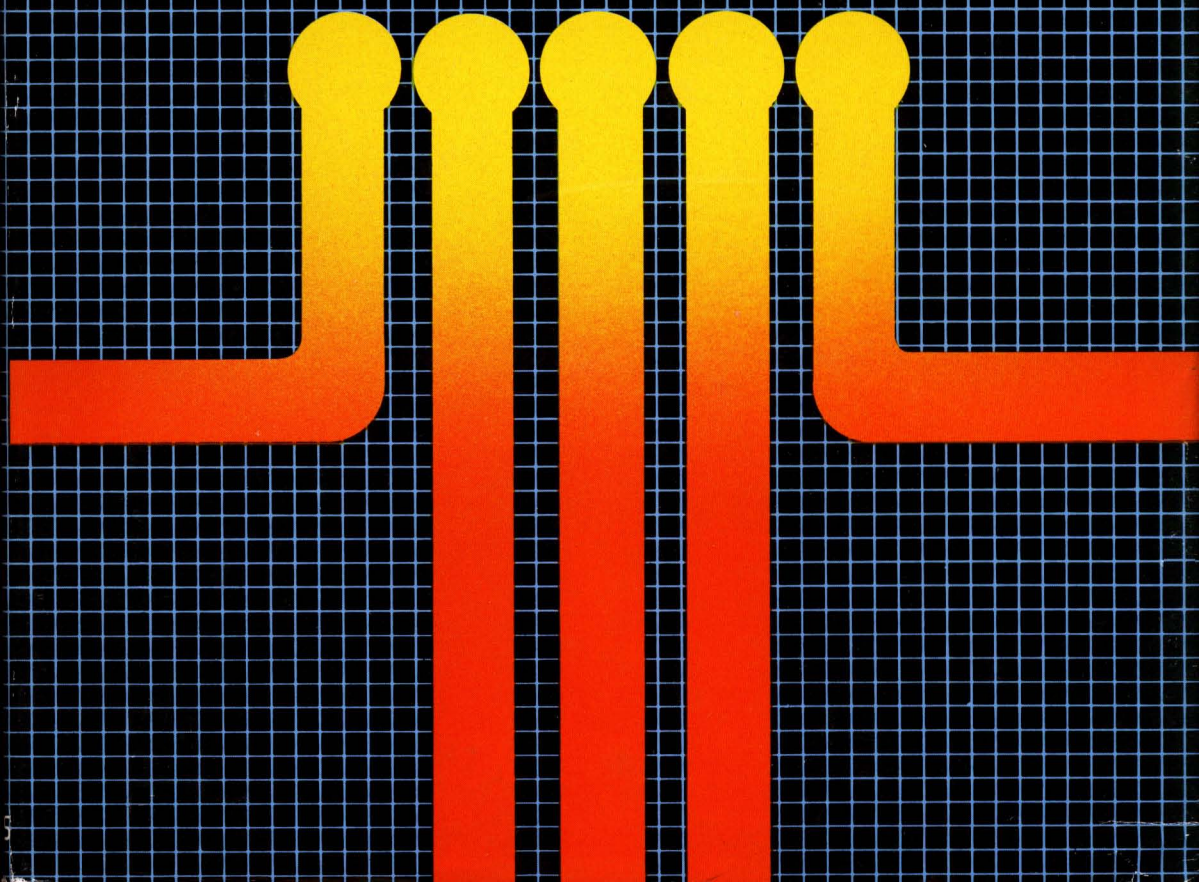
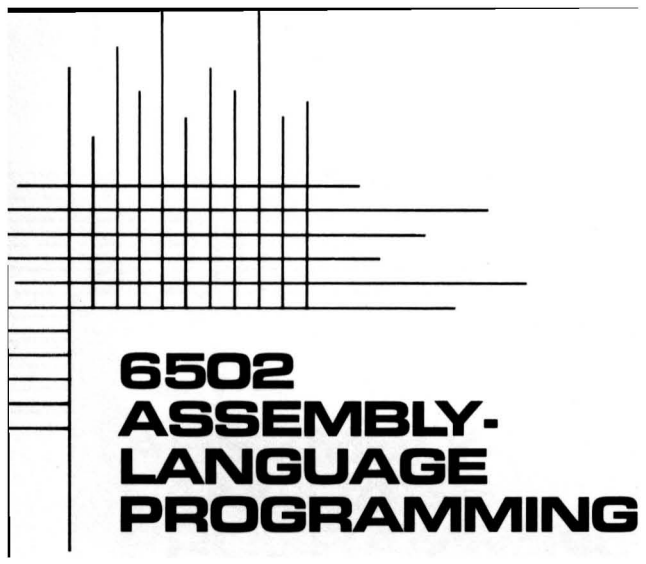
Computer Literacy Skills

CHRISTOPHER LAMPTON

6502 ASSEMBLY-LANGUAGE PROGRAMMING

FOR APPLE, COMMODORE, AND ATARI COMPUTERS





6502 ASSEMBLY-LANGUAGE PROGRAMMING

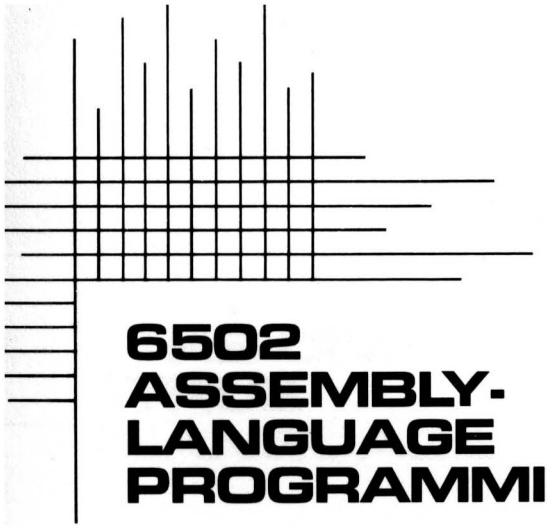
6502 ASSEMBLY-LANGUAGE PROGRAMMING

for Apple, Commodore, and Atari Computers

by Christopher Lampton

A Computer Literacy Skills Book
FRANKLIN WATTS 1985
New York London Toronto Sydney

CONTENTS



6502 ASSEMBLY-LANGUAGE PROGRAMMING

1 THE ONE TRUE LANGUAGE

It may shock you to hear this, but there is only one programming language that is understood by your computer.

Oh, yes, we know that you've been told about BASIC and FORTRAN and Pascal and Logo and any of a number of so-called *high-level computer languages,* with which one can construct the complex lists of instructions known as computer programs. You probably even know how to write programs in one or more of these languages. But, truth to tell, none of these languages is actually "understood" by the computer. They are designed to be understood by human beings. To the computer, programs written in these languages are pure gibberish.

The language that is understood by a computer is called *machine language,* and it is made up of electronic signals, as would befit the language of an electronic device. All computer programs must be written in, or translated into, machine language before they are acceptable to a computer.

High-level computer language—An Englishlike programming language that must be translated into machine language before it can be understood by the CPU of a computer.

Machine language—A programming language that consists of a series of electronic signals that can be executed directly by the CPU of a computer.

When you write a program in the language called BASIC, for instance, it is translated into the machine language of your computer by a second program called the BASIC *interpreter.* This translation takes place while the BASIC program is actually being executed by the computer, which explains why BASIC programs run so slowly. A program written in machine language will execute as much as 300 times faster than an equivalent program written in BASIC.

This book is about machine language. More specifically, it is about an important tool that we use for writing machine-language programs. This tool is called *assembly language.* Assembly language is by far the most powerful and flexible method available for creating computer programs. An assembly-language program can coax a computer into doing anything that it is physically equipped to do, at the top speed of which it is capable.

If assembly language is so powerful, you might ask, why bother with other programming languages at all? The answer is that assembly language requires greater effort from the programmer than high-level languages do. Developing an assembly-language program takes more time than developing a program in a high-level language, and it requires that the programmer keep track of a larger range of details. Nonetheless, if you are interested in coaxing the last ounce of power out of your computer, you may find this extra time and effort worthwhile.

Assembly language also requires a more intimate knowlege of the internal workings of your computer, as opposed to high-level languages, which insulate you from such knowledge. Thus, it requires a greater effort to learn assembly language than to learn a high-level language. There is so much more that you must know before you can even begin to program in assembly language.

Interpreter—A program that translates the instructions of the high-level language into machine language.

Assembly language—A symbolic language in which each instruction represents a single machine-language instruction.

In this book, we will assume from the outset that you are already familiar with at least one high-level programming language. It doesn't matter if this language is BASIC or Pascal or Logo or whatever, though we will occasionally draw examples from the BASIC language to illustrate certain programming principles. What matters is that you be familiar with the essential concepts of programming—loops, subroutines, variables, and so forth—and that you know your way around the computer keyboard, disk drive, cassette recorder, and so on. From that base of understanding, we will be able to take you deeper into the recesses of the computer, where the electronic code that we call machine language reigns supreme.

****

The key to understanding machine language is the *microprocessor.* This tiny chip of silicon is the core of the *central processing unit* (or *CPU*) of a microcomputer; within it reside thousands of interconnected transistors, too small to be seen with a conventional microscope but capable of processing millions of electronic messages in a second. It is the microprocessor that actually "obeys" the instructions in our machine-language programs, turning those instructions into actions. The microprocessor plus the internal memory—more about that in a moment—and a few other electronic chips and parts make up the CPU board of a microcomputer.

This book is concerned primarily with computers based on a microprocessor called the 6502, and other microprocessors (such as the 6510, 6509, and 7501) in its immediate family. These computers include the Apple II series, the Commodore Pet, VIC, 64, and Plus 4, and the Atari 400, 800, and XL series.

A microprocessor alone does not a computer make,

Microprocessor—An integrated circuit that constitutes the major portion of the CPU of a microcomputer.

Central processing unit (CPU)—The portion of a computer where information is manipulated as a series of electronic signals.

however. So it will also be necessary to discuss computer *memory.* Memory is actually a series of electronic circuits. These circuits are designed in such a way that they can store information, albeit in a coded form; they also store machine-language programs while these programs are being executed by the microprocessor. Most computers use two kinds of memory: internal memory, which is immediately available to the CPU but rather limited in the amount of information it can store, and external memory, which is indirectly available to the CPU (and therefore somewhat slower to access) but of potentially unlimited size. External memory devices include cassette recorders and disk drives.

Furthermore, computers use two basic types of internal memory: read-only memory (ROM) and random-access memory (RAM). A primary difference between these two types of memory is that the information stored in RAM can be changed, under the control of a computer program, while the information stored in ROM is placed there in the factory and cannot be changed by programming. ROM is generally used for storing the built-in programs that come with the computer when you buy it (the BASIC interpreter, for instance). RAM is used for storing the transient programs that you buy or store on cassette or floppy disk, or the data that is processed by those programs.

When we combine a microprocessor with memory, we have gone a long way toward creating a computer. Still, there is one element that is missing. Since a computer is a device for processing information, it is necessary to get information into the computer, and to get it back out again once we are finished with it. This process is called *input/output* (*I/O*, for short). It is performed by devices separate from the microprocessor, devices that are often small computers in their own right. These devices, however, operate

---

Memory—A series of electronic circuits, within a computer, capable of storing information as electronic signals.

Input/output (I/O)—The movement of information between the CPU of a computer and the external world.

under the control of the CPU, waiting obediently to follow its commands. Thus, when you program the CPU in machine language, you indirectly program these devices.

The most popular input device on small computers is the keyboard; the most popular output device, the video display. Other I/O devices include disk drives (which are also external memory storage devices), printers, "mice," graphics tablets, and modems.

Microprocessor, memory, and I/O—these are the elements with which the machine-language programmer is chiefly concerned. We shall study these elements in more detail in the chapters to come, as we plunge into the world of assembly-language programming.

2
THE MICROPROCESSOR ZONE

It lies at the heart of your computer, an electronic circuit among electronic circuits—but what a circuit it is! The microprocessor is the brain of your computer, the central hub of a network of information-processing machinery. As much as any piece of electronic equipment is capable of thought, the microprocessor is the part of your computer that does the thinking.

We won't concern ourselves here with the actual physical construction of the microprocessor, the way in which the transistors are wired and interconnected. As programmers, we need only understand those parts of the microprocessor that actually process information. These are the parts that we can control with our machine-language programs. In this view, the most important parts of any microprocessor, including the 6502, are the *registers.*

What are registers? A register is an electronic circuit capable of holding a series of electronic signals. These signals, in turn, may represent information in a coded form. A typical register within the 6502 microprocessor holds either eight or sixteen of these signals. Since each signal can take only two possible forms—a high-voltage signal or a low-voltage signal—we can represent these signals symbolically

Register—Memory circuit within the CPU of a computer in which information can be stored and manipulated.

as 0s and 1s, with each high-voltage signal represented by a 1 and each low-voltage signal represented by a 0. In fact, it is not even necessary for us to think of the microprocessor registers as containing electronic signals; we may imagine, throughout the rest of this book, that they actually contain numbers.

The contents of a typical microprocessor register, represented symbolically, might look like this:

01011110

There is a numbering system, with which you might already be familiar, called *binary*. Unlike the familiar decimal numbering system, which uses ten different digits for counting, binary only uses two digits: 0 and 1. If we counted the fingers of our two hands in binary, we would count like this:

1,10,11,100,101,110,111,1000,1001,1010

Every number in binary has an equivalent in decimal. In the above example, the binary number 11 is equivalent to the decimal number 3, the binary number 111 is equivalent to the decimal number 7, the binary number 1010 is equivalent to the decimal number 10, and so forth.

It is possible, therefore, to view the contents of a microprocessor register as a binary number. The contents of the register shown earlier would translate into decimal as 94, which is equivalent to the binary number 01011110. Thus, we can also view this memory register as containing the number 94.

There are six registers in the 6502 microprocessor, and each has a name. They are called A, SR, X, Y, SP, and PC. The first five—A, SR, X, Y, and SP—hold eight *bits* (*b*inary dig*its*) apiece. Thus, each can contain any number from

---

Binary—A numbering system that uses two different digits, 0 and 1.

Bit—A single digit, either 0 or 1, within a binary number; short for binary digit.

00000000 to 11111111—or 0 to 255 in decimal. Programmers have a special name for an eight-digit binary number; they call it a *byte.* Each of these registers, then, can hold one byte. The other register—PC— can hold sixteen bits, or two bytes. Most operations that we perform with the 6502 microprocessor, however, will involve eight-digit binary numbers. (For this reason, the 6502 is referred to as an eight-bit microprocessor.)

In order for information to be processed by a microprocessor, it must first be translated into numbers (which represent high- and low-voltage signals). We may then request the microprocessor to perform operations on these numbers one (or two) byte(s) at a time. As this book progresses, we shall discuss many different ways in which we may encode information as numbers.

What sort of operations can the microprocessor perform on these numbers? Quite a variety, actually. For instance, we can instruct the microprocessor, in machine language, to add a number in one register to a number stored in the computer's memory, and to leave the sum in one of the registers. Thus, we can use the microprocessor to perform arithmetic. Or we can instruct the microprocessor to move a number from one register into another, from a register to the computer's memory, or from the computer's memory to a register. Thus, we can use the microprocessor to rearrange and redistribute information. Or we can even instruct the microprocessor to manipulate the individual bits within a single register, changing and rearranging them. Thus, we can use the microprocessor to change one number into another number.

The complete set of instructions that we can give to a microprocessor, in machine language, is called the *instruction set* of that microprocessor. Anything that can be done with a given computer can be done using some combination of the instructions in the instruction set of its microprocessor.

Byte—Eight bits.

Instruction set—The set of all machine-language instructions that can be executed by a given CPU.

You will soon see, however, that the instructions in the instruction set tend to be very simple—that is, they instruct the microprocessor to perform very uncomplicated tasks, such as moving a number from one place inside the computer to another. There are no instructions in 6502 machine language for printing a word on the screen of the computer, or for multiplying two numbers together and assigning the result to a variable. In fact, there are no instructions for any number of operations that are quite simple to specify in a high-level language. We can perform all of these operations in machine language, of course, but we must often combine a great many relatively simple machine-language instructions to accomplish a single, seemingly trivial task.

By this time, you must be curious as to what a machine-language instruction looks like. Here, then, is an instruction from the instruction set of the 6502 microprocessor:

```
10101000
```

This instruction tells the microprocessor to move the number stored in the A register into the Y register. You will note, however, that this instruction is in binary. Like all other kinds of information inside a computer, machine language is stored as a series of binary electronic signals. This is fine for the microprocessor, of course, for which binary is the native language. However, it is not so fine for human programmers, who prefer to write programs in a less cumbersome, and easier to understand, form. Thus, it would be best if we found some other method of representing machine-language instructions.

We could, for instance, translate this instruction into decimal, so that it would become:

```
168
```

This is certainly less cumbersome than the binary representation, but it is no easier to understand. We need yet another method of representing machine-language instructions, so that they are easier for human beings to comprehend.

This is where assembly language comes in. Assembly language is a symbolic method of representing machine-

language instructions. In assembly language, we use three-letter words called *mnemonics* (from a Greek word meaning "memory," because they are easy to remember) to represent machine-language instructions. Once we have written a computer program in assembly-language mnemonics, we can use a program called an *assembler* to translate it into actual binary instructions (hence the term "assembly language"). The assembly-language mnemonic for the instruction on page 10 is:

```
TAY
```

The T in this instruction stands for "transfer," the A stands for the A register, and the Y stands for the Y register. Thus, we can see (with a little prompting) that this instruction means "transfer the contents of the A register to the Y register." By this same token, we can also write such assembly-language instructions as TAX ("transfer the contents of the A register to the X register"), TYA ("transfer the contents of the Y register to the A register"), and TXA ("transfer the contents of the X register to the A register"), each of which represents a binary instruction that performs that task. On the other hand, we may not write TYX ("transfer the contents of the Y register to the X register") or TXY ("transfer the contents of the X register to the Y register") because these instructions, though they make logical sense, are not in the instruction set of the 6502 microprocessor and thus have no equivalent binary instructions.

The instructions discussed in the last paragraph are complete within themselves—that is, the operation to be performed and the registers on which it is to be performed are included in the instruction. In many instances, this will not be the case. In order for the instruction to be executed by the microprocessor, you will need to include information in addition to that implied by the mnemonic. This

---

Mnemonics—Symbolic assembly-language instructions that represent single machine-language instructions.

Assembler—A program that translates assembly-language mnemonics into machine-language instructions.

information is sometimes referred to as the *operand* of the instruction. The operand is a value, or the location of a value, on which the operation is to be performed. For instance, suppose we wish to add 15 to the value stored in the A register. We would write the assembly-language instruction for such an operation like this:

```
ADC #15
```

The mnemonic ADC means "add with carry." We'll explain the "carry" part in a later chapter of this book; the "add" should be self-explanatory. The A register is not mentioned specifically in the instruction, but there's no need for it to be. All arithmetic operations are performed on values in the A register. In fact, programmers have a special name for the A register that reflects this fact. It is called the *accumulator,* which is a programming term that means, roughly, "the place where arithmetic is performed."

The number sign (#) indicates that what follows is a number to be added to the value in A. When addition is performed, the result of the addition is stored in the A register, where we may perform further machine-language operations on it if we wish.

When this instruction is assembled—that is, translated into machine language by the assembler—it will look like this:

```
01101001 00001111
```

The 01101001 tells the microprocessor that addition is to be performed on the value in the A register, and that the value to be added will be found immediately after the instruction. And, sure enough, 00001111 is the binary representation of the number 15. As an assembly-language programmer, you will rarely be aware of the actual binary representation of

---

Operand—The data manipulated by machine-language instructions.

Accumulator—The register within the CPU where arithmetic operations are performed.

the instructions you write; nonetheless, it is good to have an idea of how we arrive at these binary representations.

In programming terms, the first byte of the instruction (01101001, or ADC in assembly language) is called the *operation code,* or *op-code,* for short. The second (and, in some cases, third) byte is called the operand, discussed on page 12. We will study further op-codes, their operands, and the ways in which they are used in the succeeding chapters.

First, however, there is one more topic that must be discussed before you embark on your career as an assembly-language programmer: the hexadecimal numbering system. You may groan at the thought of having to learn yet another numbering system—you've already been exposed to decimal and binary in the course of this chapter—but it is necessary, and it will make your work as an assembly-language programmer much easier. Most of the numbers used in the remainder of this book will be in hexadecimal.

Hexadecimal—or *base 16,* as it is sometimes called—uses sixteen different digits to represent numbers. Because the more common decimal numbering system uses only ten digits, it is necessary for us to borrow six digits from another source—the alphabet. The digits of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Taken by themselves, A is equal to 10, B to 11, C to 12, D to 13, E to 14, and F to 15. Once we have counted through F in hexadecimal, we start again at 10, then 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20, and so forth.

Why would we wish to count in hexadecimal, when the more familiar decimal numbering system is readily available? Because there is a close mathematical relationship between hexadecimal and binary that doesn't exist between decimal and binary. This means that it's easier to translate binary numbers into hexadecimal than into decimal. In

---

Operation code (op-code)—The portion of a machine-language instruction that specifies the type of operation to be performed on data.

Hexadecimal (base 16)—A numbering system that uses sixteen different digits.

fact, translations from binary to hexadecimal are surprisingly simple. Programmers can perform the necessary calculations in their heads, should this be required. The following short chart gives you all the information you will ever need to translate binary numbers into hexadecimal numbers:

```
0000—0  0100—4  1000—8  1100—C
0001—1  0101—5  1001—9  1101—D
0010—2  0110—6  1010—A  1110—E
0011—3  0111—7  1011—B  1111—F
```

We won't ask you to memorize this chart, though if you continue programming in assembly language you probably will, even if you don't make a deliberate effort to do so. The chart shows the relationships between all four-digit binary numbers and one-digit hexadecimal numbers: 0010 is equivalent to 2, 1101 is equivalent to D, and so forth. The remarkable thing about this chart, though, is that we can use it to translate binary numbers of any length, not just four digits. We need merely divide the binary number into four-digit chunks, adding leading zeroes to the leftmost chunk if it has fewer than four digits in it.

For instance, suppose we know that a register within the 6502 microprocessor contains the eight-digit binary number 01110011. How can we use the chart to translate this into a hexadecimal number? Simple. We break the number into two four-digit chunks: 0111 and 0011. Looking at the chart, we see that the first chunk is equal to 7 and the second chunk is equal to 3. Thus, the binary number 01110011 is equivalent to the hexadecimal number 73. Translating this number into decimal would be a more complicated operation, and we will not attempt to explain how to do it here. It is similarly easy to program the microprocessor to perform an automatic conversion from binary to hexadecimal, should we need to print the value of a register on the video display of the computer. We'll see how this conversion can be programmed in a later chapter.

It is traditional, when writing programs in 6502 assembly language, to identify hexadecimal numbers by a leading dollar sign, like this:

```
$5AB0
```

This is the convention that we will follow throughout this book. Numbers without a leading dollar sign can be assumed to be decimal, unless stated otherwise.

Because the numbers manipulated by the 6502 microprocessor tend to be either eight bits or sixteen bits in length, the hexadecimal numbers that we use to represent these numbers will be either two digits or four digits in length, with one hexadecimal digit always representing four binary digits (and two hexadecimal digits representing a byte).

Now that we have some idea of how information, such as numbers and instructions, can be represented by the assembly language programmer, let's take a look at how the computer represents information within its own memory.

3
DOWN MEMORY LANE

Quick! What do we call the numbering system that uses only the digits 0 and 1? That's right: binary. You read about it in the last chapter.

And you remembered.

Human memory is an amazing, and perplexing, thing. When you memorize a word or a name or a fact, where do you put it? Where is that information stored? Somewhere within your brain, no doubt, but what form does that memory take? How do you retrieve it? Why is it that you can remember some things more easily than others?

You don't know? Don't fret about it—no one does. The ability of the human mind to record and retrieve information is one of the great mysteries of modern science. No one understands how it works. There are theories, of course, but none fully explains this marvelous property of the brain.

Fortunately, computer memory is a much less complex thing than human memory. It works in very clearly defined ways. It is more accurate than human memory, but it is also less versatile. A computer can remember a number to one thousand-digit accuracy, if necessary, but it cannot remember how wood smoke smells on a cold autumn afternoon. This is fine from the programmer's point of view, but it's certainly the computer's loss.

When we refer to internal computer memory, we are actually speaking of a series of electronic circuits, each capable of holding an eight-digit binary number in electrically coded form, just as the registers in the 6502 micropro-

cessor can hold eight-digit binary numbers. There are machine-language instructions that allow us to move numbers between memory circuits and microprocessor registers and vice versa; thus, the value stored in any memory circuit is directly accessible to the microprocessor.

Each memory circuit within a computer is given an identifying number, called an *address*. The first memory circuit is given an address of 0, the second memory circuit is given an address of 1, the third memory circuit is given an address of 2, and so forth. This convention of identifying memory circuits by addresses gives us the ability to store information in such a way that we can retrieve it when we need it, as we shall see.

Different computers contain different numbers of memory circuits and therefore differ in the range of memory addresses available. Obviously, the more memory circuits the computer has, the more information, and program instructions, it can store. However, there is a limit to the number of memory addresses that can be directly accessible to the microprocessor at any one time. For computers based on the 6502 microprocessor, this limit is 65,536 addresses. Thus, the highest memory address in such a computer cannot be larger than 65,535. (The 65,536th memory address is 0.)

Why this magic limit? Why can't we have more than 65,536 memory addresses if we want to? Well, perhaps a slight change in perspective will make matters clearer. In hexadecimal, the range of addresses accessible to the 6502 are $0000 through $FFFF. That last number—$FFFF—is the largest possible four-digit hexadecimal number. Translated into binary (and you may verify this translation with the chart in the last chapter), it becomes 1111111111111111—the largest possible sixteen-bit binary number. This is no coincidence.

When writing machine-language instructions that access memory, we must specify the exact memory location that we wish to access. In 6502 machine language there are

---

Address—A unique identifying number assigned to a single memory circuit within a computer.

sixteen bits of space within which to specify this address and not a bit more (though, in some cases, less). Thus, even if the computer contained an address greater than $FFFF, there would be no way to specify it, so it might as well not exist.

The PC register of the microprocessor also plays a role here. PC stands for *program counter.* This register always holds a sixteen-bit binary number, which represents the memory address of the next machine-language instruction to be executed. Before executing an instruction, the microprocessor checks the address contained in register PC, then fetches the instruction at that address and executes it. Because the PC register cannot hold a number larger than $FFFF, the microprocessor would not be capable of executing instructions placed higher in memory.

(There is, nonetheless, a special system called *bank-switching* that allows us to place more than 65,536 memory circuits inside a 6502-based computer, though even with bank-switching only 65,536 memory addresses are accessible to the microprocessor *at any one time.* This system allows such computers as the Commodore 64, the Atari 800XL, and the Apple IIe to address larger amounts of memory than the average 6502-based computer.)

Since each memory circuit contains a single byte, we use the byte as a standard measure of computer memory. A *kilobyte,* or *K* for short, contains 1,024 bytes. A computer with a full 65,536 bytes of memory is said to have 64K of memory, because 65,536 divided by 1,024 equals 64.

Let's look at memory from a slightly different perspective. Since most 6502-based computers contain a built-in BASIC interpreter—or a BASIC interpreter that is readily

---

Program counter (PC)—The register within the CPU that contains the address of the next machine-language instruction to be executed.

Bank switching—A method by which more than 64K of memory can be placed in a 6502-based computer.

Kilobyte (K)—A thousand bytes.

available on disk—prepare your computer for programming in BASIC, then type the following BASIC program:

```
10 FOR I=0 TO 32767
20 PRINT PEEK(I);
30 NEXT I
```

Type RUN. A stream of numbers will flow across the screen.

The BASIC function PEEK returns the value of a specified memory address within your computer. Thus, this program cycles through the first 32K of your computer's memory and prints out the contents of each address as a decimal number between 0 and 255. The values are printed in decimal because the PRINT command performs an automatic binary-decimal conversion as it outputs the number. Few versions of BASIC allow us to print numbers in binary or hexadecimal. (More's the pity.)

The distribution of the numbers appearing on your video display may seem quite random (though occasionally it will fall into a monotonous regularity, as the program scans through a segment of memory where no information is stored). Much of what you see, however, has a very specific meaning to the computer.

For instance, machine-language programs themselves—the BASIC interpreter, for instance—must be stored in the computer's memory as a sequence of bytes and will therefore appear as a stream of apparently meaningless numbers when scanned by this program. Of course, if you had a list of machine-language instructions and their decimal equivalents, you might be able to translate those numbers into an actual machine-language program. (Note, however, that there are programs called *disassemblers* that will perform such translations automatically, thereby saving you a great deal of work.) Similarly, the information to be processed by a machine-language program must also be

---

Disassembler—A program that translates machine-language instructions into assembly-language mnemonics.

stored in memory and will also appear as a stream of apparently meaningless numbers.

Some of what you are seeing might be text—that is, strings of alphanumeric characters that have been converted into numeric form for computer storage. A word-processing program, for instance, stores text in the computer's memory as a series of numbers, and the BASIC interpreter stores BASIC programs in similar form.

When we store text in the computer's memory, we use a special coding system called the *American Standard Code for Information Interchange—ASCII* (pronounced Ask ee), for short. The ASCII code assigns a number to each letter of the alphabet, both uppercase and lowercase, and additional codes for numerals, punctuation marks, and so on. For instance, the upper case letter 'A' is represented in ASCII by the decimal number 65 ($41 in hexadecimal), the numeral '1' is represented by the number 49 ($31), the blank space by the number 32 ($20), and so on.

The ASCII code uses the decimal numbers 0 through 127 (hexadecimal numbers $00 through $7F); therefore, an ASCII character will fit neatly into a single, byte-sized memory circuit or microprocessor register, with a little room to spare. (Some computers use the numbers 128 through 255—hexadecimal $80 through $FF—for a special, non-ASCII character set of their own.)

To see the character set of your computer, run the following BASIC program:

```
10 FOR I=32 TO 255
20 PRINT CHR$ (I);
30 NEXT I
```

The BASIC function CHR$ converts a numeric code into the ASCII character represented by that number; this program prints on the video display all the characters represented by codes 32 through 255. We have deliberately

---

American Standard Code for Information Interchange (ASCII)—A coding system that represents alphanumeric characters as numbers from 0 to 127.

avoided characters in the 0 to 31 range because the ASCII code uses these numbers to represent *control characters*—that is, codes for actions that can take place on the video display of the computer and for signals used when two computers exchange information with one another. For instance, 13 represents a carriage return, 8 represents a backspace, 6 acknowledges the receipt of a stream of data, and so on.

You can inspect your own computer's memory for ASCII text by typing the following modification to the memory-scan program given earlier:

```
5 REM KEEP AN EYE OUT FOR THESE WORDS!
10 FOR I=0 TO 32767
20 IF PEEK (I)> 31 THEN PRINT PEEK (I);
30 NEXT I
```

Be sure to include the REM statement in line 5. Run the program and watch the screen of the computer. Instead of numbers, you should see a stream of characters flash past. Most will be gibberish, but occasionally a meaningful pattern will emerge. Before long, you should see the words KEEP AN EYE OUT FOR THESE WORDS; this is the text of our REM statement, stored in the computer's memory as a string of ASCII characters. Oddly, you will not see the word REM. Neither will you see the words FOR, TO, IF, PEEK, THEN, PRINT, or NEXT. On the other hand, you should see the numbers 0, 32767, and 31, as well as the variable name I, the equals sign, and the semicolon. Most BASIC commands (such as REM, FOR, etc.) are invisible because they are stored in "tokenized" form, which is yet another method of encoding information. (That, however, is a subject for another time and another book.) The rest of the program, except for the line numbers, is stored as ASCII text.

So this is how the computer's memory works. All information placed in the computer must be encoded as a sequence of numbers in the 0 to 255 range. Once it is in this

---

Control characters—ASCII code numbers that represent non-printing signals or actions, such as carriage returns or end-of-transmission markers.

form, the information can be easily manipulated and processed by the 6502 microprocessor, which is equipped to perform a wide variety of operations on numbers in this range.

To find out just what operations it can perform, however, we must take a close look at the 6502 instruction set. That is the subject of the next chapter.

4
A 6502 VOCABULARY

Every language must have words; they are the medium of communication, the carrier wave on which information is transmitted.

Machine language is no exception. The "words" of a microcomputer's machine language are the instruction set of its microprocessor. Every program that can be written for a given computer can be written with some combination of these words. They control the actions of the microprocessor as a puppeteer's strings control the actions of a marionette. Once you know the words in the 6502 instruction set, and understand the ways in which those words can be used, you will be able to make the 6502 microprocessor do anything that it is capable of doing.

We can divide these instructions into separate categories, like this:

DATA TRANSFER INSTRUCTIONS—These are instructions that transfer data from one place inside the computer to another. As we mentioned earlier, the 6502 provides instructions for moving values between memory and the microprocessor, between the microprocessor and memory, and between registers within the microprocessor. Examples of the latter type of instruction were shown in Chapter Two:

TAX—Transfer A to X.
TXA—Transfer X to A.
TAY—Transfer A to Y.
TYA—Transfer Y to A.

These instructions move values between the named microprocessor registers. (It would be more accurate to say that these instructions "copy" values between registers, since the *source register*—that is, the register from which the value is being copied—is unchanged by the operation.)

There are two other register transfer instructions available to the 6502 programmer:

TXS—Copy a number from the X register to the SP register.
TSX—Copy a number from the SP register to the X register.

Instructions that copy a number from a memory address to a microprocessor register are called load instructions, because they "load" the register with a number. These instructions all begin with the letters LD (for "load"), followed by a letter identifying the register into which the number is to be loaded. Here are the mnemonics for 6502 load instructions:

LDA—Load a number into the A register.
LDX—Load a number into the X register.
LDY—Load a number into the Y register.

This mnemonic must be followed, in turn, by a number indicating which memory address contains the value to be loaded. There are a number of different methods by which we may specify this address, as we shall see in the next chapter. The simplest is to write the address of the memory location immediately after the mnemonic, like this:

```
LDA $5608
```

This tells the microprocessor to copy the number currently stored at memory address $5608 into the A register.

Source register—The microprocessor register from which data is to be copied by a machine-language instruction.

Similarly, this instruction:

```
LDY $8A3B
```

tells the microprocessor to copy the number currently stored at memory address $8A3B into the Y register.

We may also specify the actual number to be loaded into the microprocessor register, like this:

```
LDX #$A4
```

This loads the hexadecimal number $A4 into register X. The number sign (#) tells us that this is the actual number to be loaded rather than the address of the number to be loaded. (The dollar sign [$], of course, tells us that it is in hexadecimal.)

Instructions that copy a number from a microprocessor register to a memory address are called store instructions, because they "store" the number in memory for safekeeping. These instructions all begin with the letters ST (for "store"), followed by a letter identifying the register that contains the value to be stored. Here are the mnemonics for 6502 store instructions:

STA—Store the number from the A register.
STY—Store the number from the Y register.
STX—Store the number from the X register.

As before, this mnemonic must be followed by a number specifying the address of the memory location where the value is to be stored, like this:

```
STA $0801
```

This tells the microprocessor to store the value in the A register at address $0801.

There are no instructions in the 6502 instruction set that can move numbers between memory addresses. To copy a number from one memory location to another, you must first use a load instruction to copy the number from the first address into one of the microprocessor registers, then use a store instruction to copy the number to the new

address. For instance, to copy a number from address $9C0E to address $3344, you could write the following pair of instructions:

```
LDY $9C0E
STY $3344
```

ARITHMETIC INSTRUCTIONS—These instructions cause the microprocessor to perform arithmetic or arithmeticlike operations on numbers stored in the microprocessor registers or in memory. Here are some examples:

ADC: Adds the value in the A register to the number stored at a specified memory address or to a specified number and loads the result back into the A register. For example, the instruction:

```
ADC $45FF
```

adds the value stored at memory address $45FF to the value currently in the A register and loads the result back into the A register. The instruction:

```
ADC #$F8
```

adds the number $F8 to the value currently in the A register and loads the result back into the A register. Note that a number to be added to the value in A must not exceed eight bits (that is, eight binary digits or two hexadecimal digits).

SBC: Subtracts a value in memory or a specified number from the value in the A register and loads the result back into the A register. For example, the instruction:

```
SBC $DC00
```

subtracts the number stored at address $DC00 from the number currently in the A register and loads the result back into the A register. The instruction:

```
SBC #$0B
```

subtracts the number $0B from the one currently stored in the A register and loads the result back into the A register.

INC: Adds one ("increments") the value stored at a memory location and stores the result back in that memory location. For instance, the instruction:

```
INC $5C93
```

increases the value stored at memory address $5C93 by one. If the number stored at that address was $56 before this instruction was executed, then it would be $57 after this instruction was executed.

INY: Adds one to the value in the Y register.
INX: Adds one to the value in the X register.

If an increment instruction is performed on a memory address or register containing the value $FF (the highest possible value that can be stored in a memory address or register), the value will "roll over" to 0. Note, incidentally, that there is no INA instruction to increment the value in the A register.

DEC: Subtracts one ("decrements") the value stored at a memory location and stores the result back at that location. For instance, the instruction:

```
DEC $ABC9
```

decreases the value stored at address $ABC9 by one.

DEY: Subtracts one from the value in the Y register.
DEX: Subtracts one from the value in the X register.

If a decrement instruction is performed on a memory address or register containing the value 0, the value will "roll over" to $FF. As before, there is no DEA instruction to decrement the value in the A register.

AND: Peforms a logical AND on the value in the A register and the value stored at a memory location and loads the

result back into the A register. AND is a *logical operation* that compares the individual bits in the A register with the bits in the memory location and sets the bits in the result accordingly. If a given bit in the A register is a 1 *and* the corresponding bit in the operand location is a 1, then the corresponding bit in the result will be a 1. On the other hand, if one bit is a 0 and the other bit is a 1, or if both bits are 0s, then the corresponding bit in the result will be a 0. For instance, the instruction:

```
AND $4C50
```

performs a logical AND between the A register and the value stored at address $4C50 and stores the result back in A. The instruction:

```
AND #$77
```

performs a logical AND between the value stored in the A register and the number $77.

We use the AND instruction primarily to "zero out" selected bits within an eight-bit number and retain the value of other bits, a process called *masking*. The key to masking a number is this: any bit that is ANDed with a 1 will retain its previous value, while any bit that is ANDed with a 0 will become a 0. Suppose, for instance, that we wish to change the first four bits of the number stored at address $9999 to 0s, but we want to leave the rest of the number unchanged. With the aid of the AND instruction, this is a fairly simple matter:

```
LDA #$0F
AND $9999
STA $9999
```

Logical operation—An operation that manipulates individual bits within a binary number according to predetermined rules.

Masking—The selective zeroing of individual bytes within a binary number.

First we load the hexadecimal number $0F into the A register. In binary, $0F looks like this: 00001111. Then we AND the A register with address $9999. Suppose, for the sake of example, that memory address $9999 contains the number $A9. In binary, $A9 looks like this: 10101001. When the first digits of the two numbers—0 in the A register and 1 at address $9999—are ANDed, the corresponding digit in the result becomes 0, because a 0 ANDed with a 1 produces a 0. Similarly, the second digit of the result becomes a 0, because a 0 ANDed with a 0 also produces a 0. In fact, *any* bit ANDed with a 0 becomes a 0, so the first four digits of the result are all 0s. On the other hand, the fifth digit of the result becomes a 1, because a 1 ANDed with a 1 produces a 1. And the sixth digit becomes a 0, because a 0 ANDed with a 1 produces a 0.

The final result of the AND operation is the number 00001001. This result is then stored back at location $9999. Note that the first four digits of the number stored at address $9999 have been reduced to 0s, as we had intended, but that the last four digits have emerged unscathed, also as we intended. Once again, any number ANDed with a 0 will become a 0, while any number ANDed with a 1 will remain as it was.

ORA: Performs a logical OR on the value in the A register and the value stored at a specified memory location, and loads the result back into the A register. OR, like AND, is a logical operation that compares the individual bits within two binary numbers and sets the bits in the result accordingly. If a given bit in one number is a 1 *or* the corresponding bit in the other number is a 1 *or* if both are 1s, then the corresponding bit in the result is a 1. Only if both bits are 0s is the corresponding bit in the result a 0. For instance, the instruction:

```
ORA $4000
```

performs a logical OR on the value in the A register and the value stored at memory location $4000, and stores the result back in the A register.

The instruction:

```
ORA #$34
```

performs a logical OR on the value stored in the A register and the number $34, and stores the result back in the A register.

We use the logical OR primarily to change selected bits within a number into 1s—the opposite of the AND operation, which can change selected bits to 0s. Any number ORed with a 1 will become 1, while any number ORed with a 0 will retain its previous value—the opposite of the AND operation.

For instance, suppose we wish to change the third digit of the number stored at address $0405 to a 1 without affecting any of the other digits. We could achieve this change by using the following series of instructions:

```
LDA #$20
ORA $0405
STA $0405
```

First we load the A register with the number $20. In binary, the number $20 looks like this: 00100000. Note that only the third bit is a 1; all other bits are 0s. We then OR this value with the number stored at address $0405. Suppose, for the sake of example, that the number stored at $0405 is $9C. In binary, $9C looks like this: 10011100. When we OR the first digit in A with the first digit at address $0405, the corresponding bit in the result is 1, because 0 ORed with 1 is 0. When we OR the second digit in A with the second digit at address $0405, the corresponding bit in the result is 0, because 0 ORed with 0 is 0. Then, when we OR the third digit in A with the third digit at address $0405, the corresponding bit in the result is 1, because 1 ORed with 0 is 1. Thus, the final result of ORing $20 and $9C is $BC, or 10111100, which we store back at $0405. All bits in the result are the same as the original bits at $0405, except for the third bit, which has become a 1. Once again, all bits ORed with 1 become (or remain) 1s, while all bits ORed with 0s remain as they were.

EOR: Performs an Exclusive-OR on the value in the A register and the value stored at a memory location and loads the result back into the A register. Exclusive-OR is a logical operation that compares the individual bits within two binary numbers and sets the bits in the result accordingly. If

a given bit in one register *or* the other register is a 1 or if the bit in one register *or* the other register is a 0, then the corresponding bit in the result is set to 1. But if both bits are 1s or both are 0s, then the corresponding bit in the result is set to 0. The instruction:

```
EOR $411D
```

performs an exclusive-OR on the value in A and the value stored at address $411D, and loads the result back into the 0 register. The instruction:

```
EOR #$45
```

performs an exclusive-OR on the value in A and the number $45, and loads the result back into the A register.

The exclusive-OR is commonly used to "invert" a bit or bits in a memory location—that is, to turn a 1 into a 0 or a 0 into a 1. Any digit that is EORed with a 1 will be inverted. Any bit that is EORed with a 0 will be unchanged. To invert *all* the bits in a number, we EOR the number with $FF (which is 11111111 in binary). If you wished to invert all the bits in the number stored at address $C91C, you would write the following series of instructions:

```
LDA #$FF
ORA $C91C
STA $C91C
```

If the number stored at $C91C was $95 (or 10010101) before these instructions were executed, then it would be $6A (or 01101010) after these instructions are executed. This is often useful in graphics operations, where light and dark points on the screen are represented by 0s and 1s, respectively, in memory. EORing these bits with $FF will produce a negative image on the video display.

ASL: Arithmetic Shift Left. Shifts every binary digit in the 0 register or a specified memory address one position to the left and forces a 0 into the rightmost digit position. For instance, the instruction:

```
ASL A
```

will shift every digit in the A register one position to the left. If the A register contains the number $D7 (or 11010111) before this instruction is executed, then it will contain the number $AE (or 10101110) after it is executed. Note that the digit in the leftmost position disappears completely after the shift takes place (though it is not completely lost, as we shall see in the chapter on arithmetic). Similarly, the instruction:

```
ASL $898A
```

will shift every digit at memory address $898A one position to the left.

We use the ASL instruction primarily for two purposes. One is the deliberate isolation of specific bits within a number: We simply shift the undesirable bits off the end of the register (or memory location) and into oblivion. We can do this with an AND instruction as well, of course, but the ASL instruction can help us relocate the isolated bits into a more desirable position. We'll see examples of this later.

The second purpose of ASL is multiplication—shifting the digits in a binary number one position to the left is equivalent to multiplying that number by 2, just as shifting the digits in a decimal number to the left is equivalent to multiplying that number by 10. Since there are no multiplication instructions in the 6502 instruction set, we must often resort to such indirect means of performing math.

LSR: Logical Shift Right. Shifts every binary digit within the A register or a specified memory address one position to the right and forces a zero into the leftmost digit position. For instance, this instruction:

```
LSR A
```

will shift every digit in the A register one position to the right. If the A register contained the number $35 (or 00110101) before this instruction was executed, then it would contain the number $1A (or 00011010) after it is executed. The rightmost digit disappears off the end of the register. Similarly, the instruction:

```
LSR $0AAA
```

shifts every digit at address $0AAA one position to the right.

We use the LSR instruction primarily for two purposes. The first is the isolation of digits within a number, as described on page 34. The second is division—shifting the digits in a binary number one position to the right is equivalent to dividing that number by 2, just as shifting the digits in a decimal number to the right is equivalent to dividing that number by 10. As you might guess, there is no division instruction in the 6502 instruction set.

ROL: Rotate Left. Shifts every digit in the A register or a specified memory location one position to the left, but also rotates the contents of the carry flag into the rightmost digit position. We'll explain this in more detail after we study the carry flag. For instance, this instruction:

```
ROL A
```

shifts every digit in the A register one position to the left and rotates the contents of the carry flag to the rightmost position.

ROL instructions are commonly used in conjunction with ASL instructions, to produce shift operations that extend across more than one memory location.

ROR: Rotate Right. Shifts every digit in the A register or a specified memory location one position to the right and rotates the contents of the carry flag into the leftmost position. Operation and use are essentially identical to the ROL instruction.

CMP: Compares the value in the A register with a value in memory. Actually, the CMP instruction is a simulated subtraction instruction but produces no numeric result (and does not affect the value in A). The sole purpose of the CMP instruction is to affect the value of individual bits in the 6502 status register (SR) and thus obtain information about the nature and relative size of the two numbers. More about this in the chapter on decision-making.

CPX: Compares the value in the X register with a value in memory.

CPY: Compares the value in the Y register with a value in memory.

BRANCH AND JUMP INSTRUCTIONS—These instructions transfer control of the microprocessor from one portion of a machine-language program to another. They function in much the same manner as GOTO instructions in BASIC, except that they pass control to an instruction at a specified memory address rather than to an instruction on a specified program line.

The simplest of the branch and jump instructions is JMP, which stands (as you may have guessed) for "jump." The JMP instruction transfers control to the instruction at a specific address, which must be placed immediately after the JMP instruction, like this:

```
JMP $3990
```

This will cause the microprocessor to begin executing instructions beginning at memory address $3990.

What happens when no program instructions have been placed at that address? The program will crash (i.e., the computer may freeze up and refuse to respond), and you'll have a problem. It is the responsibility of the programmer to make sure that jump instructions always transfer control to a location at which appropriate instructions have been stored. In the chapter on using the assembler we'll take a look at how this task is facilitated by the use of labels and equates.

Another commonly used jump instruction is JSR—Jump to Subroutine—which, as the name implies, passes program control to a subroutine. This is almost precisely equivalent to the BASIC instruction GOSUB, except that (once again) it passes control to the instruction at a memory address rather than to a program line. The transfer of control is not permanent, however. As soon as an RTS (Return from Subroutine) instruction is encountered, program control will return to the instruction immediately following the JSR instruction. RTS is equivalent in function to the BASIC instruction RETURN.

When you write a JSR instruction, it must be followed by the address to which program control is to be transferred, like this:

JSR $C09F

This passes control to a subroutine beginning at memory address $C09F. When an RTS is encountered, control will return to the instruction following the JSR instruction.

There is one exception to this rule: If another JSR instruction is encountered before the RTS, control will be passed to the specified subroutine. When an RTS is subsequently encountered, it will return control to the instruction following the *second* JSR. The *next* RTS will return control to the instruction following the original JSR.

In fact, any number of JSR instructions (within certain limitations) may be encountered before an RTS is encountered. An RTS instruction will always return control to the most recent JSR that has not yet been matched with an RTS. This is consistent with the way in which GOSUB and RETURN instructions are used in BASIC. When a subroutine is called from within a subroutine in this manner, we say that the subroutines are nested.

You might wonder how the 6502 microprocessor always knows what address to return to when an RTS is encountered. It is not absolutely necessary for a machine-language programmer to understand this process, but it can be helpful, especially inasmuch as certain other machine-language instructions (to be discussed in a moment) have the potential to interfere with this process if used indiscriminately.

There is a special area of the computer's memory, located between addresses $0100 and $01FF, called the *stack*. The 6502 microprocessor uses this area for temporary storage of numeric values, including the return addresses associated with subroutine calls. When a JSR instruction is executed, the address of the instruction *following* the JSR is placed in the stack by the 6502. When an RTS is encountered, the 6502 retrieves this address.

This seems simple and straightforward enough, but what if more than one JSR instruction has been executed before the RTS is encountered? In that case, there will be

---

Stack—A portion of internal memory set aside for the LIFO ("last-in first-out") storage of data.

more than one return address in the stack. How does the 6502 know which one to retrieve?

The secret is in the SP register. SP is short for *stack pointer*; as its name implies, the SP register "points" at the stack. That is, it always contains the address within the stack at which the next return address will be stored, assuming an RTS instruction is not encountered in the meantime. For instance, if the SP register currently contains the number $E4 and we execute a JSR instruction, the two bytes of the return address for that JSR will be placed in the stack at address $01E4 (the address "pointed" to by the stack pointer) and address $01E3 (the address immediately before it). The SP register will then automatically be decremented twice so that it will point at address $01E2, the address at which the *next* return address will be stored. If another JSR is encountered before an RTS is encountered, the return address for that JSR is placed at $01E2, and the SP register will be decremented to point at $01E0.

Suppose, now, that an RTS is encountered. The 6502 will increment the value of the SP register twice, to point at the most recent return address. It will fetch the address from the stack and return program control to that address. When the second RTS is encountered, the 6502 will increment the SP register yet again, to point to the earlier return address. It will then return to *that* address.

The point of using a stack is that the most recent value placed on the stack must always be the first value removed from it—what programmers call a LIFO ("last-in first-out") structure. This ensures that RTS return addresses are always executed in the correct order. Note, however, that the stack is only capable of holding 128 return addresses at one time. Thus, it is impossible to nest subroutines more than 128 levels deep. (It is unlikely that you will ever need to nest subroutines this deeply, however.)

As we shall see in a moment, there are instructions that allow the programmer to access the 6502 stack directly. These instructions should be used with great care, however,

---

Stack pointer—The register that contains the address within the stack at which the next value will be stored.

because they can alter the sequence of subroutine return addresses and thereby affect the proper execution of a program. The TXS and TSX instructions, described earlier in this chapter, can also alter the sequence of the stack and should only be used to set an initial position for the stack, at the beginning of a program, before any JSR instructions are executed.

The remaining jump and branch instructions fall into the category of branch instructions. Although superficially similar to jump instructions (they transfer control to specified memory addresses), there are some essential differences in the way in which branch instructions are executed. Branch instructions are both more limited and more versatile than jump instructions.

Branch instructions all involve the use of the 6502 status register. We will discuss the status register in some detail in the chapter on arithmetic, and we will discuss branch instructions in considerable detail in the chapter on decision-making. Here, however, is a list of the 6502 branch instructions and their meanings, which may seem a trifle obscure until we can explain them at greater length:

BEQ—branch on equal
BNE—branch on not equal
BCC—branch on carry clear
BCS—branch on carry set
BMI—branch on minus
BPL—branch on plus
BVS—branch on overflow set
BVC—branch on overflow clear

The above may look like complete gibberish, but a quick perusal of the list may at least tip you to an important fact. Branch instructions are, in essence, jump instructions that can be made contingent on other events taking place within the microprocessor. For instance, the BEQ (branch on equal) instruction will only be executed if the result of the last arithmetic operation was equal to zero; otherwise it will be completely ignored by the computer.

In this manner we can create machine-language IF-THEN statements—complete sequences of instructions that will be executed or not executed depending on whether a branch instruction is executed. We'll explain this process in

considerable detail later; it is one of the most important principles of machine-language programming.

The other important thing you should know about branch instructions is that they are extremely limited in the distance over which they can branch. A jump instruction can be used to transfer control to any instruction within the addressable memory of the computer. A branch instruction will only transfer control over a range of 256 addresses.

Why is this? When a jump instruction is translated into machine language by the assembler, the address to which control is to be transferred is translated into an actual sixteen-bit number. For instance, suppose we write the following instruction:

```
JMP $8392
```

Once assembled into machine language, this instruction will look like this:

```
01001100 10010010 10000011
```

In hexadecimal, it would look like this:

```
4C 92 83
```

The $4C is the machine-language representation of the JMP instruction. The following two bytes are the actual jump address. (Yes, the two halves of the address are in reverse order. This is a rather odd convention, observed by certain eight-bit microprocessors, that will be explained in more detail in the next chapter.)

When we write a branch instruction, on the other hand, the actual branch address is *not* translated into machine language. Rather, the assembler calculates the distance from the current instruction to the destination instruction and places this distance (called a *displacement*) into the program. The displacement is an eight-bit number representing the number of memory addresses over which the

---

Displacement—A number representing the range over which a branch instruction is to transfer control.

branch will jump. For instance, if the destination of the branch is 24 addresses after the address of the branch instruction itself, the displacement would be 24—and this number will be placed directly after the machine-language representation of the branch instruction itself. (Actually, it will be 22, for reasons to be explained in a moment.)

To give one example, suppose we write the following instruction:

```
BEQ $4040
```

Suppose this instruction resides in the computer's memory at address $4000. If the branch is executed, it will transfer control to an instruction $40 memory addresses beyond the branch instruction. We can guess, then, that the displacement for this branch will be $40. Well, not quite. Actually, the displacement is calculated beginning with the instruction immediately following the branch instruction, which will be at address $4002; thus, the displacement will be $3E. In hexadecimal, the above branch instruction will look like this:

```
F0 3E
```

where $F0 is the machine-language representation of BEQ and $3E is the displacement.

How is the displacement calculated if the branch instruction branches to an instruction *before* the current address? All displacement values greater than $7F are treated by the 6502 as *negative* displacements. A displacement of $FF branches backward one address, a displacement of $FE branches backward two addresses, and so on. The maximum possible forward displacement is 129 ($81) addresses from the address of the branch instruction; the maximum possible backward displacement is 126 ($7E) addresses before the address of the branch instruction. For any greater distance, a JMP instruction is required.

If all this information is a little hard to assimilate, relax. The assembler will take care of most of these details for you. However, should you receive a BRANCH OUT OF RANGE error when the assembler attempts to translate your code, you will know that you attempted to branch outside of the allowable number of memory addresses.

As promised, we will discuss the uses of branch instructions in more detail in a later chapter.

STACK INSTRUCTIONS—There are only four of these. They allow the programmer to store eight-bit values in the 6502 stack. For a discussion of how the stack operates, see the description of JSR instructions earlier in this chapter. When an eight-bit value is placed in the stack, the value of the SP (stack pointer) register is decremented by 1; when an eight-bit value is removed from the stack, the value of the SP register is incremented by 1. When we place a value on the stack, we say that we have "pushed" that value. When we take a value off the stack, we say that we have "popped" or "pulled" that value.

Here are the 6502 stack instructions:

PHA: Push the contents of the A register onto the stack.

PHP: Push the contents of the processor status register (SR) onto the stack.

PLA: Pull an eight-bit value from the stack into the A register.

PLP: Pull an eight-bit value from the stack into the processor status register.

Advanced machine-language programmers have many uses for the stack. As a novice programmer, you will use the stack primarily for temporary storage of the value in a register while the register is used briefly for another purpose (usually while we call a subroutine). Storing a value on the stack is faster and more efficient than storing it in a memory address.

****

These are the major categories of 6502 instructions. Here are a few miscellaneous instructions that may prove useful from time to time:

BIT: Allows us to test the value of individual bits within a specified memory location.

BRK: Transfers control of the microprocessor to a prearranged memory address. Used to aid in the debugging of computer programs.

CLC: Affects a specific bit within the status register (SR), called the carry flag. More about this later.

SEC: Similar to the above instruction, but with a slightly different effect.

CLV: Affects a specific bit within the status register, called the overflow flag. We will not be discussing the overflow flag in this book, except in passing.

CLI: Affects the manner in which interrupts are recognized. We will not be discussing interrupts in this book.

SEI: Similar to the above, with a different effect.

SED: Prepares the microprocessor for decimal arithmetic. We will not be discussing decimal arithmetic in this book.

CLD: Disables decimal arithmetic.

NOP: No operation. Does nothing except waste time. Used when precise timing is necessary.

And that is that—the complete instruction set of the 6502 microprocessor. Quite a mishmash, isn't it? You may be left wondering how anyone, even the most experienced of machine-language programmers, can produce anything useful out of that set of instructions.

That, of course, is what the rest of this book is about.

5 ADDRESSING MODES

As we have seen, many instructions in the 6502 instruction set require that the programmer specify a memory location that will serve as the source or destination of eight-bit data. For instance, if we wish to load a microprocessor register from a memory address, we must not only specify which register is to be loaded (a specification that is implicit in the instruction itself, as in LDA or LDX or LDY), but we must specify the memory address from which the data is to be copied. So far we have done this in the simplest of manners. We have written the address itself immediately after the instruction, like this:

```
LDY $4554
```

You may think that this is a very sensible way of specifying an address—and you would be correct. However, the 6502 offers a large number of alternative ways in which we can specify memory addresses after an instruction, and some of these other methods are also very useful. In fact, some operations would be impossible (or at least extremely cumbersome) without these alternative methods. The various methods by which addresses can be specified are called the *addressing modes* of the 6502.

Addressing modes—The methods by which the location of data to be manipulated can be specified.

The following is a list of addressing modes, with a description of each:

ABSOLUTE ADDRESSING—This is the form that we have been using so far on all instructions that directly specify memory addresses. In absolute addressing, the source or the destination of the data is expressed as a sixteen-bit number placed immediately after the op-code. For instance, if we wish to copy the value in the A register to memory address $EC0B, we would write:

```
STA $EC0B
```

When assembled into machine language, this instruction will look like this:

```
10001101 00001011 11101100
```

Or, in hexadecimal:

```
8D 0B EC
```

The $8D part of this instruction tells the microprocessor that it is to store the value of the A register (STA) and that absolute addressing is being used. (The machine-language representation of an instruction such as STA changes depending on the addressing mode being used. This informs the microprocessor what type of operand it should expect.)

The $8D is followed by the address that we specified in the assembly-language instruction. Note that the address is two bytes—sixteen bits—long. When this machine-language instruction is stored in the computer's memory, prior to execution by the microprocessor, the two bytes of the address will be stored in consecutive memory addresses. Thus, the complete instruction will take up three bytes of memory. This is true of all instructions that use absolute addressing. One byte will be used for the instruction itself and two for the address. Note also that the two bytes of the address are in reverse order. This is a peculiar convention observed by most eight-bit microprocessors: When addresses are stored in memory, the bytes are always reversed. We'll point out other instances of this convention later in this chapter.

With a few exceptions, which we will note at the appropriate moment, all 6502 instructions that require a memory address as an operand can use absolute addressing. Here is a list of the instructions with which absolute addressing can be used: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY.

ZERO-PAGE ADDRESSING—Zero-page addressing is a special case of absolute addressing that can only be used when the address to be specified is in the range 0000 to 00FF, a range of 256 addresses. This range of memory is sometimes referred to as the *zero page.* (A page is 256 bytes of memory, beginning at an address that is evenly divisible by 256. The zero page begins at address $0000. Page one begins at address $0100, and so on.) Note that all addresses in this range begin with a byte of $00. Thus, it is possible to specify a zero-page address using an eight-bit number, with the first eight bits understood to be $00. There are certain advantages to doing this, as we shall see.

A zero-page instruction may be written exactly as any other instruction, but the assembler will recognize that the zero page is being referenced and will translate the instruction into machine language somewhat differently. For instance, suppose once again that we wish to store a value from the A register into a memory location. This time, it is a memory location on the zero page, say $0045. We may write the instruction like this:

```
STA $0045
```

or, alternatively:

```
STA $45
```

Both will be translated in the same manner, into this machine-language instruction:

```
10000101 01000101
```

Zero page—The first 256 bytes of a computer's memory, from address $0000 to $00FF.

or this hexadecimal instruction:

```
85 45
```

Note that the machine-language representation of STA is slightly different this time. This signals the 6502 that zero-page addressing is being used. The $45 tells the 6502 in what address on the zero page to store the contents of A.

There are two advantages to zero-page addressing. Because the address requires only one byte rather than two, programs that use zero-page addressing take up less room in the computer's memory. And, because shorter instructions tend to execute more quickly than longer instructions, programs that make use of zero-page addressing will run faster than those that do not.

As noted before, most assemblers will assume that zero-page addressing is being used if your instruction references an address on the zero page; thus, you need not give any special indication of the type of addressing desired. However, if you wish to take advantage of zero-page addressing, you should make an attempt to use zero-page storage locations whenever possible. Of course, many zero-page locations on your computer will probably be used by your operating system, so you must be careful that there is no conflict. We will study the operating system in the chapter on input/output.

Most instructions that can use absolute addressing can also use zero-page addressing; JMP and JSR are the exceptions. The instructions that can use zero-page addressing are: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY.

INDEXED ADDRESSING—Now things start getting a little tricky. The X and Y registers of the 6502 are known as the *index registers.* They may be used, through indexed addressing, to help specify an address in an assembly-language instruction. When indexed addressing is used, a six-

---

Index registers—Registers within the CPU that can be used in indexed addressing, to aid in the calculation of the effective address.

teen-bit address is placed immediately after the op-code, as in absolute addressing. This address is followed by a comma (,) and the name of one of the two index registers, like this:

```
LDA $5600,Y
```

The actual (or "effective") address referenced by this instruction is $5600 *plus* whatever number is currently in the Y register. Thus, if the Y register contains $98, this instruction will tell the 6502 to load the A register with the value contained at memory address $5698. Therefore, we say that the effective address of this instruction is $5698.

Why would we use indexed addressing? Well, suppose that you have 256 bytes (or less) of data stored in consecutive memory addresses, and you wish to process that data with a minimum of fuss. If that data begins at address $5600, the above instruction could be used to access every byte of that data simply by changing the value stored in the Y register. It is not necessary to write a separate instruction to get every one of the 256 bytes when one indexed instruction will do.

An especially common use for indexed instructions is the processing of strings of characters stored in memory. As long as the string is no longer than 256 bytes, a single indexed load instruction can be used to move the string byte by byte into the microprocessor preparatory to sending each byte to an output device, such as the video display. We'll describe this technique in more detail in the chapter on input/output.

Not every instruction that makes reference to a memory address can use indexed addressing. For instance, the following instruction is clearly illegal:

```
LDX $0980,X
```

This is because the destination register and the index register are both X, a situation that would be extremely limited (and confusing) in application. The following instructions can be used with the X register as index: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA.

The following instructions can be used with the Y regis-

ter as index: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA.

Indexed addressing may be used with zero-page addressing (that is, a zero-page address plus an index value), but in this case only the X register may be used as an index. The following instructions can be used with indexed zero-page addressing: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, STY.

INDIRECT ADDRESSING: There is only one instruction in 6502 machine language that can use the indirect addressing mode, but the concept of indirect addressing is echoed in several other 6502 addressing modes, so it is worth talking about in some detail. With indirect addressing, the actual address of the operand is not given in the instruction. Rather, the instruction contains the address of a memory location that in turn contains the address of the operand.

For instance, suppose the value that we wish to manipulate with our instruction is stored at memory address 5044. With indirect addressing, we would first store the number 5044 in two consecutive memory addresses, then use the first of those addresses as the indirect address in the instruction itself. If this seems less than clear, don't worry. It's hard to understand indirect addressing without first seeing an example or two.

The only instruction in the 6502 instruction set that can use true (or "absolute") indirect addressing is the JMP instruction. As you'll recall from the last chapter, JMP is used to transfer control of the program to the instruction located at a specified memory address. Ordinarily, we would specify this memory address immediately after the JMP instruction. For instance, if we wished to transfer control to the instruction at address $7500, we would write JMP $7500.

There will be times, however, when we do not know at the time the program is written just what address we wish to transfer control to. We may want the program to calculate this address at execution time. Once this calculation is performed, we can instruct the microprocessor to store the address in memory, in standard reversed order. If the location at which we store this address is $A890, we may then transfer control to that address with this instruction:

```
JMP ($A890)
```

The parentheses tell us that indirect addressing is in effect. Suppose that address $A890 contains the number $40 and that the address immediately following $A890 (which will be $A891) contains the number $10. The above instruction will then transfer control to the instruction at address $1040, because this is the indirect address stored beginning at address $A890.

When one location in a computer's memory contains the address of another location in the computer's memory, we say that the first location "points" to the second. Thus, indirect addressing allows us to create pointers in memory. Such pointers have many uses. For instance, if a piece of data (or the location at which we want to store a piece of data) tends to "wander" through the computer's memory, we may wish to establish a pointer to that data, which can be changed so that it always identifies the current address of that data, should we need to access the data in a hurry.

Although the JMP instruction is the only 6502 instruction that uses true indexed addressing, there are many instructions that use variations on this addressing mode.

INDIRECT INDEXED ADDRESSING: This is one of the most valuable addressing modes offered by the 6502; it is also one of the most complicated. You may even want to read this section twice, if you find it difficult to grasp the first time.

As the name implies, indirect indexed addressing bears some similarities to both indexed and indirect addressing. In fact, it neatly combines the two modes. An indirect indexed instruction looks like this:

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

The , Y tells us that the instruction is indexed; the parentheses tell us that it is indirect. The $2A inside the parentheses is a zero-page address. It is not, however, the address of the operand, even when the index value has been added in. Rather, $2A is the address at which we can find the address of the operand.

Suppose that zero-page address $002A contains the number $14. And suppose that zero-page address $002B

(the next address in sequence after $002A) contains the number $71. Observing the convention that addresses are stored backward in memory, we can look at these two memory locations as containing, between them, the address $7114. And this is exactly how the 6502 will look at these addresses when we write the instruction on page 51. STA ($2A) , Y tells the microprocessor to store the value of A at an effective address computed by taking the address stored at zero-page locations $002A and $002B, then adding in the value of the Y register. If we make things easier by assuming that the value in the Y register is 0, the effective address would be $7114, or in this example, equivalent to STA $7144,Y. If addresses $002A and $002B contained $A3 and $30, it would be equivalent to STA $30A3,Y.

Why in the world would we want to use an addressing mode this complex? Well, it's possible that at the time we write an assembly-language instruction we will not know what address we wish that instruction to obtain its data from. This information might not be known until the program is actually running. Thus, we can have the program itself compute this address and store it in a zero-page location, then use indirect addressing to tell the microprocessor where on the zero page the actual address is stored.

This is similar to the way in which we use arrays in languages such as BASIC. For instance, the variable A(X) in BASIC can be made to refer to any of a large number of storage locations within the computer simply by altering the value of the variable X. Thus, arrays are used to process large strings of information in a very efficient manner. The indirect address in an instruction such as CMP ($40) ,Y can be looked on as a kind of array subscript, where the array is the entire memory of a 6502-based computer.

By changing the value stored at location $0040, we can change the effective address of the instruction and thus refer to a different element in the array. We can also change the value in the Y register to process data over a range of 256 bytes. However, it is not unusual for a programmer to use indirect indexed addressing with a value of 0 in the Y register, and never change the value in the Y register in the course of processing data.

Note that there are a couple of serious limitations to indirect indexed addressing—for example, the indirect address must be stored at a zero-page location and the

index register used must be Y. Also, only a limited range of instructions may be used with indirect indexed addressing. These are: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

INDEXED INDIRECT ADDRESSING: No, we're not repeating ourselves. The name of this addressing mode is not the same as the name of the last one. The first two words have been reversed.

In indexed indirect addressing, the indirect address itself is indexed. That is, to calculate the actual address of the data, you must first add the index to the zero-page address to compute a new zero-page address. That address, in turn, will contain the address of the data.

Confused? This example should help:

```
SBC ($A4,X)
```

Note that the index notation (,X) is now inside the parentheses rather than outside them. This indicates that the index value is to be added to the zero-page address itself, rather than to the address contained at that address.

Suppose that zero-page address $00A4 contains the number $3D. We add this to address $00A4, which produces an address of $00D1. This is the address that actually contains the indirect address. If address $00D1 contains the number $67, and address $00D2 contains the address $B4, then the effective address of the instruction is $B467, and it is at this address that we will find (or place) our data.

Indexed indirect addressing allows us to store entire tables of addresses on the zero page. However, because zero-page space is at a premium on most 6502-based computers, it is unlikely that you will often have a chance to do so. This is probably the least used of all 6502 addressing modes.

Like indirect indexed addressing, indexed indirect addressing has its limitations. The indirect address must be stored on the zero page and only the X register can be used for indexing. The instructions that can use indexed indirect addressing are: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

IMMEDIATE ADDRESSING—If the descriptions of the last few addressing modes have left you a little dizzy, relax.

Things get easier from here on out. Immediate addressing, for instance, is one of the simplest addressing modes offered by the 6502, and we've already used it several times in this book. With immediate addressing, the actual data to be manipulated by the instruction is placed after the op-code.

For instance, the instruction:

```
LDY #$45
```

loads the immediate number $45 into the Y register. And the instruction:

```
CMP #$BB
```

compares the number in the A register with the immediate number $BB. The number sign (#) tells the assembler that we are using immediate addressing, and the instruction is translated into machine language accordingly. The above instruction, for instance, would assemble into this sequence of machine-language instructions:

```
C9 BB
```

The op-code $C9 indicates a CMP instruction using immediate addressing and the immediate number $BB resides in memory directly after it.

You might wonder why this is called an "addressing mode" at all, since we are including the actual data in the instruction rather than the address of that data. However, a little reflection will reveal that we are, in a sense, telling the microprocessor where the data is stored: It is stored in the byte of memory directly after the op-code. When the 6502 sees that immediate addressing is in effect, it says to itself, "I know where the data is stored! It's in the next byte of memory!" It then fetches the data from that location.

Obviously, immediate addressing is limited to certain kinds of instructions. It would be meaningless, for instance, to say STA #$80, since this fails to provide the microprocessor with a location in which to store the contents of A. (Well, it could store the contents of A in the memory location following the op-code of the instruction, but this would be a useless and confusing activity.) The instructions that

can use immediate addressing are: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, SBC.

IMPLIED ADDRESSING—This, you'll be happy to hear, is the simplest addressing mode of all. In a few instances, the source or destination of data does not need to be specified at all, because it is implicit in the instruction itself. In the instruction TAX (transfer A to X), for example, the source of the data is the A register and the destination is the X register. This is stated in the actual instruction, so you need not include an operand further specifying such information. When the instruction TAX is assembled into machine language it will be only one byte long and will look like this:

$AA

The instruction PLA also uses implied addressing. It copies an eight-bit number from memory to the A register, but the source of the number is always the memory address being pointed to by the stack pointer (SP) register. Thus, no specific address need be stated; the address is implied by the instruction. Similarly, the instruction PHA stores the eight-bit number in the A register at a memory address, but the destination address is always the address being pointed to by the stack pointer.

Instructions that use implied addressing are: BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA.

RELATIVE ADDRESSING—This form of addressing specifies the address as a displacement from the memory location of the current instruction. Although some of the more advanced microprocessors use relative addressing with a wide variety of instruction types, the 6502 uses it only with branch instructions. See the discussion of branch instructions in the last chapter for a more detailed description.

The major advantage of relative addressing is that a relative address, like a zero-page address, takes up less space in memory than an absolute address. Thus, instructions using relative addressing execute more quickly.

****

Those are the 6502 addressing modes. We will be using most of them in upcoming chapters. If you don't understand how they work now, be prepared to refer back to this chapter at appropriate moments. The 6502 addressing modes are every bit as important to the programmer as the actual instructions in the instruction set, and a good understanding of them will make you a good assembly-language programmer.

6
USING THE ASSEMBLER

Before we begin writing actual assembly-language programs, we must know something about using an assembler. Because the assembler is the program that translates our assembly-language instructions into machine language, we must know exactly what the assembler expects to find in our program and the format in which it expects to find it. Like all computer programs, the assembler is very literal-minded, and though we may present it with a perfectly workable program, it will not assemble it correctly (or at all) if we have not presented it in the proper form.

This sounds annoying, but it really isn't. Once you know how to use the assembler you'll discover that it provides a wealth of utilities that will make your job as an assembly-language programmer much easier. So we'll also discuss those utilities in this chapter.

When you write an assembly-language program, the first thing you will need is a *text editor.* This is simply a program that allows you to type programs and to save them on external storage media, such as disks. (If you plan to do very much assembly-language programming, it is strongly recommended that you use disk storage for your programs, as opposed to storage on cassette tape. Assembly-language programming requires frequent recording and retrieving of

Text editor—A program that aids in the creation of computer programs.

programs, and this task should be made as easy as possible for the sanity of the programmer.)

Many assemblers come with a built-in text editor, or a text editor that is available on the same disk with the assembler. In a few cases, you may need to supply your own text editor. Many word-processing programs will serve in this capacity.

With the text editor, you will create a program file, which can then be assembled into machine language. The program file is simply a long string of ASCII characters comprising an assembly-language program. Here is a (quite short) example of a program file:

```
;
; DEMONSTRATION PROGRAM
; Illustrates assembler program file format
;
WMSTRT=0 ;                  Warm start address
OUTCHR=$FF56 ;              Video output routine
;
*=$4000
          LDY #0 ;          Initialize index
LOOP      LDA MESSG1,Y ;    Get next character
          CMP #0 ;          Are we done yet?
          BEQ REBOOT ;      If so, return to system
          JRS OUTCHR ;      Else print it
          JMP LOOP ;        And go for more
REBOOT    JMP WMSTRT ;      Reinitialize system
MESSG1    .BYTE 'HELLO, THERE!',0
```

Once a program file such as the above has been created, it can be saved to a disk in the same fashion that a word processor saves documents on the disk. Then the assembler can be used to render it into a machine-language program, which will also be saved to the disk, usually during the assembly itself.

The assembly-language program file itself is called the *source program.* The machine-language version of the pro-

---

Source program—An assembly-language program as created by the programmer.

gram, created by the assembler, is called the *object program.* Although the object program is the one that will actually be executed by the computer, it is a good idea to save copies of both versions. If you should need to make changes in the program (and you will *always* need to make changes, if only to correct errors), you will have to make them in the source program and then use the assembler to create a new object program (unless you wish to make the changes directly in the object code, a daunting task at best).

Take another look at the program file given earlier. Even though we have covered all the instructions in the 6502 instruction set and all the 6502 addressing modes, there is still much in this program that looks unfamiliar. Notice, for instance, that there are few numbers referred to in this program, and there are a number of unfamiliar words. What does MESSG1 mean? Or LOOP? We didn't cover these words in the chapter on the instruction set.

That's true. And, truth to tell, these words are not in the 6502 instruction set. They were invented by the programmer who created the program file. This is a valuable capability offered by the assembler, and one that we will take advantage of often.

Other words and symbols in this program, such as the equals sign (=), the asterisk (*), and the word BYTE, are not parts of the 6502 instruction set, but neither were they invented by the programmer. These are called *assembler directives,* and they represent direct instructions to the assembler. Assembler directives are not translated into machine language. Rather, they give the assembler information concerning the translation of the rest of the program. We'll cover several 6502 assembler directives in this chapter.

Before we go on, however, we should offer a warning.

Object program—A machine-language program as created by an assembler, based on an assembly-language program.

Assembler directive—Instruction to the assembler that is not itself translated into machine language.

There are many different 6502-based computers and many different assemblers available for each. Although there are certain conventions followed by most of these assemblers, we cannot guarantee that each assembler will do the things described in this chapter in precisely the way we describe. Some assemblers may use different directives or impose limitations on the directives as they are described here. Or the assembler process may require a different series of steps than those just described. It is important that you at least skim through the assembler manual before attempting to use your assembler; otherwise, you may find yourself hopelessly confused as you try to transfer the techniques you learn in this book to your computer.

****

Perhaps the most important feature of any assembler is the ability it gives us to define labels. A *label* is a word that represents a number. We may assign a number to a label either through an *equate statement* or by referring to the label in the *label field.*

Equate statements resemble assignment statements in BASIC. They usually consist of the name of the label, an equals sign, and the numeric value we wish to assign to the label. There are two equate statements in the program sample on page 58. The first looks like this: `WMSTRT=0`. This assigns a value of `0` to the label `WMSTRT`. Similarly, the equate statement `OUTCHR=$FF56` assigns the value `$FF56` to the label `OUTCHR`.

Once this assignment has been made, the assembler will treat the labels as though they were these numbers. We may

---

Label—A symbolic name, defined by the programmer, representing an address within the memory of the computer.

Equate statement—An assembler directive that explicitly assigns a value to a label.

Label field—The portion of an assembly-language program where labels are placed to be assigned values by indirect reference.

then use these labels at any point in an assembly-language instruction where a number is normally expected. For instance, the instruction JSR OUTCHR is equivalent to the instruction JSR $FF56 and will be assembled accordingly.

Labels bear a superficial resemblance to BASIC variables, but do not be misled. A variable represents a location in the computer's memory where a value is stored; a label represents the value itself. Once a label is assigned a value, that value cannot (and should not) be changed. A better analogy for labels would be the constants found in Pascal and certain other high-level languages.

The rules for creating label names are similar to those for creating variable names in BASIC. The name must begin with a letter of the alphabet, which may be followed by more letters or numbers. Most assemblers place an upward limit on the number of characters in a variable name—usually either six or eight. And some assemblers will reject labels that have the same name as an assembly-language mnemonic.

The second method of assigning a value to a label is to place the label name in the label field. Note how the program given earlier is written in such a way that the page almost seems to be divided into columns. Each of these columns is called a *field,* and different types of information go into each. Some assemblers may require that you place a tab character between each column (by pressing the TAB key on the computer, or some equivalent, such as the right arrow); others simply require that you separate each field by a space.

The first field is the label field. Note that the example program has three entries in this field: LOOP, REBOOT, and MESSG1. When the assembler encounters a label in the label field, it assigns a value to that label equal to the memory address into which the current instruction is being assembled. For instance, if the instruction LDA MESSG1,Y in the example program is assembled at memory address $4002 (as it will be, if the program is assembled properly), then the label LOOP, which is in the label field along with

---

Field—An area within the text of a program reserved for a specific type of information.

that instruction, will be assigned a value of $4002. Seconds later, when the assembler encounters the instruction JMP LOOP, it treats it as though it read JMP $4002. Thus, this instruction will cause execution of the program to jump back to the instruction on the line labeled LOOP.

The advantage of assigning addresses to labels in this manner is that the programmer will probably not know in advance what addresses various instructions will reside in once the program has been translated into machine language. And, even if he or she went through the program and calculated each address by hand—a phenomenal amount of work—the addresses would change every time the program was revised. Labels, however, allow us to pass along this job to the assembler.

The equate operator (=) is an example of an assembler directive. (A few assemblers may use the word EQU in its place.) Another assembler directive, shown in the program on page 58, is the asterisk symbol (*), or *program counter.* Despite its name, this program counter is not the same as the program counter (PC) register in the microprocessor. Rather, it represents an internal variable in the assembler that keeps track of the address at which the current instruction is being assembled. It is the program counter, for instance, that determines what address value will be assigned to a label when it is encountered in the label field.

We may assign a value to the program counter with the equate operator, just as we would assign a value to a label. The value assigned to the program counter, however, unlike the value assigned to a label, may be changed at any time. By changing the value of the program counter, we may determine the address at which our program will reside in memory.

For instance, in the example program, we assign the program counter a value of $4000 with the statement *=$4000. This means that the instruction following this assignment—which happens to be the first instruction in

---

Program counter (*)—A symbol, usually an asterisk, that represents the memory address into which instructions are currently being assembled.

our program, LDY#0—will be assembled at address $4000. All succeeding instructions will be assembled in consecutive addresses, until the value of the program counter is altered. If no value is assigned to the program counter, it will automatically be assigned a value of 0. This, however, would place the opening instructions of our program on the zero page, which is never a good idea. It is best to locate your programs in an area of RAM that is not being used for other purposes.

Another directive apparent in the sample program is .BYTE. The .BYTE directive allows us to insert numeric (or ASCII) values of our own choice into memory, bypassing the normal translation. This is a convenient way of initializing areas of a program used to store data. For instance, suppose we wish to place the numbers 1, 2, and 3 into the computer's memory so that our program may perform some manipulation on these values. We could use the .BYTE directive like this:

```
.BYTE 1,2,3
```

The values listed after the .BYTE directive will be placed, by the assembler, in three consecutive memory addresses, beginning at the current address of the program counter. The .BYTE directive itself goes in the second program column. We call this column the *instruction field* because it is where assembler directives (instructions to the assembler) and assembly-language op-codes (instructions to the microprocessor) are placed. The instruction field is immediately followed by the *operand field,* where the operands of assembly-language op-codes and the data to be processed by the assembler directives are placed. In this case, the operand field contains the numbers to be placed into memory by the .BYTE directive, each number separated from the others by a comma.

Instruction field—The portion of an assembly-language program where op-codes are placed.

Operand field—The portion of an assembly-language program where operands are placed.

In our sample program, however, the .BYTE directive is used in a slightly different manner than we have described here. It is used to place a string of characters in memory. (Some assemblers may have a different directive for this purpose.) The string is placed in the operand field and surrounded by single quote marks—that is, apostrophes—like this: .BYTE 'HELLO, THERE'. The .BYTE directive tells the assembler to break this string into individual ASCII code numbers, one for each character, and insert them into consecutive memory locations, beginning at the current location of the program counter.

The .BYTE directive has a crucial limitation: As its name implies, it cannot handle numbers larger than 255, the maximum value that will fit into an eight-bit memory address. Should you need to place a sixteen-bit value into memory, you will need to use a different directive: .WORD. The .WORD directive can take as an operand any integer value up to 65,535. It will place this value into *two* consecutive memory addresses, and it will reverse the order of the halves of the number, in standard 6502 format. (Should you not want the halves reversed, some assemblers offer an alternative directive: .DWORD.) For instance, if you wrote this directive:

```
.WORD $4005,7
```

it would place the two values, $4005 and 7, in memory like this: $05,$40,$07,$00. The second value, which is (considerably) less than 255, is nonetheless treated as a full sixteen-bit word and assembled into two memory locations, with the second being given a value of 0.

When we place a value in memory, we will often need to make reference to the location of that value elsewhere in the program. The simplest way to do this is to place a label in front of the .BYTE or .WORD directive and refer to the location of the data by label. In the sample program, for instance, we have placed the label MESSG1 on the line with the .BYTE directive that is used to store the string of characters. And the second instruction of the program, LDA MESSG1,Y, makes a direct reference to this location.

If we are not certain of a numeric value when we want to insert it into memory or assign its value to a label, we may ask the assembler to calculate it for us, using a simple

numeric expression with ordinary numeric operators (such as +, −, *, /). For instance, if we know that a certain value will be located nine memory addresses after an address that we have labeled STORE, we can make reference to that data in an instruction like this:

```
LDA STORE−9
```

This tells the assembler to calculate the operand of the instruction by adding 9 to the value of label STORE. If label STORE represents the address $712A, then this instruction would be treated as though it read LDA $7133.

Similarly, we can equate a label to an expression in the same manner, like this:

```
THERE=HERE−$25
```

This sets the label THERE equal to the value of label HERE minus $25.

Most assemblers offer the standard operators for addition (+), subtraction (−), multiplication (*), and division (/), though the division is usually integer—that is, any fractional portion of the result will be dropped. In addition, some assemblers will offer additional operators such as AND and OR. Check your manual for details.

Two more directives that you will find useful are < and >. These directives, when used at the beginning of a label representing a sixteen-bit number, tell the assembler that you only wish to use the low byte (that is, the rightmost eight bits) or the high byte (the leftmost eight bits) of the number, respectively. For instance, suppose that label FIRST is equal to the 16-bit number $5690. If we write the label as <FIRST, the assembler will treat it as though it were the number $90—the low byte of $5690. Similarly, if we write the label as >FIRST, the assembler will treat it as though it were the number $56—the high byte of $5690.

Why would we want to do this? Well, suppose that we wanted to store the value of the label FIRST at a zero-page memory location, where it would become a pointer to a value elsewhere in memory. We cannot place it in memory with a single sequence of load and store instructions because it is a sixteen-bit number and load and store instructions can only manipulate eight bit numbers. Thus,

we must break it into two halves, each eight bits long, and store them separately, like this:

```
LDA #<FIRST
STA $05
LDA #>FIRST
STA $06
```

This stores the sixteen-bit value of label FIRST on the zero page, starting at address $0005—but it does the job eight bits at a time.

Another significant benefit that we get from the assembler is the ability to include comments in our source code. A *comment* is much like a remark statement in BASIC. It is a note, placed in the program by the programmer, that helps to clarify the intent of the program. In our sample program, there is a comment at the beginning that gives the program a name and describes its purpose. In addition, each line of the program ends with a comment, explaining the action of the instruction on that line.

A comment may be placed at any point in a program line. It must be preceded by a semicolon (;). All text following the semicolon is ignored by the assembler, until the end of the line (i.e., the point at which the RETURN key was pressed) is reached. Thus, no assembly-language instructions or assembler directives should follow a comment on a program line.

Although comments are important in all programming languages, they take on extra importance in assembly language. Assembly-language code tends to be obscure, even unreadable, compared with most high-level languages. Even shortly after a program is written, it is possible for the programmer to forget what task a line, or even an entire section, of code is to perform. Thus, if the program is to be properly debugged, or modified, at a later date, comments are essential. In some cases, a light commentary (i.e., comments after every other instruction or so) will be satisfactory; some programmers, however, prefer to comment volu-

---

Comment—Text inserted within a program that is not assembled into machine language but is intended for the benefit of the reader.

minously, placing detailed summations at the beginning of each machine-language routine, explaining what the routine will do, the state of the microprocessor registers at the start and the end of the routine, which registers are involved in the routine, and so on. How much commentary you include is a matter of taste; nonetheless, it is generally better to overcomment than to undercomment. This is true for programming in any language.

****

Now that you have some idea of how the assembler works, go back and see if you can figure out the machine-language routine at the beginning of this chapter. Don't feel bad if you still have trouble. There are several principles used in this program that we have not yet studied. Note, for instance, that the program calls a subroutine named OUTCHR that is not part of the program itself. This routine is assumed to be part of the operating system of the computer. We will study operating systems in more detail in the chapter on input/output. However, we can safely say that the role of subroutine OUTCHR is to print on the video display of the computer the ASCII character represented by the code number in the A register.

This program also uses branch instruction to make decisions concerning the output of characters. We will study this process in the chapter on decision-making; first, however, we must take a more detailed look at arithmetic on the 6502 microprocesser—and study the extremely important role of the status register.

7
WAVING THE FLAGS

We've already seen how to solve simple arithmetic problems on the 6502 microprocessor, such as those involving addition and subtraction of eight-bit numbers. The process is pretty straightforward: you load one number into the A register, store one number in a memory address, and perform an ADC instruction, like this:

```
LDA#$45
STA  $5600
LDA#$3F
CLC
ADC  $5600
```

This series of instructions adds the numbers $45 and $3F and loads the result back into A. Simple enough.

However, as the complexity of the arithmetic problem increases, so does the complexity of solving that problem on the 6502. Suppose, for instance, that we wish to add or subtract numbers larger than 255? We lose the ability to add the numbers with a single ADC instruction, because the numbers will no longer fit in an eight-bit memory address or microprocessor register. Nonetheless, it is possible to add sixteen-bit (and larger) numbers on the 6502 microprocessor by taking advantage of the *status register.*

Status register—The register within the CPU that contains the flags.

The status register (SR) does not function in the same manner as the other registers in the 6502 microprocessor. It can contain eight bits of information, but we cannot deliberately load an eight-bit number into it (except via the PLP instruction). This is immaterial, however; as programmers, we will never worry about the number contained in the status register. Rather, it is the individual bits of that number with which we will be concerned.

The binary digits within the status register are called *flags,* because they are used to indicate the current status of events within the microprocessor. These flags are affected in various ways by other instructions executed by the 6502. Since each flag consists of a single bit, it can take only two different values: zero and one. Thus, each flag can be looked at as representing the answer to a yes-or-no question. It is the programmer's job to ask the appropriate questions; the flags will provide the answers.

Here is a description of the information provided by the six 6502 flags, each of which is in the status register:

ZERO FLAG—Tells us whether the result of the last operation was a zero. If the result was zero, the zero flag will have a value of zero; if the result was not zero, the zero flag will have a value of one.

CARRY FLAG—Tells us if the result of the last operation (such as an ADC or SBC) was larger than eight bits or smaller than 0.

OVERFLOW FLAG—Tells us if the leftmost bit of a number was altered during an operation. This flag is used for signed arithmetic—that is, arithmetic on numbers that can be positive or negative—a type of arithmetic we will not be studying in this book.

NEGATIVE (or SIGN) FLAG—Tells us if the result of the last operation was positive or negative. This flag is also used for signed arithmetic.

Flag—An individual bit within the status register that indicates the outcome of a previous operation.

DECIMAL MODE FLAG—This tells us if the decimal mode, which we will be be discussing in this book, is on or off.

INTERRUPT DISABLE FLAG—Controls the interrupt status, which we will not be discussing in this book.

In this book, the two flags that we will be concerned with are the zero flag and the carry flag. As you can see from the above description, the state of these flags is affected by other operations performed by the 6502. Not all operations affect the state of the flags, and not all operations that do affect the flags affect the same flags. For instance, the LDA, LDX, and LDY instructions affect the zero flag and the sign flag, while the ADC and SBC instructions affect the negative, zero, carry, and overflow flags; the STA, STX, and STY instructions affect no flags at all. A list of flag effects is contained in the Appendix, along with a list of 6502 instructions.

Once the flags have been set by an instruction, how do we read them? Well, we cannot read the flags directly—that is, there are no 6502 instructions that will tell us if a specific flag is a 1 or 0. In truth, there would be no point in this. The specific condition of the flags is irrelevant. What matters is the manner in which they can be made to affect the behavior of still other 6502 instructions. We'll study some of these effects in the next chapter, when we discuss decision-making. For now, we are chiefly interested in the effect that the flags have on arithmetic—and the effect that arithmetic has on the flags.

Addition and subtraction on the 6502 must be performed eight bits at a time—that is, if you wish to add or subtract two numbers greater than eight bits in size, you must break them up into eight-bit segments and perform arithmetic on each segment separately. This process bears a striking resemblance to the way in which addition and subtraction are traditionally performed with pencil and paper, where two numbers are added or subtracted one column at a time.

Suppose, for instance, that you wish to add the numbers 56 and 23. Assuming that you are not in the mood to perform the operation in your head, you would write the two numbers on a sheet of paper, like this:

```
 56
 23
```

You would then add the two numbers a column at a time, adding the numbers in the rightmost column (6 and 3) and writing the result (9) in the rightmost column of the answer, then adding the numbers in the second column (5 and 2) and placing the result (7) in the leftmost column of the answer. The answer, of course, is 79.

Suppose, however, that the result of one of these two additions had been a two-digit number—that is, a number that would not fit into the single digit position that we allotted for it in the answer? What we would do, as you probably know, is take the extra digit and use it as a carry—that is, add it into the total in the next column. For instance, if we wished to add the numbers 45 and 98, we would place them, one above the other, like this:

```
 45
 98
```

We would then add the numbers in the first column. The result, however, would be 13, which is too large to place in the first column of the answer. Instead, we would place the 3 in the first column, then add the 1 to the total for the second column, which would therefore be 14. This, in turn, generates a carry of 1, which can be added into the total of the nonexistent third column, producing a total of 1 in the third column of the answer, for a grand total of 143.

We can do precisely the same thing in binary arithmetic. Suppose, for instance, that we wish to add the two hexadecimal numbers $4A91 and $334A. We can add them together as we would add two decimal numbers, bearing in mind that each column can hold any of the numbers 0 through F rather than 0 through 9. If we need to place a number in a column that is larger than F, we take the second digit and use it as a carry, just as we did above. Thus, the result of adding $4A91 and $334A would be:

```
 4A91
 334A
 7DDB
```

We can program the 6502 to perform this operation, but we must do it in a special fashion. We must first break these sixteen-bit numbers into eight-bit halves and treat the halves separately. First we add the *least significant bytes*—that is, the right halves of the numbers. Then we add the *most significant bytes*—that is, the left halves of the numbers. However, when we add the most significant bytes, we must add in any carry generated by the addition of the least significant bytes, just as we add the carry from the previous column when we perform arithmetic on paper.

How do we know if there has been a carry? This information will be contained on the carry flag. The carry flag, you will recall, is made up of a single binary digit, which can be either 0 or 1. If a carry is generated by an ADC instruction (i.e., if the value of the result is greater than 255), the carry flag will become a 1. If no carry is generated, the carry flag will become a 0. Since it is impossible to generate a carry larger than 1 in binary arithmetic, the carry flag can be looked at as representing the actual carry itself: 1 if there was a carry and 0 if there was not.

The instruction ADC, you may recall, stands for "add with carry." It will automatically add the value of the carry flag into any addition operation we perform. Thus, when we use ADC to add the most significant bytes of our numbers, it will adjust the result to take the carry into account.

Before we perform an addition operation on two sixteen-bit numbers, we will generally store those numbers in the computer's memory, with the numbers in two consecutive memory addresses. Typically, we will reverse the bytes of these numbers, in standard 6502 fashion, so that the least significant bytes precede the most significant bytes.

Assume that we have stored two numbers in memory, one beginning at the address labeled NUMBR1 and one

---

Least significant byte—The right half of a sixteen-bit binary number. (One-half of sixteen bits equals eight bits, or one byte.)

Most significant byte—The left half of a sixteen-bit binary number.

beginning at the address labeled NUMBR2. Here is a short subroutine, which we have called ADDEM, that will add the two numbers and store the result back at the address labeled NUMBR1:

```
ADDEM  CLC
       LDA NUMBR1
       ADC NUMBR2
       STA NUMBR1
       LDA NUMBR1+1
       ADC NUMBR2+1
       STA NUMBR1+1
       RTS
```

The first instruction, CLC, sets the carry flag to 0, thus removing any irrelevant carries left over from earlier operations, so that they will not be added into the least significant bytes by the ADC operation. *It is important that you always perform a CLC before beginning an addition operation, or you may get inaccurate results!* The second instruction loads the contents of location NUMBR1 into the A register. This location presumably contains the least significant byte of one of the numbers to be added. The third instruction adds this number to the contents of location NUMBR2, which should contain the least significant byte of the other number. The next instruction stores the result back at location NUMBR1.

The next three instructions repeat this process with the numbers stored at NUMBR1+1 and NUMBR2+1, which should contain the most significant bytes of the first and second numbers, respectively; the result is stored at NUMBR1+1. (Note that we are using a simple assembler expression here to represent the bytes of memory immediately following addresses NUMBR1 and NUMBR2.) We do not clear the carry flag before this second addition, so that any carry from the first addition will be added to the most significant bytes.

The RTS instruction then returns control to the calling routine, which will find the sixteen-bit result of the addition starting at location NUMBR1.

In the same fashion, we can perform subtraction on sixteen-bit numbers. When we use the SBC instruction, the carry flag becomes a borrow flag, indicating whether the

result of the subtraction operation was less than 0. If the carry flag indicates that a borrow has been generated by the previous operation, it will automatically subtract the borrow from the current operation.

Note, however, that the carry flag does not indicate a borrow in the same way that it indicates a carry. When a borrow is generated, the carry flag becomes 0. If no borrow is generated, the carry flag becomes a 1. Thus, if we do not wish for the borrow to be figured into an SBC operation, we must first set the carry flag to 1, rather than clearing it to 0. This is done with the SEC (Set Carry flag) instruction.

Here is a subroutine, similar to ADDEM on page 74, that will subtract two sixteen-bit numbers stored at locations NUMBR1 and NUMBR2:

```
SUBEM   SEC
        LDA NUMBR1
        SBC NUMBR2
        STA NUMBR1
        LDA NUMBR1+1
        SBC NUMBR2+1
        STA NUMBR1+1
        RTS
```

The calling routine will find the result of the subtraction at address NUMBR1. The process, as you can see, is virtually identical to that in ADDEM, except that we substitute an SEC for the CLC instruction and SBCs for the ADC instructions.

Through the use of the carry flag and the ADC and SBC instructions, we can (theoretically, at least) perform addition and subtraction operations on numbers of any size—32 bits, 128 bits, 2,000 bits, and so on. We simply break the numbers into eight-bit segments and add or subtract each with an ADC or SBC instruction, remembering to clear or set the carry flag before the first operation is performed.

The INC and DEC instructions, which are really specialized addition and subtraction instructions, affect the flags in exactly the same manner as ADC and SBC. (So do the INY, INX, DEY, and DEX instructions.) If the INC instruction generates a value greater than 255, for instance, the carry flag will be set to 1, indicating that a carry has taken place. Similarly, if the DEC instruction generates a value smaller

than 0, the carry flag will be cleared to 0, indicating that a borrow has taken place. There is no way that we can induce the INC and DEC instructions to add in the value of the carry or subtract the value of the borrow, but we would rarely want to do so. These instructions have other uses, as we shall see in the next chapter.

The CMP instruction is quite similar to the SBC instruction. In fact, CMP actually performs a subtraction of two numbers *without* figuring in the borrow and sets the flags according to the result of the subtraction. However, the result of the subtraction is thrown away, and only the value of the flags is retained, leaving the contents of the A register unchanged. We'll see how the CMP instruction is used in the next chapter (as well as the very similar CPX and CPY instructions).

Sharp-eyed readers may notice that we have yet to say anything in this chapter about multiplication and division. There is a reason for this. Multiplication and division are major operations in machine language, requiring complex sequences of instructions. Although some of the more advanced (and expensive) microprocessors have machine-language instructions that perform these operations on binary numbers, the 6502 does not, and we do not have space here to discuss all the steps necessary to create an all-purpose multiplication and division routine. We will leave that discussion to a more advanced book on assembly-language programming.

Nonetheless, you might be interested in a couple of shortcuts that can be used to perform multiplication and division in specialized circumstances. It is often the case, for instance, that a machine-language programmer is called upon to multiply or divide a number by a power of 2 (2 multiplied by itself a certain number of times). This specialized form of division and multiplication is relatively easy to perform in machine language, using the shift and rotate instructions that we described in Chapter Four. You might want to turn back to that chapter and take a second look at the descriptions of the instructions ASL, LSR, ROL, and ROR.

The ASL instruction, which shifts every bit in a binary number one position to the left, is equivalent to multiplying a number by 2, the same way that shifting all the digits of a decimal number to the left is equivalent to multiplying

it by 10. (The decimal number 1234, for instance, becomes 12340 when multiplied by 10.)

It follows that shifting the digits of the binary number to the left *more* than once is equivalent to multiplying the number by a power of 2. For instance, this sequence of instructions:

```
ASL A
ASL A
```

is equivalent to multiplying the number in A by 4 (which is 2 to the second power). This sequence of instructions:

```
ASL A
ASL A
ASL A
ASL A
```

is equivalent to multiplying the number in A by 16 (which is 2 to the fourth power).

It is helpful to note that a digit is "caught" on the carry flag every time it is shifted off the end of the A register. For instance, if the leftmost digit in the A register is 1 and we perform an ASL A instruction, that digit will disappear from the A register—but the carry flag will be set to 1. In the same way, if the leftmost digit in A is 0 and we perform an ASL A instruction, the digit will be lost from A, but the carry flag will be cleared to 0.

In this way, we can combine the ASL instruction with an ROL instruction to shift the digits of a sixteen-bit (or larger) number. The ROL instruction, even as it shifts the digits of a number to the left, automatically moves the contents of the carry flag into the rightmost (least significant) digit position. Thus, the ROL instruction can be used to rescue the contents of the carry flag and shift it between two memory addresses. The following routine rotates the contents of a sixteen-bit number stored (in typical reversed order) beginning at location NUMBR1, effectively multiplying the sixteen-bit number by 2:

```
MULT2 ASL  NUMBR1
      ROL  NUMBR1 + 1
      RTS
```

Quite a simple process, actually. By repeating it, we can multiply by any power of 2.

The LSR instruction is the opposite of the ASL instruction. It shifts every digit in a binary number one position to the right. This is equivalent to dividing the number by 2, just as shifting every digit of a decimal number to the right is equivalent to dividing the number by 10. (The number 1234, for example, becomes 123.4 when divided by 10.)

The following instructions effectively divide the contents of the A register by 4:

```
LSR A
LSR A
```

Of course, any fractional value of the division is lost. We could take elaborate steps to preserve the fractional value, but such an operation would be beyond the scope of this book.

In the same fashion that the ROL instruction can be used to extend a multiplication operation beyond a single byte, so the ROR instruction (which also "rescues" the contents of the carry flag) can be used to extend the division beyond a single byte, though the bytes must be processed in reverse order, from most significant to least significant.

8
DECISIONS, DECISIONS

The ability of computers to perform arithmetic, no matter how difficult it is at times to implement that arithmetic in machine language, is impressive. And yet a computer is not really a calculating machine. It is a logic machine, capable of making decisions. And that is what sets a computer apart from, say, a pocket calculator.

In a language such as BASIC, decision-making is easy. We simply say IF (such and such is true) THEN (do this), and the computer makes a decision based on the truth or falsehood of the logical statement following the word IF. Based on that decision, it either performs or refrains from performing the action following the word THEN.

We don't have it quite so easy in machine language. Most decision-making on the 6502 microprocessor is controlled by branch instructions. But we can write branch instructions that will execute or not execute depending on the outcome of certain other events within the microprocessor. This allows us to branch or fail to branch to selected routines in the computer's memory—and thus we can duplicate the sort of IF-THEN logic offered by high-level programming languages.

The action of the branch instructions is dependent on the status of the flags in the status register. Here is a list of the 6502 branch instructions, similar to the one in Chapter Four, but with an added description of the specific effects of the flags:

BEQ—Branch on equal. The branch will take place if

the value of the zero flag is equal to 0, indicating that the last operation produced a 0.

BNE—Branch on not equal. The branch will take place if the value of the zero flag is 1, indicating that the last operation did not produce a 0.

BCS—Branch on carry set. The branch will take place if the value of the carry flag is 1, indicating that the last operation generated a carry or did not generate a borrow.

BCC—Branch on carry clear. The branch will take place if the value of the carry flag is 0, indicating that the last operation generated a borrow or did not generate a carry.

BMI—Branch on minus. The branch will take place if the value of the negative (or sign) flag is 1, indicating that the result of the last operation was negative.

BPL—Branch on plus. The branch will take place if the value of the negative flag is 0, indicating that the result of the last operation was positive.

BVS—Branch on overflow set. The branch will take place if the value of the overflow flag is 1.

BVC—Branch on overflow clear. The branch will take place if the value of the overflow flag is 0.

Now we can make the flow of our program contingent on the outcome of various operations within the microprocessor. For instance, suppose we have a value stored at location NUMBR1 that may or may not be equal to 0. If it is equal to 0, we want to execute a routine named ZERO; if it is not equal to 0, we want to execute a routine called NOZERO. We need merely load the value from NUMBR1 into the A register and test the value of the zero flag with a BEQ (or BNE) instruction, like this:

```
        LDA NUMBR1
        BEQ ZERO
NOZERO  ....
        ....
        JMP NEXT
ZERO    ....
        ....
NEXT    ....
```

If the value at NUMBR1 is 0, the act of loading it into the A register will set the zero flag to 0, since the LDA instruction

is one of the instructions that affects the zero flag. The BEQ instruction tests the value of the zero flag and branches to routine ZERO if it is equal to 0. Otherwise, the flow of the program automatically proceeds to routine NOZERO. At the end of NOZERO, the instruction JMP NEXT guides the flow of the program around routine ZERO so that it will not be executed as well.

This is roughly equivalent to this statement in BASIC:

```
IF N1=0 THEN .... ELSE ....
```

where N1 is a variable equivalent to location NUMBR1 and the statements following the THEN and the ELSE are equivalent to routines ZERO and NOZERO, respectively.

Alternatively, we could write our machine-language program like this:

```
          LDA NUMBR1
          BNE NOZERO
ZERO      ....
          ....
          JMP NEXT
NOZERO    ....
          ....
NEXT      ....
```

We have simply reversed the earlier situation, but with identical results. Now we test for inequality to 0 rather than equality. This is equivalent to this BASIC statement

```
IF N1<>0 THEN .... ELSE ....
```

where the statements following the THEN and the ELSE are equivalent to NOZERO and ZERO, respectively.

We can also simulate more complex IF-THEN logic. Consider this BASIC statement:

```
IF N1<N2 THEN .... ELSE ....
```

The best way to simulate the less than (<) operation in machine language is to use the CMP instruction in conjunction with a BCC or BCS instruction, like this:

```
        LDA  NUMBR1
        CMP  NUMBR2
        BCC  LESTHN
NOLESS  ....
        ....
        JMP  NEXT
LESTHN  ....
        ....
NEXT    ....
```

The CMP instruction subtracts the value at NUMBR2 from the value in the A register and sets the flags accordingly (though it throws away the actual result of the subtraction, leaving the contents of A unchanged). If this subtraction produces a result that is less than 0, the carry flag will be cleared to 0, indicating that a borrow has taken place. This tells us that the value at NUMBR2 was greater than the value at NUMBR1—or, conversely, that NUMBR1 was less than NUMBR2. The BCC instruction watches for this borrow and branches to routine LESTHN if it takes place. Otherwise, program flow continues to routine NOLESS.

We could also write this routine with a BCS instruction:

```
        LDA  NUMBR1
        CMP  NUMBR2
        BCS  NOLESS
LESTHN  ....
        ....
        JMP  NEXT
NOLESS  ....
        ....
NEXT    ....
```

Once again, we have simply reversed the situation. Again, NUMBR2 is subtracted from NUMBR1. If NUMBR2 is greater than or equal to NUMBR1, no borrow will be generated, and the BCS instruction (which will execute only if there has been no borrow) will branch to routine NOLESS. This is equivalent to this BASIC statement:

```
IF N2>= N1 THEN .... ELSE ....
```

One of the primary limitations of this kind of logic is that

branch instructions will only transfer control over a range of 255 bytes, 129 forward and 126 backward. What if the routine we want to branch to falls out of this range?

In such an event, we must combine branch instructions with JMP instructions. The branch instructions will skip over the JMP instruction if we do not want to make the jump. For instance, suppose we want to jump to routine HELLO if the value in the A register is less than the value at NUMBR1. We can use the following routine:

```
        CMP  NUMBR1
        BCS  NEXT
        JMP  HELLO
NEXT    ....
```

If the value in A is less than the value at NUMBR1, the carry flag will be cleared to 0, indicating a borrow. The BCS instruction will not execute and the jump to routine HELLO will be performed. Otherwise, control will fall through to the routine beginning at NEXT.

Similarly, we can make JSR and RTS instructions contingent on the state of the flags by placing branch instructions so that the JSR or RTS is skipped if we do not want it to execute.

It takes little imagination to see that any kind of decision logic that is possible in a high-level language can be duplicated through the strategic use of branch statements. The procedure may seem more awkward than the straightforward IF-THEN statements of BASIC, but it is also more versatile—and considerably more powerful.

One important use of decision-making branches is the creation of program loops—sequences of instructions that will be repeated a specified number of times or until a certain condition is met.

For instance, suppose we want to execute a program routine exactly ten times, then continue on with the rest of the program. In BASIC, we might create a loop like this:

```
10 I=0
20 ..(body of the loop)..
30 I=I+1
40 IF I<10 THEN GOTO 20
50 .. (rest of the program)..
```

Or, alternatively, like this:

```
10 FOR I=1 TO 10
20 ..(body of the loop)..
30 NEXT I
40 ..(rest of the program)..
```

Both methods operate by the same principle: The variable I (which can be called the *index* of the loop) is used to count the number of times the loop has executed, and the loop is terminated when the value of I exceeds 10. Thus, we know that the loop will execute precisely ten times (unless the value of I is affected by some instruction within the loop, a circumstance that should generally be avoided).

In machine language, we can create a similar loop using a method quite similar to the first BASIC example. Here is a possible machine-language implementation:

```
LOOP  LDY #10
      ..(body of the loop)..
      DEY
      BNE LOOP
      ..(rest of the program)..
```

We start by loading a value of 10 into the Y register. After each pass through the loop, we decrement this value with the DEY instruction. If the decrement operation does not produce a zero—that is, if the zero flag is not equal to 1—the BNE instruction will branch back to the beginning of the loop, causing it to repeat once more. On the tenth pass through the loops, however, the value of Y will be decremented to 0, which will set the zero flag—and control will fall through to the following instruction.

Thus, the Y register becomes the index of the loop. We can use this method to create loops that will execute as many as 256 times. (To get a full 256 repetitions, we must place a value of 0 in the Y register.) If we want to repeat a loop more times than this, we must store the index in memory rather than in a register.

The following instructions create a loop that will execute $2000 times, by placing this sixteen-bit index in memory starting at location INDEX:

```
        LDA #$00
        STA INDEX
        LDA #$20
        STA INDEX+1
LOOP    ..(body of loop)..
        DEC INDEX
        BNE LOOP
        DEC INDEX+1
        BNE LOOP
NEXT    ....
```

The first four instructions set up the loop index by breaking the number $2000 into two eight-bit segments and loading them into locations INDEX and INDEX+1. The first DEC instruction decrements the value at INDEX, which contains the least significant byte of the index. If the number is not decremented to 0, the BNE instruction loops the program back to the line labeled LOOP. If it is decremented to 0, a second decrement instruction is performed, on location INDEX+1, which contains the most significant byte of the index. If the result of this decrement is not 0, the program also loops back to LOOP.

Thus, the most significant byte is decremented once for every 256 times that the least significant byte is finally decremented to 0, control falls through to the routine labeled NEXT, and the loop terminates. The total number of executions will be $2000.

By combining loops and sophisticated condition tests, a wide variety of loop types can be created, duplicating such looping structures as the WHILE-DO and REPEAT-UNTIL loops offered by languages such as Pascal.

9 INPUT AND OUTPUT

We've come quite a distance in our study of 6502 assembly language. You have now been introduced to most of the major principles of microprocessor programming. You may have noticed, however, that we have yet to put together a complete, usable program. Fragments of programs, yes, but not a complete program.

There's a reason for this. Although we have the tools for constructing such a program, the results would do us little good. The program might well be complex, elegant, and bug-free, but it would spend a great deal of time talking to itself. As yet, we have introduced no method by which the microprocessor can communicate with us, or by which we can communicate with the microprocessor, while the program is being executed.

And this presents a thorny problem. Input and output—the moving of information between the outside world and the CPU of a computer—is not the job of the microprocessor. Rather, it is the job of devices attached to the microprocessor. The microprocessor sends information to and receives information from these devices, but it is the devices themselves that do the actual communicating with the world.

Since you are reading this book, it seems likely that you have access to a 6502-based computer or hope to have access to one soon. Everything you have learned so far about programming the 6502 microprocessor will apply to that computer, whether it be an Apple, an Atari, a Commodore, or whatever.

From now on, however, we are on somewhat shakier ground. There are many different devices that may be attached to the 6502 microprocessor for input and output. The specific sequence of instructions required to transmit information even to so seemingly standard a device as the video display may vary drastically from one computer to the next. Thus, it is impossible for us to lay down absolute rules as to how input and output are performed on your computer.

Fortunately, there may already be a program built into your computer—or available on a floppy disk—that can help us circumvent the differences between various computers. This program is called an *operating system.* Almost every computer has an operating system available, though you might not even be aware of it.

In the Commodore computers, for instance, the operating system is called the Kernal. It is built into the computer, in read-only memory (ROM), between addresses $E000 and $FFFF. Similarly, the Atari computers have an operating system called, simply, the OS. It is also in ROM and resides between memory addresses $D800 and $FFFF.

The operating system contains built-in subroutines for most of the types of input and output you will want to perform. You may call these subroutines with a JSR instruction, just as though they were part of your program.

We are not going to go into detail about all the subroutines offered by your operating system. The manual that came with your computer—or the technical manual that should be available, on request, from the computer's manufacturer—contains this information. In this book we will concentrate on two specific kinds of operating system routines: those that output ASCII characters to the video display and those that input ASCII characters from the keyboard.

Every operating system contains a subroutine to output an ASCII character to the video display. We will use the routine from the Commodore Kernal to demonstrate how

---

Operating system—A series of machine-language subroutines within the internal memory of a computer that can be used for the input and output of data.

such a routine is used, then show you how this information can be extended to other computers.

At address $FFD2 in the memory of *every* Commodore computer resides a routine that the manual calls CHROUT. (Actually, address $FFD2 contains a JMP instruction that transfers control to the actual address of routine CHROUT, but this is irrelevant from the programmer's point of view.) This routine contains the machine-language instructions necessary for outputting information to a wide variety of devices, including the video display, disk drive, printer, modem, cassette recorder, and so on. If you don't tell it otherwise, however, CHROUT will send data to the video display by default.

To output a character to the display, we place the ASCII code for the character into the A register and instruct the 6502 to JSR $FFD2. Better still, we can use an equate to create a label called CHROUT, equal to the sixteen-bit address $FFD2, and write the instruction as JSR CHROUT. That's all we have to do. The character will be printed at the current cursor position. This is usually the position immediately after the last character printed (or in the upper left-hand corner of the screen, if no character has been printed yet), unless a cursor control code, such as a carriage return (ASCII 13), has been printed since the last character.

For instance, suppose we wish to print the letter A at the current cursor position. The ASCII code for A is 65; therefore, we can print an A with these instructions:

```
LDA #65
JSR CHROUT
```

Some assemblers make our task easier for us by allowing us to specify an ASCII character directly, with the assembler itself calculating the ASCII code for the character. Here is a typical assembler format for such an instruction:

```
LDA #'A
JSR CHROUT
```

The number sign (#) indicates that immediate addressing is in effect. The single quote/apostrophe (') is an assembler directive of sorts. It tells the assembler that what follows is

an ASCII character and that the actual number we wish loaded into the register is the ASCII code number for that character. Thus, this program segment is precisely equivalent to the one immediately before it.

We will use the CHROUT routine for output throughout the rest of this book, and we will equate it to address $FFD2. If you are using a computer other than a Commodore, you should change the equate to make CHROUT equal to the address of the similar routine in your computer's operating system, and you must add any extra instructions needed to make the routines compatible.

Atari owners will need to perform a somewhat more complicated series of actions before they can output to the screen. On all Atari computers, the equivalent routine to CHROUT is located at $E456 and is called CIO (central input/output) by the manual. CIO is somewhat more complex than CHROUT. As its name implies, it can perform both input and output.

Unfortunately, CIO will not perform default output to the video display, as will CHROUT. You must carefully tell CIO what device you will be communicating with and whether you will be performing input or output. This is done by setting up an I/O control block, or IOCB, for short. It is recommended that you consult the Atari technical manual for specifics of this process.

What if we want to output an entire sentence to the video display? There may or may not be a routine in your operating system to perform this task; however, it is easy enough to write a short routine of our own that will do the job. Here's an example, called OUTLIN:

```
OUTLIN  LDY  #0 ; INITIALIZE INDEX
LOOP    LDA  MSG1,Y ; GET CHARACTER OF MESSAGE
        CMP  #0 ; ARE WE DONE YET?
        BEQ  DONE ; IF SO, JUMP
        JSR  CHROUT ; ELSE OUTPUT CHARACTER
        INC  Y ; POINT TO NEXT CHARACTER IN MESSAGE
        JMP  LOOP ; AND GO GET IT
DONE    JMP  DONE ; DELAY FOREVER
MSG1    .BYTE 'THIS WILL BE PRINTED ON VIDEO DISPLAY',0
```

If you've been paying attention, you'll recognize this as a slight variation on the example program in Chapter Six.

Though you may have had trouble grasping the principles of that routine then, you should have no such troubles now, since we have discussed all the necessary concepts in the intervening chapters.

The first line initializes the value of index register Y. The second line then uses indexed addressing to get the first of the characters contained in the string beginning at address MSG1. The program then uses a CMP instruction to see if this character is a 0. Why a 0? If you look down at the string of characters defined by the .BYTE statement on the last line, at address MSG1, you will see that we have marked the end of that line with a 0. This allows our output routine to identify the last character when it finds it. A 0 is a good character to use for this purpose, since it represents neither a control character nor a printable character in the ASCII code.

If the character is a 0, the BEQ instruction branches to the routine labeled DONE. (You might notice that the CMP #0 instruction in the third line is actually unnecessary, since the LDA MSG1,Y instruction that precedes it will be sufficient to set the zero flag by itself, should the character at MSG1,Y turn out to be a 0. Nonetheless, we have included the CMP #0 instruction to make the function of this routine clearer to the reader. You may drop it from your own programs, if you wish.)

If the character is not a 0, the program falls through to the next line, JSR CHROUT, which calls the CHROUT routine and prints the character in the A register. On the first pass through this loop, that character will be the ASCII code for the letter T, the first character of our string. The INC Y instruction increases the value in the index register and the JMP LOOP instruction transfers control back to the line labeled loop. Now MSG1,Y points to the beginning of the string at MSG1,Y plus the value in the index register, which is 1; thus, LDA MSG1,Y now fetches the second character in the string, which is H. The following instructions print this character, increment Y again, and loop back for more.

This process continues, with the index register pointing at each subsequent character, until the 0 is encountered. Then the loop ends. The instruction JMP DONE, which is itself at address DONE, will continue looping to itself until we press the RESET button (or turn the computer off and

on). This should give us sufficient time to admire the message printed by the program.

The routine on page 90 has two rather severe limitations. One is that it cannot output a string of more than 256 characters, because this is the largest index value that the Y register can hold. For some programs this may not prove serious, but there may come a time when we wish to display strings of greater lengths. The second limitation is that the program can only output the string at location MSG1. This is fine if our program will be outputting only a single string, but should we want to output a second string we will need to write a second such routine for the task, which is not very efficient programming.

We can solve both problems by using indirect indexed addressing rather than simple indexed addressing to point to the characters in the string. This results in an all-purpose subroutine that can output any string of nearly any length. We need merely place the address of the string beginning at location PNTER (so-named because it is a "pointer" to the string) in standard reverse order and then use a JSR instruction to call this new, improved version of OUTLIN:

```
        ; SUBROUTINE OUTPUT-LINE
        ; Outputs a string of characters
        ; to the video display. The address
        ; of the string must be stored at
        ; location PNTER. The string must
        ; be terminated with a 0 byte.

OUTLIN LDY  #0 ; INITIALIZE INDEX
LOOP    LDA  (PNTER),Y ; GET CHARACTER OF MESSAGE
        CMP  #0 ; ARE WE DONE YET?
        BEQ  DONE ; IF SO, JUMP
        JSR  CHROUT ; ELSE OUTPUT CHARACTER
        INC  PNTER ; POINT TO NEXT CHARACTER
        BNE  LOOP ;
        INC  PNTER+1
        JMP  LOOP
DONE    RTS  ; MISSION ACCOMPLISHED
```

You should recognize the instructions that increment location PNTER from our discussion of the INC instruction in Chapter Seven. If the first half of the pointer reaches 0 dur-

ing any one increment, the second half is incremented. Either way, the program loops back to LOOP and goes for the next character.

The routine that calls this subroutine might look like:

```
        LDA #<MSG1 ; GET LOW BYTE OF MESSAGE
ADDRESS
        STA PNTER ; STORE IN POINTER
        LDA #>MSG1 ; GET HIGH BYTE OF MESSAGE
ADDRESS
        STA PNTER+1 ; STORE IN POINTER
        JSR OUTLIN ; PRINT MESSAGES
        JMP MORE ; CONTINUE
MSG1    .BYTE 'THIS TOO WILL BE PRINTED',0
MORE    . . .
```

LDA #<MSG1 and LDA #>MSG1 load the high and low byte, respectively, of the address of MSG1 into PNTER.

Just as important as output to the video display is input from the keyboard. Without it, we could have only limited real-time interaction with the computer (though some recent computer models make extensive and creative uses of input devices such as the mouse, which point to options presented on the screen).

The Commodore Kernal contains a routine called GETIN, at memory address $FFE4, which handles keyboard input. Call this routine with a JSR instruction and the ASCII code of the character most recently typed at the keyboard will be placed in the A register. If no character has been typed, a 0 will be placed in the A register.

We can combine this routine with the CHROUT routine to create a short, simple program that will allow us to type on the keyboard and see the characters echoed to the display. Here's the program:

```
        ; PROGRAM SCREEN ECHO
        ; Accepts characters from the keyboard
        ; and echoes them to the video display.
        ;
CHROUT=$FFD2
GETIN  =$FFE4
        ;
*=$4000
```

```
        ;
SCECHO  JSR GETIN
        CMP #0
        BEQ SCECHO
        JSR CHROUT
        JMP SCECHO
```

This program will continue accepting characters until we press RESET or turn off the computer. Note that, after we call the GETIN routine, we check the A register for a 0, which would indicate that no character has been typed. If this is the case, we loop back for another look, not bothering to call CHROUT and echo the non-existent character.

We can use GETIN to create a subroutine that will allow us to type a complete string of characters at the computer's keyboard, ending with a carriage return, and have that string stored in memory. Such a routine might look like this:

```
        ; SUBROUTINE INPUT-LINE
        ; Accepts a string of characters from
        ; the keyboard terminating with a
        ; carriage return. Stores string
        ; at location STRING and auto-
        ; matically terminates string
        ; with 0 byte.
        ;
GETIN=$FFE4
      ;;
*=$4500
INLINE      LDA #<STRING ; GET LOW BYTE OF STORAGE
ADDRESS
            STA  PNTER ; STORE IN POINTER
            LDA  #>STRING ; GET HIGH BYTE OF STORAGE
ADDRESS
            STA  PNTER+1 ; STORE IN POINTER
            LDY  #0 ; INITIALIZE INDEX
INLOOP      JSR  GETIN ; GET CHARACTER
            CMP #0 ; NOTHING TYPED?
            BEQ  INLOOP ; IF SO, TRY AGAIN
            JSR  CHROUT ; ELSE SHOW US WHAT WE TYPED
            STA  (PNTER),Y ; AND SAVE IT
            INY  ; POINT TO NEXT STORAGE ADDRESS
            BNE  INNEXT ; AND CONTINUE
```

```
            INC PNTER+1 ; IF OUT OF INDEX RANGE,
INCREMENT POINTER
INNEXT      CMP #13 ; WAS IT A CARRIAGE RETURN
            BEQ INDONE ; IF SO, WE'RE FINISHED
            JMP INLOOP ; ELSE GO FOR MORE
INDONE      LDA #0 ; TERMINATE WITH 0 BYTE
            STA (PNTER),Y ; AT END OF STRING
            RTS ; AND RETURN TO SENDER
STRING=*
*=*+1024
```

Note that we use a new method of incrementing the pointer (at PNTER) in this routine. Instead of incrementing the low byte of PNTER each time through the loop and incrementing the high byte every time the low byte reaches 0, we increment the Y register each time through the loop and increment the high byte every time the Y register reaches 0. It is never necessary to increment the low byte of the pointer. This saves us one or two bytes of program space and allows our program to run minutely faster.

Notice also that we have set label STRING equal to the value of the program counter at the end of the routine, and then reset the value of the program counter to its current value plus 1024. What does this do? Essentially, it creates a blank, uninitialized area of 1,024 bytes in the middle of our program (assuming that more instructions follow these) and sets STRING equal to the first address of that area. This is the section of memory—the *buffer,* in programming terminology—where the string that we input from the keyboard will be stored. Since this buffer is only 1,024 bytes long, it is possible for the string to overflow and damage information (or programming) stored at the end of the buffer. The reader may be interested in adding instructions to this routine that would count the number of characters inputted and stop or automatically return to the calling routine if 1,024 characters are received.

There are no editing facilities provided in this routine. If the user should make a mistake and need to back up and change a character, there is no way that he or she can do so.

Buffer—A section of internal memory set aside for the temporary storage of data.

You may want to add such a feature. This is not an entirely trivial task, but it might make an interesting exercise.

Now that we have the capability for communicating with the user of the computer, we are in a position to put together a useful program, or at least a program that completes an entire task, from beginning to end, that you can run on your computer. And, in the next chapter, that is precisely what we will do.

10
DOING IT ALL

Machine language is a complex and subtle means of communication between programmer and computer. As we pointed out earlier, any type of program that can be written on a given computer can be written in machine language. There are no restrictions in terms of speed and versatility, as there are in high-level languages. Machine language gives you absolute control.

The price you pay is one of complexity. Writing programs with an assembler is not an easy job; it requires considerable thought, planning, and attention to detail. For that reason, our final project will be a relatively simple one, a program that would seem trivial in a high-level language. In machine language, however, we will discover unexpected ramifications of seemingly simple procedures, and we will learn that it is necessary to plan every step of the program carefully, considering the impact of each step on every other. (These are not bad rules to follow when working in high-level languages, either, but they become even more important in machine language.)

Our program will do this: It will allow the user to input ten numbers from the keyboard, and it will print the total of those ten numbers. Then it will give the user the option of typing another ten numbers and seeing them added as well.

First, however, we will outline the program in pseudocode. Pseudocode is a language that resembles a programming language but is not an actual language. There are no rules for writing pseudocode, and there is no fixed vocabu-

lary; you make up the rules and terminology as you go along. A well-written pseudocode program can then be translated into an appropriate programming language.

Here is a pseudocode outline of our program:

```
REPEAT
  Prompt user for input
  REPEAT
    Get number from keyboard.
    Store in array.
  UNTIL 10 NUMBERS ARE RECEIVED
  Initialize total to 0.
  Initialize pointer to first array element.
  REPEAT
    Add current array element to total.
    Advance pointer to next array element.
  UNTIL COMPLETE ARRAY TOTALED
  Display total.
  Ask if user wants to repeat program.
UNTIL USER DOES NOT WANT TO REPEAT
```

This, of course, is only one possible outline for the program. We could, for instance, add each number to the previous number as they are input at the keyboard, thus removing the need to store the numbers in an array. However, retaining the array allows us to illustrate certain principles of data storage.

To contrast our machine-language solution to this problem with a typical high-level language solution, let's translate this program first into BASIC. A BASIC version of this outline might look like this:

```
10 DIM A(10)
20 PRINT "Type ten numbers, pressing RETURN after each."
30 FOR I=0 TO 9
40     INPUT A(I)
50 NEXT I
60 T=0
70 FOR I=0 TO 9
80     T=T+A(I)
90 NEXT I
100 PRINT "The total of the ten numbers is ";T
```

```
110 PRINT "Would you like to try again (Y/N)?"
120 INPUT C
130 IF C="Y" THEN GOTO 20
140 IF C="N" THEN END
150 GOTO 110
```

The translation to BASIC was fairly painless. The ten numbers are held in variable array A and are input and totaled through a pair of FOR-NEXT loops. There is nothing terribly difficult to understand about this program, for anyone who knows the essentials of BASIC programming.

In machine language, however, we may find ourselves running up against unforeseen obstacles, portions of the outline that need to be elaborated in still greater detail before we are ready to write the actual assembly-language code. For instance, it is simple enough to input a number from the keyboard in BASIC and then perform arithmetic operations on that number. We simply store the number in a numeric variable, and input the value of that variable with an INPUT statement. How, on the other hand, do we go about inputting a number in machine language?

In the last chapter, we saw that characters could be received from the keyboard and processed by a machine-language program simply by calling the appropriate operating system routine. Certainly we can input numbers in this manner, since numbers are available on the keyboard of a computer just as letters and other characters are. But you must bear in mind that what we receive from the keyboard are not the actual numbers to be processed, but the ASCII codes representing those numbers. This is an important consideration.

For instance, if the user types 601 at the keyboard, what we will receive internally, to be processed by our program, are the ASCII code numbers 54, 48, and 49. Each of these will presumably be stored in a separate memory address. If we attempt to perform addition using these numbers, the result will be strange, at best.

Well, you ask, can we convert these numbers into a form that we can work with? Yes, we can. On paper, the answer is simple enough. Since the ASCII code for a 0 is 48, we need merely substract 48 from the ASCII code number to get the actual digit. Thus, 54 minus 48 is 6, 48 minus 48 is

0, and 49 minus 48 is 1. And those are the same numbers that the user typed: 6, 0, and 1. Yet, each is still stored separately, in its own memory address; they are not yet in a form that we can use. How can we combine them into a single binary number?

The formula for this is simple. You might try figuring it out for yourself. We multiply each digit by the power of 10 appropriate to its position within the number. The rightmost, or least significant, digit is multiplied by 1, the middle digit by 10, and the leftmost, or most significant, digit by 100; then we add the resulting numbers together. The complete formula would look like this: (6*100)+(0*10)+(1*1). This procedure yields a value of 601—the number we started out with, but if we perform this process in machine language, using binary arithmetic, the result will be a binary number that the 6502 can deal with.

This is called decimal-to-binary conversion (though technically it should be called a decimal-ASCII-to-binary conversion) and is a very common process in machine-language programs that deal with the input and output of numbers. It is not an easy routine to write. For one thing, we saw earlier that the 6502 microprocessor lacks instructions to perform multiplication on binary numbers. Thus, the portion of the procedure that multiplies the digits by powers of 10 is actually quite complex and involves a large number of instructions. Similarly, the process of converting a binary number back into decimal ASCII, for output to the video display or other device, involves repeated division by 10, another process that is difficult to perform on the microprocessor level.

Thus, we will not be discussing decimal-binary and binary-decimal conversions in this book; we lack the space. We will, instead, use the far simpler process of hexadecimal-binary and binary-hexadecimal conversion. The user of our program will simply have to be content to type numbers in hexadecimal. To make our lives even easier—and our program simpler to understand—we will restrict input to eight-bit numbers, each containing two hexadecimal digits.

Because hexadecimal is so close mathematically to binary, we will need to perform almost no true multiplication and division in the conversion. And what little multi-

plication and division we do perform will be disguised as shift operations, as demonstrated in Chapter Eight.

The first stage of the hexadecimal-binary conversion is the same as that for the decimal-binary conversion—that is, we subtract a number from the ASCII code number of the hex digit to get the actual number represented by that digit—except that, in this case, the microprocessor will need to make a choice as to exactly what number it should subtract, depending on the hexadecimal digit. Here is the procedure, in pseudocode:

IF the hex digit is '1' – '9', THEN subtract 48 (ASCII '0').
ELSE IF the hex digit is 'A' – 'F', subtract 55.

The extra 7 subtracted from the digits A – F compensates for the seven-character gap in the ASCII code between the codes for 9 and A.

Remember that every ASCII character is equivalent to a four-digit binary number. Once we have subtracted the appropriate number, we will have that four-bit number, but we must then move those digits into the proper position within the memory address that is to hold the binary number. For instance, if we are translating the hexadecimal number $4A into binary, the four bits representing 4 (0100) will go into the left half of the resulting byte and the four bits representing A (1010) will go into the right half, like this: 01001010. Thus, we must move these bits into position with a shift instruction, where necessary. We must then combine the two halves into a single byte with an ORA instruction. For a binary-hexadecimal conversion, this process is reversed. If this seems less than clear, you will see it in action in the machine-language version of our program.

And now that we have taken numerical conversion into account, we are ready to do the assembly-language translation of our routine. It is presented here as a fully commented listing, so that you will be able to understand how each portion of it works and how the various portions interact. Study it carefully. Some of the subroutines, such as the numerical conversions and the string output sections, may be lifted bodily out of this program so that you can use them in your own programs.

```
; *** ADDITION PROGRAM ***
;
; INPUTS TEN 2-DIGIT HEXADECIMAL
; NUMBERS FROM THE KEYBOARD,
; STORES THEM IN MEMORY, ADDS
; THEM TOGETHER, AND PRINTS THE
; TOTAL ON THE VIDEO DISPLAY.
;
CHROUT=$FFD2            ; VIDEO OUTPUT ROUTINE
GETIN=$FFE4             ; KEYBOARD INPUT ROUTINE
WMSTRT=64738            ; WARM-START ADDRESS
PNTER=$FB
TOTAL=$FD
TEMP=$FE
TEMPX=TOTAL
TEMPY=TEMP
CR=13                   ; CARRIAGE RETURN CHARACTER
;
*=$4200                 ; SAFE MEMORY LOCATION
ADDTEN LDA #<MSG1       ; POINT TO FIRST PROMPT
       STA PNTER
       LDA #>MSG1
       STA PNTER+1
       JSR OUTLIN       ; GO PRINT IT
       LDX #0           ; INITIALIZE INDEX TO ARRAY
LOOP1  LDA #'?          ; PRINT '?' PROMPT
       JSR CHROUT
       LSA #32          ; ADD A SPACE
       JSR CHROUT
       JSR HXINPT       ; GET HEX NUMBER
       LDA #CR          ; PRINT A CARRIAGE RETURN
       JSR CHROUT
       JSR HEXBIN       ; CONVERT INPUT TO BINARY
       STA ARRAY,X      ; SAVE IT IN ARRAY
       INX              ; POINT TO NEXT POSITION IN ARRA
       CPX #10          ; ARE WE FINISHED?
       BNE LOOP1        ; IF NOT, GET MORE
       LDA #0           ; ELSE INITIALIZE TOTAL
       STA TOTAL        ;
       STA TOTAL+1
       LDX #0           ; REINITIALIZE INDEX
LOOP2  CLC
       LDA ARRAY,X      ; GET NEXT ARRAY ELEMENT
       ADC TOTAL        ; ADD TO RUNNING TOTAL
       STA TOTAL
       LDA #0           ; CARRY TO 2ND BYTE OF TOTAL
       ADC TOTAL+1
```

```
        STA TOTAL+1
        INX
        CPX #10
        BNE LOOP2
        LDA #<MSG2      ; POINT TO 2ND PROMPT
        STA PNTER
        LDA #>MSG2
        STA PNTER+1
        JSR OUTLIN      ; PRINT IT
        LDA #<BUFFER    ; POINT TO BUFFER
        STA PNTER       ; (WHERE TOTAL WILL BE)
        LDA #>BUFFER
        STA PNTER+1
        LDA TOTAL+1     ; GET HI-BYTE OF TOTAL
        JSR BINHEX      ; CONVERT TO HEX
        JSR OUTLIN      ; AND PRINT IT
        LDA TOTAL       ; GET LO-BYTE OF TOTAL
        JSR BINHEX      ; CONVERT TO HEX
        JSR OUTLIN      ; AND PRINT IT
        LDA #CR         ; ADD A CARRIAGE RETURN
        JSR CHROUT
        LDA #<MSG3      ; POINT TO 'AGAIN?' PROMPT
        STA PNTER
        LDA #>MSG3
        STA PNTER+1
        JSR OUTLIN      ; PRINT IT
LOOP3   JSR GETTIN      ; LOOK FOR RESPONSE
        CMP #0          ; NOTHING TYPED?
        BEQ LOOP3       ; LOOK AGAIN
        CMP #'Y         ; WAS IT 'Y'?
        BNE NEXT        ; IF NOT, SKIP
        JMP ADDTEN      ; ELSE REPEAT PROGRAM
NEXT    CMP #'N         ; WAS IT 'N'?
        BNE LOOP3       ; IF NOT, LOOK AGAIN
        JMP WMSTRT      ; ELSE END PROGRAM
;
; **LINE OUTPUT ROUTINE**
;
; OUTPUTS A STRING OF TEXT TO THE
; VIDEO DISPLAY. ADDRESS OF
; STRING MUST BE AT LOCATION
; 'PNTER'. STRING MUST TERMINATE
; WITH A 0 BYTE.
;
OUTLIN  LDY #0          ; POINT TO FIRST CHARACTER
OLLOOP  LDA (PNTER),Y   ; GET CHARACTER
        CMP #0          ; ARE WE FINISHED
```

```
        BEQ OLDONE      ; IF SO, JUMP
        JSR CHROUT      ; ELSE PRINT IT
        INY             ; POINT TO NEXT CHARACTER
        BNE OLLOOP      ; IF INDEX NOT 0, GO FOR MORE
        INC PNTER+1     ; ELSE INCREMENT POINTER
        JMP OLLOOP      ; AND GO FOR MORE
OLDONE RTS              ; GO HOME
;
; **HEXADECIMAL INPUT ROUTINE**
;
; INPUTS A 2-DIGIT HEXADECIMAL
; NUMBER FROM THE KEYBOARD AND
; STORES IT AT LOCATION 'BUFFER'.
; WILL NOT ALLOW INPUT OF NON-HEX
; CHARACTERS.
;
HXINPT  LDY #0          ; BEGIN COUNT & INDEX
HXLOOP STX TEMPX        ; SAVE CONTENTS OF X,Y REGS
        STY TEMPY       ; SINCE GETIN USES THEM
        JSR GETIN       ; GET CHARACTER
        LDY TEMPY       ; NOW RESTORE X,Y REGS
        LDX TEMPX       ; TO PREVIOUS VALUES
        CMP #'0;        ; NOTHING TYPED
        BEQ HXLOOP      ; IF SO, TRY AGAIN
        CMP #'0         ; LESS THAN '0'?
        BCC HXLOOP      ; IGNORE & TRY AGAIN
        CMP #'G         ; GREATER THAN 'F'?
        BCS HXLOOP      ; IGNORE & TRY AGAIN
        CMP #'A         ; BETWEEN 'A' & 'F'?
        BCS OKAY        ; GREAT! GO WITH IT!
        CMP #':         ; BETWEEN '9' AND 'A'?
        BCS HXLOOP      ; IGNORE & TRY AGAIN
; ALL CHARACTERS THAT GET THIS FAR ARE HEXADECIMAL
OKAY    STA BUFFER,Y    ; SAVE IT
        JSR CHROUT      ; PRINT IT
        INY             ; COUNT IT
        CPY #2          ; TWO DIGITS YET?
        BNE HXLOOP      ; IF NOT, GO FOR MORE
        RTS
;
; **BINARY-HEX CONVERSION ROUTINE**
;
; CONVERTS A BINARY NUMBER TO A
; HEXADECIMAL ASCII STRING.
; EXPECTS BINARY NUMBER IN
; ACCUMULATOR (A REGISTER).
; LEAVES HEX STRING AT LOCATION
; 'BUFFER.' TERMINATES WITH O.
```

```
;
BINHEX  TAY                 ; SAVE BINARY NUMBER
        AND #$F0            ; REMOVE LEAST SIGNIFICANT DIGIT
        LSR A               ; SHIFT HIGH HALF TO LOW HALF
        LSR A
        LSR A
        LSR A
        JSR CONVT1          ; ADD ASCII VALUE
        STA BUFFER          ; SAVE IN BUFFER
        TYA                 ; GET BINARY NUMBER
        AND #$0F            ; REMOVE MOST SIGNIFICANT DIGIT
        JSR CONVTI          ; ADD ASCII VALUE
        STA BUFFER+1        ; SAVE IN BUFFER
        LDA #0
        STA BUFFER+2
        RTS
CONVTI  CLC                 ; ADD ASCII VALUE
        ADC #48
        CMP #58             ; WAS IT 'A' OR GREATER?
        BCC CVT12           ; IF NOT, RETURN
        CLC
        ADC #7              ; ELSE ADD ANOTHER 7
CVT12   RTS
;
; **HEX-BINARY CONVERSION ROUTINE**
;
; CONVERTS A 2-DIGIT HEXADECIMAL
; STRING TO AN 8-BIT BINARY
; NUMBER. EXPECTS HEX STRING AT
; LOCATION 'BUFFER'. LEAVES
; BINARY NUMBER IN ACCUMULATOR.
;
HEXBIN  LDA BUFFER          ; GET 1ST CHARACTER
        JSR CONVT2          ; SUBTRACT ASCII VALUE
        ASL A               ; SHIFT TO HIGH HALF
        ASL A
        ASL A
        ASL A
        STA TEMP            ; SAVE HIGH HALF
        LDA BUFFER+1        ; GET NEXT CHARACTER
        JSR CONVT2          ; SUBTRACT ASCII VALUE
        ORA TEMP            ; COMBINE WITH HIGH HALF
        RTS
CONVT2  SEC                 ; SUBTRACT ASCII VALUE
        SBC #48
        CMP #10             ; WAS IT 'A' TO 'F'?
        BCC CVT22           ; IF NOT, RETURN
        SEC
```

```
        SBC #7              ; ELSE SUBTRACT ANOTHER 7
CVT22   RTS
BUFFER=*                    ; OPEN BUFFER
*=*+3                       ; ALLOT 3 BYTES
ARRAY=*                     ; OPEN SPACE FOR ARRAY
*=*+10                      ; ALLOT 10 BYTES
MSG1 .BYTE 'PLEASE TYPE 10 2-DIGIT HEX NUMBERS', 130,
MSG2 .BYTE 'THE TOTAL OF THE 10 NUMBERS IS ',0
MSG3 .BYTE 'WOULD YOU LIKE TO TRY AGAIN (Y/N)?',130,
```

Although you should read through this program and analyze its workings on your own, there are several points of interest that we will discuss here.

For instance, we have created, through an equate at the beginning of the programming, an address called WMSTRT. This represents the address of your computer's warm-start routine. If you pass control to this routine, your computer will reinitialize its operating system and place you in a position to load a new program. On many computers (including the Commodore) this will put you into the BASIC interpreter, where you may begin giving commands in the BASIC language. On other computers, this will put you in the disk operating system, where you may give commands that activate various utility programs available on the disk. If you do not know the warm-start address of your computer, you may wish to omit the routine that references this address. The address for WMSTRT used in this program is the address of the warm-start routine on the Commodore 64 computer.

We have also established several addresses for the temporary storage of data and placed these addresses on the zero page, for efficient access. These addresses are given the labels PNTER, TOTAL, TEMP, TEMPX, and TEMPY. (Note that TEMPX and TEMPY have been given the same addresses as TOTAL and TEMP, since these labels are not used by the same routine. This allows us to save zero-page space while retaining distinct label names.) When choosing zero-page addresses to use on your computer, you should be careful not to interfere with the addresses used by the computer's operating system; this could interfere with the CHROUT and GETIN routines, which are called by this program. The addresses used in this program are considered "free" zero-page space on the Commodore 64, because they

are the only zero-page addresses on that computer not used by the built-in software. To locate equivalent locations on your computer, you will need a *memory map*—that is, a list of all the addresses used by your computer's internal software.

The data input from the keyboard is stored in a buffer we call ARRAY. This buffer is ten bytes long, sufficient to hold the ten eight-bit numbers that the user will input from the keyboard. Index register X is used to point to the current position within the buffer.

If you have questions about this program, you must attempt to answer them yourself. If something seems puzzling, puzzle it out. The comments and labels should help. One of the best ways to learn assembly language, aside from writing your own assembly-language programs, is to read other programmers' assembly-language programs.

You might even want to read your way through some machine-language programs that you have purchased commercially. This isn't easy, since these programs are stored on disk and in your computer's memory as a sequence of binary numbers. However, with the aid of a disassembler, you can translate these programs back into assembly-language mnemonics. Of course, you won't have comments and labels to help you understand what is happening in these programs, but the very act of deciphering the output of the disassembler will teach you to think like an assembly-language programmer. You might want to buy a machine-language *monitor,* a program that will allow you to inspect the contents of your computer's memory—and any programs contained therein—in minute detail. A good monitor will include not only a disassembler but also a *sin-*

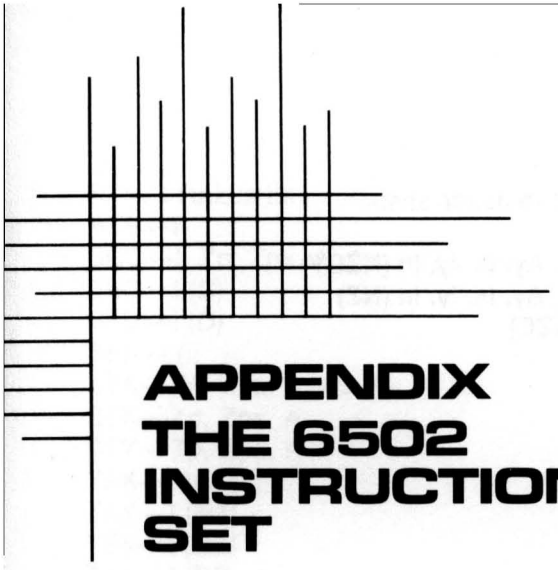
Memory map—A list of important memory locations within the computer describing how those locations are used by the operating system or by peripheral devices.

Monitor—A program that allows the user to examine the contents of the computer's memory and the microprocessor registers, and to execute a machine-language program one instruction at a time.

gle stepper—a program that lets you execute a machine-language program one instruction at a time, observing the status of the microprocessor registers and memory after each.

And, once you have examined assembly/machine-language programs written by others, and written a few such programs of your own, you'll be an assembly-language expert yourself.

Single stepping—A process by which the instructions in a machine-language program can be executed one at a time.



APPENDIX THE 6502 INSTRUCTION SET

The following is a list of all 6502 instructions. Each instruction is followed by a list of the allowable addressing modes for that instruction and, in parentheses, a list of the flags affected by that instruction. The following symbols represent the addressing modes:

immediate—I
zero page—Zp
zero page, X—Zpx
zero page, Y—Zpy
absolute—A
absolute, X—Ax
absolute, Y—Ay
implied—Im
relative—R
indirect, X—Ix
indirect, Y—Iy
indirect—In

The following symbols represent the flags:

negative—N
zero—Z
carry—C
interrupt—I
decimal mode—D
overflow—V

Here now are the 6502 instructions:

ADC—I, Zp, Zpx, A, Ax, Ay, Ix, Iy, In (NZCV)
AND—I, Zp, Zpx, A, Ax, Ay, Ix, Iy, In (NZ)
ASL—Zp, Zpx, A, Ax (NZC)
BCC—R
BCS—R
BEQ—R
BIT—Zp, A (NZV)
BMI—R
BNE—R
BPL—R
BRK—(I)
BVC—R
BVS—R
CLC—I (C)
CLD—I (D)
CLI—I (I)
CLV—I (V)
CMP—I, Zp, Zpx, A, Ax, Ay, Ix, Iy (NZC)
CPX—I, Zp, A (NZC)
CPY—I, Zp, A (NZC)
DEC—Zp, Zpx, A, Ax (NZ)
DEX—I (NZ)
DEY—I (NZ)
EOR—I, Zp, Zpx, A, Ax, Ay, Ix, Iy (NZ)
INC—Zp, Zpx, A, Ax (NZ)
INX—I (NZ)
INY—I (NZ)
JMP—A, In
JSR—A
LDA—I, Zp, Zpx, A, Ax, Ay, Ix, Iy (NZ)
LDX—I, Zp, Zpy, A, Ay (NZ)
LDY—I, Zp, Zpx, A, Ax (NZ)
LSR—Zp, Zpx, A, Ax (NZ)
NOP—I
ORA—I, Zp, Zpx, A, Ax, Ay, Ix, Iy (NZ)
PHA—I
PHP—I
PLA—I (NZ)
PLP—I (takes flag values from stack)
ROL—Zp, Zpx, Ax, Ay (NZC)
ROR—Zp, Zpx, Ax, Ay (NZC)

RTI—I (takes flag values from stack)
RTS—I
SBC—I, Zp, Zpx, A, Ax, Ay (NZCV)
SEC—I (C)
SED—I (D)
SEI—I (I)
STA—Zp, Zpx, A, Ax, Ay, Ix, Iy
STX—Zp, Zpx, A
STY—Zp, Zpx, A
TAX—I (NZ)
TAY—I (NZ)
TSX—I (NZ)
TXA—I (NZ)
TXS—I
TYA—I (NZ)

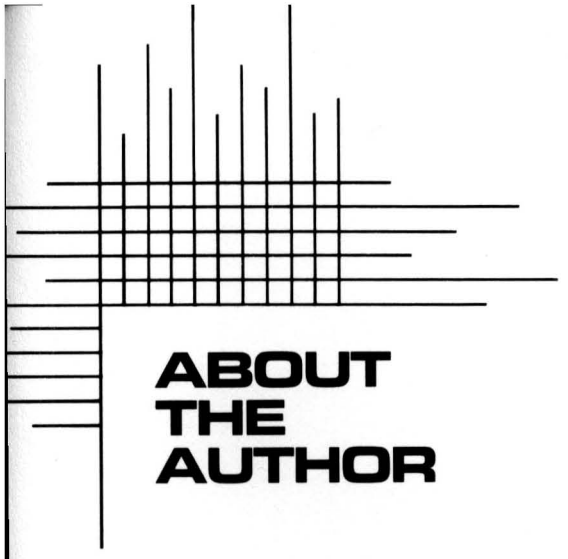
FOR FURTHER READING

Findley, Robert. *6502 Software Gourmet Guide & Cookbook.* Hasbrouck Heights, NJ: Hayden, 1979.

Leventhal, Lance A. *6502 Assembly Language Programming.* Berkeley, CA: Osborne/McGraw-Hill, 1979.

Zaks, Rodney. *Programming the 6502.* Berkeley, CA: Sybex, 1980.

INDEX



ABOUT THE AUTHOR

Christopher Lampton is the author of more than twenty books for Franklin Watts, including a number of popular First Book and Impact titles. Of late he has turned his attention to the world of computers, writing on a variety of programming languages and teaching the basics to beginners.

Chris first became a computer enthusiast when he purchased a Radio Shack computer to use for word processing. He has since acquired eight more computers.

Chris lives in Maryland, just outside Washington, D.C., and has a degree in broadcast journalism. In addition to his writings in the areas of science and technology, he has authored four science-fiction novels.

531-04923-X
AA09