--	--	--	ok
	BNE	--	--	--	--	--	--	--	--	ok
	BPL	--	--	--	--	--	--	--	--	ok
	BVC	--	--	--	--	--	--	--	--	ok
	BVS	--	--	--	--	--	--	--	--	ok

(b) The branch goes backward more than 2000 bytes; this is too far.

11. Suppose we want to code a fairly long routine (about 60 lines) that is to keep repeating until a counter in register Y reaches 0. Would this work?

```
      START   EQU  *
              . . .
   -- about 60 instructions --
              . . .
              DEY
              BNE  START
              . . .
```

No, we can't branch that far back. We can, however, use an unconditional jump to move as far as we please. One way to accomplish our purpose is:

```
      START   EQU  *
              . . .
   -- about 60 instructions --
              . . .
              DEY
              BEQ  CONTIN
              JMP  START
      CONTIN  EQU  *
              . . .
```

Here's another example:

```
START EQU  *
      JSR  INPUT        ; READ AND ECHO
      JSR  OUTPUT
      TAX               ; PUT IT IN X
      SEC
      SBC  #$0B         ; CHECK FOR VERTICAL TAB
      BEQ  ENDIT        ; IF SO, QUIT
      TXA
      . . .
-- about 50 instructions --
      . . .
      JMP  START        ; REPEAT LOOP
ENDIT EQU  *
      . . .
```

The BEQ instruction won't work because the range is too long. See if you can fix it.

— — — — — — — — — —

```
START   EQU  *
        JSR  INPUT         ; READ AND ECHO
        JSR  OUTPUT
        TAX                ; PUT IT IN X
        SEC
        SBC  #$0B          ; CHECK FOR VERTICAL TAB
        BNE  CONTIN        ; IF SO, GO ON
        JMP  ENDIT         ; OTHERWISE, QUIT
CONTIN  EQU  *
        TXA
        . . .
-- about 50 instructions --
        . . .
        JMP  START         ; REPEAT LOOP
ENDIT   EQU  *
        . . .
```

12. Figure 14 diagrams the general logic of an open loop. One or more instructions are executed in sequence. Then a question is asked. In a program, that means a condition is tested. If the condition is true, control is returned to the beginning of the loop. If the condition is false, control falls through to the next statement.

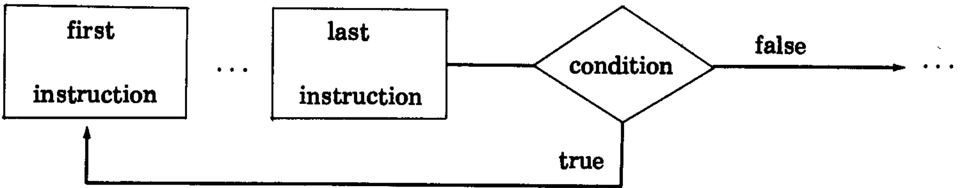


FIGURE 14. Open Loop

Here is an example of an open loop in Assembly Language:

```
MIXER   EQU  *
        JSR  INPUT
        CLC
        ADC  #1
        JSR  OUTPUT
        SEC
        SBC  #'1'
        BNE  MIXER
```

After reading a value from the terminal, we add one to it. Then we write the new value to the terminal and subtract ASCII '1' from it. If the result does not equal binary zero, we loop back to MIXER. If the result does equal binary zero, control falls through to the next instruction. The total effect is to read characters from the terminal until an ASCII zero is found. The character we echo is one higher than the character that was typed.

(a) What's the difference between a closed loop and an open loop?

____

____

(b) What type of jump is used to escape a loop—conditional or unconditional?

____

(c) Code a routine to read and echo characters until the user types a carriage return. Then let control fall through to the next instruction.

— — — — — — — — — —

(a) a closed loop has no natural exit while an open loop does; (b) conditional;

(c)
```
ECHO    EQU  *
        JSR  INPUT
        JSR  OUTPUT
        SEC
        SBC  #$0D
        BNE  ECHO
```

13. We frequently want to execute a loop a specific number of times. Figure 15 shows the logic for executing a loop five times. First we set the X register equal to 5. Then we enter the loop. Each time the loop is executed, we subtract one from the X register. When the X register reaches zero, we know we have executed the loop five times so we allow control to fall through to the next instruction.

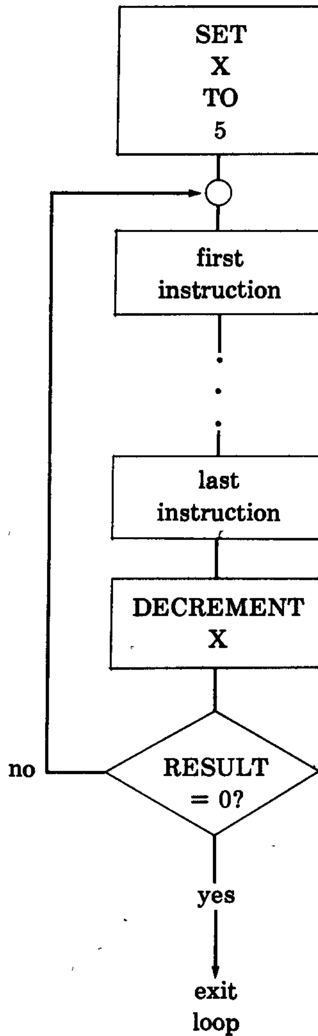


FIGURE 15. Counting Five Loops

We can use the X register to count loops when the loop itself doesn't need to use the X register. Otherwise, we would need to keep a loop counter in another register or a memory location.

The Assembly Language routine would look like this:

```
        LDX   #5
LOOP    EQU   *
        first instruction
        . . .
        last instruction
        DEX
        BNE   LOOP
```

Code a routine that will write the letter 'B' on the terminal three times. Then stop processing.

— — — — — — — — — —

```
        LDX   #3
        LDA   #'B'
BOUT    EQU   *
        JSR   OUTPUT
        DEX
        BNE   BOUT
DONE    JMP   DONE
```

Your routine may not look exactly like ours but it should be close. Did you notice that you need to put 'B' in the accumulator only once? When you are coding loops, don't repeat instructions unnecessarily; they waste time. But be sure your subroutines leave your registers intact.

14. The following routine was supposed to be executed ten times. But the programmer made a very common error and has created a closed loop instead.

```
CLEAR   EQU   *
        LDY   #10
        first instruction
        ...
        last instruction
        DEY
        BNE   CLEAR
        ...
```

What is the error? ______________________________

____

____

____

— — — — — — — — — —

The label, CLEAR, is in the wrong place. It should precede "first instruction," not LDY. The way this loop is written, the Y register is reset to 10 every time the loop is executed and so will never reach zero.

15. The following routine was supposed to be executed three times. But the programmer has made another very common error.

```
        LDX   #3
MIXER   EQU   *
        JSR   INPUT
        CLC
        ADC   #1
        JSR   OUTPUT
        TAX
        STA   MEMORY,X
        DEX
        BNE   MIXER
        ...
```

What is the error? ______________________________

____

____

____

(Extra thought question) How can it be corrected? ______________________________

____

____

____

— — — — — — — — — —

When the TAX instruction is executed, the loop counting value is destroyed. Since this loop uses the X register, we must keep the loop counter somewhere else (in Y for example). A correct routine would be:

```
        LDY   #3             ; USE Y TO COUNT
MIXER   EQU   *
        JSR   INPUT
        CLC
        ADC   #1
        JSR   OUTPUT
        TAX
        STA   MEMORY,X
        DEY                  ; DECREASE COUNT
        BNE   MIXER
```

16. See if you can write a loop that will display the numbers from 0 through 9 on the terminal, then halt.

— — — — — — — — — —

Here's our loop. Yours may be somewhat different. We've numbered the lines so we can discuss them below.

```
10              LDX   #'0'
20       WRINUM EQU   *
30              TXA
40              INX
50              JSR   OUTPUT
60              SEC
70              SBC   #'9'
80              BNE   WRINUM
90       DONE   JMP   DONE
```

Line 10 sets up the initial value in X. Line 30 copies the value into the accumulator; line 50 writes it out. Line 40 increments X so that the next loop will write out the next number. The value in A is tested for '9' in lines 60 and 70. If it's not '9,' we loop back to WRINUM. If it is '9', control falls through to the closed loop.

17. Write a loop that will accept a single digit from the terminal (no echo) and print that number of X's on the terminal, then stop. Assume that the input digit is between '1' and '9.' Don't forget it will be in ASCII.

— — — — — — — — — —

```
10                  JSR   INPUT
20                  SEC
30                  SBC   #$30 ; REMOVES ASCII BITS
40                  TAX
50                  LDA   #'X'
60        LOOP      EQU   *
70                  JSR   OUTPUT
80                  DEX
90                  BNE   LOOP
100       DONE      JMP   DONE
```

Line 10 gets the digit from the terminal and moves it into the accumulator. Lines 20 and 30 convert the value from ASCII code to plain binary. (We say it strips out the most significant bits.) Line 40 moves the value to the X register. Line 50 sets up 'X' in the accumulator. Then we enter the loop, which is controlled by the value in the X register. Line 70 writes an 'X'. Line 80 decrements the X register, turning the zero flag on or off. Line 90 returns control to the top of the loop if the zero flag is off. If the flag is on, control falls through to line 100 and the program repeats line 100 until halted by the operator.

SENDING MESSAGES

18. In this frame, we're going to show you how to write out a message from storage. Suppose we want to write the message "PLEASE TYPE YOUR NAME:" The program is shown in Figure 16.

```
 10               LDY   #22
 20               LDX   #0
 30      OUTLP    EQU   *
 40               LDA   MESSAG,X
 50               JSR   OUTPUT
 60               INX
 70               DEY
 80               BNE   OUTLP
 90      DONE     JMP   DONE
100      MESSAG   ASC   'PLEASE TYPE YOUR NAME:'
```

FIGURE 16. Writing a Message

Line 10 moves decimal 22 into the Y register. There are 22 characters in the message, so we'll write 22 characters. Another way to control the loop would be to check for the last byte in the message, ':'.

Line 20 sets the X register to zero. We will use the X register as an index to address the characters in our message.

Line 40 begins the loop by moving a byte from memory into the accumulator. Line 50 writes the byte. Line 60 increments the X register, so it can now be used to address the next memory byte.

Lines 70 and 80 check for the end of the loop. Line 70 subtracts 1 from the loop counter in Y. Line 80 jumps back to the head of the loop if the counter in Y has not reached zero.

When the loop counter reaches zero, control falls through to the next instruction, a closed loop.

Line 100 defines a data storage area named MESSAG that contains the message we want to print.

Code a routine to write the message 'THANK YOU' on the terminal, then halt.

— — — — — — — — — —

```
        LDY    #9
        LDX    #0
WRITER  EQU    *
        LDA    OUTMSG,X
        JSR    OUTPUT
        INX
        DEY
        BNE    WRITER
DONE    JMP    DONE
OUTMSG  ASC    'THANK YOU'
```

Be careful that control does not fall through to the ASC instruction. The computer might try to execute 'THANK YOU' as an instruction, causing all kinds of strange errors. Be sure you assigned a label to the message in an ASC instruction. You should have used the label in the LDA instruction.

19. Code a routine that reads a message from the terminal. Store the message, but do not echo it. When the user types a carriage return, write the following:

- carriage return and line feed (to start a new line)
- the message
- a question mark

Programming Notes: Be careful the output message does not contain the carriage return typed by the user. Either don't store the CR or overlay it with a question mark.

Don't control the length of the message. But in defining your data area, assume that it will be 80 characters or less.

Write your routine on a separate piece of paper.

— — — — — — — — — —

```
10    ZERO   EQU 0
20    INPUT  EQU $FDOC      ;FOR APPLE
30    OUTPUT EQU $FDED      ;FOR APPLE
40    CR     EQU $0D
50    LF     EQU $0A
60    QMARK  EQU '?'
70           ORG $5000
80           LDX #ZERO
90    READIT EQU *
100          JSR INPUT
110          STA TEXTIN,X
120          INX
130          SEC
140          SBC #CR
150          BNE READIT
160          DEX
170          LDA #QMARK
180          STA TEXTIN,X
190          TXA
200          CLC
210          ADC #3
220          TAY
230          LDX #ZERO
240   WRITIT EQU *
250          LDA ANSWER,X
260          JSR OUTPUT
270          INX
280          DEY
290          BNE WRITIT
300   DONE   JMP DONE
310   ANSWER DFB CR,LF
320   TEXTIN DS  80
```

Notice how we set up the data storage area (lines 310 and 320) so that the address ANSWER refers to ASCII CR, which is followed by ASCII LF, which is followed by the area named TEXTIN where we store the input data.

Register X is used both as an index and to count the number of input characters (lines 80 and 130). We add 3 to it and put the result in register Y to count the number of output loops.

Lines 100–150 are the input loop, reading characters and storing them in memory starting at TEXTIN.

Lines 160–180 replace the final character (ASCII CR) with '?'. Lines 240–290 are the output loop.

COMPARISONS

You've learned to use the conditional instructions and you've seen how open loops can be created. You've also seen that the status flags are affected by arithmetic operations and any change to a register. Now we're going to look at some instructions that set the flags without any arithmetic or changes being performed.

20. Let's look again at the routine that reads and stores bytes from a terminal until a carriage return ($0D) is encountered:

```
         LDX  #0
READIT   EQU  *
         JSR  INPUT
         STA  SAVE,X
         INX
         SEC
         SBC  #$0D
         BNE  READIT
```

Can we change this routine so the carriage return does *not* get stored? If we continue to test by subtraction,

```
         LDX  #0
READIT   EQU  *
         JSR  INPUT
         SEC
         SBC  #$0D
         . . .
```

We changed the byte in A, which means we won't store the correct byte. We will need to save the original byte somewhere else, then make the test and branch out of the loop if the flag is on. If it is off, we can continue on to store the character and return to read again. For example:

```
         LDX  #0
READIT   EQU  *
         JSR  INPUT
         TAY                    ; PRESERVE BYTE IN Y
         SEC
         SBC  #$0D
         BEQ  NOMORE
         TYA                    ; STORE PRESERVED BYTE
         STA  TEXTIN,X
         INX
         JMP  READIT
NOMORE   EQU  *
         . . .
```

Here is another way to accomplish the same thing:

```
        LDX  #0
READIT  EQU  *
        JSR  INPUT
        CMP  #$0D
        BEQ  NOMORE
        STA  TEXTIN,X
        INX
        JMP  READIT
NOMORE  EQU  *
        . . .
```

Notice the CMP instruction. It stands for *CoMPare*.

It compares the value in the accumulator with the byte addressed by the operand and treats the flags accordingly. How does it "compare" them? By pretending to subtract the byte from the accumulator. The flags are set as if the subtraction had taken place. But the value in the accumulator is not altered. (Don't forget that the carry flag is set if there is *no borrow*.)

(a) Write an instruction to compare the accumulator to an ASCII space.

____

(b) Suppose the accumulator contains an ASCII zero. What will be the effect of the above instruction on the zero flag? __________ The sign flag? __________ The carry flag? __________ The accumulator? __________

(c) Suppose the accumulator contains an ASCII space. What will be the effect of the above instruction on the zero flag? __________ The sign flag? __________ The carry flag? __________ The accumulator? __________

(d) Suppose you want to jump to PUTNEX if the value in the accumulator is an ASCII space. Otherwise, control should fall through. Write the branch instruction that follows the compare. __________

— — — — — — — — — —

(a) CMP #' '; (b) cleared, cleared, set (no borrow), no change; (c) set, cleared, set, no change; (d) BEQ PUTNEX

21. Here are some more problems using CMP.

(a) Write a set of instructions to compare the accumulator to \$50. If it does not equal \$50, jump to NEXONE. If it does equal \$50, let control fall through.

____

(b) Write a set of instructions to compare the accumulator to ASCII A. If it is equal to or greater than A, jump to LETTER. If it's less than A, let control fall through.

(a)
```
CMP   #$50
BNE   NEXONE
```

(b)
```
CMP   #'A'
BCS   LETTER
```

(Any value smaller than 'A' will cause a borrow, thus clearing the carry flag. If the value is 'A' or larger, the carry flag will be set.)

22. You have learned to code the CMP instruction. There are also CPX (Com*P*are to *X* register) and CPY (Com*P*are to *Y* register) instructions. They compare the value in a register with the value of the byte addressed by the operand.

(a) Write an instruction to compare the X register to ASCII A.

____

(b) Write an instruction to compare the Y register to 7. ________________

(c) Write a set of instructions to compare the X register with the Y register.

(d) Write a set of instructions to compare the accumulator to the X register.

(e) Write a set of instructions to read a value from the terminal. If it is equal to the value in the X register, jump to SAMVAL. Otherwise, let control fall through.

(a)
```
CPX   #'A'
```

(b)
```
CPY   #7
```

(c) Here are two ways to solve the problem:

```
STX   SAVE                    STY   SAVE
CPY   SAVE                    CPX   SAVE
```

They have different effects. In the first, the value in X is "subtracted" from the value in Y. In the second, the value in Y is "subtracted" from the value in X.

(d)
```
STA   SAVE
CPX   SAVE
```

(e)
```
JSR   INPUT
STA   SAVE
CPX   SAVE
BEQ   SAMVAL
```

ALTERNATE PATHS

23. You've learned how to code loops. Another extremely important program structure is illustrated by Figure 17. We call this alternate paths although there are many other names for the structure.

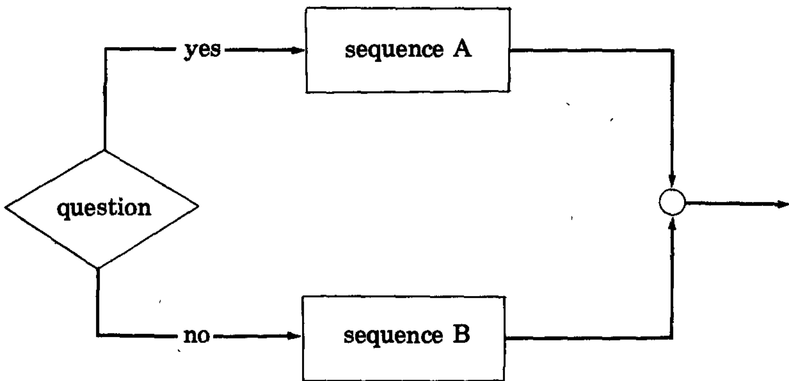


FIGURE 17. Alternate Paths

In this structure, a yes-no question is asked (or a true-false condition tested). If the answer is yes, one path is taken. If the answer is no, the other path is taken.

For example, suppose we want to read and edit the user's input. If the user types a digit between 0 and 9, we store the digit. If the user types any other character, we write an error message.

(a) In our example, what is the yes-no question? ______________________

____

(b) What is the "yes" path? ______________________

(c) What is the "no" path? ______________________

— — — — — — — — — —

There are two ways to answer these questions:

(a) Is the input value between 0 and 9?; (b) store the digit; (c) write error message

or

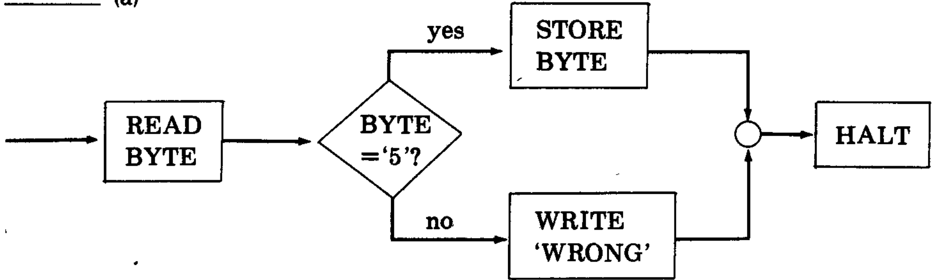
(a) Is the input value outside of the range of 0–9? (b) write error message; (c) store the digit

24. In Assembly Language, alternate paths are coded using comparisons and conditional jumps. Here's an example:

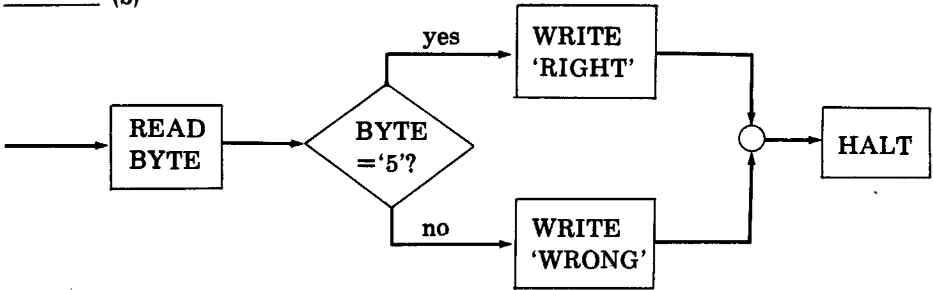
```
 10          JSR  INPUT
 20          CMP  #5
 30          BNE  NOTFIV
 40   FIVE   EQU  *
 50          LDX  #0
 60          JMP  OUTMSG
 70   NOTFIV EQU  *
 80          LDX  #5
 90   OUTMSG EQU  *
100          LDY  #5
110   OUTLOP EQU  *
120          LDA  FIVMSG,X
130          JSR  OUTPUT
140          INX
150          DEY
160          BNE  OUTLOP
170   DONE   JMP  DONE
180   FIVMSG ASC  'RIGHT'
190   NOTMSG ASC  'WRONG'
```

Which of the following diagrams correctly depicts what this routine does?

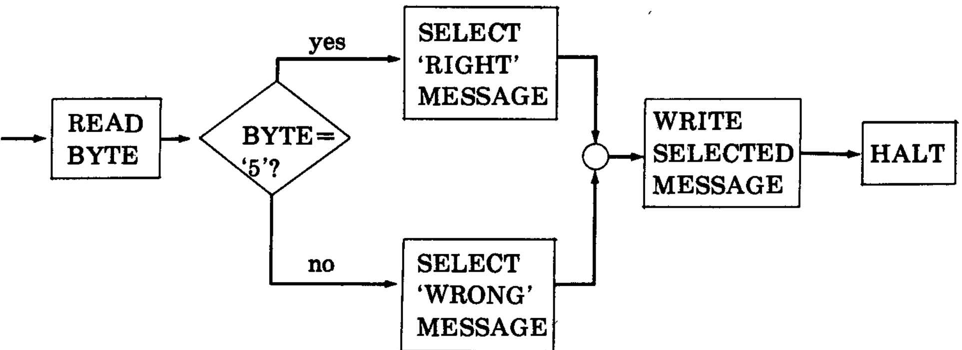
_______ (a)



_______ (b)



_______ (c)



\- - - - - - - - - -

(c) is the most correct answer; (b) is close but not completely right.

25. Now it's your turn. Write a routine to read and echo two characters from the terminal. If they're the same, write 'SAME'. If they're not the same, write 'DIFF'.
Use a separate sheet of paper.

— — — — — — — — — —

```
        LDY   #4          ; LOAD Y WITH LENGTH OF OUTPUT MESSAGE
        JSR   INPUT       ; READ, ECHO, STORE FIRST BYTE
        JSR   OUTPUT
        STA   SAVE
        JSR   INPUT       ; READ, ECHO SECOND BYTE
        JSR   OUTPUT
        CMP   SAVE        ; COMPARE TWO BYTES
        BNE   NOTSAM      ; IF EQUAL, NEXT LINE, OTHERWISE NOTSAM
 SAM    EQU   *
        LDX   #0          ; SET INDEX FOR 'SAME' MESSAGE
        JMP   MESSAG
NOTSAM  EQU   *
        LDX   #4         ; SET INDEX FOR 'DIFFERENT' MESSAGE
MESSAG  EQU   *
        LDA   SAME,X     ; WRITE OUT A LETTER OF MESSAGE
        JSR   OUTPUT
        INX              ; INCREASE INDEX
        DEY              ; DECREASE COUNTER
        BNE   MESSAG     ; IF NOT ALL WRITTEN, REPEAT LOOP
DONE    JMP   DONE
SAME    ASC   'SAME'
DIFF    ASC   'DIFF'
SAVE    DS    1
```

26. Often an alternate path structure has an empty "yes" or "no" path. Figure 18 depicts the logic diagrams of such structures. Some of the routines you worked earlier in this chapter had this structure.

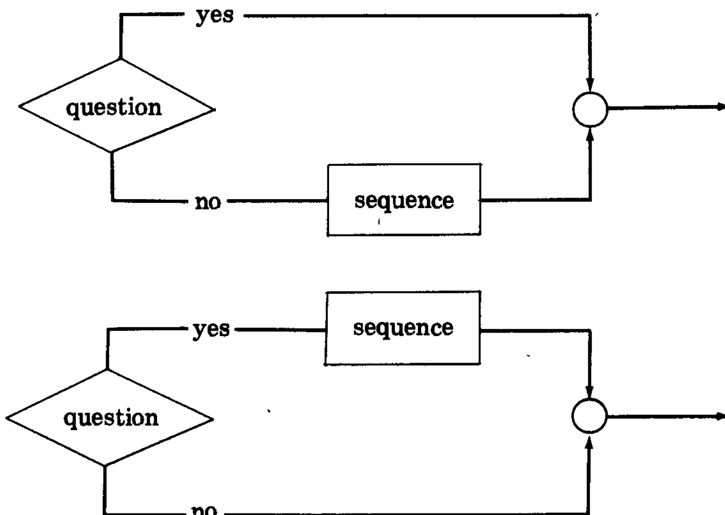


FIGURE 18. Empty Path Structures

Suppose we want to read a byte and, if it's not a space, store it in memory and increment the X register. If it is a space, do nothing. Here's the code:

```
        JSR   INPUT
        CMP   #' '
        BEQ   NEXTEP
NOTSPA  EQU   *
        STA   SAVE,X
        INX
NEXTEP  EQU   *
```

Whether it's the "yes" path or the "no" path that's empty is immaterial. The empty path jumps around the not-empty path, to the point where they rejoin.

Code a routine that will read and echo a byte. If it's a carriage return, also write a line feed. Then store the byte in memory.

- - - - - - - - - -

```
        JSR   INPUT
        JSR   OUTPUT
        TAY               ; COPY CHARACTER TO Y
        CMP   #$0D        ; COMPARE TO CR
        BNE   STORIT
        LDA   #$0A        ; WRITE LINE FEED
        JSR   OUTPUT
STORIT  EQU   *
        STY   SAVE        ; STORE CHARACTER
```

REVIEW

In this chapter, you have learned how to use the conditional instructions to handle loops and alternate path structures.

- The conditional instructions are based on the flags.
- The conditional branch instructions are:
  - BEQ (*B*ranch if *EQ*ual to zero)
  - BNE (*B*ranch if *N*ot *E*qual to zero)
  - BCC (*B*ranch if *C*arry *C*lear)
  - BCS (*B*ranch if *C*arry *S*et)
  - BMI (*B*ranch if *MI*nus)
  - BPL (*B*ranch if *PL*us)
  - BVC (*B*ranch if o*V*erflow *C*lear)
  - BVS (*B*ranch if o*V*erflow *S*et)
- The comparison instructions cause the status flags to be set without altering the value in the register. They are:
  - CMP (*C*o*MP*are to accumulator)
  - CPX (Com*P*are to register *X*)
  - CPY (Com*P*are to register *Y*)
- An open loop is usually coded with a conditional jump. If the condition proves false, control falls through to the next instruction.
- In a counted loop, the count value is placed in an index register or memory byte. At the end of each loop, the loop counter is decremented. When it reaches zero, control falls out of the loop.
- An alternate path structure asks a yes-no question. One path is taken if the answer is yes and another is taken if the answer is no. Either path may be empty. The structure is coded using conditional jumps. In Assembly Language, the structure looks like this:

```
        BNE  NOPATH
YESPTH  EQU  *
        . . .
        JMP  REJOIN
NOPATH  EQU  *
        . . .
```

If there is an empty path, the structure looks like this:

```
        BNE  REJOIN
PATH    EQU  *
        . . .
REJOIN  EQU  *
```

CHAPTER 7 SELF-TEST

Part I. Code instructions to solve the following problems.

1. Jump to START if the zero flag is off.

   ____

2. Jump to CARRY if the carry flag is on.

   ____

3. Jump to NEGIVE if the sign flag is on.

   ____

4. If the accumulator equals an ASCII space, jump to SPACE. Otherwise, jump to NOTSPA.

   ____

   ____

   ____

5. If the X register is greater than 10, jump to MORE.

   ____

   ____

   ____

Part II. In this exercise you will code a data compression program. Data compression is used when you have a lot of data to store and you want to conserve some space. It works by removing repeated characters from the data. For example (ƀ indicates a space):

```
JOHNƀJONESƀ[ƀƀƀƀ]21201ƀ[ƀƀƀ]180[000000]
```

The boxed characters would be removed. We have to tell the system that some characters have been removed. We do this by storing a warning flag followed by the

count of the number of characters that were removed. So the above data would be stored this way ($FF is the warning flag):

* * NOTE * *

'J'	'O'	'H'	'N'	' '	'J'	'O'	'N'	'E'	'S'	' '	FF	04	'2'	'1'	etc.

The three bytes—' 'FF04—tell the computer that four spaces were removed.

Your job is to write a program that will read a string of characters from the terminal and store them in compressed format. End the program when the user types a carriage return.

Figure 19 shows our program logic.

Programming Notes: Even though it's not completely efficient, compress any repeated character even if it's repeated only once.

Assume that the input string is less than 80 characters.

Strategy: In order to identify repeated characters, each character we read must be compared with the preceding character. We'll use the X register as an index for storage. We won't increment X until ready to store a character; so until then, it will be pointing to the last character stored. When we find repeated characters, we must count them. We'll keep the count in register Y.

1. Initialization.
   a. Store a byte at the beginning of the text string that can't be matched, such as $FF. This will force the first comparison to fail and the first input character to be stored.
   b. Set registers X and Y to zero.
2. Read and echo one character. Compare it to the previous byte stored.
3. If the new byte matches the previous byte, go to step 4. If it doesn't, check the count in the Y register.
   a. If the count is greater than zero, do the following:
      (1) temporarily save the new character (because you'll need the A register)
      (2) load $FF into the accumulator
      (3) increment X, and write $FF to TEXT+X
      (4) increment X, and write the count from the Y register to TEXT+X
      (5) clear the Y register
      (6) restore the new character to the accumulator
      (7) go on to b
   b. Whether or not the count was greater than zero, do the following:
      (1) increment X and store the new character
      (2) compare the new character to carriage return
      (3) quit if it is a carriage return; otherwise return to step 2
4. When the new byte matches the previous byte, all you have to do is increment the count in the Y register and return to step 2.

FIGURE 19. Compression Program Logic

Self-Test Answer Key

Part I.

1.
```
BNE   START
```

2.
```
BCS   CARRY
```

3.
```
BMI   NEGIVE
```

4.
```
CMP   #' '
BEQ   SPACE
JMP   NOTSPA   or BNE   NOTSPA
```

5.
```
CPX   #11
BPL   MORE
```

Part II.

```
         LDA   #$FF            ; INITIALIZE FIRST BYTE SO
         STA   TEXT            ;     COMPARISON WON'T MATCH
; NOTE THAT STRING WILL BE STORED STARTING AT TEXT+1
         LDX   #0              ; SET UP INDEX
         LDY   #0              ; INITIALIZE COUNT
RLOOP    EQU   *
         JSR   INPUT
         JSR   OUTPUT
         CMP   TEXT,X          ; COMPARE TO PREVIOUS BYTE
         BEQ   REPEAT
NEWONE   EQU   *               ; CHARACTERS WON'T MATCH
         CPY   #0              ; DO WE NEED TO STORE FLAG AND COUNT?
         BEQ   STORIT
         STA   SAVEIT          ; TEMPORARILY SAVE CHARACTER
         LDA   #$FF            ; STORE FLAG
         INX
         STA   TEXT,X
         INX                   ; STORE COUNT
         TYA
         STA   TEXT,X          ; CAN'T USE INDEX WITH STY
         LDA   SAVEIT          ; RESTORE CHARACTER TO A
         LDY   #0              ; REINITIALIZE Y
STORIT   EQU   *
         INX
         STA   TEXT,X
         CMP   CR              ; CHECK FOR CR
         BEQ   ENDIT
         JMP   RLOOP
REPEAT   EQU   *               ; IF THE INPUT CHARACTER
         INY                   ;     MATCHES THE LAST
         JMP   RLOOP           ;         CHARACTER
ENDIT    JMP   ENDIT
CR       DFB   $0D
TEXT     DS    81
SAVEIT   DS    1
```

CHAPTER EIGHT

LOGICAL OPERATIONS

So far, you have learned to code instructions for data movement, arithmetic operations, comparisons, and jumps. In this chapter, you'll learn a set of instructions that are used for logical operations. These include the logical operations of AND, OR, and EXCLUSIVE OR, which will be defined, as well as bit rotation.

When you have finished this chapter, you'll be able to:

- code the following instructions:
  - AND (AND with accumulator)
  - ORA (OR with accumulator)
  - EOR (EXCLUSIVE OR with accumulator)
  - BIT (bit test)
  - ASL (arithmetic shift left)
  - LSR (logical shift right)
  - ROL (rotate left)
  - ROR (rotate right);
- solve the following types of problems:
  - turn specified bits on or off in a value
  - test specified bits against a mask
  - clear the accumulator using a logical operation
  - test the least significant or most significant bit of a value
  - shift a value left or right.

THE AND AND OR OPERATIONS

1. The logical operations, AND and OR, compare two bits and set a result bit to show the result of the comparison.

The AND operation says that if both bit A and bit B are on, turn the result bit on. Otherwise, turn it off.

If we use the $\wedge$ symbol to represent the AND operation, we can write the four AND facts this way:

```
  0        0        1        1
 ∧0       ∧1       ∧0       ∧1
 --       --       --       --
  0        0        0        1
```

Notice that the result bit is on (1) only if both of the ANDed bits are on.

(a) If bit A is on and bit B is off, what is the result of A Λ B? ________________

(b) If both bit A and B are off, what is the result of A Λ B? ________________

(c) If bit A and B are both on, what is the result of A Λ B? ________________

— — — — — — — — — —

(a) 0; (b) 0; (c) 1

2. To AND multiple-bit values, do it one column at a time. Each column is independent. There are no carries or borrows to worry about.

```
  1011001
 Λ0110101
  -------
  0010001
```

Show the results of the following AND operations.

```
(a)  11010001          (b)  00001111
    Λ10101000              Λ01010101
     --------               --------
```

— — — — — — — — — —

(a) 10000000; (b) 00000101

3. The OR operation turns on the result bit if either A or B or both are on. If we use V to represent the OR operation, the OR facts are:

```
   0       0       1       1
 V 0     V 1     V 0     V 1
   -       -       -       -
   0       1       1       1
```

Notice here that the result bit is off (0) only if both the ORed bits are off.

Show the results of the following operations.

```
(a)   10101111         (b)   01100110
    V 01000110             V 11010100
      --------               --------
```

— — — — — — — — — —

(a) 11101111; (b) 11110110

4. The EXCLUSIVE OR operation is similar to OR, but if both bits are on, the result bit is turned off. We use the symbol ⊻ for EXCLUSIVE OR. Here are the EXCLUSIVE OR facts:

```
  0      0      1      1
⊻ 0    ⊻ 1    ⊻ 0    ⊻ 1
  0      1      1      0
```

The last fact, 1 ⊻ 1, is what makes EXCLUSIVE OR exclusive. If either of the ORed bits is on, the result bit is on. If both are on, the result is off.

Show the results of the following operations.

```
(a)  10110101          (b)  10001101
    ⊻00001111              ⊻01100110
```

(a) 10111010; (b) 11101011

5. To summarize the logical facts, complete the three tables below.

∧	0	1
0		
1		

AND

∨	0	1
0		
1		

OR

⊻	0	1
0		
1		

EXCLUSIVE OR

∧	0	1
0	0	0
1	0	1

AND

∨	0	1
0	0	1
1	1	1

OR

⊻	0	1
0	0	1
1	1	0

EXCLUSIVE OR

THE AND INSTRUCTION

6. In Assembly Language, we use the AND instruction to accomplish the AND function. Valid addressing modes for AND are the same as the ones for LDA and ADC.

AND causes the addressed byte to be logically ANDed with the accumulator. The accumulator is changed to reflect the result of the operation. The sign and zero flags are also affected.

For example, the instruction AND INMASK will cause the contents of register A to be logically ANDed with the value at address INMASK. The result will be placed in the accumulator.

(a) Fill out Appendix C for AND.

(b) Code an instruction to AND the immediate value 1 to the accumulator.

___

(c) Which flags will be affected when the above instruction is executed?

___

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) AND	ok	ok	ok	XYok	Xok	–	ok	ok	–

(b) AND #1

(c) sign and zero

7. Suppose the accumulator contains %10100001 and the byte at address $0516 contains %11110000. What effect will AND $0516 have:

(a) on the accumulator? ___

(b) on the byte at $0516? ___

(c) on the sign flag? ___

(d) on the carry flag? ___

(e) on the zero flag? ___

— — — — — — — — — —

(a) set to %10100000; (b) no effect; (c) set; (d) no effect; (e) cleared

8. Can you write an AND instruction to *clear* the accumulator regardless of its current value? (That is, set the accumulator to zero.)

____

— — — — — — — — — — —

AND #0 will do it

9. Now that you can code an AND operation, let's talk about how we might use it. AND operations are usually used when we want to turn off specific bits in a value.

Examine the instruction AND #%00001111. This instruction would force the highest four bits in the A register to be turned off, regardless of what's currently in there. The lowest four bits retain their present value. Let's see why.

```
  00001111
∧ XXXXXXXX
  --------
  0000XXXX
```

Without knowing the value of X, we know that $0 \wedge X = 0$. If $X = 0$, $0 \wedge 0 = 0$. If $X = 1$, $0 \wedge 1 = 0$.

On the other hand, we know that $1 \wedge X = X$. If $X = 0$, $1 \wedge 0 = 0$. If $X = 1$, $1 \wedge 1 = 1$.

(a) AND operations are used to turn bits (on/off) _______________

(b) $0 \wedge X =$ __________

(c) $1 \wedge X =$ __________

(d) What operand would you use in an AND instruction to turn off the least significant bit in the A register and leave the rest alone?

____

— — — — — — — — — — —

(a) off; (b) 0; (c) X; (d) #%11111110

10. In the instruction AND #%00001111, the operand is called a *mask* because it is a pattern that blocks out (or affects) some bits but allows others through (or leaves them alone).

(a) In an AND mask, what value will turn off the corresponding bit in the accumulator? ______

(b) What value will leave the corresponding bit in the accumulator alone? ______

— — — — — — — — — — —

(a) 0; (b) 1

11. Code AND instructions to solve these problems. Masks are usually expressed in binary so you can see which bits are on and which are off. You can use equivalent hex or decimal values if you wish.

(a) Turn off the most significant bit in the accumulator. Leave the other bits alone. ______________________

(b) Turn off the third and fourth bits from the left. Leave the other bits alone.

____

— — — — — — — — — —

(a) AND #%01111111; (b) AND #%11001111

12. Here are some practical problems that involve turning bits off.

(a) Code a routine that reads an ASCII character from the terminal, strips out (or turns off) the ASCII zone bits (the high order four bits), and stores the result at INBYTE. (This means that '1,' 'A,' and 'a' would all be stored as $01, whereas if stored normally, they'd be $31, $41, and $61.) This is part of the process of converting ASCII to binary code.

(b) Code a routine that reads an input digit from the terminal. If the digit is even, jump to INEVEN. If the digit is odd, jump to INODD.

— — — — — — — — — —

(a)

```
JSR  INPUT
AND  #%00001111 ; CLEAR HIGH ORDER FOUR BITS
STA  INBYTE
```

(b)

```
JSR  INPUT
AND  #%00000001 ; CLEAR ALL BUT LOW-ORDER
BEQ  INEVEN     ; IF ZERO, ORIGINAL BYTE WAS EVEN
JMP  INODD      ; OTHERWISE, IT WAS ODD
```

THE OR INSTRUCTION

13. Now let's consider the ORA (*OR* with *A*) instruction. It is just like AND with respect to addressing modes and flags.

The ORA instruction causes the addressed byte to be logically ORed with the accumulator, changing the accumulator to reflect the result.

(a) Fill out Appendix C for ORA.

(b) Code an instruction to OR the value 1 to the accumulator.

____

(c) Code an instruction to OR the value at byte 5 on page zero to the accumulator.

____

(d) Which flags will be affected when the above instruction is executed?

____

— — — — — — — — — —

		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a)	ORA	ok	ok	ok	XYok	Xok	--	ok	ok	--

(b) ORA #%00000001; (c) ORA $05; (d) sign and zero

14. OR operations are used to turn bits on. A one in the mask will force the corresponding bit on, since 1 V X = 1. A zero in the mask will leave the corresponding bit alone since 0 V X = X.

(a) If X = 1, 1 V X = _______.

(b) If X = 0, 1 V X = _______.

(c) Therefore, 1 V X = _______.

(d) If X = 1, 0 V X = _______.

(e) If X = 0, 0 V X = _______.

(f) Therefore, 0 V X = _______.

(g) OR operations are used to turn bits (on/off) _______.

— — — — — — — — — —

(a) 1; (b) 1; (c) 1; (d) 1; (e) 0; (f) X; (g) on

15. Code OR operations for the following problems.

(a) Turn all the bits in the accumulator on. ____________________

(b) Don't change the value in the accumulator but set the sign and zero flags.

____

(c) Turn on the high order bit in the accumulator. Leave the other bits alone.

____

— — — — — — — — — —

(a) ORA #%11111111; (b) ORA #0; (c) ORA #%10000000

THE EOR INSTRUCTION

16. The EXCLUSIVE OR instruction is EOR. Its operands are the same as AND and ORA, and it sets the same flags.

(a) Code an instruction to EXCLUSIVE OR the byte at address $25 (zero page) with A.

____

(b) Code an instruction to EXCLUSIVE OR %10001000 with A.

____

(c) Which flags will be affected by the above instruction?

____

— — — — — — — — — —

(a) EOR $25; (b) EOR #%10001000; (c) sign and zero (Be sure to fill out appendix C for EOR.)

17. EXCLUSIVE ORs are usually used to *complement* bits. A bit is complemented when its value is reversed; a one becomes a zero and a zero becomes a one.

To complement a bit, the EXCLUSIVE OR mask should contain a one in the corresponding bit. To leave a bit alone, the mask bit should contain a zero.

(a) $1 \veebar 0 =$ ________

(b) $1 \veebar 1 =$ ________

(c) Therefore, $1 \veebar X =$ ________

(d) $0 \veebar 0 =$ ________

(e) $0 \veebar 1 =$ ________

(f) Therefore, $0 \veebar X =$ ________

(g) Code an instruction to complement (reverse) the least significant bit of the accumulator.

____

(h) Code an instruction to complement the entire accumulator.

____

— — — — — — — — — —

(a) 1; (b) 0; (c) the opposite of X; (d) 0; (e) 1; (f) X; (g) EOR #%00000001;
(h) EOR #%11111111

18. Summary:

AND — Mask value of 0 in a bit turns off the corresponding bit in the accumulator. Mask value of 1 leaves the corresponding bit unchanged.

OR — Mask value of 1 forces the corrsponding bit on. A 0 leaves the corresponding bit alone.

EOR — Used to complement bits. A 1 complements the corresponding bit. A 0 leaves it alone.

You have now seen how to use the various AND and OR operations. To practice them, code instructions for the following.

(a) Turn off the high order bit. ______________________

(b) Turn on the high order bit. ______________________

(c) Complement the high order bit. ______________________

(d) Zero the accumulator. ______________________

(e) Set the accumulator to all ones. ______________________

(f) Convert a single digit in the accumulator between 0 and 9 to its ASCII code. Currently, the digit is in this form: %0000XXXX. You want to change it to this form: %0011XXXX.

____

(g) Convert a lower case ASCII letter in the accumulator to its upper case form. Currently, the value is in this form: %011XXXXX. You want to change it to this form: %010XXXXX.

____

— — — — — — — — — —

(a) AND #%01111111; (b) ORA #%10000000; (c) EOR #%10000000; (d) AND #0;
(e) ORA #%11111111; (f) ORA #%00110000; (g) AND #%11011111

THE BIT INSTRUCTION

19. The BIT instruction is similar to the AND instruction except that it does not change the value in the accumulator. You use it to test the value in the addressed memory byte; the settings of the flags show the result of the test.

The flags that are affected are the sign, zero, and overflow flags. They are not set in the normal manner however. The zero flag is set or cleared according to the result of the AND operation. But the sign flag is set or cleared according to the high order bit of the *memory byte*. The overflow flag is set or cleared according to the value of the next lower bit in the memory byte.

The only two addressing modes allowed are direct and zero page direct.

(a) BIT tests the value in (the accumulator/memory) ______

(b) BIT is exactly like AND except that the accumulator isn't changed. True or false? ______

(c) After a BIT test, what does the overflow flag show? ______

______

(d) Fill out Appendix C for BIT.

— — — — — — — — — —

(a) memory; (b) false [the flags are also different, as are the addressing modes]; (c) the value of the next-to-highest order bit of the memory byte;

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(d) BIT	--	ok	ok	--	--	--	--	--	--

20. Suppose the accumulator contains %10000011 and byte $1520 contains %11110000. The result of BIT $1520 would be %10000000 (which would not be stored anywhere). The zero flag would be cleared because the result is not zero. The sign flag and the overflow flag would both be set because the first two bits at $1520 are on (regardless of the AND operation).

Suppose the accumulator contains %00000001 and the byte at $0023 contains %10000000. Show the effect of BIT $23:

(a) on the zero flag: ______

(b) on the sign flag: ______

(c) on the overflow flag: ______

— — — — — — — — — —

(a) turned on (set to 1); (b) turned on; (c) turned off

21. Suppose you read a byte into INBYTE and you want to check whether it's even or odd without moving or destroying the byte. If it's even you want to jump to EVEN. Complete the routine below.

```
JSR   INPUT
JSR   OUTPUT
STA   INBYTE
LDA   #%00000001
BIT   INBYTE
____  EVEN
```

— — — — — — — — — —

BEQ (If the input byte was even, the result of the AND operation will be zero; otherwise it will be one.)

22. You want to test the value of a status byte at STATUS. If both high order bits are on, let control fall through. Otherwise, jump to NEWTRY. (This is a very common pattern in an I/O routine.)

— — — — — — — — — —

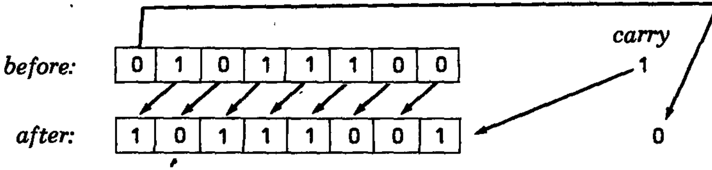
Here's our routine using BIT. The value in the accumulator doesn't matter since we're not going to test the zero flag.

```
BIT   STATUS
BPL   NEWTRY ; BRANCH IF MSB ≠ 1 (SIGN FLAG OFF)
BVC   NEWTRY ; BRANCH IF 2ND BIT ≠ 1
```

REGISTER ROTATION

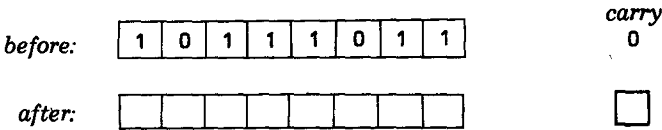
Assembly Language includes a set of instructions to rotate the value of a byte. The following frames discuss register rotation.

23. A value is rotated when all the bits, including the carry flag, are moved over one. A simple rotation to the left looks like this:

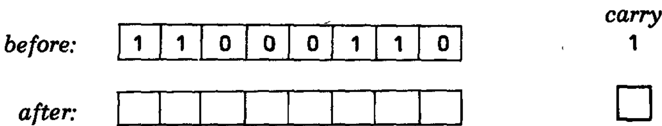


Notice that the most significant bit wraps all the way around to the carry flag. The carry flag goes into the least significant bit.

(a) Show the results of a simple rotate to the right.



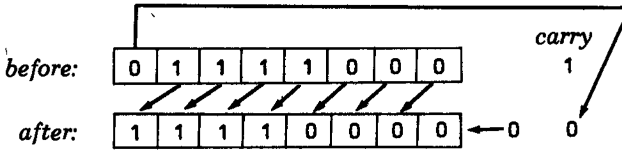
(b) Show the results of a simple rotate to the left.



— — — — — — — — — —

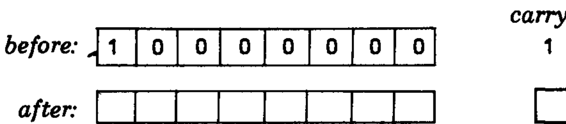
(a) 01011101, carry 1; (b) 10001101, carry 1

24. Another form of rotation is called a shift. Here is a shift left.

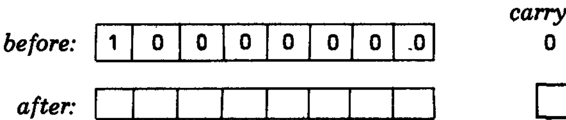


The most significant bit is shifted into the carry flag; the former value of the carry flag is lost. A zero is shifted into the least significant bit. In a right shift, the carry is replaced by the least significant bit and a zero is shifted into the most significant bit.

(a) Show the result of a shift right.



(b) Show the result of a shift left.



— — — — — — — — — —

(a) 01000000, carry 0; (b) 00000000, carry 1

25. The four rotate/shift instructions are: ROL (*RO*tate *L*eft), ROR (*RO*tate *R*ight), ASL (*A*rithmetic *S*hift *L*eft), and LSR (*L*ogical *S*hift *R*ight). They each permit the same addressing modes:

- direct
- zero-page direct
- indexed direct (X only)
- zero-page indexed direct (X only)

You can also reference register A as an operand, so that you can rotate and shift the accumulator.

(a) Fill out Appendix C for ROL, ROR, ASL, and LSR.

(b) Code an instruction to shift the value in the byte named NUMBER to the left.

____

(c) Code an instruction to rotate the byte at address $05 (zero page) to the right.

____

(d) When the above instruction is executed, suppose a one is rotated out of the low-order bit. Where does it go to? (Choose one.)

_____ the byte at $06

_____ the carry flag

_____ the overflow flag

_____ the high-order bit of $05

_____ it gets lost

(e) Code an instruction to rotate the accumulator to the right.

____

— — — — — — — — — —

(a)	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
ASL	——	ok	ok	Xok	Xok	——	——	——	——
LSR	——	ok	ok	Xok	Xok	——	——	——	——
ROL	——	ok	ok	Xok	Xok	——	——	——	——
ROR	——	ok	ok	Xok	Xok	——	——	——	——

(b) ASL NUMBER; (c) ROR $05; (d) the carry flag; (e) ROR A

26. All four rotate/shift instructions affect the sign and zero flags, in addition to the carry flag.

Suppose the byte at $05 contains %01000001. What will be the affect of ROL $05:

(a) on the sign flag? ______________________________

(b) on the zero flag? ______________________________

(c) on the carry flag? ______________________________

— — — — — — — — — —

(a) set; (b) cleared; (c) cleared (a zero is carried around)

27. One of the four rotate/shift instructions will always cause the sign flag to be cleared. Which one, and why? ______________________________

____

____

— — — — — — — — — —

LSR, because a zero is always shifted into the high-order bit by this instruction.

28. Use rotate/shift instructions to solve each of the following problems.

(a) Read a byte. If it's even, jump to INEVEN. If it's odd, jump to INODD.

(b) We *normalize* a value by shifting it left until the first bit is one. Write a routine that will normalize the value in the accumulator. Keep track of the number of shifts in SHIFTS. (Assume that register A contains a non-zero value. Don't forget to clear SHIFTS before you start.)

— — — — — — — — — —

(a)

```
JSR  INPUT
JSR  OUTPUT
ROR  A          ; ROTATE LOW BIT INTO CARRY
BCC  INEVEN     ; CARRY = 0 IF BYTE WAS EVEN
JMP  INODD      ; OTHERWISE IT'S ODD
```

(LSR would do as well as ROR; in both cases the carry flag is turned on if the least significant bit was one and turned off if it was zero.)

(b)

```
        LDX  #0       ; CLEAR X
        STX  SHIFTS   ; CLEAR SHIFTS
ROUND   EQU  *
        INC  SHIFTS
        ASL  A
        BPL  ROUND
```

REVIEW

In this chapter, you have learned several instructions that can be used to manipulate individual bits. They are called the logical instructions.

- The AND operation has these results:

```
  0     0     1     1
^_0   ^_1   ^_0   ^_1
  0     0     0     1
```

  The AND instruction may use the same addressing modes as LDA. It operates on the accumulator, which is changed to show the result. The sign and zero flag are affected.

- We usually use AND to turn off individual bits. A zero in an AND mask forces the corresponding bit to be turned off. A one leaves the corresponding bit alone.

- The OR operation has these results:

```
  0     0     1     1
v_0   v_1   v_0   v_1
  0     1     1     1
```

  The OR instruction may use the same addressing modes as AND. It operates on the accumulator, which is changed to show the result. The sign and zero flags are affected.

- We usually use OR to turn on bits. A one in the OR mask will turn the corresponding bit on. A zero will leave the corresponding bit alone.

- The EXCLUSIVE OR operation has these results:

```
   0      0      1      1
  ⊻0     ⊻1     ⊻0     ⊻1
  --     --     --     --
   0      1      1      0
```

  The EOR instruction may use the same addressing modes as the AND instruction. It affects the flags in the same way.

- We usually use EXCLUSIVE OR instructions to complement bits. A one in the mask causes the corresponding bit to be complemented. A zero leaves the corresponding bit alone.

- Bits can be tested without changing them using the BIT instruction. It can use either direct or zero-page direct addressing. It ANDs the memory byte and the accumulator without changing either. The zero flag is set or cleared according to the result. The sign flag is set or cleared according to the value of the most significant bit of the memory byte and the overflow flag is set or cleared according to the next bit.

- Bit rotation can be used for a variety of functions. Bits can be tested by rotating them into the carry flag position.

- Bits can be rotated left or right one position at a time. If rotating through the carry flag, the flag becomes the ninth bit in the rotation cycle standing in between the MSB and the LSB. If shifting, bits are shifted into the carry flag, but zeros are shifted into the byte from outside.

- The rotation instructions are:

  ROL
  ROR
  ASL
  LSR

  They may use direct, zero-page direct, indexed direct (X only), and zero-page indexed direct (X only) addressing. They affect the sign, zero, and carry flags.

CHAPTER 8 SELF-TEST

Code instructions to solve the following problems:

1. Turn on the LSB in the accumulator. ____________________
2. Turn off the MSB in the accumulator. ____________________
3. Complement the third and fourth bits in the accumulator. ____________________
4. Use AND to zero the accumulator. ____________________

5. Rotate the accumulator right through the carry flag. ____________

6. Shift the accumulator right. ____________

7. Rotate the accumulator left through the carry flag. ____________

8. Shift the accumulator left. ____________

9. Test the value of the byte at STATUS. ____________

10. Test the value of the byte at LOCAL to see if it's negative.

    ____

11. Read a digit from the terminal. Remove the ASCII character bits, leaving the binary value with at least four leading zeros. Then shift the number to the left three times, effectively multiplying it by eight. Store the result in PRODCT.

12. Test the byte in the memory area named DECIDE. If the LSB is zero, jump to ROUTE1. Otherwise, jump to ROUTE2.

Self-Test Answer Key

1.
```
ORA #1
```

2.
```
AND #%01111111
```

3.
```
EOR #%00110000
```

4.
```
AND #0
```

5.
```
ROR A
```

6.
```
LSR A
```

7.
```
ROL A
```

8.
```
ASL A
```

9.
```
BIT STATUS
```

10.
```
BIT LOCAL
BMI NEGLOC
```

11.
```
JSR   INPUT
AND   #%00001111
ASL   A
ASL   A
ASL   A
STA   PRODCT
```

12.
```
LDA   #1
BIT   DECIDE
BEQ   ROUTE1
JMP   ROUTE2 (or BNE)
```

or

```
LDA   DECIDE
ROR   A
BCC   ROUTE1
JMP   ROUTE2 (or BCS)
```

CHAPTER NINE

THE STACK

The stack is used primarily for the temporary storage of data. This chapter reviews the concepts associated with the stack then introduces the instructions you can use to manipulate it.

When you have finished this chapter, you will be able to:

- code the following instructions:
  - PHA (*P*us*H* *A*ccumulator into the stack)
  - PHP (*P*us*H* status register (*P*) into the stack)
  - PLA (*P*ul*L* *A*ccumulator from the stack)
  - PLP (*P*ul*L* status register (*P*) from the stack)
  - TXS (*T*ransfer *X* to *S*tack pointer)
  - TSX (*T*ransfer *S*tack pointer to *X*);
- code routines to accomplish the following functions:
  - Preserve the register values in the stack
  - Restore a register from the stack
  - Set the address in the stack pointer.

REVIEW OF CONCEPTS

1. The stack is a LIFO (last in, first out) storage area in memory. We use it for temporary storage. The stack pointer register points to the position where the next item will be stored.

Suppose there are five items in the stack, which we'll call A, B, C, D, and E. A was the first item stored and E the last.

(a) If you remove an item from the stack, which item will you get?

____

(b) If you remove another item from the stack, which item will you get?

____

— — — — — — — — — —

(a) E; (b) D

2. The stack is considered to have a top and a bottom. The bottom is the highest memory address and the top is the lowest memory address. The stack must be on page one. It usually starts at address $01FF and builds toward address 0100 as items are added to it.

Suppose that a stack in memory looks like this:

XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX
01F2	01F3	01F4	01F5	01F6	01F7	01F8	01F9	01FA	01FB	01FC	01FD	01FE	01FF

(a) What address is the top of the stack? ____________

(b) What address is the bottom of the stack? ____________

(c) At what address would you expect the SP to be pointing? ____________

— — — — — — — — — —

(a) $01F2; (b) $01FF; (c) $01F1

3. If your program wants to use the stack, you write the storage and retrieval instructions. Because it's limited to page one, it can be no longer than 256 bytes.

The JSR instruction also uses the stack, so any time your program contains a JSR, you're using the stack.

Which is the most accurate statement?

____ (a) Every program must use a stack.

____ (b) 256 bytes of memory are set aside for the stack, whether your program uses them or not.

____ (c) If you want to use a stack, you must reserve memory space for it.

— — — — — — — — — —

(b)

4. Match.

____ (a) add to stack

____ (b) retrieve from stack

1. SP is incremented
2. SP is decremented
3. move towards lower address
4. move towards higher address

— — — — — — — — — —

(a) 2, 3; (b) 1, 4

5. Whenever you write a program that uses the stack either directly or indirectly (through JSR), you'll have to consider the initial value of the stack pointer. Many systems automatically initialize the stack pointer to $01FF. If yours doesn't, you can initialize the stack pointer to $01FF using the TXS instruction (*T*ransfer *X* to *S*tack pointer). Here's an example:

```
LDX  #$FF
TXS
```

The system assumes the $01 page, so all you have to load is the $FF address.

(a) Suppose you want to use a stack starting at address $0199. (You're using $01A0 - $01FF for something else.) Write instructions to load the stack pointer correctly.

(b) Fill out Appendix C for TXS.

— — — — — — — — — —

(a)
```
LDX  #$99
TXS
```

(This is not considered a good idea. It's best to devote page one to the stack; use the rest of memory for other data.)

(b)

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
TXS	--	--	--	--	--	--	--	--	--

THE PUSH AND PULL INSTRUCTIONS

There are four push and pull instructions you can use to put items in the stack or get them out—PHA, PHP, PLA, and PLP.

6. A byte is added to the stack by a push instruction. PHA copies the accumulator into the stack. PHP copies the status register into the stack. Neither instruction has an operand, and no flags are affected. After the copy is made, the stack pointer is decremented to point to the new top of the stack.

(a) Which of the following is a legal push instruction?

____ `PHA`

____ `PHP $2319`

____ `PHA $05`

____ `PHP INBYTE`

(b) Fill out Appendix C for PHA and PHP.

— — — — — — — — — —

(a) `PHA`

(b)

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
PHA	–	–	–	–	–	–	–	–	–
PHP	–	–	–	–	–	–	–	–	–

7. Suppose you want to call a subroutine, MIXUP, that uses the accumulator. Before you call it, you want to preserve the *current* contents of the accumulator as well as the status flags. Write the instructions to store the status register and then the accumulator in the stack.

```
__________
__________
JSR  MIXUP
```

— — — — — — — — — —

```
PHP
PHA
```

8. An item is retrieved from the stack by a pull instruction. PLA pulls a byte into the accumulator. PLP pulls a byte into the status register. They have no operands. The zero and sign flags are affected.

Fill out Appendix C for PLA and PLP.

— — — — — — — — — —

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
PLA	–	–	–	–	–	–	–	–	–
PLP	–	–	–	–	–	–	–	–	–

9. A pull instruction virtually deletes an item from the stack, even though it doesn't erase the item. When the item is pulled, the stack pointer is positioned so that the next push will overlay the pulled item.

Which of the following statements is true?

____ (a) The same item can be pulled from the stack several times.

____ (b) When an item is pulled from the stack, it is the same as if the item was erased from the stack.

— — — — — — — — — —

(b)

10. Let's continue the problem you worked on before. You stacked the accumulator and the status register. then called the MIXUP subroutine. When control returns from MIXUP, you want to restore the accumulator and the status register to their former values. Be sure to restore the status flag byte to the status register and the accumulator byte to the accumulator. In other words, pull the bytes in the reverse order of the way they were pushed.

```
PHP
PHA
JSR   MIXUP
```

____

____

— — — — — — — — — —

```
PLA
PLP
```

11. See if you can write a routine that will stack the values of the X and Y registers, then call a subroutine named LOGON, then restore X and Y to their former values.

— — — — — — — — — —

Here's how we solve this problem:

```
TXA
PHA
TYA
PHA              ; Y REGISTER IS ON TOP
JSR  LOGON
PLA
TAY              ; SO PULL Y FIRST
PLA
TAX
```

Be sure your solution did not interchange the values in the X and Y registers.

12. Because we can't stack X and Y easily, we usually preserve them by storing them somewhere in memory.

The routine you coded in the previous frame could be more easily coded as:

```
        STX SAVEX
        STY SAVEY
        JSR LOGON
        LDX SAVEX
        LDY SAVEY
        . . .
SAVEX   DS  1
SAVEY   DS  1
```

Code a routine that preserves the value of register Y, calls a subroutine named NEWLOG, then restores Y.

— — — — — — — — — —

Here's how we would code it:

```
        STY SAVEY
        JSR NEWLOG
        LDY SAVEY
        . . .
SAVEY   DS 1
```

(This is a dangerous technique unless you *know* for a fact that NEWLOG will not use the data area containing SAVEY. Your registers are safer in the stack.)

THE TSX INSTRUCTION

13. You've seen how to initialize the stack pointer and how to move data into and out of the stack. One more instruction, TSX, allows you to copy the value of the stack pointer into the X register. It has no operands. The sign and zero flags are affected by the value that is copied.

(a) Fill out Appendix C for the TSX instruction.

(b) What instruction copies a value from SP to X? ______________________

(c) What instruction copies a value from X to SP? ______________________

— — — — — — — — — —

(a)	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
TSX	–	–	–	–	–	–	–	–	–

(b) TSX; (c) TXS

14. Suppose you need to increment the stack pointer by one (thus removing an item from the stack without pulling it). Write a routine that will retrieve the stack pointer into X, increment it, and replace it in the stack pointer.

— — — — — — — — — —

```
TSX
INX
TXS
```

REVIEW

In this chapter, you have learned how to set up and use one or more stacks.

- A stack is a LIFO storage area addressed by the SP (stack pointer) register. The top of the stack is the lowest address and the bottom is the highest. SP points the top, where items are added.
- The stack must be on page one. You should reserve all of page one for the stack.

- The PHA instruction copies the value from A to the top of the stack. PHP copies the value from the status register to the stack. The stack pointer is decremented by either operation.
- The PLA instruction copies the value from the top of the stack into the accumulator. PLP copies it from the top of the stack into the flag register. The stack pointer is incremented by either operation.
- The stack pointer should be initialized before you use it, unless your system does so automatically. TXS can be used to load the stack pointer.
- The TSX instruction copies SP into X. It has no operands.
- The TXS instruction copies X into SP. It has no operands.

CHAPTER 9 SELF-TEST

Code instructions to solve the following problems.

1. Point the stack pointer at $01FF.

2. For the above SP value, what address will be used for:

   the first stack entry?____________

   for the second entry?____________

3. Save all the registers except SP and PC in the stack.

4. Restore the registers again.

5. Store a byte of zeros in the stack.

Self-Test Answer Key

1.
```
LDX   #$FF
TXS
```

2.
```
$01FF
$01FE
```

3.
```
PHP
PHA
TXA
PHA
TYA
PHA
```

4.
```
PLA
TAY
PLA
TAX
PLA
PLP
```

(Be sure you pulled the registers in exact reverse order that you pushed them.)

5.
```
LDA   #0
PHA
```

One extreme
learned how
subroutines.
taking a clo
When

- co
- gi
su
- gi
su

WHAT A

1. Figure
throughout
validates ea
how it work

- T
lo
- T
d
- T
a
a
- T
- T
i
N

CHAPTER TEN

SUBROUTINES

One extremely important program structure is the subroutine. You have already learned how to call a subroutine and you have been calling INPUT and OUTPUT subroutines. In this chapter, you will learn how to code subroutines. We'll also be taking a closer look at input/output (I/O) subroutines.

When you have finished this chapter, you will be able to:

- code an instruction to return from a subroutine (RTS):
- given specifications, code a complete subroutine (including I/O subroutines);
- given specifications for a complete program including one or more subroutines, code the complete program.

WHAT ARE SUBROUTINES?

1. Figure 20 shows a complete program that we will use as an example throughout this chapter. The program reads and adds two digits between 0 and 4. It validates each input byte and writes an error message if a byte is invalid. Here's how it works:

- The GETN1 routine (lines 6-16) reads and validates the first input digit. It loops until a valid digit is obtained. SAVEN1 is part of GETN1.
- The GETN2 routine (lines 17-25) reads and validates the second input digit. It also loops until a valid digit is obtained.
- The ADDIT routine (lines 26-33) adds the two digits and writes the answer. Control is returned to GETN1 to start the next problem (creating a closed loop).
- The NEWLIN routine (lines 34-41) starts a new line on the terminal.
- The CHECK routine (lines 42-51) validates the byte in A. If the input byte is between '0' and '4,' the X register is set to 0. If not, it is set to $FF. NOGOOD is part of the CHECK routine.

- The ERROR routine (lines 52-61) puts out a message if the input doesn't validate. ERLOOP is part of this routine.
- The INPUT routine (lines 62-69) reads one byte from the keyboard and places it in the accumulator. TESTIN is part of this routine.
- The OUTPUT routine (lines 70-72) prints one byte from the accumulator to the terminal.

```
1                 ORG   $8000
2       ZERO      EQU   '0'
3       FIVE      EQU   '5'
4       CR        EQU   $0D
5       LF        EQU   $0A
6       GETN1     EQU   *
7                 JSR   INPUT
8                 JSR   OUTPUT
9                 JSR   NEWLIN
10                JSR   CHECK
11                CPX   #$FF
12                BNE   SAVEN1
13                JSR   ERROR
14                JMP   GETN1
15      SAVEN1    EQU   *
16                STA   NUM1
17      GETN2     EQU   *
18                JSR   INPUT
19                JSR   OUTPUT
20                JSR   NEWLIN
21                JSR   CHECK
22                CPX   #$FF
23                BNE   ADDIT
24                JSR   ERROR
25                JMP   GETN2
26      ADDIT     EQU   *
27                CLC
28                ADC   NUM1
29                SEC
30                SBC   #$30
31                JSR   OUTPUT
32                JSR   NEWLIN
33                JMP   GETN1
34      NEWLIN    EQU   *
35                PHA
36                LDA   #CR
37                JSR   OUTPUT
38                LDA   #LF
39                JSR   OUTPUT
40                PLA
41                RTS
42      CHECK     EQU   *
43                CMP   #ZERO
44                BMI   NOGOOD
45                CMP   #FIVE
46                BPL   NOGOOD
47                LDX   #0
48                RTS
49      NOGOOD    EQU   *
50                LDX   #$FF
51                RTS
```

FIGURE 20. Sample Program

```
52    ERROR    EQU   *
53             LDX   #0
54    ERLOOP   EQU   *
55             LDA   NOMSG,X
56             JSR   OUTPUT
57             INX
58             CPX   #38
59             BNE   ERLOOP
60             JSR   NEWLIN
61             RTS
62    INPUT    EQU   *
63             LDA   #%10000000
64    TESTIN   EQU   *
65             BIT   $C000        ; $C000 IS SET BY KEYBOARD INPUT
66             BEQ   TESTIN
67             LDA   $C000
68             STA   $C010
69             RTS
70    OUTPUT   EQU   *
71             JSR   $FDED
72             RTS
73    NOMSG    ASC   'YOU MUST ENTER A '
74             ASC   'DIGIT BETWEEN 0 AND 4'
75    NUM1     DS    1
```

FIGURE 20. Sample Program *(continued)*

What happens if the user types a 3?

____ (a) It is accepted and the program continues normally.

____ (b) The error message is written and the program loops back to get another digit.

____ (c) The program terminates itself.

What happens if the user types a 7?

____ (d) It is accepted and the program continues normally.

____ (e) The error message is written and the program loops back to get another digit.

____ (f) The program terminates itself.

— — — — — — — — — —

(a) and (e) are correct

2. Most Assembly Language programs contain three major parts:

- The *main line* is the code that's logically between the start (the first executed instruction) and the stop (however that happens). If the main line contains more than one routine (as indicated by labeled statements), the routines receive control by fall-through or branches.
- The *subroutines* are sections of code that are not in the main line. They receive control only by 'jump to subroutine' commands. They are positioned and coded so that they never receive control by fall-through, jumps, or branches.
- The *data area definitions* reserve memory bytes to hold data. They are positioned so that they will never receive control.

Refer to the example in Figure 20.

(a) Which lines comprise the main line? ____________________

(b) Which lines comprise the subroutines? ____________________

(c) Which lines comprise the data area definitions? ____________________

— — — — — — — — — —

(a) 6-33; (b) 34-72; (c) 73-75 [Note: Lines 1-5 are assembler directives, which don't fit into the other categories.]

3. A subroutine is a routine that receives control only by the JSR instruction. It usually performs one function (such as reading a byte from the terminal into the accumulator). It releases control by a *return*. Control returns to the instruction following the JSR.

(a) How does a subroutine receive control?

___ by JMPs

___ by JSRs

___ by fall-through

(b) How does a subroutine release control?

___ by executing a return

___ by reaching the last line

___ by jumping

(c) How many functions do most subroutines accomplish? ____________________

— — — — — — — — — —

(a) by JSRs; (b) by executing a return; (c) one

4. Using JSR to transfer control to a subroutine is usually referred to as 'calling' the subroutine. In many other assembly languages, the command to transfer control is actually 'CALL.' Throughout this book, we will use 'call' to mean 'transfer control by using JSR.'

Here's how a call works:

- The address in the PC register is pushed into the stack. At the time it is pushed, it is pointing at the third byte of the JSR instruction.
- The address in the JSR instruction operand is loaded into the PC. Thus, that address becomes the next instruction address.

Here's how a return works:

- The top of the stack is pulled into the PC. This should be the address that was pushed by the JSR instruction.

Suppose you have this situation:

```
address                 instruction

0110                    JSR   GETIT
0113                    STA   HERE
.
.
.
0200            GETIT   EQU   *
0200                    LDA   THERE
.
.
.
020A                    RTS
```

(a) What address is pushed by the JSR instruction? ____________

(b) What address is loaded into the PC by the JSR instruction? ____________

(c) What is the next instruction to be executed after the JSR instruction?

____

(d) What address is popped from the stack into the PC by the *Re*Turn from *S*ubroutine (RTS) instruction? ____________

(e) What is the next instruction to be executed after the RTS instruction?

____

— — — — — — — — — —

(a) $0112; (b) $0200; (c) LDA THERE; (d) $0112; (e) STA HERE

5. Examine the program in Figure 20 again. Which of the following are subroutines?

___ (a) GETN1	___ (f) OUTPUT	___ (k) NOGOOD
___ (b) GETN2	___ (g) SAVEN1	___ (l) ERROR
___ (c) ADDIT	___ (h) NOMSG	___ (m) ERLOOP
___ (d) INPUT	___ (i) NEWLIN	
___ (e) ZERO	___ (j) CHECK	

— — — — — — — — — —

(d), (f), (i), (j), (l)

[(a) - (c) are part of the main line; (e) is a symbolic value; (g) is part of the main line since control falls through; (h) is a data area; (k) is part of CHECK since it is reached by a branch; (m) is part of ERROR since control falls through]

CODING A SUBROUTINE

6. You must be careful when you code a subroutine. You want it to perform its function completely and accurately and have no unexpected side effects. And it must include at least one return instruction. It may have more than one return if alternate paths are established.

What are three characteristics of a good subroutine?

(a) ______________________________

(b) ______________________________

(c) ______________________________

— — — — — — — — — —

(a) complete and accurate; (b) no side effects; (c) at least one return

PRESERVING ORIGINAL VALUES

7. A subroutine avoids side effects by returning the registers and the stack in exactly the same condition that it receives them. Of course, it may use these areas. But it also restores them to their original values.

The exception is any area that is *supposed* to be affected by the subroutine's function. For example, the INPUT subroutine reads a byte into the accumulator. The accumulator and status byte register come out of the subroutine with their values changed. (Remember, any move to the accumulator, or X or Y register, affects the status flags.) The X and Y registers and the stack should be unchanged.

The OUTPUT subroutine writes one character from register A to a terminal. What areas would you expect to have different values after the routine has returned control?

___ (a) X register

___ (b) accumulator

___ (c) Y register

___ (d) status register

___ (e) the stack

___ (f) none of the above

— — — — — — — — — —

(f) is the correct answer. The accumulator still contains the byte that was written.

Caution: Don't assume that your system subroutines, such as the input and output ones, will leave your registers in good shape. Many of them don't. You'll want to either fix the subroutines or preserve your registers before calling.

8. The CHECK subroutine (see Figure 20) analyzes a value in register A. It places its results in the X register. If the value in A is between '0' and '4,' 0 is placed in X. Otherwise, $FF is placed in X.

What areas would you expect to have different values after CHEKIT returns control?

___ (a) accumulator

___ (b) status

___ (c) X register

___ (d) Y register

___ (e) the stack

___ (f) memory

___ (g) none of them

— — — — — — — — — —

(b) and (c)

9. If your subroutine needs to use a register. you preserve the incoming contents of that register by pushing it into the stack or saving it in a special memory location. Then you restore it before returning control. You may need to save the status register in the stack also.

Suppose your subroutine uses the accumulator and the X register. Both of them, and the status register, should be returned in their original condition. Write the necessary instructions to save and restore these from the stack.

- - - - - - - - - -

```
PHP
PHA
TXA
PHA
...
PLA
TAX
PLA
PLP
```

10. The accumulator, X, and Y registers can be saved at data locations; but the only way to save the status register is to put it in the stack. Write the instructions necessary to save and restore the status, X, and Y registers. Use data locations for storage wherever possible.

— — — — — — — — — —

```
PHP
STX   SAVEX
STY   SAVEY
...
LDX   SAVEX
LDY   SAVEY
PLP
```

(Notice that it doesn't really matter what order you use to restore data from memory; but the status register needs to be saved first and restored last, or it may be changed by the other operations.)

11. It's critically important that your subroutine pulls anything that it pushes. Remember that the return address is in the stack. If you leave the stack unbalanced, the remainder of the program won't work.

If your subroutine pushes registers X and Y, then what must it do before returning control?

— — — — — — — — — —

pull registers Y and X

RETURNING FROM A SUBROUTINE

12. A subroutine returns control when it reaches a return instruction. The operation code is RTS. There is no operand and no flags are affected.

Code instructions to return the value of X from the stack, then return control to the calling routine.

— — — — — — — — — —

```
PLA
TAX
RTS
```

— — — — — — — — — —

```
STALL   EQU  *
        PHP
        PHA
        STX  SAVEX
        LDX  #0
OUTLOP  EQU  *
        LDA  WAITMS,X
        JSR  OUTPUT
        INX
        CPX  #27
        BNE  OUTLOP
        LDX  SAVEX
        PLA
        PLP
        RTS
        ...
WAITMS  ASC  'PLEASE WAIT -- I''M THINKING'
SAVEX   DS   1
```

16. Code a subroutine that adds 5 to the contents of register A. If a carry results, reset the register to zero. Otherwise, just return control. It is not necessary to preserve the contents of any registers for this subroutine.

— — — — — — — — — —

```
INC5    EQU  *
        CLC
        ADC  #5
        BCC  ENDING
        LDA  #0
ENDING  RTS
```

PASSING DATA

17. Many subroutines require data to be passed to them. For example, look at the CHECK routine in Figure 20. It operates on a value in register A. That value was placed in there by the calling routine. We call such values *passed data.*

Which of the following types of subroutines would require data to be passed to them?

____ (a) a routine to read one byte from a terminal

____ (b) a routine to write one byte to the terminal

____ (c) a routine to start a new line on the terminal

— — — — — — — — — —

(b)

18. Suppose you are calling a subroutine that expects to print out one of several possible messages. All the possible messages are stored one after the other in a data area called 'TEXT.' The subroutine expects to find the length of the current message in the Y register. It expects to use the X register to find the first byte of the actual message. Code a set of instructions that will call this subroutine (named MESOUT) for a 15-character message starting at TEXT+9.

— — — — — — — — — —

```
LDY  #15
LDX  #9
JSR  MESOUT
```

I/O SUBROUTINES

In the preceding frames, you have learned how to code and call subroutines. Now we want to take a look specifically at input and output (I/O) subroutines. We can't show you exactly what subroutines you should use, but we can show you some common ones.

13. You should now be able to write subroutines. These next few frames will give you some practice.

Code a subroutine called ECHO that reads a byte into the accumulator, echoes it, and stores it in memory at INCHAR. Leave all the registers intact when you return control to the calling routine.

— — — — — — — — — —

```
ECHO    EQU  *
        PHP
        PHA
        JSR  INPUT
        JSR  OUTPUT
        STA  INCHAR
        PLA
        PLP
        RTS
```

If your output routine does not preserve the accumulator, you need to store INCHAR before calling the routine.

14. Code a subroutine that reads a byte from the terminal. If the input byte is less than $20, return control. If the byte is $20 or more, move it to the X register and then return control. Note that the contents of the X register are intended to be changed by this subroutine and should not be preserved.

– – – – – – – – – –

```
GEDATA EQU  *
       PHP
       PHA
       JSR  INPUT
       CMP  #$20
       BMI  ENDING
       TAX
ENDING EQU  *
       PLA
       PLP
       RTS
```

15. Code a subroutine called STALL that writes out this message: PLEASE WAIT – – I'M THINKING. (Use two single quotes to store one single quote.)

19. One major problem with I/O is that the microprocessor, which has no moving parts, can work so much faster than most I/O devices. In fact, the average 6502 microprocessor can read and store many thousand bytes per second. But a very fast keyboard can only send about 960 bytes per second. The average typist can type about four bytes per second. The bytes can't be read any faster than they become available.

On the output side, again the microprocessor can write many thousand bytes per second (more if it's not retrieving them from memory), but a very fast line printer can only type about 120 bytes per second and 30 is a more common speed. A byte can't be written until the previous byte is completed.

(a) Suppose you're writing an input subroutine to read one byte. What condition would you wait for before using the input data?

____

(b) Suppose you're writing an output subroutine to write one byte. What condition would you wait for before sending the output data?

____

(c) What do you think an I/O subroutine spends *most* of its time doing?

____

— — — — — — — — — —

(a) when the input device has a byte available; (b) when the output device is ready for one; (c) stalling, pausing, waiting, spinning its wheels. slowing down the computer

20. The primary input device of a 6502 microprocessor is usually some kind of keyboard operated by a human being (we hope). The output goes to a television screen, terminal, or printer.

There are several ways to coordinate the transfer of data to and from such devices. We'll describe one of the most common methods.

There are connections between the device and the microprocessor at two memory addresses. One location contains I/O data. The other one is for status information. The status information tells whether the device has an input byte ready to be read or is ready to receive an output byte. Some systems use separate sets of bytes for input and output.

(a) When you want to read a byte of data from the keyboard, which byte do you look at first—the data byte or the status byte?

____

(b) What does the status byte tell you? ______________________________

____

\- - - - - - - - - -

(a) the status byte; (b) whether the keyboard has sent data

21. Here's another method of handling input and output readiness. This is how our system handles it.

$C000 is the memory location for data from the keyboard. The keyboard sends standard seven-bit ASCII characters to this location. When a character arrives, a signal turns on the high-order bit. This means that $C000 has a value of $80 (128) or more whenever a byte of input data has arrived there.

Look at the input subroutine on lines 62-69 in Figure 20. You'll see we test the high-order bit of $C000. When we find it set, we move the input byte to the accumulator.

Any reference to location $C010 in our system will turn off the high-order bit in $C000. It doesn't matter what command we use; in Figure 20, we used an STA $C010 command on line 65. Turning off the high-order bit in $C000 keeps us from using the same input byte twice if we check the INPUT location again before the next byte arrives.

One problem with this input method is that our input characters don't match the standard ASCII characters—ours are all $80 higher.

(a) Using this method, how do we know when a new byte has arrived?

____

(b) How do we know when we have already read the byte at $C000?

____

\- - - - - - - - - -

(a) the high-order bit is on; (b) the high-order bit is off

22. Assume your processor receives input at location $2000 and has an input status byte at location $2001. This byte will be 'on'—set to $80—when an input byte has arrived at $2000; otherwise, it will be off. Code a subroutine to test the status byte until data is received; then fall through to instructions to put the byte in the accumulator, turn off the status byte, and return.

– – – – – – – – – –

```
INPUT   EQU  *
        STX  SAVEX
        LDA  #%10000000  ;FOR BIT TEST
TEST    EQU  *
        BIT  $2001
        BEQ  TEST
        LDA  $2000     ;GET NEW BYTE
        PHP            ;SAVE ITS STATUS
        LDX  #0
        STX  $2001
        LDX  SAVEX
        PLP            ;RESTORE ITS STATUS
        RTS
```

(Notice that we saved the status register after we moved the input byte to the accumulator; we might want to use it to test the new data, but the X register manipulations will reset some of the flags.)

23. A basic output subroutine shouldn't be any more difficult than an input subroutine. For example, suppose your processor has an output location at $5000 and an output status byte at $5050, which is set to $80 when the output device is ready; $00 otherwise. Code an appropriate output subroutine.

- - - - - - - - - - -

```
OUTPUT  EQU  *
        STX  SAVEX
        TAX
        LDA  #%10000000     ;FOR BIT TEST
TEST    EQU  *
        BIT  $5050
        BEQ  TEST
        TXA  ;RESTORE A REGISTER
        STX  $5000
        LDX  #$0
        STX  $5050
        LDX  SAVEX
        RTS
```

24. A simple output subroutine like the one in the preceding frame won't work if your output device is a television screen. The screen doesn't know where to put its output; the program needs to keep track of which position on the screen is "next." Subroutines to put out data on a video terminal can get pretty complicated.

On our system, there are standard input and output subroutines loaded into memory every time the system is started up. They are part of the system monitor. If you look at the output subroutine we used in Figure 20, you can see we call a subroutine at $FDED. This is our system's single-character output subroutine. We found the address for this subroutine in the reference manual that came with the computer.

One caution about using system subroutines—you'll need to find out what areas of memory these routines use, so you don't overlay them with your program's instructions and data area. Also, you'll need to see if they preserve your registers properly. If not, you'll have to preserve them yourself before calling the subroutine.

(a) If you want to do output to the video, what should you look for?

____

____

(b) If you use your system's standard input/output routines, what two precautions should you observe?

____

____

- - - - - - - - - - -

(a) look for a standard output subroutine in your system monitor; (b) don't overlay the subroutine with your own program, and make sure it preserves the registers properly

26. One type of output on some 6502 microprocessors uses a memory-mapped screen. This lets you build a complete screenful of data in memory and then transmit it all at once to the video screen.

Refer back to Figure 10. You see that the routine puts an output message in location $0400 - $0421. This is the first part of our screen map area. The map extends from $0400 - $07FF. Since we didn't clear the rest of the area. the text already on the screen will remain the same except for the first 22 bytes.

Once our screen map was built, we executed commands using addresses $C054 and $C051. It wouldn't matter what commands we used; any series of commands us ing these two addresses in order would display our revised output page on the screen.

Which of the following describe a memory-mapped screen? (Choose more than one.)

____ (a) Sends one character at a time to the screen.

____ (b) Sends a whole screenful at a time to the screen.

____ (c) A special part of memory is reserved for screen data.

____ (d) You send the data to the screen with the MAP instruction.

____ (e) You send the data to the screen by referencing a special address in memory.

— — — — — — — — — —

(b), (c), and (e) are correct

You have now seen how we write a complete input or output subroutine. Of course, they can be much fancier, especially if you want to do some error or parity checking. Routines that access printers look much the same as these. Routines for devices that use disk, tape, and cards usually require a lot more control code and timing routines. We cannot cover them in this book.

As to how you access your own devices, you'll need to find out how they communicate with the microprocessor. What are their memory addresses and how do they transmit status information? What system routines are available? Your manuals or your technical representative may be able to help you.

PROGRAM DESIGN

How do you decide what to code as a subroutine and what to put in the main line? There's no hard and fast rule. But the following frames present some guidelines.

26. Subroutines make programs easier for people to read and write. At one extreme, your main line could consist entirely of subroutine calls. All the detail work would be done by the subroutines. The logic of a program written this way is usual-

ly very clear. But the overhead (extra computer time) is tremendous! A call and return can take four times as long to execute as a jump instruction. And all those pushes and pulls add to the time—and use up more memory space.

Of course, we're talking about time differences measured in *microseconds* (one-millionths of a second). For the average application program, the extra time and storage involved by using a lot of subroutines will never be noticed. But most system programs must make the best possible use of time and space.

(a) Subroutines use (more/less) ________ time and space than main line routines.

(b) Most application programs should use (many/few) ________ subroutines.

(c) Most system programs should use (many/few) ________ subroutines.

— — — — — — — — — —

(a) more; (b) many; (c) few

27. We do use subroutines in system programs, but we use them only where they're really needed. One situation where we usually create a subroutine is when the same routine is executed at several different places in the program. A subroutine saves us from having to write the code several times. Jumping doesn't work as well because there's no mechanism for returning to the previous place when the routine finishes.

Refer to Figure 20 to answer the following questions.

(a) How many different places call NEWLIN? ________________________

(b) Why did we make NEWLIN a subroutine? ________________________

____

— — — — — — — — — —

(a) 4; (b) because it's used in four different places in the program.

28. Another reason we use a subroutine in a system program is that it's the same routine that appears in many programs. For example, we almost always use the INPUT and OUTPUT subroutines even though a particular program may only call them once each. Why? They save us coding time and they've been thoroughly tested.

Suppose your system includes a memory-mapped screen. You want to code a routine that clears the map of the screen.

(a) Would you make it a subroutine? ________________________

(b) Why or why not? ________________________

____

— — — — — — — — — —

(a) we would; (b) because you'll use it over and over again in many different programs

29. Let's review what you've learned about program design with respect to subroutines.

(a) Which type of program can maximize the use of subroutines — system or application? ____________

(b) List two primary types of routines that you should consider making into subroutines. ____________

____

— — — — — — — — — —

(a) application; (b) ones that are used more than once in the same program and ones that are used in many different programs

REVIEW

In this chapter, you've learned how to code and call subroutines, especially I/O subroutines.

- A subroutine is a routine that is not in the main line of control. It receives control by being called and it returns control to the calling routine when it finishes.
- A good subroutine accomplishes one function completely and accurately, has no unexpected side effects, and contains at least one return instruction.
- Side effects are avoided by returning the stack and the registers in their original condition, except for those that are supposed to be affected. Push and pull instructions can be used to preserve the registers and restore them. Be sure to pull all items, in reverse order, that have been pushed. You can also save them at memory locations, if you prefer.
- The return instruction is RTS (*R*e*T*urn from *S*ubroutine). It has no operand.
- For many I/O subroutines for terminals and printers, it's necessary to test a status byte before reading or writing data. The status byte is set to a value indicating whether the terminal is ready to send or receive data.

- Application programs should make free use of subroutines because they make the program logic easier to understand. However, subroutines take more time and space and in general should be minimized in system programs. Routines that are good candidates for subroutines are those that are used several times in the same program and those that are used in many programs within a system.

CHAPTER 10 SELF-TEST

Part I. Code the calling routines or subroutines specified below.

1. This subroutine writes out a message from the data area TEXT to an output device. The displacement for the first byte of the message is passed in the X register. The length of the message is passed in the Y register. Call the OUTPUT routine to actually write each byte.

2. This subroutine reads, echos, and validates an incoming byte. A byte is valid if it is an upper case letter. If invalid, write the message WRONG – – TRY AGAIN. Continue reading until a valid byte is obtained. Leave the byte in the accumulator and return control.

   Use the INPUT and OUTPUT subroutines to read and write single bytes. Call the subroutine you wrote for question 1 to write the message.

3. This routine—not a subroutine—reads and stores an incoming message in memory. It should loop until the letter Z is read. That terminates the message. (Store the Z.)

   Each character of the incoming message should be read, echoed, and validated using the subroutine you wrote for question 2 above.

4. This subroutine writes one character from the accumulator on a printer. Use these features:

   - Status byte at address $C50F.
   - MSB indicates output status; on for ready.
   - Data byte at address $C50A.

5. This subroutine reads one character from a keyboard. The terminal status byte is at $DD1C. The first and second bits (LSB and LSB+1) are on when a byte is ready to be read. The data byte is at address $DD1D. Put the character in the accumulator.

6. This subroutine prints text on the printer by calling the print subroutine you wrote for question 4 above. After each character is printed, check for an input byte from the terminal described in question 5 above. If any byte is input, discontinue printing.

   The following data is passed to this subroutine: (a) the beginning displacement of the text is in register X; (b) the length of the text is in register Y.

7. This routine causes the following message to be printed, preceded by form feed and carriage return:

   NOW IS THE TIME FOR ALL GOOD PEOPLE TO COME TO THE AID OF THEIR PARTY.

   Use the subroutine you wrote for question 6 above to print the text.

Self-Test Answer Key

Part I.

1.
```
WRITE   EQU  *
        PHP
        PHA
OUTLOP  EQU  *
        LDA  TEXT,X
        JSR  OUTPUT
        INX
        DEY
        BNE  OUTLOP
        PLA
        PLP
        RTS
```

2.
```
UPPERA  EQU  $41
ZPLUS1  EQU  $5B                ; THIS IS 'Z' PLUS 1
GETLET  EQU  *
        STY  SAVEY
TRYONE  EQU  *
        JSR  INPUT
        JSR  OUTPUT
        CMP  #UPPERA
        BMI  ERROR
        CMP  #ZPLUS1
        BMI  OK
ERROR   EQU  *
        LDY  MESLNG
        LDX  #0
        JSR  WRITE
        JMP  GETLET
OK      EQU  *
        LDY  SAVEY
        RTS
SAVEY   DS   1
MESLNG  DFB  18
TEXT    ASC  'WRONG -- TRY AGAIN'
```

3.
```
        LDX  #0
READIN  EQU  *
        STX  SAVEX
        JSR  GETLET
        LDX  SAVEX
        STA  INMES,X
        INX
        CMP  #'Z'
        BMI  READIN
DONE    JMP  DONE
INMES   DS   80
```

4.
```
PRINT   EQU  *
        STX  SAVEX
        PHA
        LDA  #%10000000
TEST    EQU  *
        BIT  $C50F
        BEQ  TEST
        PLA
        STA  $C50A
        LDX  #0
        STX  $C50F
        LDX  SAVEX
        RTS
```

5.
```
INPUT   EQU  *
        STX  SAVEX
        LDA  #%00000011
TEST    EQU  *
        BIT  $DD1C
        BEQ  TEST
        LDA  $DD1D
        PHP
        LDX  #0
        STX  $DD1C
        LDX  SAVEX
        PLP
        RTS
```

6.
```
PTEX    EQU  *
        LDA  TEXT,X
        JSR  PRINT
        LDA  #%00000011
        BIT  $DD1C
        BNE  ENDSUB
        INX
        DEY
        BNE  PTEX
ENDSUB  EQU  *
        RTS
```

7.
```
NOWPRN  EQU  *
        LDY  #72
        LDX  #0
        JSR  PTEX
DONE    JMP  DONE
TEXT    DBF  $0D,$0A
        ASC  'NOW IS THE TIME FOR ALL '
        ASC  'GOOD PEOPLE TO COME TO THE '
        ASC  'AID OF THEIR PARTY.'
```

CHAPTER ELEVEN

NUMERIC MANIPULATION

You have already learned how to add and subtract numbers up to 255 using one byte. But many computer applications require much larger numbers than this. Multiplication and division are also necessary, as well as the ability to handle negative numbers.

In this chapter, we'll introduce you to some techniques for handling numbers with Assembly Language. There isn't room in this book to cover them all, but we will show you how to code a routine that will add numbers several bytes long. We'll show you how to handle basic multiplication and division. And finally, we'll show you how to use twos complement notation to handle negative numbers.

When you have finished this chapter, you will be able to:

- code the following instructions:
  - SED (*SE*t *D*ecimal mode)
  - CLD (*CL*ear *D*ecimal mode)
  - CLV (*CL*ear o*V*erflow flag);
- code routines to solve the following types of problems:
  - convert ASCII to binary coded decimal (BCD)
  - convert BCD to ASCII
  - add multibyte BCD values
  - subtract multibyte BCD values
  - multiply multibyte values
  - divide single-byte values
  - convert ASCII to twos complement notation
  - convert twos complement notation to ASCII.

MULTIBYTE ADDITION

1. Multibyte arithmetic is done on values stored in memory. Each byte is moved into the accumulator as it is needed.

In multibyte arithmetic, we usually don't work with values that have been converted to pure binary form. We work instead with a data representation system called *binary coded decimal* (BCD). You'll also hear it referred to as *packed decimal.*

Here are some decimal values as they are represented in binary, hexadecimal, BCD, and the hexadecimal representation of BCD.

decimal value	*pure binary*	*(hex)*	*BCD*	*(BCD-hex)*
21	%00010101	($15)	%00100001	($21)
18	%00010010	($12)	%00011000	($18)
30	%00011110	($1E)	%00110000	($30)

Notice the correlation between the decimal value and the BCD hex value. Give the BCD values for these decimal numbers. Show your answers in hex.

(a) 05 = ______(BCD)

(b) 19 = ______(BCD)

(c) 54 = ______(BCD)

— — — — — — — — — —

(a) $05; (b) $19; (c) $54

2. The BCD system simply splits the two halves of a byte apart and treats them as separate storage areas. Half a byte is called a nibble (that's someone's idea of a joke), and we'll speak of the least significant or lower nibble and the most significant or upper nibble.

In BCD, each nibble will hold a value from 0 to 9. Values A through F are forbidden. The normal binary equivalents of 0 to 9 are used (see Figure 1).

For example, the BCD equivalent of 83 holds 8 in the upper nibble, 3 in the lower nibble. In hex BCD notation, it is coded $83; in binary BCD, it is coded %10000011.

Give the BCD equivalents of these decimal values. Write your answers in both binary and hex.

(a) 32 = %____________ (BCD) = $____________ (BCD)

(b) 10 = %____________ (BCD) = $____________ (BCD)

— — — — — — — — — —

(a) %00110010, $32; (b) %00010000, $10 (Be sure you translated each digit separately.)

3. Which of the following values are illegal in BCD?

___ (a) $39

___ (b) $4B

___ (c) $20

___ (d) $FF

— — — — — — — — — —

(b) and (d) are illegal because they contain digits above 9

4. Assume we have two one-byte BCD values in memory.

```
ADD1   $05
ADD2   $04
```

We want to add these together, with the result in ADD1. Write instructions to accomplish the following:

(a) Put the ADD2 value in the accumulator.

____

(b) Add the ADD1 value to it and put the result in ADD1.

(c) What is the hex form of the value in ADD1 now? ________________

(d) Is this a valid BCD value? ________________

(e) Suppose the original values were both 5. Would the result be a valid BCD value?

____

— — — — — — — — — —

(a)
```
LDA   ADD2
```

(b)
```
CLC
ADC   ADD1
STA   ADD1
```

(c) $09; (d) yes; (e) no—A is not valid in BCD

5. The result in the accumulator may no longer be a BCD number because the computer uses binary/hexadecimal arithmetic. Here are some examples:

```
  $03              $06        $27        $39
+ $04            + $05      + $36      + $08
  ---              ---        ---        ---
  $07  (ok!)       $0B        $5D        $41
```

To prevent this problem, we need to use an instruction called SED (*SE*t *D*ecimal mode). This instruction, which has no operands, instructs the processor to use BCD arithmetic until further notice.

When you're doing BCD arithmetic, before you start you should use the ________ instruction.

— — — — — — — — — —

```
SED
```

6. Now let's assume that we have two BCD values in memory, each two bytes long.

ADEND1

0X	XX

ADEND2

0Y	YY

We want to add these two values together, storing the sum in ADEND2. Notice that each of them has at least one leading zero. This will make sure the result will fit in ADEND2. First we'll add the rightmost bytes, using indexed addressing.

Write a set of instructions that will move the least significant byte of ADEND1 into the accumulator, then add the least significant byte of ADEND2 to it using BCD arithmetic. Put the result in the least significant byte of ADEND2.

Write your code on a separate piece of paper using pencil. You're going to add to and change this routine until you've built a complete addition program.

Programming Note: Right now we're working with only two bytes. But eventually you want to be able to add fields of any size. Don't use an expression such as ADEND1+1 to access these bytes. Use indexed addressing.

— — — — — — — — — —

```
SED
LDX  #1
LDA  ADEND1,X
CLC
ADC  ADEND2,X
STA  ADEND2,X
```

7. Now we're ready to handle the most significant byte. The only difference between this byte and the least significant byte is the possibility of a carry coming from the previous addition.

Remember that when we added the least significant bytes of ADEND1 and ADEND2, we stored the sum in ADEND2. The carry flag was left intact. If there was a carry out of that least significant byte, the carry flag will be on. Otherwise, it will be off.

Now add a set of instructions to treat the most significant bytes. Write a routine that will:

(1) move the ADEND1 byte into the accumulator

(2) add the ADEND2 byte to it, plus the value in the carry flag

(3) store it in the correct byte of ADEND2

— — — — — — — — — —

Here is our whole routine so far:

```
ADDER   EQU   *
        SED
        LDX   #1
        CLC
        LDA   ADEND1,X
        ADC   ADEND2,X
        STA   ADEND2,X
      ( DEX
NEW  {  LDA   ADEND1,X
        ADC   ADEND2,X
      ( STA   ADEND2,X
```

The new part is marked.

8. Notice that the routine so far adds two bytes. It contains two similar sets of instructions. Each set:

— stores one byte in accumulator

— adds in other byte

— stores result in memory

While the operations are accomplished a bit differently, we can create a loop that will add two numbers of any size.

Adapt your addition routine so that it is one loop that adds two five-byte numbers and stores the sum.

Hints: The first loop adds the least significant bytes. Make sure the carry flag is set to 0 before entering the loop so you can use the ADC instruction. End the loop when register X equals 0.

— — — — — — — — — —

```
ADDER   EQU  *
        SED
        LDX  #4
        CLC
ADDBYT  EQU  *
        LDA  ADEND1,X
        ADC  ADEND2,X
        STA  ADEND2,X
        DEX
        BPL  ADDBYT
```

This routine will work with any size numbers up to 255 bytes long if you adjust the value in X.

BCD CONVERSION

Now that you've seen how to add two BCD numbers, let's talk about where they came from in the first place. In the following set of frames, we'll show you how to expand the program you've already coded so that it reads a set of ASCII digits from a terminal and converts them to BCD.

9. Let's start by reading one digit from the terminal and converting it from ASCII to binary. All you have to do is turn off all the bits in the most significant nibble. Leave the result in the accumulator.

Write your code on a separate piece of paper so you can build an entire routine as before. Use JSRs for any I/O you need.

— — — — — — — — — —

```
JSR  INPUT
JSR  OUTPUT
AND  #%00001111
```

10. In BCD, it's very important that only valid decimal digits are used. Convert the routine you coded in the previous frame into a complete subroutine that gets one valid digit:

(1) Read and echo one character.

(2) Validate range '0' to '9' (make sure the character is in the range).

(3) If the character is out of range, write an error message and try again.

(4) When a valid digit is obtained, convert it to binary and return control. Leave the new value in A.

Don't forget to make it a subroutine.

— — — — — — — — — —

```
GETDIG EQU  *
       STX  HOLDX
AGAIN  EQU  *
       JSR  INPUT        ; GET ONE DIGIT
       JSR  OUTPUT
       CMP  #'0'         ; RANGE CHECK
       BMI  ERROR
       CMP  #':'         ; COMPARE TO CHARACTER AFTER '9'
       BPL  ERROR        ; ERROR IS EQUAL OR GREATER THAN ':'
       AND  #%00001111   ; CONVERT ASCII TO BCD
       LDX  HOLDX
       RTS
ERROR  EQU  *
       LDX  #0
MSGOUT EQU  *
       LDA  ERRMSG,X
       JSR  OUTPUT
       INX
       CPX  #26
       BNE  MSGOUT
       JMP  AGAIN
HOLDX  DS   1
ERRMSG ASC  'INVALID DIGIT -- TRY AGAIN'
```

11. Now we have a subroutine that will put one valid binary digit in register A. How do we get from there to BCD? For correct BCD format, we have to work with two digits at a time. We combine them into one byte this way:

(1) rotate the first digit into the upper nibble

(2) add in the second digit

Code a routine that will get two valid decimal digits (by calling GETDIG) and create one BCD byte. Store the byte at ADEND1.

— — — — — — — — —

```
GETNUM EQU  *
       JSR  GETDIG    ; GET MSBYTE
       ASL  A
       ASL  A
       ASL  A
       ASL  A
       STA  MSB
       JSR  GETDIG    ; GET LSBYTE
       CLC
       ADC  MSB
       STA  ADEND1
       ...
MSB    DS   1
```

12. Now complete your routine so that it gets and stores all of ADEND1 (ten digits = five bytes) and ADEND2 (also five bytes). For the user's benefit, start a new line on the display screen after each 10-digit number. You'll want to use a "nested loop." Have an inner loop that is executed five times for one 10-digit number. Have an outer loop that executes the inner loop twice for the two numbers.

— — — — — — — — — —

Our complete routine looks like this:

```
        CR      EQU   $0D
        LF      EQU   $0A
        GETADS  EQU   *
                LDX   #0          ; INITIALIZE INDEX
NEW             LDA   #2          ; CONTROL OUTER LOOP FOR 2 NUMBERS
                STA   COUNT2
        GETAD   EQU   *
                LDA   #5          ; CONTROL INNER LOOP FOR 5 BYTES PER NUMBER
                STA   COUNT5
        GETNUM  EQU   *
                JSR   GETDIG      ; GET MSBYTE
                ASL   A
                ASL   A
                ASL   A
                ASL   A
                STA   MSB
                JSR   GETDIG      ; GET LSBYTE
                CLC
                ADC   MSB
                STA   ADEND1,X    ; IF X > 4, WE'RE IN ADEND2
                INX
                DEC   COUNT5      ; COUNT INNER LOOP
                BNE   GETNUM
                LDA   #CR         ; CR DEFINED AT BEGINNING
                JSR   OUTPUT
NEW             LDA   #LF         ; LF DEFINED AT BEGINNING
                JSR   OUTPUT
                DEC   COUNT2
                BNE   GETAD       ; GET 2ND ADDEND
                ...
        COUNT2  DS    1
        COUNT5  DS    1
        ADEND1  DS    5
        ADEND2  DS    5
        MSB     DS    1
```

13. You've seen how to get the BCD addends and how to add them. Now we need to convert the result (in ADEND2) back into ASCII. Code a routine that will:

(1) get one byte from ADEND2

(2) split the two nibbles

(3) convert to ASCII

(4) write both digits

(5) repeat steps (1) through (4) until the entire sum is written out.

— — — — — — — — — —

```
MASCII EQU  %00110000
WRITIT EQU  *
       LDX  #0
CONVRT EQU  *
       LDA  ADEND2,X     ; GET BYTE
       LSR  A
       LSR  A
       LSR  A
       LSR  A
       ORA  #MASCII      ; CONVERT TO ASCII
       JSR  OUTPUT
       LDA  ADEND2,X     ; GET IT AGAIN
       AND  #%00001111   ; USE LOWER NIBBLE
       ORA  #MASCII      ; CONVERT TO ASCII
       JSR  OUTPUT
       INX
       CPX  #5
       BNE  CONVRT
```

Figure 21 shows our entire program to read and add two ten-digit numbers. This program has certain awkward points. The user must type all ten digits, including leading zeros. Also, we haven't taken any steps to prevent overflow of the sum. The error message routine dumps the message in the middle of the user's number. To clean up these difficulties would require a lot of code that is not the subject of this chapter. But you would want to do so before actually using this program.

```
MASCII EQU  %00110000
CR     EQU  $0D
LF     EQU  $0A
GETADS EQU  *
       LDX  #0           ; INITIALIZE INDEX
       LDA  #2           ; CONTROL OUTER LOOP
       STA  COUNT2
GETAD  EQU  *
       LDA  #5           ; CONTROL INNER LOOP
       STA  COUNT5
GETNUM EQU  *
       JSR  GETDIG
       ASL  A
       ASL  A
       ASL  A
       ASL  A
       STA  MSB
       JSR  GETDIG       ; GET LSB
       CLC
       ADC  MSB
       STA  ADEND1,X     ; IF X > 4, WE'RE IN ADEND2
       INX
       DEC  COUNT5       ; COUNT INNER LOOP
       BNE  GETNUM
       LDA  #CR          ; CR DEFINED AT BEGINNING
       JSR  OUTPUT
       LDA  #LF          ; LF DEFINED AT BEGINNING
       JSR  OUTPUT
       DEC  COUNT2
       BNE  GETAD        ; GET 2ND ADDEND
ADDER  EQU  *
       SED
       LDX  #4
       CLC
ADDBYT EQU  *
       LDA  ADEND1,X
       ADC  ADEND2,X
       STA  ADEND2,X
       DEX
       BPL  ADDBYT
WRITIT EQU  *
       LDX  #0
CONVRT EQU  *
       LDA  ADEND2,X     ; GET BYTE
       AND  #%11110000   ; USE UPPER NIBBLE
       LSR  A
       LSR  A
       LSR  A
       LSR  A
       ORA  #MASCII      ; CONVERT TO ASCII
       JSR  OUTPUT
       LDA  ADEND2,X     ; GET IT AGAIN
       AND  #%00001111   ; USE LOWER NIBBLE
       ORA  #MASCII      ; CONVERT TO ASCII
       JSR  OUTPUT
       INX
       CPX  #5
       BNE  CONVRT
DONE   JMP  DONE
```

FIGURE 21. Multibyte Addition Program

```
MASCII EQU  %00110000
CR     EQU  $0D
LF     EQU  $0A
GETADS EQU  *
       LDX  #0          ; INITIALIZE INDEX
       LDA  #2          ; CONTROL OUTER LOOP
       STA  COUNT2
GETAD  EQU  *
       LDA  #5          ; CONTROL INNER LOOP
       STA  COUNT5
GETNUM EQU  *
       JSR  GETDIG
       ASL  A
       ASL  A
       ASL  A
       ASL  A
       STA  MSB
       JSR  GETDIG      ; GET LSB
       CLC
       ADC  MSB
       STA  ADEND1,X    ; IF X > 4, WE'RE IN ADEND2
       INX
       DEC  COUNT5      ; COUNT INNER LOOP
       BNE  GETNUM
       LDA  #CR         ; CR DEFINED AT BEGINNING
       JSR  OUTPUT
       LDA  #LF         ; LF DEFINED AT BEGINNING
       JSR  OUTPUT
       DEC  COUNT2
       BNE  GETAD       ; GET 2ND ADDEND
ADDER  EQU  *
       SED
       LDX  #4
       CLC
ADDBYT EQU  *
       LDA  ADEND1,X
       ADC  ADEND2,X
       STA  ADEND2,X
       DEX
       BPL  ADDBYT
WRITIT EQU  *
       LDX  #0
CONVRT EQU  *
       LDA  ADEND2,X    ; GET BYTE
       AND  #%11110000  ; USE UPPER NIBBLE
       LSR  A
       LSR  A
       LSR  A
       LSR  A
       ORA  #MASCII     ; CONVERT TO ASCII
       JSR  OUTPUT
       LDA  ADEND2,X    ; GET IT AGAIN
       AND  #%00001111  ; USE LOWER NIBBLE
       ORA  #MASCII     ; CONVERT TO ASCII
       JSR  OUTPUT
       INX
       CPX  #5
       BNE  CONVRT
DONE   JMP  DONE
```

FIGURE 21. Multibyte Addition Program

```
GETDIG EQU   *
       STX   HOLDX
AGAIN  EQU   *
       JSR   INPUT
       JSR   OUTPUT
       CMP   #'0'   ; RANGE CHECK
       BMI   ERROR
       CMP   #':'
       BPL   ERROR
       AND   #%00001111
       LDX   HOLDX
       RTS
ERROR  EQU   *
       LDX   #0
MSGOUT EQU   *
       LDA   ERRMSG,X
       JSR   OUTPUT
       INX
       CPX   #26
       BNE   MSGOUT
       JMP   AGAIN
COUNT2 DS    1
COUNT5 DS    1
ADEND1 DS    5
ADEND2 DS    5
MSB    DS    1
HOLDX  DS    1
ERRMSG ASC   'INVALID DIGIT -- TRY AGAIN'
```

FIGURE 21. Multibyte Addition Program *(continued)*

MULTIPLICATION

Multiplication is really a process of repeated addition. In the frames that follow, we'll show you how to multiply numbers in Assembly Language. First you'll see how to do it in pure binary. Then we'll show you how to do it in BCD.

14. There are no multiply instructions. You multiply by shifting left. Each shift multiplies the accumulator by two.

Suppose you want to multiply the accumulator by two. Write an instruction that will do it.

— — — — — — — — — —

```
ASL  A
```

15. The first shift left multiplies by two. The second doubles the first, or multiplies by four. The third doubles the second or multiplies by 8.

Using ASL, write a set of instructions to multiply by 16.

— — — — — — — — — — —

```
ASL  A   ;  TIMES 2
ASL  A   ;  TIMES 4
ASL  A   ;  TIMES 8
ASL  A   ;  TIMES 16
```

16. Write a set of instructions to read an ASCII byte, convert it to binary, multiply it by 8, and store it at BYTEX8.

— — — — — — — — — — —

```
JSR  INPUT
JSR  OUTPUT
AND  #%00001111
ASL  A           ; TIMES 2
ASL  A           ; TIMES 4
ASL  A           ; TIMES 8
STA  BYTEX8
```

17. Multiplying by a power of two is easy. But what about multiplying by a number that's not a power of two? We can accomplish any multiplicand by adding together the various powers of two. For example, to multiply by 5:

- put the original value in HOLDA;
- shift left once (2A);
- shift left again (4A);
- *add* the original byte from HOLDA (4A + A = 5A).

(a) Write a routine to multiply the value in the accumulator by 7.

(b) Write a routine to multiply the value in the accumulator by 10.

— — — — — — — — — —

(a)
```
STA   HOLDA
ASL   A        ; TIMES 2
CLC
ADC   HOLDA    ; TIMES 3
ASL   A        ; TIMES 6
ADC   HOLDA    ; TIMES 7
```

(b)
```
ASL   A        ; TIMES 2
STA   HOLD2A   ; SAVE
ASL   A        ; TIMES 4
ASL   A        ; TIMES 8
CLC
ADC   HOLD2A   ; TIMES 10
```

18. So far, you've been writing multiplication routines that operate on pure binary values. The same techniques will not work in BCD arithmetic. BCD multiplication must be accomplished by successive additions.

In the preceding frame, you wrote a routine to multiply by 10. Convert that routine to work on a one-byte BCD value. Assume for now that the result will not overflow the register.

— — — — — — — — — —

```
        SED
        LDX   #10
        STA   HOLDA
MULTY   EQU   *
        DEX
        BEQ   DONE
        CLC
        ADC   HOLDA
        JMP   MULTY
DONE    EQU   *
```

19. Now let's look at how we multiply a multibyte BCD value. Suppose we have a three-byte BCD value called MULTER.

MULTER ⟶

99	99	99

We want to multiply it by six and store the product in MULTED, which has 4 bytes and an initial value of 0.

MULTED ⟶

00	00	00	00

All we have to do is write a loop that adds MULTER to MULTED. Then execute the loop six times. Try it. (Watch out for that high order byte of MULTED. Don't forget to add any carries into it.)

— — — — — — — — — —

```
         SED
         LDA  #6        ; LOOP
         STA  COUNT6    ; COUNTER
ONEADD   EQU  *
         LDX  #2        ; INDEX FOR MULTER
         LDY  #3        ; INDEX FOR MULTED
         CLC
NEXBYT   EQU  *
         LDA  MULTED,Y
         ADC  MULTER,X
         STA  MULTED,Y
         DEY
         DEX
         BPL  NEXBYT
         LDA  MULTED,Y
         ADC  #0        ; ADD CARRY TO MSB OF MULTED
         STA  MULTED,Y
         DEC  COUNT6
         BNE  ONEADD
```

DIVISION

20. Division is a process of repeated subtraction. Before we start, review these terms.

```
           QUOTIENT
DIVISOR )DIVIDEND
          XXXXXXXX
          REMAINDER
```

Use this division problem to answer the questions below.

```
    2
7 )15
   14
    1
```

(a) What is the dividend? _______

(b) What is the remainder? _______

(c) What is the divisor? _______

(d) What is the quotient? _______

— — — — — — — — — —

(a) 15; (b) 1; (c) 7; (d) 2

21. Here's how we divide by two. Assume that the dividend is already in the accumulator. The quotient will be stored in X.

```
        LDX  #0      ; CLEAR THE QUOTIENT
SUBIT   EQU  *
        CMP  #2      ; CAN WE MAKE ANOTHER SUBTRACTION?
        BMI  DONE
        SEC
        SBC  #2      ; SUBTRACT DIVISOR FROM A
        INX          ; COUNT SUBTRACTION
        JMP  SUBIT
DONE    EQU  *
        ...
```

The quotient will end up in X and the remainder in A. What we do is count the number of times we can subtract the divisor (2) from the dividend.

(a) Change the above routine to divide by any number between 1 and 255. Assume the divisor is in DIVISR.

```
        LDX  #0      ; CLEAR THE QOUTIENT
SUBIT   EQU  *
        CMP  DIVISR ; CAN WE SUBTRACT?
        BMI  DONE
        SEC
        SBC  DIVISR ; SUBTRACT DIVISOR FROM A
        INX
        JMP  SUBIT
DONE    EQU  *
```

(Note that a 0 in DIVISR would cause a closed loop.)

HANDLING NEGATIVE NUMBERS

So far in this book, we've been assuming that all values are positive. But the arithmetic routines should also be able to handle negative numbers. Negative numbers are stored using *twos complement* notation. In this section, we'll teach you how to use twos complement notation.

22. When we want to work with negative numbers, we usually use twos complement notation to represent negative numbers. When we're working with eight-bit numbers, twos complements are two numbers that add up to %100000000. When we add complementary numbers in the accumulator, the result is zero with the carry flag on. For example, %10000001 and %01111111 are twos complements.

In binary, there's a very simple way to find the twos complement of any number. Complement (reverse the value of) each bit and add 1 to the result. For example:

```
%01100011   (value)
%10011100   (complemented bits)
       +1
---------
%10011101   (twos complement)
```

Find the twos complements of the following numbers.

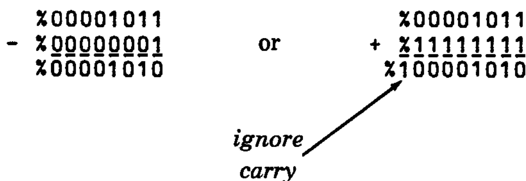
(a) %00001111 : __________

(b) %01101110 : __________

— — — — — — — — — —

(a) %11110001; (b) %10010010

23. The twos complement of a number has a very interesting property: As long as you ignore the carry flag, it acts just like the negative of the original value. Thus, %11111111 acts just like −%00000001. And %00000001 acts just like −%11111111. For example, suppose we want to subtract 1 from 1011. There are two ways to do it:



Which of the following looks like the easier way to handle negative numbers?

____ (a) Convert all negative numbers to twos complement notation as soon as they're entered. Then let all additions and subtractions proceed as if only positive numbers were in use, except ignore the carry flag.

____ (b) Store all numbers with either plus or minus signs. For each arithmetic operation, examine the signs of both operands and decide whether addition or subtraction is more appropriate. When subtracting two numbers. be sure to subtract the smaller from the larger absolute value.

— — — — — — — — — —

(a) is much easier (and more efficient in terms of computer time and space)

24. In 6502 Assembly Language, it's fairly easy to get the twos complement of a number. All you have to do is complement all the bits and add one. Here's how:

```
LDA  NEGNUM
EOR  #%11111111
CLC
ADC  #1
```

The accumulator now holds the twos complement of NEGNUM.

Write a set of instructions to find the twos complement of SUBBER. Leave the result in SUBBER.

— — — — — — — — — —

Here are two ways:

```
LDA  SUBBER
EOR  #%11111111
STA  SUBBER
INC  SUBBER
```

or

```
LDA  SUBBER
EOR  #%11111111
CLC
ADC  #1
STA  SUBBER
```

25. When we're using a twos complement system, we let the most significant bit act as the sign indicator. If it's on, the value is negative. If it's off, the value is positive. This means that we have to limit positive values to %01111111 per byte.

To convert to decimal, you examine the sign bit. If it's clear, the number is positive. Just convert it directly. If it's set, find the twos complement of the number and put a minus sign in front of it.

In decimal, what are the equivalents of the following signed binary values?

(a) %00000000 = ________

(b) %01111111 = ________

(c) %10000000 = ________

(d) %11111111 = ________

— — — — — — — — — —

(a) 0; (b) 127; (c) −128; (d) −1

26. In a twos complement system, what is the maximum positive value per byte?

____

What is the most negative value that will fit in a byte? ________

— — — — — — — — — —

127; −128

27. Which flag will tell you whether the number you're working with is positive or negative when you're using twos complement notation?

____

— — — — — — — — — —

the sign flag (remember the sign flag is set to match the MSB in a result byte)

28. Now we'll start building a program that reads two single digits, adds them, and reports the sum. Either digit may be negative.

Let's start by coding a subroutine that gets one digit.

Read one byte. If it's '−', read another byte (a digit) and find the twos complement of that value. In either case, convert the value to binary and leave it in A. (Use separate paper, as you'll add to this routine in later frames.)

— — — — — — — — — —

```
GETBYT EQU  *
       JSR  INPUT        ; GET BYTE
       JSR  OUTPUT
       CMP  #'-'         ; IS IT A MINUS SIGN?
       BEQ  NEGIVE
       AND  #%00001111   ; CONVERT TO BINARY
       JMP  ENDING
NEGIVE EQU  *            ; IT'S A MINUS SIGN
       JSR  INPUT        ; GET THE DIGIT
       JSR  OUTPUT
       AND  #%00001111   ; CONVERT TO BINARY
       EOR  #%11111111   ; GET
       CLC               ;     TWOS
       ADC  #1           ;           COMPLEMENT
ENDING EQU  *
       RTS
```

29. Now code a routine that gets two bytes. Store the first byte in memory. Leave the second byte in A. Add the two bytes and leave the sum in A. Call the subroutine you coded in the previous frame to get each byte.

— — — — — — — — — —

```
ADDER  EQU  *
       JSR  GETBYT
       STA  ADDEND
       JSR  GETBYT
       CLC
       ADC  ADDEND
```

30. Now it's time to report the result. Add the instruction BPL SUMPOS to the routine you just wrote. If the result is positive and under ten, we want to convert it to ASCII and write it out.

Code a routine (SUMPOS) to check the size of the result. If it's under ten, convert the value in A into ASCII and write it out. If the value is ten or more, branch to a routine named TWODIG (don't code TWODIG yet.)

— — — — — — — — — —

```
SUMPOS EQU  *
       CMP  #10
       BCS  TWODIG
       ORA  #%00110000        ; CONVERT TO ASCII
       JSR  OUTPUT
```

31. Now code the TWODIG routine. If the sum is positive and larger than nine, you must convert it to two decimal digits. Since you haven't been using BCD arithmetic, here's what you have to do.

(1) divide the value in A by ten

(2) the quotient is the most significant digit, convert it to ASCII and write it out

(3) the remainder is the least significant digit, convert it to ASCII and write it out.

— — — — — — — — — —

```
TWODIG EQU  *
       LDX  #0          ; X WILL HOLD QUOTIENT (HIGH ORDER DIGIT)
DIVIDE EQU  *
       SEC
       SBC  #10
       INX
       CMP  #10         ; ARE THERE ANY 10'S LEFT?
       BCS  DIVIDE      ; BRANCH IF 10 OR GREATER
       TAY              ; TEMP HOLD REMAINDER (LOW ORDER DIGIT)
       TXA              ; PUT QUOTIENT INTO A
       ORA  #%00110000  ; CONVERT TO ASCII
       JSR  OUTPUT
       TYA              ; PUT REMAINDER INTO A
       ORA  #%00110000  ; CONVERT TO ASCII
       JSR  OUTPUT
```

32. Now let's deal with a negative result. All you have to do is write a '–', find the twos complement of the value in the accumulator, then follow the same routine as SUMPOS. Write the code and fit it between BPL SUMPOS and the SUMPOS routine in the program.

— — — — — — — — — —

```
       BPL  SUMPOS
       TAX              ; TEMP STORE
       LDA  #'-'
       JSR  OUTPUT
       TXA              ; GET IT BACK
       EOR  #%11111111  ; GET
       CLC              ;     TWOS
       ADC  #1          ;          COMPLEMENT
SUMPOS EQU  *
```

Our entire program is shown in Figure 22.

```
ADDER    EQU  *
         JSR  GETBYT
         STA  ADDEND
         JSR  GETBYT
         CLC
         ADC  ADDEND
         BPL  SUMPOS
         TAX                  ; TEMP STORE NEG SUM
         LDA  #'-'
         JSR  OUTPUT
         TXA                  ; GET NEG SUM
         EOR  #%11111111      ; GET
         CLC                  ;     TWOS
         ADC  #1              ;          COMPLEMENT
SUMPOS   EQU  *
         CMP  #10
         BCS  TWODIG
         ORA  #%00110000      ; CONVERT TO ASCII
         JSR  OUTPUT
QUIT     JMP  QUIT
TWODIG   EQU  *
         LDX  #0          ; X WILL HOLD QUOTIENT (HIGH ORDER DIGIT)
DIVIDE   EQU  *
         SEC
         SBC  #10
         INX
         CMP  #10         ; ARE THERE ANY 10'S LEFT?
         BCS  DIVIDE      ; BRANCH IF 10 OR GREATER
         TAY              ; TEMP HOLD REMAINDER (LOW ORDER DIGIT)
         TXA              ; PUT QUOTIENT INTO A
         ORA  #%00110000  ; CONVERT TO ASCII
         JSR  OUTPUT
         TYA              ; PUT REMAINDER INTO A
         ORA  #%00110000  ; CONVERT TO ASCII
         JSR  OUTPUT
         JMP  QUIT
GETBYT   EQU  *
         JSR  INPUT          ; GET BYTE
         JSR  OUTPUT
         CMP  #'-'           ; IS IT A MINUS SIGN?
         BEQ  NEGIVE
         AND  #%00001111     ; CONVERT TO BINARY
         JMP  ENDING
NEGIVE   EQU  *              ; IT'S A MINUS SIGN
         JSR  INPUT          ; GET THE DIGIT
         JSR  OUTPUT
         AND  #%00001111     ; CONVERT TO BINARY
         EOR  #%11111111     ; GET
         CLC                 ;     TWOS
         ADC  #1             ;          COMPLEMENT
ENDING   EQU  *
         RTS
INPUT    EQU  *
         JSR  $FD0C
         RTS
OUTPUT   EQU  *
         JSR  $FDED
         RTS
ADDEND   DS   1
```

FIGURE 22. Adding Positive and Negative Digits

33. So far, we've worked on twos complement problems that do not involve any carries. When we work with larger values, we have to worry about the result exceeding the range of −128 through +127 and affecting the sign bit.

The overflow flag warns us of any overflow into the most significant bit as a result of addition or subtraction. It is wise to check the status of the overflow flag after every signed arithmetic operation. If it's set, branch to a routine that handles the carry and corrects the sign.

The overflow flag can be cleared with the CLV (*CL*ear o*V*erflow) instruction, which has no operand.

(a) When using signed arithmetic, which flag indicates that a carry has occurred?

____

(b) Suppose the name of your overflow routine is OVERFL. Code an instruction to branch to that routine if the result of the addition shown below overflows the accumulator.

```
ADC ADDEND
```

____

(c) Fill out Appendix C for the CLV instruction.

— — — — — — — — — —

(a) overflow; (b) BVS OVERFL;

(c)	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
CLV	–	–	–	–	–	–	–	–	–

REVIEW

In this chapter, we have expanded the subject of numeric manipulation.

- Multibyte addition and subtraction are usually done in binary coded decimal (BCD) notation because it's easier to work with. In BCD, each nibble (half-byte) represents a decimal digit from 0 to 9. To convert ASCII input to BCD, the first digit is converted to binary then rotated to the upper nibble. The second digit is converted to binary then placed in the lower nibble.
- To use BCD notation, set the decimal flag with SED. All arithmetic will be done in BCD format until the flag is cleared with CLD.
- To convert from BCD to ASCII, you'll need two copies of the same byte. For the upper nibble (the first digit), eliminate the lower nibble, rotate the upper nibble into the lower nibble, and convert to ASCII. For the lower nibble, eliminate the upper nibble and convert to ASCII.

- We use BCD because the conversion procedures between ASCII and pure binary are so complex. Also because there's no limit on the size of a value that can be handled that way.
- Multiplication is a process of repeated addition—a value is added to itself. Each addition multiplies the value by a power of two. To multiply by a factor that is not a power of two, save the needed partial products, such as 1X and 2X, and add them in.
- Division is done by repeated subtraction. The divisor is subtracted from the dividend until the remainder of the dividend is smaller than the divisor. Each subtraction is tallied in another register. The tally becomes the quotient and the amount left over in the accumulator is the remainder.
- Negative values are usually handled by twos complement notation. To convert a binary value in the accumulator to twos complement notation, use EOR #%11111111, which complements all the digits; then add 1 to the result. When working with twos complement notation, limit all positive values to seven bits (127). Negative values range from %11111111 (−1) to %10000000 (−128). There is no −0.

You have only begun to solve the problems of numeric manipulation. We'll have to leave the rest up to you. Here are some areas you may want to explore: multibyte division, multiplication and division of negative numbers, fractional quantities, and finding roots. There are no additional 6502 instructions that you'll need. It's simply a matter of how you combine the ones you already know.

CHAPTER 11 SELF-TEST

Part I. Code the instructions described below.

1. Add ADDEND to the accumulator, with carry. ____________________
2. Subtract 5 from the accumulator, with borrow. ____________________
3. Subtract SUBBER from the accumulator, with borrow. ____________________
4. Add 1 to the accumulator, with carry. ____________________
5. Complement all the bits in the accumulator. ____________________

Part II. Code routines to accomplish the following functions.

1. Registers A and X contain two ASCII digits. Convert them to one BCD byte and leave it in A. The most significant digit is in X and the least significant digit is in A. They have both been validated to be between '0' and '9'.

2. Register A contains a positive BCD value. Convert it to ASCII and write it out.

3. These two fields are currently holding BCD values. Add them, leaving the sum in ADEND1.

ADEND1	0X	XX	XX
ADEND2	0Y	YY	YY

4. Convert your answer to the previous problem to subtract ADEND2 from ADEND1, leaving the answer in ADEND1.

5. Multiply ADEND1 (from question 3 above) by 9. Store the result in PRODCT.

6. Register A is holding a positive, pure binary value between 0 and 255. Divide it by 20. Leave the quotient in QUOTNT and the remainder in A.

7. Register A is holding an ASCII value between 0 and 127. Convert it to twos complement notation.

8. Register A is holding a negative value between −1 and −9 in twos complement notation. Convert it to ASCII.

Self-Test Answer Key

Part I.

1. `ADC ADDEND`

2. `SBC #5`

3. `SBC SUBBER`

4. `ADC #1`

5. `EOR #%11111111`

Part II.

1.
```
AND  #%00001111          ; CONVERT LSD TO BINARY
STA  TEMSLD              ; TEMP SAVE LSD
TXA                      ; GET MSD
AND  #%00001111          ; CONVERT TO BINARY
ASL                      ; SHIFT
ASL                      ;          TO
ASL                      ;               UPPER
ASL                      ;                         NIBBLE
CLC
ADC  TEMLSD              ; ADD LSD TO MSD
```

2.
```
STA  TEMVAL              ; SAVE COPY OF VALUE
AND  #%11110000          ; MASK OUT LOWER NIBBLE
LSR                      ; ROTATE
LSR                      ;          TO
LSR                      ;               LOWER
LSR                      ;                         NIBBLE
ORA  #%00110000          ; CONVERT TO ASCII
JSR  OUTPUT              ; WRITE FIRST DIGIT
LDA  TEMVAL              ; GET COPY
AND  #%00001111          ; MASK OUT UPPER NIBBLE
ORA  #%00110000          ; CONVERT TO ASCII
JSR  OUTPUT              ; WRITE IT OUT
```

3.

```
         LDX  #2           ; INDEX FOR LSB
         SED               ; FOR BCD ARITHMETIC
         CLC               ; CLEAR CARRY FOR FIRST ADDITION
LOOP     EQU  *
         LDA  ADEND1,X     ; GET BYTE
         ADC  ADEND2,X     ; ADD IT
         STA  ADEND1,X     ; STORE SUM
         DEX               ; DECREMENT INDEX
         BPL  LOOP
```

4.

```
         LDX  #2           ; INDEX FOR LSB
         SED               ; FOR BCD ARITHMETIC
         SEC               ; FOR FIRST SUBTRACTION
LOOP     EQU  *
         LDA  ADEND1,X     ; GET BYTE
         SBC  ADEND2,X     ; SUB IT
         STA  ADEND1,X     ; STORE DIFFERENCE
         DEX
         BPL  LOOP
```

5.

```
MULT1    EQU  *
         LDX  #2
         SED
         CLC
ADD1CE   EQU  *
         LDA  PRODCT,X
         ADC  ADEND1,X
         STA  PRODCT,X
         DEX
         BPL  ADD1CE
         DEC  COUNT9
         BNE  MULT1
DONE     EQU  *
         . . .
COUNT9   DFB  9
PRODCT   DFB  0,0,0
```

6.

```
         LDX  #0                ; CLEAR
         STX  QUOTNT            ; QUOTNT
DIVIDE   EQU  *
         CMP  #20
         BMI  DONE
         SEC
         SBC  #20
         INC  QUOTNT
         JMP  DIVIDE
DONE     JMP  DONE
         ...
QUOTNT   DS   1
```

7.

```
AND  #%00001111          ; CONVERT TO BINARY
EOR  #%11111111          ; COMPLEMENT IT
CLC
ADC  #1                  ; ADD ONE
```

8.
```
EOR   #%11111111          ; COMPLEMENT IT
CLC
ADC   #1                  ; ADD ONE
ORA   #%00110000          ; CONVERT TO ASCII
```

CHAPTER TWELVE

ADDITIONAL INSTRUCTIONS

You have learned to code the most heavily used Assembly Language instructions. In this final chapter, we'll briefly introduce some instructions that are less frequently used. Some day you might be trying to solve a problem that requires one of these instructions and you'll remember that it's available. These instructions are presented out of the context of programs because the programs that use most of them would be quite complex. You'll just learn what the instructions are and how they function.

When you have finished this chapter, you will be able to code the following instructions:

- NOP (*N*o *OP*eration)
- SEI (*SE*t *I*nterrupt mask)
- CLI (*CL*ear *I*nterrupt mask)
- RTI (*R*e*T*urn from *I*nterrupt)
- BRK (*BR*ea*K*)

THE NOP INSTRUCTION

1. NOP (no operation) does absolutely nothing active. It takes up one byte of memory space and uses up a little bit of time.

With very primitive systems, NOPs were important. The programmer inserted several NOPs between all the instructions to leave room for insertions later. This isn't necessary for systems with an editor and a terminal because it's fairly easy to insert instructions.

What instruction causes nothing active to happen? ________________

- - - - - - - - - -

NOP

2. Many microcomputers can receive a signal from the outside to get their attention. Such a signal is called an interrupt request because it asks the microcomputer to *interrupt* whatever it is doing and service some more urgent need. The computer finishes the instruction it is processing, then acknowledges the request and services it by running a special program. It then continues the interrupted program where it left off. The interrupt system is dependent on the hardware. On some systems, I/O operations are handled this way. The instructions you'll read about below are all concerned with programming that uses interrupt I/O.

An interrupt occurs when one program is interrupted, an interrupt service program is executed, then the first program is picked up where it left off.

You can only interrupt if you have an external device capable of sending an "interrupt request." If your system has a monitoring device attached, for example, its input might be processed on an interrupt basis rather than a normal read basis. The external monitor would send an interrupt request when it sensed a situation that needed immediate processing.

(a) Which of the following best describes an interrupt?

____ Pausing in program A to execute program B, then resuming program A at the same point.

____ Discontinuing program A to run program B.

____ Stopping program A to run program B, then restarting program A from the beginning.

(b) How do you cause an interrupt?

____ By typing any key when the program isn't expecting input.

____ By hitting the "break" key on any device.

____ By causing an I/O device to send an interrupt request.

____ By pulling the plug.

— — — — — — — — — —

(a) pausing in program A to execute program B, then resuming program A at the same point; (b) by causing an I/O device to send an interrupt request (the means is device specific)

SEI AND CLI INSTRUCTIONS

3. The 6502 microprocessor has two forms of interrupts: masked and unmasked. Unmasked interrupts are always recognized and processed. The 6502 will only respond to a masked interrupt request if interrupts are enabled. You must enable masked interrupts for each program that you want to be interruptable by using the CLI instruction. CLI uses only the operation code; it has no operand. It clears the interrupt disable flag in the status register. When the flag is clear, the system will respond to interrupts.

(a) When will the 6502 respond to interrupts?

___ When the interrupt disable flag is set.

___ When the interrupt disable flag is clear.

(b) What Assembly Language instruction enables interrupts? ______________

— — — — — — — — — —

(a) when the interrupt flag is clear; (b) CLI

4. You may have certain routines that you don't want interrupted. For example, if you've written a timing loop to count off exactly 2.3 microseconds, an interrupt would destroy the timing. If so, you might want to disable interrupts when you enter the loop and enable them again when the loop is over. You disable interrupts with the SEI instruction. Like CLI, SEI has no operand.

(a) What instruction enables interrupts? ______________

(b) What instruction disables interrupts? ______________

(c) Can you disable unmasked interrupts? ______________

— — — — — — — — — —

(a) CLI; (b) SEI; (c) no

5. Interrupt processing itself automatically disables interrupts. That is, when the microprocessor interrupts program A and gives control to program B, it automatically issues an SEI instruction. This is to prevent the interrupt service program (program B) from being interrupted.

If you want the interrupt program to be interruptable, you include a CLI instruction at the beginning of it.

(a) How does the microprocessor prevent interrupt routines from being interrupted? ______________

(b) What do you do if you want an interrupt routine to be interruptable? ______________

— — — — — — — — — —

(a) by disabling interrupts when it transfers control to the interrupt program; (b) code CLI at the beginning

THE RTI INSTRUCTION

6. An interrupt routine works similarly to a subroutine. When it is called, the PC and the status registers are pushed into the stack.

The RTI (*Re*T*urn from *I*nterrupt) instruction causes a return to the interrupted program. The PC and status registers are pulled from the stack. Since the interrupt flag is in the status register, its former value is restored when the status register is pulled from the stack.

(a) Suppose you are coding an interrupt routine. Code the instruction to terminate the routine. ______________________________

(b) What will be the status of the interrupt flag after a return from an interrupt?

____

— — — — — — — — — —

(a) RTI; (b) whatever it was before the interrupt

THE BRK INSTRUCTION

7. The BRK (*BR*ea*K*) instruction causes an interrupt from your Assembly Language program. It sets the break flag.

When coding your interrupt routine, you can check the value of the break flag to find out if the interrupt came from outside or from a BRK instruction.

(a) What are the major causes of interrupts? ______________________________

____

____

(b) How can you tell the difference between an external interrupt and an interrupt from a BRK instruction? ______________________________

____

— — — — — — — — — —

(a) unmasked external interrupts, masked external interrupts, and interrupts from BRK instructions; (b) the break flag will be set in the status byte in the stack if the interrupt was caused by BRK

The previous frames have just brushed on the area of interrupt processing so you'll know it exists. How, where, and when you use the interrupt instructions to build a coordinated interrupt system are topics beyond the scope of this book.

REVIEW

In this chapter, we have briefly introduced some instructions that are useful in certain situations.

- NOP causes nothing to happen.
- Several instructions are designed for use in interrupt processing:
  - CLI enables maskable interrupts
  - SEI disables maskable interrupts
  - BRK causes an interrupt from within a program
  - RTI causes a return from an interrupt program

You have now learned, or at least been introduced to, all the 6502 Assembly Language instructions. After you complete the following Self-Test, you'll be done with this book. You'll be able to use the documentation for your system as you modify or create Assembly Language programs.

CHAPTER 12 SELF-TEST

1. Code an instruction to do nothing. ______________________

2. Code an instruction to disable maskable interrupt processing. __________

3. Code an instruction to enable maskable interrupt processing. __________

4. Code an instruction to return from an interrupt program. __________

5. Code an instruction to force an interrupt. ______________________

6. Fill out Appendix C for all these instructions.

Self-Test Answer Key

1. `NOP`

2. `SEI`

3. `CLI`

4. `RTI`

5. `BRK`

6.

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
NOP	--	--	--	--	--	--	--	--	--
SEI	--	--	--	--	--	--	--	--	--
CLI	--	--	--	--	--	--	--	--	--
RTI	--	--	--	--	--	--	--	--	--
BRK	--	--	--	--	--	--	--	--	--

You have finished your Self-Teaching Guide on 6502 assembly language. You are familiar with all the instructions and operand formats, and you have had a lot of practice using the ones that are most common. You're also equipped to begin reading some of the many excellent advanced texts (not Self-Teaching) dealing with this subject.

Have fun!

Appendix A

HEXADECIMAL ADDITION - SUBTRACTION TABLE

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Appendix B
ASCII CODE

	first hex digit (8-F not used)							
second hex digit	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P		p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	2F	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

EXPLANATION OF CONTROL CHARACTERS

NUL - null character; eight zero bits

The following are used in data communications (transmitting data via phone lines):

SOH - start of heading
STX - start of text
ETX - end of text
EOT - end of transmission
ENQ - enquire ("Are you there?")
ACK - acknowledge ("Yes")
DLE - data link escape
NAK - negative acknowledgement ("No")
SYN - synchronous idle
ETB - end of transmission

BEL - bell; rings the terminal alarm
BS - backspace
HT - horizontal tab; tab across to next tab stop
LF - line feed; move down one line
VT - vertical tab; tab down to next vertical tab stop
FF - form feed; go to top of next page
CR - carriage return; go to beginning of line
SO - shift out; shift out of ASCII code
SI - shift in; shift back into ASCII code
DC1, DC2, DC3, DC4 - device controls 1-4; the meaning of these four codes depends on the terminal equipment
CAN - cancel
EM - end of medium
SUB - substitute
ESC - escape; used like a shift key to extend ASCII code
FS - file separator
GS - group separator
RS - record separator
US - unit separator
DEL - delete

Appendix C
INSTRUCTION SUMMARY

		[1] immediate	[2] direct	[3] zero-page direct	[4] indexed direct	[5] zero-page indexed direct	[6] indirect	[7] pre-indexed indirect	[8] post-indexed indirect	[9] relative	FLAGS carry	FLAGS zero	FLAGS sign	FLAGS overflow	FLAGS other
register operations	LDA	ok	ok	ok	XYok	Xok	—	ok	ok	—		X	X		
	LDX											X	X		
	LDY											X	X		
	STA														
	STX														
	STY														
	TAX											X	X		
	TAY											X	X		
	TXA											X	X		
	TYA											X	X		
	INX											X	X		
	INY											X	X		
	DEX											X	X		
	DEY											X	X		
	TXS														
	TSX											X	X		
arithmetic operations	ADC										X	X	X	X	
	SBC										X	X	X	X	
	SED														D
	CLD														D
	DEC											X	X		
	INC											X	X		
	CLC										0				

flag operations	SEC										1				
	CLV													0	
	CMP										X	X	X		
	CPX										X	X	X		
	CPY										X	X	X		
program control	JMP														
	JSR														
	RTS														
	BEQ														
	BNE														
	BMI														
	BPL														
	BCS														
	BCC														
	BVS														
	BVC														
	BRK														I,B
logical operations	AND											X	X		
	ORA											X	X		
	EOR											X	X		
	BIT											X	X	X	
	ASL										X	X	X		
	LSR										X	X	X		
	ROL										X	X	X		
	ROR										X	X	X		
stack operations	PHA														
	PHP														
	PLA											X	X		
	PLP										X	X	X	X	all
other	CLI														I
	NOP														
	SEI														I
	RTI										X	X	X	X	

Appendix D

FINISHING YOUR PROGRAM

In order to get a program running, you must complete these steps:

1. Code the complete program, including all subroutines.
   a. Allocate memory space for all data areas. Chapter 6 tells you how in general, but you'll need to find out specifically how your system works.
   b. If you are not coding your own I/O subroutines, be sure your program calls the appropriate subroutines from the system monitor. Chapter 9 tells you how in general, but you'll need to find out specifically how your system works.
2. Use your system editor to enter the code into your computer. You'll have to find out how your system editor works. Your assembler manual should explain it to you.
3. Use your system assembler to assemble your code. Again, you'll need your assembler manual.
4. Use the assembler error messages to correct your code. (Keep using that assembler manual.)
5. Once the program has successfully assembled, you can run your program. With most computers, you would call up the program just the same way you call up any other program. (Probably by entering the file name, perhaps preceded by a RUN command.)
6. If the program doesn't work properly, try to figure out what it *is* doing so you can decide how to correct the code.
7. If your program ends in a closed loop, you'll have to interrupt it to get back to the system monitor in order to do anything else. Try pushing a key labeled BREAK or RESET. If you can't figure out what to do, you can always reboot (restart) the system.
8. Fix your code and assemble and run it again until it works.
9. If you need help beyond your manual, see if you can find someone who knows how to use your system to help you.

GOOD LUCK!

INDEX

A register, 9, 15, 63
accumulator (also see A-register), 3, 9, 15, 48, 84, 86, 87, 88, 94, 95, 97, 176, 179, 185, 188, 189, 193, 195, 200, 210
actual adddress, 66, 69, 71
ADC, 48, 52, 62, 95, 102
addition (see also multibyte arithmetic), 33-35, 95-97, 102
address, 8, 11, 27, 64, 70, 110, 113-126, 134
addressing modes, 61, 69, 77, 84
  also see:
    immediate addressing
    direct addressing
    zero-page direct addressing
    indexed direct addressing
    zero-page indexed direct addressing
    indirect addressing
    pre-indexed indirect addressing
    post-indexed indirect addressing
    relative addressing
alphanumeric, 1, 6
alternate program paths, 141, 164-168, 169, 208
AND, 173, 174, 176-178, 181, 188
APPLE, 40
application programs, 3, 15
ASC, 120, 135
ASCII, 6-7, 15, 19, 40-42, 43, 152, 157, 178, 222, 237, 240, 255-256, 270-271
ASL, 173, 185, 189
assembly language, 1-3, 15, 25, 47
ATARI, 40

BASIC, 2
BCC, 141, 146, 169
BCD, 231-241, 245-247, 255-256
BCS, 141, 146, 169
BEQ, 52, 62, 141, 146, 150, 169
binary (numbers), 6, 15, 19, 21, 22, 23, 25, 26-28, 31-32, 34-35 37-38, 43
binary coded decimal, see BCD
bit, 1, 6-7, 15
BIT, 173, 182-183, 189
BMI, 141, 146, 169
BNE, 141, 146, 169
BPL, 141, 146, 169
branch instruction (see also jump instruction), 73, 100-101
break flag, 13, 15, 142, 266
BRK, 263, 266, 267
BVC, 141, 146, 169
BVS, 141, 146, 169
byte, 1, 5-7, 15

call, 207
carry flag, 13, 15, 94, 95, 96, 97-98, 102, 142, 143, 146, 147 184, 189, 235, 249
chip, see microprocessor
CLC, 96, 102
CLD, 231, 255
CLI, 263, 264, 267
CLV, 231, 255
CMP, 52, 62, 141, 162, 169
COBOL, 2
comments, 47, 48, 49, 53-55, 56, 57
comparisons, 161-164
compiler, 2
complement, 97, 102, 180, 181, 189
conditional instructions, 142-169
CPX, 141, 163, 169
CPY, 141, 163, 169
current memory address, 122, 131, 135

data names, 49
data storage area, 49, 119
data transfers, 14
DEC, 99, 103

decimal (numbers), 6, 19-24, 25, 26, 28-32, 43
decimal flag, 13, 15, 142
decrement, 99, 103
DEX, 99, 103
DEY, 99, 103
DFB, 120, 135
direct address, 64, 68, 77, 84, 85, 87, 90, 99, 100, 110, 112, 185, 189
directive, 107-135
disk unit, 4
division, 247-249, 256
DS, 119, 134

EBCDIC, 6-7, 15, 19
echo, 92, 152
empty path, 167-168, 169
ending a program, see loop, closed
EOR, 173, 180-181, 189
EQU, 126-134, 135
EXCLUSIVE OR (also see EOR), 175, 180-181, 189
expresion, 76, 78

flags, 12-13, 15
also see:
zero flag
sign flag
carry flag
overflow flag
interrupt disable flag
break flag
decimal flag
FORTRAN, 2

general purpose registers, 9, 10, 15
hexadecimal, 8, 19, 20, 22, 23, 25-28, 29-30, 33-34, 36, 43
high-level language, 1, 2, 14

immediate addressing, 61-63, 68, 77, 84, 85, 111, 112, 134
indirect addressing, 69-70, 77, 85, 100, 110, 112
indexed direct addressing, 66, 68, 77, 84, 85, 87, 99, 110, 112, 185
indexed addressing (also see index registers), 15, 67, 234
index registers, 10, 15, 66, 77
increment, 99, 103
INC, 99, 103
input/output routines, 4, 90-94, 103, 215-220, 222
instructions, 82-103
instruction format, 47-57
interrupt, 264, 266, 267
interpreter, see compiler
interrupt disable flag, 13, 15, 142, 264, 265
INX, 99, 103
INY, 99, 103
I/O, see input/output routines

JMP, 48, 52, 55, 62, 69, 100, 103, 144
JSR, 48, 90, 103, 194, 195, 206, 207
jump instructions, 12, 15, 49, 115, 116

label, 47, 48, 49-52, 56, 57, 65, 75-76, 78, 117-119, 121, 126, 134, 135, 148
LDA, 48, 52, 55, 62, 66, 84, 86, 102
LDX, 85, 86, 102
LDY, 85, 86, 102
LIFO, 193, 199
listing (assembler listing), 25, 33, 108
logical operations, 173-188
loop, closed, 101, 103, 154
loop, open, 141, 151, 169
low-level language, 1, 5, 14, 15
LSR, 173, 185, 189

machine language, 1-3, 57, 73, 108-113
main line, 206, 222
main storage (also see memory), 5, 15
mask, 173, 177, 179, 180, 181, 188, 189
memory, 5, 7-9, 48, 83, 84, 87, 135
memory mapped screen, 220
memory stack, see stack
microprocessor, 1, 5, 7, 8, 9, 10, 15, 216
multibyte arithmetic, 231-248, 255
multiplication, 242-247, 256

nibble, 232
NOP, 263, 267
normalize, 187

operands, 47, 48, 53, 55-56, 57, 61-78, 84, 85, 88, 96, 97, 98, 99, 134-135
operation codes, 48, 50, 51-42, 56, 57, 62, 108-113, 134
or, 173, 174, 179-180, 181, 188
ORA, 173, 179-180
ORG, 114, 122-126, 135
overflow flag, 13, 15, 94, 102, 142, 146, 182, 189, 255

packed decimal, see BCD
page, 64
page one, 64, 194, 195, 199
PC register, 9, 12, 15, 100, 115, 116, 209, 266
peripheral device, 4
PET, 40, 64, 65

PHA, 193, 195, 200
PHP, 193, 195, 200
PLA, 193, 195, 196, 200
PLP, 193, 195, 196, 200
post-indexed indirect addressing, 72-73, 77, 84, 85, 87, 111, 112
pre-indexed indirect addresing, 71, 77, 84, 85, 87, 111, 112
pointer, 4, 216
program counter, see PC register
program design, 220-222
program layout, 132-134
pseudo-operation, see directive

register, 1, 3, 5, 9-14, 15, 50, 63, 83, 209
register rotation, 184-188
relative addresing, 73-74, 77, 85, 111, 112, 148
result bit, 173
ROL, 173, 185, 189
ROR, 173, 185, 189
RTI, 263, 266, 267
RTS, 203, 211, 222

SBC, 97-98, 102
SEC, 98, 102
SED, 231, 233, 255
SEI, 263, 265, 267
shift left, 185
shift right, 185
signed arithmetic, 249-255, 256
sign flag, 13, 15, 86, 89, 94, 99, 102, 103, 142, 144, 146, 147, 182, 186, 188, 189, 196, 199
sign bit, 251, 255
SP register, 9, 11-12, 14, 193, 195, 199, 200
special purpose register, 9, 11, 15
STA, 52, 62, 87, 102
stack, 11, 15, 193-200, 207, 209, 210, 266
stack pointer register, see SP register
status byte, 216, 217, 222
status register, 9, 12-13, 15, 87, 193, 195, 200, 210, 266
STX, 87, 102
STY, 87, 102
subroutine, 48, 90, 91, 103, 203-223
subtraction, 36-40, 97-98, 102
system program, 3, 4, 15
symbolic address (also see label), 126, 134-135
system monitor, 123

tape unit, 4
TAX, 56, 88-90, 102
TAY, 88-90, 102
terminal, 4,6
TSX, 193, 199-200
TXA, 88-90, 102
TXS, 193, 195, 200
TYA, 88-90, 102
two's complement, 231, 249-251, 256

unconditional jump (also see JMP), 149

X register, 9, 10, 15, 63, 66, 71, 77, 85, 86. 87, 88, 99, 102, 103, 153, 210

Y register, 9, 10, 15, 66, 71, 77, 85, 86, 87, 88, 99, 102, 103, 210

zero flag, 13, 15, 86, 89, 94, 98, 99, 102, 103, 142, 143, 146, 182, 186, 188, 189, 196, 199
zero page, 64
zero page direct addressing, 64-66, 68, 77, 85, 87, 99, 111, 112, 185, 189