HOW TO PROGRAM THE APPLE II
USING 6502
ASSEMBLY LANGUAGE
With an Introduction to Sweet-16

DATAMOST
by Randy Hyde

DATAMOST
8943 Fullbright Ave., Chatsworth, CA 91311 (213) 709-1201

cover

****

USING 6502
ASSEMBLY
LANGUAGE

****

USING 6502 ASSEMBLY LANGUAGE

How Anyone Can Program the Apple II

By Randy Hyde

A Product of

DATAMOST, INC.
8943 Fullbright Avenue
Chatsworth, CA 91311
(213) 709-1202

1st Printing October 1981
2nd Printing December 1982

****

-ACKNOWLEDGMENTS-

This book represents many hours of dedicated work by myself and everyone involved in its generation. While their names do not appear on the cover, special credit is due to David Gordon, Larry Bouyer, and my wife Mandy. The management and marketing efforts by Dave made this book possible (although it took a long time...). Larry and Mandy transformed a computer programmer's "illiterate" rough draft into this document. Many thanks also to Glynn Dunlap, whose wonderful cartoons added greatly to this book. I owe these four people a great deal.

The material included in Appendix A is reproduced with the permission of Apple Computer, Inc. It is originally printed in "The Apple II Reference Manual" copyrighted by Apple Computer. Thanks is hereby given to Apple Computer for allowing reproduction herein.

****

TABLE OF CONTENTS

NOTE
An alphabetical index is located in the back of this manual.

****

****

****

****

APPLE II COMPUTER TABLES,
CHARTS, AND GRAPHS A-1

****

CHAPTER 1

INTRODUCTION

PURPOSE OF MANUAL.

This manual provides 6502 assembly language instructions addressed directly to APPLE II computer applications. The information contained herein is intended for use by beginning, intermediate and advanced programmers.

SCOPE OF MANUAL.

This manual contains explanations of basic symbols and terminology used by programmers and engineers. Included is an introduction to computer concepts, simple assembly language instruction examples, and detailed 6502 assembly language instructions as related to APPLE II computer requirements.

GENERAL.

Why another book on 6502 assembly language? Well, there are several reasons. First, there were only two books available on the subject when I began writing this book. Second, none of the available books address themselves directly to the APPLE II computer. While assembly language theory can be learned from books, examples that run on other computers using 6502 assembly language are of little use to the APPLE II computer owner.

This book is the product of my experiences as a 6502 assembly language instructor. The material chosen for this book is easily learned by the beginner. No promises can be made concerning your individual levels of expertise achieved after reading this book, but the material presented here should raise you to the level of an intermediate 6502 assembly language programmer. The "expert" status is achieved only through years of experience.

This book is intended for the beginner. Intermediate and advanced programmers may find several items of interest in this book, but it was written with the beginner in mind. If you have had

****

prior 6502 experience, the first few chapters may contain information which you have seen previously. AVOID THE TEMPTATION TO SKIP ANY MATERIAL! If one important detail is not understood, the remainder of the book may prove impossible to

understand. So take the time to review all of the available material and make sure that you understand the reviewed section before going on. Obviously, if you are a beginner it is very important that you understand each section before continuing.

Since there are so many excellent books on computer theory, microcomputers, etc., I will try to keep the discussion of these subjects to a minimum. There are several books you should own if you are interested in learning 6502 assembly language. Books I highly recommend include:

HOW TO PROGRAM MICROCOMPUTERS
by William Barden Jr.

PROGRAMMING THE 6502
by Rodney Zaks

PROGRAMMING A MICROCOMPUTER
by Caxton C. Foster

6502 ASSEMBLY LANGUAGE PROGRAMMING
by Lance Leventhal

6502 SOFTWARE GOURMET GUIDE & COOKBOOK
by Robert Findley

While all of the previously mentioned text books are excellent, they were not written with the APPLE II computer in mind. This text presents practical applications instead of just the theory. Since each of the above books present 6502 assembly language in a different manner you may refer to them should you encounter any difficulties understanding the material presented here. If you are serious about learning assembly language you should have access to the previously mentioned text books as well as this manual.

Before getting into assembly language, it would be very wise to aquaint you with some of the 'jargon' that will be used throughout this manual.

****

RAM: User memory. Programs and data are stored in the RAM.
(RAM is an acronym for Random Access Memory)

ROM: Used to hold the Apple monitor and BASIC. You cannot store data or programs in the ROM.
(ROM is an acronym for Read-Only Memory.)

MONITOR: A set of subroutines in ROM which allow you to read the keyboard, write characters to the video screen, etc.

BASIC: When the word "BASIC" is used, it means Integer

BASIC. Applesoft BASIC is referred to as "Applesoft."

K: When "K" is encountered, you simply substitute "x 1024" (i.e, multiplied by 1024). Generally used to denote a memory size (such as 48K).

MEMORY: Combination of all RAM and ROM locations.

SIGNED NUMBER: Any legal positive or negative integer ("legal" as defined by the current operation).

UNSIGNED NUMBER: Any legal positive (only) number. Negative numbers are not allowed.

BYTE: One unit of memory. A byte can represent up to 256 different quantities (such as the numbers 0-255).

WORD: Two bytes stuck back to back. With a word you can represent up to 65,536 different quantities (such as the numbers 0-65,535 or the signed numbers (-32768) to (32767)).

SYNTAX: The rules governing sentence structure in a language, or statement structure in a language such as that of a compiler program.

ADDRESS: Two bytes used to point to one of the 64K available memory locations in the APPLE II computer. An Address is also a Word but a Word is not necessarily an Address.

PAGE: The 65,536 bytes in the address range of the APPLE II computer are broken into 256 blocks blocks of 256 bytes each. These blocks are numbered 0 to 255 and are called pages.

ZERO PAGE: The first 256 bytes in the memory space (page number 0) of the APPLE II computer are often referred to as the "zero page" or "page zero." Naturally there is a "page one," a "page two," etc., but the use of the first 256 bytes in the machine occurs so often that the term, "zero page," has come into common use.

****

SLOT: One of the peripheral connectors (0-7) on the APPLE II computer.

I/O: An acronym for input/output.

LISA: An acronym for Lazer Systems Interactive

Symbolic Assembler, pronounced LI ZA,
not LE SA.

PERIPHERAL: An I/O device (such as a disk or printer) connected externally to the computer.

It is assumed, in this manual, that the reader is familar with Apple BASIC. BASIC will only be used in a few examples, but familiarity with BASIC means that you have mastered at least the elementary programming techniques. Assembly language is not the place for an absolute beginner to start. You should be somewhat familar with programming concepts before attacking assembly language. Assembly language is a very detailed programming language and it is easy to get lost in the details if you are trying to learn elementary programming at the same time.

Learning any program language, especially assembly language, requires "hands-on" experience. All of the examples presented in this book use LISA (a disk-based 6502 assembler for the APPLE II computer). LISA is excellent for beginners because it is interactive, meaning it catches syntax errors immediately after the line is entered into the system. This is very much like Integer BASIC in the APPLE II computer. Since LISA catches syntax errors, learning assembly language will be easy. It is doubtful that you will ever "outgrow" it. This is not true for many other assemblers available for the APPLE II computer. If you decide to purchase an assembler now, keep in mind that, for the most part, you are stuck with it for life, since none of the assemblers available are compatible with one another. So software which you create on one assembler cannot be loaded into another assembler, even though they are both for the APPLE II computer! Even if LISA

****

were not interactive, I would still recommend it, since it is very powerful and will suit your needs for quite a while to come.

WHY USE ASSEMBLY LANGUAGE?

The fact that you have read the text this far shows that you have an interest in the subject. Nevertheless, some of you are certain to have some misconceptions about the language. Assembly language should be used when speed is the foremost requirement in a program, or possibly when you need to control a peripheral device, or maybe you have a specialized application that cannot be executed easily (or cleanly) in one of the high-level languages on the APPLE II computer.

You should not use assembly language for business or scientific purposes. Pascal, FORTRAN, or Applesoft are better suited for these applications. Floating point arithmetic, although not impossible or even especially hard, is not something a beginner, or even an intermediate programmer would want to tackle.

Another advantage provided by assembly language programs is the possibility of interfacing them to existing BASIC, Applesoft, and Pascal programs. You can program the time critical sections of code in assembly language; the rest of the code can be written in BASIC.

Once you become experienced in assembly language programming you will discover that you can write and debug assembly language programs as fast as BASIC programs!

Good luck. Hopefully, you will find machine language programming as easy as BASIC!

LISA is available from your local computer store, or directly from:

DATAMOST, INC.
8943 Fullbright Avenue
Chatsworth, CA. 91311
(213) 709-1202

****

CHAPTER 2

SYMBOLISM

GENERAL.

When you see the number 4, what do you think? The number 4 is simply a symbol connected with the concept of four items. When humans communicate, they use several symbols to relay their ideas. As such, humans are very adaptive. If I told you that from now on we'll use the symbol "- -" to represent four, you could make the change. It might not be easy, but the change is possible.

Computers, on the other hand, are very stupid. They are not adaptive and understand only a very low-level language which humans have considerable trouble understanding. This language is not "assembly" or "machine" language. Assembly, or machine language, is actually a human convention that makes an even lower-level language acceptable! The actual low-level language understood by a computer consists of different voltage levels on different wires within the machine. Although, with lots of education, humans can understand what each of these voltage levels mean (and in fact your friendly neighborhood computer repair man should), it certainly isn't very convenient. As such, we usually

****

rename the voltage levels something else (bits, true, false, 0, 1, etc.). We do the same thing in spoken languages all the time. For instance, "deux" (French) usually gets translated to "two" (English). Renaming voltage levels "bits" and groups of bits "words" performs this same function. We're merely taking one symbol, which is hard to understand, and translating this symbol to one easier to understand.

The translation occurs in several distinct steps. These steps include:

```
VOLTAGE    =>   BINARY   =>   CHARACTERS
 LEVELS    =>   DIGITS   =>    NUMBERS
(+5v,0v)   =>    (0,1)   =>     ETC.
```

Note that this translation is not performed by the computer. It is performed by humans. Remember, computers are dumb.

Once we realize that computers only represent "things" with voltage levels, a natural question is: 'How do we represent "things" with voltage levels?' Well, as it turns out, representing binary digits (or bits) is really quite simple. We have two voltages (+5v and 0v) and two binary digits (0 and 1) to work with. Since we have a one-to-one correspondence, we'll just arbitrarily assign "1" to +5v and "0" to 0 volts. The assignment is perfectly arbitrary. We could have defined the binary digit "0" to be +5v and the binary digit "1" to be 0 volts. By convention (which means everyone has more or less agreed upon it), however, we'll stick to the former definition.

With one bit, we can represent two different values or "states." Examples include the so-called Boolean values (true or

****

false), signs (+ or -), yes or no, on or off, and any other user-defined binary quantities (husband/wife, boy/girl, ... you get the idea).

Now that we have a bit to play around with, would you like to play around a bit? Let's define some operations on this bit. First, we need to define an ordinality for our binary values. This is necessary because often we need to compare one value to another to determine which is the greater. "0" and "1" are easy, one is always greater than zero. For the other binary values we need to use our intuition to decide on the ordinality. "True" should be greater than "false," so let's assign true the value "1" (or +5v) and false the value "0" (or 0v). Yes/no, on/off, etc., should be assigned in a similar manner. When it comes to data types, such as male/female, the choice is arbitrary. If you're a male you'll probably pick the "male" data type as being larger; if you're a female you'll probably pick "female" as being the greater value.

Keep in mind that our usage of +5v and 0v becomes very context-dependent. Sometimes +5v will be used to denote the

number "1," other times it will be used to denote the "true" value and in other instances it will be used as "on," etc. Try not to get confused about the type of data you are trying to represent as this can cause all kinds of problems. From this point on I will universally use "1" to denote +5v and "0" to denote 0v. For example, when I say that "true" is defined as the value "1," I really mean that true is defined as +5v.

BIT STRINGS.

Up to this point we have limited ourselves to one binary digit, or "bit." Although there are several applications where one bit provides enough information for our needs, there are other times when we need to represent more than two different values. A good example would be the base ten digits (0 thru 9). In this example we need to represent ten different values but our bit can only supply us with two. Well, why not use more than one bit to represent the different values? Specifically, let's use 10 bits and label them 0 thru 9. Now, to represent the digit "5," for example, we can set the sixth bit to "1" (leaving all others zero). To represent the value "0" we would set the first bit to "1," leaving the rest "0." To represent the digit 9 we would set the tenth bit to "1," leaving all others at "0."

****

Each decimal digit would require 10 bits and would be laid out as follows:

DECIMAL DIGIT	BIT NUMBER 0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

Note that the bits are numbered 0 thru 9. When numbering bits within a bit string, we will always start at bit number 0. Bit number 0 is the first bit, bit number 1 is the second bit, ..., bit number 9 is the tenth bit, etc. It is possible to have only a single bit "set" (set means equal to one) in our bit string. A value of 100100100 is not defined. This scheme would probably work just fine, except it is not very efficient. We have a unique string of bits for each value, but as we have defined it here there are several combinations that are unique but undefined. Since each bit we use will cost us money (since it takes one of those 16K RAM chips to equal one bit) we would like to define a bit string which

****

uses memory efficiently, thereby lowering the cost of our computer.

To make our discussion easier to understand, let's just consider two bits. As per the previous discussion we can represent two different values with the two bits, zero and one. Wait a minute! Previously We discovered that we could represent two different values with only one bit! This means, that right off the bat, we are wasting at least half of our memory! So why don't we define the numbers zero and one as having the following two-bit values:

value	bit string
0	00
1	01

Note that we are using the value and simply tacking on a leading zero. Now consider the following bit strings:

value	bit string
?	10
?	11

Notice that the value is undefined. We can't use zero or one because these two bit strings are quite obviously two different values from zero and one as previously defined.

Since we now have two additional values, why not use them to represent the values two and three? If we do this, we wind up with the following:

value	bit string
0	00
1	01
2	10
3	11

So now we can represent four different values with only two bits! We save two bits over the previous method by defining our data this way!

Now suppose we use a bit string of length three to represent our values. As before, if the left-most bit is zero, we can simply ignore it (the left-most bit is often called the "high-order" bit). This leads to:

value	bit string
0	000
1	001
2	010
3	011
?	100
?	101

?	110
?	111

****

Notice that we now have FOUR undefined values. Continuing as expected, we will define these next four values to be the values 4 thru 7. Now we are saving quite a bit of memory. Remember, previously it took eight bits to represent the values 0 thru 7, now it only takes three! We have cut our memory usage down to almost one third of that previously required! Since we want to be able to represent the decimal digits 0 thru 9, it looks like we will need to add another bit to our bit string since three bits can only represent the values 0 thru 7. Upon appending this extra bit we obtain the following:

value	bit string
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
?	1000
?	1001
?	1010
?	1011
?	1100
?	1101
?	1110
?	1111

By adding the extra bit we have added EIGHT new values to our number system. We only needed two more values however! Since we now have 16 different values on our hands, we can represent the values 0 thru 15. But, since we only needed to represent the values 0 thru 9, we will leave the bit combinations 1010 thru 1111 undefined. Yes, we are wasting some memory, but remember, we only wanted to represent the values 0 thru 9 so the waste can be considered undesirable, but required in this case. Notice the final memory savings - only four bits are required as opposed to ten! In general, each time we add a bit to our bit string we DOUBLE the number of possible combinations. For instance, with eight bits we can represent 256 different values, with ten bits we can represent 1024 different values, and with 16 bits we can represent 65,536 different values.

We have just invented the binary numbering system which is used by computers! Each bit in our bit string represents a power of two.

****

```
     7   6   5   4   3   2   1   0

      7   6   5   4   3   2   1   0
     2   2   2   2   2   2   2   2
```

The first bit represents 2^0 (any number raised to the power "0" is one), the second bit represents two raised to the first power (i.e, 2^1), the third bit represents two raised to the second power (2^2), etc. For example, binary 1100101 represents 1x2^6 + 1x2^5 +0x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0 or 101 in decimal.

With eight bits we can represent up to (128) + (64) + (32) + (16) + (8) + (4) + (2) + (1) plus one (since we can also represent zero which is distinct from all the other values) or 256 different values. In general, to represent 2^n - 1 distinct values (such as the numbers 0 to 2^n - 1) we will need n bits. For instance, to represent the ten decimal digits 0-9, three bits are not enough as (2^3) - 1 equals 7, we still need two more values. In order to get these two extra values we must add another bit even if it means some of the available combinations must be wasted. Converse to all of this, if we are limited to n bits we can only represent 2n different values (such as the numbers 0 to (2n) - 1).

Remember, we can represent quantities other than numbers with our bit strings. For instance the colors RED, BLUE, YELLOW, and GREEN as follows:

COLOR	BINARY CODE
RED	00
BLUE	01
YELLOW	10
GREEN	11

Or possibly the alphabetic characters:

Character	Binary Code
A	00000
B	00001
C	00010
D	00011
E	00100
F	00101
.	.
.	.
.	.
X	10111
Y	11000
Z	11001
(UNUSED)	11010
(UNUSED)	11011
(UNUSED)	11100
(UNUSED)	11101
(UNUSED)	11110
(UNUSED)	11111

****

Since there are 26 characters, we'll need 5 bits (2^5=32). Four bits simply aren't enough (2^4=16).

BINARY ARITHMETIC.

Now that we know how to represent data, let's see how to manipulate this data.

BASIC ADDITION RULES:

First let's review what happens when we add two numbers in the decimal (base ten) system. If we were to add 95 and 67, we would perform the following steps:

-First we add 5 and 7

```
 95
+67  add 5 to 7
---
  2  result is 2, carry is 1.
```

Next, we add 9 and 6, plus one since there was a carry.

```
 95
+67  add 9 to 6 plus one (from the carry).
---
 62  result is 6, carry is 1.
```

After the carry is added in, we get the final result of 162.

Binary addition works the same way, but is even easier. It's based on seven rules:

1) 0 + 0 = 0; carry = 0
2) 1 + 0 = 1; carry = 0
3) 0 + 1 = 1; carry = 0
4) 1 + 1 = 0; carry = 1
5) 0 + 0 + carry = 1; carry = 0
6) 1 + 0 + carry = 0; carry = 1
7) 1 + 1 + carry = 1; carry = 1

So, now we can add any n-bit binary quantity as follows:

STEP 1) Add 0 to 1 in the first column, which generates 1, carry = 0.

```
 0110
 0111
 ----
    1  C = 0
```

****

STEP 2) Add 1 to 1 in the second column, giving zero and carry = 1.

```
  0110
  0111
  ----
    01  C = 1
```

STEP 3) Add 1 and 1 plus 1 (from the carry). This gives us 1 and the carry remains set (equal to one):

```
  0110
  0111
  ----
   101  C = 1
```

STEP 4) Add 0 to 0 plus 1 (from the carry). The result is one, and the addition is complete.

```
  0110
  0111
  ----
  1101  C = 0
```

This procedure can be carried on for any number of bits. Examples of binary addition:

```
  01101100        1101101
  11101011        1111011
  --------        -------
 101010111       11101000
```

UNSIGNED INTEGERS.

Up to this point we've made the assumption that we have as many bits as we need at our disposal. In the 'real' world, this is simply not the case. Usually we are limited to a fixed number of bits (usually 8 or 16). Due to this restriction, the size of our numbers is limited. With 16 bits we can represent numbers in the range 0 to 65,535 (2^16 - 1 = 65,535). With eight bits we can represent values in the range 0 to 255. Since the 6502 is an 8-bit machine (we are limited to using 8 bits at a time), it would seem that we can only handle numbers in the range 0-255. Luckily this is not entirely true, multiple precision routines will be studied later on. An unsigned integer will be defined as any value between 0 and 65,535, so an unsigned integer will need 16 bits.

****

NIBBLES (NYBBLES?), BYTES, and WORDS.

In our discussions, we will often use bit strings of length 4,

8, and 16. These lengths are not arbitrary, but rather they are dependant upon the hardware being used. The 6502 likes its data in chunks of 4, 8, and 16 bits.

Since we use these lengths all the time, we have special names for them. A "NIBBLE" is a bit string of length four. As you may recall from the previous discussion, it takes at least four bits to represent a single decimal digit. Sometimes decimal numbers are represented by strings of nibbles (i.e, groups of four bits) in a form known as binary coded decimal. Binary coded decimal arithmetic is possible on the 6502 and will be discussed later. Often, binary coded decimal is abbreviated to BCD.

A "BYTE" is a bit string of length eight. The byte is the most common data type used by the 6502 because the data width of the 6502 is eight bits (that is, the 6502 is an eight bit processor):

A "WORD" is a bit string of length 16. Words are used primarily to hold addresses and integer values. With a word it is possible to represent up to 65,536 different values (64K). This is the reason the 6502 can directly address up to 64K of memory.

Note that there are two nibbles in a byte and two bytes in a word. This generates some additional terminology. Each bit string has a low-order bit and a high-order bit. The low-order bit is always bit number 0, and the high-order bit is equal to (n - 1) where n is the number of bits in the bit string. For a nibble, n is four so the high-order bit is bit number three (remember, we start with zero!). For a byte (n = 8) the high-order bit is bit number 7 and for a word (n = 16) the high-order bit is bit number 15.

EXAMPLES:

```
Bit #   15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
-----------------------------------------------

                                          NIB
                                        1 0 1 0

                                         BYTE
                                    0 0 1 1 0 0 1 1

                         WORD
                  0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 1
```


****

Additional terminology results from the symmetry of nibbles, bytes, and words. Since there are two nibbles in every byte, we can speak of a "high-order nibble" and a "low-order nibble." The low-order nibble is comprised of bits 0 thru 3 and the high-order nibble is comprised of bits 4 thru 7 in any given byte. Likewise, the low-order byte in a word consists of bits 0 thru 7 and the high-order byte consists of bits 8 thru 15. These definitions come in handy when we have to work with data in groups of eight bits, and

it's nice to be able to relate words and nibbles to bytes.

SIGNED INTEGERS.

On many occasions a range of zero to (2n - 1) is simply not enough. To represent values larger than (2n - 1) all we need to do is add additional bits to our bit string and the range of our numbers is increased proportionately. But sometimes we need to be able to represent numbers less than zero. Unfortunately, this cannot be accomplished with the number system we have described so far. In order to represent negative numbers we must abandon the binary numbering system we have created and devise a new numbering system that includes negative numbers.

While many numbering systems exist that allow negative numbers, we are forced to use the so-called two's complement numbering system. This choice has to be made because of the 6502 arithmetic hardware. The two's complement system uses the following conventions:

1) The standard binary format is used
2) The high-order bit of a given binary number is assumed to be the sign bit. If this bit is set, the number is negative. If this bit is clear, the number is positive.
3) If the number is positive, its form is identical to the standard binary format.
4) If the number is negative, it is stored in the two's complement format.

The two's complement format is achieved by taking a positive number, inverting all the bits (that is, if a bit is zero change it to one; if a bit is one change it to zero), and then adding one to the inverted result. For example, given that the positive 16-bit representation for two is:

0000000000000010

****

then the two's complement of two (i.e, minus two) is computed by inverting all the bits:

1111111111111101

and adding one to the inverted result:

1111111111111110

Therefore, 1111111111111110 is the two's complement representation for minus two. The two's complement operation, also called negation, can be thought of as a multiplication by minus one. In fact, if you take the two's complement of a negative number, you wind up with its positive counterpart. Consider minus two:

1111111111111110

To take the two's complement of minus two, we first invert all the bits:

0000000000000001

Next, one is added to the result so that we obtain:

0000000000000010

which is the binary representation for two!

Why even bother with such a weird format? After all, it's probably much simpler to just use the high-order standard binary format. Well, a simple addition problem may help clear things up. Consider the addition of two plus minus two.

```
0000000000000010
1111111111111110
----------------
0000000000000000  carry = 1
```

Note that if we ignore the carry out of bit #15, we wind up with a zero result, exactly what we expect. It is easy to prove to ones self by the use of examples that if the carry is ignored, the result is always what one would expect.

If the carry out of the sixteenth bit is meaningless, how does one detect an overflow? If the sign bit is treated as a separate entity from the rest of the number, bit #14 is technically the high-order bit. A carry out of this bit will be what we test for to determine two's complement overflow.

****

HEXADECIMAL NUMBERS.

Binary numbers are fine for examples. But when used for conveying information to people, they tend to be too bulky. Can you imagine having to write out one hundred 16-bit numbers in binary? Or having to read them? Several years ago programmers began using the octal (base eight) numbering system to compact the large binary numbers. With the octal system it is possible to cram 16 bits of information into six digits. The octal numbering system is still popular on several minicomputers today. When microcomputers came along, manufacturers switched to the hex-adecimal numbering system which made it possible to get 16 bits of information into only four digits! The only drawback to the hex-adecimal numbering system is that most people are not familiar with it. The hexadecimal system (base 16) contains 16 distinct digits. The first ten digits are the familar numeric characters 0 thru 9 and the last six digits are the alphabetic characters A thru F. Hexadecimal numbers have the values:

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

Why all the fuss over hexadecimal numbers (or hex numbers as they are usually referred to)? They are easy to convert to binary and vice versa. Decimal numbers, unfortunately, are not as easy to use. For example, 11111100 is not easily converted to 252 decimal, but it is a trival matter to convert it to the hexadecimal number FC. Clear as mud, right? It's actually quite simple once you learn one little trick. In order to convert a binary number to a hexadecimal number you must first adjust the binary number so that it contains the number of bits which are a multiple

****

of four (four, eight, twelve, sixteen, etc.). You accomplish this by adding leading zeros to the left of the binary number. Next, you start from the right and divide the bit string into groups of four bits each. Look up each of these "quadruples" in the chart above and replace them with the corresponding hexadecimal value. In the previous example, 11111100 is split up into two groups of four bits yielding 11111100. Looking up 1111 in the chart yields the hexadecimal digit "F". The binary number 1100 corresponds to the hexadecimal digit "C".

Going in the other direction, converting hexadecimal to binary, is just as easy. Simply look up the binary equivalent of each hexadecimal character in a hex string and substitute the binary value. Don't forget to include leading zeros in the middle of a hex string. For example, EFC4 converts to 1110 1111 1110 0100.
Although hexadecimal numbers may seem cumbersome to the new programmer, they are in fact a great convenience.

RADIX AND OTHER NASTY DISEASES.

Now we have decimal, binary, and hexadecimal numbers. If you were to find "100" printed somewhere, how would you be able to tell which base, or "radix," the number is represented in? Does "100" mean 100 base two (ie., decimal four), 100 base 10 (i.e., one hundred), or "100" hex (i.e., 256 decimal)?

To avoid confusion the radix is usually specified by some leading character. If a number is prefaced by a percent sign the number will be considered to be a binary number. If the number is preceded by a dollar sign the number will be assumed to be hexadecimal. A exclaimation point is used to denote a decimal number. Decimal numbers may also appear without a radix prefix, so if a string of digits appears without a leading radix character the decimal number system is assumed. The use of the radix prefix prevents ambiguity.

ASCII CHARACTER SET.

As has been continually pointed out, binary values may be used to represent values other than numeric quantities. A computer is required to handle text consisting of alphabetic charac-

****

ters, numeric characters, and several punctuation symbols as often as it must perform numeric manipulation. Since character manipulation is very important, we must define a character set, that is, a set of unique binary values for each of the valid characters we wish to represent.

As you may remember, it requires a minimum of five bits (or 32 distinct values) to represent the characters of the alphabet. When you add to that the numeric characters 0 thru 9, it becomes apparent that six bits are going to be required. When you add the lowercase letters and several punctuation characters, the number of required characters jumps to 96. Finally, by adding several "device-control" characters such as return, cursor control, tab, the total jumps to 128 characters. To represent 128 different values requires seven bits. To allow other special characters (such as inverted or blinking characters) another bit will be used to bring the bit total to eight bits, yielding a maximum of 256 distinct characters.

Now the only problem that remains is to assign these 256 different characters a unique 8-bit code. Rather than create our own character code, we will use the American Standard Code for Information Interchange (ASCII) character set. The ASCII character set is used by almost all computer manufacturers. Even IBM, which has used its own character set since the early sixties, has finally started using ASCII characters in some of its equipment. The first 32 values in the ASCII character set are the so-called control codes. These include carriage return, line feed, backspace, tab, and several other non-printing characters reserved for device control use. The next 32 characters are reserved for the often used punctuation characters (such as period, comma, space) and the numeric characters. The following 32 characters are reserved for the uppercase letters and some infrequently used punctuation characters. The final 32 values in the ASCII character set are reserved for the lowercase letters and some little-used punctuation characters.

ASCII does not define the final 128 characters in the character set. These are user-definable characters. On the Apple II, the remaining characters comprise the inverted and blinking character set. For a full description of the Apple/ASCII character set, see Appendix A.

****

USING BIT STRINGS TO REPRESENT INSTRUCTIONS.

Until now we have assumed that bit strings are used only to represent data of some type. This is not always the case. A bit string can also be used to represent a command.

Imagine, if you will, a small subset of the commands humans obey every day. One command might be the alarm clock ringing in the morning, causing you to get out of bed. A second command might be, "Get dressed." A third command could be, "Drive to work." A fourth command could be, "Perform all actions required at work." Another command could be, "Drive home from work." And a last command could be, "Go to bed." To represent these six commands we need three bits. The commands could be assigned as follows:

bit string	command
000	Get out of bed.
001	Get ready for work.
010	Drive to work.
011	Perform required duties.
100	Drive home from work.
101	Go to bed.

With these simple commands the apparent actions of a human being can be performed. Each command will be assumed to be given sequentially. This does not mean numerically (i.e, in the order given above), but rather it means that the human executes one instruction at a time. Although it may not make much sense, it is perfectly valid to give the commands out of numerical order. For example, suppose the person drove to work and then realized that he left something at home which was required to perform his job-related duties. This situation would require the instruction sequence:

000	Get out of bed.
001	Get ready for work.
010	Drive to work.
100	Drive home and pick up forgotten items.
010	Drive back to work.
011	Perform required duties.
100	Drive home from work.
101	Go to bed.

Obviously, several other schemes are possible with some

yielding weird results. Commanding objects other than human

****

beings is also possible. Examples include automated machinery, programmable toys, and, of course, the computer. The fact that commands can be represented as bit strings is the whole basis for the computer programming to be studied in the following chapters.

****

CHAPTER 3

REGISTERS, INSTRUCTION FORMATS, AND ADDRESSING

GENERAL.

Up until now, our discussion of data types has been, for the most part, unrestricted. Unfortunately, in the "real" world of computers several restrictions apply which limit the size and feasibility of the operation we wish to perform. In order to be able to write good programs the user must first learn the limitations, and advantages, of the APPLE II computer.

The APPLE II computer consists of three major parts:

1) Central Processing Unit (6502 Microprocessor)
2) Input/Output (Keyboard, Video Display, Disk, Etc.)
3) Memory

Memory in the APPLE II computer is arranged as 65,536 8-bit bytes. Each byte is individually addressable; that is, if we want to, we can perform our data operation on any of the 65,536 locations available to us.

Several of these locations (5120 in fact) are specifically reserved for Input/Output (I/O) purposes 1024 of these locations comprise the screen memory, and storing data in any of them (located from $400 thru $7FF in memory) is likely to affect the video display. Another 4K (4096) of these memory locations is reserved for use by the peripheral cards which plug into your Apple. The remaining 59K bytes (ie, 60,416 bytes) are used to hold variables, your program, BASIC, Pascal, etc. Typically, the user has 48K at his disposal for program storage (minus any language requirements such as DOS, etc.).

The Central Processing Unit (CPU) is where all the action takes place. The CPU is the "brains" behind the computer. Data

****

is transferred to and from memory and I/O devices, arithmetic is performed, comparisons are made, etc., within the CPU. So, the CPU will function as a "middleman" in most of our operations.

Let's define the 6502 microprocessor. Internally the 6502 microprocessor consists of an Arithmetic/Logical Unit (ALU) where additions, subtractions, etc., take place, a control unit

****

which moves data to and from memory, decodes the instructions, and accesses six special memory locations called, registers. Five of these registers are 8 bits wide (just like our memory) and one of them is 16 bits wide (the same as the 6502 address bus).

These six registers each serve a special purpose, therefore they have been given special names as follows:

1) Accumulator (A or ACC)
2) X-register (X)
3) Y-register (Y)
4) Stack Pointer (SP)
5) Program Status Word (P or PSW)
6) Program Counter (PC)

A separate description of each register is given in the following paragraphs:

ACCUMULATOR (A or ACC).

The accumulator is where most of the data transactions occur. Numbers are added and subtracted here. Data transfer from memory location to memory location usually goes through the accumulator. All logical operations occur in the accumulator. For most of our purposes, the accumulator will be the general purpose register that we utilize.

X-REGISTER (X).

The X-register in the 6502 is a special purpose register. We cannot add or subtract numbers with it, however the X-register is used for accessing elements of simple arrays, strings, pointers, etc. Using the X-register to access elements of an array is called "indexing." Often, the X-register is called the X-index register. We will discuss indexing later in the text.

Y-REGISTER (Y).

The Y-register, identical to the X-register, is reserved for indexing purposes. Two different index registers allow us to per-

form such functions as substring, concatenation, and other array functions.

****

STACK POINTER (SP).

The Stack Pointer is another special purpose register in the 6502. It is used when calling subroutines and returning from subroutines, as well as when saving temporary data. Since it is 8 bits wide, the stack pointer can only be used to address 256 different locations in the 6502 address space. These 256 locations occur from location $100 to location $1FF.

NOTE

Since locations $100 thru $1FF are reserved for the Stack Pointer register, NEVER use these locations for data or program storage.

PROGRAM STATUS WORD (P or PSW).

The program status word (also called the processor status register) is not a register in the true sense of the word. It is simply a convenient collection of seven status bits which will be used by such things as conditional branches (to be described later).

PROGRAM COUNTER (PC).

The program counter is a register used by the computer to point to the instruction currently being executed. This register is unique in that it is the only 16-bit register on the 6502. It is 16 bits wide since 16 bits are required to access the 65,536 different locations (the address space) on the 6502.

INSTRUCTION FORMAT (6502).

Thus far we have discussed the ways computers store data and where the data is manipulated (i.e., the registers). We have not discussed how we tell the computer what to do with this data. A computer instruction is used to tell the 6502 which operation to perform. What is an instruction? An instruction is simply another 8-bit code stored in memory. Since each instruction is 8 bits wide there is a maximum of 256 possible instructions. In the 6502, however, there are only about 120 actual instructions. The instruction codes corresponding to these 110 to 120 instructions are called valid instruction codes, or valid opcodes. The remaining

****

136 to 146 invalid instructor codes are referred to as the invalid instruction codes, or invalid/illegal opcodes.

The opcodes (computer instructions) are stored in memory in a manner identical to date. How then does the computer differentiate between data and instructions? Clearly, the meaning of a byte in memory is very context-dependant. A byte in memory is assumed to be a computer instruction if the program counter is ever allowed to "point" at (i.e., contain the address of) that particular byte in memory. Also, programs are assumed to be stored sequentially in memory (with some exceptions). That is, the second instruction immediately follows the first instruction, the third instruction follows the second, etc.

EXAMPLE:

```
    MEMORY
1st INSTRUCTION     <- PROGRAM COUNTER
2nd INSTRUCTION
3rd INSTRUCTION
4th INSTRUCTION
5th INSTRUCTION
```

The program counter is loaded with the address of the first instruction. The processor loads and then executes this instruc-

****

tion. The program counter is then incremented by one so that it points to the second instruction. This instruction is fetched and the cycle is repeated.

Always remember that the computer cannot tell the difference between data and instructions. Whatever the program pointer points to will be interpreted as an instruction.

TWO AND 3-BYTE INSTRUCTIONS.

Many instructions require more than one byte. For instance, suppose we want to load the accumulator with the 8-bit constant $FF. The 6502 has an instruction which will load the accumulator with an 8-bit constant. The only problem is how do you specify the constant? Why not immediately follow the instruction with the constant! Well, this is exactly what's done. The hex code $A9, when executed, tells the 6502 to load the accumulator with the 8-bit constant located in the next byte, so the two bytes ($A9, $FF) instruct the 6502 to load the accumulator with the constant $FF. Loading the accumulator with a constant (or load the accumulator immediate, as it's often called) is an example of a 2-byte instruction. Rather than using just one byte to perform the operation, we need two. Naturally, the program counter is incremented by two instead of one so that the constant does not get executed as the next 6502 instruction.

****

In addition to the 2-byte instructions, there are also 3-byte instructions. One good example is the "store the accumulator in an absolute memory location" instruction. This instruction (which consists of $8D followed by a 16-bit address) will store the contents of the accumulator at any of the 65,536 different memory locations available in the 6502 memory space. For example, ($8D, $00, $10) will store the accumulator at location $1000, and ($8D, $C3, $48) will store the accumulator at location $48C3.

Remember, whenever a multibyte instruction is encountered, the program counter is automatically incremented past the additional data.

EXAMPLE:

```
A9 INSTRUCTION #1  LOAD ACC WIUH $FF
FF

8D INSTRUCTION #2  STORE ACC AT LOCATION $1234
34
12

--  ETC.

--

--
```

WARNING

Remember, there is nothing sacred about the location of your program instructions. The computer cannot differentiate between data and valid instructions. In the previous example, if the program began at location $1234 we would have loaded the accu-

****

mulator with $FF and then proceeded to destroy the first instruction ($A9 stored at location $1234) by storing a $FF over the top of the $A9, leaving you with the following code:

LOC	DATA/CODE
1234	FF
1235	FF
1236	8D
1237	34
1238	12
1239	--
ETC.	ETC.

With this in mind, be very careful where you store data since you can easily wipe out your program if you are not careful.

6502 ADDRESSING MODES.

The 6502 microprocessor utilizes 56 distinct instructions. Previously it was said that there are about 120 different instruction codes. Why the difference? Well some operations can be carried out in one of several ways. For instance, one type of operation on the 6502 is that of loading the accumulator with an 8-bit value. The operation is called, "the load the accumulator operation" and is often abbreviated LDA. There are several LDA instructions. You can load the accumulator with a constant, load the accumulator with the value contained in one of the 65,536 memory locations, load the accumulator with an element of an array or string, etc. All of these operations have one thing in common- the end result is that the 6502 accumulator is loaded with a new value. Although the operation is the same (loading the accumulator) the method used to load it is different. Since it is a different operation (so to speak) on a very low level, the 6502 uses a different opcode for each variance of the LDA instruction. These variances on the LDA instruction are often called, "addressing modes." Whereas an instruction tells the computer what to do, the addressing mode tells the computer where to get the data (or operand).

IMMEDIATE ADDRESSING MODE.

The immediate addressing mode tells the computer that the data to be used is an 8-bit constant, which immediately follows the instruction code. Remember, the $A9 in one of the previous

****

examples, $A9 says, "Load the accumulator with the value contained in the following byte." This is an example of the immediate addressing mode. The instruction could be worded as, "Load the accumulator with the byte immediately following the instruction byte." With this wording the term "immediate addressing mode" makes a little more sense. Instructions using the immediate addressing mode are always two bytes long: one byte for the instruction and one byte for the immediate data.

ABSOLUTE ADDRESSING MODE.

Sometimes, rather than loading the accumulator with a constant, we need to be able to load the accumulator with a variable that is stored in memory. As with the immediate addressing mode we need one byte to specify the instruction (LDA or load the accumulator). Next, to be able to uniquely specify one of the 65,536 different locations in the 6502 address space, we need a 2-byte address. This type of addressing mode is called, "absolute addressing mode" (since we are loading the accumulator from an absolute memory location). Obviously this instruction must be three bytes long: one byte for the instruction and two bytes for the address. The actual instruction code for the LDA absolute instruction is $AD. This instruction code is always followed by a 2-byte address; the low-order byte comes first followed by the high-order byte. If we wanted to instruct the 6502 to load the accumulator

from memory location $1234, the code sequence to do this would be: ($AD, $34, $12 or AD3412). Yes, it does look funny seeing the 34 before the 12, but get used to it. You will see this (byte-reversed order) used all the time on the 6502.

ZERO PAGE ADDRESSING MODE.

The 6502 incorporates a special form of the absolute addressing mode known as the "zero page addressing mode." In this addressing mode the 6502 loads the accumulator from the specified memory location, just like the absolute addressing mode. The only difference is that the instruction is only two bytes long: one byte for the instruction and one byte for the address.

****

Since eight bits only allow 256 different values you are limited to 256 different addresses. In the 6502 address space this corresponds to the first 256 locations in the machine (location $00 to location $FF, also known as 'page zero'). Since the zero page addressing mode strongly restricts its usage (you're only allowed to access 1/256th the amount of data possible with the absolute addressing mode), why should you even bother using it? The first part of the answer should be obvious. The absolute addressing mode results in 3-byte instructions whereas the zero page addressing mode uses only two bytes. You save memory by using the zero page addressing mode. The second, and less obvious, reason is that instructions using the zero page addressing mode execute faster than instructions using the absolute addressing mode. Page zero is often used for variable storage, and the other memory locations are often used for program, array, and string storage.

INDEXED ADDRESSING MODE.

As mentioned previously, the X- and Y-registers are used as index registers. An index register is used to access elements of a small array or a string. Remember, in integer BASIC, when you use an array you specify the element of the array by placing an "index" within parentheses after the variable name (e.g., M(I): I is the index). The X- and Y-registers are used in place of the variable I (or whatever you happen to be using). For instance, the instruction code $BD tells the 6502 to load the accumulator from the absolute memory location specified in the next two bytes AFTER the contents of the X-register are added to this value. If the computer executes the instruction sequence BD 34 12, and the X-register contains 5, then the accumulator will not be loaded from location 1234, but rather from location 1239 (1234 + 5). In general, if you have an array (containing less than 256 elements) you can access any element of this array by loading the X-register with the desired value and then loading the accumulator from the first element of the array indexed by X.

NOTE

The Y-register can be used in an identical manner. Naturally, the instruction code is changed, but the effect is the same.

****

INDIRECT ADDRESSING.

The indirect addressing mode is rather tricky. Rather than using the 2-byte address which follows the instruction, we go to the address specified and use the data contained in that location as the low-order byte of the actual address. To get the high-order byte of the actual address we must add one to the 16-bit value following the instruction and go to that address which will contain the high-order byte of the actual address. Now that a low- and high-order byte are obtained, the address is fully specified, and we can continue on our merry way. Yes, this description is worthless and you do need several examples to demonstrate how indirect addressing is used. Rather than give these examples now, their presentation will be deferred until the addressing mode is actually used in a program.

INDIRECT INDEXED BY Y

As with the indirect addressing mode, the indirect indexed by Y mode is mentioned solely for completeness. A full discussion will be presented later in the text.

INDEXED BY X, INDIRECT.

Again, this discussion must be deferred.

IMPLIED ADDRESSING MODE.

The implied addressing mode means exactly that--the instruction itself implies what type of data is to be operated on. Instructions that use the implied addressing mode are always one byte long.

ACCUMULATOR ADDRESSING MODE.

The accumulator addressing mode specifies an operation upon the accumulator. The instructions in this class are all one byte long.

****

NOTE

It may seem that many of the operations in the 6502 should be considered in the class of accumulator addressing mode instructions. The difference between the true accumulator addressing mode instructions and the other instructions is that the accumu-

lator addressing mode instructions reference only the accumulator. They do not require any operands in memory.

RELATIVE ADDRESSING MODE.

The relative addressing mode is used by a group of instructions known as the branch instructions. The description of the relative addressing mode is beyond the scope of this chapter and will be considered in a later chapter. Once again it is mentioned solely for sake of completeness.

ADDRESSING MODE WRAP-UP.

If this discussion of addressing modes doesn't make much sense, don't worry about it. This section was intended only as a crude introduction to make you aware of the fact that addressing modes do indeed exist. The use of a particular addressing mode will become obvious in the next few chapters.

****

CHAPTER 4

SOME SIMPLE INSTRUCTIONS

NEW INSTRUCTIONS:

```
EQU EPZ DFS
LDA LDX LDY STA STX STY INC DEC
TAX TAY TXA TYA INX INY DEX DEY
```

GENERAL.

Until now, everytime we wanted the computer to perform some action, we pulled a magic little number out of the hat and used it as an instruction code. Unfortunately, there are about 120 different instruction codes. Trying to memorize all of these would be mind boggling. It would certainly be quite a bit nicer if we could use phrases like, "load the accumulator with the constant $FF," or "store the contents of the accumulator at location $1234." This idea was so good that several people have indeed done this. LISA is an example of a computer program that takes phrases (such as LDA for load the accumulator) and converts them to one of the 120 or so valid instruction codes. Programs which do this for you are called, "assemblers." Rather than using long phrases, such as "load the accumulator", short mnemonics were chosen instead. Mnemonics are three-character representations of the desired phrases. For instance, LDA replaces "load the accumulator," and STA replaces "store the accumulator." Although you must take the time to learn these mnemonics, the payoff is rather good. When entering a program, you will only have to type three letters instead of an entire phrase!

ASSEMBLY LANGUAGE SOURCE FORMAT.

The actual machine language code that the 6502 understands is often called, "object code." The mnemonics that

****

humans understand are often called, the text file, or source code. Unlike BASIC, which has few restrictions concerning the arrangement of statements on a line, assembly language has a very rigid format. An assembly language source statement is divided into four sections known as "fields." There is a label field, a mnemonic field, an operand field, and a comment field. Fields in an assembler are usually separated by at least one blank; often two or three of these fields are optional.

SOURCE FORMAT:

```
LABEL MNEMONIC OPERAND ;COMMENTS
```

The label field contains a label that is associated with the particular source line. This is very similar to the line number in BASIC. All branches and jumps (a GOTO in BASIC) will refer to this label. Unlike BASIC, this label is not a number, but rather a string, usually one to eight characters long beginning with an uppercase alphabetic character. Labels should only contain uppercase characters and digits.

EXAMPLES OF VALID LABELS:

```
LABEL
LOO1
A
MONKEY
```

EXAMPLES OF INVALID LABELS:

```
1HOLD       (BEGINS WITH "1")
HELLOTHERE  (LONGER THAN 6 CHARS)
LBL,X       (CONTAINS ",")
```

Labels are not required on every line like line numbers in BASIC. Labels are only required when you need to access a

****

particular statement. Labels must begin in column one of the source line, and there must be a blank between the label and the following mnemonic.

As previously mentioned, labels are optional. If you do not wish to enter a label on the current line you must be sure that column one of the source line contains a blank (see the LISA

documentation for furthur details on labels).

MNEMONIC FIELD.

The mnemonic field follows the label field. A three-character LISA mnemonic is expected in this field. These include instructions such as LDA, LDX, LDY, STA, ...

OPERAND FIELD.

The operand field follows the mnemonics field. The operand field contains the address and the addressing mode, if required. If an address appears all by itself the absolute (or zero page, if possible) addressing mode will be used. This address can be an "address expression." An address expression is similar to an arithmetic expression one would find in Integer BASIC except that only addition and subtraction are allowed. (Some versions of LISA allow other operators as well.) For instance, $1000 + $1 will return the value $1001. If you had an instruction of the form "LDA $1000+$1" (LDA stands for load the accumulator), the accumulator would be loaded from the contents of memory location $1001. The discussion of address expressions will be considered in greater detail later in the text.

To specify a constant (the immediate addressing mode), you must preceed a 16-bit address expression with either a "#" or a "/". If you use the "#", the low-order byte of the address expression will be used. If you use the "/", the high-order byte of the address expression will be used as the 8-bit immediate data.

The indexed addressing modes are specified by following an address expression with ",X" or ",Y" depending on whether you wish to use indexed by X or indexed by Y addressing. If possible, the zero page form will be used.

To specify the implied addressing mode, or the accumulator addressing mode, you must leave the operand field blank. Any-

****

thing but a comment (see the next section) will produce an error. The syntax for the indirect, indirect indexed by Y, and indexed by X indirect will be considered later.

COMMENT FIELD.

Following the operand field you can optionally place a remark on the same line as the instruction. Just make sure that the operand field and the comment field are separated by at least one blank, and also make sure that your comment begins with the special character ";" (semicolon).

INTRODUCTION TO REAL INSTRUCTIONS.

So far, we've only discussed assembly language in a very

general way, making use of relatively few concrete examples. Now, let's focus our attention on some of the real commands at our disposal.

LOAD GROUP.

There are three distinct instructions in the load group category. They are: LDA (load accumulator), LDX (load the X-register), and LDY (load the Y-register). These instructions go to the location specified in the operand field, make a copy of the data stored there, and then enter this data into the specified register.

To load the accumulator with the data (contained in one of the 6502's 65,536 different memory locations) simply follow the LDA instruction with the address of the desired memory cell.

```
LDA $1FA0 -- LOADS ACC FROM LOCATION $1FA0.
```

Note that the content of the specified memory location is not altered. A copy is made and placed in the accumulator; the memory location's data is not altered. In general, LDA $nnnn (where nnnn is a one to four digit hex number) will load the accumulator from location $nnnn.

Examples:

```
LDA $11F0 - LOADS THE ACC FROM LOCATION $11F0
LDA $127F - LOADS THE ACC FROM LOCATION $127F
LDA $0    - LOADS THE ACC FROM LOCATION $0000
```


****

Loading a constant into the accumulator is just as easy. Simply preceed the constant with the "#" or the "/" depending on whether you wish to load the low-order eight 8 bits or the high-order eight bits of the 16-bit expression given in the operand field.

Examples:

```
LDA #$1000 - Loads the ACC with the value $00
             ($00 is the low-order byte of
              $1000, the high-order byte,
              $10, is ignored).

LDA #$FF   - Loads the ACC with the value $FF.
             $FF is really $00FF.  The high-order
             byte in this case is $00, once
             again, it is ignored.

LDA #$0    - Loads the ACC with the value $0.
             $0 is really $0000, whose low-
             order (as well as high-order
             byte) is zero.

LDA /$1000 - Loads the ACC with $10.  $10 is
```

the high-order byte of the value $1000. The low-order byte ($00) is ignored.

LDA /$FF - Loads the ACC with $00. $FF is really $00FF whose high-order byte is $00. The low-order byte ($FF) is ignored.

LDA /$0 - Loads the ACC with $00. Both the low- and high-order bytes of $0 (same as $0000) are $00.

In all of the previous examples, the operation performed was that of loading the accumulator. You can load the X-register or the Y-register in a similar manner simply by substituting the LDX (Load the X-register) or the LDY (Load the Y-register) instruction in place of the LDA instruction. There are several other methods used in loading registers (i.e, different addressing modes) other than the ABSOLUTE and IMMEDIATE addressing modes described here. These methods will be considered in later chapters.

STORE INSTRUCTIONS.

Now that we can move data into the accumulator, let's discuss how to store data from the accumulator, X- or Y-register into external memory. The 6502 store instructions provide us with this

****

capability. There are three store instructions: STA (store the accumulator), STX (store the X-register), and STY (store the Y-register). To store the register information at some memory location simply follow the store instruction with the address of the desired storage location.

Examples:

STA $1000 - Stores a copy of the contents of the accumulator at location $1000. The contents of the ACC are not disturbed .

STA $2563 - Stores the contents of the accumulator at location $2563.

STA $FF - Stores the contents of the accumulator at location $FF.

By using the STX and STY instructions, we can store data from the X- and Y-registers in a similar manner.

STX $1500 - Stores a copy the contents of the X-register at location $1500 in memory.

```
          STY $220  - Stores the contents of the
                      Y-register at location
                      $220 in memory.
```

Remember, the store instructions do not alter the contents of the register being stored; only the memory location where the data is being stored.

Now that we know a few basic commands, let's write a simple assembly language program. This program will simply transfer the data contained in locations $1000 and $1001 to locations $2000 and $2001 respectively. After this program is executed, location $2000 and $1000 will contain the same value and locations $1001 and $2001 will contain the same value. The (seemingly) easiest way to do this is to execute an instruction sequence "LET $2000 EQUAL $1000, and LET $2001 EQUAL $1001." Unfortunately, there is no memory transfer function which will perform this task for us. What we can do, however, is to perform this action in an indirect manner. Instead of a straight memory transfer, we can load the accumulator with the data contained in location $1000 and then store the accumulator at location $2000.

****

This process can then be repeated for locations $1001 and $2001. The final assembly language program is:

```
          LDA $1000
          STA $2000
          LDA $1001
          STA $2001
```

All data transfers must be routed through one of the 6502 registers, and generally we will use the accumulator since it is the general purpose register on the 6502.

DATA TRANSFER INSTRUCTIONS.

Now that we know how to exchange data between a register and memory, what about transfering data between registers? The 6502 has the capability to transfer data from the accumulator to the X- or Y-registers and likewise from the X- or Y-register to the accumulator. There are also two instructions which allow the 6502 to transfer data from the X-register to the Stack Pointer and to transfer data from the Stack Pointer to the X-register. The mnemonics for these instructions are:

```
          TXA - Transfers data from the X-register to ACC.
          TYA - Transfers data from the Y-register to ACC.
          TAX - Transfers data from the ACC to the X-register.
          TAY - Transfers data from the ACC to the Y-register.
          TXS - Transfers data from the X-register to SP.
          TSX - Transfers data from SF to the X-register.
```

You will notice that there are no explicit instructions for transfering data from the X-register directly to the Y-register and vice versa. Should this need arise two instruction sequences can be used:

```
          Transfer X to Y     Transfer Y to X
               TXA                 TYA
               TAY                 TAX

              - OR -              - OR -

               STX $nnnn           STY $nnnn
               LDY $nnnn           LDX $nnnn
```


****

The register transfer instructions require no operands. In fact, if an attempt is made to place an operand after the transfer mnemonic, an error message will be displayed. This is an example of the 'implied' addressing mode. The instruction itself implicitly defines the location of the data being operated upon (the registers).

REGISTER INCREMENTS AND DECREMENTS.

Being able to load and store data is not particularly interesting by itself, so now we are going to discuss some instructions which operate on the data in a register. The first four instructions we will study in this category are the X- and Y-register increment and decrement instructions (increment means to add one; decrement means to subtract one).

INX - Takes the value contained in the X-register, adds one, and leaves the result in the X-register.

INY - Takes the value in the Y-register, adds one, and leaves the result in the Y-register.

DEX - Takes the value contained in the X-register, subtracts one, and leaves the result in the X-register.

DEY - Takes the value in the Y-register, subtracts one, and leaves the result in the Y-register.

The above instructions are handy for simple register arithmetic. Since (as you will soon find out) most of the time we are adding one to or subtracting one from these registers, the increment and decrement instructions are very useful.

There is one slight problem with the increment and decrement register instructions. What happens when you try to increment a register which contains $FF (the maximum value possible

for an 8-bit register) or decrement $00 (the smallest value possible for an 8-bit register)? When a register containing $FF is incremented, the computer will "wrap-around" and end up with the value $00. Likewise, whenever you decrement the value $00 you will end up with $FF.

****

Like the transfer instructions, INX, INY, DEX, and DEY are implied addressing mode instructions and require no operands.

INCREMENT AND DECREMENT
INSTRUCTIONS.

The increment (INC) and decrement (DEC) instructions are special. They operate directly on memory without the need to go through the accumulator, X-, or Y-registers as an intermediate step. These two instructions increment and decrement values at specified memory locations.

EXAMPLES:

INC $2255 - Takes the value at location $2255, adds one, and then leaves the result at location $2255

DEC $15 - Takes the value contained at location $15, subtracts one, and then leaves the result in location $15.

The INC and DEC instructions are not implied addressing mode instructions. They require an absolute or zero page address, like the load and store instructions. Keep in mind, you are limited to eight bits; as such, "wrap-around" will occur if you attempt to increment $FF or decrement $00.

LABELS AND VARIABLES.

Until now, everytime we wanted to use a variable, the actual memory address of that variable had to be specified. This is inconvenient (This situation is similar to using all POKE instructions instead of variable names in BASIC). For instance, suppose we have a value giving the X-coordinate of a point we wish to plot on the screen. XCOORD would be much more meaningful than $800. It would be nice to be able to write LDA XCOORD instead of 'LDA $800.' Labels allow us to do exactly this! Somewhere in our program we define a label to be equal to some value (an address). Thereafter, whenever that label is referenced, the address is used instead. In the previous example you would equate the value $800 with the label XCOORD, then you could write LDA XCOORD, and the assembler would automatically sub-

****

stitute $800 for you! A label may be used in place of an address. Remember, the assembler simply substitutes the assigned value upon encountering a label. LDA XCOORD does not mean load the accumulator with the value XCOORD, but rather, load the accumulator with the value contained in the memory location $800. Labels allow you to assign more meaningful names to memory locations.

There is one catch however. Somewhere within the program you must equate the value with the label. How is this accomplished? Very simply. To equate a label with an address you use the EQU pseudo opcode. First, what is a pseudo-opcode? A pseudo opcode is simply an instruction to LISA embedded within your assembly language source file. When encountered, a pseudo opcode tells LISA to do something special with the following data. A pseudo opcode generally does not emit any instruction code for use by the 6502 microprocessor.

The EQU pseudo opcode has the form:

```
LABEL  EQU  <value>
```

Both the label and the value (an address expression to be described later) are required. The EQU pseudo opcode tells LISA to take the label and store it with its corresponding address value in the assembler symbol table. Later, when you use the label in your program, LISA looks up the label in the symbol table and substitutes the address for the label. The assembler remembers ugly things like addresses for you, and all you have to do is remember which variable name (or label) you used.

EXAMPLES OF LABELS:

```
XCOORD  EQU $800   - XCOORD is assigned the value $800
LABEL   EQU $1000  - LABEL is assigned the value $1000

        LDA XCOORD - Same as LDA $800
        STA LABEL  - Same as STA $1000

CONST   EQU $FF22  - CONST is assigned the value $FF22

        LDA #CONST - The value $22 is loaded into
                     the acc ($22 is the low order
                     byte of CONST).
```


****

```
        LDA /CONST - The value $FF is loaded into
                     the acc ($FF is the high order
                     byte of CONST).

        INC XCOORD - Increments the value at location
                     $800.
```

```
                DEC LABEL  - Decrements the value at location
                             $1000
```

When a label is defined using the "EQU" pseudo opcode, absolute addressing will always be used (even if the value is less than $FF). In order to use zero page addressing another pseudo opcode (equate to page zero) must be used. This pseudo opcode is EPZ and it has the same syntax as EQU.

EXAMPLE:

```
     LABEL  EPZ  <value>

          <value> must be less than or equal to $FF.
```

Sometimes, when defining a variable, even worrying about where the data should be stored in memory is too much of a bother. It would be nice if one could say, "Hey, I need a one-byte variable, but let LISA worry about its actual location in memory." The DFS (or define storage) pseudo opcode will do exactly that for you. The DFS pseudo opcode uses the syntax:

```
     LABEL  DFS  <value>
```

Unlike EQU the value does not specify where the data is to be stored, but rather how many bytes you wish to reserve for your variable. Usually this value will be one or two.

DFS simply uses the current code location as the address for the variable. Because of this you must be careful to place the DFS pseudo opcode in your program where it will not be executed as an instruction. We'll discuss how you do this later on.

EXPRESSIONS IN THE OPERAND FIELD.

Suppose in our previous example that XCOORD was a 16-bit value located in bytes $800 and $801. How can we access

****

these locations using our label scheme? LISA allows simple arithmetics to be used in the operand field. The operators "+" and "-" are allowed.

EXAMPLE:

```
     XCOORD EQU $800
            LDA #$0       -CLEAR THE ACCUMULATOR
            STA XCOORD    -CLEAR LOCAUION $800
            STA XCOORD+$1 -CLEAR LOCATION $801
```

Some versions of LISA also allow multiplication, division, and some logical operations in address expressions. For more details consult the LISA reference manual.

****

CHAPTER 5

ASSEMBLY LANGUAGE

NEW INSTRUCTIONS:

```
BRK JMP BCC BCS BEQ BNE BMI CLD
BPL BVC BVS BLT BGE BFL BTR SED
CMP CPX CPY CLC CLV SEC CLI SEI
END
```

GENERAL.

The load and store instructions discussed in the previous chapter are examples of sequentially executing instructions. After a load or store is executed, the computer proceeds to the next instruction and continues processing there. As in BASIC, we often need to interrupt this sequential program flow and continue execution elsewhere. Unlike BASIC, we do not have a GOTO,

****

FOR/NEXT, or IF/THEN instruction at our disposal. In their place the 6502 microprocessor has a group of jump and branch instructions.

Finally, we need to be able to tell the computer to stop and return control to the user. There are several methods of achieving this goal. The easiest and most straight-forward method is probably the BRK, or break instruction. When executed, the BRK instruction will beep the bell and relinquish control to the Apple monitor. A nice feature of the BRK instruction is that it prints the contents of the 6502 registers before returning to the monitor. This is a very simple form of output which we will make use of until more sophisticated I/O routines are possible.

EXAMPLE PROGRAM.

Let's try writing a program using loads and the BRK instruction. First, access LISA (see its accompanying documentation for details) and proceed as follows:

1) When the prompt (!) is displayed, type INS and depress return (CR) key.
2) Response-LISA will display a 1 on the next line.
3) Enter a space, type LDA #$0 and depress CR.
4) Response-LISA will display a 2 on the next line.
5) Enter a space, type LDX #$1 and depress CR.
6) Response-LISA will display a 3 on the next line.

7) Enter a space, type LDY #$2 and depress CR.
8) Response-LISA will display a 4 on the next line.
9) Enter a space, type BRK and depress CR.
10) Response-LISA will display a 5 on the next line.
11) Enter a space, type END and depress CR.

The execution of step 11 informs LISA that the end of the program has been reached.

12) Response-LISA will display a 6 on the next line.

Since you have completed source code entry:

13) Type a control-E as the first character of line six and depress the return key.
14) Response--The ! prompt will be displayed on the next line.

****

If all has gone well the display will appear as indicated below:

```
!INS
   1  LDA #$0
   2  LDX #$1
   3  LDY #$2
   4  BRK
   5  END
   6
```

LISA is now waiting for your next command. Before any program can be run it must be assembled. To assemble your program, simply type ASM when you get the "!" prompt back. LISA will flash an assembly listing on the screen while the program is being assembled. Ignore this for now. When you get the "!" prompt back, type BRK (this is a LISA command as well as a 6502 instruction) which will place you in the Apple monitor. To run your program type 800G when you get the monitor '*' prompt character. Immediately after pressing return, the speaker should beep and the screen should look like:

```
0808- A=00 X=01 Y=02 P=30 S=F0
```

(The value after "S=" may be different.) The 0808 is the address in memory of the BRK instruction PLUS TWO. This means that the BRK instruction is really located at memory location $806. The reason for having two added to the true value will be discussed in the section on debugging your programs.

The next five entries on the line are the values contained in the accumulator, X-register, Y-register, PSW, and stack pointer when the BRK occurred. As mentioned previously, we will use the fact that the BRK instruction prints these registers to perform simple I/O. In essence, the BRK instruction is very similar to the END and STOP instructions in BASIC.

JMP INSTRUCTION.

The 6502 JMP (jump) instruction is an unconditional branch. It is used in a manner identical to the GOTO instruction in BASIC. The difference is that you specify an absolute memory address instead of a BASIC line number. The following infinite loop continually copies location "J" into location "I" and then sets location

****

J to zero (obviously, after the first time through, location I will also contain zero).

EXAMPLE:

```
     PROGRAM LOC.   STATEMENT
          $800  I    EQU $0
          $800  J    EQU $1
          $800       LDA J
          $803       STA I
          $806       LDA #$0
          $808       STA J
          $80B       JMP $800
          $80B       END
```

(Note that the EQU and END statements do not take up a program location.)

The JMP instruction is always three bytes long: the JMP instruction code, followed by the low-order and then high-order byte of the jump to address.

Obviously, using absolute addresses, as in the previous example, presents a problem. First, at the time the text file is created, the actual destination address of the JMP instruction is not usually known. To overcome this difficulty we use labels as the destination address of the JMP instruction, much like we used labels in the load and store instructions. Unfortunately, using labels seems to be a matter of simply delaying the inevitable. After all, if a label is used it must be declared using the EQU,

****

EPZ, or DFS pseudo opcodes, right? Not always. If a label appears on the same line as an instruction, and it begins in column one of the source line, then the address of that instruction will be used as the value equated to the label. The code sequence presented previously can be replaced by:

```
     I      EQU $0
     J      EQU $1
```

```
          LABEL  LDA J
                 STA I
                 LDA #$0
                 STA J
                 JMP LABEL
                 END
```

and the assembler will worry about where LABEL is supposed to be.

You will note that this is even easier to use than the GOTO in BASIC because you don't have to worry about using sequential line numbers, especially when you are branching forward. The assembler detects a label on the current line by checking column one. If column one contains an uppercase alphabetic character, the following characters (up to a space or ":") are assumed to be part of the label. You must separate the label and the mnemonic field by at least one space. Also (as mentioned in a previous chapter), if a label does not appear on the current line, there must be a blank in column one. If you do not place a blank in column one, the assembler will treat the mnemonic as a label and attempt to use the operand (if any) as your mnemonic. The result? An illegal mnemonic error most likely, so always remember to place a space in column one if a label does not appear on the current line of text. One final remark: since LISA detects a label by checking column one of the current source line, labels such as LDA, LDX, INC, or any other 6502 mnemonic are perfectly valid. For clarity's sake however, you should avoid mnemonic names as statement labels.

PROCESSOR STATUS REGISTER (P or PSW).

The 6502 instruction set does not include an "IF/THEN" or "FOR/NEXT" instruction. Conditional testing is accomplished by testing bits in the processor status register.

The processor status register is unlike the other registers in

****

the 6502 processor. Rather than being an 8- or 16-bit register whose data is treated as eight or 16 bits of information, the processor status register is simply a collection of eight bits where each bit is treated separately (actually only seven bits are used; one of the bits is ignored).

Four of these bits are set by the result of the previous instruction. For instance, there is a zero flag in the P-register which is set if the last result was a zero and reset otherwise. Two of the remaining flags are explicitly set or cleared by 6502 instructions, and one of the flags is set if the last interrupt serviced was due to the execution of the BRK instruction.

BREAK FLAG (B).

The break flag (bit number four in the processor status register) is set only if the last interrupt detected was due to the execution of the BRK instruction. You will notice that whenever you execute a break instruction, the P-register is usually displayed as P=30 (sometimes other values will creep in). If you convert this hex number to binary, you will find that bit number four is always set, because the last instruction executed was a BRK instruction.

DECIMAL FLAG (D).

The decimal flag (bit number three in the PSW) is set only by the SED (set decimal) instruction. It can be cleared by the CLD (clear decimal flag) instruction. The decimal flag is used to determine what type of arithmetic will be used by the 6502 microprocessor. More information will be given on the decimal flag in the next chapter.

INTERRUPT DISABLE FLAG (I).

Interrupts are beyond the scope of this book. For completeness however, it should be mentioned that one of the flags in the processor status register is used to prevent interrupts from occurring. This flag (bit number two in the PSW) can be set by the SEI instruction, and it can be cleared with the CLI instruction. The 6502 IRQ line is disabled when the interrupt disable flag is set.

****

CONDITION CODE FLAGS (N,V,Z,C).

The condition code flags are the flags affected by the normal operation of the 6502 microprocessor. The Z (or zero) flag is set when the last operation executed produced a zero result. For instance, loading the accumulator with zero sets the zero flag; decrementing a register when that register previously contained one gives a zero result and, as such, sets the zero flag; incrementing $FF results in a wrap-around to zero giving a zero result. There are no explicit instructions for setting or clearing the zero flag. If you want to set it, simply load a register with zero. If you want to clear it, simply load a register with a value other than zero. Sneaky trick: If you can't afford to bother the contents of any of the 6502 registers, simply increment a memory location known to contain $FF. Location $FFC3 in the Apple monitor is such a location (both for the old monitor and the new Auto-start ROM). If you issue the instruction "INC $FFC3," the zero flag will be set. Likewise, to reset the zero flag without affecting any of the 6502 registers, simply increment a location which does not contain $FF. Location $F800 is a good choice. The Z flag resides in bit number one of the PSW.

The 6502 N flag is set if the last result was a negative value. Wait a second! All along we've been saying that there are no negative values in the 6502 registers. Well, if you remember the section on two's complement, we used the high-order bit as a sign

flag. If it was set, the number was negative. If it was reset, the number was positive. We will discuss signed arithmetic later. Here it is useful to note that the negative (N) flag will contain whatever was in bit number seven of the previous result. This is sometimes useful in itself, just to be able to check the status of one of the bits in a memory location. The N flag resides in bit number seven of the PSW.

As with the zero flag, there are no explicit set or clear instructions associated with the N flag. To set the N flag simply increment any location which contains a value in the range $7F to $FE. The result of such an increment will always be negative i.e., bit number seven of the result will always be one. Location $F804 in the Apple monitor is a good choice. To reset the negative flag simply increment a memory location which contains a value in the range $00 to $7E, or $FF. The result of such an increment is always

****

positive i.e., bit seven of the result will always be zero (memory location $F800 in the Apple monitor is a good choice).

The carry (C) flag in the 6502 microprocessor is affected by additions, subtractions, comparisons, and logical operations. It can also be explicitly set or cleared with the SEC (set carry) and CLC (clear carry) instructions. The carry flag resides in bit number zero of the processor status register. We will return to the discussion of the carry flag when the discussion of the aforementioned operations is taken up.

The last flag in the processor status register is the overflow (V) flag. This flag is used for signed arithmetic and is affected only by the addition, subtraction, and bit test operators. It can be explicitly cleared with the CLV instruction but there is no "set overflow flag" instruction. This flag resides in bit number six of the processor status word. We will discuss its use when signed arithmetic is considered.

The unused bit in the processor status word is bit number five. Usually it contains a one (i.e, it's set), but you are not guaranteed this. None of the 6502 instructions access this flag.

You might try running the following programs noticing their affects on the P-register:

```
PGM1:
        LDA #$0
        BRK
        END
PGM2:
        LDA #$1
        BRK
        END
PGM3:
        CLC
```

```
        BRK
        END
PGM4:
        SEC
        BRK
        END
PGM5:
        LDA #$80
        BRK
        END
PGM6:
        LDA #$7F
        BRK
        END
```


****

BRANCH INSTRUCTIONS (6502).

Now you know that certain operations affect the flags in the processor status register. Big deal! How does this help us simulate the "IF/THEN" Statement in BASIC? By themselves the status flags are not very useful in this capacity. Fortunately, the 6502 allows us to test each of the condition code flags with some branch instructions.

A branch instruction is very similar to a JMP instruction. Under certain circumstances it causes program flow to continue at a different location. Unlike the JMP instruction, which is an unconditional branch, the branch instructions do not always jump to the specified location. Before a branch is made, one of the flags in the processor status word is tested and, if the test is met, then (and only then) will the branch be taken. Should the test fail, the program continues executing at the next instruction, just like the IF/THEN in BASIC.

Using the branch instructions we can test any of the condition code flags to see if they are set or cleared. The allowable branches are:

- BCC - Branch if the carry flag is clear.
- BCS - Branch if the carry flag is set.
- BEQ - Branch if the zero flag is set.
- BNE - Branch if the zero flag is clear.
- BMI - Branch if minus (N=1).
- BPL - Branch if plus (N=0).
- BVS - Branch if overflow is set (V=1).
- BVC - Branch if overflow is clear.

Just as with the JMP instruction you must specify an address (or label) in the operand field.

EXAMPLE:

```
        LDA #$0
```

```
            BEQ LBL1
      LBL2  LDA #$FF
      LBL1  BEQ LBL2
```

In this example the accumulator is loaded with the value zero. This sets the zero flag which causes the following branch to be taken. At LBL1 there is another branch if equal to zero instruction. Since we have not modified any registers or memory locations, the zero flag has not had a chance to be affected so the branch will be taken. This leads us to LBL2 where we load the accumulator with the value $FF. The next instruction (at loca-

****

tion LBL1) tests the zero flag. Since the last result obtained was $FF (from the LDA instruction), this branch will not be taken and the program will fall through to the next instruction after LBL1.

If you would like to test a memory location to see if it is zero, and increment it if it is zero, you could use the following code:

```
      LDA $1F    ;GET THE VALUE CONTAINED IN LOCATION $1F
      BNE LBL    ;IF IT IS NOT ZERO BRANCH TO "LBL".
      INC $1F    ;ADD ONE TO THE VALUE AT LOCATION $1F
  LBL ---        ;NEXT INSTRUCTION
      ETC.
```

LOOPS.

One of the more powerful features of a computer is it's ability to repeat a section of code over and over for a specified number of times. This technique is called looping. In BASIC you might use the "FOR/NEXT" loop to accomplish this task. In assembly language there is no "FOR/NEXT" loop so this function has to be synthesized.

Possibly the easiest way to synthesize a loop is to load a memory location with an initial value and then decrement the memory location until it becomes zero. By using the BNE instruc-tion you can cause the body of the loop to be executed until the memory location becomes zero.

****

As an example, suppose you wanted to add 10 to the vari-able J. Since we have not yet discussed addition on the 6502 we will have to use the increment instruction. Since INC only adds one to a memory location, we will have to repeat this instruction 10 times. We could simply type ten INC J instructions in a row, but this would be somewhat inefficient. Instead, let's store 10 in some memory location (e.g. I) and then set up a loop whereby we increment J ten times. The actual program to do this could be:

```
I   EQU 0
J   EQU 1
    LDA #!10    ;INITIALIZE I TO 10
    STA I
    LDA #$0     ;INITIALIZE J TO 0
    STA J
LP  INC J       ;NOW, INCREMENT J 10 TIMES
    DEC I
    BNE LP
    LDA J       ;LOAD J SO WE CAN DISPLAY IT
    BRK         ;BREAK AND DISPLAY J (IN THE ACC)
    END
```

A "step size" of minus one is not always convenient, not to mention that we can only end our loop when I becomes zero. To learn how to alleviate this problem, read on...

COMPARISONS.

Unfortunately, in the real world we need to be able to test other things besides just our condition code flags. For instance, sometimes it would be nice if we could determine whether or not I=5,or possibly if (X=6) AND (J<=(Ix5+2)) OR (L=M). Other times we might want to have a loop with an indexing variable which is initialized to one and is incremented until it becomes some other non-zero value such as 10. In order to perform these types of operations, we will have to use the 6502 compare instructions.

The CMP (compare to accumulator) instruction compares the memory operand specified against the accumulator. How is the comparison made? The data in the operand field is subtracted from the accumulator. The PSW flags are set according to the result obtained and then the difference obtained from the subtraction is discarded. After the compare instruction, both the accumulator and the memory operand contain their original

****

values. So what good is the CMP instruction if the results are lost? Even though the result of the subtraction is not kept around, the condition code flags are set, depending upon the status of the subtraction. If the contents of the accumulator equals the contents of the specified memory location, the result of the subtraction will be zero. You can then use the BEQ branch instruction immediately after a compare to test for equality (doesn't BEQ, branch if equal, make a little more sense now?). Likewise, if the contents of the accumulator do not equal the data contained in the specified memory location, the zero flag will be reset, and you can use the BNE (branch if not equal) to test for this condition.

The N and C flags are affected in a reverse fashion. If the C flag is set or the N flag is clear, the value in the accumulator is greater or equal to the contents of the specified memory locations.

If the N flag is set or the C flag is clear, the value in the accumulator is less than the contents of the specified memory location. These tests are so useful that two instructions have been added to LISA's repertoire: BGE (for branch if greater than or equal to) and BLT (for branch if less than). These two instructions generate the same machine code as BCS or BCC respectively. Why have two mnemonics which mean the same thing? For the same rea-

****

son you program in assembly language instead of machine language: these extra mnemonics are easier to remember when testing for the greater than or equal to and the less than conditions.

The overflow flag (V) is not affected by the compare instruction, so the use of the BVC or BVS instructions after a compare is futile.

You can also compare the X- and Y-registers against some memory operand by using the CPX and CPY instructions respectively. The same condition code flags are set, and you can use the branch instructions to test for the same conditions as with the CMP instruction.

One thing you have probably noticed is the lack of BGT (branch if greater than) and BLE (branch if less than or equal) instructions. These instructions are simply not available on the 6502 microprocessor. Even though they are not available as discrete instructions, they may be synthesized by using the BEQ, BNE, BLT, and BGE instructions. Suppose you wanted to compare I with J and jump to LBL if I is less than or equal to J. This could be accomplished with the following code:

```
          LDA I
          CMP J
          BLT LBL
          BEQ LBL
```

If I is less than J, the first branch encountered will be taken; if I is equal to J, the first branch will not be taken, but the second branch will be taken. If I greater than J, then neither branch will be taken, and the program will simply fall through.

Testing for the greater than function is only slightly more difficult. To compare I with J and branch to LBL if I is greater than J, you could use the code:

```
          LDA I
          CMP J
          BEQ EQL
          BGE LBL
      EQL ---
          ETC.
```

In this example I is compared with J. If they are equal, I

cannot be greater than J so a branch around the following BGE instruction is made. If I does not equal J, then it can only be less than or greater than J. If I is greater than J, the branch to location

****

LBL will be taken; if I is less than J, the program will simply fall through to the instruction at location EQL.

More efficient methods of simulating the BGT and BLE instructions will be considered later.

IF/THEN STATEMENT SIMULATION.

The IF/THEN statement in BASIC (or Pascal for that matter) has the form:

IF <LOGICAL EXPRESSION> THEN <STATEMENT>

where <STATEMENT> gets executed if and only if the logical expression is TRUE. For instance, the BASIC statement IF X>= 7 THEN Y=0 would set Y to zero if and only if X is currently greater than or equal to seven. To simulate the IF statement in assembly language you would use the opposite type branch to jump around the statement to be executed. As an example, if you wanted to convert the previous BASIC statement to assembly language, you would use the code sequence:

```
        LDA X
        CMP #$7
        BLT LBL
        LDA #$0
        STA Y
LBL     ---
        ETC.
```

In this example, if X is greater than or equal to seven, the program simply drops through the branch instruction and sets Y to zero. If X is less than seven, the branch if less than instruction causes the code which sets Y to zero to be skipped.

Naturally, a block of instructions can be executed by placing these instructions between the branch instruction and the target label of that particular branch.

FOR/NEXT LOOP REVISITED.

As mentioned previously, it would be nice if we could end our loops at some value other than zero. Now that we have the CMP instruction under our belts we can do just that! If you wish to start your loop index variable with the value $1 and increment

****

it until 10 is reached you could use the following code:

```
          LDA #$1
          STA I
     LOOP LDA I
          CMP #!10
          BEQ LBL1
          BGE LOPX
     LBL1:

     ;NOTE: The normal code within your loop body goes here.

          INC I
          JMP LOOP
     LOPX BRK
          ETC.
```

  If you would like a step size of two, simply increment I twice before jumping to LOOP. One last improvement which can be made is in the testing process. Since we want to test I to see if it is greater than 10, we must synthesize the BGT branch using the BEQ and BGE branches. One other method of doing this is to test to see if I is greater than or equal to 11. Since we have a BGE branch, this will save us some code. The resulting program would be:

```
             LDA #$1
             STA I
     LOOP    LDA I
             CMP #$B  ;$B = 11 DECIMAL
             BGE LOOPX

     ;NORMAL LOOP BODY GOES HERE

             INC I
             JMP LOOP
     LOOPX   BRK
             ETC.
```

     This small simplification makes life much easier for us.

     - TWO FINAL WARNINGS -

     Up to this point our discussion has concerned itself with unsigned values. Signed comparisons, which will be considered

****

later, follow a completely different set of rules. BE AWARE OF THIS.

     Also, in the discussion of the branch instructions, it was implied that you could branch anywhere in memory. This is not

the case. Branches use a special addressing mode called, "relative addressing." Unlike the JMP instruction which is followed by a 16-bit absolute address, the branch instructions are followed by a one-byte displacement. This displacement is added to the address of the instruction which follows the branch instruction to give an address which is in the range -126 to +129 bytes from the beginning of the branch instruction.

What does this mean to your program? Usually nothing, since most branches fall within this range. Once in a great while a branch will be out of this range and the assembler will give you a "branch out of range" error message. Since we cannot increase the range of the branch instruction, another method must be used to correct this problem. Simply replace the branch instruction with the opposite type branch (e.g., if a BEQ is out of range, use a BNE branch) and use the strange looking address of "*+$5" for your operand. Immediately after the branch instruction, enter a JMP instruction using the address of the original branch.

First, what does "*+$5" mean? Whenever a 6502 assembler encounters the asterisk in the operand field it will substitute the address of the beginning of the current instruction for the '*'. The "*+$5" means add five to the address of the branch instruction and go there if the condition is satisfied. Since the branch instruction is two bytes long and the following JMP instruction is three bytes long the branch to "*+$5" will branch to the instruction following the JMP instruction.

EXAMPLE: BEQ LBL is out of range, fix it.

Simply substitute:

```
BNE *+$5
JMP LBL
```

If the last operation set the zero flag, the program will drop through to the JMP instruction and then jump to location LBL. If the zero flag was not set after the last operation, a branch will occur to the next instruction after the JMP instruction. This effectively simulates a "LONG BRANCH IF EQUAL" to location LBL.

****

A Table of Branches and the Long Branch Form

IF THIS BRANCH IS OUT OF RANGE	USE THIS
BEQ LBL	BNE *+$5 JMP LBL
BNE LBL	BEQ *+$5 JMP LBL

```
BCC LBL         BCS *+$5
                JMP LBL

BCS LBL         BCC *+$5
                JMP LBL

BVC LBL         BVS *+$5
                JMP LBL

BVS LBL         BVC *+$5
                JMP LBL

BMI LBL         BPL *+$5
                JMP LBL

BPL LBL         BMI *+$5
                JMP LBL

BGE LBL         BLT *+$5
                JMP LBL

BLT LBL         BGE *+$5
                JMP LBL

BTR LBL         BFL *+$5 (SEE THE NEXT SECTION)
                JMP LBL

BFL LBL         BTR *+$5
                JMP LBL
```

The asterisk can be used in other address expressions as well as the branch instructions, however its use is not really recommended.

****

TESTING BOOLEAN VALUES.

Remember the values true and false? Often within a program you will use certain variables to hold flags for use in other parts of the program. Since the use of such Boolean variables occurs often, it would be nice to define the Boolean values TRUE and FALSE. As per the discussion in Chapter 2, we will let FALSE be represented by the value $00 and TRUE by the value $01. Now we can use the BEQ and BNE instructions to test for true or false. The only problem with this scheme is that we use the branch if not equal instruction, to test for true and the branch if equal to test for false. This may seem incongruent. Rather than leaving you feeling strange about using these tests, LISA incorporates two additional branch instructions, BTR and BFL (branch if true and branch if false, respectively), which generate the same code as BEQ and BNE. The former instructions are simply easier to remember.

While on the discussion of true and false, it should be men-

tioned that you should include the statements:

```
FALSE  EQU $0
TRUE   EQU $1
```


****

at the beginning of your program. True and false will not be used as memory locations, but rather as symbolic constants. Now your programs will read:

```
LDA #FALSE
STA I
LDA #TRUE
STA FLAG
```

instead of:

```
LDA #$0
STA I
LDA #$1
STA FLAG
```

Obviously, the first version is much more readable. Incidently, the use of symbolic constants is not limited to true and false. Anytime you use some hex value which has special significance (for instance the ASCII code for carriage return), it should be declared as a symbolic constant. Symbolic constants make your programs much easier to read and modify.

****

CHAPTER 6

ARITHMETIC OPERATIONS

NEW INSTRUCTIONS:

ADC SBC

GENERAL.

The art of assembly language programming is actually the art of learning to do things a piece at a time. Arithmetics cannot be performed with a single statement as in BASIC. Rather, 10, 20, or even 50 lines of machine language code may be required to perform a specific operation.

There are three basic types of arithmetic operations performed by the 6502 microprocessor: (1) unsigned binary, (2) signed binary, and (3) unsigned decimal arithmetic. DO NOT CONFUSE

THESE! Each type of arithmetic follows its own set of rules; intermixing these operations and/or rules may cause invalid results.

UNSIGNED INTEGER (BINARY) ARITHMETIC.

When working with unsigned values, the 6502 processor can handle numbers in the range of 0 thru 255. Although the range is not very good, eight bits are suitable for many applications. As with the decrement instructions, wrap around occurs if you try to add two numbers whose sum exceeds the range of 0 thru 255, likewise, wrap around occurs if you try to subtract a large number from a smaller one.

Do not worry about the range limitation at this time. Multi-precision operations which allow numbers to greatly exceed the 0 thru 255 limitation will be discussed later.

Unlike your handy pocket calculator, the 6502 cannot perform functions such as SIN, COS, 1/X, LOG, or TAN. In fact, the

****

6502 cannot even multiply or divide two numbers. The only arithmetic operations the 6502 microprocessor can perform are addition and subtraction. All of the other fancy operations can be simulated by using addition and subtraction.

The 6502 instruction mnemonic for addition is ADC (add with carry). This instruction takes a memory operand and adds it to the accumulator. Once this is accomplished, the value contained in the carry flag (zero or one) is also added to the accumulator. The reason behind this will become clear when we discuss multi-precision arithmetic. In any case, your first unsigned arithmetic rule is: ALWAYS CLEAR THE CARRY FLAG BEFORE PERFORMING AN ADC. Obviously, if you do not explicitly clear the carry flag before performing an addition, you stand a 50/50 chance of ending up with the intended sum PLUS ONE.

EXAMPLES:

```
CLC     ;ALWAYS!
LDA #$5
ADC #$3
BRK     ;PRINTS RESULT OF ADDITION
END


CLC
LDA #7
ADC #$3
BRK
END


CLC
```

```
          LDA #$FC
          ADC #$20
          BRK
          END
```


****

In the last example overflow will occur and you will end up with the result $1C.

What happens if an overflow occurs? Unlike BASIC, a machine language program will not abort with a "** >255" error. In some repects this is friendly; no more nasty error messages. Unfortunately, instead of being nice and informing you of a problem, the 6502 will go on about its business as though nothing had happened. This can lead to very unpredictable results! Luckily, the 6502 does provide us with a flexible error checking facility. If an overflow occurs during an addition instruction, the carry flag will be set. By using the BCC and BCS instructions you can test for overflow immediately after an addition.

EXAMPLE:

```
          CLC
          LDA I
          ADC J
          BCS ERROR ;GO TO ERROR IF OVERFLOW
          ETC...    ;OTHERWISE CONTINUE PROCESSING.
```

The use of the carry flag to inform us of an overflow is very useful. Now, if we want to, we can elect to ignore an overflow condition. Or, if we're absolutely positive that an overflow will not occur (e.g, I and J are always in the range $0-$F) we don't have to waste time or memory checking for the overflow.

When an overflow does occur, you will be guaranteed one thing: the true sum will fall somewhere in the range of $100 to $1FE. This is verified quite easily by adding the two largest values representable in eight bits (namely $FF + $FF) together and examining the results. $FF plus $FF is $1FE. Any other addition using any other values will always produce a result less than $1FE. If you don't belive me try it out for yourself. When an overflow does occur, the value will be in the range $100 to $1FE and the low-order eight bits of this value (i.e. $00-$FE) will be left in the accumulator.

EXAMPLES OF OVERFLOW:

```
          CLC           CLC
          LDA #$FF      LDA #$F0
          ADC #$1       ADC #$20
          BRK           BRK
          END           END
```

****

```
          CLC           CLC
          LDA #$80      LDA #$80
          ADC #$80      ADC #$FF
          BRK           BRK
          END           END
```

RULES FOR UNSIGNED ADDITION.

1) Do not confuse these rules with the rules which follow for subtraction, signed and decimal arithmetic.

2) Always clear the carry before performing an addition.

3) Test for overflow with the BCS instruction. The carry flag will be set if overflow occurs.

SUBTRACTION.

Subtraction is performed in a similar manner to addition with three differences: (1) the SBC (subtract with carry) instruction is used; (2) THE CARRY FLAG MUST BE SET BEFORE PERFORMING A SUBTRACTION; and (3) if the carry flag is clear after a subtraction, an underflow has occurred.

In practice, points (2) and (3) are totally opposite that of the ADC instruction. Be aware of this! Many beginners consistently forget that the carry must be SET before performing a subtraction, and end up with invalid results.

EXAMPLE: SUBTRACT I FROM J AND STORE THE RESULT IN L

```
               SEC     ;ALWAYS BEFORE A SUBTRACTION!
               LDA J
               SBC I
               STA L
               BCC ERROR
               LDX #50
               BRK
          ERROR LDX #$FF
               BRK
               END
```

In this example, the X-register will be displayed as $00 if things proceeded smoothly. If an underflow occurred the X-register will be displayed with the value $FF. You can experiment with this code sequence by initializing I and J with some LDAs and STAs prior to the execution of the subtraction. Naturally you must define

****

the locations where I and J are supposed to be with the EQU or EPZ pseudo opcodes.

The SBC instruction affects the processor status register in a manner identical to the CMP instruction. Because of this you can use the branch instructions after a subtraction in a manner identical to that of the CMP instruction. Although this is of little value here (after all, the CMP instruction is easier to use); the generalization to multi-precision operations becomes very important later on.

The N and V flags have no meaning in an unsigned arithmetic operation.

RULES FOR UNSIGNED SUBTRACTION.

1) Don't confuse these rules with those for addition, signed, or decimal arithmetic.

2) Always set the carry before performing a subtraction.

3) After the subtraction operation, the carry will be clear if an underflow occured. The carry will be set otherwise.

SIGNED ARITHMETIC.

What happens when you subtract $10 (16) from $8? You would normally expect to get -$8. The computer, however, will give you an underflow (i.e., the carry will be cleared) since negative numbers are not allowed in the unsigned number system. Negative numbers, despite the fact that they are not defined in our number system, are useful on several occasions. Because of this, a method for defining signed binary numbers had to be developed.

If you remember the section on two's complement in Chapter 2, you're probably thinking, "Why not use the high-order bit as a sign bit?" (If you don't remember this, review Chapter 2). The 6502 processor has implemented the two's complement number system for dealing with signed numbers. In this numbering system the 6502 can represent values in the range of -128 to +127 (using eight bits). Signed arithmetic is performed in a manner identical to unsigned arithmetic. You use the ADC and SBC instructions, and you must clear the carry flag before an addition and set the carry flag before a subtraction.

The only difference between a signed arithmetic operation

****

and an unsigned arithmetic operation is that the carry flag is no longer significant. The carry flag is used to flag a carry out of bit 7. Since bit 7 is our sign bit, overflow (when using signed arithmetic) occurs when there is a carry out of bit 6. Since a carry out of bit 6 does not affect the carry flag you cannot test the carry flag

to check for overflow or underflow. Instead, you use the overflow (V) flag in the 6502 microprocessor. This flag is set whenever there is a carry out of bit 6 into bit 7.

When an overflow/underflow occurs, the overflow flag is set; if the allowable range is not exceeded, the overflow flag will remain clear. Unlike the unsigned tests which are opposite for addition and subtraction, the BVS test is used for both overflow (in the case of addition) and underflow (in the case of subtraction). If the overflow flag is clear (testable by using BVC), the previous operation was performed correctly.

EXAMPLE PROGRAMS:

OVERFLOW OCCURS

```
    LDA #$7F ;127 DECIMAL
    ADC #$1  ;1   DECIMAL
    BRK      ;RESULT = -128
    END

    CLC
    LDA #$80 ;-128 DECIMAL
    ADC #$80 ;-128 DECIMAL
    BRK      ;RESULT = 0
    END

    SEC
    LDA #$80 ;-128 DECIMAL
    SBC #$1  ;1    DECIMAL
    BRK      ;RESULT = +127
    END
```

OVERFLOW DOES NOT OCCUR

```
    LDA #$1
    ADC #$2
    BRK      ;RESULT = 3
    END

    CLC
    LDA #$FF ;-1 DECIMAL
    ADC #$2  ; 2 DECIMAL
    BRK      ;RESULT = 1
    END

    SEC
    LDA #$FF ;-1 DECIMAL
    SBC #$1  ; 1 DECIMAL
    BRK      ;RESULT=-2 ($FE)
    END
```

TESTING FOR UNDERFLOW/OVERFLOW:

```
    CLC                            SEC
    LDA #$FF                       LDA #$23
    ADC #$25                       SBC #$43
    BVS ERROR <- GO IF OVERFLOW ->  BVS ERROR
    BRK       <- STOP OTHERWISE ->  BRK
    END                            END
```


****

SIGNED ARITHMETIC RULES.

1) Don't confuse these rules with the rules for unsigned or decimal arithmetic operations.

2) Always clear the carry bit before an addition operation and set the carry bit before a subtraction operation.

3) Test for overflow/underflow using the BVS/BVC instructions (overflow/underflow occurred if V=1).

SIGNED COMPARISONS.

Signed comparisons are made by testing the overflow (V) flag, the sign (N) flag, and the zero (Z) flag. As usual, if the two operands are equal when they are compared the zero flag will be set. This allows you to use the BEQ/BNE instructions to test for equality. Inequalities are a little more difficult. The signed value in the accumulator will be greater than or equal to the value in the memory operand if and only if the overflow flag equals the negative (sign) flag. Likewise, the contents of the accumulator are less than the memory operand if and only if the overflow flag (after the comparison, of course) does not equal the negative (sign) flag.

There are only two problems which surface. First, there is no explicit instruction (such as, the BGE or BLT for unsigned comparisons) which tests the sign and overflow flags. Secondly, the 6502 CMP instruction does not modify the overflow flag.

The second of these two problems is the easiest to handle. Although the CMP instruction does not modify the overflow flag, the SBC instruction does; the SBC instruction affects the flags (with the noted exception of the overflow flag) in a manner identical to that of the CMP instruction. Therefore, a signed compare instruction can be simumated by setting the carry (always before a subtraction) and then using the SBC instruction in place of the CMP instruction.

The former problem is a little bit more sticky to handle. The following code will simulate a signed BGE and a signed BLT instruction:

```
            SEC
            LDA A
```



```
**************************************************************

            SBC B
            BMI LBL
            BVC GE
        LT:         <- BRANCH TO HERE IF A < B
             .
             .
             .
             .
        LBL BVC LT
        GE:         <- BRANCH TO HERE IF A >= B
```

BINARY CODED DECIMAL ARITHMETIC.

In Chapter 2 both unsigned and signed arithmetic were discussed. In this section, a third numbering system will be discussed. Binary Coded Decimal, or BCD, is a numbering system that is convenient mostly for input/output purposes, instrumentation purposes, and a few other special cases. BCD is simply a

convenient method of representing decimal digits in a binary format, and is represented in the following form:

DECIMAL DIGIT	BINARY REP.	
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001

So far, BCD and binary representations look exactly alike, but watch what happens when numbers beyond 9 are used.

DECIMAL DIGIT	BINARY REP.	
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101
16	0001 0000	0001 0110

In BCD the low-order nibble is used to represent the low order decimal digit and the high-order nibble is used to hold the

****

high-order decimal digit. The bit patterns 1010 thru 1111 are not allowed in either nibble of a BCD number. With eight bits you can represent numbers in the range of 0 thru 99. BCD digits are specified in your assembly language programs in a manner identical to hex constants. Always preceed your BCD constants with a dollar sign.

UNSIGNED BCD ARITHMETIC.

As with signed and unsigned binary arithmetic, all additions and subtractions are performed using the ADC and SBC instructions. Likewise you must clear the carry before an addition and set the carry before a subtraction. Upon completion of the decimal addition, the carry flag is set if an overflow occurred. For the same reason, the carry flag will be clear after a decimal subtraction if underflow occurred.

Since an unsigned decimal arithmetic operation looks exactly like an unsigned binary arithmetic operation, there has to be some way of determining whether the processor is to perform a decimal or binary operation. This is accomplished through the

use of a programmable flag (the D, or decimal flag) in the 6502 processor status register. If the decimal flag is set, all arithmetic operations will be carried out in the decimal mode. If the decimal flag is reset, all arithmetic operations will be carried out in the binary mode. As mentioned in the last chapter, the decimal flag is set with the SED instruction and is cleared with the CLD instruction. Other types of microprocessors have special decimal-adjust instructions which must be executed after every BCD addition or subtraction.

The decimal flag is very flexible. You can set it once and not worry about decimal arithmetic until you manually reset the decimal flag using CLD. This flexibility has one disadvantage. If you set the decimal flag and forget to reset it, any further binary arithmetic operations you attempt will become invalid. Although the decimal flag can be used with great flexibility, care should be exercised when using it. For this very reason, the first instruction in your program should be a CLD instruction, unless, of course, you plan to perform decimal arithmetic right away (Use SED if so). This will initialize the decimal flag to a known state (and only

****

God knows what it is before your program is run), thus preventing a 'surprise' when your program does not work properly.

There are two other considerations one must make when using decimal arithmetic on the 6502 microprocessor. First, the result of a decimal operation is invalid if any of the nibbles in either operand contain a value in the range of 1010 thru 1111. Second, due to a bug in the 6502 itself, you must explicitly compare the accumulator with zero to check for a zero result after an addition. The SBC instruction (alias CMP) works as expected.

DECIMAL ARITHMETIC EXAMPLES:

```
SED       ;SET DECIMAL MODE
CLC       ;ALWAYS BEFORE AN ADDITION
LDA #$25  ;INITIALIZE ACC TO 25 (DECIMAL/BCD)
ADC #$10  ;ADD 10 (DECIMAL/BCD)
BRK       ;RESULT IS 35
END

SED
SEC       ;ALWAYS BEFORE A SUBTRACTION
LDA #$52  ;INIT TO DECIMAL 52
SBC #$22  ;SUBTRACT 22 (DECIMAL/BCD)
BRK       ;RESULT IS 30
END

SED
CLC       ;ALWAYS BEFORE AN ADDITION
LDA #$99  ;LOAD WITH 99 (DECIMAL/BCD)
ADC #$1   ;ADD 1 (DECIMAL/BCD)
BRK       ;RESULT IS 0, CARRY = 1
```

```
          END

          SED
          SEC
          LDA #$00
          SBC #$1
          BRK       ;RESULT IS 99, CARRY = 0;
          END
```

UNSIGNED ARITHMETIC RULES.

1) Use the SED instruction to set the decimal mode.

2) Clear carry before an addition; set carry before a subtraction.

3) Make sure operands contain valid BCD digits, or an invalid result will be obtained.

****

4) If the operation is addition and the Z flag is to be tested after the addition, you must explicitly test for zero with 'CMP #$00.'

5) Overflow, after a decimal addition, is indicated by the presence of the carry flag. In this case a value greater than 99 occurred.

6) Underflow, after a decimal subtraction, is indicated by the absence of the carry flag (i.e., C = 0). In this case you are trying to represent a number less than 00.

SIGNED BCD ARITHMETIC.

The 6502 does not support signed decimal arithmetic. If you need signed arithmetic, stick to binary numbers.

ARITHMETIC REVIEW.

In this chapter we have discussed three types of number systems: unsigned binary, signed binary, and unsigned decimal (BCD). Why should we bother with three different number systems when, by initial observation, it looks like signed binary integers will meet most of our needs?

BCD is very useful when you are performing I/O operations and very few computation operations. Several instruments, such as voltmeters, frequency counters, and clocks output BCD data. If you are going to interface the APPLE II computer with such a device, you will probably have to use BCD.

Unsigned binary arithmetic is used when processing positive-only numbers. This is approximately 95% of the time (at least in most assembly language applications). If you are not going to

use negative numbers, why not double your range and use unsigned integers only? Signed arithmetic should be used ONLY where negative numbers are actually expected. Unsigned binary arithmetic is faster, easier to perform, and generally more useful than signed or BCD arithmetic.

RULES FOR 8-BIT ARITHMETIC.

1) Maximum value

a) Signed = 127
b) Unsigned = 255
c) Decimal = 99

****

2) Minimum value:

a) Signed = -128
b) Unsigned = 0
c) Decimal = 0

3) User must supply routine to handle overflow and underflow by testing C or V bits, if desired.

4) Always clear the carry before an addition and set the carry before a subtraction

5) Remember, all arithmetic goes through the accumulator.

8-bit arithmetic has several serious restrictions, the most prominent being the range limitation. Handling larger numbers will be considered in the chapter on multiple-precision arithmetic.

****

CHAPTER 7

SUBROUTINES AND STACK PROCESSING

NEW INSTRUCTIONS:

PHA	PLA	JSR
PHP	PLP	RTS

GENERAL.

As in BASIC, assembly language programmers often need to be able to branch to a section of code, execute it, and then return back to the next available instruction.

This mechanism is the subroutine. In BASIC you would use the GOSUB statement (go to subroutine) to branch to a subroutine. When the desired task had been accomplished, you would use the RETURN statement to return from the subroutine.

Assembly language subroutines are handled in an identical manner, except you use the JSR instruction (Jump to Subroutine) to call (or "invoke") a subroutine, and you use the RTS (Return from Subroutine) instruction to return from the subroutine. The JSR instruction is syntactically identical to the JMP instruction: a 1-byte instruction code followed by a 2-byte absolute address. The RTS instruction is a 1-byte (implied addressing mode) instruction.

The need for subroutines in assembly language is much greater than the need for subroutines in BASIC. Subroutines in BASIC are often used for initialization purposes, or perhaps to prevent code repetition. Subroutines are required in assembly language for these same reasons of course, but an even more important use of the subroutine is that it can be used to break up a complex task into small (and easier to handle) sub-tasks. As

****

mentioned in the chapter on arithmetic, the 6502 can only add or subtract. What happens when you wish to perform a multiplication or division? A subroutine would be used. In BASIC you certainly wouldn't use a subroutine to perform a multiplication because multiplication is built into that language. However, in assembly language it is necessary to use a subroutine since there is no multiply instruction. As you can see, places you would not have dreamed of using a subroutine before will require a subroutine in assembly language. I/O is another area where subroutines are required. The 6502 itself does not support a "PRINT" or 'INPUT' statement, these have to be synthesized using subroutines.

The most important use of the subroutine may be that it allows you to break a task into smaller modules, each of which are easy to code compared to coding the whole problem all at once. This type of approach is called the "TOP-DOWN" program development method. If you have read any of the computer magazines available, you are probably sick and tired of the phrases "structured programming" and "top-down program design." Admittedly, everyone and his brother who wanted to see their name in print has written an article for BYTE magazine about the joys of structured programming. Some articles have been good, some have been poor, and, in fact, some have been downright misleading. There have even been some articles about "structured programming in assembly language." Structured program-

****

ming and top-down program design are often confused in these articles. Structured programming entails the use of "program structures" such as the FOR loop, the REPEAT/UNTIL loop, the WHILE loop, the IF/THEN/ELSE statement, and BEGIN/END (or equivalent) block structures. Obviously, structured programming is not possible in assembly language because the aforementioned program statements are not available in assembly language. Sure, you can simulate those instructions using JMPs and CMPs, but then you can simulate these statements in BASIC or FORTRAN by using GOTO's and IF statements. The idea of structured programming disallows the use of the GOTO (or JMP) instruction, however, so by definition, structured programming is not possible in assembly language. Nevertheless, it is a good idea to simulate these programming constructs (the IF/THEN/ELSE, REPEAT/UNTIL, etc.). This topic will be discussed later in the chapter.

Top-down program design is another fancy buzz word making the rounds these days. The concept behind top-down program design is as follows: First, define the problem in very gross terms. Do not fill in any of the details. Next, break each of these gross terms down (one at a time, of course) into little pieces, each a little more detailed and refined than the previous generalization. Now take each of these little pieces and continue breaking them down until the assembly language code to implement that particular detail is obvious.

Each step in the definition of the problem should correspond to an assembly language subroutine. The main program should simply be a few initialization statements, a few tests, and then a number of JSR instructions to the various detail-handling subroutines. Likewise, each of these subroutines should simply be a collection of initializations, tests, maybe a little data manipulation, and a lot of JSR instructions.

Eventually the task will be broken down to a point where, at the lowest level, the subroutine simply consists of some assembly language statements without any JSR instructions. Naturally, the depth of subroutine nesting is dependent upon the application. Simple programs may have only one or two levels of subroutines, while complex programs may have 10 or 20 levels of subroutines. The actual level to which you should nest your subroutines

****

depends on several factors, the most important of which is how detailed you as the programmer wish to get.

Programs written in this manner are immeasurably easier to debug, and, as you can probably tell, assembly language programs are very hard to debug. You may save two days by not using a top-down approach when writing your code, but be prepared to spend a week, instead of another two days, debugging your code.

VARIABLE PROBLEMS.

Subroutines in assembly language suffer from many of the same problems as subroutines in BASIC. For example, consider the following BASIC program:

```
10 FOR I=1 TO 10
20 GOSUB 50
30 NEXT I
40 END
50 I=1
60 RETURN
```

In this example, the FOR/NEXT loop is slated to execute 10 times. Unfortunately the subroutine at line number 50 resets I to 1 each time it is called. This means that an infinite loop is formed since I will never be allowed to advance beyond two.

The same thing can happen in assembly language programs. "So what?" you're probably asking. Just make sure that you don't use Ioop index variables in your subroutines (i.e., use a different name). In assembly language programs you can use different names for variables, just as in BASIC, BUT WHAT ABOUT THE REGISTERS? Consider the following code:

```
      LDX #$F
LBL   JSR SETX
      DEX
      BNE LBL
      BRK
;
;
SETX  LDX #$10
      RTS
```

In this example, the X-index register is used as the indexing variable for the loop. The subroutine SETX loads the X-register with the value $10 (16) and returns. Upon returning, the X-register

****

will be decremented (and will become $F or 15), and will contain a non-zero result. Because of this, the loop will be repeated, once again loading the X-register with $10 and decrementing, etc. A good example of an infinite loop.

Obviously, you cannot "rename" the X-register. Yet not allowing a program to use the X-register (or accumulator or Y-register for that matter) is asking too much. Rather than disallow the use of the 6502 registers in a subroutine, you can save the affected 6502 registers upon entry into the subroutine (and before they are used) and then restore the registers with their original values prior to returning from the subroutine. The previous example could be safely coded as:

```
XSAVE   EPZ $0      ;SAVE LOCATION FOR THE X REGISTER
        LDX #$F
LBL     JSR SETX
        DEX
        BNE LBL
        BRK
;
;
SETX    STX XSAVE
        LDX #$10
        LDX XSAVE
        RTS
        END
```

Although this example does not accomplish much, at least the main program does what is expected of it, namely calling SETX 15 times and then stopping.

This does not completely solve all of the problems with sub-routines and register usage. Consider the following code:

```
XSAVE   EPZ $0
        LDX #$F
LBL     JSR SETX
        DEX
        BNE LBL
        BRK
;
;
SETX    STX XSAVE
        LDX #$10
        JSR SETX2
        LDX XSAVE
        RTS
;
;
```


****

```
SETX2   STX XSAVE
        INX
        LDX XSAVE
        RTS
        END
```

This program starts out by setting up a loop, as before. Within the loop the subroutine SETX is called. To prevent an infinite loop, the value contained in the X-register is stored at location XSAVE. Afterwards the X-register is loaded with the value $10, and then a call to SETX2 is made. As per the preceeding discussion, the X-register is saved because SETX2 modifies its contents. One problem develops here though. The current value of the X-register ($10 obtained from loading the X-register with $10 in subroutine SETX) wipes out the previous value of XSAVE used to hold the value of the X-register in the main program. So when you return

to SETX, things are okay; the X-register is loaded with $10, just as before, the call to SETX2 was made. Now, however, when the program attempts to restore the X-register to its original value (in the main program,) it will load the X-register with $10 instead of the actual original contents of $F. Once again the program is in an infinite loop.

The solution to this problem? Simply use a new (and unique) variable name (with a corresponding unique address) for the reg-

****

ister save locations in each subroutine.

EXAMPLE:

```
XSAVE1  EPZ $0
XSAVE2  EPZ $1
;
        LDX #$F
LBL     JSR SETX
        DEX
BNE     LBL
BRK
;
;
SETX    STX XSAVE1
        LDX #$10
        JSR SETX2
        LDX XSAVE1
        RTS
;
;
SETX2   STX XSAVE2
        INX
        LDX XSAVE2
        RTS
        END
```

This program will work as intended without getting itself into an infinite loop. When using different variable names for each sub-routine, it is probably better to use the DFS (define storage) pseudo opcode and reserve one byte (for each register) imme-

****

diately before the subroutine.

EXAMPLE:

```
        LDX #$F
LBL     JSR SETX
```

```
        DEX
        BNE LBL
        BRK
;
;
XSAVE1  DFS 1    ;RESERVE ONE BYTE FOR THE X REGISTER.
;
SETX    STX XSAVE1
        LDX #$10
        JSR SETX2
        LDX XSAVE1
        RTS
;
;
XSAVE2  DFS 1
;
SETX2   STX XSAVE2
        INX
        LDX XSAVE2
        RTS
        END
```

Variables that are referenced only by one subroutine are said to be LOCAL to that routine. Local variables should always be defined immediately before the subroutine in which they are used. This will help avoid confusion when reading the program later on. By contrast, variables that are used by several subroutines (and possibly the main program) are said to be GLOBAL variables. For most applications of assembly language on the APPLE II computer using local variables to save the registers is fine.

There are two types of subroutines which cannot use local variables. The so-called, "REENTRANT" subroutine (which can be an interrupt driven subroutine or a recursive subroutine) and the "ROMABLE" subroutine. It is possible for a reentrant subroutine to call itself (hence the name reentrant). If local storage is used for these types of subroutines, the registers will surely be "clobbered."

Romable subroutines, on the other hand, represent a different problem. Since the program is to be stored in ROM you cannot use the DFS statement to reserve memory because the location reserved for storage would be in ROM! The EQU psuedo

****

opcode could be used to define a location which is in RAM, but this may be inconvenient at times.

It would be very nice if there were some magic memory location where you could store a value, and then be able to store another value on top of it, but rather than destroying the original contents of that memory location, the original contents would magically be saved somewhere else for us. Then, whenever one

of the registers is loaded from the magic location, it would give us the last value that was stored there. Immediately after that value is loaded into the register, the previous contents would be loaded back into our magic memory location. With this magic memory location we could have the code:

```
                LDX #$F
        LBL     JSR SETX
                DEX
                BNE LBL
                BRK
        ;
        ;
        SETX    STX "MAGIC"
                LDX #$10
                JSR SETX2
                LDX "MAGIC"
                RTS
        ;
        ;
        SETX2   STX "MAGIC"
                INX
                LDX "MAGIC"
                RTS
                END
```

In this example we load the X-register with the value $F and then call SETX. At SETX we save the X-register into our magic memory location. We then load the X-register with $10. Next the program calls SETX2. Upon entry into SETX2 the X-register is once again saved into our magic memory location. This causes the previous contents to be saved somewhere else (and it's all automatic). Next, the X-register is incremented by one, giving us $11. The next instruction loads the X-register from our magic memory location thus restoring $10 in the X-register. Also, this causes the original value stored in the magic memory location ($F) to be reloaded into the magic memory location. SETX2 then returns to its calling procedure, namely SETX. SETX then loads the X-register from the magic memory location (which now contains $F)

****

and then returns to the calling procedure with the contents of the X-register being the same as when SETX the call invoked.

Our magic memory location is an example of a LIFO (Last In, First Out) data structure, which is usually called a stack. The classical analogy is that of a dish well in a restaurant. As the bus boy brings more dishes out, the first dishes placed in the dish well are 'pushed' down into the well. Then, as the waitress picks dishes out of the dishwell, the last ones placed in the dish well are the first ones she can get her hands on. Eventually (assuming the bus boy is slow), the waitress will take the last plate out of the well, which was the first plate stored there.

The 6502 microprocessor supports a LIFO stack. You can push data in the accumulator onto the stack with the PHA (push accumulator) instruction. Likewise data can be "pulled" from the 6502 stack and placed in the accumulator with the PLA (pull accumulator) instruction. The PHA instruction becomes our method of storing the accumulator at the "magic" memory location, and likewise the PLA instruction becomes our method of loading the accumulator from the "magic" memory location.

If you push the contents of the accumulator onto the stack and never pull it off, then that result is simply left of the top of the stack. What happens if you pull a value off the stack without first pushing data onto the stack? To answer this question, the actions of push and pull must be further explained. The 6502 stack revolves around an 8-bit register within the CPU called the stack pointer. $100 is added to the contents of the stack pointer to get a value in the range of $100 thru $1FF. Whenever data is pushed onto the stack, the data is stored in the memory location pointed to by the stack pointer in page one of memory. Immediately after the data is pushed onto the stack, the stack pointer is decremented by one. The next time data is pushed onto the stack, it will be stored on the memory location immediately below the previous entry. The stack pointer always points to the next available memory location. When data is pulled off of the stack, the stack pointer is first incremented by one, and then the accumulator is loaded from the memory location pointed at by the stack pointer. So each time you use the PHA instruction, you will be guaranteed that the accumulator will be stored in a new and unique location...with one exception. Since the stack pointer is only eight bits wide you can push a maximum of 256 bytes onto the stack before

****

the wrap-around function causes the first byte you pushed onto the stack to be overwritten. In general, 256 is plenty. A typical program usually requires, at most, 64 bytes for temporary storage.

There are no explicit instructions for pushing and pulling the X- or Y- registers. To push these registers onto the stack you should transfer the desired register to the accumulator (with the TXA or TYA instruction) and then push the accumulator.

EXAMPLES:

```
PUSH THE Y REG  PUSH THE X REG
--------------  --------------
     TYA             TXA
     PHA             PHA
```

Be aware that this will destroy the contents of the 6502 accumulator.

Now we can use the PHA and PLA instructions to save the registers for us in a subroutine. This is accomplished as follows:

```
        LDX #$F     ;INIT INDEX COUNT
LBL     JSR SETX
        DEX         ;DECREMENT COUNT
        BNE LBL     ;LOOP IF NOT THROUGH
        BRK         ;STOP
;
;
SETX    PHA         ;SAVE THE ACCUMULATOR
        TXA         ;SAVE THE X REGISTER
        PHA
        TYA         ;SAVE THE Y REGISTER
        PHA
;
        LDX #$10
        JSR SETX2
;
        TXA         ;NOW RESTORE THE Y REGISTER
        TAY
        PLA         ;RESTORE THE X REGISTER
        TAX
        PLA         ;RESTORE THE ACCUMULATOR
;
;
SETX2   PHA         ;SAVE ACC
        TXA         ;SAVE X REG
        PHA
        TYA         ;SAVE Y REG
        PHA
;
        INX
;
```


****

```
        PCA
        TAY         ;RESTORE THE Y REG
        PLA
        TAX         ;RESTORE X REG
        PLA         ;RESTORE ACC
        RTS
```

You will notice that the registers were pulled off of the stack in the reverse order that they were pushed onto the stack. Remember, the stack is a LIFO (last in, first out) data structure.

Have you ever wondered how the 6502 remembered what address to return to after a subroutine execution? The return address is pushed onto the stack when the JSR instruction is executed, and is popped off of the stack when a RTS instruction is executed. This feature allows nested, and reentrant subroutines. There is one problem however. If you push data onto the stack, and forget to pull it off before executing a RTS instruction, the data pushed onto the stack will be used as part of the return address. This brings up one very simple, yet often violated rule:

always remove data pushed onto the stack before executing a RTS instruction. Likewise, don't pull too much data off the stack or you will lose part of the return address (and whatever garbage is located just above the true return address will be considered part of it).

This helps enforce one very strong point of top-down program design: subroutines should have one entry point and one exit point ONLY! If you place multiple return points within a subroutine, chances are you will forget to pull all the data off the stack in at least one of these locations. The solution is simple. Rather than placing several RTS instructions within a subroutine, simply JMP to the single return from subroutine sequence (i.e., PLA and RTS instructions) within the subroutine.

EXAMPLE:

```
SUBRT   PHA
;
        LDA LOC1
        CMP #$50
        BLT SUB1
;
        LDA #$0
        STA LOC1
        JMP SUBX
;
SUB1    INC LOC1
SUBX    PLA
        RTS
```


****

Since the X- and Y-registers were not used within this subroutine, there was no need to save them onto the stack.

Since subroutines are useful in instances where code is replicated, why not write a subroutine which pushes the registers onto the stack, and its corresponding inverse function, which the data off of the stack? These routines could be called SAVE and RESTR and are often coded by inexperienced programmers as:

```
SUBRT   JSR SAVE


  -SUBROUTINE-
  -CODE  GOES-
  -   HERE   -


        JSR RESTOR
        RTS
;
;
```

```
SAVE    PHA
        TYA
        PHA
        TXA
        PHA
        RTS
;
;
RESTR   PLA
        TAX
        PLA
        TAY
        PLA
        RTS
        END
```

Avoid the temptation to do this! When you push the accumulator, X-register, and Y-register onto the stack, then execute a RTS instruction, the 6502 attempts to use the last two bytes pushed onto the stack as a return address. The previous contents of the X- and Y-registers will probably not make a very good return address.

PASSING PARAMETERS.

A parameter is simply a variable used to pass data to a subroutine. For example, in SIN(X), X is a parameter of the function SIN. This function, when called, returns the value of the tri-

****

gonometric sine of X (in Applesoft, not assembly language). POKE is also a procedure that has parameters. POKE, in fact, has two parameters: a memory address where the data, specified in the second parameter, is to be stored. Some procedures only require that data be passed to them. POKE is a good example of such a procedure. Other procedures and functions return data as well. SIN(X) and PEEK are two good examples of functions that return data.

There are several useful methods for passing data to a 6502 subroutine. Possibly the easiest method is to pass the data in the 6502 registers. Although this method is simple to use (and in fact it is used all the time), it has one major drawback. You're limited to only three bytes for your parameters. For some applications (such as printing a single character to the video screen, or reading a key from the keyboard) this is sufficient. As an example, the Apple monitor ROM contains two routines, one which prints the character in the accumulator onto the screen as an ASCII character, and another routine which reads the keyboard and returns the ASCII code of the key pressed in the accumulator. These routines are located at addresses $FDED and $FD0C respectively. You can turn your APPLE II computer into an "electronic typewriter" by running the following program:

```
          COUT    EQU $FDED   ;USE A SYMBOLIC LABEL FOR
                              ;CHARACTER OUTPUT


          RDKEY   EQU $FD0C   ;SAME FOR KEYIN ROUTINE


          LOOP    JSR RDKEY
                  JSR COUT
                  JMP LOOP
                  END
```

Incidently, to stop this program, hit the reset key.

When you need to pass more than three bytes to a subroutine, a different method must be used to pass the parameters. One method is to store the parameters in some known locations and then access these known locations from within the subroutine. Likewise, after returning from a subroutine the calling procedure can look at some known location to retrieve returned data. As an example, consider the following program (SUM) which

****

sums four bytes together and returns the sum of these four bytes:

```
                  LDA I
                  STA PARM1
                  LDA J
                  STA PARM2
                  LDA K
                  STA PARM3
                  LDA L
                  STA PARM4
                  JSR SUM
                  LDA RESULT
                  BRK
          PARM1   DFS 1
          PARM2   DFS 1
          PARM3   DFS 1
          PARM4   DFS 1
          RESULT  DFS 1
          SUM     CLC
                  LDA PARM1
                  ADC PARM2
                  CLC
                  ADC PARM3
                  CLC
                  ADC PARM4
                  STA RESULT
                  RTS
                  END
```

Suitable checks could be made, if desired, for overflow after any of the additions. As with the register storage scheme, parameters

should be local variables, not accessed by any other subroutines (other than for data transfer between the two).

****

CHAPTER 8

ARRAYS, ZERO PAGE, INDEXED, AND INDIRECT ADDRESSING

```
     HEX ORG OBJ DFS ASC
```

GENERAL.

So far, the only addressing modes we have used are the absolute (16-bit address), immediate (8-bit data), and relative (8-bit displacement) addressing modes. Although these are the most commonly used, they are not the only addressing techniques available.

ZERO PAGE ADDRESSING.

THE 64K address space of the 6502 is broken up into 256 blocks of 256 bytes. These blocks are called, "pages." These pages are numbered sequentially starting with 0 and ending with $FF. Page one, of course, is reserved for the 6502 stack. Page zero (the first 256 locations in the machine) is usually used for variable and pointer storage. As such, page zero is somewhat special. Page zero locations are used extensively by the Apple monitor, DOS, and most languages such as BASIC and Pascal. If you are going to be calling a machine language program from one of these languages (or using DOS from an assembly language program), you have to be very careful about using zero page locations. If you attempt to use a zero page location that is being used by the host language (or subsystem such as Apple DOS), then a "zero page conflict" may arise, and your program may not behave properly. To help you avoid using zero page

****

locations that may be utilized by one of the high-level languages, you should check out the zero page memory map in the new Apple reference manual (The White Book) on pages 74 thru 75.

Since a conflict may arise, why even use zero page locations? After all, there are 48,496 other RAM locations which can be used for variable storage; what's the big deal with zero page? Well, the designers of the 6502, realizing that page zero would be used quite often for variable storage, implemented a "zero page addressing mode." A zero page addressing mode instruction consists of a 1-byte instruction code followed by a 1-byte

address. Since one byte can only uniquely specify 256 different memory locations, this type of instruction can only reference one page of memory. You got it: page zero. Thus, a zero page instruction, since it only requires two bytes, saves you some memory (remember, absolute addressing mode instructions require three bytes). Another advantage to using zero page instructions is that zero page instructions execute faster than absolute addressing mode instructions (in fact a zero page addressing mode instruction executes in three-fourths the time required by an absolute addressing mode instruction).

An instruction automatically uses zero page address when:

1) The address reference is non-symbolic (i.e., a label is NOT used) and the value is less than $100.

   EXAMPLE:

```
        LDA $1
        STA $FF
        LDX $1
        ADC $25
```

2) The address reference is symbolic and the symbol was declared using the "EPZ" (Equate to Page Zero) pseudo opcode.

   EXAMPLE:

```
LBL     EPZ $0
LBLA    EQU $0
        LDA LBL          ;ZERO PAGE ADDRESSING USED
        LDA LBLA         ;ABSOLUTE ADDRESSING USED
```


****

Remember, the zero page addressing mode will only be used when the label is defined with the EPZ pseudo opcode. In all other cases (except non-symbolic mentioned above) the absolute addressing mode will be used.

ARRAYS IN ASSEMBLY LANGUAGE.

Single variables are nice, but often strings and arrays are required to accomplish a desired task. An array is a collection of data, each element of the array being of identical length (i.e., the same data type) to every other element. Arrays are stored in consecutive memory locations and any element of an array can be accessed by adding a displacement to the address of the first element.

First, how are arrays defined in an assembly language program? There are several methods. Basically, to reserve a block of memory you simply have to decide where in memory the array

is going to be located, and then not utilize that memory for anything else. Using this criterion, an array can be declared using the EQU pseudo opcode. For instance, let's assume you want to reserve 40 bytes (possibly to hold up to 40 characters for use in a display driver). Next, decide where in memory you want the array stored. Make sure, of course, that you do not define your array such that it will be sitting on top of your code or some other code such as DOS. Page three is a good place to store small arrays (unless, of course, you have some sort of driver already down there!). To define an array beginning at location $300 simply use the statement:

```
     ARRAY EQU $300 ;ARRAY IS $28 (40) BYTES LONG.
```

This statement says the array ARRAY begins at location $300; that's all this statement says. You, as the programmer, must make a mental note that the locations from $300 to $327 are being utilized by the array (hence the comment to the right). If you were to declare another array, say 10 bytes long, you would include:

```
     ARRAY EQU $300 ;ARRAY IS $28 (40) BYTES LONG
     ARRAY2 EQU $328 ;ARRAY IS $A (10) BYTES LONG.
```

This ensures that the memory space for ARRAY2 does not conflict with the memory space for ARRAY1

Obviously, performing the arithmetic yourself (especially if you don't have a TI Programmer) is quite tedious and error prone. A better way to declare arrays is:

****

```
ARRAY   EQU $300
ARRAY2  EQU ARRAY+$28
;
;ARRAY2'S LENGTH IS $A
;
```

This statement says that ARRAY2 begins 40 locations ($28) beyond the start of ARRAY. The comment after ARRAY2 simply states how long ARRAY2 is supposed to be, so you can add more data onto this list later on, should you so desire.

Using the EQU statement has two disadvantages. The first disadvantage is that you must know, as you are writing the program, where the array will be stored in memory. Generally, this leads to inefficient coding, especially when declaring large arrays, because you are never sure where the end of your program is (the end is generally a good place to put an array so that the program consists of one big chunk). The second drawback is the fact that you must always remember the length of the last declared array in the event you wish to add more arrays later on. It would be nice if one could say, "Hey! Reserve me 10 bytes here (wherever "here" is) for my array, and then continue with the code after these 10 bytes.

The ORG pseudo opcode allows you to do exactly that. The ORG pseudo opcode (program ORiGin) simply sets the value of the location counter to the address specified in the address expression in the operand field. The location counter is a pointer that determines where the current assembly language code is supposed to be stored. Generally, when you assemble a program (without an explicit ORG) the program is automatically stored beginning at location $800. During assembly, as each byte of code is created, it is stored at the location pointed to by the location counter, and then the location counter is incremented by one. Each time a label is encountered within the assembly language program, the symbol is stored in the symbol table along with the contents of the location counter when the symbol was defined (with the obvious exception of EQU and EPZ which store the address in the operand field in the symbol table along with the label). Consider the following assembly language program:

```
                ORG $800        ;DEFAULT VALUE
                JMP LABEL       ;THREE BYTE INSTRUCTION
          ARRAY ORG $903
          LABEL ---             ;THE REST OF YOUR PGM GOES HERE
```


****

In this example the assembler is instructed to begin the program at location $800 (the default value). Immediately following the ORG statement is a JMP instruction. Since JMP's are always three bytes long, the next code generated will be stored at location $803. The next instruction contains a label ('ARRAY'), so this label is stored in the symbol table along with the current value of the program counter. Note that, with the exception of EQU and EPZ, the label is stored in the symbol table before the code for the instruction on that line is emitted (or, if you have a pseudo opcode such as ORG, before the instruction is executed). As a result of this, ARRAY is stored in the symbol table with the value $803 (the current value of the location counter). Next, the ORG pseudo opcode gets executed and the location counter is forced to contain the value $903. Notice that LISA has just made room for a 256-byte array within the program itself. The only drawback to this method of reserving memory is that you must know the current value of the program counter in order to use it. Often this is impossible, so it seems to exclude the use of the ORG statement as a means of reserving memory.

But wait! Whenever the assembler sees an asterisk ('*') in the operand field, it will substitute the current value of the location counter in its place. Rather than guessing (educated or otherwise) about the current value of the location counter, you can use the '*' and be assured of getting the correct value. Now you can reserve 256 bytes as follows;

```
                ORG $800
                JMP START
          ARRAY ORG *+$100     ;RESERVE 256 LOCATIONS
```

```
          START   ---           ;CODE GOES HERE
```

Another problem which surfaces is the age old problem known as "separation of program and data." If you place an array inside your program, you must insure that the code will not get executed as data. Otherwise, unexpected results may be obtained. In the previous example you will note that a JMP instruction caused program execution to jump over the array. In general, arrays and other data stored within your program should only follow instructions which unconditionally alter the flow of the program. Such instructions include JMP, RTS, and BRK.

Another problem when using the ORG pseudo opcode surfaces when using the OBJ pseudo opcode. First, since it is a new

****

instruction, what is the OBJ pseudo opcode? LISA, during assembly, uses every memory location in the APPLE II computer except memory in the range $0800 to $1800. (This may vary depending upon how you have initialized the system, but the above are the default values.) What happens if you want to be able to run your assembly language program at location $4000? You cannot assemble your program at location $800 (the default location), and then simply move your code to location $4000 and execute it. Most 6502 assembly language programs are not RELOCATABLE. (Relocatable means that a program can be executed anywhere in memory without any problems.) Unfortunately, all those JMP's, JSR's, etc. reference absolute memory locations. If you assembled the program:

```
                ORG $800
                JMP LBL
          ARRAY ORG *+$100
          LBL   LDA ARRAY
                STA ARRAY+$1
                BRK
                END
```

and then moved it to location $4000 before running it, the program would not execute as planned. Since the program was ORG'd for location $800, all absolute addresses (such as the address of LBL and ARRAY) will simply be offsets from this initial address. In this case ARRAY will be assigned the address $803 and LBL will be assigned the address $903. Now, when the code is generated for this program, $903 will be substituted for LBL, which means the first JMP instruction will be converted to JMP $903. If you move the code to location $4000 and try to execute it with the 4000G monitor command, the first instruction (JMP $903) will simply continue execution of the program at the location at which the program was originally assembled.

Does this mean that all assembly language programs you write must reside in the locations $800 to $2000? Definitely not! It simply means that while you are assembling your code, the

object code (the machine language instructions produced by LISA) has to be stored in this range. Now, whenever the ORG pseudo opcode is encountered, the code counter (a pointer that determines where the code will be stored in memory) is changed to the value in the operand field as is the location counter. This means that if your program contains an ORG $4000 instruction,

****

not only will the code be assembled to run at location $4000, but it will also be stored there. Since this is a no-no location for object code (LISA normally stores the textfile in this region and storing object code here may "clobber" part of your textfile) some means must be used to make ensure that the object code gets stored in the range $800-$1800.

The OBJ pseudo opcode does this for you. The OBJ pseudo opcode simply changes the value in the code counter (the pointer to where the object code is being stored) to whatever address appears in the operand field. If you want to assemble your program to run at location $4000, you should use the code:

```
                ORG $4000
                OBJ $800
                LDA #$0
                STA LBL
                BRK
        LBL     EQU $0
                END
```

What does all this have to do with declaring arrays? Let's consider the following program:

```
                ORG $4000
                OBJ $800
                JMP LBL
        ARRAY   ORG *+$100
        LBL     LDA ARRAY
                STA ARRAY+$1
                BRK
                END
```

The ORG $4000 pseudo opcode insures us that the correct code will be generated. The OBJ $800 pseudo opcode ensures that the code will be stored in the memory range $800 - $1800. The JMP instruction ensures that the data will not get interpreted as instruction code. BUT, the ORG *+$100, used to reserve memory for the array, causes a slight problem. Remember, the ORG pseudo opcode resets the location counter as well as the code counter. This means that when "ORG *+$100" is encountered, The location counter will be set to $4103 (which we want), but the code counter will also be set to $4103 (which we don't want), thereby clobbering the textfile. Including the instruction "OBJ *+$100" does not complete solve the problem either. Since the location counter is already $4103 by the time the OBJ *+$100

would get executed, the inclusion of such an instruction would be

****

futile. There are ways of handling this problem, but fortunately LISA offers a better alternative.

LISA provides the programmer with another instruction, DFS (define storage), to handle this problem for you. When a DFS pseudo opcode is encountered, LISA will increment both the program counter and the code counter by the number of bytes specified in the operand field. To reserve 256 bytes as in the last example, you would write:

```
         ORG $4000
         OBJ $800
         JMP LBL
ARRAY    DFS $100
LBL      LDA ARRAY
         STA ARRAY+$1
         BRK
         END
```

and the DFS statement would automatically reserve the memory bytes for you. When using the DFS pseudo opcode, you must still place the array where it will not get executed as code, and likewise, since the data will be stored within your program, programs which use the DFS statement are not ROMable. DFS, incidently, can be used to define single variables as well as arrays. Simply use DFS $1.

INITIALIZING ARRAYS AT ASSEMBLY TIME.

Sometimes it is necessary to initialize an array at assembly time. For instance, you may need to store a data table in memory, or initialize some string, or define some one time initialization data. None of the methods discussed so far allow for this. The memory space was allocated but no particular values were stored in the array. Typically, you will need to store two types of data in an array. Either numeric data (be it binary, decimal, or hex) or string data (ASCII characters). Memory can be initialized with hexadecimal data by using the HEX pseudo opcode. This pseudo opcode is particularly useful in setting up tables. The HEX pseudo opcode is used in your program as follows:

```
         JMP LBL
ARRAY    HEX 00010203
LBL      LDA ARRAY
         STA $0
         BRK
         END
```

****

In this example, the accumulator is loaded with the value contained at location ARRAY, which is zero. The next instruction stores the accumulator at location $0 in memory. The HEX pseudo opcode expects two hex digits for each entry, otherwise you will get an error. You will note that two digits had to be typed for each hexadecimal value (complete with leading zeros) in the previous example. Each value (starting at the first value in the hex string, naturally) is stored in successive memory locations.

To initialize some memory locations with ASCII data you should use the ASC pseudo opcode. It is used as follows:

```
        JMP LBL
ARRAY   ASC "HI THERE"
LBL     LDA ARRAY
        JSR $FDED
        LDA ARRAY+1
        JSR $FDED
        LDA ARRAY+2
        JSR $FDED
        LDA ARRAY+3
        JSR $FDED
        LDA ARRAY+4
        JSR $FDED
        LDA ARRAY+5
        JSR $FDED
        LDA ARRAY+6
        JSR $FDED
        LDA ARRAY+7
        JSR $FDED
        BRK
        END
```

In case you are wondering, this program prints the message "HI THERE" on the video screen (without the quotes). The ASCII string following the ASC pseudo opcode must be enclosed in quotes or apostrophes. For now, always use the quotes. The use of the apostrophe will be discussed later.

What happens if you want to include a quote within the quoted string? Simply double up the quotes to get a single quote character stored in memory.

EXAMPLE:

```
ASC "HOW'S ""THIS"""
```

The first occurrence of the quote establishes the quote as the "delimiter" character. Since the quote is the delimiter, apostrophes can freely appear in the string. To include a quote within a quoted string use two quotation marks in succession. This is a

****

signal to LISA that the quote is not actually a delimiter, but the quote character. The above example would generate the following charters in memory:

```
HOW'S "THIS"
```

Now that you can reserve memory for array storage, how are array elements accessed? Accessing individual elements is very easy. You use the address of the first element of an array and add in a displacement to this address. For example, if you want to access the tenth element of the array ARRAY, you would use ARRAY+$9 as your address (remember, arrays in assembly language start with an index of 0). Several of the previous examples have used this method for accessing array elements. This, however, is a static displacement, meaning that the address remains constant at run time, and is calculated only at assembly time. Thus, if you try a statement of the form;

```
LDA ARRAY+I
```

your program will not add the contents of memory location I to the address of ARRAY and load the accumulator from that location. Rather, LISA will immediately add the address of I to the address of ARRAY and use that as the location from which the accumulator will be loaded. If the value contained in I was stored at location $1000 and the array ARRAY began at location $2000, the LDA ARRAY+I statement would load the accumulator from location $3000 which is the address formed by the computation of ARRAY+I. So how does one simulate the variable index feature of arrays found in high-level languages? To get that question answered, read on...

USING INDEX REGISTERS TO ACCESS ARRAY ELEMENTS.

The 6502 X- and Y-index registers can be used to dynamically access elements of an array. This type of operation is known as indexing, hence the name index register. When you use the indexed by X or indexed by Y addressing modes, the following procedure is carried out:

1) Add the contents of the desired index register to the address that follows the instruction.

****

2) Use this address as the actual address when referencing memory. To specify the indexed by X addressing mode, you use the syntax:

   <mnemonic> <address expression>,X

   EXAMPLES:

```
        LDA LBL,X
```

```
                STA ARRAY,X
                ADC $1,X
                SBC $FFFF,X
```

To specify the indexed by Y addressing mode, you use the syntax:

```
        <mnemonic> <address expression>,Y

        EXAMPLES:
                LDA LBL,Y
                STA ARRAY,Y
                ADC $1,Y
                SBC $FFFF,Y
```

Now consider the following examples:

****

```
        LDX #$1
        LDA ARRAY,X   ;LOADS ACC FROM LOCATION ARRAY+$1
        LDY #$FF
        STA STRING,Y  ;STORES ACC AT LOCATION STRING+$FF
```

Nothing really special happened here. The program loaded the accumulator from location ARRAY+$1 and then stored it at location STRING+$FF. We could have done this without using the index registers.

The beauty of the indexed addressing modes is that the X- and Y-registers can be changed under program control. As an example, suppose you want to clear the 256 bytes starting at location ARRAY (clearing an array means each element gets set to zero). To perform this operation, you could use the code:

```
                LDX #$0         ;INIT FOR 256 BYTES
                TXA             ;SET ACC = 0
        LOOP    STA ARRAY,X     ;STORE ZERO INTO MEMORY LOC
                INX             ;MOVE TO NEXT LOCATION
                BNE LOOP        ;DONE YET?
                BRK             ;IF SO, QUIT
        ARRAY   DFS $100        ;ARRAY STORAGE BEGINS HERE
                END
```

In this example, the X-register and the accumulator are loaded with $0. The accumulator is then stored at location ARRAY+X. Since the X-register contains zero, the accumulator is simply stored at location ARRAY. After this is accomplished the X-register is incremented by one, and now it contains the value one. Since the last result obtained was one, not zero, the BNE instruction causes a branch to location LOOP where once again the accumulator is stored at location ARRAY+X. The difference is that the X-register now contains one so the accumulator is stored at location ARRAY+$1. This loop is repeated over and over again until the X-register contains the value $FF. At that point incrementing the X-register will give you zero which will cause

the loop to terminate.

Since the X- and Y-registers are only eight bits long, you are limited to a range of 256 bytes when using the indexed addressing modes. For most applications this is sufficient (and, in fact, for

****

strings it is ideal). If you need to access more than 256 bytes in an array, read on: the next couple of sections will describe how.

INDIRECT ADDRESSING MODE.

The indirect addressing mode is a special addressing mode used only by the JMP instruction. It is presented here only because the discussion of the indirect indexed by Y and the indexed by X indirect addressing modes build on the concept of indirect addressing.

An indirect address is the address of the address of the desired location. Sound confusing? The following examples may help.

```
          JMP $800        ;JUMPS TO LOCATION $800
          JMP ($800)      ;JUMPS TO THE ADDRESS CONTAINED
                          ;IN BYTES $800 AND $801.
                          ;LOCATION $800 CONTAINS THE
                          ;LOW ORDER BYTE OF THE ADDRESS
                          ;AND $801 CONTAINS THE HIGH
                          ;ORDER BYTE OF THE ADDRESS.
```

For instance, if location $800 contained $4 and location $801 contained $09, then a jump would be made to location $904. This addressing mode allows you to simulate the CASE statement which appears in many languages (the ON...GOTO is the equivalent of the CASE statement in BASIC). In the following program, a jump will be made to location $800 if the X-register contains $0, to location $900 if the X-register contains $2, and to location $1000 if the X-register contains $4.

```
                  LDA LOCADR,X
                  STA JMPADR
                  INX
                  LDA LOCADR,X
                  STA JMPADR+$1
                  JMP (JMPADR)
          JMPADR  DFS 2           ;RESERVE TWO BYTES FOR JMPADR
          LOCADR  HEX 0008        ;ADDRESS TABLE IN BYTE
                  HEX 0009        ;REVERSEE ORDER
                  HEX 0010
                  END
```

The result of this program, if X contains 1, 3, or 5, is usually garbage. Why on earth would anyone want to make such a simple problem so complex when the following code accomplishes the same thing?

```
        CPX #0
        BNE LBL0
        JMP $800
LBL0    CPX #2
        BNE LBL1
        JMP $900
LBL1    JMP $1000
```

After all, the latter method takes less memory and seems much simpler to program. For this simple example, yes, the latter method is probably better, but keep in mind, for every additional jump you wish to handle, you need to add seven bytes to the latter version (a compare, a branch, and a jump instruction) and only two bytes to the former program segment (an address). The break-even point (in terms of code) is between four and five jumps.

Indirect jumps are useful for controlling the flow of a program in ways other than simulating a CASE statement. For example, suppose you want to write a character output routine for the APPLE II computer that will output the character in the accumulator to the Apple video screen. Once this task is accomplished, suppose you wish to expand your routine a little to allow output to a printer, modem, plotter, etc. Yet, you wish to keep the same entry point, so that a program that outputs data to the screen can just as readily output it to the printer. This can be accomplished readily by setting up a flag byte somewhere that outputs data to the screen if the flag byte is zero, to the printer if the flag byte is one, to the modem if the flag byte is two, etc. The routine to handle all this might be:

```
PUTCHR  PHA             ;SAVE CHARACTER TO BE OUTPUT
        LDA     FLAG    ;SEE WHERE THE OUTPUT GOES
        BEQ     SCROUT  ;OUTPUT TO THE SCREEN IF 0
        CMP     #1
        BEQ     PRTOUT  ;OUTPUT TO PRINTER IF 1
        CMP     #2
        BEQ     MODEM   ;OUTPUT TO MODEM IF 2
        ETC...
```

As before, this method works quite well if there are only a few types of output devices. However, there are two problems with this method. First, you cannot anticipate all the devices which will

****

be attached to a computer when writing this type of program. The second problem with this approach is that peripheral initialization has not been taken into account. With some peripherals you must jump to an initialization address the first time the peripheral is accessed and then to a "normal" entry address thereafter.

The indirect addressing mode can solve all of these problems. Instead of reserving a memory location for a flag register, let's reserve two memory locations to contain the address of the device handler we wish to access. Let's just arbitrarily choose locations $36 and $37 in zero page to hold the low and high order bytes (respectively) of the address of the routine we wish to access. Normally, these two locations will contain the address of our video output routine. When we wish to direct the output to the printer, we simply place the low-order byte of the address of the printer routine in location $36 and the high-order byte of the address of the printer routine in location $37. Our character output routine will now consist of exactly one instruction, a "JMP ($36)" instruction. Jumping to this instruction will cause the program to transfer control to the currently active device.

How does using the indirect jump solve the two aforementioned problems? The first problem (not knowing which devices will eventually be used with the computer) is not a problem at all. The indirect jump I/O handler takes all devices into account. If you wish to output data to some new type of device, all you need to do is load the address of the device handler into locations $36 and $37.

The second problem (device initialization) is also easy to handle. When a device is turned, on the address of its initialization routine is loaded into locations $36 and $37. The initialization routine initializes the device and then loads the address of the normal driver into locations $36 and $37. This causes subsequent accesses to jump directly to the normal entry point of the device handler.

More details on this type of operation will be considered later. For now, the concept of indirect addressing is all that is important.

INDIRECT INDEXED ADDRESSING.

Indirect addressing is only available for the JMP instruction and is not available for the loads, stores, compares, etc. For these

****

types of instructions, two composite forms of indirect addressing are available; indirect indexed by Y addressing and indexed by X indirect addressing.

The indirect indexed addressing mode is actually two addressing modes in one. It combines the indirect addressing mode with the indexed by Y addressing mode. The effective address is computed by going to the address specified after the opcode, and getting the indirect address stored there and in the succeeding location. Once this indirect address is obtained, the value contained in the Y-register is added in to give the final, effective address. Note that if the Y-register contains zero, true indirect addressing is performed.

The indirect indexed by Y addressing mode has one further restriction. The address at which the indirect address is stored must be a zero page location. Accordingly, all instructions which use the indirect indexed by Y addressing mode are two bytes long (one byte for the opcode and one byte for the zero page address). Also, should you use a symbolic reference, it must be declared using the EPZ pseudo opcode or you will get an error.

EXAMPLE OF INDIRECT INDEXED BY Y ADDRESSING MODE:

```
                LDA #$0         ;INIT FOR LOCATION $900
                STA $FE         ;L.O. BYTE IN LOCATION $FE
                LDA #$9         ;
                STA $FF         ;H.O BYTE IN LOCATION $FF
                LDY #$0         ;INIT TO START AT LOCATION $900
                TYA             ;INIT ACC TO ZERO
        LOOP    STA ($FE),Y     ;STORE AT LOCATION POINTED AT
                INY             ;BY ($FE,$FF) + CONTENTS OF Y
                BNE LOOP        ;GO TO NEXT, DONE YET?
                BRK             ;IF SO; QUIT
                END
```

This program should look familar; it's the memory clear routine which we used earlier with the straight indexed by X and Y addressing modes.

So far, we haven't really done much other than make the solution more complex (by adding indirection), and we still can't access more than 256 bytes at a time. To give a preview of things to come, it is possible to increment the two memory locations $FE and $FF (a sixteen-bit memory increment). With this in mind we can leave the Y-register at zero and simply perform an extended

****

increment on the memory locations $FE and $FF. Since this is a sixteen-bit address, you can access any memory location in memory by using this technique. Incrementing sixteen bits will be described in later chapters, so file this knowledge away for a while.

INDEXED INDIRECT ADDRESSING MODE.

In the indirect addressing mode, the indirect address was determined and then the Y-register was added to this address to

give an effective address. As the name implies, the indexing (by the Y-register) is performed after the indirect address calculation.

Indexed indirect addressing, as its name implies, performs the indexing operation first. This addressing mode uses the X-index register and has the following syntax:

<mnemonic> (<address>,X) The contents of the X-register are added to whatever value the address expression may have. The resulting zero page address (wrapping around if necessary) and the following byte contain the address of the location to be used.

The indexed by X,indirect addressing mode would probably be useful when you have a table of pointers in page zero and need to access different sections of memory depending upon some value in the X-register. Zero page, however, is a limited resource and using it to hold large tables is not a good idea. This author, after programming the 6502 for three years, has needed to use the indexed by X, indirect addressing mode only once or twice. Further uses of the indexed by X, indirect addressing mode will be left to the discovery of the reader.

Since indexed indirect addressing is not used nearly as often as indirect indexed addressing, a lengthy discussion of this addressing mode will not follow.

Whenever you want to perform an indirect load, store, or other operation, you can use the indirect indexed by Y addressing mode or the indexed by X,indirect addressing mode with the respective register set to zero.

Indirect addressing techniques are very useful and give the 6502 microprocessor quite an advantage over other processors which do not have this addressing mode available. Since this concept will be used throughout the rest of this book, make sure that you understand this addressing mode before proceeding.

****

CHAPTER 9

LOGICAL, MASKING, AND BIT OPERATIONS

NEW INSTRUCTIONS:

AND ORA XOR/EOR BIT
ASL LSR ROL ROR

GENERAL.

In the wonderful world of computers, data is not always treated as characters and numbers. As such, arithmetic and comparisons (the operations required for operation on numbers and

characters) do not prove sufficient for all computer applications. One important data type, which has not yet been discussed much, is the Boolean data type. Arithmetic has no meaning for the Boolean data type, and, therefore, some new operations have to be included in our basic instruction set.

There are four basic Boolean operations. They are complement, AND, OR, and exclusive-OR. The AND and OR operations should already be familar to BASIC programmers, as these operations are included in the BASIC instruction set. The complement and exclusive-OR operations will probably prove completely foreign. Even the AND and OR operations, which sound familar to the BASIC programmer, are actually a little different from their BASIC counterparts.

In order to help make Boolean functions easier to understand, this book will use "truth tables" to help demonstrate the actions of the various functions. A truth table is no more than a listing of the possible output values for all possible inputs. A function may only have one output value (by definition), but it may have as many input values as desired. In our Boolean functions, all functions will be restricted to one or two input values and only

****

one output value. Since Boolean values are either false or true (represented by zero or one respectively), single input functions can produce only one of two outputs. Likewise, two input Boolean functions can produce one of four output values (not necessarily different).

COMPLEMENT FUNCTION.

The complement function is a one-input function. If you input a bit, the complemented (i.e., opposite) value is returned. If you pass the complement function the true value (one), then the false value (zero) is returned. If you pass the complement function the false value (zero), then the true value (one) is returned. The complement function truth table is shown in table 9-1.

Table 9-1. Complement Function Truth Table

Input bit	Output bit
A	X
0	1
1	0

Table 9-1 (above) simply states that if you give the complement function a value "A" of 0, you will be returned a value "X" of 1. Conversely, if you pass the complement function a value "A" of 1 then you will be returned a value "X" of 0. The complement function is sometimes called the 'NOT' function (i.e., NOT TRUE

is false and NOT FALSE is true), and sometimes it is called the "one's complement."

AND FUNCTION.

The AND operation requires two input values; it returns one result value. The result returned is TRUE if and only if the two input values are true. The result is FALSE otherwise. The AND function truth table is shown in table 9-2.

****

Table 9-2. AND Function Truth Table

Input bits		Output bits
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Table 9-2(above) states that if A and B are 1 (true), then X is 1; if A or B is 0 (false), then X is 0.

Obviously, the AND function has uses in extended comparisons (e.g., simulating IF((A=B) AND (C<=D))). What is probably not so clear is the AND function's masking abilities. The AND function allows you to force a bit off, should this action be required. Assuming the input A to be variable, you can always force the output to become zero by setting the input B to zero. By setting input B to one, you will pass A unchanged. By studying the truth table, you will notice that whenever B is zero, the output is also zero. Also, whenever B is one, the output corresponds exactly to the A input. This feature is known as "masking" and will be used considerably later on.

OR FUNCTION.

Like the AND function, the OR function requires two inputs and produces a single bit output. The OR function returns true if A or B (or both) is true, and returns false otherwise (i.e., if A and B are both false). The OR function truth table is shown in table 9-3.

Table 9-3. OR Function Truth Table

Input bits		Output bits
A	B	X
0	0	0
0	1	1

****

1	0	1
1	1	1

The OR function has the obvious use of simulating the "IF (A=B) OR (C<=D)" statement, but, just like the AND function, the OR function has an additional masking use. If A is a variable, then OR'ing A with B, when B is zero, always returns A. OR'ing A with B, when B is one, always returns one. This function allows you to force a bit on. This masking function will prove to be very useful later on.

EXCLUSIVE-OR FUNCTION.

The exclusive-OR function is another two-input, single-output type function. It returns true if A or B, but not both, is true. It returns false if A and B are both true or A and B are both false. The exclusive-OR (often abbreviated XOR) function truth table is shown in table 9-4.

Table 9-4. Exclusive-OR Function Truth Table

Input bits		Output bits
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

The XOR function has two interesting features. First, in masking operations, it can be used to invert (i.e., complement) the desired input. In this mode, if B is zero, A is passed unchanged. If B is one, the result of the XOR function is the complement of A. This can be verified by studying the XOR truth table.

The XOR function is also a "not equals" function. XOR returns false if A equals B, and XOR returns true if A does not equal B. Although the 6502 CMP instruction can be used for this type of testing, the XOR function is necessary when the contents

****

of the carry flag cannot be modified. The CMP instruction modifies the carry flag whereas the XOR function does not.

BIT STRING OPERATIONS.

So far, all logical functions have been defined in terms of

one or two input bits and one output bit. Unfortunately, the 6502 (being an eight-bit microcomputer) works with eight bits at a shot. As such, the logical operations have to be defined in terms of eight bits.

The process used to logically operate on eight bits is known as the "bit-by-bit" operation. To perform a "bit-by-bit" logical operation, you must take two bytes, perform the operation on bit zero of each byte, and store the result in bit zero of the result byte. You then perform the operation on bit one of each byte and store the result in bit one of the result byte. This process is repeated for bits two, three, four, five, six, and seven.

```
(10011110) AND (11000111) = (10000110)
(11110000) OR  (00001111) = (11111111)
(11001100) XOR (11110000) = (00111100)
           NOT (11011011) = (00100100)
```

INSTRUCTIONS FOR LOGICAL OPERATIONS.

AND INSTRUCTION.

The 6502 allows you to AND the value in the accumulator with a value in memory or a constant. The result is left in the accumulator and the Z and N flags are set accordingly. The 6502 instruction mnemonic is AND.

EXAMPLES:

```
LDA #$FF  - LOAD ACC WIUH $FF     1111 1111
AND #$0F  - "AND" ACC WITH $F     0000 111
          - RESULT LEFT IN ACC    ---------
            IS $F                 0000 1111

LDA #$2F  - LOAD ACC WIUH $2F     0010 1111
AND #$01  - AND WITH $1           0000 0001
          - RESULT LEFT IN ACC    ---------
            IS $1                 0000 0001
```


****

The N flag is set if bit seven of the result is one. The zero flag is set if the result of the AND operation is zero. One interesting use of the AND instruction is to test to see if a bit is set or not. For instance, if you want to see if bit zero of the accumulator is set, simply AND the accumulator with the value $1. If bit zero is set (i.e. one), the accumulator will contain one after the AND operation, and likewise the Z flag will be reset, so you can use the BNE instruction to test for this. If bit zero of the accumulator is not set, the accumulator will contain zero after the AND operation and the BNE test will fail. To test whether or not a memory location contains a zero or one in bit zero, load the accumumator with the constant $1 and then AND the accumulator with that memory location.

The bit test feature of the AND instruction is very useful, except that the contents of the accumulator are modified. When performing simple bit tests, it is sometimes convenient to leave the contents of the accumulator alone. This can be accomplished by the use of the BIT (BIt Test) instruction. The BIT instruction AND's the accumulator with an absolute or zero page memory location (only!) and the results of this AND operation are used to set the N, Z, and V flags. The flags are set according to the following rules:

1) Bit seven of the memory location is loaded into the N flag (not the result of memory bit seven AND ACC bit seven)

2) Bit six of the memory location is loaded into the V flag.

3) The result of ACC AND memory is used to set the Z flag (identical to the AND instruction).

The BIT instruction is especially useful for input/output hand-shaking and control. This instruction will be discussed in more detail in later chapters.

ORA INSTRUCTION.

To perform the logical OR function, the 6502 ORA instruction (OR Accumulator) is used.

EXAMPLES:

```
LDA #$00 -LOAD ACC WITH $0      0000 0000
ORA #$FF -OR ACC WITH $FF       1111 1111
         -RESULT OF $FF IS      ---------
         -LEFT IN THE ACC       1111 1111
```


****

```
LDA #$04 -LOAD ACC WITH $4      0000 0100
ORA #$30 -OR ACC WITH $30       0011 0000
         -THE RESULT OF $34     ---------
         -IS LEFT IN THE ACC    0011 0100
```

XOR/EOR INSTRUCTION.

The standard 6502 exclusive-OR mnemonic is "EOR." Since XOR is frequently used in the world of digital computers, LISA supports the use of both XOR and EOR as mnemonics for the exclusive-OR function. Both generate identical code; the choice of which mnemonic to use is strictly up to you.

EXAMPLES:

```
LDA #$AA -LOAD ACC WITH $AA     1010 1010
XOR #$01 -XOR WITH $01          0000 0001
         -RESULT IS $AB WHICH   ---------
         -IS LEFT IN THE ACC    1010 1011
```

```
          LDA #$AA -LOAD ACC WITH $AA     1010 1010
          EOR #$01 -XOR WITH $01          0000 0001
                   -RESULT ($AB) IS       ---------
                   -LEFT IN THE ACC       1010 1011
```

COMPLEMENTING THE ACCUMULATOR.

The 6502 does not have a complement instruction in its basic instruction set. Since the XOR function can be used to invert selected bits, the XOR instruction will be used to invert the accumulator. This is accomplished by exclusive-OR'ing the accumulator with the constant $FF.

EXAMPLES:

```
          LDA #$00 -LOAD ACC WITH $00     0000 0000
          XOR #$FF -INVERT ACC            1111 1111
                   -RESULT ($FF) IS       ---------
                   -LEFT IN ACC           1111 1111

          LDA #$AA -LOAD ACC WITH $11     1010 1010
          XOR #$FF -INVERT ACC            1111 1111
                   -RESULT ($55) IS       ---------
                   -LEFT IN ACC           0101 0101

          LDA #$55 -LOAD ACC WITH $55     0101 0101
          XOR #$FF -INVERT ACC            1111 1111
                   -RESULT ($AA) IS       ---------
                   -LEFT IN ACC           1010 1010
```


****

MASKING OPERATIONS.

Up to this point, the discussion of AND, OR, and XOR/EOR has been rather academic. Why you would want to use these instructions, as well as when you should use them, has not really been addressed. Setting specific bits (using the ORA instruction), clearing specific bits (using the AND instruction), and inverting specific bits (using EOR/XOR) seem "neat," but of what practical value are they?

MASKING OUT.

Suppose memory location VAR contains two distinct values, one value in the high-order nibble and another value in the low-order nibble. In a particular application, we may be interested only in the value contained in the low-order four bits. The high-order four bits should be set to zero. Solving this problem does not prove to be too difficult. By loading the accumulator from location VAR; then AND'ing the accumulator with the constant $0F, the high-order nibble is "masked out" leaving zero in the

****

high-order (H.O.) four bits and the low-order (L.O.) nibble in the low-order four bits.

EXAMPLE:

```
LDA VAR    -GET VAR INTO ACC
AND #$0F   -MASK OUT H.O. NIBBLE LEAVING L.O. NIBBLE
```

Another problem frequently encountered is that of packed data. Suppose, in order to save memory, you have packed eight Boolean values (each requiring one bit) into one byte. Somewhere within the program you wish to test a Boolean flag to see if it is true or false. Loading the accumulator with that particular byte and then using the BTR and BFL instructions is not sufficient. The BTR branch would be taken if any of the bits are set (remember, BTR is the same as BNE). Likewise, BFL will only be taken if all of the bits are false (because the BEL instruction is really the BEQ instruction). Some means of testing only one bit is highly desirable. This can be accomplished using the AND instruction. If you want to test a particular bit, simply mask out all the other bits. If the desired bit is false (i.e. zero), the zero flag will be set and the BFL instruction can be used to test this condition. If the desired bit is true (i.e. one), the AND'ing operation will leave a one bit set somewhere within the byte and the BTR instruction can be used to test this condition. EXAMPLE:

```
TO TEST BIT #0
     LDA BITS
     AND #%1
     BTR THERE

TO TEST BIT #1
     LDA BITS
     AND #%10
     BTR THERE

TO TEST BIT #2
     LDA BITS
     AND #%100
     BTR THERE

TO TEST BIT #3
     LDA BITS
     AND #%1000
     BTR THERE
```


****

```
TO TEST BIT #4
     LDA BITS
     AND #%10000
```

```
          BTR THERE

     TO TEST BIT #5
          LDA BITS
          AND #%100000
          BTR THERE

     TO TEST BIT #6
          LDA BITS
          AND #%1000000
          BTR THERE

     TO TEST BIT #7
          LDA BITS
          AND #%10000000
          BTR THERE
```

Another use of the AND instruction is that of the MOD function. (The MOD function is the remainder function; that is, X MOD Y returns the remainder after the division of X and Y.) AND'ing with $1 returns the value in the accumulator MOD two. AND'ing the accumulator with $3 returns the value in the accumulator MOD four. AND'ing with $7 returns the value in the accumulator MOD 8. AND'ing with $F returns the value in the accumulator MOD 16. AND'ing with $1F returns the value in the accumulator MOD 32. AND'ing with $3F returns the value in the accumulator MOD 64. AND'ing with $7F returns the value in the accumulator MOD 128. AND'ing with $FF simply returns the value in the accumulator.

This feature can be utilized in several instances. In the previous example, testing a particular bit, the programmer had to know which bit needed testing. Sometimes it would be nice if the program itself could decide which bit to test. In this capacity, a subroutine could be made of the bit testing procedure, and some dynamic value could be passed to the subroutine specifying which bit is to be tested. If the value is passed in the X-register, then the data at location 'BITS' can be tested against a value contained in a table. The indexed by X addressing mode could be used to specify which data byte is to be used as a mask.

****

EXAMPLE:

```
     TSTBIT  LDA BITS
             AND TBL,X
             RTS

     TBL     BYT %00000001
             BYT %00000010
             BYT %00000100
             BYT %00001000
             BYT %00010000
             BYT %00100000
```

```
                BYT %01000000
                BYT %10000000
```

Now, to test a particular bit, simply load the X-register with the bit number of the desired bit (0-7) and JSR TSTBIT. Upon return, the zero flag will be set if the particular bit is one, and the zero flag will be reset if the particular bit is zero. There is one slight problem with this scheme. What happens if the X-register contains a value outside the range 0-7? Obviously, the memory locations past the eighth byte in the table will be used as the mask. This usually gives you junk as a result. What is required is some means of insuring that the value in the X-register never exceeds $7. There are two simple ways of accomplishing this task. The first is to explicitly compare the X-register to eight, and abort if the X-register is greater or equal to eight. The other method is to AND the value in the X-register with #$7 which will return the original contents MOD eight. The AND'ing version is a little cleaner and should be used if you can tolerate testing bit zero when the X-register contains eight. It should be noted that the AND instruction can be used to force 'wrap around' during increments, decrements, additions, etc., long before $FF is reached.

Since the X-register cannot be directly AND'ed with a memory location, we will have to transfer the X-register to the accumulator, AND the accumulator with a value in the table, and then transfer the value back to the X-register. Some of this work can be eliminated by passing the index to the subroutine in the accumulator to begin with. The completed subroutine would look something like:

****

EXAMPLE:

```
     TSTBIT  AND #%0111
             TAX
             LDA BITS
             AND TBL,X
             RTS

     TBL     BYT %00000001
             BYT %00000010
             BYT %00000100
             BYT %00001000
             BYT %00010000
             BYT %00100000
             BYT %01000000
             BYT %10000000
```

The AND instruction can also be used to set one of the particular Boolean values to false, since the AND instruction can be used to force a particular bit to zero. The following is a program which is used to set the desired bit to false within the byte 'BITS.' As before, the accumulator contains the index of the element that

is to be set to false.

EXAMPLE:

```
SETFLS  AND #$7
        TAX
        LDA BITS
        AND TBLS,X
        STA BITS
        RTS

TBLS    BYT %11111110
        BYT %11111101
        BYT %11111011
        BYT %11110111
        BYT %11101111
        BYT %11011111
        BYT %10111111
        BYT %01111111
```

Note that there are two differences between this program and the last. First, the value is stored back into BITS after the AND operation. This assures us that the value will be around when we need it later on. Second, the data in the table is inverted. The data is inverted (as compared to the previous table) because we do not want to mask out the undesired values, only the value we wish set to false.

****

MASKING IN.

The 6502 ORA instruction can be used to force bits on. This feature allows us to set a particular value in our bit array to true. The code to perform this operation is:

```
SETRUE  AND #$7
        TAX
        LDA BITS
        ORA TBL,X
        STA BITS
        RTS

TBL     BYT %00000001
        BYT %00000010
        BYT %00000100
        BYT %00001000
        BYT %00010000
        BYT %00100000
        BYT %01000000
        BYT %10000000
```

Once again, the index is passed in the accumulator and the resultant value is stored in BITS.

The ORA instruction has several other uses besides setting Boolean variables to true. It can be used, for instance, to see if two or more bytes in memory are all equal to zero. To perform this function simply load the accumulator with the first byte, and then OR the accumulator with each of the successive bytes. Upon termination, the Z flag will be set if all the bytes contained a zero result. If any of the bytes in question did not contain zero, the Z flag will be reset.

EXAMPLE:

```
            LDA BYTE1
            ORA BYTE2
            ORA BYTE3
            ORA BYTE4
             .    .
             .    .
             .    .
            ORA BYTEn
            BEQ ALLZER
```

SHIFT AND ROTATE INSTRUCTIONS.

The 6502 supports four shift and rotate instructions. They are: arithmetic shift left, logical shift right, rotate left, and rotate

****

right. These instructions, in their simplest form, operate directly upon the accumulator contents. This is the 6502 accumulator addressing mode.

ARITHMETIC SHIFT LEFT (ASL) INSTRUCTION.

The arithmetic shift left instruction shifts all the bits in the accumulator one position to the left. Bit zero is shifted into bit one, bit one is shifted into bit two, bit two is shifted into bit three, etc. A zero is shifted into bit zero, and bit seven is shifted into the carry flag. The 6502 mnemonic for arithmetic shift left is 'ASL.' When shifting the contents of the 6502 accumulator, this instruction does not have an operand.

EXAMPLE OF ASL:

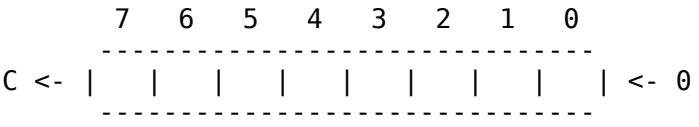
```
     MOVE THE LOW ORDER NIBBLE
     INTO THE HIGH ORDER NIBBLE

            LDA VALUE
            ASL
            ASL             ;FOUR SHIFTS MOVE THE L.O.
            ASL             ;FOUR BITS INTO THE H.O.
            ASL             ;FOUR BITS (L.O. FOUR BITS
            STA VALUE       ;BECOME ZERO)
```

Since the carry out of bit seven ends up in the carry flag, you can

use the BCC and BCS instructions to test for a 'shift overflow.'
Note (as demonstrated in the example) that the ASL instruction

****

ASL

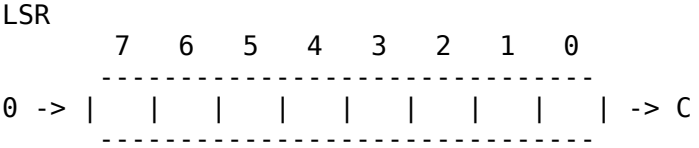


only shifts one bit. If you need to shift more than one bit position, you must execute several ASL instructions.

LOGICAL SHIFT RIGHT (LSR) INSTRUCTION.

The logical shift right instruction shifts data to the right (obviously). Zero is shifted into bit seven, bit seven is shifted into bit six, bit six is shifted into bit five, etc. Bit zero is shifted into the carry flag. Suppose you have two BCD digits which you want to separate into two bytes. (i.e., the low-order nibble goes into the first byte, and the high-order nibble goes into the low-order nibble of the second byte, in both cases the high-order nibble of the resulting bytes should be zero.) It's very easy to get the low-order

LSR



nibble into the first byte. Just load the accumulator from the memory location containing the BCD value, then AND the accumulator with $F and store the result in the first byte.

EXAMPLE:

```
LDA VALUE
AND #$F
STA LOC1
```

AND'ing the value with $F0 to get the high-order byte is not entirely satisfactory, because the value we desire will still be in the high-order nibble of the accumulator. By using the LSR instruction, this data can be moved down into the low-order four bits of the accumulator, at which point the data can be stored in the second byte of the destination address.

EXAMPLE:

```
LDA VALUE
AND #$F0
LSR
LSR
LSR
```

```
                LSR
                STA LOC1+$1
```


****

Since zeros are automatically shifted into bit number seven, after the four LSR instructions are executed, the accumulator would have all zeros in the high-order nibble anyway, so there is no need for the extra AND #$F instruction. BETTER CODE:

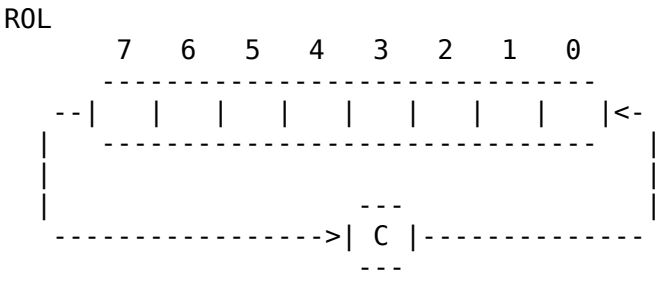
```
                LDA VALUE
                LSR
                LSR
                LSR
                LSR
                STA LOC1+$1
```

The final code segment might be:

```
                LDA VALUE
                AND #$F
                STA LOC1
                LDA VALUE
                LSR
                LSR
                LSR
                LSR
                STA LOC1+$1
```

ROTATE LEFT (ROL) INSTRUCTION.

The rotate left instruction is very similar to the arithmetic shift left instruction, with one difference. Instead of shifting zero into bit number zero, the previous contents of the carry flag are shifted into bit number zero; so for a rotate left, bit zero is shifted to bit one, bit one is shifted to bit two, bit two is shifted to bit three, ... , bit seven is shifted into the carry, and the carry is shifted into bit zero. Note that if you execute nine rotate left instructions in a



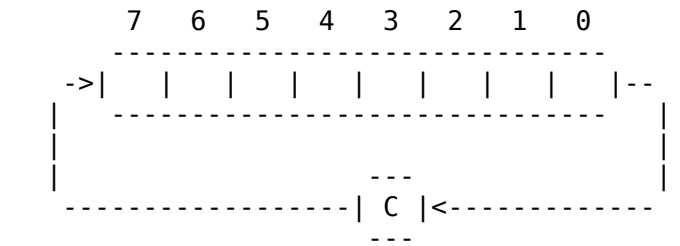
row, you end up with the value you started with in the accumulator. The 6502 mnemonic for the rotate left instruction is ROL.

****

ROTATE RIGHT (ROR) INSTRUCTION.

This instruction rotates the accumulator right with the carry flag going into bit seven and the carry out of bit zero ending up in the carry flag. The mnemonic for this instruction is ROR. As

ROR



with the ROL instruction, after nine rotates you end up with the value you started with in the accumulator.

SHIFTING AND ROTATING MEMORY LOCATIONS.

Until now, all shifts and rotates have only been used with the 6502 accumulator. The 6502 shift and rotate instructions can also be used to shift or rotate data in memory locations, effectively bypassing the accumulator (this is similar in operation to the INC and DEC instructions). If the operand field is not blank (which is required for the accumulator addressing mode), the operand field will be assumed to contain an absolute (or zero page) memory address. The contents of this memory location will be shifted or rotated with the same results as would be obtained if the accumulator had been operated upon. The indexed by X addressing mode is also available.

EXAMPLES:

```
ASL LOC1   -SHIFTS MEMORY LOCATION LOC1 LEFT
LSR TEMP   -SHIFTS MEMORY LOCATION TEMP RIGHT
ROL LBL+$1 -ROTATES MEM LOC. LBL LEFT
ROR X+$1   -ROTATES MEM LOC. X+$1 RIGHT
ASL        -SHIFTS THE ACCUMULATOR LEFT
LSR        -SHIFTS THE ACCUMULATOR RIGHT
ROL        -ROTATES ACC TO THE LEFT
ROR        -ROTATES ACC TO THE RIGHT
```


****

USING ASL TO PERFORM MULTIPLICATION.

Shifting any number to the left one position is identical to multiplying that number by its particular radix (i.e., base) For example, if you shift the decimal number 93 to the left one position you get 930 which is definitely ten times 93. In the

same way, shifting a binary value to the left one position is the same as multiplying it by two. A double shift to the left is identical to a multiplicauion by four; three shifts to the left, to a multiplication by eight; four shifts to the the left, a multiplication by sixteen; etc. In general, multiplication by powers of two is very easy, simply using one to seven ASL instructions to multiply by two, four, eight, 16, 32, 64, or 128.

EXAMPLES:

```
1) MULTIPLY ACC BY EIGHT
        ASL     ;TIMES 2
        ASL     ;TIMES 4
        ASL     ;TIMES 8

2) MULTIPLY ACC BY 32
        ASL     ;TIMES 2
        ASL     ;TIMES 4
        ASL     ;TIMES 8
        ASL     ;TIMES 16
        ASL     ;TIMES 32
```

You can test for overflow by sandwiching a BCS instruction between each ASL instruction. Should overflow occur (i.e., a carry out of bit number seven), the carry flag will not necessarily be set at the end of the shift left sequence, since the following ASL may clear the carry flag.

EXAMPLE:

MULTIPLICATION BY 16, TESTING FOR OVERFLOW

```
        ASL
        BCS ERROR
        ASL
        BCS ERROR
        ASL
        BCS ERROR
        ASL
        BCS ERROR
```

Often, the need arises to multiply by a constant other than a power of two. This is accomplished by breaking the multiplica-

****

tion problem down into several distinct steps and adding the results of these intermediate steps together. For example, multiplying the accumulator by three could be broken down into a multiplication by two and a multiplication by one (which is the original value itself).

EXAMPLE:

MULTIPLICATION BY THREE

```
        STA TEMP ;MAKE A TEMPORARY COPY
        ASL      ;MULTIPLY ACC BY TWO
        CLC      ;ADD IN THE ORIGINAL VALUE
        ADC TEMP ;TO GET 2xACC + ACC = 3xACC
```

To multiply by some other constant is just as easy. For instance, multiplication by six breaks down to a multiplication by four plus a multiplication by two.

EXAMPLE:

MULTIPLICATION BY SIX

```
        ASL      ;GET ACCx2
        STA TEMP ;AND SAVE
        ASL      ;MULTIPLY ACC BY FOUR
        CLC      ;ADD IN TEMP VALUE
        ADC TEMP ;TO GET 2xACC + 4xACC = 6xACC
```

One very important multiplication is multiplication by ten. This particular multiplication will be used quite a bit when converting between the binary and decimal bases. A multiplication by ten breaks down into a multiplication by two plus a multiplication by eight.

EXAMPLE:

MULTIPLICATION BY TEN

```
        ASL      ;MULTIPLY BY TWO
        STA TEMP ;SAVE
        ASL      ;MULTIPLY BY FOUR
        ASL      ;MULTIPLY BY EIGHT
        CLC      ;ADD IN TEMP VALUE
        ADC TEMP ;TO GET 10xACC
```


****

USING SHIFTS TO UNPACK DATA.

In Chapter 2, a discussion of packed data was briefly presented. Two BCD digits can be packed into one byte; eight Boolean values can be packed into a single byte; etc. Packing techniques save you memory at the expense of execution time and code complexity.

Data is "unpacked" by masking out all of the unwanted bits in a particular byte, and then shifting the data so that it is right-justified in the byte. A BCD number, for example, contains two fields: the high-order decimal digit and the low-order decimal digit. If you are interested in only the low-order decimal digit, all you have to do is AND the value with $F. This masks out all the unwanted bits (the high-order nibble) leaving the desired data right-justified. Getting at the high-order digit is not quite as simple.

In this case, the data must be shifted to the left four times so that it is right justified. Zeros are automatically shifted into the high-order nibble (refer to the discussion on shifts).

     But BCD is not the only case where data packing is performed. Some situations may require three data fields within a single byte. For example, you may have a Boolean value in bit seven; an Apple slot number in bits four, five, and six; and a hex value in the range $0-$F in the low-order nibble. Getting at the 4-bit value is easy, just AND the accumulator with $F. Getting at the three bits in the middle of the data structure is a little more complicated. First, you must shift the accumulator to the left four bits to right-justify the data field and to eliminate the low-order four bits. Next, the accumulator has to be AND'ed with $7 to eliminate the Boolean value and preserve the low-order three bits.

     EXAMPLE:

          UNPACKING THE MIDDLE FIELD

```
                LDA VALUE
                LSR           ;SHIFT RIGHT FOUR TIMES
                LSR           ;TO RIGHT JUSTIFY FIELD
                LSR           ;AND ELIMINATE L.O.
                LSR           ;NIBBLE
                AND #%0111    ;MASK OUT BOOLEAN VALUE
```

To unpack the Boolean field, you could perform seven LSR instructions. There is, however, a better way. First, AND the

****

accumulator with $80 to eliminate everything except the Boolean value. Next, shift the accumulator LEFT. Whatever value is contained in the Boolean variable will end up in the carry flag. Now, rotate the accumulator left to move the carry flag (i.e., the Boolean value) into the low-order bit of the accumulator.

     EXAMPLE:

          UNPACKING THE BOOLEAN FIELD

```
                LDA VALUE
                AND #$80
                ASL
                ROL
```

Obviously, if you just want to test the boolean value, you do not need to right-justify it. You need only to use the BTR/BFL instructions after the AND #$80 (or even the BMI/BPL instructions after the LDA value).

USING SHIFTS AND ROTATES TO PACK DATA.

     Having the capability to unpack data isn't particularly useful

if you cannot pack data as well. Packing data is a little more complicated than unpacking it, and can be accomplished in two

****

steps. First, the bits where the data is to be stored have to be forced to zero. This is accomplisied using the AND instruction. Next, the data to be placed in the desired field has to be shifted so that it is aligned properly. The data is then OR'ed into the zeroed field, resulting in a packed data record.

EXAMPLE:

PACKING THE SLOT # FIELD
FROM THE PREVIOUS EXAMPLE

```
        PHA             ;SAVE DATA TO BE PACKED
        LDA VALUE
        AND #%10001111  ;MASK OUT SLOT # FIELD
        STA VALUE       ;SAVE
        PLA             ;RESTORE ACC
        ASL             ;ALIGN FIELDS
        ASL
        ASL
        ASL
        ORA VALUE       ;PUT INTO VALUE
        STA VALUE
```

Additional packing techniques will be discussed as the need arises.

****

CHAPTER 10

MULTIPLE-PRECISION OPERATIONS

GENERAL.

Until now, all operations utilized have worked with only eight bits. For some operations this is fine. For others, being limited to eight bits is intolerable. Nevertheless, the 6502 is limited to working with eight bits at a time. In order to handle data types of larger sizes, such as 16-bit integers and 32-bit floating point numbers, we must break them up into several 8-bit operations. For example, a 16-bit addition is handled as two 8-bit additions.

MULTIPLE-PRECISION LOGICAL OPERATIONS.

The multiple-precision logical operations (AND, OR, and XOR) are the easiest to handle. Assuming you have two 16-bit

operands at locations A, A1, B, and B1, the logical AND of A and B is (A AND B), (A1 AND B1). This simply means that you take the data at location A and then AND it with the data at location B. The result is the low-order byte of the logical AND. Next, the data at location A1 is AND'ed with the data at location B1, which gives the high-order byte of the result.

EXAMPLE: 'AND' A WITH B AND STORE THE RESULT AT C.

```
LDA A
AND B
STA C
LDA A+$1
AND B+$1
STA C+$1
```


****

The ORA and EOR (XOR) instructions are handled in a similar manner.

EXAMPLES:

```
LDA A
ORA B
STA C
LDA A+$1
ORA B+$1
STA C+$1
```

```
LDA A
XOR B
STA C
LDA A+$1
XOR B+$1
STA C+$1
```


****

MULTIPLE-PRECISION SHIFTS AND ROTATES

Shifts and rotates are not extended beyond one byte in a manner similar to the simple logical instructions. Consider the ASL instruction. If you were to shift the low-order byte one position to the left, zero would end up in bit zero and the carry out of bit seven would end up in the carry flag. Now, if you were to perform an ASL on the high-order byte, the carry out of the previous bit seven (which would be in the carry) would not be shifted into bit zero, as should happen with a 16-bit ASL. Instead, zero would once again be shifted into bit zero and the carry out of the low-

order byte would be ignored.

This problem can be rectified by using a ROL instruction for the high-order byte, instead of the ASL instruction:

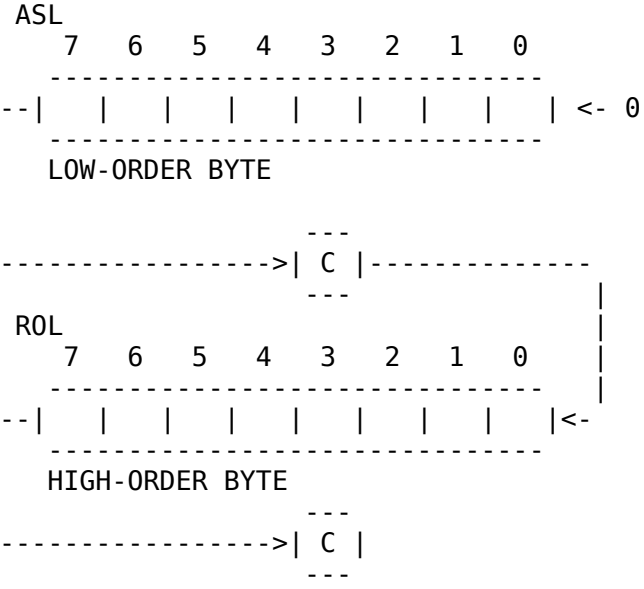
```
ASL LOBYTE
ROL HOBYTE
```

In this case, the carry out of the low-order byte ends up in the carry flag, and then the second instruction (ROL) shifts the carry flag into the low-order bit of the high-order byte (just as we expect). Naturally, the high-order bit ends up in the carry flag. A three-byte ASL can be manufactured by tacking another ROL instruction onto the end of this sequence:

```
ASL BYTE
ROL BYTE+$1
ROL BYTE+$2
```

Similarly, an "n"-byte ASL can be manufactured by tacking on additional ROL instructions to the sequence.

TWO-BYTE ASL

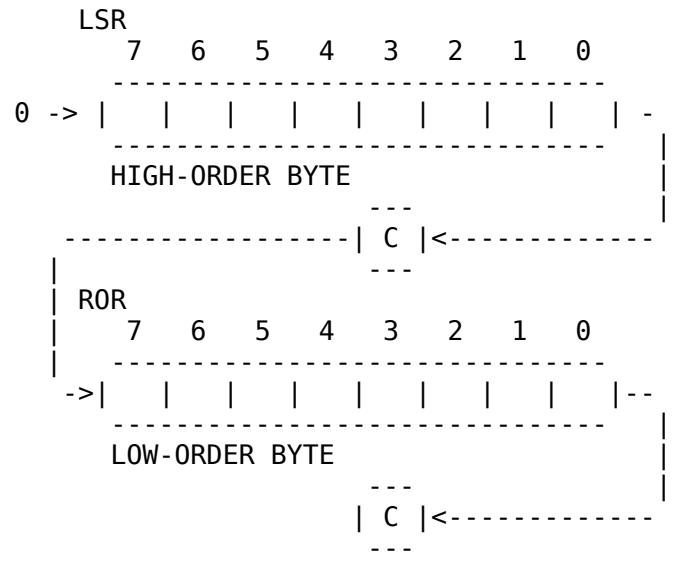


****

MULTIPLE-PRECISION LOGICAL SHIFT-RIGHT SEQUENCES.

The multiple-precision logical shift right operation is handled in a similar manner, except you must begin the process with the high- order byte. Remember, with a LSR instruction, zero gets

TWO-BYTE LSR



shifted into the high- order bit and then the low-order bit gets shifted into the carry flag. A 2-byte LSR would be coded as:

```
LSR BYTE+$1
ROR BYTE
```

Similarly, a three-byte LSR would be coded as:

```
LSR BYTE+$2
ROR BYTE+$1
ROR BYTE
```

N-byte LSR's can be simulated by using the LSR instruction on the high-order byte and then ROR all successive bytes.

MULTIPLE-PRECISION
ROTATE-LEFT SEQUENCES.

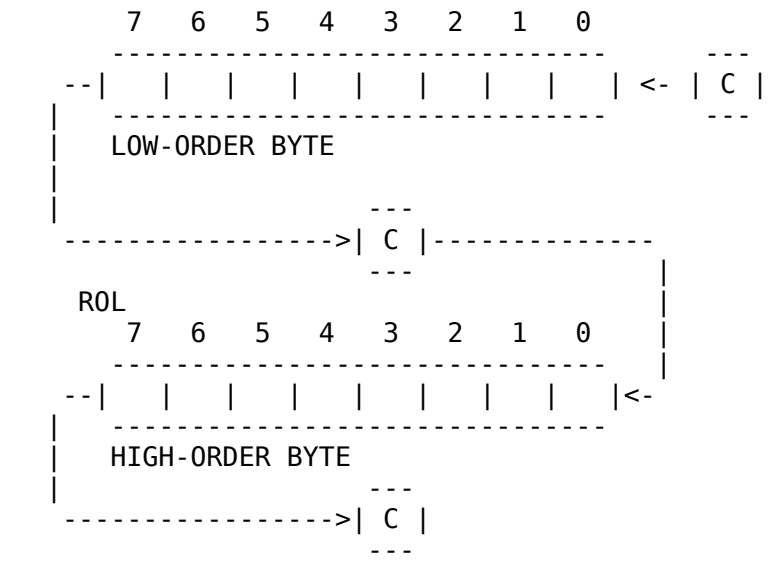
The multiple-precision rotate-left operation is easily handled. First, rotate the low-order byte, then the high-order byte (s). A 16-bit ROL could be written:

```
ROL BYTE
ROL BYTE+$1
```


****

TWO-BYTE ROL

ROL



The carry is shifted into bit zero, as we expect; bit seven is shifted into the carry, and then into bit eight because of the second ROL instruction. Finally, bit fifteen is shifted into the carry flag thereby performing a 16-bit ROL. A 3-byte ROL instruction is written:

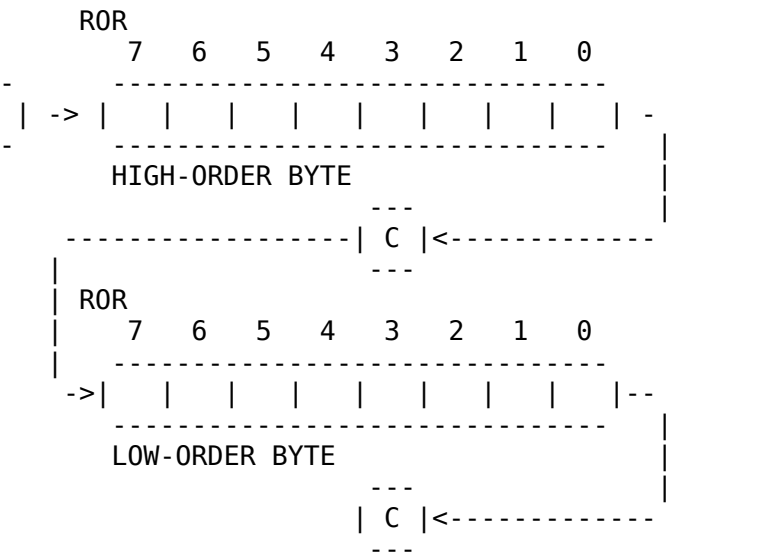
```
ROL BYTE
ROL BYTE+$1
ROL BYTE+$2
```

MULTIPLE-PRECISION ROTATE-RIGHT SEQUENCES.

As with the LSR instruction, multiple-precision ROR sequences must work on the high-order bytes first, and the low-order bytes last. A 16-bit ROR is written as:

```
ROR BYTE+$1
ROR BYTE
```

TWO-BYTE ROR



****

And a 3-byte ROR is written as:

```
ROR BYTE+$2
ROR BYTE+$1
ROR BYTE
```

MULTIPLE-PRECISION UNSIGNED ARITHMETIC.

Being limited to one byte when performing arithmetic is unthinkable. Most of the time we need to represent values greater than 255. If 16-bit arithmetic were available, we could represent values in the range 0-65,535; with 24 bits we could represent values in the range 0-16,777,215; by using four bytes (32 bits), numbers in excess of four billion could be represented. Multiple-precision arithmetic is handled in a manner similar to multiple-precision logical operations. You must perform the operations a byte at a time.

In order to perform extended precision arithmetic we must have some mechanism for "capturing" all the lost data when an arithmetic overflow (or underflow) occurs. First, let's determine how much data must be saved when an overflow occurs. Obviously, the largest number obtainable when adding two 8-bit numbers together is the value obtained by adding $FF and $FF. Since the result of this sum, $1FE, or 510 decimal, requires nine bits to represent it, we will need a 1-bit extension to perform extended arithmetic operations.

As you may recall, you can check the carry flag after an addition; it will be set if an overflow (into the "ninth" bit) occurred and reset otherwise. As such, we can use the carry flag as our ninth bit when performing arithmetic. Fine, but how is this going to allow us to perform arithmetic on 16, 24, or 32 bits? Remember our rules for unsigned addition? One rule states that the carry must be cleared before the addition takes place, because the carry flag gets added in as part of the operand. This means that if the carry flag is set, then you do not end up with the sum of the accumumator and the operand, but rather you get the sum of the accumulator and the operand plus one. Naturally, if the carry is clear, you get the sum of the accumulator and the operand plus

****

the carry flag (which is zero), thus giving you the true sum.

In this manner, the carry flag becomes the 'carry out' of an 8-bit addition and will contain the value which must be added to the addition of the high-order bytes in order to obtain the final

"adjusted" value. In reality, the 6502 adds numbers the same way you and I do, except it works with bytes instead of digits. The addition of the 16-bit quantities OP1 & OP2 with the sum being stored in RESULT, would be written as:

```
                CLC             ;ALWAYS BEFORE AN ADDITION
                LDA OP1
                ADC OP2
                STA RESULT


                LDA OP1+$1
                ADC OP2+$1
                STA RESULT+$1
```

Note that the carry flag is not cleared between the additions here! Remember, the carry flag contains vital information for successful multiple-precision addition. A 3-byte addition operation would be coded:

```
                CLC
                LDA OP1
                ADC OP2
                STA RESULT
                LDA OP1+$1
                ADC OP2+$1
                STA RESULT+$1
                LDA OP1+$2
                ADC OP2+$2
                STA RESULT+$2
```

and so forth for an 'n'-byte addition.

RULES FOR UNSIGNED N-BYTE ADDITION.

1) Do not confuse these rules with any of the other arithmetic rules.

2) Always clear the carry before performing the addition.

3) Add the first bytes together and store the results.

4) Add the second, third, ..., nth pairs of bytes together and store the results. Do not clear the carry flag before these additions.

5) After the nth addition, the carry flag will be set if an over-flow occurred, otherwise the carry flag will be cleared.

****

MULTIPLE-PRECISION UNSIGNED SUBTRACTION.

Multiple-precision subtraction is handled in much the same

way as multiple-precision addition. It is a logical extension of the single-precision subtraction, and as such, you must set the carry before the multiple precision subtraction takes place. Once this is accomplished, you subtract the low order bytes (storing the results) and then the high order bytes (also storing the results). Once the subtraction is complete, the absence of carry (i.e, carry=0) means an underflow occurred, and the presence of carry means thing went just fine.

EXAMPLE OF TWO-BYTE SUBTRACTION:

```
SEC             ;ALWAYS!
LDA OPRND1      ;GET L.O. BYTE OF OPERAND #1
SBC OPRND2      ;SUBTRACT L.O. BYTE OF OPERAND #2
STA RESULT      ;SAVE IN L.O. BYTE OF RESULT
LDA OPRND1+$1   ;GET H.O. BYTE OF OPERAND #1
SBC OPRND2+$1   ;SUBTRACT H.O. BYTE OF OPERAND #2
STA RESULT+$1   ;SAVE IN H.O. BYTE OF RESULT
BCC ERROR       ;TEST FOR OVERFLOW
```

To generalize to n bytes simply stick more SBC instructions on the end of the sequence. Remember not to set the carry flag between the multiple-precision subtraction sequences.

RULES FOR UNSIGNED MULTIPLE-PRECISION SUBTRACTION.

1) Do not confuse these rules with any of the other arithmetic rules.

2) Always set the carry flag before a subtraction.

3) Subtract the low-order bytes and store the results.

4) Subtract the second, third, ..., nth bytes and store the results.

5) After the nth bytes are subtracted, the carry will be clear if underflow occurred. If the carry flag is set, then no overflow occurred.

****

MULTIPLE-PRECISION SIGNED ARITHMETIC.

Multiple-precision signed arithmetic is handled in a manner identical to multiple-precision unsigned arithmetic. When pre-forming an addition, you must first clear the carry and then per-form a byte-by-byte addition. With subtraction, you first set the carry and then perform a byte-by-byte subtraction.

The difference lies in testing for overflow/underf low. As with the single-precision signed arithmetic, you must test the overflow flag instead of the carry flag (remember, carry detects a carry out of bit 7, overflow detects a carry out of bit 6). The overflow (V) flag

will be set if overflow or underflow occurred. The overflow flag will be reset if overflow did not occur. Note that the overflow flag is set if an underflow occurred during a subtraction. This is opposite in practice to the use of the carry flag in unsigned arithmetic.

MULTIPLE-PRECISION DECIMAL ARITHMETIC.

The 6502 can perform multiple-precision BCD arithmetic by first setting the decimal flag. After setting the decimal flag, follow the conventions for unsigned addition or subtraction. Don't forget to clear the decimal mode after the operation is complete. As with the single-precision decimal arithmetic, you cannot perform signed BCD arithmetic; only unsigned decimal arithmetic is allowed.

MULTIPLE-PRECISION INCREMENTS.

Sometimes it would be nice to be able to use the INC instruction to increment two 8-bit memory locations which are being treated as a 16-bit value. In conjunction with the indirect indexed by Y addressing mode, this capability is highly useful.

Unfortunately, the INC instruction does not affect the carry flag, so we can't use the carry flag to detect an 8-bit overflow. The INC instruction will alter only the 'Z' and 'N' flags. Fortunately, we will be able to use the 'Z' flag as though it were a carry flag. Why? Because the increment instruction will cause an overflow only when the prior result was $FF, and, when you increment $FF by one, you wind up with $0. Voila! The zero flag will be set whenever an overflow occurs while using the increment instruction. Thus,

****

the zero flag can be tested (using the BNE/BEQ instructions) to determine whether or not the high-order byte(s) should be incremented.

EXAMPLE OF 16-BIT INCREMENT:

```
          INC LOC
          BNE LBL
          INC LOC+$l
     LBL:
```

Likewise, a 3-byte increment could be synthesized as:

```
          INC LOC
          BNE LBL
          INC LOC+$1
          BNE LBL
          INC LOC+$2
     LBL:
```

Higher precision increments can be handled in a similar manner. Note that these increments are for unsigned quantities only.

Signed increments are possible, but it's simpler just to add one to the memory locations using the ADC instruction.

MULTIPLE-PRECISION DECREMENTS.

Just as useful as the multiple-precision increment is the multiple-precision decrement. The multiple-precision decrement is handled in a manner similar to the multiple-precision increment (what did you expect!). There is one problem, however: overflow occurs when the operand is decremented from $0 to $FF. Since the zero flag is not set on this transition, we must test the Z flag before the decrement instruction is used. Unfortunately, there is no safe way to test a memory location to see if it contains zero without explicitly loading that memory location into one of the registers. A 16-bit decrement must be handled as follows:

```
          LDA OPRND ;set Z flag if OPRND is zero
          BNE LBL
          DEC OPRND+$1
LBL       DEC OPRND
```


****

A 3-byte decrement could be handled as:

```
          LDA OPRND
          BNE LBL1
          LDA OPRND+$1
          BNE LBL2
          DEC OPRND+$2
LBL2      DEC OPRND+$1
LBL1      DEC OPRND
```

Beyond three bytes the SBC sequence becomes more economical than the DEC instruction sequence. As with the INC instruction, this multiple-precision DEC sequence is for unsigned values only. Signed decrements are much easier to perform using the SBC sequence.

MULTIPLE-PRECISION UNSIGNED COMPARISONS.

Once you know how to add and subtract multiple-precision values the next step is to learn how to compare them. Sadly, the generalization from one byte to n bytes we have enjoyed for arithmetic no longer applies to multiple-precision comparisons. For each type of comparison there is a completely different algorithm which must be followed. These, unfortunately, must be committed to memory as all of them are special cases. We'll start with the easy ones first.

TESTING A 16-BIT VALUE FOR ZERO.

To test an 8-bit variable against zero you simply load the accumulator with the contents of the variable and then test the

zero flag. When performing this same operation on a 16-bit value, you must load the accumulator with the low-order byte, then OR the accumulator with the high-order byte. If any of the 16 bits are set (which means the value is non-zero), the zero flag will be reset. If all of the sixteen bits are zero, the zero flag will be set (hence the value is zero).

EXAMPLE:

```
            LDA TSTZER
            ORA TSTZER+$1
            BEQ ISZERO
```


****

TESTING A 16-BIT VALUE TO SEE IF IT IS NEGATIVE.

To test a 16-bit value to see if it is negative is easy. The sign bit is bit 15 (the sixteenth bit) which is bit seven of the high-order byte. By loading the high-order byte into the accumulator, the 'N' flag will be set to reflect the sign of the entire 16-bit number. The BIT instruction could also be used to test the high-order byte and set the "N" flag accordingly.

TESTING FOR EQUALITY AND INEQUALITY

The test for equality is not quite as simple. This test must be handled in two parts. First, the low-order bytes are compared. If they are not equal, then a branch should be taken to some location further on in the code stream. If they are equal, you should drop down and compare the high-order bytes. If the high-order bytes are not equal a branch should be taken to the same location as in the previous branch. If the second test for inequality fails you know that the two operands are equal. The following code will jump to EQUALS if the two operands specified are equal, or it will jump to NOTEOL if the operands specified are not equal.

```
            LDA OPRND1
            CMP OPRND2
            BNE NE
            LDA OPRND1+$1
            CMP OPRND2+$1
            BNE NE
            JMP EQUALS
      NE    JMP NOTEQL
```

This sequence can be used to test for equality or inequality. By removing the JMP NOTEQL instruction, this becomes a 16-bit BEQ instruction, with the program dropping through to the next location (at location NE) should the operand prove to be not equal. The same test can be manufactured for NOT EQUALS by using the following code:

```
            LDA OPRND1
```

```
                CMP OPRND2
                BNE NE
                LDA OPRND1+$1
                CMP OPRND2+$1
                BEQ EQL
          NE    JMP NOTEQL
          EQL:
```


****

The extension to three or more bytes is simply a generalization of this technique.

EXAMPLE OF A THREE-BYTE TEST FOR NOT EQUALS:

```
                LDA OPRND1
                CMP OPRND2
                BNE NE
                LDA OPRND1+$1
                CMP OPRND2+$1
                BNE NE
                LDA OPRND1+$2
                CMP OPRND2+$2
                BEQ EQL
          NE    JMP NOTEQL
          EQL:
```

The inequalities (<, <=, >, & > =) turn out to be easier to program than the test for equals/not equals. If you will remember the discussion of the CMP instruction, it was mentioned that this instruction is really nothing more than a subtraction. The only difference is that the result is not kept around. Since a straight subtraction is performed (as opposed to a subtract with carry), a mutiple-precision CMP instruction is not technically possible. It can be simulated, however, by the combined use of the CMP and SBC instructions. The CMP instruction is used to compare the low-order bytes (this instruction is used so that the carry does not have to be explicitly set), and then the SBC instructon is used to compare successive bytes. After the last bytes are compared (using SBC) the BGE (or BCS) and BLT (or BCC) instructions may be used to test the result.

EXAMPLES:

```
                X>=Y
                ----
                LDA X
                CMP Y
                LDA X+1
                SBC Y+1
                BGE GE


                X<Y
                ----
```

```
LDA X
CMP Y
LDA X+1
SBC Y+1
BLT LT
```


****

To test for greater than, or less than or equal to, we could employ the methods described previously (in the chapter on single-byte compares). The only problem with this approach is the fact that too much code is required to perform two 16-bit tests. A better method, which also works for 8-bit comparisons but requires some knowledge of mathematics, is to alter the sequence by which the operands are compared. First, consider the test for X>= Y. It could be coded as:

```
X >= Y
------
LDA X
CMP Y
LDA X+$1
SBC Y+$1
BGE THERE
```

When you say that X is greater than or equal to Y, you are also stating that Y is less than or equal to X, so the above comparison is also testing to see if Y is less than or equal to X. To perform the comparison X<=Y, use the code:

```
X <= Y
------
LDA Y
CMP X
LDA Y+$1
SBC X+$1
BGE THERE
```

Which uses X as the value being compared to and the BGE branch. The test for greater than is exactly the same except you use the BLT branch instead of the BGE branch.

Comparisons of more than two bytes can be achieved by tacking on more SBC instructions for each succeeding byte.

Keep in mind that these comparisons are for unsigned values only (both in binary and decimal mode). For a description of how to compare signed values read on....

SIGNED COMPARISONS.

First, to test for equals, not equals, zero, or minus, you use the same tests as you would for an unsigned value. Testing for the inequalities '<=','<', '>= not as straight forward. Without a lengthy discussion of the 6502 hardware and two's complement

****

idiosyncrasies, you'll have to accept, on faith, that a comparison is greater than or equal if the XOR of the overflow and sign flag is one and the comparison is less than if the XOR of the overflow and sign flag is zero. One final note: the CMP instruction does not affect the overflow flag, so a full subtract with carry must be used. The following sequences test for the annotated condition:

```
                X >= Y
                ------
                SEC
                LDA X
                SBC Y
                LDA X+$1
                SBC Y+$1
                BVS LBL1
                BMI LT
        LBL2    JMP GTREQL
        LBL1    BPL LBL2
        LT:


                X <= Y
                ------
                SEC
                LDA Y
                SBC X
                LDA Y+$1
                SBC X+$1
                BVS LBL1
                BMI GT
        LBL2    JMP LESEQL
        LBL1    BPL LBL2
        GT:


                X < Y
                ------
                SEC
                LDA X
                SBC Y
                LDA X+$1
                SBC Y+$1
                BVS LBL1
                BPL GE
        LBL2    JMP LESS
        LBL1    BMI LBL2
        GE:
```


****

```
                X <= Y
                ------
                SEC
                LDA Y
                SBC X
                LDA Y+$1
                SBC X+$1
                BVS LBL1
                BPL
        LBL2    JMP LESEQL
        LBL1    BMI LBL2
        LT:
```

Of course, there are many variations on the comparisons and branches presented here. These examples are definitely not the most efficient or the only possible way of coding. By experimenting and 'adjusting,' you can probably come up with the combination you need.

****

CHAPTER 11

BASIC I/O

GENERAL.

The remaining chapters of this book will present programming examples with brief explanations. Should a particular piece of code be unclear, the reader is urged to review previous chapters in this book.

CHARACTER OUTPUT.

In BASIC all output is handled by the PRINT statement. In the not-so-wonderful world of assembly language there is no "PRINT" statement. In fact, input/output (I/O) is not provided for in the 6502 instruction set at all. Since the 6502 does not provide a scheme for I/O, the question naturally arises, "How does one output data, anyway?" To make a long story short, all I/O devices are treated as though they were memory locations. As such, input and output is performed using load and store instructions. The Apple video screen, in fact, resides in memory (but more on that later).

Since the 6502 is capable of working with only one byte at a time (i.e. one character), all I/O will have to be on a character-by-character basis. Typically, a user program will load the accumulator with a character and jump to a subroutine that outputs the character. Strings are output by repeatedly loading the accumulator and jumping to this subroutine. Integers and floating point numbers are output by converting each number to a string of characters and outputting the converted string.

The standard output device on the APPLE II computer is the video screen. The Apple video screen is an example of the so-

****

called "memory-mapped video display." A memory-mapped video display uses one byte of memory for each character position on the video screen. By storing data into the video memory you can put characters onto the Apple video screen. Luckily, you need not concern yourself with the actual addresses in memory used by the Apple's video screen. A subroutine within the Apple monitor has been provided which allows you to output a character (in the accumulator, of course) onto the video screen. Where does it output the character on the screen? Right after the previous character that was output. The subroutine is located at $FDF0 in the Apple monitor ROM and may be used as follows:

```
        LDA #"A"
        JSR $FDF0
        LDA #"B"
        JSR $FDF0
        LDA #"C"
        JSR $FDF0
        RTS
        END
```

This program outputs ABC (without the quotes) to the video screen and then returns to the monitor (or other calling routine).

****

Obviously, this form of output is very crude. It would be insane to expect anyone to output a string such as "I WON! CARE TO PLAY AGAIN?" to the video screen using this method.

A much better method of outputting chararacters requires the use of the 6502 index registers. To perform this type of I/O function the string is stored somewhere in memory (where it will not be executed as code) using LISA's 'STR' pseudo opcode. The index register is set equal to one (to skip over the length byte emitted by the STR pseudo opcode) and then all characters are output until the index register contains a value greater than the length of the string.

```
        LDX #$0
LOOP    INX
        LDA STRING,X
        JSR $FDF0
        CPX STRING
        BLT LOOP
        RTS
;
```

```
STRING  STR "I WON! CARE TO PLAY AGAIN?"
        END
```

In this example the X-register is initialized to zero, then incremented to one, before fetching the first character to output (remember, the length byte has to be skipped over). After the character is output to the video display, the X-register is compared with the length byte; if it is less than the length byte, another character is output.

The previous example has only one problem. What happens if the length of the string is zero? At least one character is output anyway. Sometimes it's possible for a string to have a length of zero, which means that the above procedure will not work in an entirely pleasant manner. In order to allow strings of length zero to be output (or actually NOT output in this case), the following code should be used:

```
        LDX #$0
LOOP    CPX STRING
        BGE EXIT
        LDA STRING+$1,X
        JSR $FDF0
        INX
        JMP LOOP
EXIT    RTS
STRING  STR "I WON! CARE TO PLAY AGAIN?"
        END
```


****

This routine attacks the problem in a slightly different manner. Rather than increment the X-register to compensate for the length byte, an offset is added to the address of the STRING when loading the A-register. By using this offset method, the X-index register will be equal to the length of the string MINUS one when the last character of the string is loaded into the accumulator. Whenever the X-register becomes equal to the length byte, the routine is finished. The BGE instruction is used rather than the BEQ instruction "just in case." True, under almost all circumstances, the BEQ instruction would have worked fine, but an ounce of prevention...

Although this method is considerably better than outputting the string a character at a time, it still leaves a lot to be desired. What happens if you wish to output several lines? In BASIC you would simply use additional print statements. In assembly language you have to repeat the above sequence over and over again. Not a nice thought.

To avoid this, a second method of outputting characters must be used. Rather than using a length byte to inform the print routine about the length of the string, a trailing end-of-text byte is used to terminate the string. Now, the print routine simply prints all characters until this end-of-text character is encountered, which

means control characters such as RETURN and LINE FEED may be imbedded directly in the string.

The ASCII character set does include a special 'ETX' (for 'end-of-text') character ($83, or control-C), but sometimes you may need to output this character to some device. As a result, it is better to select a character code that will almost never be output to a peripheral device. Such a character is the inverted at sign ('@') which has a character code (on the APPLE II compter) of $00. The choice of the character code is arbitrary, but it is very easy to test for zero, so that's what will be used in the following examples:

```
        LDX #$0         ;INIT POINTER TO CHARACTERS
LOOP    LDA STRING,X    ;GET NEXT CHARACTER
        BEQ EXIT        ;IF ZERO, QUIT
        JSR $FDF0       ;OTHERWISE OUTPUT
        INX
        JMP LOOP
EXIT    RTS
STRING  ASC "I WON!  CARE TO PLAY AGAIN?"
        BYT $0
        END
```


****

Note that the ASC pseudo opcode was used rather than the STR pseudo opcode. Remember, the STR pseudo opcode outputs a length byte before it outputs the string. This feature is not desirable here.

Outputting several lines at once is simply a matter of imbedding carriage returns within the text:

```
        LDX #$0
LOOP    LDA STRING,X
        BEQ EXIT
        JSR $FDF0
        INX
        JMP LOOP
;
EXIT    RTS
STRING  ASC "I WON!  CARE TO PLAY AGAIN?"
        BYT $8D
        ASC "(Y/N):"
        BYT $0
        END
```

The previous example outputs two separate lines before termination. Any number of lines (almost!) may be output by embedding a return character ($8D) within the text string.

The procedures thus far presenting this point suffer from one drawback. Since the X-register is used to access elements of the strings being output, you are limited to a maximum of 255

characters in your strings. Although this may seem like a lot for just one string, remember that when outputting several lines the 255 character limitation (i.e. six lines) becomes critical. In fact, the last example had a small "bug" in it. Should you try to output more than 255 characters, the X-register will wrap around to zero and then the routine will begin printing the string from the beginning again. The end result is that you will wind up in a infinite loop with a lot of redundant material ending up on the screen. To prevent the infinite loop from occurring, you should use the following code:

```
        LDX #$0
LOOP    LDA STRING,X
        BEQ EXIT
        JSR $FDF0
        INX
        BNE LOOP
;
EXIT    RTS
STRING  ASC "> 255 CHARACTERS HERE"
        BYT $0
        END
```


****

In this example the JMP LOOP instruction was replaced with the BNE LOOP instruction. Should the X-register overflow and wrap around to zero, the routine will exit rather than continuing on its merry way. It should be noted that this "fix" does not allow you to output more than 255 characters, it simply terminates output once 255 characters have been output. As a result, part of your string may not be displayed, but then your program will not cause an infinite amount of 'garbage' to be written to the screen either.

To output strings whose length is greater than 255, a 16-bit pointer must be used. This means that the indirect, indexed by Y addressing mode must be used. The following routine allows you to output strings of any length (less than 65,535 characters, of course):

```
        LDA #STRING     ;MOVE ADDRESS OF STRING
        STA $0          ;INTO LOCATIONS $0 AND
        LDA /STRING     ;$1
        STA $1
        LDY # 0         ;INIT Y REGISTER
LOOP    LDA ($0),Y
        BEQ EXIT
        JSR $FDF0
        INY
        BNE LOOP        ;IF NO OVERFLOW, KEEP IT UP
        INC $1          ;INCREMENT BEYOND B BITS
        BNE LOOP
EXIT    RTS
STRING  ASC "STRING OF ANY LENGTH"
```

```
                HEX 00
                END
```

This routine has a couple of interesting features. First, note that the Y-register, rather than location $0 was incremented. This saves a byte of code and lets the routine run a little faster. Also note that locations $0 and $1 had to be set up before the routine was executed. Although considerably more code was required to write this outine, in the end it pays off because the routine can be turned into a generalized subroutine. Consider:

```
          PRTSTR  STA $0
                  STY $1
                  LDY #$0
          LOOP    LDA ($0),Y
                  BEQ EXIT
                  JSR $FDF0
                  INY
                  BNE LOOP
                  INC $1
                  BNE LOOP
          ;
          EXIT    RTS
```


****

With this subroutine all you need to do is load the accumulator and Y-register with the address of the string to be output (low-order byte into ACC, high-order byte into Y-register) and then JSR PRTSTR.

Example:

```
                LDA #STRING
                LDY /STRING
                JSR PRTSTR
                RTS
          STRING  ASC "STRING OF ANY LENGTH"
                  BYT $0
                  END
```

Now only three lines of code (plus the string) are required to output a string of characters. That's quite a bit better than the seven to ten lines required by the other methods. Nevertheless, this method has two drawbacks. First, three lines are still two lines more than one. Second, this method requires that data be passed to the subroutine in the accumulator and Y-register. Typically, one likes to avoid the use of the registers for parameter passing as much as possible (since the registers are much more useful for indexing and counter purposes).

The final method presented here is based on the previous example. That is, the address of a string is passed to a subroutine which outputs all data from that address forward until a zero is encountered. The approach used by this method is different

because the 6502 stack will be used to pass the address to the routine. Consider the following assembly language sequence:

```
        JSR PRINT
        ASC "HELLO THERE"
        HEX 00
        RTS
        END
```

This section of code would jump to the 'PRINT' subroutine and then return to the next instruction- which is the character 'H.' Wait a minute, this won't work as planned! The string has to be placed where it won't be executed as code. Or does it? As you may recall, when a subroutine is called, the return address minus one is pushed onto the stack. If the address is popped off the stack and incremented by one, the address will point to the "H" in "HELLO." By using this pointer, it is possible to output all the data until a $00 is encountered. When the zero is encountered, the

****

next byte will (hopefully) contain a valid instruction so the address can be pushed back on the stack and a normal RTS instruction can be executed. Upon return, the 6502 will continue program execution at the point just beyond the $00. Another alternative is to increment the address by one (upon encountering $0) and then jump indirect through that address. This simulates the RTS instruction with a small space savings. The final PRINT subroutine might be:

```
PRINT   STA ASAVE       ;SAVE ACC
        STY YSAVE       ;SAVE Y REG
        PLA
        STA ZPAGE
        PLA
        STA ZPAGE+$1
        JSR INCZ
        LDY #$0
PLOOP   LDA (ZPAGE),Y
        BEQ EXIT
        JSR $FDF0
        JSR INCZ
        JMP PLOOP
;
EXIT    JSR INCZ
        LDA ASAVE
        LDY YSAVE
        JMP (ZPAGE)
;
INCZ    INC ZPAGE
        BNE INCZ0
        INC ZPAGE+$1
INCZ0   RTS
        END
```

This routine is called with the string immediately following the JSR instruction, terminated of course by a hex 00.

EXAMPLES:

```
            JSR PRINT
            ASC "I WON!  CARE TO PLAY AGAIN?"
            BYT $8D
            ASC "(Y/N):"
            BYT $0
             .
             .
             .
            JSR PRINT
            ASC "HELLO THERE, HOW ARE YOU!"
            BYT $0
            JSR PRINT
            BYT $8D
            ASC "I AM A SMART COMPUTER!"
            BYT $0
             .
             .
            ETC....
```


****

STANDARD OUTPUT AND PERIPHERAL DEVICES.

Until now all output was assumed to be directed to the Apple's video display. To output a character to the video display you simply load a character into the accumulator and JSR to $FDF0. Since it is not a good idea to use absolute addresses within your assembly language programs, you should define a symbolic label (using the EQU pseudo opcode) that is equal to $FDF0. A good label to use is COUT1, because that's the label used in the Apple monitor listings, and if someone else reads your code, they will probably associate the video output routine with the COUT1 label.

Sometime you will want to output data to some peripheral device other than the video display. Output is handled in a manner identical (in most cases) to the video display. That is, you load the accumulator with the character you wish to output and JSR to the routine that handles the output for you. The address of this routine is typically $Cn00 where n is the slot number of the peripheral deviceland is in the range of 0 thru 7. Note that this scheme only works for the so-called, "intelligent" peripherals which have an on-board ROM. "Dumb" peripherals, such as those purchased from Electronic Systems and Microproducts, use a totally different scheme for "driver software" storage. You should also be aware that this scheme does not work for the Disk II or the Tape II devices as they use the ROM area for a bootstrap loader. Let's assume you have a printer interface in slot #1. All you have to do to output a character to the printer is load the accumulator with

that character and JSR $C100.

But it is even easier to use Apple's "Standard Output." Rather than jumping to the subroutine at $Cn00, simply JSR to location $FDED (label = COUT) in the Apple monitor. This causes the output to be directed to the currently active peripheral. Peripherals are made active by simulating the PR#n and IN#n commands from assembly language. To simulate a PR#n command, first load the accumulator with the slot number, and then JSR to location $FE95 in the Apple monitor (routine 'OUTPORT'). To simulate an IN#n command, load the accumulator with the slot number and JSR to location $FE8B (routine 'INPORT'). To reset the I/O vectors to the video screen or keyboard (the equiv-

****

alent of a PR#0 or IN#0 command), just load the accumulator with zero before jumping to the desired routines. Alternately, you may simulate a PR#0 command by JSR'ing to $FE93 you may simulate an IN#0 command by JSR'ing to $FE89.

When you execute the routine at location $FDED, the first instruction to be executed is a JMP ($36). Normally, locations $36 and $37 contain $F0 and $FD, which means that whenever you JSR $FDED (or JSR COUT), the COUT1 routine gets executed. If a PR#n command (or equivalent) is executed prior to the output of a character, $00 will be stuffed into location $36 and $Cn will be stuffed into location $37. Now the character is routed to the routine stored at location $Cn00... automatically. Naturally you can 'poke' the address into locations $36 and yourself:

```
-SIMULATION OF A PR#3
        LDA #$00
        STA $36
        LDA #$C3
        STA $37
```


****

```
-SIMULATION OF A PR#0
        LDA #$FDF0
        STA $36
        LDA /$FDF0
        STA $37

-CAUSE OUTPUT TO BE ROUTED TO USER ROUTINE
 AT LOCATION $300
        LDA #$300
        STA $36
        LDA /$300
        STA $37
```

The last example is important because it demonstrates how one activates a user-defined output routine. An example of such a user routine is:

```
                ORG $300
                LDA #DBLVSN
                STA $36
                LDA /DBLVSN
                STA $37
                RTS
          ;
          DBLVSN  JSR $FDF0
                  JMP $FDF0
                  END
```

Assemble this routine, then execute the Apple monitor 300G command and watch what happens. The fact that the standard output can be "directed" is one of the more powerful features of the Apple monitor, and is the primary reason that the Apple II is easily expandable.

CHARACTER INPUT.

Just as with character output, character input is handled a character at a time. The Apple II keyboard appears as two memory locations to the user program. Location $C000 in memory will contain the ASCII code of the last key pressed. If bit seven is set (i.e., the high-order bit is one), a valid key has been pressed. If bit seven is clear, then a key has not yet been pressed and the data at location $C000 is invalid. Accessing location $C010 clears bit seven to allow additional keys to be pressed and acknowledged.

****

Therefore, to read a key from the Apple keyboard you would perform the following steps:

1) Read location $C000 and loop until bit seven is set.

2) Load the accumulator from location $C000 to enter the keycode into the accumulator.

3) Store the accumulator into location $C010 to clear the keyboard strobe, which makes location $C000 ready for the next input.

A suitable program for accomplishing this task might be:

```
          KEYIN   LDA $0000
                  BPL KEYIN
                  STA $C010
                  RTS
```

You will notice that any key read in this manner will not be

'echoed' onto the Apple screen. To perform this function (that of an 'electronic typewriter'), use the following code:

```
TPWRTR  JSR KEYIN
        JSR COUT
        JMP TPWRTR
KEYIN   LDA $C000
        BPL KEYIN
        STA $C010
        RTS
COUT    EQU $FDF0
        END
```


****

To exit this program, depress the RESET key on the Apple II keyboard.

When using the Apple keyboard and the video display, the Apple monitor provides a very handy character input subroutine. It is located at $FD0C and it sets the current cursor location to the flashing mode. Upon keyboard entry the flashing cursor is replaced with the data originally under the cursor. A better 'electronic typewriter' might be:

```
LOOP    JSR RDKEY
        JSR COUT
        JMP LOOP
RDKEY   EQU $FD0C
COUT    EQU $FDED
        END
```

The routine at location $FD0C does not 'echo' the character back to the display, hence the JSR COUT.

Just as the routine at location $FDED handles I/O parameters through the standard output (allowing you to output data to several different peripherals), the routine at location $FD0C gets its input from the 'standard input.' By JSR'ing through location $FD0C it is possible to read data from peripherals such as the Disk II, Mountain Computer's Apple Clock, external terminals, etc.

There are two differences between the way standard output is handled and the way the standard input is handled. First, locations $38 and $39 are used to hold the address of the routine from which the input is coming. Second, the input data is returned in the accumulator.

An IN# command can be simulated by loading the accumulator with the desired slot number and JSR'ing to the routine at location $FE8B. An IN#0 command can be simulated by JSR'ing to the routine at location $FE89. Input must be handled a little more cautiously than output; the reader is advised to study the input routines in the Apple monitor ROM's from location $FD0C to $FD2E.

INPUTTING A LINE OF CHARACTERS.

Obviously, to input a line of characters- all one needs to do is continually read a single character and store the data in successive memory locations until a carriage return (ASCII CODE

****

= $8D) is received. Although, on the surface the routine seems trival to write, there are several little "gotcha's" which sneak up on you. For instance, when you press the backspace key, the ASCII code $88 is returned. If you print a backspace, the cursor will indeed back up; however, typically you do not want to enter the backspace character into the line of text, but rather you wish to delete the previously entered character. Also, the right arrow key (which is the same as control-U) will not copy the data under the cursor, but rather return the ASCII code $95. Furthermore, the ESC editing functions are not supported, unless of course, you write the handler routines yourself. As you can see, the trival routine turns out to be not-quite-so-trival!

Luckily, a line input routine has already been written for us. The address of this routine is $FD67 and it is called, "GETLNZ." When called, it outputs a carriage return, prints a 'prompt' character (more on that later), and then reads a line of text from the current input device. Whatever character resides in location $33 is used as a prompt character, so, if you wish to use a new and unique prompt (perhaps ":" or "-" or "="), simply store the character at location $33 before calling GETLNZ.

GETLNZ has two alternate entry points. GETLN (at location $FD6A) does not output a carriage return before outputting the prompt character. GETLN1 (at location $FD6F) outputs neither the prompt character nor the carriage return. Both of these entry points will be useful on occasion.

So where does the text end up when you call GETLNZ, GETLN, or GETLN1? All text is stored sequentially in memory beginning at location $200. A maximum of 256 characters are allowed to be entered without having the line rejected. Because of this, page two should never be used for program code or data. Upon return from the GETLNZ, GETLN, or GETLN1 routine the X-register contains the number of characters actually input (not including the carriage return). The GETLN routines echo all input so the user can see what's going on. Furthermore, all Apple screen editing features are supported. Just exactly how one would use the line input routines will be discussed in following chapters.

****

CHAPTER 12

NUMERIC I/O

GENERAL.

Inputting and outputting characters is fine for many purposes. However, sometimes the need arises to input or output numeric data. This chapter will cover four types of numeric I/O:

1) Hexadecimal I/O

2) Byte/numeric I/O

3) Integer (16-bit or more) I/O

4) Signed integer (16-bit two's complement) I/O

HEXADECIMAL OUTPUT.

The easiest type of data to output numerically is a hexadecimal number. Although we could write a routine to do this (and in fact one is presented for your education), there is no need. The Apple monitor provides us with a very good routine. The address of the routine is $FDDA and this routine prints the contents of the accumulator as two hex digits. The contents of the accumulator are destroyed, but no other registers are affected. The Apple monitor name for this routine is PRBYTE, but HEXOUT is usually used in user programs. It should be noted that the hexadecimal output and BCD output routines are one and the same, so if you wish to output a BCD number, use the routine at location $FDDA.

To output a number (BCD or HEX) that is greater than one byte, load the accumulator with the most-significant byte and JSR HEXOUT. Repeat this for all the other bytes (the next most-significant byte down to the least-significant byte) until the entire number is output.

****

The PRBYTE routine in the monitor is reproduced here (with some minor changes) for illustrative purposes:

```
PRBYTE  PHA             ;SAVE ACC FOR USE LATER ON
        LSR             ;SHIFT H.O. NIBBLE
        LSR             ;DOWN TO THE L.O. NIBBLE
        LSR             ;CLEARING THE H.O. NIBBLE
        LSR
        JSR PRHEXZ      ;PRINT L.O. NIBBLE AS A DIGIT
        PLA             ;GET ORIGINAL VALUE BACK
PRHEX   AND #$F         ;MASK H.O. NIBBLE
PRHEXZ  ORA #$B0        ;CONVERT TO ASCII
        CMP #$BA        ;IF IT IS A DIGIT FINE, OTHER-
        BLT PRTIT       ;WISE IT MUST BE CONVERTED TO A
        ADC #$6         ;LETTER IN THE RANGE A-F
```

```
PRTIT   JMP COUT
COUT    EQU $FDED
        END
```

The CMP #$BA is required because the letter A does not immediately follow the digit 9 in the ASCII character set. Since BLT is the same as BCC, the processor is guaranteed to have the carry flag set if the ADC #$6 is encountered. In effect, we are adding seven to the contents of the accumulator. $BA plus $7 is $C1 which is the ASCII code for the letter A, exactly what we want.

****

OUTPUTTING BYTE DATA AS A DECIMAL VALUE.

Hex numbers are fine for computer type people, but when trying to present information to others, the decimal number system should be used. The monitor does not contain a facility for outputting decimal numbers (except BCD) so we will have to write one ourselves. In this section, a method for outputting a single byte as an unsigned integer in the range 0 to 255 will be explored.

The algorithm for outputting a byte as a decimal integer is actually quite simple. The binary number is compared with 100; if greater or equal, then 100 is continually subtracted until the desired value is less than zero. After each subtraction, a memory location is incremented so that when the number is less than one hundred, the hundreds digit is saved in this memory location. This data may then be output to the video screen. This process is repeated, only 10 is subtracted this time instead of 100. Once the number is less than 10, the corresponding digit counter is output. Since the remaining number is less than 10, its output is accomplished rather easily.

In addition to these steps, a flag must be used to suppress the output of leading zeros. This is accomplished by initializing a memory location to a positive value, which is set negative (i.e.,

****

high-order bit = 1 Before outputting a digit, this flag is checked to make sure that a zero (should the digit be a zero) can be output. The program is written as follows:

```
PRTBYT  PHA             ;SAVE REGISTERS
        TXA
        PHA
;
        LDX #$2         ;MAX OF 3 DIGITS (0-255)
        STX LEAD0       ;INIT LEAD0 TO NON-NEG VALUE
PRTB1   LDA #"0"        ;INITIALIZE DIGIT COUNTER
```

```
        STA DIGIT
;
PRTB2   SEC
        LDA VALUE       ;GET VALUE TO BE OUTPUT
        SBC TBL10,X     ;COMPARE WITH POWERS OF 10
        BLT PRTB3       ;IF LESS THAN, OUTPUT DIGIT
;
        STA VALUE       ;DECREMENT VALUE
        INC DIGIT       ;INCREMENT DIGIT COUOTER
        JMP PRTB2       ;AND TRY AGAIN
;
PRTB3   LDA DIGIT       ;GET CHARACTER TO OUTPUT
        CPX #$0         ;CHECK TO SEE IF THE LAST DIGIT
        BEQ PRTB5       ;IS BEING OUTPUT
        CMP #"0"        ;TEST FOR LEADING ZEROS
        BEQ PRTB4
        STA LEAD0       ;FORCE LEAD0 NEG IF NON-ZERO
;
PRTB4   BIT LEAD0       ;IF ALL LEADING ZEROS, DON'T
        BPL PRTB6       ;OUTPUT THIS ONE
PRTB5   JSR COUT        ;OUTPUT DIGIT
PRTB6   DEX             ;MOVE TO NEXT DIGIT
        BPL PRTB1       ;QUIT IF THREE DIGITS HAVE
        PLA             ;BEEN HANDLED
        TAX
        PLA
        RTS
TBL10   BYT !1
        BYT !10
        BYT !100
;
COUT    EQU $FDED
LEAD0   EPZ $0
DIGIT   EPZ LEAD0+$1
VALUE   EPZ DIGIT+$1
        END
```

To use this routine, load into the location VALUE the byte to be printed; then JSR PRTBYT. The decimal number corresponding to the byte stored in location VALUE will be output to the screen (or other output device).

****

OUTPUTTING 16-BIT UNSIGNED INTEGERS.

Obviously we have to work with quantities which cannot be contained in only eight bits. With two bytes, unsigned values in the 0-65,535 range can be represented. Output of integers in this range is accomplished quite easily by extending the previous routine to test for values in the 1000 to 10,000 range. The final routine appears similar to the following list:

```
PRTINT  PHA             ;SAVE REGISTERS
        TXA
```

```
        PHA
        LDX #$4         ;OUTPUT UP TO 5 DIGITS
        STX LEAD0       ;INIT LEAD0 TO NON-NEG
;
PRTI1   LDA #"0"        ;INIT DIGIT COUNTER
        STA DIGIT
;
PRTI2   SEC             ;BEGIN SUBTRACTION PROCESS
        LDA VALUE
        SBC T10L,X      ;SUBTRACT LOW ORDER BYTE
        PHA             ;AND SAVE
        LDA VALUE+$1    ;GET H.O BYTE
        SBC T10H,X      ;AND SUBTRACT H.O TBL OF 10
        BLT PRTI3       ;IF LESS THAN, BRANCH
;
        STA VALUE+$1    ;IF NOT LESS THAN, SAVE IN
        PLA             ;VALUE
        STA VALUE
        INC DIGIT       ;INCREMENT DIGIT COUNTER
        JMP PRTI2
;
;
PRTI3   PLA             ;FIX THE STACK
        LDA DIGIT       ;GET CHARACTER TO OUUPUT
        CPX #$0         ;LAST DIGIT TO OUTPUT?
        BEQ PRTI5       ;IF SO, OUTPUT REGARDLESS
        CMP #"0"        ;A ZERO?
        BEQ PRTI4       ;IF SO, SEE IF A LEADING ZERO
        STA LEAD0       ;FORCE LEAD0 TO NEG.
;
PRTI4   BIT LEAD0       ;SEE IF NON-ZERO VALUES OUTPUT
        BPL PRTI6       ;YET.
PRTI5   JSR COUT
PRTI6   DEX             ;THROUGH YET?
        BPL PRTI1
        PLA
        TAX
        PLA
        RTS
;
T10L    BYT !1
        BYT !10
        BYT !100
        BYT !1000
        BYT !10000
;
```


****

```
T10H    HBY !1
        HBY !10
        HBY !100
        HBY !1000
        HBY !10000
;
```

```
COUT    EQU $FDED
LEAD0   EPZ $0
DIGIT   EPZ LEAD0+S1
VALUE   EPZ DIGIT+$1
        END
```

To use this routine, load VALUE and VALUE+$1 with the binary integer you wish output. Then JSR PRTINT and let the routine do the rest of the work for you. This routine is fairly general and can be expanded to output numbers greater than two bytes in length. All that is required is one additional subtraction between the PRTI2 and PRTI3 labels to handle the most-significant byte, and the inclusion of another table of bytes giving the most-significant byte values for the data you wish output. Finally, the LDX #$4 instruction has to be changed to reflect the maximum number of digits to be output, MINUS ONE. Beyond that, this routine can be used to output unsigned integers of any size.

OUTPUTTING SIGNED 16-BIT INTEGERS.

Outputting a two's complement signed value turns out to be quite simple. Check the high-order bit of the number. If it is clear, jump to the PRTINT routine just described. If the high-order bit is set then you must output a "-", take the two's complement of the number; then jump to the PRTINT routine. The code is written as follows:

```
PRTSGN  BIT VALUE+$1    ;TEST SIGN BIT
        BPL PRTINT      ;IF POSITIVE, GO TO PRTINT
        PHA             ;SAVE ACC
        LDA #"-"        ;OUTPUT A
        JSR COUT
        SEC             ;TAKE TWO'S COMPLIMENT OF
        LDA #$0         ;VALUE.
        SBC VALUE
        STA VALUE
        LDA #$0
        SBC VALUE+$1
        STA VALUE+$1
        PLA
PRTINT  ---             ;INSERT PRTINT ROUTINE HERE
```


****

AN EASY METHOD OF OUTPUTTING INTEGERS.

Although these decimal printing routines are fairly easy to use, they do require a substantial amount of setup code. VALUE and VALUE+$1 must be loaded with the integer to be output before the JSR is executed. This setup code requires 8 to 10 bytes and four lines of code. The following routine (that works in a manner similar to the print routine developed in the last chapter) allows you to specify the address of the integer which you wish output immediately after the JSR statement. This only requires one extra line and two bytes of code, which makes it almost as

easy to use as the PRINT I command. The routine works in the following manner:

1) The return address is popped off the stack and stored in VALUE.

2) VALUE is incremented by two and pushed back onto the stack. This fixes the return address so that the 6502 will return to the point immediately following the 2-byte address.

3) VALUE is decremented by one. It now points to the 2-byte address that follows the JSR instruction.

4) The two bytes pointed to by (VALUE) and (VALUE)+$1 (which is the address of the integer we wish to print) are loaded into VALUE.

5) The data bits pointed to by VALUE (i.e., the data to be output) are then loaded into VALUE.

6) PRTINT or PRTSGN is called to output the number. The code used to achieve all of this is:

```
          STA ASAVE       ;SAVE ACC
          STY YSAVE       ;SAVE Y REGISTER
          PLA             ;GET RETURN ADDRESS
          STA VALUE
          PLA
          STA VALUE+$1
          JSR INCV        ;INCREMENT VALUE BY TWO
          JSR INCV
          LDA VALUE+$1    ;PUSH RETURN ADDRESS
          PHA
          LDA VALUE
          PHA
          JSR DECV        ;MAKE VALUE POINT TO DATA
          JSR LVIV        ;GET DATA POINTED AT BY
                          ;DATA FOLLOWING JSR
```


****

```
          LDA ASAVE       ;RESTORE ACC
          LDY YSAVE       ;RESTORE Y REGISTER
          JMP PRTINT      ;CHANGE TO PRTSGN IF SIGNED
;                         ;OUTPUT IS DESIRED
LVIV:
          JSR LAIA
          JSR LAIA
LAIA      LDY #$0
          LDA (VALUE),Y   ;GET L.O. BYTE
          PHA
          INY
          LDA (VALUE),Y   ;GET H.O. BYTE
          STA VALUE+$1    ;AND REPLACE VALUE
```

```
          PLA
          STA VALUE
          RTS
ASAVE     EPZ $4          ;ACC SAVE AREA
YSAVE     EPZ ASAVE+$1    ;Y REG!SAVE AREA
          END
```

NUMERIC INPUT.

HEXADECIMAL and BCD.

Numeric input is just as important as numeric output. In this section we will explore the various methods of inputting numeric data.

BCD input is by far the easiest to accomplish. The only operations required here are some masking and shifting operations. BCD input uses the following algorithm:

1) Initialize some location (VALUE) to zero. In these examples a 2-byte input will be used, but the generalization to more (or fewer) bytes should be apparent.

2) All input will be assumed to be stored in page two (so that it is compatible with the GETLN routines) and the Y-register will point to the first character to be input.

3) The end of the BCD string will be considered to be the first non-decimal digit encountered.

4) Each digit is read in and the high-order nibble (which always $B) is shifted out with four successive ASL instructions. The low-order nibble of the original number is left in the high-order nibble of the accumulator.

5) This value is shifted into VALUE using the ROL instruction. First, some routines which will prove to be useful:


```
*****************************************************************

          ; TSTDEC: TEST THE CHARACTER IN THE ACCUMULATOR.
          ; IF A VALID DECIMAL DIGIT, THEN THIS ROUTINE RETURNS
          ; WITH THE CARRY FLAG SET.  IF THE CHARACTER IN THE
          ; ACCUMULATOR IS NOT A DECIMAL DIGIT, THEN THIS ROUTINE
          ; RETURNS WITH THE CARRY FLAG CLEAR.

          TSTDEC  CMP #"0"        ;BRACKET TEST FOR A DIGIT
                  BLT NOTDEC
          CMP #"9"+$1             ;IS IT GREATER THAN NINE?
                  BGE NOTDEC
                  SEC             ;IT IS A DECIMAL DIGIT
                  RTS             ;SO SET THE CARRY AND RETURN
          ;
          ;
          NOTDEC  CLC             ;NON-DIGIT WAS FOUND
```

```
        RTS

; SHFTIN: SHIFTS THE L.O. NIBBLE OF THE ACCUMULATOR
; INTO "VALUE".

SHFTIN  ASL             ;MOVE LOW ORDER NIBBLE
        ASL             ;INTO HIGH ORDER NIBBLE
        ASL             ;OF THE ACCUMULATOR
        ASL
;
        JSR SHFT2
SHFT2   JSR SHFT1
SHFT1   ASL             ;SHIFT ACC INTO VALUE
        ROL VALUE       ;NOTE: FOUR SHIFTS ARE
        ROL VALUE+$1    ;PERFORMED HERE!
        RTS
;
```

The code for SHETIN should be studied carefully. You should manually trace the code beginning at the JSR SHFT2 instruction and convince yourself that four shifts are performed by this code sequence. With these two routines, BOD input becomes very easy. The BCD input routine is coded as follows:

```
;BCDIN: CONVERTS ASCII STRING IN PAGE TWO (POINTED
;AT BY THE Y REGISTER) INTO A BCD VALUE. ALL DIGITS
;ARE CONVERTED UNTIL A NON-DIGIT IS ENCOUNTERED.
;
BCDIN:
        LDA #$0         ;INITIALIZE VALUE TO ZERO
        STA VALUE
        STA VALUE+$1
;
BCDLP   LDA PAG2,Y      ;GET NEXT CHARACTER
        JSR TSTDEC      ;IS IT A DECIMAL DIGIT?
        BCC BCDONE      ;IF NOT, QUIT
        JSR SHFTIN      ;IF IT IS, SHIFT INTO VALUE
        INY             ;INDEX TO NEXT CHARACTER
        BNE BCDLP       ;AND REPEAT
```


****

```
;
BCDONE  RTS

PAG2    EQU $200        ;GETLN INPUT BUFFER
VALUE   EPZ $2
        END
```

The following short program demonstrates the use of the BCD input routine from within a program.

```
BCDTST  JSR PRINT       ;PRINT ROUTINE (SEE LAST CH)
        ASC "ENTER A NUMBER:"
        HEX 00
```

```
        ;
                JSR GETLN1      ;GET A LINE OF TEXT (NO PROMPT)
                LDY #0
                JSR BCDIN
                JSR PRINT
                ASC "YOU ENTERED:"
                HEX 00
        ;
                LDA LDA VALUE+$1
                JSR HEXOUT
                LDA VALUE
                JSR HEXOUT
                RTS
        ;
        GETLN1  EQU $FD6F
        HEXOUT  EQU $FDDA
                END
```

Inputting a hexadecimal number is handled in an identical manner, except "TSTDEC" is replaced by "TSTHEX" which tests the character in the accumulator to see if it is a valid hexadecimal digit. In addition to testing for a valid hex digit, TSTHEX also converts the letters "A" to "F" to the hex values $BA-$BF so that the 16 hexadecimal values are contiguous.

```
                CMP #"0"        ;BRACKET TEST FOR DECIMAL D
                BLT NOTHEX
                CMP #9"+$1
                BLT ISHEX
                CMP #"A"        ;BRACKET TEST FOR "A" t
                BLT NOTHEX
                CMP #"G"
                BGE NOTHEX      ;SAME AS BCS (SEE NEXT INSTR)
                SBC #$6         ;CONVERT FROM $C1 TO $BA ...
        ISHEX   SEC             ;SIGNAL VALID HEX DIGIT
                RTS
        ;
        NOTHEX  CLC             ;SIGNAL INVALID HEX DIGIT
                RTS
```


****

To input a hexadecimal number, use this routine in place of TSTDEC and replace the JSR TSTDEC with JSR TSTHEX in BCDIN. Obviously, the name should be changed to HEXIN so that it makes a little more sense.

UNSIGNED DECIMAL INPUT.

Decimal input of numeric data (with conversion to binary) is only slightly more difficult than BCD or hexadecimal input. The algorithm to accomplish decimal input is roughly as follows:

1) Input a character and test for validity (i.e., is it in the range 0-9?).

2) Strip the high-order four bits to give the numeric representation of the digit.

3) Multiply a 16-bit memory location by ten and add the stripped digit to this 16-bit location.

4) When all the digits have been shifted in, the 16-bit val-
contained in the two memory locations is the binary contained in the two memory locations is the binary representation of the decimal value.

****

Parts one and two are accomplished in the same manner as for BCD numbers. As such, they will not be discussed further here. The third part of this algorithm (multiplying a 16-bit value by ten) is easily accomplished using the multiply routines in the next chapter. However, a more specialized multiplication routine (a simple mulitplication by ten) is much faster and requires less code. A routine which multiplies the 16-bit value held in locations VALUE and VALUE+$1 by ten is:

```
MUL100
        PHP
        PHA
        ASL VALUE       ;MULTIPLY VALUE BY 2
        ROL VALUE+$1
        LDA VALUE+$1    ;SAVE A COPY OF VALUE
        PHA             ;MULTIPLIED BY 2
        LDA VALUE
        ASL VALUE       ;NOW MULTIPLY VALUE BY 8
        ROL VALUE+$1    ;SINCE VALUE HAS ALREADY
        ASL VALUE       ;BEEN MULTIPLIED BY 2
        ROL VALUE+$1    ;A SIMPLE MULTIPLY BY 4 GIVES
        CLC
        ADC VALUE       ;ADD IN 2xVALUE TO 8xVALUE
        STA VALUE       ;TO OBTAIN 10xVALUE
        PLA
        ADC VALUE+$1
        STA VALUE+$1
        PLA
        PLP
        RTS
```

Each time this routine is called, it multiplies the contents of VALUE by ten, leaving all registers unchanged.

The final step in the algorithm (adding in the digit to the 16-bit number) is trival at this point. The final decimal input routine could be:

```
; DECIMAL INPUT ROUTINE
;
; NOTE: THIS ROUTINE ASSUMES THAT GETLN HAS BEEN
```

```
;       CALLED AND THAT THE X-REGISTER POINTS TO
;       THE FIRST VALID DECIMAL DIGIT IN PAGE 2.
;       UPON EXIT, THE X-REGISTER POINTS TO  THE
;       FIRST NON-DIGIT ENCOUNTERED.
;
;
DECINP:
        PHP             ;SAVE STATUS
        PHA             ;AND ACC
;
        LDA #$0         ;INITIALIZE VALUE
        STA VALUE       ;TO ZERO
        STA VALUE+$1
;
```



```
**************************************************************

DECLP   LDA INPUT,X     ;GET THE NEXT DIGIT
        JSR TSTDEC      ;IS IT REALLY A DIGIT?
        BCC ALDONE      ;IF NOT, QUIT
        AND #$F         ;OTHERWISE CONVERT TO A NUMBER
        JSR MULT10      ;MULTIPLY VALUE BY 10
        CLC
        ADC VALUE       ;AND ADD IN CURRENT DIGIT
        STA VALUE
        BCC DECLP       ;IF NOT CARRY, LOOP BACK
        INC VALUE+$1    ;IF A CARRY EXISTS, ADD ONE
        JMP DECLP       ;TO VALUE+$1 AND LOOP BACK
;
ALDONE  PLA             ;RESTORE REGISTERS
        PLP
        RTS
;
;
; TSTDEC: TEST ACC TO SEE IF IT IS A VALID DECIMAL
;         DIGIT.  IF SO, THE CARRY FLAG IS SET.
;         OTHERWISE THE CARRY FLAG IS CLEAR.
;
TSTDEC:
        CMP #"0"
        BLT NOTDEC
        CMP #"9"+$1
        BGE NOTDEC
        SEC
        RTS
;
NOTDEC  CLC
RTS
;
;
; MULT 100 MULTIPLIES VALUE BY TEN
;        (SEE ABOVE)
;
MULT100
        PHP
```

```
            PHA
            ASL VALUE
            ROL VALUE+$1
            LDA VALUE+$1
            PHA
            LDA VALUE
            ASL VALUE
            ROL VALUE+$1
            ASL VALUE
            ROL VALUE+$1
            CLC
            ADC VALUE
            STA VALUE
            PLA
            ADC VALUE+$1
            STA VALUE+$1
            PLA
            PLP
            RTS
;
; THAT'S ALL FOLKS...
;
```


****

This routine does suffer from a few drawbacks. First, it does not check for overflow. Second, it terminates entry upon the first non-digit encountered, which means that bad data entries will go undetected. Finally, if the first character encountered is not a decimal digit, the routine immediately returns and zero is returned in value.

Luckily, these three problems are easily handled. To check for overflow, check the carry flag to see if it is set after each ROL instruction in the MULT10 routine, and check the carry flag after the addition in the DECINP routine. If the carry flag is ever set at any of these points overflow has occurred.

The second problem (termination on the first non-digit) is a problem because it allows illegal data entries to go unchecked. Typically, numeric input should be terminated by either a space, a return, or a comma (or any other special character you might think of). If one of these special characters is not encountered, an input error should result. This problem is easily handled by checking the first non-digit character to make sure it is one of the allowable delimiters.

The last problem (invalid first character) is simply an extension of the second problem. Handling this problem is likewise an easy one to solve. First, delete all leading blanks (since leading blanks should be allowable in a number). Next, test the first non-blank to insure that it is a valid decimal digit. If not, report an error. The following routine takes all of these factors into account and more or less simulates the integer input in Apple's Integer BASIC:

```
DECINP:
        PHP
        PHA
;
DOIT:
        LDA #$0         ;INIT VALUE
        STA VALUE
        STA VALUE+$1
        JSR BLKDEL      ;DELETE LEADING BLANKS
        JSR TSTDEC      ;IS FIRST NON-BLANK A DIGIT?
        BCC BADDIG      ;IF NOT, INFORM THE USER
;
DECLP   LDA INPUT,X     ;GET NEXT (OR FIRST) DIGIT
        INX             ;MOVE TO NEXT CHARACTER
        JSR TSTDEC      ;IS IT A DIGIT?
        BCC ALDONE      ;IF NOT, QUIT
        AND #SF         ;CONVERT TO A NUMBER
        JSR MULT10      ;MULTIPLY VALUE BY 10
        BVS OVRFLW      ;IF OVERFLOW, INFORM USER
        CLC
```


****

```
        ADC VALUE       ;ADD CURRENT DIGIT
        STA VALUE       ;TO VALUE
        BCC DECLP
        INC VALUE+$1    ;IF CARRY, INCREMENT VALUE+
        BNE DECLP       ;IF NO OVERFLOW, LOOP BACK
        JMP OVRFLW      ;IF OVRFLOW, INFORM USER
;
ALDONE  CMP #","        ;TEST FOR VALID DIGIT
        BEQ QUIT        ;DELIMITERS
        CMP #" "
        BEQ QUIT
        CMP #$8D        ;RETURN IS VALID
        BEQ QUIT
        JSR PRINT       ;PRINT ROUTINE FROM A PREVIOUS
        HEX 8D          ;CHAPTER
        ASC "RETYPE NUMBER"
        HEX 8D00
        JSR GETLN       ;READ A LINE OF TEXT
        LDX #$0
        JMP DOIT
;
OVRFLW  JSR PRINT
        HEX 8D
        ASC ">65535"
        HEX 8D00
        JSR GETLN       ;GET A NEW LINE OF TEXT
        LDX #$0
        JMP DOIT
;
;
QUIT:
        PLA
```

```
        PLP
        RTS
;
;
; BLANK DELETION ROUTINE
;
BLKDEL  LDA INPUT,X
        CMP #" "
        BNE BLKD1
        INX
        BNE BLKDEL
;
BLKD1   RTS
;
; MULTIPLY BY 10 ROUTINE
;
MULT10  PHA             ;CAN'T SAVE CARRY, V IS OVRFLW
;
        ASL VALUE
        ROL VALUE+$1
        BCS MOVRFL
        LDA VALUE+$1
        PHA
        LDA VALUE
        ASL VALUE
        ROL VALUE+$1
```


****

```
        BCS MOVRFL
        ASL VALUE
        ROL VALUE+$1
        BCS MOVRFL
        CLC
        ADC VALUE
        STA VALUE
        PLA
        ADC VALUE+$1
        STA VALUE+S1
        BCS MOVRFL
        PLA
        BIT             ;SET V FLAG TO ZERO
        RTS

MOVRFL
        BIT OVERFL      ;SET V FLAG TO ONE
        RTS
;
NOVRFL HEX 00
OVERFL HEX 40
;
;
; TSTDEC: TESTS CHARACTER IN ACC TO SEE IF IT IS
;         A VALID DECIMAL DIGIT
;         CARRY IS SET IF IT IS
```

```
          ;
          TSTDEC:
                  CMP #"0"
                  BLT NOTDEC
                  CMP #9"+$1
                  BGE NOTDEC
                  SEC
                  RTS
          ;
          NOTDEC  CLC
                  RTS
          ;
          ;
          INPUT   EQU $200
          VALUE   EPP $0
          ;
          GETLN   EQU $FD67
          ;
          ; NOTE: THE PRINT ROUTINE PROVIDED IN THE PREVIOUS
          ;       CHAPTER MUST BE INCLUDED HERE
          ;
          ;
```

To use this routine, read a line of data using GETLN. Set up the X-register so that it points to the desired decimal digits to be input (leading blanks allowed) and then JSR DECINP. Upon returning from DECINP the desired number (in binary form) will be stored in VALUE and VALUE+$1. There are some improvements you may want to make to this basic routine, such as:

****

1) Modification to handle three, four (or more) byte integers

2) The ability to specify an address after the JSR DECINP with the resulting input integer being stored at that address (sort of the inverse of the decimal output routine presented earlier in this chapter).

SIGNED DECIMAL INPUT.

Once we have the unsigned decimal input routine, the signed decimal input routine becomes very easy. All we have to do is check to see if the first non-blank character is a minus sign. If it is, increment to the next character and call the unsigned decimal input routine. Upon return from the unsigned decimal input routine, check the high-order bit of VALUE+$1. If it is set, an overflow has occured. If it is not set, then take the two's complement of the VALUE and VALUE+$1 if a minus sign was used; otherwise, leave the number alone. The actual routine is:

```
          ; SIGNED DECIMAL INPUT
          ;
          SNGDEC:
                  PHP
```

```
        PHA
;
DOSGN:
        JSR BLKDEL
        CMP #"-"
        BNE SGN1
        LDA #$1         ;SET A FLAG SIGNIFYING
        STA SIGN        ;A MINUS VALUE
        INX
        JMP SGN2
;
SGN1    LDA #$0         ;SET A FLAG SIGNIFYING
        STA SIGN        ;A POSITIVE NUMBER
;
SGN2    JSR DECINP      ;GET THE UNSIGNED NUMBER
        LDA VALUE+$1    ;TEST FOR OVERFLOW
        BMI SGNOVR
        LDA SIGN        ;TEST TO SEE IF 2'S COMP
        BFL DONE        ;IS REQUIRED
        SEC             ;PERFORM 2'S COMP
        LDA #$0         ;OPERATION
        SBC VALUE
        STA VALUE
        LDA #$0
        SBC VALUE+$1
;
DONE    PLA
        PLP
        RTS
```


****

```
;
SGNOVR  JSR PRINT
        HEX 8D
        ASC ">32767  REENTER"
        HEX 8D00
        JSR GETLN
        LDX #$0
        JMP DOSGN
SIGN    EPZ VALUE+$2
```

That completes the general numeric I/O routines required for normal "BASIC-LIKE" operations. These routines present the basis for almost all other types of numeric I/O. By modifying these routines you can perform multi-byte inputs, single-byte inputs, etc. Other types of numeric input, such as octal or binary, are accomplished by simply modifying the MULT10 and TSTDEC routines to reflect the new radix, (e.g., you would use MULT8, and TSTOCT for octal input). In fact, you could write a general routine that could input data using any radix, but see the multiply routines in the next chapter first.

****

CHAPTER 13

MULTIPLICATION AND DIVISION

GENERAL.

As we've mentioned before, the 6502 microprocessor does not have a multiply or divide instruction. Obviously, a multiply or divide instruction would be very handy to have. Since the 6502 does not support these functions, we are forced to write subroutines to provide this capability for us.

MULTIPLICATION.

Multiplication in binary is very, very simple. In fact, it is identical to decimal multiplication. Consider the following DECIMAL multiplication problem:

```
 10110
x  110
 -----
```

Just add (0 x 10110) plus (10 x 10110) plus (100 x 10110) and you've got the result. Multiplication by 10 is very easy; just shift the number one place to the left of the decimal point. Incidentally, the answer to the above problem is 1112100 (decimal).

The same procedure is used in multiplying two binary numbers. Just add (0 x 10110) plus (10 x 10111) plus (100 x 10110) get the final result. For multiplication by powers of two, just use the ASL or ROL instructions to perform the multiplication by the desired power of two. The answer to the above problem (in binary) is 10000100.

****

It should be noted at this point that an n-bit by m-bit multiplication can result in a maximum of an (m+n) bit result. Therefore, an 8-bit by 8-bit multiplication can produce results up to 16-bits in length. Likewise a 16-bit by 16-bit multiplication can result in values up to 32 bits in length. With this in mind we must make sure that there are enough memory locations reserved to hold the results produced by our multiplication routine.

The following multiplication routine uses six zero page memory locations. They are used to hold the multiplicand, multiplier, and partial result. These locations (all 16-bits for this example) will be labeled MULCND, MULPLR, and PARTIAL. After the multiplication is complete, the low-order 16 bits of the result will be left in locations (MULPLR, MULPLR+$1) and the high-order 16 bits of the product (if not zero) will be left in locations (PARTIAL,

PARTIAL+$1). This routine will compute the value:

     (MULPLR, PARTIAL) = MULPLR x MULCND + PARTIAL.
You will note that PARTIAL is added into the result of the multiplication. This is useful in several mathmatical calculations, including extended-precision multiplication. For our purposes, however, just remember to set locations PARTIAL and PARTIAL+$1 to zero before calling the multiply routine.

****

     The following multiplication routine is a modified version of the multiply routine found in the Apple monitor at location $FB63. (Note: this routine is available only in the older version of the Apple monitor. It is not available in the newer Autostart ROM monitor.)

```
          ; USMUL- UNSIGNED 16-BIT MULTIPLICATION.
          ;        32 BIT RESULT IS RETURNED IN LOCATIONS
          ;        (MULPLR, PARTIAL).
          ;
          ;
          USMUL:
                  PHA
                  TYA
                  PHA
          ;
          USMUL1  LDY #$10        ;SET UP FOR 16-BIT MULTIPLY
          USMUL2  LDA MULPLR      ;TEST L.O. BIT TO SEE IF SET
                  LSR
                  BCC USMUL4
          ;
                  CLC             ;L.O. BIT SET, ADD MULCND TO
                  LDA PARTIAL     ;PARTIAL PRODUCT
                  ADC MULCND
                  STA PARTIAL
                  LDA PARTIAL+$1
                  ADC MULCND+$1
                  STA PARTIAL+$1
          ;
          ; SHIFT RESULT INTO MULPLR AND GET THE NEXT BIT
          ; OF THE MULTIPLIER INTO THE LOW!ORDER BIT OF
          ; MULPLR
          ;
          USMUL4  ROR PARTIAL+$1
                  ROR PARTIAL
                  ROR MULPLR+$1
                  ROR MULPLR
          ;
          ; SEE IF DONE YET
          ;
                  DEY
                  BNE MUL2
                  PLA
                  TAY
```

```
        PLA
        RTS
;
;
MULPLR  EQP $50
PARTIAL EPZ MULPLR+$2
MULCND  EPZ PARTIAL+$2
```

The following example demonstrates the use of the multiply function:


```
*****************************************************************

        ; COMPUTE 25 x 66 AND LEAVE RESULT IN
        ; "RESULT"

        EXMPL:
                LDA #!25        ;25 DECIMAL
                STA MULPLR
                LDA /!25        ;H.O. BYTE OF 25
                STA MULPLR+$1
                LDA #!66
                STA MULOND
                LDA /!66
                STA MULCND+$1
                LDA #$0         ;MUST SET PARTIAL TO ZERO
                STA PARTIAL
                STA PARTIAL+$1
                JSR USMUL       ;PERFORM THE MULTIPLICATION
                LDA MULPLR      ;MOVE PRODUCT TO RESULT
                STA RESULT
                LDA MULPLR+$1
                STA RESULT+$1


                ETC...
```

If you are performing a 16-by-16-bit multiplication and the result is going to be stored in a 16-bit memory location, you may check for overflow by OR'ing PARTIAL and PARTIAL+$1 together. If the result is not zero, then overflow has occurred into the high-order 16 bits.

As mentioned previously, PARTIAL can be used to generalize this routine so that 24-, 32-, 48-, 64-, etc. bit multiplications can be performed. It is easier, though, just to modify the existing routine for the higher precision routines. To do this, simply load the Y-register with the number of bits you wish to multiply together and then modify the multiprecision ROR sequence and the multiprecision ADC sequence to reflect the precision you choose. Oh yes, don't forget to reserve more room for MULCND, PARTIAL, and MULPLR! An example of a 24 by 24-bit multiplication giving a 48-bit result might be:

```
; USMUL- UNSIGNED 24-BIT MULTIPLICATION
;        48 BIT RESULT IS RETURNED IN LOCATIONS
;        (MULPLR, PARTIAL)
;
;
USMUL:
        PHA
        TYA
        PHA
```


****

```
;
USMUL1  LDY #$18        ;SET UP FOR 24-BIT MULTIPLY
USMUL2  LDA MULPLR      ;TEST L.O. BIT TO SEE IF SET
        LSR
        BCC USMUL4
;
        CLC             ;L.O. BIT SET, ADD MULCND TO
        LDA PARTIAL     ;PARTIAL PRODUCT
        ADC MULCND
        STA PARTIAL
        LDA PARTIAL+$1
        ADC MULCND+$1
        STA PARTIAL+$1
        LDA PARTIAL+$2
        ADC MULCND+$2
        STA PARTIAL+$2
;
; SHIFT RESULT INTO MULPLR AND GET THE NEXT BIT
; OF THE MULTIPLIER INTO THE LOW ORDER BIT OF
; MULPLR
;
USMUL4  ROR PARTIAL+$2
        ROR PARTIAL+$1
        ROR PARTIAL
        ROR MULPLR+$2
        ROR MULPLR+$1
        ROR MULPLR
;
; SEE IF DONE YET
;
        DEY
        BNE MUL2
        PLA
        TAY
        PLA
        RTS
;
;
MULPLR  EPZ $50
PARTIAL EPZ MULPLR+$3
MULCND  EPZ PARTIAL+$3
```

It should be stressed that the above routines are for UNSIGNED mulitiplication only. Signed multiplication is accomplished by first noting the signs of the multiplier and multiplicand and setting a sign flag if the sign bits do not equal each other. The absolute value of the multiplier and multiplicand is then taken, and the unsigned mulitplication routine is used. After the unsigned multiplication takes place, the sign flag is tested. If it indicates that the original sign bits were not equal to one another, the product must be negated.


```
**************************************************************

          ; SIGNED 16-BIT MULTIPLICATION
          ;
          SMUL:
                  PHA
                  TYA
                  PHA
          ;
                  LDA MULCND+$1   ;TEST SIGN BITS
                  XOR MULPLR+$1   ;TO SEE IF H.O BITS ARE UNEQU
                  AND #$80
                  STA SIGN        ;SAVE SIGN STATUS
                  JSR ABS1        ;TAKE ABSOLUTE VALUE OF MULPLR
                  JSR ABS2        ;TAKE ABSOLUTE VALUE OF MULCND
                  JSR USMUL       ;UNSIGNED MULTIPLY
                  LDA SIGN        ;TEST SIGN FLAG
                  BPL SMUL1       ;IF NOT SET, RESULT IS CORRECT
                  JSR NEGATE      ;NEGATE RESULT
          ;
          SMUL1   PLA
                  TAY
                  PLA
                  RTS
          ;
          ;
          ABS1    LDA MULPLR+$1   ;SEE IF NEGATIVE
                  BPL ABS12
          ;
          NEGATE:
                  SEC             ;NEGATE MULPLR
                  LDA #$0
                  SBC MULPLR
                  STA MULPLR
                  LDA #$0
                  SBC MULPLR+$1
                  STA MULPLR+$1
          ;
          ABS12   RTS
          ;
          ;
          ABS2    LDA MULCND+$1   ;SEE IF NEGATIVE
```

```
                BPL ABS22
                SEC             ;NEGATE MULCND
                LDA #$0
                SBC MULCND
                STA MULCND
                LDA #$0
                SBC MULCND+$1
                STA MULCND+$1
        ;
        ABS22   RTS
```

As with the unsigned multiply routine, you can check for overflow by OR'ing PARTIAL with PARTIAL+$1 and checking for zero. A signed multiply routine is provided in the older Apple monitor at location $FB60. You should study the technique used in the Apple

****

monitor for multiplication, as it is somewhat different than the technique employed here. It is certainly more complex and likewise more difficult to understand, but it is a good exercise in how to reduce code at the expense of clarity and speed.

DIVISION ALGORITHMS.

As with multiplication, the algorithm used for binary division is identical to the algorithm most people use when performing long division. First, you take the high-order bit of the divisor, if set, and then you see if the dividend is divisible. If it is, you note this in the running quotient and subtract the current divisor value from the dividend. When these steps have been completed for all digits (or bits), the division is complete. The division routine is coded as follows:

```
        ; UNSIGNED 16-BIT DIVISION
        ; COMPUTES (DIVEND,PARTIAL) / DIVSOR
        ; (I.E., 32 BITS DIVIDED BY 16 BITS)
        ;
        USDIV:
                PHA
                TYA
                PHA
                TXA
                PHA

                LDY #$10        ;SET UP FOR 16 BITS
        USDIV2  ASL DIVEND
                ROL DIVEND+$1
                ROL PARTIAL
                ROL PARTIAL+$1
                SEC             ;LEAVE DIVEND MOD DIVSOR
                LDA PARTIAL     ;IN PARTIAL
                SBC DIVSOR
                TAX
```

```
                LDA PARTIAL+$1
                SBC DIVSOR+$1
                BCC USDIV3
        ;
                STX PARTIAL
                STA PARTIAL+$1
                INC DIVEND
        ;
        USDIV3  DEY
                BNE USDIV2
        ;
                PLA
                TAX
                PLA
                TAY
                PLA
                RTS
```


****

```
        ;
        ;
        DIVEND  EPZ $50
        PARTIAL EPZ DIVEND+$2
        DIVSOR  EPZ PARTIAL+$2
```

It should be mentioned that this routine also computes DIVEND MOD DIVSOR, and this result is left in PARTIAL. Should division by zero be attempted, $FFFF will be returned in DIVEND. Your program can check for this problem by AND'ing DIVEND and DIVEND+$1 together, and then compare the result with the value $FF. Because of the method used to check for zero division, an ambiguity arises since $FFFF divided by one is also $FFFF. This problem can be remedied by explicitly checking for division of $FFFF by one before calling USDIV. This division routine can be expanded to any number of bytes of precision by loading the Y-register with the number of bits of precision required, extending the precision on the ROL instruction sequence, and extending the precision on the SBC sequence.

To use this routine, load a 32-bit dividend into locations DIVEND, DIVEND+$1, PARTIAL, and PARTIAL+$1 (low-order byte into DIVEND, the most-significant byte into PARTIAL+$1) and the 16-bit divisor into DIVSOR. Once this is accomplished, simply JSR to USDIV. If you only need to perform a 16-bit by 16-bit division, just store zeros into PARTIAL and PARTIAL+$1.

```
        ; EXAMPLE: DIVIDE 195 BY 24 AND PUT THE QUOTIENT
        ;          INTO "RESULT"
        ;          STORE THE MODULO OF 195/24 IN LOCATION
        ;          "MODULO"
        ;
        EXMPL:
                LDA #!195       ;DECIMAL 195
```

```
                STA DIVEND
                LDA /!195
                STA DIVEND+$1
                LDA #!24        ;DECIMAL 24
                STA DIVSOR
                LDA /!24
                STA DIVSOR+$1
                LDA #$0         ;PERFORMING A 16 BY 16 DIVISION
                STA PARTIAL
                STA PARTIAL
                JSR USDIV
                LDA DIVEND
                STA RESULT
                LDA DIVEND+$1
                STA RESULT+$1
```


****

```
                LDA PARTIAL
                STA MODULO
                LDA PARTIAL+$1
                STA MODULO+$1


                ETC ...
```

Signed division turns out to be only somewhat more complicated than unsigned division. As with the signed multiply routine, a sign flag is set up to determine the final sign of the result. Likewise, the absolute value of the dividend and divisor is taken, and then the unsigned division routine is called. Finally, the quotient is negated if the sign flag is set.

But there is one little "gotcha" which didn't occur with the multiply routine. If a division by zero occurs (within the unsigned multiply routine) $FFFF is returned. The only way (using the unsigned routine) that $FFFF can be returned is if you divide $FFFF by one. With the signed routines, however, you get a result of $FFFF (which is -1 in decimal) by dividing $FFFF by one, one by $FFFF, or in fact any division where both the positive and negative versions of a number end up in the divisor and dividend. Zero division causes the result of $FFFF to be returned. Since these cases are not all that rare, some steps have to be taken to correct the possible ambiguity. In the signed division routine which follows, the overflow flag is set or cleared depending on whether or not a zero division has occurred. If a division by zero occurred, the overflow flag will be set. If a division by zero did not occur, then the overflow flag will be cleared. Your programs can check the overflow flag upon return from the divsion routine and then take the appropriate action. You can also use this technique with the unsigned divsion routine to handle the case of $FFFF divided by one, if desired.

```
          ; SIGNED 16-BIT DIVISION ROUTINE
```

```
          ; V FLAG IS RETURNED SET IF ZERO DIVIDE OCCURS
          ;
          ; THIS ROUTINE COMPUTES (DIVEND,PARTIAL)/DIVEND
          ; AS WELL AS (DIVEND,PARTIAL) MOD DIVEND

          SDIV:
                  PHA
                  LDA DIVEND+$1   ;CHECK SIGN BITS
                  XOR DIVSOR+$1
```



```
**************************************************************

                  AND #$80
                  STA SIGN
                  JSR DABS1       ;ABSOLUTE VALUE OF DIVSOR
                  JSR DABS2       ;ABSOLUTE VALUE OF DIVEND
                  JSR USDIV       ;COMPUTE UNSIGNED DIVISION
                  LDA DIVEND      ;CHECK FOR ZERO DIVIDE
                  AND DIVEND+$1
                  CMP #$FF
                  BEQ OVRFLW
                  LDA SIGN        ;SIGN IF RESULT MUST BE
                  BPL SDIV1       ;NEGATIVE
                  JSR DIVNEG
          ;
          SDIV1   CLV             ;NO ZERO DIVISION
                  PLA
                  RTS

          OVRFLW  BIT SETOVR      ;SET OVERFLOW FLAG
                  PLA
                  RTS
          ;
          SETOVR  HEX 40
          ;
          ;
          DIVNEG:
                  LDA DIVSOR+$1
                  BPL DABS12
                  SEC
                  LDA #$0
                  SBC DIVSOR
                  STA DIVSOR
                  LDA #$0
                  SBC DIVSOR+$1
                  STA DIVSOR+$1
          ;
          DABS12  RTS
          ;
          ;
          DABS2   LDA DIVEND+$1
                  BPL DABS22
                  SEC
                  LDA #$0
                  SBC DIVEND
```

```
                STA DIVEND
                LDA #$0
                SBC DIVEND+$1
                STA DIVEND+$1
          ;
          DABS22  RTS
```

That pretty much wraps up multiplication and division. The basic routines in this chapter can be modified for special purpose applications quite easily. These routines were written with speed and ease of understanding in mind. Obviously, quite a bit of code

****

can be saved by using loops in several places, especially when expanding beyond 16 bits, but generally speed is much more important than four or five bytes.

****

CHAPTER 14

STRING HANDLING OPERATIONS

STRING HANDLING.

Numbers are okay, but string handling, as in BASIC, is the part that is fun. Character strings are represented in computer memory in a multitude of ways, but despite how a string is implemented in computer memory, it always has at least three attributes: (1) a maximum length, i.e., the number of bytes allocated to it; (2) a dynamic "run-time" length giving the current number of bytes currently being used in the string and (3) a starting address in memory.

Without going into the gory details of how any particular language stores its strings, certain conventions will be adopted due to the structure of LISA. Strings, for the remainder of this book, will take one of three forms:

1) A string will consist of a group of characters starting at a specified address and terminated by a special byte value, such as $00, (used in the PRINT routine several chapters ago).

2) A string may consist of a group of characters starting at a known location and terminated by a character whose high-order bit is opposite the rest of the string.

3) A string will consist of a length byte followed by the num-

ber of characters specified in the length byte.

The first two versions of a string presented here are useful mainly for input/output purposes. $00 is usually used as a delimiter for outputting characters, since it allows the entire 128 normal ASCII characters to be output. For input, $8D (carriage return) is

****

usually used, since carriage return is used to terminate input in most cases. The second version of strings presented here is a specialized version of Type 1. By specifying that the last byte in the string contains an inverted high-order bit, there is no need for a trailing byte. It should be noted that this method restricts you to a maximum of 128 characters, as opposed to a maximum of 255 characters, possible with Type 1 strings, but you save a byte for each declared string. LISA has a special pseudo opcode that stores strings in this manner. The pseudo opcode is called, "DCI," and it stores strings in memory with the last character containing an inverted high-order bit. Refer to the LISA documentation for further details.

The third type of string (a length byte followed by the string itself) is the most common type of string used, because it is the most convenient to use. With it, string functions such as concatenation, length, and substring become trivial. For most of the string handling routines presented in this chapter, this type of string will be used. Since it is possible to have Type 1, 2, and 3 strings within a program, it seems we will need conversion routines to be able to convert Type 1 and Type 2 strings to Type 3 strings. These routines are very easy to write, so let's tackle them first.

To convert Type 1 strings to Type 3 strings we must have three pieces of information. First, we need to know the beginning address of the Type 1 string. Second, we need to know the beginning address of the Type 3 string, where the converted Type 1 string is to be stored. Finally, we need to know the value of the delimiting character used in the Type 1 string. For our routine, we will assume that these three pieces of information are passed in locations Type1, Type3, DLMTR. Both Type1 and Type3 will be 16-bit addresses and will require two zero page locations each. DLMTR, obviously, will require only one byte in page zero. These locations must be set up with the appropriate data before our subroutine is called.

The routine will pick a character out of the string pointed to by Type1 and store it in the corresponding location in the string pointed to by Type 3, with one slight change. Since the first byte of the string pointed to by Type3 must be reserved for the length of the string, it becomes necessary to increment the value in Type3 by one before storing the string in the designated area.

****

Finally, when the string has been transferred, the length of the string must be stored in the first location. The routine which does all of these mystical and magical things follows:

```
          ; TYPE1 TO TYPE3 STRING CONVERSIONS
          ;
          ; POINTERS TO THE RESPECTIVE DATA AREAS
          ; ARE PASSED IN "TYPE1" AND "TYPE2"
          ; "DLMTR" CONTAINS THE STRING DELIMITER BEING USED
          ;
          TYPE1   EPZ $0
          TYPE3   EPZ TYPE1+$2
          DLMTR   EPZ TYPE3+$2
          ;
          ;
          T1TO3:
          ;
                  PHP             ;SAVE ALL THE REGISTERS
                  PHA
                  TYA
                  PHA
          ;
                  INC TYPE3       ;ADD ONE TO TYPE3 POINTER
                  BNE T1TO3A      ;SO THAT IT POINT TO THE FIRST
                  INC TYPE3+$1    ;AVAILABLE CHAR PAST THE LENGTH
          ;
          T1TO3A:
                  LDY #$0         ;SET UP INDEX TO ZERO
          T1TO3B  LDA (TYPE1),Y   ;FETCH TYPE1 CHARACTER
                  CMP DLMTR       ;IS IT THE DELIMITER?
                  BEQ T1TO3C      ;IF SO, PREPARE TO QUIT
                  STA (TYPE3),Y   ;OTHERWISE TRANSFER
                  INY             ;MOVE TO NEXT CHARACTER
                  BNE T1TO3B      ;DON'T ALLOW STRINGS > 255
                  DEY             ;IF OVERFLOW OCCURS, TRUNCATE
          ;
          ;
          T1TO3C  LDA TYPE3       ;DECREMEOT TYPE 3 POINTER SO
                  BNE T1TO3D      ;IT POINTS TO LENGTH BYTE AGAIN
                  DEC TYPE3+$1
          T1TO3D  DEC TYPE3
          ;
                  TAY             ;TRANSFER LENGTH OF STRING TO A
                  LDY #$0         ;SET UP INDEX TO LENGTH BYTE
                  STA (TYPE3),Y   ;STORE LENGTH IN FIRST BYTE
          ;
                  PLA             ;RESTORE THE REGISTERS
                  TAY
                  PLA
                  PLP
                  RTS
```

If you read a line of text from the Apple keyboard, using the monitor GETLNZ routine, you could convert it to a Type 3 string using the following code sequence:

```
*****************************************************************

                LDA #$200       ;INIT TYPE1 TO $200
                STA TYPE1
                LDA /$200
                STA TYPE1+$1
        ;
                LDA #STRING     ;PUT ADDRESS OF DESTINATION
                STA TYPE3       ;STRING INTO "TYPE3"
                LDA /STRING
                STA TYPE3+$1
        ;
                LDA #$8D        ;INITILIZE THE DELIMITER
                STA DLMTR       ;CHARACTER TO RETURN
                JSR T1TO3       ;PERFORM THE CONVERSION


                ETC.
```

Type 2 strings are converted in a similar manner. The routine to perform the conversion is listed below:

```
        ;TYPE 2 TO TYPE 3 STRING CONVERSION
        ;TYPE 2 STRING IS ASSUMED TO BE A STRING WHOSE HIGH
        ;ORDER BITS ARE ALL SET EXCEPT FOR THE LAST CHARACTER
        ;WHOSE HIGH ORDER BIT IS CLEAR
        ;THIS CAN BE MODIFIED BY REPLACING THE "BPL"
        ;INSTRUCTION WITH A "BMI" IF DESIRED

        TYPE2   EPZ $0
        TYPE3   EPZ TYPE2+$2
        ;
        ;
        T2TO3:
                PHP             ;SAVE THE REGISTERS
                PHA
                TYA
                PHA
        ;
                INC TYPE3       ;MOVE PAST THE LENGTH BYTE
                BNE T2TO3A
                INC TYPE3+$1
        ;
        T2TO3A:
                LDY #$0         ;INITIALIZE STRING INDEX
        T2TO3B  LDA (TYPE2),Y
                BPL T2TO3C
                STA (TYPE3),Y
                INY
                BNE T2TO3B      ;PREVENT OVERFLOW
                DEY             ;TRUNCATE TO 255 CHARS
        ;
        T2TO3C  ORA #$80        ;STORE LAST CHARACTER
                STA (TYPE3),Y
```

```
                INY             ;ADJUST LENGTH
                BNE T2TO3D      ;TEST FOR OVERFLOW
                DEY             ;TRUNCATE IF > 255 CHARS
```


****

```
        ;
        T2TO3D  LDA TYPE3       ;MOVE TYPE3 POINTER BACK
                BNE T2TO3D      ;LENGTH BYTE
                DEC TYPE3+$1
        T2TO3E  DEC TYPE3
        ;
                PLA             ;RESTORE THE REGISTERS
                TAY
                PLA
                PLP
                RTS
```

Going in the other direction (from Type 3 strings to Type 1 or Type 2 strings) is rarely used, but just as simple to perform. Since this type of conversion is not used that much its design will be left to the reader as an exercise, should this type of conversion be required.

DECLARING LITERAL STRINGS.

Not all strings used within a program are likely to be input from the keyboard. Some ability must be provided to enter literal strings within a program.

You could count up all the characters in a string and manually preface the string with a length byte, but that would be very tedious. You could enter the string as a Type 1 or Type 2 string; and then use the conversion routines presented earlier to convert them to a Type 3 string, but that's still quite a bit of work. Luckily, LISA provides a pseudo opcode that does all the work for you. The pseudo opcode is "STR" and it outputs a string of ASCII characters prefaced automatically with a length byte. STR is very useful for declaring string constants. Since the APPLE II computer likes to have the high-order bit on for most applications, strings declared when using the STR pseudo opcode should always be enclosed by quotes (as opposed to apostrophes).

STRING ASSIGNMENTS.

Probably the most basic and useful operation that can be perfomed on a string is a string assignment. In its simplist form a string assignment is nothing more than a small in-line coded loop that transfers data from one location to another. Assuming you want to transfer the string in "STR1," to the string at "STR2," you might use the following:

```
*****************************************************************

                LDY STR1        ;GET THE LENGTH BYTE
          LOOP  LDA STR1,Y      ;TRANSFER STRING
                STA STR2,Y
                DEY
                BNE LOOP
                LDA STR1        ;TRANSFER THE LENGTH OVER
                STA STR2
```

As you can see, data bytes 1 through n (where n is the length of the string) are transferred and then the length of STR1 is stored in the length byte of STR2.

This, of course, is a very simple string assignment loop, yet it is small enough to be coded in-line in most cases. If you perform quite a few string assignments within a program, it might be worth your while to write a routine that allows you to specify the addresses of the two strings after a JSR, as an example:

```
                JSR SASIGN      ;STRING ASSIGNMENT
                ADR DEST        ;DEST = SOURCE
                ADR SOURCE
```



```
*****************************************************************
```

which only requires 7 bytes per assignment. Another version of this special string assignment might take the form:

```
                JSR SASGMI      ;IMMEDIATE STRING ASSIGNMENT
                ADR DEST        ;ADDRESS OF STRING
                STR "HELLO"     ;STRING TO BE ASSIGNED
```

This form allows you to assign string constants to a desired string with a minimum of complexity. These two methods will be left for the reader to write as an exercise. You should look at the print routines presented in an earlier chapter and use them as a template for the string assignment subroutines.

STRING FUNCTIONS.

One of the most basic string functions is the length function. It will be the basis of many other string functions which follow. Its implementation is trivial. Since the length of a string is always stored in the first byte of a string, the length function is simply a load instruction. For example, if we have the following string declaration:

```
                STRING  STR  "HELLO THERE"
```

then a simple LDA STRING will load the length of the string into the accumulator.

With the length function out of the way, string output is next on the list. String output is very easy. The following routine will output the string stored at location 'STRING:'

```
        LDA STRING      ;CHECK LENGTH TO INSURE
        BEQ XIT         ;IT IS NOT ZERO
;
        LDY #$0         ;SET UP INDEX TO FIRST CHAR
LOOP    LDA STRING+$1,Y ;GET THE NEXT CHARACTER
        JSR COUT        ;OUTPUT IT
        INY
        CPY STRING      ;DONE YET?
        BLT LOOP
```

Note that the Y-register is loaded with zero, and then the accumulator is loaded from location STRING plus one. This insures us that the Y-register will be equal to the length of the string when it is pointing to one character beyond the end of the string, so that the Y-register will always be less than the length of the string

****

while it contains a valid index. This allows us to use the BLT instruction to terminate the loop.

A much better string output routine would be a subroutine that causes the address of the string to be output immediately after the JSR, much like the print routines presented earlier in the book. This routine would be coded as follows:

```
PRTSTR:
        STA ASAVE
        STY YSAVE
        PLA             ;GET RETURN ADDRESS FROM
        STA RTNADR      ;THE 6502 STACK
        PLA
        STA RTNADR+$1
;
        JSR INCRTN      ;INCREMEOT THE RETURN ADDRESS
        LDY #$0
        LDA (RTNADR),Y  ;GET L.O. ADDRESS OF STRING
        STA ZPAGE
        INY
        LDA (RTNADR),Y  ;GET H.O.ADDRESS OF STRING
        STA ZPAGE+$1
;
        JSR INCRTN      ;MOVE RTNADR PAST THE ADDRESS
        JSR INCRTN      ;BYTES
;
;
;  AT THIS POINT, ZPAGE POINTS TO THE STRING WHICH
;  IS SUPPOSED TO BE OUTPUT
;
        DEY             ;RESET Y REG TO ZERO
```

```
                LDA (ZPAGE),Y   ;GET THE LENGTH OF THE STRING
                STA LENGTH      ;AND STORE IT IN "LENGTH"
        PRTS1   INY             ;MOVE TO THE NEXT CHARACTER
                CPY LENGTH      ;ARE WE THROUGH YET?
                BEQ PRTS2
        ;
                LDA (ZPAGE),Y   ;GET THIS CHARACTER
                JSR COUT        ;AND OUTPUT
                JMP PRTS1       ;MOVE TO NEXT CHAR AND REPEAT
        ;
        PRTS2   LDA ASAVE       ;RESTORE THE REGISTERS
                LDY YSAVE
                JMP (RTNADR)    ;SIMULATE AN RTS
        ;
        ;
        ;
        ASAVE   EPZ $0          ;ZERO PAGE WORKSPACE
        YSAVE   EPZ ASAVE+$1
        ZPAGE   EPZ YSAVE+$1
        RTNADR  EPZ ZPAGE+$2

        COUT    EQU $FDED       ;COUT ROUTINE
                END
```


****

This routine is used by JSR'ing to PRTSTR and following the JSR with the address of the string to be output.

EXAMPLE:

```
                JMP START
        STRING  STR "HELLO THERE"
        ;
        START   JSR PRTSTR
                ADR STRING


                ETC.
```

prints "HELLO THERE" onto the current output device. Naturally, any string may be output using PRTSTR, not just strings declared using the STR pseudo opcode.

STRING CONCATENATION.

String concatenation is the operation of taking two strings and joining them together to make a single long string. Typically, two strings are combined and their concatenated result is stored in a third string.

String concatenation is accomplished in the following manner. First, the lengths of the two source strings are added together. If this result is less than the maximum length of the destination

string, then things are fine. If the length is greater than the maximum length of the string, then an error must be reported. If the

****

sum of the two source string lengths is less than the maximum number of characters possible for the destination string, the sum of the two lengths is stored in the first byte of the destination string. This will be the length of the new string. Next, the first string is transferred to the destination string. Finally, the second source string is transferred to the destination string immediately after the first string. A short routine which concatenates STR1 and STR2 storing the result at STR3, is:

```
          ; STRING CONCATENATION EXAMPLE
          ;
          ;
          ;
          ;
          ;FIRST, CHECK LENGTHS
          ;
                  CLC
                  LDA STR1
                  ADC STR2
                  BCS ERROR       ;> 255 CHARS IS ALWAYS BAD
                  STA STR3        ;STORE LENGTH IN STR3
                  LDA MAXLEN      ;GET MAXIMUM LENGTH OF STR3
                  CMP STR3        ;AND COMPARE TO DESIRED LENGTH
                  BLT ERROR       ;IF LESS THAN, AN ERROR MUST
                                  ;BE FLAGGED
          ;
          ;
          ; THINGS ARE FINE HERE, SO MOVE STR1 TO STR3
          ;
                  LDY #$0
          CONCT1  LDA STR1+$1,Y   ;GET CHAR FROM STR1
                  STA STR3+$1,Y   ;AND MOVE TO STR3
                  INY
                  CPY STR1        ;DONE YET?
                  BLT CONCT1
          ;
          ;
          ; NOW, TRANSFER STR2 TO THE TAIL END OF STR3
          ;
                  LDX #$0
          CONCT2  LDA STR2+$1,X   ;GET CHAR FROM STR2
                  STA STR3+$1,Y   ;TRANSFER TO STR3
                  INY
                  INX
                  CPX STR2        ;DONE YET?
                  BLT CONCT2


                  ETC...
```

SUBSTRING OPERATIONS.

One very important string function is the substring function, which allows the programmer to extract a portion of a string and

****

assign the extracted portion to another string.

To extract a substring, we need four pieces of information: the address of the source string, the address of the destination string, a value specifying the start of the substring, and a length of the substring. The specified length is checked to make sure it is not greater than the maximum permissible string length for the destination string. If it is, an error must be reported. If the length of the substring is less than the maximum length allowable by the destination string, then the length byte is stored in the first location of the destination string. One final check must be made. We must insure that there are at least "length" characters the source string beginning at the index specified. Otherwise, unfortunately, an error must be reported. The following routine extracts the substring beginning at location "START" in string "STR1" and of length "LENGTH." The resulting substring is stored into "STR2."

```
          ; SUBSTRING EXAMPLE
          ;
          STR1    EPZ $0
          STR2    EPZ STR1+$2
          START   EPZ STR2+$2
          LEN1    EPZ START+$1
          MAXSTR  EPZ LEN1+$1
          INDEX   EPZ MAXSTR+$1
          LENGTH  EPZ INDEX+$1
          ;
          ;
          SUBSTR:
                  PHP
                  PHA
                  TYA
                  PHA
          ;
          ;
          ; CHECK TO SEE IF LENGTH OF SUBSTRING IS GREATER
          ; THAN THE LENGTH OF STR2 (PASSED IN MAXSTR)

                  LDA MAXSTR
                  CMP LENGTH
                  BLT ERROR

          ;
          ; CHECK TO SEE IF ENOUGH CHARS IN STR1
          ;
                  CLC
                  LDA INDEX
                  BEQ ERROR       ;INDEX OF ZERO NOT ALLOWED
```

```
                ADC LENGTH
                BCS ERROR       ;IF > 255 THEN ALWAYS AN ERROR
```



```
*****************************************************************

                LDY #$0
                LDA (STR1),Y    ;GET LENGTH OF SOURCE STRING
                CMP LEN1        ;SEE IF GREATER OR EQUAL
                BLT ERROR       ;ERROR OTHERWISE
        ;
        ; NOW, TRANSFER THE SUBSTRING
        ;
                LDA LENGTH
                STA (STR2),Y    ;INIT LENGTH
                CLC             ;SET UP POINTER TO BEGINNING
                LDA STR1        ;OF SUBSTRING
                ADC INDEX
                STA STR1
                BCC SUBST1
                INC STR1+$1
        ;
        SUBST1  INC STR2        ;INCREMENT PAST LENGTH BYTE
                BNE SUBST2
                INC STR2+$1
        ;
        SUBST2  CPY LENGTH
                BGE SUBST3
                LDA (STR1),Y
                STA (STR2),Y
                INY
                JMP SUBST2
        ;
        SUBST3  PLA
                TAY
                PLA
                PLP
                RTS
```

STRING COMPARISONS.

Probably the most important string handling tool is the ability to compare two strings to see if they are equal or not equal. The ability to see if one string is less than or greater than another string is also quite useful for such functions as alphabetizing lists and so on. These string relations are defined as follows:

1) Two strings are equal if and only if their lengths are equal and each character in the first string equals the corresponding character in the second string.

2) Two strings are not equal if either their length bytes do not match up or one of the characters in the first string does not match the corresponding character in the second string.

3) A string is less than a second string if, while traversing the string from the first character to the length of the small-

****

est string, a character is encountered in one string which is less than the corresponding character in the second string. If the lengths are not equal, and all the characters match up to the length of the shorter string, then the shorter string is considered to be less than the longer string. These requirements allow "ABC" to be less than "SUN" and to be less than "ABCD." This type of ordering is called, "lexicographical ordering," which is used in dictionaries and the like.

4) The requirements for a string to be greater than a second string are identical to the less than requirements, except you must substitute "greater than" for all the occurrences of "less than" in the preceeding paragraph.

In the subroutines which follow, tests are made for equality/inequality, less than/greater or equal, and greater than/less than or equal. In each case, the accumulator is returned with the value TRUE (i.e. $1) if the first condition is true (i.e. EQUAL / LESS THAN / GREATER THAN), or it is returned with FALSE ($0) if the second condition is true (i.e. NOT EQUAL / GREATER OR EQUAL / LESS THAN OR EQUAL). In each case, a pointer to the first string is passed in (STR1, STR1 +$1) and a pointer to the second string is passed (STR2, STR2+$1). These locations must be set up before the routine is called.

```
; STRING COMPARE #1
; TEST FOR EQUALITY
;
; THIS ROUTINE COMPUTES THE COMPARISON
;      (STR1) = (STR2)
; AND RETURNS TRUE OR FALSE IN THE ACCUMULATOR

STREQU:
        PHP             ;PRESERVE C & V FLAGS
        TYA
        PHA             ;SAVE THE Y REGISTER
;
        LDY #$0
        LDA (STR1),Y
        CMP (STR2),Y    ;COMPARE LENGTHS
        BNE NOTEQL      ;AND QUIT IF NOT EQUAL
;
; IF LENGTHS ARE EQUAL, SET UP INDICIES
; TO THE BEGINNING OF THE STRINGS
;
        STA LENGTH      ;SAVE LENGTH OF STRINGS
        INC STR1
        BNE SEQU1
```

```
*****************************************************************

                INC STR1+$1
        ;
        SEQU1   INC STR2
                BNE SEQU2
                INC STR2+$1
        ;
        SEQU2   LDA (STR1),Y    ;PERFORM COMPARISONS
                CMP (STR2),Y
                BNE SEQU3
                INY
                CPY LENGTH
                BLT SEQU2
        ;
        ;
        ; THE STRINGS ARE EQUAL
        ;
        ;
                JSR DECSTR      ;RESTORE STR1,STR2
        ;
                PLA             ;RESTORE Y & PSW REGISTERS
                TAY
                PLP
                LDA #TRUE       ;RETURN TRUE
                RTS
        ;
        ;
        ; STRINGS ARE NOT EQUAL HERE
        ;
        SEQU3   JSR DECSTR
        NOTEQL  PLA             ;RESTORE Y & PSW
                TAY
                PLP
                LDA #FALSE      ;RETURN FALSE IF NOT EQUAL
                RTS
        ;
        ;
        ;
        ;
        ; DECSTR- RESETS STR POINTERS TO THEIR ORIGINAL
        ;         VALUESVALUES

        DECSTR:
  s66           LDA STR1        ;RESTORE STRn POINTERS
                BNE SEQU4
                DEC STR1+$1
        SEQU4   DEC STR1
                LDA STR2
                BNE SEQU5
                DEC STR2+$1
        SEQU5   DEC STR2
                RTS
        ;
```

```
          ;
          ;
          ;
          ;
          ; STRING COMPARE #2
          ; TEST FOR LESS THAN
```



```
**********************************************************************

          ;
          ; THIS ROUTINE COMPUTES
          ;   STR1 < STR2
          ;
          ; ON RETURN, IF STR1 < STR2 THEN THE ACCUMULATOR IS
          ; RETURNED WITH TRUE. IF STR1 >= STR2 THEN THE
          ; ACCUMULATOR IS RETURNED WITH THE VALUE FALSE
          ;
          ;
          STRLES:
                  PHP             ;PRESERVE C & V FLAGS
                  TYA             ;SAVE Y REGISTER
                  PHA
          ;
                  LDY #$0
                  LDA (STR2) ,Y   ;COMPUTE THE MINIMUM LENGTH
                  STA MINLEN
                  CMP (STR1) ,Y
                  BGE STRLS1
                  LDA (STR1) ,Y
                  STA MINLEN
          ;
          STRLS1  INY             ;TEST LOOP
                  LDA (STR1) ,Y
                  CMP (STR2) ,Y
                  BGE NOTLES
                  CPY MINLEN
                  BLT STRLS1
                  BEQ STRLS1
          ;
          ; ALL CHARACTERS UP TO THE MINIMUM LENGTH ARE EQUAL
          ; NOW SEE IF THE LENGTH OF STR1 IS LESS THAN THE
          ; LENGTH OF STR2
          ;
                  LDY #$0
                  LDA (STR1),Y
                  CMP (STR2),Y
                  BGE NOTLES
          ;
          ; NOW STR1 < STR2
          ;
                  PLA             ;RESTORE THE Y REGISTER
                  TAY
                  PLP             ;RESTORE PSW
                  LDA #TRUE       ;TRUE BECAUSE STR1 < STR2
                  RTS
```

```
;
;
NOTLES  PLA
        TAY
        PLP
        LDA #FALSE
        RTS
;
;
;
; STRING COMPARE #3
; TEST TO SEE IF STR1 > STR2
;
```


****

```
; THIS ROUTINE COMPUTES THE RELATION:
;       STR1 > STR2
;
; THE ACCUMULATOR IS RETURNED WITH THE VALUE TRUE ($1)
; IF THE RELATION HOLDS, FALSE ($0) IS RETURNED
; OTHERWISE.
;
;
STRGTR:
        PHP
        TYA
        PHA
;
        LDY #$0         ;GET THE MINIMUM LENGTH
        LDA (STR2),Y
        STA MINLEN
        CMP (STR1),Y
        BGE SGTR1
        LDA (STR1),Y
        STA MINLEN
;
SGTR1   INY
        LDA (STR2),Y
        CMP (STR1),Y
        BGE NOTGTR
        CPY MINLEN
        BLT SGTR1
        BEQ SGTR1
;
; STRINGS ARE EQUAL UP TO THE MINIMUM LENGTH
;
        LDY #$0
        LDA (STR1),Y
        CMP MINLEN
        BEQ NOTGTR
;
        PLA
        TAY
        PLP
```

```
        LDA #TRUE
        RTS
;
NOTGTR  PLA
        TAY
        PLP
        LDA #FALSE
        RTS
;
;
;
TRUE    EQU $1
FALSE   EQU $0
STR1    EPZ $0
STR2    EPZ STR1+$2
        END
```

Once again, it will be left to the reader to implement better parameter passing techniques. These routines are presented

****

here solely as examples. It is probably more practical to pass the string addresses after the JSR as we did with the print subroutine. The string compare subroutines might be called in one of the following manners:

```
        JSR STREQU      ;IS STR1 = STR2?
        ADR STR1
        ADR STR2


        - OR -


        JSR STRLES      ;IS STR1 < STR2?
        ADR STR1
        ADR STR2


        - OR -


        JSR STRGTR      ;IS STR1 > STR2?
        ADR STR1
        ADR STR2


        ETC...
```

HANDLING ARRAYS OF CHARACTERS.

Sometimes the character strings being compared do not

have variable lengths. As such, the extra code and time required to test for the lengths of the two strings being compared is not necessary. For example, all the mnemonics used by LISA are three characters long. This means that all that has to be done is insure that the mnemonic typed in by the user is three characters in length and then compare those three characters to the character triplets in the mnemonic table. The following routine takes "NUMCHR" characters from the in buffer and compares them against characters within the table beginning at location "TABLE:"

```
NUMCHR  EQU $3          ;INIT FOR THREE-CHAR LOOK-UP
PTRSAV  EPZ $0          ;POINTER SAVE AREA
TBLADR  EPZ PRTSAV+$1   ;USED TO HOLD TABLE ADDRESS
INPUT   EQU $200        ;GETLN INPUT BUFFER
;
; THIS ROUTINE IS ENTERED WITH THE X-REGISTER POINTING
; TO THE FIRST CHARACTER TO BE COMPARED IN THE INPUT
; BUFFER (PAGE TWO)
```


****

```
; ON RETURN, THE X REGISTER POINTS TO THE FIRST CHAR
; PAST THE ARRAY OF LENGTH "NUMCHR" (DEFINED ABOVE)
;
;
;
LOOKUP:
        PHP
        TYA
        PHA
;
        STX PRTSAV      ;SAVE INDEX TO CHAR ARRAY
        LDA #TABLE      ;SET UP POINTER TO TABLE
        STA TBLADR
        LDA /TABLE
        STA TBLADR+$1
;
        LDY #$0
LOOP    LDA INPUT.X
        CMP (TBLADR),Y
        BNE NXTENT
        INX
        INY
        CPY #NUMCHR
        BLT LOOP
;
; GOOD MATCH HERE,  RETURN TRUE
;
        PLA
        TAY
        PLP
        LDA #TRUE
        RTS
;
```

```
         ;
         ; CURRENT CHARACTER ARRAY DOES NOT MATCH, SET UP INDEX
         ; TO THE NEXT ELEMENT IN THE TABLE (IF ONE EXISTS)
         ;
         NXTENT:
                 CLC
                 LDA TBLADR
                 ADC #NUMCHR
                 STA TBLADR
                 BCC NXTE1
                 INC TBLADR+$1
         ;
         ; RESTORE X REGISTER
         ;
         NXTE1   LDX PTRSAV
                 LDY #$0         ;RE-INIT Y REGISTER
         ;
         ;
         ; CHECKTO SEE IF AT END OF TABLE
         ;
         ;
                 LDA TBLADR
                 CMP #TBLEND
                 LDA TBLADR+$1
                 SBC /TBLEND
                 BLT LOOP
```



```
         ;
         ; NO MORE ENTRIES, RETURN FALSE AND LEAVE X REGISTER
         ; POINTING TO THE BEGINNING OF THE TABLE
         ;
                 PLA
                 TAY
                 PLP
                 LDA #FALSE
                 RTS
         ;
         ;
         ;
         ; SAMPLE TABLE, EACH ENTRY MUST CONTAIN "NUMCHR" NUM
      ; OF CHARACTERS (IN THIS CASE, THREE)
         ; OF CHARACTERS (IN THIS CASE, THREE)
         ;
         ;
         TABLE   ASC "ABC"
                 ASC "DEF"
                 ASC "GHI"
                 ASO "JKL"
                 ASC "MNO"
                 ASC "PQR"
                 ASC "STU"
                 ASC "VWX"
                 ASC "YZ "
```

```
        ASC "ETC"
TBLEND  EQU *
        END
```

Note that TBLEND is defined as the next available location after the table.

Table operations on the 6502 microprocessor can be handled very efficiently. Especially when the table is less than 256 bytes in length. The previous routine was written as a general purpose table look-up routine. It will work for tables of any length (representable in the 6502 memory space). For tables less than 256 bytes in length (such as the alphabet table used in the previous routine), lots of code and time can be saved by incrementing the Y-index register instead of a 16-bit memory location. Additional time can be saved by using the indexed by Y addressing mode instead of the indirect indexed by Y addressing mode. The former routine, rewritten for small tables, is:

```
NUMCHR  EQU $3          ;INIT FOR THREE-CHAR LOOK-UP
PTRSAV  EPZ $0          ;POINTER SAVE AREA
PYSAV   EPZ PRTSAV+$1   ;Y REG SAVE AREA
INPUT   EQU $200        ;GETLN INPUT BUFFER
BUFFER  EQU $300        ;BUFFER SAVE AREA
;
; THIS ROUTINE IS ENTERED WITH THE X-REGISTER POINTING
```



```
;*****************************************************************

; TO THE FIRST CHARACTER TO BE COMPARED IN THE INPUT
; BUFFER (PAGE TWO)
; ON RETURN, THE X REGISTER POINTS TO THE FIRST CHAR
; PAST THE ARRAY OF LENGTH "NUMCHR" (DEFINED ABOVE)
;
;
;
LOOKUP:
        PHP
        TXA
        PHA
        TYA
        PHA
;
; TRANSFER INPUT TO BUFFER SAVE AREA
;
        LDY #$0
LOOP    LDA INPUT,X
        STA BUFFER,Y
        INX
        INY
        CPY #NUMCHR
        BLT LOOP
;
; NOW, COMPARE BUFFER SAVE AREA TO DATA IN TABLE
```

```
;
        LDX #$0
        LDY #$0
LOOP0   LDA BUFFER,X
        CMP TABLE,Y
        BNE NXTENT
        INY
        INX
        CPX #NUMCHR
        BLT LOOP0
;
; A MATCH IS FOUND HERE
;
        PLA
        TAY
        PLA
        TAX
        INX
        INX
        INX             ;LEAVE POINTING AT NEXT CHAR
        PLP
        LDA #TRUE
        RTS
;
;
; INCREMENT TO THE NEXT EXTRY (IF IT EXISTS)
;
NXTENT  CPX #$2
        BGE NXT1
        CPX #$1
        BGE NXT2
        INY
```


****

```
NXT2    INY
NXT1    INY
        LDX #$0
        CPX TBLENG
        BLT LOOP0
;
; END OF TABLE HAS BEEN REACHED
;
        PLA
        TAX             ;LEAVE X REG POINTING TO CHARS
        PLA
        TAY
        PLP
        LDA #FALSE      ;STRING NOT FOUND
        RTS
;
;
; SAMPLE TABLE, EACH ENTRY MUST CONTAIN "NUMCHR" NUM
; OF CHARACTERS (IN THIS CASE, THREE)
;
```

```
          ;
          ;
          TABLE   ASC "ABC"
                  ASC "DEF"
                  ASC "GHI"
                  ASC "JKL"
                  ASC "MNO"
                  ASC "PQR"
                  ASC "STU"
                  ASC "VWX"
                  ASC "YZ
                  ASC "ETC"
          TBLENG  EQU *-TABLE
                  END
```

Note that a table length "TBLENG" is used instead of the end of table pointer. Remember the Y-register is only eight bits long.

Obviously, there are many different ways to compare strings against other strings, be they in tables or whatever. This book is not attempting to cover all possible cases (an impossible task), but rather, to cover a few cases which may be of general interest. The techniques used in the preceeding examples can be applied to other methods of comparing string data. Hopefully, these examples have been somewhat of an awakening so that you can go out and write your own string handling functions.

****

CHAPTER 15

SPECIALIZED I/O

APPLE I/O STRUCTURE.

One of the reasons the APPLE II computer is so popular is its powerful I/O structure. The APPLE II computer was the first personal computer to feature the "Game I/O" connector with analog inputs and digital input and output lines. Although intended primarily for games and entertainment purposes, the game I/O lines have been utilized for such things as printer interfaces, RS-232 lines, and even industrial controllers. A myriad of peripherals, including paddles, joysticks, light pens, and color guns, have

****

been interfaced to the game I/O connector. In all, the game I/O connector makes the APPLE II computer one of the most flexible computers around.

To understand the flexiblity of the game I/O connector, it is

necessary to first describe what types of I/O are available at the game I/O connector. First, there are three "flag" (or pushbutton) inputs. There are four "annunciator" outputs. There are four 8-bit analog-to-digital inputs which can measure a resistance between 150 ohms and 150K ohms. And finally, there is a utility strobe line available on the game I/O connector.

In addition to the I/O on the game I/O connector, there are some other specialized I/O devices available on the APPLE II computer. These include the built-in speaker, the cassette input and output, and the Apple keyboard. There are also several video display modes available to the user including LORES and HIRES graphics, with or without four lines of text at the bottom of the page. Controlling these display modes, as well as the I/O on the game I/O connector, is a simple matter of accessing memory locations within the APPLE II computer's memory space.

These memory locations all fall within the 128 bytes in the $C000 to $C07F range. For instance, we've already encountered the Apple keyboard whose input can be obtained at location $C000. If bit seven of location $C000 is set, then a key has been pressed on the Apple keyboard. If bit seven is clear, then no key has been pressed and the program must wait for a key to be pressed if the program requires input. In order to clear bit seven of the keyboard location after the desired data has been retrieved (so that the next time $C000 is accessed you won't read the same key code again), location $C010 must be accessed. Accessing location $C010 clears bit seven of location $C000 so that another key can be read from the Apple keyboard. The following routine works fine as a keyboard input routine:

```
KEYIN   LDA $C000
        BPL KEYIN       ;IF NO KEY PRESSED, LOOP BACK
        STA $C010       ;CLEAR BIT #7 OF THE KEYBOARD
        RTS
```

KEYIN, when called, waits until a key is pressed and then returns with the ASCII code of the key pressed in the accumulator. The accumulator is stored into location $C010 to clear bit seven of the keyboard for reasons previously mentioned.

****

Sometimes, it is useful to access the keyboard just to see if a key has been pressed. The BIT instruction comes in very handy here. By BIT'ing location $C000, the N flag will be set if a valid key has been pressed. The BMI/BPL instructions may then be used to test to see if a key has been pressed. Another interesting subroutine which is useful on occassion is "KEYPRS." KEYPRS returns the value TRUE if a key has been pressed, and it returns the value FALSE if a key has not been pressed. It is coded in the following obscure fashion:

```
        ; FUNCTION KEYPRS.  RETURNS TRUE IF A KEY HAS BEEN
        ; PRESSED, FALSE OTHERWISE. THIS VALUE IS RETURNED
```

```
; IN THE ACCUMULATOR.
;
;
KEYPRS:
        LDA $C000
        ROL             ;SHIFT SIGN BIT (#7) INTO
        ROL             ;THE LO. BIT OF THE ACC.
        AND #$1         ;MASK OUT ALL BUT BIT #0.
        RTS
```

In this routine, bit seven is shifted into the carry and then back into bit zero of the accumulator. The accumulator is then AND'ed with $1 so that only bit zero is left in the accumulator. If a key has been pressed, the result of the AND #$1 is one. If a key has not been pressed, the result of the AND #$1 is zero.

Location $C020 is the cassette output toggle. Normally this output is used as an interface to the audio cassette mass storage unit connected to diskless Apples. This output can, however, be connected to the high-level input of any stereo or sound system, and you can use the cassette output toggle in the same manner as you would the built-in Apple speaker. Location $C030 is the Apple speaker. Since the cassette output and the Apple built-in speaker are treated in a similar manner, the discussion which follows will apply to both.

Sound is generated by causing the Apple speaker to move outward and then back inward. Each time the speaker goes in and out, one cycle is produced. Humans can hear sound from approximately 20 cycles per second (also called, "hertz" in the engineering field) to about 20,000 cycles per second. Theoretically, if you were to set up a time delay loop that caused the speaker to move outward and then back inward at some rate

****

between 20 Hz (Hz is an abbreviation for Hertz) and 20,000 Hz (or 20KHz), you should be able to produce an audible tone. Unfortunately, since the 2-inch speaker supplied with the APPLE II computer is not exactly a high fidelity unit, the theoretical maximum is unobtainable. Typically, tones in the range 60 Hz to about 10,000 Hz can be reproduced satisfactorily on the built-in 2-inch speaker. It should be noted that, if you connect your cassette output jack to a good stereo system, this problem is alleviated. One last thing. To cause the speaker (or cassette output) to toggle between the outward and the inward position, simply use a load instruction to access location $C030 or location $C020 (for the cassette output). The load instruction can be LDA, LDX, LDY, BIT, ADC, AND, CMP, CPX, CPY, or ORA. Store instructions absolutely will not work. This is due to the way in which the 6502 writes data to a memory location. First, while writing to a memory location, the 6502 READS the memory location, then it writes to it. These two operations occur about 92 nanoseconds apart. If you try to store to the speaker or cassette outputs, the following will

happen. During the write operation the 6502 will read the memory location. This causes the speaker or cassette output to toggle outward (for instance). 92 nanoseconds later the 6502 writes to the same memory location accessing it again. This causes the speaker or cassette output to toggle back to the position it was in before the store type instruction was executed. Even the finest stereo gear in the world (and especially not the "massive" 2-inch speaker provided with the APPLE II computer) can respond to a pulse 92 nanoseconds wide. As a result, absolutely nothing will happen. Long before the speaker ever gets a chance to move outward, the 6502 tells it to move back inward. As a result, the speaker does not move at all and no sound is produced.

Location $C040 accesses the utility strobe on the game I/O connector. Loading from this location causes a single pulse on pin 5 of the game I/O connector. Storing to this address causes two pulses to be generated (see the discussion above). Unfortunately, the discussion of hardware interfacing is beyond the scope of this book, and since the use the $C040 strobe requires some hardware interfacing, an in-depth description of the $C040 strobe is not possible.

Locations $C050 to $C057 are used to switch among the various display modes. Location $C050, when accessed, sets the

****

graphics mode. Location $0051 does just the inverse, it sets the text mode. The text mode is available in two forms: primary page and secondary page text. The primary text page resides in memory from location $400 through location $7FF. The secondary text page resides in memory from location $800 through location $BFF. Location $C052 sets the no mix (or full graphics) mode. Accessing this location produces visible results only if the APPLE II computer is currently in the graphics mode. In the text mode, accessing location $C052 produces no visible effect. Location $C053 is used to set the mixed graphics mode. In this mode, four lines of text are displayed at the bottom of the screen. Obviously, this mode is valid only when graphics are in effect. Location $C054 selects the primary display page. For the text page and LORES graphics, the memory area which will be utilized is $400 thru $7FF. For HIRES graphics, locations $2000 thru $4000 will be used. Accessing location $C055 selects the secondary display page. This is $800 thru $BFF for text and LORES graphics, $4000 through $7FFF for HIRES graphics. Accessing location $C056 sets up the APPLE II computer for LORES GRAPHICS. The graphics mode must also be set for this to take effect. Accessing location $C057 sets up the APPLE II computer HIRES graphics. Once again, the graphics mode must be set (location $C050) before HIRES graphics will be displayed.

Locations $C058 through $C05F are used to control the annunciator outputs. These are TTL outputs and will require buffers if they are to be used to drive current requiring devices such as L.E.D.'s. Annunciator zero (AN0) is set to the off position by

accessing location $C058. AN0 is set to the on position by acces-
siog location $C059. AN1 is set to the off position by accessing
location $C051 and is set to the on position by accessing location
$C05B. AN2 is set to the off position by accessing location $C050
and is set to the on position by accessing location $C05D. AN3
is turned off by accessing location $C05E and turned on by
accessing location $C05F. Sadly, there is no way to determine
the status of the annunciator outputs. Either always be sure of
yourself, or store the current value in some memory location for
future reference.

     Location $C060 is a very interesting input port. It is the input
bit for the cassette I/O port. You're probably wondering why the
cassette input port is so useful. After all, isn't the disk much better

****

than the audio cassette for mass storage? Well the disk is cer-
tainly much better for mass storage (yes, you can sleep easy on
that tonight), but the audio cassette input allows you to perform
something which the disk could never do. It allows you to input
and digitize speech and other natural sounds. The following rou-
tine can be considered something of a "teaser." If you connect a
crystal microphone (or other high-output microphones) to the cas-
sette input jack and run the following routine -- lo and behold, what
goes into the microphone comes out of the speaker. Try it!

```
          LOOP    LDA $C060       ;TEST CASSETTE INPUT PORT
                  BPL LOOP
                  LDA $C030       ;TOGGLE SPEAKER
          LOOP2   LDA $C060
                  BMI LOOP2
                  LDA $C030
                  JMP LOOP
                  END
```

Beyond this basic loop it is possible to get the data from the
cassette input, pack it, and store it into successive memory loca-
tions so that it can be saved to disk and output at a later date.
Several experiments in speech synthesis and speech recognition
can be performed without spending an extra nickel for additional
hardware (except, of course, for a cheap microphone to plug into
the back of the APPLE II computer).

     Locations $C061, $C062, and $C063 are used to detect the
pushbutton inputs (PB1 =$C061, PB2=$C062, PB3=$C063).
If a pushbutton is pressed then bit seven of its corresponding
location is set (i.e. "1"). If the pushbutton is not pressed then bit
seven will be reset (i.e. "0"). The following code tests the push-
buttons and beeps the speaker (by printing the bell character) if
the pushbutton is pressed.

```
          LOOP    BIT PB1
```

```
                BPL LOOP
                LDA #BELL       ;LOAD BELL CHARACTER INTO ACC
                JSR COUT1       ;OUTPUT BELL CHARACTER
                JMP LOOP
PB1     EQU $C061
COUT1   EQU $FDF0
BELL    EQU $87
                END
```


****

Locations $C064 through $C067 correspond to the game controller (analog) inputs. The analog inputs work in the following manner. First, you must initialize the hardware by accessing location $C070. This causes a little timing device to start running. This timing device (a 558 timer, in case you're wondering) is connected to bit seven of location $C064, $C065, $C066, or $C067 (depending upon which game controller you're interested in). While this 558 timer is running, bit seven of the corresponding controller location is set, so that by forming a little counter loop it is possible to determine the setting of the desired analog input. The following routine (straight out of the Apple monitor) reads game paddle #x where x is passed in the X-register. Upon return, the Y-register contains a value in the range $0 to $FF depending upon the setting of the game controller.

```
PREAD   LDA $C070       ;TRIGGER PADDLES
        LDY #$0         ;INIT COUNT
        NOP             ;DELAY REQUIRED FOR HARDWARE
        NOP             ;PURPOSES
PREAD2  LDA $C064,X     ;TEST DESIRED PADDLE
        BPL RTS2D
        INY
        BNE PREAD2      ;QUIT IF > $FF
        DEY             ;SET TO $FF
RTS2D   RTS
        END
```

There are two things to keep in mind when using the analog inputs. First, you cannot read two paddle inputs immediately after one another. Due to the hardware used, you must delay a little while before reading another input. The loop:

```
        LDX #$0
LOOP    DEX
        BNE LOOP
```

works just fine.

The second thing to keep in mind is that reading the paddle inputs does take some time. You should be aware of this if you are writing time critical code and are using the paddles.

Apple's built-in I/O is very useful for games and measurement purposes. Many programs, games or not, can be improved

****

greatly by accepting input from the paddles or input switches. Programs such as "SLOW LIST," "CURSOR EDITING," are all enhanced by the use of the game controller inputs.

****

CHAPTER 16

AN INTRODUCTION TO SWEET-16

SWEET-16

Deep inside the Integer BASIC ROMs lives a mysterious program known as "Sweet-16." Sweet-16 is a meta processor which is implemented interpreter style. Its virtues include a bunch of 16-bit instructions, most of which are implemented with one-byte opcodes. Since performing 16-bit operations with normal 6502 code requires several two- and three-byte instructions, Sweet-16 code is very compact. In this chapter we will explore

****

the possibilities of the Sweet-16 interpreter, its advantages and disadvantages.

First, just exactly what is a "meta processor" and what does an interpreted implementation imply? A meta processor is simply a fantasy machine, one which does not exist as a physical machine, but simply as a design tool. A meta processor has the capability of taking on almost any instruction set. Since there are only a few pieces of hareware actually capable of performing this task (and the 6502 is not such a piece of hardware), a meta processor implementation must be handled in a somewhat different way on the 6502. An interpreter must be written, with a single subroutine for each instruction code to be implemented. A small control program picks up the Sweet-16 opcodes from memory, decodes the instruction, and then passes control to the appropriate subroutine. Once the desired subroutine is finished execution, the code control is returned to the control program which accesses the n byte of Sweet-16 code and continues the process.

So far everything sounds wonderful. But what are the dis-

advantages of Sweet-16 code? First, and probably most important, Sweet-16 programs run much slower than the same algorithm coded entirely in 6502 assembly language, five to seven times slower in fact. Another mark against Sweet-16 code is that the Sweet-16 interpreter exists only in the Integer BASIC ROMs (which is no big deal if you have an APPLE II computer, a language card, or an Integer BASIC card), but, if you only have an APPLE II Plus computer without Integer BASIC, or you wish to sell your programs to others who may not have the Integer BASIC language, you will either have to forget about Sweet-16 altogether or inject the code for the Sweet-16 interpreter into your program. Since the Sweet-16 interpreter is about 400 bytes long, you would have to write more than one kilobyte of code in Sweet-16 before it would pay to include the interpreter within your programs. Because of this problem, Sweet-16 should only be used where the Integer BASIC language is available. The interpreter is already provided there for you (free-of-charge even!).

What does Sweet-16 look like? Sweet-16 is a 16-bit computer complete with sixteen 16-bit registers. These registers are used to hold addresses and intermediate values for use in address calculations. These registers are numbered R0 to RF

****

(hex) for reference purposes. Several of these registers are special purpose. They include R0, RC, RE, and RF. R0 is the Sweet-16 accumulator. Sweet-16 can only perform the addition, subtraction, and comparison operations, and these must all be routed through the Sweet-16 accumulator. RC is the Sweet-16 stack pointer used when Sweet-16 subroutines are called. RE is used to hold the Sweet-16 processor status data and RF is the Sweet-16 program counter. Except for these four registers which are for special use only, all the Sweet-16 registers are general purpose address registers.

Before discussing how the Sweet-16 instruction set is used, entering and exiting the Sweet-16 mode must be covered. A program toggles back and forth between Sweet-16 code and 6502 code in much the same manner as you would toggle between the decimal mode and binary mode. A program enters the Sweet-16 mode with a JSR SW16 instruction. SW16 is located at address $F689. Once this is accomplished, all further code is assumed to be Sweet-16 code. To terminate the Sweet-16 mode of operation, the Sweet-16 instruction "RTN" (for ReTurN to 6502 mode) must be execute immediately after the RTN instruction, valid 6502 instructions are expected. A quick excursion into Sweet-16 with an immediate return to 6502 mode would consist of the code sequence:

```
SW16    EQU $F689
;
        JSR SW16
        RTN
;
```

```
                RTS
                END
```

If this short program were executed, the JSR SW16 instruction would cause a transfer to the Sweet-16 mode to take place. All further instructions are assumed to be Sweet-16 instructions. The next instruction is the Sweet-16 RTN instruction which causes a transfer back to the 6502 mode. All instructions following the RTN instruction are assumed to be valid 6502 instructions. The next instruction is the familar 6502 RTS instruction which causes a return to the Apple monitor. This simple sequence of instructions, although trivial and producing no noticeable results, demonstrates how to enter and terminate the Sweet-16 mode. Normally,

****

several Sweet-16 instructions would be sandwiched between the JSR SW16 and the RTN instructions.

The Sweet-16 processor status word holds several conditions. A carry flag, zero flag, and negative flag are implemented. A test for minus one ($FFFF) is also implemented.

The Sweet-16 SET instruction allows the programmer to set the contents of any Sweet-16 register to a desired value. Its 6502 equivalent is the load immediate instruction. The SET instruction has the syntax:

```
                SET Rn,<16-BIT VALUE>
```

The 16-bit value can be any valid LISA address expression. 'n' is simply a hex value in the range $0-$F and denotes which register is to be loaded with the declared value. Examples of the SET instruction:

```
          LABEL   SET R0 LABEL    ;LOADS THE CURRENT ADDRESS
                                  ;INTO R0
                  SET R1,$25      ;LOADS $0025 INTO R1
                  SET R5,$800     ;LOADS $0800 INTO R5
```

The SET instruction is three bytes long: one byte for the SET opcode and two bytes for the 16-bit value that is to be loaded into the specified register. SET RF,<VALUE> is a very special case. Since RF is the Sweet-16 program counter, loading immediate data into register $F is the same as performing an absolute jump instruction. RC and RE must be treated carefully as well since they are used to hold the Sweet-16 stack pointer and status register. If zero is loaded into the specified register, the Sweet-16 zero flag is set; otherwise it is cleared. If minus one ($FFFF) is loaded into the specified Sweet-16 register, the minus one flag is set; otherwise the minus one flag is cleared. The Sweet-16 carry flag is always cleared after a SET instruction is executed.

The next instruction in the Sweet-16 instruction set is the load register or LDR instruction. This instruction loads the Sweet-16 accumulator (R0) from the register specified in the operand field. The term 'load' is somewhat misleading as this instruction really a register transfer instruction not unlike the 6502 TYA and TXA instructions. The LDR instruction has the syntax:

```
LDR Rn
```


****

Where n is the Sweet-16 register number in the range $0-$F Note that LDR R0 is perfectly allowable and performs the operation of making a copy of R0 into R0, a somewhat useless instruction (except, possibly, for comparison purposes) but nevertheless valid. The LDR instruction is a one-byte instruction and will cause 16 bits to be transferred to the Sweet-16 accumumator. If zero is transferred between the registers, then the Sweet-16 zero flag is set; otherwise the zero flag is cleared. If minus one is transferred to the accumulator, the minus one flag is set; otherwise the minus one flag is cleared. The negative flag is set according to the data transferred to the Sweet-16 accumulator. The negative flag always reflects the contents of the sixteenth bit, not the eighth bit as in the 6502 status register. The Sweet-16 carry flag is always cleared.

STO (store register) is the inverse operation to LDR. STO stores the contents of the Sweet-16 accumulator into the specified Sweet-16 register. This is similar to the 6502 instructions TAY & TAX. The Sweet-16 status bits are affected in the same manner as with the LDR instruction, and the STO instruction is one byte long, just like the LDR instruction.

You will note that there is no direct way to transfer the data from one register to another without going through the Sweet-16 accumulator. For example, to transfer the data from R5 to R6 you must execute the code sequence:

```
LDR R5
STO R6
```

As you can see, the Sweet-16 accumulator is destroyed during such transfers. For this very reason, the Sweet-16 accumulator should not be used to hold important data. It should be used totally as a transient register used only for calculations.

The Sweet-16 interpreter allows two types of arithmetic. 16-bit addition and subtraction. Addition is performed with the Sweet-16 ADD instruction. It takes a single register as its operand. This register is added to the Sweet-16 accumulator and the result is left in the accumulator. The syntax for the ADD instruction is:

```
ADD Rn
```

Where n is a hex value in the range $0-$F. Note that the instruction 'ADD R0' is very useful; it doubles the value in the Sweet-16

****

accumulator. If there is a carry out of the 17th bit during the addition, the carry is noted in the Sweet-16 carry flag. An add with carry instruction is not possible, so the carry flag is useful only for detecting overflow. All the other condition codes are set according to the outcome of the addition operation. The Sweet-16 ADD instruction is a one-byte instruction.

Subtraction is performed using the Sweet-16 SUB instruction. The register specified in the operand field is subtracted from the accumulator with the results being left in the accumulator. The SUB instruction can be used as a compare instruction in a manner similar to the SBC instruction on the 6502. If the value in the accumulator (prior to the SUB instruction) is greater than or equal to the value in the specified register, the carry flag will be set after the SUB instruction occurs. If the value in the accumulator is less than the value in the specified register, the carry flag will be clear after the SUB instruction is executed. If the two registers are equal, then the zero flag is set; if they are not equal, the zero flag is reset. Note that the SUB R0 instruction can be used as a one-byte clear accumulator instruction. It performs the same function as SET R0,0 yet requires only one third the memory.

Comparisons can also be performed using the CPR (compare register) instruction. CPR performs the same function as the SUB instruction, except that the results are placed in RD instead of the ACC. Any tests following the CPR instruction will test the value in RD instead of the accumulator. Register RD can be thought of as an auxiliary processor status register. As such, its use should also be avoided.

Conditions in the Sweet-16 processor status register are tested in a manner very similar to the 6502 microprocessor. That is, branch instructions are used to test conditions. Branches on the Sweet-16 processor use relative addressing, just like their 6502 counterparts. The branch instructions include: BRA (branch always, an unconditional branch), BNC (branch if no carry), BIC (branch if carry), BIP (branch if positive), BIM (branch if minus), BIZ (branch if zero or branch if equal), BNZ (branch if not zero or not equal), BM1 (branch if minus one), BNM (branch if not minus one), and BSB (branch to Sweet-16 subrouuine). All Sweet-16 branches are two bytes long.

The branch to subroutine (BSB) instruction really needs some additional explanation. When a Sweet-16 subroutine is

****

called, the return address is pushed onto the Sweet-16 return address stack. The stack pointer is RC. Wherever RC happens to be pointing when the BSB instruction is executed, the return address will be stored. If you have not initialized the Sweet-16 stack pointer (RC), it could be pointing anywhere in memory, which means that a BSB instruction could potentially wipe out valuable program and data storage.

The cure for these ailments is always to initialize the Sweet-16 stack pointer prior to using Sweet-16 subroutines. This is accomplished quite easily by using the SET instruction and loading RC with an intial stack pointer value (this is similar to using the 6502 sequence: LDX #VALUE, TXS). Unlike the 6502 stack pointer which is an 8-bit register that wraps around, the Sweet-16 stack pointer is a 16-bit register which can take on any 16-bit va!ue. This means that if you're not very careful, it is possible to have the stack go wild and wipe out everything in memory. Typically, you will not have to even use Sweet-16 subroutines, but should the need arise, be very careful.

To return from a Sweet-16 subroutine you must use the RSB (return from subroutine) instruction. The RSB instruction is a single byte instruction.

Register increments and decrements are performed by the INR and DCR instructions. INR increments the register specified in the operand field by one; DCR decrements the specified register by one. All branch conditions are set to reflect the final results in the specified register. The INR and DCR instructions are both one byte long.

So far, only a discussion of the arithmetic and conditional testing capabilities of the Sweet-16 processor have been presented. Although these instructions are useful, they do not really present anything new that was not already available in the 6502 microprocessor instruction set. Sweet-16's real power comes from its pointer and data movement capabilities. Several powerful load and store instructions are available which allow the programmer to perform certain actions in one byte that would take eight to sixteen bytes on the 6502. These instructions revolve around the idea of loading the Sweet-16 accumulator indirectly through a specified register.

The first instruction in this family of instructions is the load indirect instruction. It uses the syntax:

****

LDR @Rn

Note that the mnemonic is the same as the normal load register instruction, but that the '@' character appears in the operand field immediately before the register specifier. This instruction is an 8-bit load instruction. It loads the low-order bits of the Sweet-16 accumulator from the memory location pointed to by the specified

register. The high-order byte of the Sweet-16 accumulator is cleared. After the accumulator is loaded with the data from the address pointed to by Rn, Rn is incremented by one. This causes the pointer register to point to the next available byte immediately after the LDR instruction is executed. This type of instruction (where the register is automatically incremented for you) is called an "auto-increment" instruction. The LDR indirect instruction is very useful for memory movements and searches. Consider the following code:

```
START   JSR SW16
        SET R1,$8000
        SET R3,$FF
LOOP    LDR @Rn
        CPR R3          ;CHECK FOR $FF
        BNZ LOOP        ;LOOP IF NOT FOUND
        RTN             ;QUIT SWEET-16, DATA FOUND
                        ;ADDRESS LEFT IN R1
```

This routine starts at location $8000 and searches diligently until a $FF is encountered.

To load two bytes into the accumulator one would use the LDD (load double indirect) instruction. It uses the syntax:

```
LDD @Rn
```

It loads the low order accumulator byte from the location pointed at by Rn; then Rn is incremented by one. After the increment is performed, the high order accumulator byte is loaded indirectly through the new value in Rn. Once this is accomplished, Rn is again incremented. The net result is that the Sweet-16 accumulator is loaded indirectly from the locations pointed at by Rn and Rn+1. Afterwards Rn is incremented twice. The branch condi-

****

tions will reflect the final accumulator contents and the carry will be cleared.

Data can also be stored indirectly through one of the registers. The store indirect instruction is the inverse of the load indirect instruction. It has the syntax:

```
STO @Rn
```

This instruction stores the contents of the low-order byte of the Sweet-16 accumulator at the location in memory pointed to by the Rn register. After the store operation is performed, Rn is

incremented by one. The branch conditions reflect the Sweet-16 accumulator contents. The store indirect instruction can be used rather well with the load indirect instruction for memory movement routines. The following routine moves data from $8000 through $9000 to the area $3000 through $4000:

```
START   JSR SW16
MOVE    SET R1,$8000    ;SET UP POINTER REG #1
        SET R2,$9000    ;SET UP FINAL VALUE REG
        SET R3,$3000    ;SET UP POINTER REG #2
;
LOOP    LDR @R1         ;GET DATA @R1


        STO @R3         ;STORE @R3


        LDR R1
        CPR R2          ;DONE YET?
        BNC LOOP        ;IF NO CARRY (I.E. LESS THAN)
        BIZ LOOP        ;IF EQUAM
        RTN
        BRK
        END
```

Compare this to the amount of code required to perform the same operation in 6502 machine code!

To store both halves of the Sweet-16 accumulator into memory, you must use the STD (store double indirect) instruction. This instruction stores the low-order byte of the Sweet-16 accumulator at the location pointed to by Rn. Rn is then incremented by one, and the high-order byte of the accumulator is then stored at the new location pointed to by Rn, after which Rn is again incremented by one.

****

The last three Sweet-16 instructions are POP (pop indirect), STP (store pop indirect), and PPD (pop double indirect). POP loads the low-order accumulator byte from the location pointed to by Rn AFTER Rn is decremented by one. POP has the syntax:

```
POP @Rn
```

User-defined stacks may be implemented using the POP Rn and STO Rn instructions (where Rn is the stack pointer). POP is also useful in implementing the "move right" routine presented elsewhere in this book.

STP is the inverse of POP. This operation causes the low-order byte of the Sweet-16 accumulator to be stored at the address pointed to by Rn after Rn is decremented by one. Single byte user-defined stacks may also be implemented using the STP

Rn and LDR Rn instructions (where Rn is the user-defined stack pointer).

PPD (pop double indirect) is the 2-byte equivalent of POP PPD performs the following action: Rn is decremented by one and the high-order accumulator byte is loaded from the location pointed to by Rn. Rn is then again decremented by one and the low-order accumulator byte is loaded from the address pointed to by Rn. PPD has the syntax:

PPD @Rn

Double byte stacks may be implemented using the PPD and STD instructions. The POP, STP, and PPD instructions are all one byte long. The carry is always cleared after one of these operations is performed. POP always results in a positive value which is never minus one. PPD and STP affect the status bits depending upon the final accumulator contents.

SWEET-16 HARDWARE REQUIREMENTS.

All of the Sweet-16 registers are implemented as zero page memory locations (in fact, the first 32 bytes of zero page are used for the Sweet-16 registers). For this reason, care must be exercised when using zero page memory in a program in which Sweet-16 is also used. R0 corresponds to memory locations $0 and $1; R1 corresponds to memory locations $2 and $3; and so on for the other registers. Since they are implemented in zero

****

page memory, it is a simple matter for 6502 programs to pass data to a Sweet-16 routine simply by shoving data into the respective registers. Likewise, Sweet-16 can return data to the 6502 program in the Sweet-16 registers. A Sweet-16 call is transparent to the 6502 program. All registers, including the processor status register, are preserved and then restored before returning to the 6502 mode. Another important fact to remember is that the 6502 must be in the binary (as opposed to decimal) mode before entering the Sweet-16 mode. Strange things happen if this is not the case. Obviously, another book the size of this one could be written on programming in Sweet-16. The purpose of this chapter is only to acquaint the user with the Sweet-16 interpreter. It is left to the reader to discover its myriad of uses.

****

CHAPTER 17

DEBUGGING 6502 MACHINE LANGUAGE PROGRAMS.

GENERAL.

Except for the most trivial of programs, very few programs run correctly the first time. Fortunately, LISA is an interactive assembler, so the amount of time required to correct syntax errors is reduced tremendously. Correcting the remaining program/syntax/addressing mode errors is usually quite trivial. That is, if you run across a duplicate label, or discover that you have used an absolute label where a zero page variable is required, the correction is usually quite straight forward and easy to accomplish. The real problem, as with any programming language, occurs when logical errors creep into a program. Common examples of logical errors might include forgetting to reset the decimal flag after a series of decimal operations, executing data as a program, wiping out a program with data, and forgetting to save the registers upon entering a subroutine. Unlike BASIC the 6502 is not nice enough to stop and print an offending error message. It goes along on its merry way producing incorrect results, destroying the program in memory and possibly even data in memory and on diskette.

Fortunately, the APPLE II computer is blessed with some very good assembly language debugging tools. Foremost is the Apple monitor. You've probably never even thought about the monitor as a debugging tool, but it is a very, very good one.

****

Packed into only two kilobytes of ROM is a disassembler, a software emulator, register and memory display and modify routines, memory move and verify commands, and a whole host of additional commands. If you have an APPLE II computer with Integer BASIC you also get a mini-assembler which allows you to create "quickie" programs and to patch existing ones with ease. In addition to the Apple monitor, there are several LISA support packages available from On-Line Systems which assist in the debugging of 6502 machine language programs. Their use will be described later.

GO COMMAND (G).

The Apple monitor GO command ('G') probably does not seem like a debugging command. After all, the go command is used to start a program executing and that's about all its good for, right? WRONG! The GO command acts just like a JSR statement within a program. It causes a jump to a subroutine at the address specified by the user. For example, 800G causes a jump to the subroutine at location $800 in memory. Just like a JSR statement, the return address is pushed onto the stack and a jump is made to the specified address. Wait a second! What return address is pushed onto the stack? The return address back to the Apple monitor, of course. This means that if you execute a program terminated with a RTS instruction, your program will return to the Apple monitor command level when the program

terminates. Ok, that's probably old news too. You've been sticking RTS instructions at the end of your programs for ages now and you're quite aware of the fact that such programs return back to

****

the monitor. Of what use is this feature when debugging programs?

If you stick a RTS instruction at the end of your program, then your program can be called as a subroutine from an outside program, so that, in reality, what your program consists of is one large subroutine. Since you can call this large subroutine from the Apple monitor and execute it, what is there to stop you from calling smaller subroutines within your program using the GO command? Nothing at all. And that is how the GO command becomes a very powerful debugging command: it allows you to test individual subroutines under isolated conditions. For example, suppose you have a main program that calls subroutines at locations $980, $AC0, and $1000. When you run the program the machine disappears on you and nothing happens. You can call the three subroutines using the 900G, AC0G, and 1000G commands to see which subroutine is ending up in some sort of loop. If the subroutine returns control to the monitor, chances are it works fine. If it does not come back, you know where part of your troubles may lie. Note that this technique assumes that absolutely no data is passed to the subroutine. If your subroutines require data (and most do), read on...

INITIALIZING REGISTERS AND MEMORY.

Except for the simplest of subroutines, most routines require data of some sort. Routines which require small amounts of data usually pass this data in one of the 6502 registers. Routines requiring more data can pass the data in a known location, on the stack, or can pass a pointer to the data. No matter how the data is passed, this data must be correctly set up before the subroutine is called, if the subroutine is expected to perform correctly. If you call a subroutine requiring data using the monitor GO command, chances are problems will develop unless you have taken the time to set up the parameters correctly.

The simplest method for passing parameters consists of passing the data in one or more of the 6502 registers. For example, the video output routine at location $FDF0 expects the character to be printed on the screen to be passed to the routine in the 6502 accumulator. How can you specify from the monitor what data is passed to a routine in the registers when the GO command is issued?

****

The control-E command in the Apple monitor allows you to display the contents of the 6502 registers. If you type control-E followed by return the APPLE II computer will display something like:

A=0A X=FF Y=D8 P=B0 S=G8

This tells you that when the GO command is issued, the 6502 registers will contain their respective displayed value. Great, so we know how to find out what data will be passed to a routine. But how can we change it? As it turns out, whenever you type control-E followed by return the monitor is set up so that if you type a colon (:) followed by some byte data, you can modify the registers.

EXAMPLE:

*control-E

A=0A X=FF Y=D8 P=B0 S=F8

*:C1

*control-E

A=C1 X=FF Y=D8 P=B0 S=F8

You can easily specify the data, which is to be passed to the program in the accumulator, by simply typing a colon followed by the data you wish to store in the accumulator.

If you wish to change the X- and Y-registers as well, simply type three successive bytes separated by spaces. The first byte will be placed in the accumulator, the second byte will be placed in the X-register, and the third byte will be placed in the Y-register. If you wish to change only one register, the data contained in previous registers must be retyped into the monitor.

EXAMPLE:

*control-E

A=C1 X=FF Y=D8 P=B0 S=F8

****

*:C1 FF D8 B0 FF

```
*control-E

A=C1 X=FF Y=D8 P=B0 S=FF
```

To use this as a feature when debugging your programs consider once again the Apple monitor video output routine at location $FDF0. It requires that data be passed to it in the 6502 accumulator. If we had just written the video output routine and we wanted to check it out without running the whole Apple monitor, we could accomplish this by using the following steps:

```
*control-E

A=0A X=FF Y=D8 P=B0 S=F8

*:C1

*FDF0G
A
*control-E

A=C1 X=FF Y=D8 P=B0 S=F8

*:C2

*FDF0G
B
*control-E   etc.
```

so it is possible to "spoon feed" subroutines which require data to be passed to them through one of the 6502 registers.

If data must be passed to a subroutine in one of the memory locations in the 6502 address space, the setup is only a little different than if the data is passed in a 6502 register. Rather than typing control-E <return> and then a colon followed by the register data, all you need type is the address of the parameter followed by a colon and the data you wish placed in that memory location.

****

EXAMPLE:

```
*F0:00 80 C0

*800G
```

This example assumes that there is some subroutine at location $800 which uses the data in locations $F0, $F1, and $F2. If you have a subroutine that needs data passed to it in memory, you can handle its testing in a similar manner.

If your routine requires special parameter handling (such as the PRINT routine presented in an earlier chapter, where the data is passed as part of the code stream), it is usually easier to write a small driver routine to set up the parameters and call the routine for you. For example, to write a short driver routine for the PRINT subroutine (assuming that the PRINT routine is located at $900), you would pick a spot in memory that is not being used and enter the following:

```
*1000:20 00 09 C1 C2 C3 00 60


*10000
ABC
*
```

If you've been learning your machine code all along (or if you're like me, you cheat and look up the opcodes on a 6502 reference card), you'll notice that the above sequence represents the code:

```
JSR $900
ASC "ABC"
HEX 00
RTS
```

By typing 1000G and executing this short routine, you can test the PRINT routine to see if it works properly. For additional information on modifying memory locations, consult Chapter 3 of the new Apple Reference Guide (the 'White' book).

MODIFYING INSTRUCTION CODE (PATCHING).

Before LISA came along, most assemblers were quite slow and required a considerable amount of setup in order to assemble

****

code. As a result, the user found it easier to replace instruction code by stuffing hexadecimal data into memory rather than reassembling the program with the proper modifications. This required considerable knowledge on the user's part (since he had to memorize a good number of the 6502 opcodes); and he had to exercise the utmost care to prevent his patches from destroying valid instruction code. LISA is so fast and easy to use, however, that extensive patching should never be attempted. It is almost always easier to reenter LISA, correct the problem, reassemble the program and try again.

In some instances a manual patch may still be faster. Examples include: replacing an implied addressing mode instruction with another implied addressing mode instruction (i.e., you meant DEX instead of DEY), replacing an n-byte instruction sequence with NOP's (deleting the existing instruction), and installing BRK's within your program. These last two examples are especially important and will be considered in more detail.

The NOP's main use in the 6502 instruction set lies in timing delays and its ability to replace existing instructions without altering any registers or memory locations. The opcode for the NOP instruction is $EA; memorize it! When debugging programs, you will often need to replace an instruction with one or more NOP's. The only easy way to enter a NOP into your instruction stream is

****

to use the monitor memory modify command (<addr>:<data>) to replace the instruction at the specified address. For example, suppose you have an extra PLA instruction at location $890 in your code and you wish to delete it. You could reenter LISA and delete this instruction, but that operation would take about one minute before you would be able to test the results of removing the PLA instruction. A better approach, which gives you the ability to immediately test the results of removing the PLA, is to replace it with a NOP instruction. This is accomplished by typing '809:EA' from the Apple monitor command level. After that, rerun your program and see if it works. If you replace a two- or three-byte instruction make sure that the entire instruction is replaced, not just the opcode. Also, don't forget to go back and modify the source of your program after you are through testing it.

As you may recall, the 6502 BRK instruction stops the program and prints out the contents of the 6502 registers. This feature will prove to be extremely useful for debugging programs. By replacing an instruction within your program with the BRK instruction, you can stop the program before (or after) some critical section of code and examine the registers or some specific memory locations. The opcode for the BRK instruction is easy to remember: it's zero.

Using the BRK instruction to stop program flow at some point is known as 'setting a breakpoint.' Breakpoints are useful when you need to test a section of code that is not a subroutine. A breakpoint lets you execute a program up to a certain point and then stop program execution. At this point the registers and any particular memory location may be inspected. The single drawback to the BRK instruction is that it is very difficult to resume program execution at the point the BRK instruction was encountered. The reasons for this are: (1) normally you have replaced an instruction with the BRK instruction, and (2) the stack is messed up when the monitor is entered. While a program could be written to allow breakpoint management, it would not work on all APPLE II computers (in particular it would not work on APPLE

II computers without the Autostart ROM), and such a program is beyond the scope of this book.

Lazer Systems (the folks who brought you LISA) have another program to assist you in debugging your 6502 programs. This program is called "TRACE/65." It is an interactive debugging

****

tool for the APPLE II computer similar to debuggers available for CP/M systems. This program allows the machine language programmer the ability to single step through a program displaying all the registers. TRACE/65 also lets the programmer set non-destructive breakpoints, and gives the user the ability to display the instructions with symbolic display of memory locations on the video screen as they are being executed. In all, TRACE/65 can help cut debugging time in half.

To use TRACE/65 you must assemble a program, using LISA, in the $800 to $47FF area of memory. These are the only locations allowed for storing the object code by the TRACE/65 program. Once the code is correctly positioned in memory, BRUN the TRACE/65 program. When you are greeted with a ")" prompt, type T followed by return. TRACE/65 will prompt you to enter a starting address. Enter the starting address of your program. The TRACE/65 program will immediately begin executing and displaying your program. You can stop the rapid display of the execution of your program by depressing the space bar. This will stop the execution of your program until you depress the space bar again. This feature gives you the ability to slowly watch the execution of your program a step at a time.

TRACE/65 incorporates two modes of operation: the execution mode and the 'parameter' mode. The execution mode is the mode whereby programs are executed and displayed on the screen. The parameter mode is the mode in which you set breakpoints, toggle the display mode, modify memory locations and registers, and exit the parameter mode.

To enter the parameter mode, type "P" from the command level or stop your program from running (by depressing the space bar) and type "P." At this point you will be greeted by a menu describing the possible options. If you press "A," you will toggle the display mode. The display mode controls the printing of the traceout. If the display mode is set, then all instructions executed will be printed on the Apple CRT screen. If the display mode is turned off, this listing will be supressed. The display off mode will typically execute a section of code 100 times faster than the display on mode. This allows you to quickly skip over sections of code that do not need debugging (such as loops to zero out memory) and then turn the listing back on when a section of questionable code is encountered. The breakpoints (described

****

next) are used to terminate a section of code that is being executed in the non-display mode.

The 'B' option in the parameter menu is breakpoint selection. A breakpoint is simply a command to the TRACE/65 program to halt execution whenever an instruction at a certain address is encountered. The breakpoint option in the parameter mode lets you set a breakpoint address. TRACE/65 provides you with up to four user-definable breakpoints. When you press 'B,' TRACE/65 will ask you which breakpoint you wish to set, and then it will prompt you to enter the breakpoint address.

The 'C' option lets you return to the executing program (or TRACE/65 command level) without executing any of the parameter mode commands. This is useful in the event 'P' is accidentally pressed.

The 'D' option is used to quit the trace mode. This option lets you terminate program interpretation after a desired section of code is checked out.

The '$' option lets you enter a monitor command during program execution. This could be used to change a memory location or 6502 register, or possibly to disassemble the next section of code that is to be executed before actually executing it. To change a memory location, simply type '$<loc>:<data>.' To change a 6502 register, you must modify one of the zero page locations $DA through $E9. The registers affected are:

```
PC  :$DA, $DB
ACC :$E5
XREG:$E6
YREG:$E7
PSW :$E8
SP  :$E9
```

Incidently, these are the only zero page locations in the range $D6-$FF you or your program should modify. Should any of these memory locations be modified, unpredictable things may happen.

PROGRAM DEBUGGING SESSION.

Consider the following program:

```
START:
        LDX #$0
LOOP    LDA MSG,X
        BEQ QUIT
        JSR $FDED
```


****

```
        DEX
        BNE LOOP
;
MSG     ASC "THIS IS A TEST"
        HEX 00
;
QUIT    BRK
        END
```

This program has one very obvious problem, DEX is used in place of INX. This program will probably print lots of garbage instead of the desired message. To trace this program using TRACE/65, you would BRUN TRACE from BASIC. Once you were into the TRACE/65 program, you would load the program in using the DOS command "BLOAD PROGRAM." Before executing the program, a breakpoint should be set. The breakpoint will be set for location $FDED. We set a breakpoint here because we already know that the Apple monitor COUT routine works and there's no sense in wasting time to debug it. A breakpoint is set by typing "P" (to get into the parameter mode) and then "B." Any of the four breakpoints may be used, let's select breakpoint #1. This is accomplished by typing "1." TRACE/65 will now ask for a breakpoint address. Enter FDED. Once FDED is entered, you will be returned to the TRACE/65 command level. Now program execution may begin.

To begin the trace mode, type "T." TRACE/65 will prompt you for a beginning address. Once this is entered, the trace mode begins. Since a breakpoint was set at location $FDED, the trace will quickly stop with the message 'BREAK POINT ENCOUNTERED AT LOCATION $nnnn. Notice right below the last instruction displayed, that the accumulator's contents are displayed. Currently the accumulator will contain $D4 (T), which is the first character in our string. Fine, things are working out okay so far. To continue execution (without executing the COUT routine), the parameter mode must be entered (by typing "P"). Now type $DA:58FF. This will point the program counter at a RTS instruction which will cause an immediate return to your program. Finally, type "C" to continue the execution of your program. Another batch of instructions will be executed, and once again the program will encounter a breakpoint. This time, however, there is probably garbage in the accumulator, and the X-index register will contain $FF. By tracing back a few instructions on the screen, you will

****

notice that the X-index register was decremented instead of incremented. Voila, the problem is solved.

While this is a very simple example, it demonstrates how breakpoints are used to skip over certain sections of code. Breakpoints can also be used to allow speedy execution (in the display off mode) until a questionable section of code is encountered, or to automatically stop program execution at any point.

Debugging code is learned mostly through experience. The more you do it, the better you get at it. As the saying goes, practice makes perfect.

****

APPENDIX A

APPLE II COMPUTER
TABLES, CHARTS,
AND GRAPHS

omitted

A-1 through A-34

****

****

N

O

P

R

S