PROGRAMMER'S GUIDE TO THE 1802

(WITH AN ASSEMBLER FOR YOUR MACHINE)

TOM SWAN

HAYDEN

Programmer's Guide to the 1802

(with an Assembler for Your Machine)

Programmer's Guide to the 1802

(with an Assembler for Your Machine)

TOM SWAN

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

To my Mother and Father

Library of Congress Cataloging in Publication Data

Swan, Tom.
Programmer's guide to the 1802.

Includes index.
1. Microprocessors—Programming. 2. Assembler language (Computer program language) I. Title.
QA76.6.S9139 001.64'2 80-29627
ISBN 0-8104-5183-2

1 2 3 4 5 6 7 8 9 PRINTING

81 82 83 84 85 86 87 88 89 YEAR

Preface

This book is different. These pages do not speak of BASIC or FORTRAN or COBOL or PASCAL. All worthy languages, but all written in the language this book teaches—assembly language. This book teaches assembly language for the 1802 microprocessor. And this book has something few assembly language primers have. An assembler.

From the binary number system to the fundamentals of machine language to the development of a working 1802 assembler, this book has something for all. Each of the 1802's instructions is explained in detail. The text is written in nontechnical language simple enough for beginners but with information that will be appreciated even by experts.

Whatever your personal interest, even if only to examine the power of the popular 1802 microprocessor, this book is for you.

To the prospective buyer: The assembler in Chapter 4 is the result of many efforts, some grander than others, to construct a tool that all 1802 users will be able to run on their computers.

Although the program does not support the use of labels, macros, or mathematical expressions, this assembler recognizes all 1802 mnemonics as recommended by the manufacturer. One bonus feature of the program is its ability to function as a disassembler as well as an assembler by using a different entry point. Also you may add mnemonics to the op code table or even change the manufacturer's standard names to your own designs if you wish.

You will need to supply input and output to the assembler depending on your computer's operating system. All coding is in standard ASCII form, however, and all parameters and entry/exit points are clearly marked and explained.

Best wishes and good luck in all of your efforts.

TOM SWAN

Contents

Programmer's Guide to the 1802

(with an Assembler for Your Machine)

1

□ □ □

A System of Numbers—A Number of Systems

1010 Little Indians

The decimal number system with its 10 symbols so familiar to us is far too complicated for most digital computers to understand. The *binary* number system, on the other hand, with its two ("bi") symbols one and zero, is perfectly suited to the internal mysteries of computers.

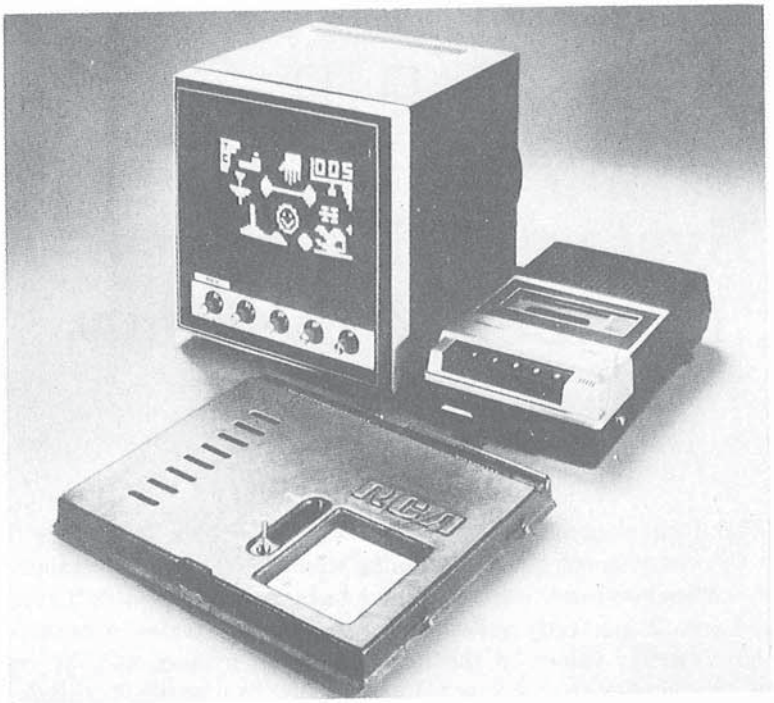
Electrically, values in the binary number system may be represented by switches that are on (1) and off (0). In a nutshell, this is how computers understand and manipulate numbers—by controlling and testing series of switches representing values in the binary number system.

A computer's power comes from its ability to control more of these switches than the number of light switches in all the homes of a good-sized town. Until a way was found to place all those switches in the small plastic chips called integrated circuits (ICs), some early computers used a room (a *big* room) full of relays, *each* relay representing either a 1 (on) or a 0 (off) in binary. Running a program in those days sounded somewhat like 10 minutes to deadline down at the newsroom of the *Daily Bugle*. And though memory was no doubt as reliable as an elephant's, it had the disadvantage of coming in a package of about the same size.

Here is the decimal number 1,232 expressed in binary:

$$1232_{10} = 0100\ 1101\ 0000_2$$

The small number *down* and to the right is there to indicate which number system was used to express the quantity. The 10 means that the decimal system was used, while the 2 tells us that the number is a binary number. This is called the *base* of the number and without it things could get very confusing. For example, what value does 10111010 represent? In decimal, this would be 10 million, one hundred eleven thousand and ten. In binary, the same number would represent the quantity 186_{10}, which is read "one hundred eighty-six, *base* 10." (Binary numbers will



RCA popular VIP computer is powered by the 1802 COSMAC Microprocessor. (*Courtesy* RCA Corporation)

usually be grouped into sections of four digits from now on to help distinguish the binary values from decimals. In obvious cases, the base will not be written.)

Learning the binary number system will be easier if we take a moment to review the decimal version of counting, which you memorized when you were a child. The word decimal (from the Latin *decima* or *decimalis*) refers to the number of symbols used to represent values. These symbols are, of course, the Arabic 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Combinations of these 10 symbols produce unique and recognizable numbers of any conceivable values.

Suppose, however, that 10 different symbols, also referred to as the *coefficients* of a number system, had been chosen instead of our familiar Arabics. For example, 0 * ( ) + % X / ; : or any others. Then we would have memorized these instead, and the telephone number 555-5657 would look like %%%-%X%/. Notice that the two numbers are the same—only the symbols used to represent the same phone number have been changed.

We may therefore conclude that any number may be represented in either of the number systems. The symbols themselves acquire what-

ever meaning we choose to give them. Given an infinite set of number systems, any number could be represented in an infinite variety of ways.

Then how does a number keep its individuality since, after all, the 10 symbols we have to use in the decimal system do not necessarily appear uniquely in that number? In fact, the same symbols may be used over and over, perhaps even duplicated many times. For example, 1,223 is a unique number, although the same symbol, the 2, appears twice. However, 1,232 is definitely not equal to 1,223, although it uses the exact same symbols. Obviously, the *position* of each symbol tells us something about the value of the number.

Each column of a number has an inherent value and each symbol appearing in a column simply tells us how many of those values are contained in the number. The *power* of a column increases in ascending order moving from right to left. Our example, 1,232, could be read as containing 1 1000; 2 100s; 3 10s; and 2 1s. For every number system we will consider—binary, hexadecimal, vigesimal, octal—all of these (including decimal) use the same idea of positioning different symbols in columns of fixed powers.

The above example in mathematical form is:

$$1{,}232 = (1 \times 1{,}000) + (2 \times 100) + (3 \times 10) + (2 \times 1)$$

Mathematicians are not fond of writing zeros apparently, so the above is usually further reduced as follows:

$$1{,}232 = (1 \times 10^3) + (2 \times 10^2) + (3 \times 10^1) + (2 \times 10^0)$$

Note that instead of writing the numbers 1,000, 100, etc., we have instead written 10^3, 10^2, etc. The little number above and to the right of each 10 is called the exponent of that number and indicates that the value we want is really that number of 10s multiplied together. In other words:

$$10^3 = 10 \times 10 \times 10 = 1{,}000$$
$$10^2 = 10 \times 10 = 100$$
$$2^3 = 2 \times 2 \times 2 = 8$$
$$2^2 = 2 \times 2 = 4$$

It is common to express the value 10^3 as "ten to the third *power*"; 10^2 as "ten to the second power"; etc. Carrying things further, we can express our example number like this:

$$\begin{aligned} 1{,}232 = 0 \times 10^4 = 0 \times 10{,}000 &= 00000 \\ 1 \times 10^3 = 1 \times 1{,}000 &= 1000 \\ 2 \times 10^2 = 2 \times 100 &= 200 \\ 3 \times 10^1 = 3 \times 10 &= 30 \\ 2 \times 10^0 = 2 \times 1 &= +\ 2 \\ &\quad 01232 \end{aligned}$$

The leading zero was included to show that, of course, any column with zero in it equals nothing. In fact, zero will mean absolutely nothing at all throughout the remainder of this book! (This is not as facetious a thing to say as you may think. The discovery that nothing, or zero, could be used to hold a position in a number signifying that the column *equals* zero was no small discovery. The revelation made modern math possible—and computers!—and is attributed to an unknown Hindu supposedly living before the ninth century according to some references, and as early as the first century according to others.)

As in decimal, binary symbols are written in columns of fixed powers. Since there are only 1s and 0s in binary, however, a column either has a value or it doesn't. To save space, let's work on a small binary number, figuring its value just as we did for the decimal number 1,232. Instead of using the powers of 10 to figure the decimal equivalent of a binary number, however, we will use the powers of 2, the base or radix of the binary number system.

$$
\begin{aligned}
\text{(Binary) } 1101_2 = 1 \times 2^3 &= 1 \times 2 \times 2 \times 2 = 8 \\
1 \times 2^2 &= \quad 1 \times 2 \times 2 = 4 \\
0 \times 2^1 &= \quad\quad 0 \times 2 = 0 \\
1 \times 2^0 &= \quad\quad 1 \times 1 = 1 \\
&\qquad\qquad\qquad\qquad 13_{10} \text{ (Decimal)}
\end{aligned}
$$

The binary number, 1101, has been converted into its decimal equivalent 13. To go the other way from decimal to binary is just as easy—simply divide by 2, writing down the remainders from right to left until you can go no further. In other words, reverse the above process. An example will help. Let's use 13_{10} again:

13/2 = 6 plus a remainder of 1
6/2 = 3 plus a remainder of 0
3/2 = 1 plus a remainder of 1
1/2 = 0 plus a remainder of 1

Reading the remainders from bottom to top, 1101 proves to be the binary equivalent of the decimal value 13. (If you write down the remainders from right to left as you divide, you'll avoid switching things around accidentally.)

To represent large quantities in binary, a *lot* of 1s and 0s are needed. Computers don't mind, but the binary number 1110 1011 1100 1101, which equals $60{,}365_{10}$, is not clearly recognizable by humans. A way around this problem will be covered later. More important is for you to understand that any value may be expressed in the decimal, binary, or any other number system!

Backing up just a little, notice that the conversion from binary to decimal used the base of the number raised to a particular power to figure the value of a digit in any column. In decimal we used the powers

of 10. In binary, the powers of 2 were used. In the hexadecimal number system, we would use the powers of 16. In the rather specialized binary system of a certain western tribe of the Torres Straits, the powers of "okosa" might be used. Whatever number system you are using, the value of any digit is equal (in decimal) to that digit times the base number raised to the appropriate power. The powers are always 0, 1, 2, 3, . . . *N* starting from the right and working to the left. An important rule to memorize is that the units column of *any* number system always has a power of 1. That is: $10^0 = 1$ and $2^0 = 1$ as well as $16^0 = 1$ and $\text{okosa}^0 = 1$.

Table 1-1 shows the binary equivalents for the decimal numbers 0 to 9.

Table 1-1.
Binary Counting

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

Another system similar to binary was used in some early relay type computers, which I've heard were known as "clack-u-lators." While Table 1-1 defines the binary number system as employed in most modern computers, Table 1-2 presents the biquinary system as it was used on those antique predecessors.

Table 1-2.
Biquinary Counting

Decimal		Biquinary	Decimal		Biquinary
0	=	01 00001	5	=	10 00001
1	=	01 00010	6	=	10 00010
2	=	01 00100	7	=	10 00100
3	=	01 01000	8	=	10 01000
4	=	01 10000	9	=	10 10000

Notice the similarity of the biquinary system to the abacus, a mechanical device reputed to be as fast in the hands of an expert as a modern calculator. Biquinary does not strictly follow the rules for positional number systems, so we will not see it again in this book.

Numbers less than 0 or fractions may also be expressed in terms of their powers. For example, the number 123.45_{10} may be written using negative exponents for the fractional part.

$$
\begin{aligned}
123.45_{10} = 1 \times 10^{2} &= 100 \\
2 \times 10^{1} &= 20 \\
3 \times 10^{0} &= 3 \\
4 \times 10^{-1} &= 0.40 \\
5 \times 10^{-2} &= +\ 0.05 \\
&\overline{\qquad\qquad} \\
&123.45_{10}
\end{aligned}
$$

The same holds true in other number systems.

$$
\begin{aligned}
110101_{2} = 1 \times 2^{2} &= 1 \times 4 &&= 4 \\
1 \times 2^{1} &= 1 \times 2 &&= 2 \\
0 \times 2^{0} &= 0 \times 1 &&= 0 \\
1 \times 2^{-1} &= 1 \times .5 &&= 0.5 \\
0 \times 2^{-2} &= 0 \times .25 &&= 0.0 \\
1 \times 2^{-3} &= 1 \times .125 &&= +0.125 \\
&&& \overline{\qquad\qquad} \\
&&& 6.875_{10}
\end{aligned}
$$

To find the value of a number raised to a negative power, divide the base you want by the multiple of the base you are trying to convert. In other words, to get to decimal from base 2 or base 6:

$$
\begin{aligned}
1 \times 2^{-2} &= 10/4 \qquad\qquad & 2 \times 6^{-1} &= 10/(6 \times 2) \\
1 \times 2^{-3} &= 10/8 \quad \text{and} \quad & 4 \times 6^{-2} &= 10/[4 \times (6 \times 6)] \\
1 \times 2^{-4} &= 10/16 \qquad\qquad & 1 \times 6^{-3} &= 10/(6 \times 6 \times 6)
\end{aligned}
$$

Although floating point representations are beyond the scope of this book, you should be aware that representing fractional numbers is no more difficult than working purely with whole integers.

Exercises

1. Why is the binary number system used to represent values in a computer?
2. Convert the following binary values to decimal:
   a) 1111 b) 1011 c) 1101 1011 d) 1110 0110
3. Convert the following decimal values to binary:
   a) 100 b) 64 c) 249 d) 87

Binary Arithmetic or "1 + 1 = 10?"

If you had written the above on a math exam, you would probably have drawn the attention of the professor and possibly raised a few eyebrows. If you had been adding *binary* numbers, however, the above answer would have been correct! In order to understand how to add and subtract binary numbers, we must again turn to the familiar decimal system for a little guidance.

Adding, with some variations, can be thought of as two processes: counting and carrying. Numbers to be added are usually placed on top of each other, and the values of each column are obtained by counting them up. Using your fingers to add is a considered embarrassment, but the 10 fingers of two hands, actually the source of our adapting the decimal (base 10) number system, show very well the second part of the adding process: carrying.

Just for illustration, add 8 + 6 on your fingers. Before you reach the answer, you run out of fingers. (No fair using your toes—at least not until we get to the vigesimal base 20 number system!) When adding, everytime you run out of digits—appendages or the numerical types—you would make a mental note or a mark on a piece of paper to signify a carry. Of course, you already know this, but understanding exactly what happens in the process of adding will help you to apply the same process to other number systems. Here is a simple example of a decimal addition and the equivalent values in binary.

Carries:			1
	1	=	1
	+ 1	=	+1
	2		10
	Decimal		Binary

EXAMPLE 1-1

I'm going to repeat the adding rule as it is crucial to your understanding of binary addition.

1. Count up the values in each column working from right to left.
2. Whenever you run out of symbols available in the number system, generate a carry of 1 to the column immediately to the left.

Since there are only two symbols in the binary number system, it does not take long to run out of symbols. In Example 1-1, two *decimal* 1s added together produce the number 2. No problem here; there are plenty more symbols available. But adding two *binary* 1s cannot produce a 2, since that symbol is not available in the binary system, which uses only 1s and 0s. Therefore, following the adding rule, write down a 0 in that column, generate a carry of 1 to the left and continue to add. Since the next two columns are blank (I could have written the binary numbers 01 + 01 instead of 1 + 1), adding the carry of 1 to the two 0s produces a 1. The answer is 10_2, which is probably best pronounced as "binary one, zero" *not* as "ten" to avoid confusion.

The following addition produces two carries out of the first column to the column on the left. The two carries added together produce another carry to the third column:

```
Carries:        1
Carries:       11
              ----
               01₂
               01₂
               01₂
               01₂
             + 01₂
              ----
             101₂ = 5₁₀
```

EXAMPLE 1-2

To prove that the answer is indeed equivalent to decimal five, multiply by the powers of 2 as previously explained.

$$\begin{aligned} 101_2 = 1 \times 2^2 = 1 \times 2 \times 2 &= 4 \\ 0 \times 2^1 = \quad 0 \times 2 &= 0 \\ 1 \times 2^0 = \quad 1 \times 1 &= 1 \\ & \overline{\quad 5_{10}} \end{aligned}$$

Now for a few more complex examples. Work through each column, counting and carrying in binary. Whenever 1 is added to 1, produce a carry to the left and continue. (I have included the leading zeros to make it easier for you.)

```
Carries:        1111          111
                              11111
              -------       -------
               01011₂        001111₂
               00110₂        001111₂
             + 01001₂      + 001111₂
              -------       -------
               11010₂        101101₂
```

Subtraction is equally simple, although later on we will examine a way to subtract by adding, since that is how a computer usually accomplishes a binary subtraction. You should understand the process of subtracting two binary numbers, however.

Instead of a carry, whenever subtracting a 1 from a 0, a borrow is needed from a column on the left. (Assume that the subtrahend on the bottom is less than or equal to the minuend. We won't worry about binary *negative* numbers just yet.)

Minuend	1	10	11	101
− Subtrahend	− 1	− 1	− 01	− 010
Answer	0	1	10	011

(All values in binary base 2 notation)

Again, in a coming chapter, I'll introduce the easy subtract method, but you should work through the examples below to be familiar with the process of binary subtraction. As your computer skills progress, you will

call on your binary math abilities more and more. I sometimes even convert values into binary from a more unfamiliar number system in order to add them together and then convert the answers back again! Even in the most complex computer programs, I find myself adding binary numbers together. Your understanding of these simple processes will reflect on your progress in programming on a machine language level. After all, if you want to communicate with your computer, you both have to speak the same language!

Exercises

1. Add the following binary values. (The base 2 indication has been left out here. Assume that the following numbers are expressed in the binary number system.)

a) 1011 1100 + 0110 0110

b) 110 0000 + 010 1111

c) 0111 1111 + 0000 0001

d) 1101 0101 0110 + 1001

e) 1111 0010 0101 + 1000

f) 1011 1101 1110 + 0111

2. Convert each of the above binary values to their decimal equivalents to prove that your answers are correct.
3. What would the decimal value 7 look like in the *unary* number system? Prove your answer.
4. Subtract the following binary values, and prove your results as in Exercise 2. (Again, assume all the values here to be base 2 expressions.)

a) 1111 1111 − 0000 0001

b) 1010 1010 − 0101 0101

c) 1101 1011 − 1011 1101

The Hexadecimal Number System

Binary computers are not the only beasts to employ number systems other than decimal. In the last section we looked at several systems, the binary system used extensively in computer programming, for example, while glancing at a few others just for fun such as the vigesimal base 20 system.

Even though you will probably never use a number system such as the vigesimal extensively, it's instructive to realize that unlimited sys-

tems for counting are possible, and that some very strange ones are in use in the world even today. Remember, no matter how the quantity 255 is represented, in any number system it is *still equal to 255 units* of whatever is being counted! Different number systems are only vehicles for expressing the same things. If you understand this, you will be well on your way to understanding the workings of binary computers. (It's the only way to get them to understand us!)

The base 20 vigesimal number system, by the way, was widely used by primitive people who counted on both their fingers and toes. The Mayas of Central America and the Aztecs of Mexico used this system. The Aztec day, for example, was divided into 20 parts, and 8,000 warriors (20^3 or $20 \times 20 \times 20 = 8{,}000$) made up a division of the Aztec army, evidence that this famous and tragic civilization preferred to count with more digits than we do.

There are modern leftovers from the vigesimal number system—you have probably encountered and even used them. Abraham Lincoln said, "Four*score* and seven years ago . . ." in his famous 1863 Gettysburg Address when "eighty-seven" ($4 \times 20 + 7$) would have done just as well mathematically if not poetically. The "score" is, of course, equal to 20 and is an English word of probable Middle English (scor) or Old Norse (skor) descent. The French also used the base 20 system in various ways, dividing corps of police sergeants into groups of 220 men for one example.

Here are the powers of the vigesimal number system believed to have been used by the ancient Mayas.[1]

Power	*Maya*	*Decimal Equivalent*
1	hun	1
20	kal	20
20^2	bak	400
20^3	pic	8000
20^4	caleb	160,000
20^5	kinchel	3,200,000
20^6	alce	64,000,000

Another number system in wide use today is the *hexadecimal* system, which will prove its value to you if you plan to do much programming on binary computers. Hexadecimals are especially useful on processors such as the 1802 that operate on binary numbers composed of multiples of eight binary digits or bits.

Why *hexa*decimal? Literally from the Greek "hex" meaning "six," hexadecimal may be translated "six and ten," although the word itself seems to be a combination born of two cultural sources—Latin and Greek. Unlike binary, which uses only two symbols, 0 and 1, hexadecimal

[1] From *Number; the Language of Science* by Tobias Dantzig, New York, Doubleday & Co., 1954.

numbers are composed of the normal 10 Arabics *plus* 6 more. You may think, "But now I've got 16 instead of 10 symbols to keep track of!" In part you are correct. Hexadecimal numbers are not the easiest things to work with.

Some important advantages outweigh this apparent complexity of the hexadecimal number system, however. The single most important reason for using hexadecimals is the ease of converting them to and from their binary equivalents. For this reason, hexadecimals serve programmers quite well. Numbers in binary form are easily recognized by computers. But to a human, long strings of 0s and 1s may appear confusing and meaningless.

Table 1-3 lists the decimal values 0 to 15 in three other number systems. Note in particular that the binary values range from 0000 to 1111. In other words, every possible value that can be expressed in binary using a space of four digits is included in the table. There are exactly 16 different binary values including 0000. Look carefully at the hexadecimal column. Six letters have been added to the 10 symbols of the decimal number system for a total of 16 symbols. By using the values in Table 1-3, any binary value may be converted into its equivalent hexadecimal.

Here are some examples of binary values and their hexadecimal equivalents:

Binary	1011	1111	0000	0100	Binary	1000	1110
Hex	B	F	0	4	Hex	8	E
Binary	0001	1001	1010	0000	Binary	1111	1111
Hex	1	9	A	0		F	F

To convert any length binary number into hexadecimal, first separate the binary number into groups of four digits, from right to left, then convert each group using Table 1-3. The reason for doing this is arbitrary. Computers only understand binary numbers. Representing binaries in hexadecimal is strictly a convenience to the programmer. Even though a computer may accept hexadecimal input, that input is actually stored in binary form.

Converting hexadecimal to decimal may be done in the same manner of converting from the binary (or other base) system. Base 16 raised to successive powers is multiplied by each digit to obtain the corresponding decimal value. For example:

$$
\begin{aligned}
C2B3_h = C \times 16^3 &= 12 \times 16^3 = 49{,}152 \\
2 \times 16^2 &= 2 \times 16^2 = \quad 512 \\
B \times 16^1 &= 11 \times 16 = \quad 176 \\
3 \times 16^0 &= 3 \times 1 = + \quad 3 \\
& \qquad\qquad\quad \overline{49{,}843_{10}}
\end{aligned}
$$

Table 1-3.
Counting in Four Number Systems

Decimal (Base 10)	Binary (Base 2)	Hexadecimal (Base 16)	Octal (Base 8)
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Table 1-4
Hexadecimal/Decimal Conversion Table

Column 3 ($H^* \times 16^3$) HEX/DEC	Column 2 ($H \times 16^2$) HEX/DEC	Column 1 ($H \times 16^1$) HEX/DEC	Column 0 ($H \times 16^0$) HEX/DEC
0 = 0	0 = 0	0 = 0	0 = 0
1 = 4,096	1 = 256	1 = 16	1 = 1
2 = 8,192	2 = 512	2 = 32	2 = 2
3 = 12,288	3 = 768	3 = 48	3 = 3
4 = 16,384	4 = 1,024	4 = 64	4 = 4
5 = 20,480	5 = 1,280	5 = 80	5 = 5
6 = 24,576	6 = 1,536	6 = 96	6 = 6
7 = 28,672	7 = 1,792	7 = 112	7 = 7
8 = 32,768	8 = 2,048	8 = 128	8 = 8
9 = 36,864	9 = 2,304	9 = 144	9 = 9
A = 40,960	A = 2,560	A = 160	A = 10
B = 45,056	B = 2,816	B = 176	B = 11
C = 49,152	C = 3,072	C = 192	C = 12
D = 53,248	D = 3,328	D = 208	D = 13
E = 57,344	E = 3,584	E = 224	E = 14
F = 61,440	F = 3,840	F = 240	F = 15

*H, hex digit.

If you are going to use hexadecimals extensively, you will probably prefer using a powers of 16 table such as the one presented in Table 1-4. This table gives each hexadecimal digit multiplied by 16 raised to the powers of 0, 1, 2, and 3. A similar table is provided with 1802-based computers such as the RCA Cosmac VIP. To convert from hexadecimal to decimal, add up each decimal value appearing to the right of the hex digit. To go from decimal to hexadecimal, start with the whole decimal value and subtract the largest possible decimal values in the table, writing down their hexadecimal equivalents as you go.

To convert hex to decimal:

$$6B4D_h = 24{,}576 + 2{,}816 + 64 + 13 = 27{,}469_{10}$$

To convert decimal to hexadecimal:

$$56{,}457 = (56{,}456 - 53{,}248) = (3{,}209 - 3{,}072) = (137 - 128) = (9 - 9) = 0$$
$$\underset{D}{} \qquad\qquad \underset{C}{} \qquad\qquad \underset{8}{} \qquad\qquad \underset{9}{}$$
$$= DC89_h$$

Out of a greater love for the ancient Greeks than the Romans, I will say "hex" from now on when I mean hexadecimal. Most computer literature seems to agree with my cultural bias, and "hex number" has become a common term among programmers.

Hex numbers are not new to programming, in fact the earliest hex system used a different set of letters instead of the accepted A, B, C, D, E, and F on top of the 10 Arabics. In the early 1960s, the lower case letters t, e, d, h, f, and i rounded out the set. Other literature refers to upper case U, V, W, X, Y, and Z for the additions. The lower-case hex set was thought to be easily remembered for the following reason:

t = *t*en
e = *e*leven
d = *d*ozen
h = t*h*irteen
f = *f*ourteen
i = f*i*fteen

Lucky for us this didn't catch on! Imagine working with the value h4tf, where t is less than f and f is greater than h even though their alphabetic order suggests the opposite. It is far more reasonable that C is greater than A and B is less than F.

One serious shortcoming of hex numbers is that people usually do not print as well as typewriters and printing presses. The typewritten D is certainly not a 0 and the B is decidedly not an 8. The author has learned from nerve-racking first-hand experience to slow down when hand-assembling a listing into hex values. I sometimes feel that U, V, W, X, Y, and Z would not have been so bad after all because these

symbols do not resemble any of the numerical digits. But you know what they say: "When in Athens, do as the Athenians do!"

Many programmers use another number system, the octal or base 8 system, although its popularity in home computing seems to have decreased severely. It probably has the advantage of unambiguity over the hex digits, although it does not translate into eight-bit binary bytes quite as nicely. The octal system uses the symbols 0, 1, 2, 3, 4, 5, 6, and 7 before repeating to 10. To convert from binary to octal, a binary number is split into groups of three digits, and charts similar to those given here for hex may be used to find the equivalent octal and decimal values.

Exercises

1. Convert the following hexadecimal numbers to their binary equivalents:
   a) DEAF b) CAFE c) FACE d) BEAD
2. Convert the following binary values to their hexadecimal equivalents:
   a) 1100 0110 b) 1111 0101 c) 1010 1100 1110 0110
   d) 1000 1001 1101 0010
3. Convert the following decimal values to hexadecimal:
   a) 32,694 b) 60,050 c) 11,112 d) 40,692
4. Convert the following hex numbers to decimal:
   a) 01F2 b) 29FF c) 846C d) F1F0
5. Perform the following math exercises using whatever means you prefer to arrive at the answers. Express your answers in decimal, hex, and binary.
   a) $2{,}192_{10} + 1011_2 + 0BB4_h =$
   b) $1110\ 0110_2 + 641C_h + 89CC_h =$
   c) $CBCB_h + 82_{10} + 1011\ 1110_2 + B9_h =$
   d) $B19C_h \times 6A_h =$

2

□ □ □

Fundamentals of Assembly Language

After presenting the rudiments of the binary number system, many computer manuals proceed directly to an explanation of the CPU (central processing unit) instruction set without giving the reader a chance to take a breath. If this is your first attempt to learn machine language programming, this may be too great a step for you to take.

Most computers contain the same basic operations in their instruction sets, however, and a general course on assembly language will give you a solid base on which to build programming experience. Although there are differences from processor to processor, and some are said to be more powerful than others, you will find that they all may be programmed to do the same things in similar ways.

I sincerely hope that after learning 1802 machine code you will go on and learn half a dozen others. This chapter is designed as a springboard toward that goal, although we will naturally give most attention to the 1802's capabilities.

Have you ever examined a listing in machine or *assembly* language only to wonder if such a confusion could ever be understood by anyone not of Einsteinian caliber? For example:

Line#	*Address*	*Machine Language*	*Labels*	*Assembly Language*	*Comments*
01	0300	E2	ADD :	SEX 2	; X=2
02	0301	46		LDA R6	; D ← operand #1
03	0302	52		STR R2	; Push
04	0303	46		LDA R6	; D ← operand #2
05	0304	F4		ADD	; D ← op#1 + op#2
06	0305	AE		PLO RE	; Answer
07	0306	D5		SEP R5	; Return

To a beginner, this may appear to be as unreadable as Old English is to a college freshman.

Admittedly, a higher-level language such as BASIC or PASCAL produces a program that is more readable than assembly language. When trying to "take apart" a mysterious routine, this becomes especially true. But when you are familiar with assembly mnemonics (pronounced "nee-mon-icks"), a defined subroutine can be as easy to understand in assembly as in any other language.

It is therefore most important to record accurately the function of a machine language routine. Note that the assembler in Chapter 4 contains comments after just about every instruction. In addition, each subroutine has a companion description that details input, output, calls to other subs, what subs call this one, and the registers that are changed.

Reading a well-documented assembly language program may take some study. Reading an undocumented program—even if you are familiar with the machine code—may take forever. For your own sake, be liberal with comments. Even though a particular sequence may seem of obvious purpose today, that obvious meaning may become oblivious even to the programmer tomorrow.

All machine languages may be broken into five general categories. Some specialized computers may require additional categories, but all of them contain instructions that fall into these five groups: (1) logic and arithmetic operations, (2) program flow operations, (3) operations on memory, (4) operations on internal registers and miscellaneous, and (5) input/output.

The word register refers to a place inside the microprocessor. Registers may be thought of as individual compartments, each of a specified length, and each capable of being manipulated or used in certain defined ways. An accumulator or *D* register in the 1802 is a special register affected by a large number of instructions. Its job is to serve as a collecting point for all sorts of operations. In the 1802 microprocessor, the eight-bit D register is the center of focus for nearly all operations.

The 1802 contains 16 other general-purpose registers, more than most microprocessors. These registers are each capable of holding binary values 16 bits long and are sometimes referred to as "scratch pad registers," since they may serve as temporary holders for intermediate values. The 1802 is still an eight-bit machine, however, since the D register, its accumulator, may contain only a binary number that is no more than eight bits long.

Other microprocessors contain a variety of register formats and constructions. There are index registers (which the 1802 does not have),

accumulator extensions, one-bit registers (the 1802 has some of these plus some four-bit registers as well) for special functions, dedicated stack and program counters, and more. But all registers are alike in one sense—they are all capable of holding binary numbers of some length, and those binary numbers may be affected in defined ways by machine language instructions.

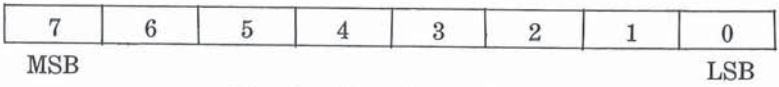


Fig. 2-1. One eight-bit byte.

Figure 2-1 shows a block sectioned into eight parts, each part capable of representing a 1 or 0 to an eight-bit computer. The word "bit" means "binary digit." A single "bit," therefore, refers to a single part of the block in Fig. 2-1.

Note that the bits are numbered backward from right to left starting with "0" not with "1." Although there are eight bits, the one furthest to the left is bit 7, and to mess things up further, bit 7 may also be called "the eighth bit." Yes, this is confusing, and for that reason this book will use "bit 7" or "bit 3" according to Fig. 2-1 rather than referring to the ambiguous "fourth bit."

There is a good reason for thinking of the first item of a set as 0 rather than 1, however. The convention goes along with the way computers tend to view the positions of lists of numbers. It's a convention that needs to be memorized and consciously applied.

Also note that under the block in Fig. 2-1 are the abbreviations "MSB" and "LSB." These mean "most significant bit" and "least significant bit" and are nicknames used extensively in computer literature. The LSB is always to the extreme right and the MSB to the extreme left. You will also see "MSD" and "LSD" occasionally, which are abbreviations for the most and least significant "digits."

Because of the way registers are constructed in the 1802 microprocessor, it is important to understand the least significant parts to be on the right, the most significant parts on the left. This goes hand in hand with the positional number systems we have discussed: the most significant parts of numbers having the highest values are to the left, and the least significant having the lowest values are to the right.

The eight binary digits or bits represented in Fig. 2-1 are commonly given the name *byte*, and in this book, one byte will always equal eight bits. (This is a rather common convention in microcomputing, but a byte does not always equal eight bits in much of the published literature.) Some books also refer to computer "words," but, since a "word" is even more loosely defined than a "byte," we will refrain from using it as a label for binary numbers. In addition, to make things come out right for eight-bit computers, leading zeros are usually written in front of

binary numbers so that all numbers come out to even multiples of eight-bit bytes.

Chapter 3 contains a detailed description of each of the 1802's instructions. If you thumb through that section, you will see each instruction listed in various forms. The blocked-in portions contain two of the forms we will explain here: the "op code" and the "mnemonic."

Op codes—"operation codes"—are the actual binary values that the computer understands as instructions. Mnemonics are descriptions of that instruction's action. Because mnemonics may be pronounced as words (e.g., LDN = "Load via N"), they are easier for a programmer to work with than the absolute (meaning "pure" or "actual") hex codes. Strictly speaking, the op codes form the "machine language," while the mnemonics form the "assembly language." The distinction is minor, and the terms are interchanged frequently, however. Some literature views assembly language as one of the higher levels above machine code, even though assembly is just another representation of the *same* programming level—machine language.

Only the op codes are stored in a computer's memory. The instructions are arranged by the programmer to perform whatever function is desired. Normal program flow will automatically cause one instruction after another to be executed in a direct line unless that flow is altered by an instruction. All programming languages assume a normal automatic program flow in a forward direction, which may be purposely altered, but when altered will again continue in the forward direction automatically.

The computer needs to know which instruction to execute next. A register is usually designated as the "program counter" (PC), which tells the computer where to obtain the next op code for execution. Many computers have a dedicated program counter—a register that may not be used for any other purpose except to control program flow. The 1802 allows any of its 16 general-purpose registers to be used as the program counter, and one of these must in fact be designated the PC at all times. Normally, R3 is used, but you are not bound to any restrictions—any 1802 register may become the PC by executing a single instruction designed for this purpose (the SEP RN instruction).

The program counter contains a 16-bit address that numerically identifies a location in memory. These addresses are simply binary numbers corresponding to physical locations, like individual post office boxes, in memory. Each location may contain an eight-bit binary value and that value may be a piece of data or it may be a computer instruction. Both data and instruction codes may occupy the same memory locations. Most eight-bit microprocessors including the 1802 may address (meaning "to point" or "to aim" toward) up to 65,536 different eight-bit memory locations corresponding to each of the hex values from 0000 to FFFF.

Usually this maximum amount of memory is referred to as "64K," although for what purpose remains a mystery (probably its divisibility by 8). When you have that much memory, a thousand or two bytes one way or another won't make much difference. But, knock on silicon, don't ever say such a thing aloud while programming. Algorithms can develop insatiable appetites for memory space given half a chance.

A typical tendency for beginners is to confuse the similarity of memory addresses and data. The fact that data may be an address or an address may be manipulated as data might appear particularly unlikely. But it's not. Addresses and data are both binary numbers capable of being manipulated by computer programs at the will (and sometimes against the will) of the programmer. When a register contains the address of some memory location, it holds a 16-bit value corresponding to a physical location in memory where some eight-bit value is stored. However, the address is only a means of finding that storage location—the actual address does not exist as a number in memory that can be obtained.

Memory circuits are usually wired to accept this 16-bit number, thereby opening channels directly to the desired eight-bit byte stored at the designated location. The action is the same for all memory locations allowing direct access to any position in memory without having to count or skip over any others. For this reason, memory is said to be "random access memory" (RAM), although the term has come to mean memory that retains values only so long as power is applied (i.e., it is "volatile").

Other forms of memory, read-only memory (ROM), programmable read-only memory (PROM), etc., also allow random access. In fact all types of computer memory may be addressed in the same way by specifying some 16-bit value intended as a pointer to a unique memory location. And any or all locations may be designated by the programmer to contain whatever is needed—data or instructions.

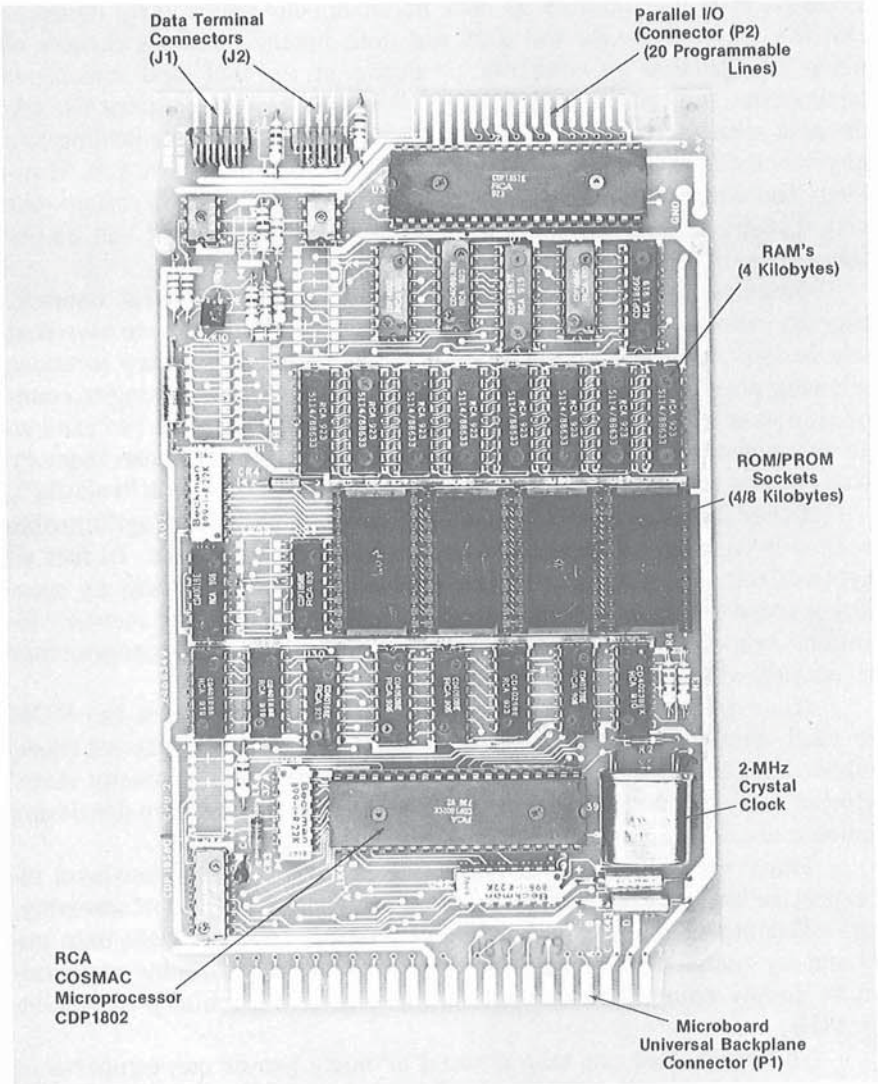
(One type of memory not usually seen in the market is the RUM or read usually memory. The author has a few of these unusual chips, which take their name as a function of their normally unsteady state. However, since this is a software manual, we will refrain from discussing much more of the hard stuff in the coming pages.)

Many programmers will choose assembly over a higher-level interpretive language for various reasons. They like the speed of assembly, the efficiency of memory use, and the freedom to manipulate data unbound by restrictions usually found with interpreters. Some programmers simply enjoy the challenge of working in a machine's native linguistics.

This book does not take a stand or make pro or con comparisons between assembly language programming and programming in the higher levels such as BASIC. It has been the author's experience that

the language should fit the application, and you the programmer have to decide which to use. I hope this chapter will help you digest the apparent complexity of assembly language, which does not have to be as difficult to use as it may seem. Rather than urge you to join the "which is best" controversy, this chapter will attempt only to provide a means for the programmer to make an intelligent choice of programming languages.

Here then are the fundamentals of machine language as they pertain to the 1802 microprocessor.



(*Courtesy* RCA Corporation)

Arithmetic and Logic Operations

All computers are capable of performing arithmetic in binary. Usually microprocessors such as the 1802 have only limited arithmetic instructions—they cannot directly multiply or divide—but these limitations should not be viewed as detriments. Even if all a computer can do is add, and they all can do that, even the most complex mathematics may be programmed.

Arithmetic operations take place in what is usually called the ALU or arithmetic logic unit, actually another register inside the microprocessor. The ALU in the 1802 processor is eight bits long, and, while all arithmetic is performed there, it is not available for direct use by the programmer.

Following an add or a subtract instruction—the only math we can perform directly with the 1802—the answer will always appear in the D register. Two binary bytes may be added or subtracted. One of these bytes is first placed in the D register and another value is then added or subtracted. The 1802 has the advantage of being able to subtract the contents of the D register *from* another value or to subtract that value *from* the D register. Most microprocessors can only subtract in one direction—that is, a value is usually subtracted from the one in the accumulator.

Just how and where that second operand is specified and how the first gets into the D register are subjects for later. For now, you only need to understand that in order to add or subtract two values: (1) the D register (accumulator) is loaded with one of the operands; (2) the instruction to add or subtract is performed; and (3) the answer appears in the D register (destroying, by the way, the original operand that was there!).

At this point, the program would presumably do something with the answer in the D register. It may decide to put it somewhere for storing until that value is needed. It may save it in part of a scratch pad register. Or it may use the answer as an intermediate value and add still other values to it. But at each step along the way, the computer is capable only of directly adding or subtracting two eight-bit bytes with the result that the operation appears in the D register.

Addition

One problem may easily occur when you are adding. For example, using the techniques presented in Chapter 1, add the following two binary values:

	Binary	*Hex equivalent*	
	1111 1111	FF	operand 1
	+ 0000 0001	+ 01	operand 2
overflow-	1/0000 0000	100_h	D register

EXAMPLE 2-1

Adding 1 to the largest possible eight-bit value (255 decimal) will result in a binary number that is nine digits long, too large to fit in the D register of our computer. A carry of 1 is generated out of the eight-bit capacity of the accumulator, which, if it were nine bits long, would contain the binary number 1 0000 0000.

Before we see how the computer deals with this situation, try adding the following two values:

	Binary	*Hex equivalent*	
	1111 1111	FF	operand 1
	+ 1111 1111	+ FF	operand 2
overflow-	1/1111 1110	$1FE_h$	D register

EXAMPLE 2-2

These examples show a very important fact about adding two binary values together. Example 2-1 shows the addition of one to the largest eight-bit binary value, and Example 2-2 shows the addition of the two largest possible eight-bit values. In each case, there was an overflow of one and only one bit out of the eight-bit capacity of the accumulator. We may conclude that if an overflow results from the addition of two binary values, that overflow will never be greater than one bit.

All computers have a way of dealing with this possible overflow bit. In the 1802 microprocessor, a one-bit register called the DF, or *D*ata *F*lag, register will contain the result of the overflow following an addition. If there was an overflow, DF will be set equal to 1. If there was no overflow, DF will be set equal to 0. To summarize, following addition (1) DF = 1 means overflow occurred and (2) DF = 0 means no overflow occurred.

Other microprocessors may call the DF register by other names. *You* may call it by several choice titles if it does not perform in the way you think it should, although the language referred to here is of a higher level than that to be presented in the following pages.

Some processors will refer to the DF as the carry flag or carry bit or just as the overflow bit or toggle. No matter, its function is identical, and regardless of its name the result is the same in all processors.

Subtraction

Subtraction is another matter. Earlier, we promised an easy way to subtract two binary values by simply adding them together. A simple decimal example will help explain the method:

$$(22 - 10) = 12 \equiv [22 + (-10)] = 12$$

(Read ≡ as "is equivalent to.") The example shows only that adding a negative value is the same as subtracting its absolute (or positive) equivalent value. The same is true in binary. When a number is to be subtracted, the computer needs only to add its negative equivalent.

An easy way to arrive at a binary negative equivalent is to take its "2s complement." Without even knowing what this is or why it works, you may easily construct the 2s complement of any value by following these simple directions.

1. Complement every bit. That is, if a bit is a one make it a zero and if the original is a zero, make it a one.
2. Add one to the result, ignoring overflow if it occurs.

To subtract any binary value B from any binary value A, simply add the 2s complement of B to A. After this addition, the overflow bit (DF in the 1802) indicates if a *borrow* was needed. If DF is equal to 0, then B was greater than A (before taking the 2s complement). If DF is equal to 1, then everything is OK; B was less than or equal to A and no borrow was needed. If DF is equal to 0 and B was greater than A, then the answer is in 2s complement (negative) form. In that case, taking the 2s complement of the answer will produce a positive value of the right absolute quantity.

Now that you know what a 2s complement is, we will proceed to describe how and why it works. If any of these processes seems strange, take a moment to work out a few examples on paper until you understand what is happening. You should not expect to grasp everything at a single reading, so don't be discouraged if the going is slow. No one has learned machine language programming or the ins and outs of binary overnight.

To further understand what happens with 2s complement subtraction, consider the following subtraction in binary. *Pretend* that there is a 1 bit to the left of the minuend:

Minuend	(1) 0000 0000	operand 1
Subtrahend	− 0000 0001	operand 2
	(0) 1111 1111	D register

We seem to have come full circle here, ending with the *largest* possible eight-bit value (255 decimal) in the D register. But if we *choose* to let that value equal −1, then we may choose to view the answer as correct. Just how we can represent negative numbers consistently in binary without such contradictions will be covered a little later. By using your basic binary math skills developed in Chapter 1, you should be able to verify that the result above is correct provided you make the assumption that a 1 bit appears to the left of the minuend. If we consider the answer to be nine bits long, then adding it back to the subtrahend should produce the minuend and this proves to be so. Try this on paper if you're not sure.

How can this subtraction be accomplished by adding? Easy. What value added to 0000 0000 will produce the same result as above?

(1) 0000 0000	operand 1
+ 1111 1111	operand 2
(0) 1111 1111	D register

Notice that the imaginary ninth bit to the left of the answer is the same in both examples. In fact, each answer is identical in all respects! If we let the DF register represent the imaginary bits to the left of operand 1 and the answer, DF's value conforms to the rules given for addition. Since there was no overflow following the add, DF equals 0, even though it was set to equal 1 before the addition was performed. In fact, its previous value has absolutely no bearing on the result. Even if DF equaled 0 before adding, the answer will be the same.

2s Complement Revealed

The only difference in the above examples is operand 2. In the first example, operand 2 equals 0000 0001 and in the second it equals 1111 1111. If we can find a relationship between 0000 0001 and 1111 1111, we will have found a way to subtract two binary values using addition. This relationship is called the "2s complement" and is easily formed as we said before.

For example, take again the value of operand 2 in the first subtraction above:

0000 0001	operand 2
1111 1110	(1) Complement
+ 0000 0001	(2) Add 1
1111 1111	(3) 2s complement of 0000 0001

All binary 2s complements may be switched back and forth, the two values actually being the 2s complements of each other:

1111 1111	2s complement of 0000 0001
0000 0000	(1) Complement
+ 0000 0001	(2) Add 1
0000 0001	(3) 2s complement of 1111 1111

Here are some more examples of binary numbers and their 2s complements.

1011 1110	1100 0110	1010 0101	Original values
0100 0001	0011 1001	0101 1010	(1) Complement
+ 0000 0001	0000 0001	0000 0001	(2) Add 1
0100 0010	0011 1010	0101 1011	(3) 2s complements

The 1802 microprocessor automatically adds the 2s complement of the subtrahend to the minuend in order to obtain the answer to a sub-

traction. Usually this is true of most binary processors, but even if all a processor could do was add, you could subtract values by adding the 2s complements of the subtrahends. Of course, you would need a way to first complement the number, but this is easily accomplished in most microprocessors.

Below are some sample binary subtractions demonstrating the use of 2s complement addition to arrive at the answers. Since we have already defined what happens to the DF register upon an overflow, the purpose of these examples is to observe the condition of DF following the operation.

Subtraction by 2s Complement Addition

Step	*Problem 1*	*Problem 2*	*Problem 3*	
(1)	0111 1001	0010 1101	1010 0101	Minuends
	− 0011 1010	− 1011 0110	− 1010 0101	Subtrahends
(2)	1100 0101	0100 1001	0101 1010	(1) Complement
	+ 0000 0001	+ 0000 0001	+ 0000 0001	(2) Add 1
	1100 0110	0100 1010	0101 1011	(3) 2s Complement
(3)	1100 0110	0100 1010	0101 1011	2s Complement
	+ 0111 1001	+ 0010 1101	+ 1010 0101	+ Minuend
	(1) 0011 1111	(0) 0111 0111	(1) 0000 0000	Answers
	DF D	DF D	DF D	

Three conditions have been represented above with the operations carried out in three steps. Step (1) states the problem. Step (2) finds the 2s complement of each subtrahend from Step (1). Step (3) adds each minuend to the 2s complement subtrahend from Step (2). The result of DF following the entire process is in parentheses to the left of the answers, which are assumed to be in the D register. The previous value of DF is unimportant and may be left undefined, since DF is *always* changed by an addition and its value is always 0 or 1 depending on whether there was an overflow (1) or not (0).

Consider Problem 3 first. Here we are attempting to subtract identical operands. Naturally, the result of this subtraction would have to be zero, so we may conclude that no borrow is needed to complete the operation. DF is equal to 1 following the 2s complement addition; therefore, it follows that DF equals 1 signals *no* borrow taking place.

Problem 1 demonstrates the condition minuend > subtrahend and Problem 2 demonstrates minuend < subtrahend. In Problem 1, we would expect the answer to be positive, since no borrow is needed to subtract a lesser value from a greater. DF is equal to 1 following the 2s complement addition. This is consistent with Problem 3—no borrow is needed in either case.

Problem 2 attempts a subtraction of a greater value from a lesser. The answer would have to be negative, of course, and you can see that

in this case DF is equal to 0 following the 2s complement addition. This would seem to be logical and perfectly suited to our needs. However, is the answer in the D register correct? Let's convert to decimal and see.

	Binary		
	0010 1101	Minuend	45
	− 1011 0110	Subtrahend	− 182
2s complement	0111 0111	Answer	− 137

But 0111 0111, the answer in binary to Problem 2, is equal to 119 in decimal, and the decimal subtraction indicates that the answer should be −137. Something is not right!

Let's try taking the 2s complement of the binary answer (119 decimal) in Problem 2:

Answer Problem 2	$0111\ 0111 = 119_{10}$
(1) Complement	1000 1000
(2) Add 1	+ 0000 0001
(3)	$1000\ 1001 = 137_{10}$

We have apparently obtained a positive value when we wanted a negative one, but the number 137 is correct. Since DF was equal to 0 only when the answer was negative, however, its value may be used to indicate the sign of the answer. If the answer is negative, its absolute value may be obtained by taking the 2s complement! We may now summarize the rules for subtraction assuming that the 2s complement of the subtrahend has been added to the minuend (automatic in most microprocessors).

Following Subtraction

DF = 1 — Answer positive—no borrow needed (minuend ≥ subtrahend)

* DF = 0 — Answer negative—borrow needed (minuend < subtrahend)

* If DF = 0, the answer is in "2s complement form" and the corresponding positive value may be obtained by 2s complementing the negative result.

When a programmer wants to be able to express negative numbers in binary, a decision must be made about the sign of the number—where it will be kept and how it will be represented.

We have just seen how the overflow flag, the one-bit DF register in the 1802, may be used to indicate the sign of a number. There is a more convenient (usually) way.

Instead of the overflow flag, most computer software routines designate the MSB of a binary value to be used as the sign bit. In an eight-bit byte, this would be bit 7, leaving bits 0 to 6 free to contain binary values. The sign now becomes an integral part of each byte, but since we have used up one precious bit, values are now limited to a seven-bit binary range.

Common convention is to let a 1 in bit 7 indicate a negative number. If bit 7 is 0, then the value is positive.

At first, this may seem overly complex. How does one keep the sign bit from interfering with additions and subtractions? Happily, when we are using 2s complement subtraction, the sign bit (if we choose to use bit 7 as the sign) will *always come out right without the programmer's having to know, worry, or figure what the sign is supposed to be*! Nothing could be more pleasant.

To prove that this is in fact the case, take again the first subtraction example presented in this section. Subtracting binary 1 from 0 resulted in the eight-bit answer 1111 1111. Note that bit 7 is a 1, indicating to us that the answer is negative. While we remember that the answer is negative (e.g., outputting a minus sign to the printer), taking the 2s complement of 1111 1111 gives us the absolute value of the negative number, in this case 1, which could also then be printed. This works for all negative binary values represented in 2s complement form.

When bit 7 is 0, the number is positive. To be consistent, we must assume the value zero to be positive. For this reason, it is possible to specify one additional negative binary number for which there is no positive equivalent. The following example shows the range of binary values when bit 7 is designated to be the sign of the number.

Binary		*Decimal*
1111 1111 to 1000 0000	=	−1 to −128
0000 0000 to 0111 1111	=	0 to +127
↑ ↑		
Sign bit Sign bit		

There are exactly 128 positive and 128 negative numbers capable of being represented in a single byte. But because one of the positive values is zero, it is not possible to represent the positive number 128 in eight bits.

Double Trouble

If you feel that a range of −128 to +127 decimal is limited, you are correct. Except perhaps to hold the score of a simple computer game or some other limited use, many programs will require a much greater range.

The arithmetic processes described in this section may be extended by using a process known as "double precision." Although we will stop with double precision, there is no reason to prevent further expansion to triple or higher precision using the techniques to be explained.

With double precision arithmetic, two adjacent bytes are used to represent values. Since the computer is still capable of operating on only single eight-bit bytes at a time, now each value will be manipulated one half at a time. More complicated, but not too much so.

Again it is common to designate the MSB of the first byte as the sign of the whole number. If we view the two bytes end to end as a single unit, this sign bit would occupy bit position 15 with the LSB of the second byte at position 0.

By using double precision with signed binary values, 15 remaining bits of the two bytes may contain values. The range for double precision numbers is similar to the range presented for signed single (one byte) precision arithmetic:

Binary		*Decimal*
1111 1111 1111 1111 to 1000 0000 0000 0000	=	−1 to −32,768
0000 0000 0000 0000 to 0111 1111 1111 1111	=	0 to +32,767
↑ ↑		
Sign bit Sign bit		

Again there is one absolute value more on the negative side than on the positive. Viewing zero as a positive number accounts for the discrepancy. The above range, by the way, is that used in many popular integer BASIC interpreters. Note that only bit 15 of the byte pair indicates the sign of the entire value contained in the remaining bits.

Now that we have decided on the format for representing larger numbers, the means must be developed for using them. The algorithms or methods to be described will demonstrate an approach to working with dual bytes *as if they formed a single unit*. In other words, we want results identical to those that could be obtained on a computer with a 16-bit accumulator.

Remember that following an addition or subtraction, the DF overflow register indicates if a carry or a borrow occurred during the operation. This action of the overflow register allows the double precision bytes to be linked together as one unit.

Most computers, the 1802 included, have instructions for adding and subtracting with or without taking the value of overflow into consideration. The process is automatic.

To add two binary double precision numbers together, first add the two lower bytes without consideration for the carry value in DF. Then add the two higher bytes with the value of DF, which may be 1 or 0 depending on the result of the first addition. Note that the programmer (or the program) never has to know what the value of DF is. The following example demonstrates a double precision add.

```
   1011 1010 0101 1000 =  BA58
 + 1100 0110 1100 1110 = +C6CE
 ---------------------   -----
  1/1000 0001 0010 0110 =  8126 (DF = 1 = overflow)
DF = (1)         (1)
```

To accomplish the above in machine language, the addition would be carried out in the following steps (hex notation used):

Step (1) 58 + CE=26 (DF =1 signals overflow)
Step (2) BA + C6 + (DF) = 81 (DF =1 signals overflow)

For precision addition of any length, the first addition of the least significant bytes ignores the value of DF. In the 1802, the instructions ADD and ADI could be used to accomplish Step 1.

To complete the addition, the high bytes to the left are added, plus the possible carry from the previous operation. The 1802 instructions used could be ADC or ADCI.

The *final* value of DF indicates whether the entire answer—over the length of precision being used—overflowed exactly as if only single bytes had been added together.

Double precision subtraction is practically a reverse of addition. The direction is the same, starting by subtracting the low bytes without regard for the value of DF, then proceeding to subtract high bytes with the possible borrow in DF from the previous operation. For example:

```
  1100 1000 0111 0010 =  C872
- 0101 0100 1010 0001  -54A1
-------------------------------
  (Add 2s compliment)
+ 1010 1011 0101 1111
-------------------------------
 1/0111 0011 1101 0001   73B1
DF = (1)       (0)
```

The above subtraction in a machine language program would be accomplished by using the following steps. The 2s complement addition is automatic with all subtraction instructions and is completely transparent to the programmer.

Step (1) 72 − A1 = B1
(DF = 0 signal a borrow)
Step (2) C8 −54 $-\overline{DF}$ =73
(DF = 1 signals no borrow—answer is positive)

Step (1) subtracts the two low bytes, ignoring the value of DF. The 1802 accomplishes this with the instructions SD, SDI, SM, and SMI. DF indicates if a borrow was needed to complete the subtraction.

Step (2) completes the double precision subtraction by subtracting the two high bytes while subtracting the opposite value of DF (indicated by the line above DF, which means "not DF"). The 1802 instructions that will accomplish this are SDB, SDBI, SMB, and SMBI.

The final value of DF indicates if a borrow was needed over the entire length of the answer. Since 2s complement notation is used, DF

may indicate whether the answer is positive or negative. The sign bit in the most significant position may be used to represent positive and negative values as previously explained.

In both cases, higher precision may extend the process and the possible range of the answer. The action is similar to the workings of a mechanical counter with a series of numbered wheels each with a metal tab next to the 0 digit. As each wheel completes a revolution, the wheel to the left is advanced one digit by the tab on the wheel to its right. Unlimited numbers of wheels could be strung together to expand the range of the counter.

The analogy isn't perfect—computers don't normally count to add. But the idea should help explain the concept of extended precision arithmetic. Bytes are linked together—an unlimited number is theoretically possible—by the action of the overflow register. Any size value of practical use could be represented in this fashion.

Floating point scientific notation is usually used to express very large (or very small) numbers. We will not discuss this here, but the process is a not too simple one of breaking numbers into a normalized integer part and an exponent (in binary). For most applications other than business or exacting scientific data processing, the extended precision techniques using integer (whole number) values will suit.

Logic Operations

Boolean algebra is a branch of mathematics that deals with very special methods of combining entities. Inside a microprocessor, the Boolean operators "AND," "OR," and "EXCLUSIVE OR (XOR)" are used to combine binary numbers with results that could not easily be duplicated without these special operations.

Table 2-1.
Truth Tables

1	2	3
"AND"	"OR"	"XOR"*
X Y Z	X Y Z	X Y Z
0 0 0	0 0 0	0 0 0
0 1 0	0 1 1	0 1 1
1 0 0	1 0 1	1 0 1
1 1 1	1 1 1	1 1 0

* Exclusive OR.

Table 2-1 lists three truth tables for the three logic operations. A truth table for our purposes is defined as a bit-by-bit chart of the results of a logic operation. X and Y are combined logically to produce the result Z (Table 2-1). Large binary values may be logically combined on a bit-by-bit basis. There are never any carries or overflows generated by the use of the logic operators, and adjacent bits never have any effect on their neighbors.

Following are definitions for two logic operations, the logical AND and the logical OR.

1. AND—a 1 will be produced if and only if both bits are equal to 1.
2. OR—a 1 will be produced if either or both bits are equal to 1.

The operation "EXCLUSIVE OR," which is normally written "XOR," is a special case of the logical "OR." It may be defined as follows:

3. XOR—a 1 will be produced only if one of the two bits is equal to 1.

Another way to define XOR is to say that zero will result if both of the inputs are equal (0 XOR 0 = 0; 1 XOR 1 = 0). Note in Table 2-1 that 0 results during an XOR operation when both inputs are equal to 0 and also when they are both equal to 1. Thus, the XOR is useful to test if a number is equal to some fixed value. If the result of XORing the two numbers is 0, then they are the same. If the answer is not 0, then the two inputs were different somehow.

The examples below demonstrate a few binary values and the results of the three logic operators. Remember, the combining is done on a bit-by-bit basis. Adjacent bit values have no bearing on the final outcome.

1110 0110	1110 0110	1110 0110
1010 1001	1010 1001	1010 1001
1010 0000	1110 1111	0100 1111
AND	OR	XOR

The three examples above combine the same two binary numbers and show the different results of the three logic operators. In a computer, these operations are useful in ways illustrated by the next three examples.

1110 0110	1110 0000	1110 0110
0000 1111	0000 0110	1110 0110
0000 0110	1110 0110	0000 0000
AND	OR	XOR

The AND operation above demonstrates an important and useful computer operation, "masking." By noting that the digit 0 ANDed with

anything produces 0, if you wish to eliminate any portion of a binary value, you only need to AND that portion with 0s. To retain a part of a binary number, even to mask all but a particular bit, AND the portion you want to keep with 1s. The portion that is retained following the AND is said to have "passed through," and the portion that was stripped off is said to have been "masked."

The OR operation performs the opposite function. Since any value ORed with a 0 will equal that value, the logical OR may be used to combine portions of binary values into single bytes. Likewise, any value ORed with a 1 will equal a 1. When such values will later be separated, perhaps with a logical AND command, the byte is said to be "packed," referring to the fact that it contains more than one distinct value. The OR example shows the results of packing two bytes into one. Note that the individual four-bit values have not been changed by the operation but they now have been combined into a single byte.

The XOR operation permits a fast and simple way to test whether a byte of unknown value is equal to some fixed amount. When two equal binary numbers are XORed together, the result will be equal to 0.

Another use of the XOR is demonstrated below.

	Binary		*Hex*
Original	– 1110 0110		E6
	1111 1111	XOR	FF
Answer	– 0001 1001		19
	XOR		

By XORing any binary number with hex FF, equal to binary 1111 1111, we have effectively and efficiently complemented the original value! Wherever there was a 1, there now appears a 0, and all 0s in the original are now equal to 1s. Adding 1 to the result would give the 2s complement of the original value, which, you will recall, may be used when two numbers are to be subtracted. The XOR function may be useful in computers not having a subtract command, although sometimes these same machines will contain a 2s complement instruction that automatically complements a number and adds 1 to it.

When a negative number is in 2s complement form, the above technique may be used to find the absolute value of that number.

The 1802 instructions that perform these three logical operations are AND, ANI, OR, ORI, XOR, and XRI. In all cases the result of the operation appears in the D register and DF is not affected.

Shifting

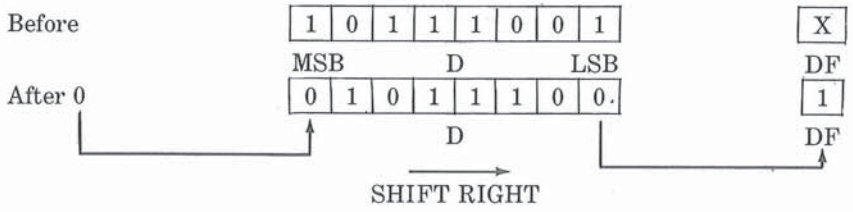
I have never seen a shiftless computer, although I wonder if some of those Las Vegas "computers," the one-armed kind, may not fit the

bill. Although the process of shifting is not truly a logic operation, it is included here. Shifting may be viewed as an arithmetic process providing a fast way to multiply and divide a binary value by powers of 2.

Shifting is exactly what it sounds like. When a binary number is shifted to the right, each bit moves one position in that direction. Shifting left produces the opposite expected result. Although there are variations from instruction set to instruction set, most microprocessors (including the 1802) contain forms of the following shifts.

1. Shift right
2. Shift left
3. Shift right with carry
4. Shift left with carry

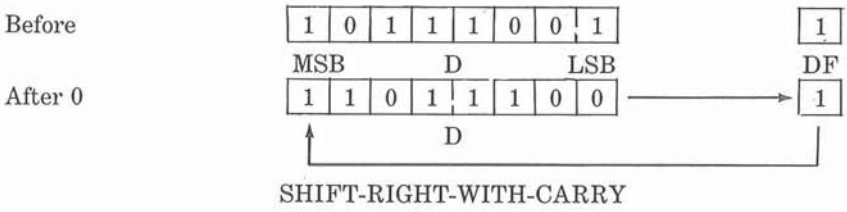
When a number is shifted to the right, the bit that originally existed furthest to the right leaves the accumulator and enters the overflow bit, the DF register in the 1802. On the other end, a 0 is shifted into the accumulator. Here is a graphic representation of a shift right.



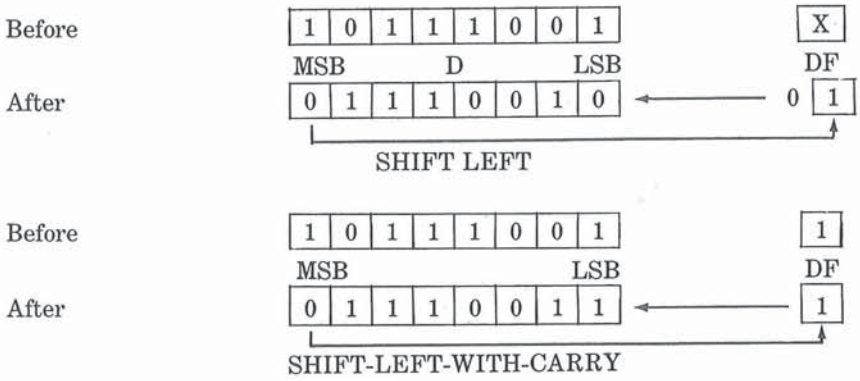
The X in the "before" DF box indicates that its previous value is unimportant. Following the shift right, the LSB of the byte in the D register is in DF and a 0 appears as the new MSB in D. The 1802 instruction performing this operation is the SHR command.

A shift-with-carry operation works in the exact same way with one important difference. The previous value of DF enters the D register as the new MSB, while at the same time the old LSB leaves D to enter DF. This resembles a circular operation that, if repeated nine times, would result in every bit of the byte having passed through DF only to return to their exact previous positions.

Here is an example of a shift right with carry performed by the 1802 instruction SHRC or its duplicate mnemonic RSHR.



It is important to realize that the value of DF is considered during a shift-with-carry and that its previous value will enter the accumulator from the opposite end of the shift direction. Shifting to the left is the opposite of the shifts to the right. Here are the same examples shifted left and shifted-left-with-carry from DF.



In the case of shift lefts, the MSB moves into the DF register on a simple shift left and a 0 is shifted into D from the right to become the new LSB. During a shift-left-with-carry, the old DF value is shifted into D as the new LSB, while the old MSB of D moves into DF at the same time. The corresponding 1802 instructions are SHL and SHLC with its duplicate mnemonic RSHL.

Shifting has various uses in computer programming but generally is employed to do one of the following things.

Bit Test by Shift

Individual bits may be shifted into the DF register for testing. In upcoming sections, you will learn certain 1802 instructions that cause program flow to be altered based on the value of DF. These instructions could direct the program operation depending on the individual bit values within bytes tested by shifting.

Unpacking and Packing by Shifting

Remember the logical OR function used in some cases to allow small individual values to be packed into one byte? When many values are needed to be stored, packing helps cut down on storage space requirements. Shifting is frequently used to move the values into position for packing, then to restore the original values following unpacking.

Arithmetic Shifts

Shifting any byte once to the left is the same as multiplying that byte by 2. Shifting twice is identical to 2 × 2; three shifts would produce 2 × 2 × 2. You can prove this to yourself by adding any binary value to itself. The result, basically a multiplication by 2, is the same as a single shift to the left.

Similarly, shifting to the right effectively divides any binary value by powers of 2. One right shift divides by 2; two right shifts divides by 2 × 2, etc. The proof of this is simply the converse of the multiplication above.

Be especially careful of overflow when multiplying and dividing by this method. Remember, bits leave the D register in the direction you are shifting. Even though it would be correct to assume eight shift lefts to be equivalent to a binary value times 2^8, if you do not do something with the bits that are shifted out, you will end with 0, and not what you expected. As in adding and subtracting large values, all processors have ways of dealing with this problem, and double-precision shifts may be used to "catch" bits as they leave D, shifting them from DF into another byte.

Other uses for shifting are discovered frequently, and you will see the instructions used in the most unlikely places. I remember a short program I once came across that calculated the sine of an angle almost purely by multiple shifting! Clearly, a shiftless computer would not have the power of one with these important instructions.

The appendix subroutine library of this book contains several examples of double-precision arithmetic and shifting routines. For a complete understanding of the processes in the preceding sections, you should study the programs carefully. The best way is to follow each step of a routine by writing on paper what the values in the registers and in D are before and after instructions are executed. This "walk through" will be more instructive than simply running the program on a computer and viewing the results.

Program Flow Operations

In terms of programming potential, instructions that alter the flow of a program lend the most power to a binary brew.

Two types of jump or branch instructions exist in all computers: conditional and unconditional. The 1802 names its jump instructions "branches," but the actions are similar to jump commands on other microprocessors. In addition to branching, the 1802 contains a special

set of skip instructions that allow a rather limited *relative* type of jump without having to specify a destination address.

Some computers allow relative branching plus or minus a certain number of bytes away from the branch origin. Unfortunately, the 1802 does not have this feature and it is always necessary, except for the skip instruction, to specify an address to which the branch is to occur. (Relative branchings may be simulated on the 1802 but at a high expense of memory space.)

If you have had experience programming in a high-level language such as BASIC, you have probably seen GOTO constructions that cause a program's execution to take a course dependent on various conditions, values of variables, keys pressed, etc. If you have attempted to follow the paths of a complex GOTO-ridden program, you will appreciate the prudence of using jumps sparingly.

Although it is usually impractical (possibly impossible) to write machine language programs without any jumps or branches, it is wise to use these powerful instructions only as necessary. Better that a program be constructed modularly using subroutines for each cog of the clock than that the parts be strung together with branches. Every time you use a branch instruction ask yourself if it is absolutely needed. If the answer is no, get rid of the branch. When debugging time rolls around—and it will, it will—you will appreciate the simplicity of a straightforward program structure.

Unconditional branches direct program flow to a specified address regardless of any values, conditions, or actions occurring or existing. When the branch instruction is executed, the branch to the specified address will always be taken.

Conditional branches allow the program to make a decision on whether to take the branch or not. Each conditional branch instruction is strictly defined to be dependent on a specific value or condition in a specific way. For example, one of the 1802's conditional branch instructions (BNZ) causes the branch to be taken *only* if the value in the D register is not equal to zero.

When a conditional branch decision point is reached in the program, an evaluation is made by the computer whether to take the branch or not. If the decision is to branch, then the jump is taken exactly as if the branch instruction had been an unconditional one. That is, none of the evaluated conditions are changed by the action of branching.

If the decision is not to branch, then the program continues in the normal forward direction executing the next instruction in line. The effect of not branching is identical to the effect that would occur if the branch instruction did not exist at all (except for the processing time needed to make the decision whether to jump). When a branch is not taken at the bottom of a loop—usually terminating the loop—the program is said to "fall through" that point.

In the 1802 there are short branches and long branches. An easy way to distinguish between the two op codes is to remember that all short branches and the one skip begin with 3 (3N) and all long branches and long skips begin with C (CN).

Short branches are limited to jumps extending no further than the boundaries of the memory page containing the short branch instruction. A memory page is arbitrarily defined as a 256-byte block, each byte's location in the block having the same high eight-bit memory address. For example, the hexadecimal addresses 0100-01FF and 1A00-1AFF define two unique blocks of memory, the first on hexadecimal page 01 and the second on the page 1A. Note that for any 256-byte memory page the low eight bits of the address are in the same range from 00 to hexadecimal FF. Also be aware that 00, *not* 01, is the first page of memory starting at 0000.

A short branch instruction may direct program flow to any address within any page of memory. Short branch op codes are followed by one-byte addresses (00-FF) telling the computer where the branch is to proceed. The high eight bits of the address are not important and are not specified. Because of this, program sections may be designed to run identically, regardless of what page the code is located on. Such code is said to be "page relocatable." (Except for subroutine calls and the pointers to data tables, the assembler in Chapter 4 is page relocatable due to its avoidance of long branch instructions.)

Remember, short branches may never cause program flow to cross page boundaries.[1] Don't make the mistake of branching to 0500 from 04F0 with a short branch to location 00. In that case, what you intended to go up will come down, a particularly distressing possibility. Short branching to 00 from 04F0 would cause program flow to be directed to 0400 because the page designation, the 04, is not changed by the short branch. Only the lower half of the address is affected.

The action that causes branching in the processor is not a complex one. Whatever address is specified following the op code is simply inserted into the register currently designated to be the program counter, and execution will then begin from the new address formed in that register. Short branch instructions cause this single byte, called the argument to the instruction, to be inserted in the low half of the program counter. The program itself may insert values into the program counter too, another way to effect a short branch and an advanced programming technique used in 1802 interpretive languages.

Long branches require a two-byte argument following the op code. With long branches, these two bytes are inserted into both halves of the program counter, thus allowing any 16-bit address to be specified from

[1] The trick of splitting a short branch across a page boundary—the op code on one side of the fence and the branch address on the other—is not recommended, although this would allow short branching forward into a following page.

hex 0000 to FFFF. Long branches may, therefore, branch to any new location in addressable memory without regard for page boundaries. The address is specified in the correct order, that is, high byte first followed by the low byte. (This may seem obvious, but many microcomputers require 16-bit addresses to be specified in reverse byte order.)

Here are some examples of unconditional long and short branches as they would appear in a machine language program.

```
0249  30 04     BR    04    ; Short branch to location 0204
024B  C0 07 00  LBR   0700  ; Long branch to location 0700
024E  30 4E     BR    4E    ; Short branch to location 024E
```

Examine closely the third program line at memory address 024E. The instruction BR (op code 30) is an unconditional short branch instruction. Since the byte immediately following BR's op code is hex 4E, this instruction is calling for a branch to location 024E. But that is where the BR instruction itself is located! The effect is to halt the program in an unending loop, the computer constantly, tirelessly, reliably executing the same branch instruction over and over and over until some kind soul kicks out the plug (or flips a reset switch).

Actually this "branch-to-itself" technique is a useful way to end a program or to insert a stop at a location when the program counter is unknown.

There are two unconditional branch instructions in the 1802 set. We have seen them both, the BR short branch and the LBR long branch.

All of the rest of the branches are conditional types. The short branch conditionals are BZ, BNZ, BDF, BNF, BQ, BNQ, B1, BN1, B2, BN2, B3, BN3, B4, and BN4. The long branch conditionals are LBZ, LBNZ, LBDF, LBNF, LBQ, and LBNQ. Each of these is explained in detail in Chapter 3.

It would seem logical to use all long branches, since these allow a much greater range than the page restricted shorties. But most 1802 software uses short instead of long branching for some very good reasons.

For one, many 1802 computers use the 1861 video display output chip in their design. The timing requirements of this chip restrict the programmer to instructions using only two machine cycles for execution. Machine cycles are derived from clock pulses applied to the processor

and are what cause the computer to run a program. Long branches and long skips (in fact all of the CN-type instructions but none of the others) take an extra machine cycle for execution. These three cycle instructions affect the critical timing of the 1861 chip, causing the display to jump and jitter. Using the three-cycle instructions can't hurt the display, but the jiggle doesn't do much for the eyes of the programmer (or the programmer's poor victims). The assembler in Chapter 4 does not use any three-cycle instructions for this reason.

Another motive for choosing short branches over long ones is to gain speed. Because only two machine cycles are used for execution, short branches will execute faster than long ones. Significant timing reductions may be had by using short branches especially in loops where the same instruction may be executed hundreds or more times. Using a long branch to a location within the same memory page is wasteful not only of time but also of a memory byte and at no advantage to the program.

Limiting yourself to short branches will also force a modular structure into your programming. Jumping around indiscriminately in memory is a sure way to add confusion to a program, possibly leading, when the program fails, to the uncontrollable urge to jump around on the computer—wearing lead boots.

After a program is debugged and has been thoroughly tested and retested, some long branches in place of sub calls at the ends of subroutines may speed up the program. This programmer prefers to keep a modular design over just about any consideration, but such an optimization technique may prove useful in some software.

Long branches are good for starting programs at widely separated memory locations. When used sparingly, they may prove to be more efficient than another structure, depending on the application.

The 1802 skip instructions are unusual in computers. Because all but one of these are of the three-cycle type, however, their use is also quite rare, especially in systems with the 1861 video output chip.

A two-cycle skip exists, the SKP (op code 38) instruction, which also goes by the name NBR. When executed, SKP causes the following byte to be passed over. That's all it does but it is sometimes handy in constructing loops. It may also replace the op code for any of the short branch instructions, providing a one byte way to disable a branch during debugging.

The rest of the skips are long skips. When executed, a long skip causes the next two bytes to be passed. The unconditional long skip is the LSKP, which also, like SKP, comes with a duplicate mnemonic,

NLBR. LSKP may be inserted in place of any of the long branch op codes to disable a branch for debugging purposes. In that case, the LSKP, or more correctly the NLBR, means do *not* branch to the following 16-bit address.

In addition to the two unconditional skips, there are seven conditional long skips. These are LSZ, LSNZ, LSDF, LSNF, LSQ, LSNQ, and LSIE.

Each one of these long skips will cause the next two bytes to be passed over based on a defined condition existing at the time the instruction is executed. If the condition is found not to exist (i.e., evaluates as false), then the skip does not happen and the following instruction or two instructions just after the skip will be executed.

Perhaps the single most important use of skip instructions is in the design of relocatable code. Such code will run regardless of its position in memory. However, because of the limited range of the skips, they will not usually take the place of branching.

The following example shows the most common use of a short branch as a control in a loop. Note that the routine begins with an SKP instruction, causing the first instruction of the loop to be passed at the start.

```
        SKP          ;Skip the next instruction
LOOP:   DEC  RF      ;Subtract 1 from loop count in RF.0
        LDA  RB      ;Get a byte from location @ RB
        STR  RA      ;Store at location addressed by RA
        INC  RA      ;Add 1 to address in RA
        GLO  RF      ;Get the loop count in RF.0
        BNZ  LOOP    ;If ≠ 0, branch back to LOOP
                     ;(continue, or fall through on RF.0 = 0)
```

This is the first actual assembly program presented in this book so some of the instructions and formats have not been covered. But we are concerned only with the action of the SKP and the BNZ. The SKP causes the DEC RF instruction not to be executed the first time. Instead, the instructions following DEC RF are performed. (You do not have to understand what these do just yet, but the action is to transfer bytes addressed by RB to a new location addressed by RA.)

The GLO RF instruction brings the value of the low eight bits of register RF into the D register. Following this, the conditional branch BNZ is executed. This instruction evaluates the D register, and if the value there is *not* equal to zero, the branch will be performed. If D is not zero, program flow would be directed back to the location labeled "LOOP" in the example. At that point the DEC RF, which had previously been skipped, would be executed, causing the value in register RF to be decremented by one. Each time through the loop, the value of RF is tested (in D) by the branch instruction. If it is not 0, a jump will occur, RF will be decremented, and the code in between the top and bottom of

the loop will be performed. When RF finally equals 0, the branch will not be taken and the program will continue past the BNZ instruction. Therefore, the loop is executed one time more than the value in register RF. (If RF = 0 at the start, the loop is still executed at least but only once.)

The following example shows a more basic construction for loops. LDI and PLO have not been covered yet. Their action is to set a value into RF that will indicate the number of times the loop is to be executed.

```
        LDI     $NN     ;Load into D the immediate value NN
        PLO     RF      ;Put NN into low part of RF as loop count
LOOP:   (       )       ;    whatever code is to be performed
        (       )       ;    during the loop goes here. It
        (       )       ;    will be executed at least one time
        DEC     RF      ;Subtract 1 from RF loop count
START:  GLO     RF      ;Get the value of RF for testing
        BNZ     LOOP    ;If ≠ 0 yet, branch to LOOP. Else fall
        (continue)      ;    through here and continue program
```

Most important to the functioning of the loop are the last three instructions. If you aren't sure what these instructions do, take a moment to read about them in the next chapter.

For this type of loop, always subtract 1 from the loop count (DEC RF), get the value of the loop count into D for testing (GLO RF), then branch back to restart the loop if D ≠ 0.

Unlike the loop that began with the SKP, this loop will be executed exactly the number of times specified in register RF. In the event RF is set to 0 before entering the loop, the code will be executed 256 times, not 0 times. This is because RF is decremented *before* it is tested, and subtracting 1 from 0 will cause the value in RF to "wrap around" to its highest eight-bit value, hex FF.

A way to cause the loop not to be executed when RF is set to 0 is to start execution of the loop following the code to be performed inside the loop. This corresponds to the location "START" in the example. A branch to START (BR START) would be inserted between the PLO RF instruction and the instruction at location LOOP. If RF equals 0, then the loop will not be performed, but now the maximum loops possible using a single-byte loop counter are 255, not 256 as before.

This loop construction is so common that it will eventually become automatic in your programming. Be careful to bring the loop count into the D register before testing if it is equal to 0. Other processors allow register values to be tested for 0 directly through the use of an automatically set flag. No such capability exists with the 1802, which requires byte values to be brought into D for evaluation.

Other common uses of branching include testing the status of individual bits in a byte. Again, the byte containing those bits must first be brought into the D register. Assuming this has been done, the LSB

of the byte may be tested by first moving it into DF by shifting. For example:

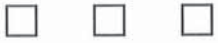
```
TSTBIT: SHR          ;Shift LSB of D into DF
        BDF  ONE     ;If DF = 1, then short branch to ONE
        BR   TWO     ;If DF = 0 (≠ 1) then branch to TWO
```

The above code will branch to either location ONE or location TWO depending on the status of the LSB of the byte in D.

Another way to accomplish this, but at the expense of an additional byte, is to use the logical AND to mask out all but the bit we want to test. The following example branches to one of two locations depending on the status of bit 2. Again, D is assumed to hold the byte containing that status bit.

```
TSTBIT: ANI  $04   ;Logically AND D with binary 0000 0100
        BNZ  ONE   ;If result is not zero, short branch to ONE
        BR   TWO   ;If result is zero, short branch to TWO
```

Only if bit 2 is set to 1 will the logical AND cause the result in D to be unequal to 0. Individual bits or combinations of those bits may be tested in this manner. The programmer may assign significance to the status of each individual bit so that it is possible to keep eight status evaluations, or "flags," in a single byte.



Apply branching with respectful reserve and it will become a most useful programming tool. Break a program into small chunks, each chunk with a single uncomplicated function to perform and all chunks linked together with subroutine calls. Try to keep all branching inside those chunks—your programs will be easier to read, debug, and understand, and you will have used branching in the correct way—as little as possible.

Operations on Memory

Some of the instructions and programming examples explained in previous sections caused bytes in the D register to be manipulated or evaluated (e.g., in the case of branching). Arithmetic instructions use the value in D as one of two operands. A way must exist for loading and storing bytes to and from memory locations and the D register. This section covers those operations.

Loading and storing values in computers gives the programmer the ability to move bytes in memory around and to put values in known places for later retrieval. In the 1802, all load instructions bring values into the D register, while all store instructions take the value of D and

put it into memory. In both cases, one of the 16 general-purpose scratch pad registers holds an address pointing to the desired memory location. Except for R0 with the LDN instruction, any of the 1802 registers may reference a memory address for a load or store operation.

Loading a value from memory does not change the value at its memory location. After bringing a value into the D register from a specified address, that value still exists at the same memory location as it was before. Loading the same value over and over does nothing to change that value. It is constant. Actually "copy a value into D" would be more descriptive of what a load instruction on the 1802 does. Other computers follow this maxim.

Storing values in memory locations is similar except for direction. When storing a value from the D register, that value is unchanged in D by the process of putting it into memory. Setting successive memory locations to some value is therefore possible without having to reset the D register following execution of the store instruction. "Copy D into memory" is highly descriptive of what a store operation accomplishes.

The 1802 instructions used to directly manipulate and work with memory bytes are LDN, LDA, LDX, LDXA, LDI, STR, and STXD. There are five load instructions and two store commands in the set.

LDI is one of the most often used instructions in 1802 software. This *load immediate* instruction will cause the D register to be set equal to a constant specified just following the op code for the LDI. Like the short branches, LDI requires a single-byte argument, the value of which is loaded into D. The following example loads FF into the D register.

```
0450  F8  FF   LDI  $FF  ;Load FF into D
```

The other four load instructions operate via registers addressing memory locations from where values are to be loaded. (LDI always uses the program counter register to "fetch" bytes into D.) LDN and LDA specify which of 16 registers (15 for LDN as R0 is "illegal" for that instruction) is to be used as a memory pointer. This is done by setting the second digit of the op code to the desired register hex number. LDX and LDXA also require one of the 16 registers to address memory but in a different way. For these two instructions, a separate 1802 register, the X register, is set to specify one of the 16 scratch pad registers. If X is set to 2, for example, then register R2 will be the active pointer during execution of LDX and LDXA. How and why to set X is discussed later.

Except for this distinction, LDN, LDA, LDX, and LDXA operate similarly. LDN and LDX bring a byte into the D register from a designated address. The register holding that address is not changed, only the value in D is subject to change. LDA and LDXA (the "A" means "advance") also bring a byte into the D register, but with these two mnemonics, the register holding the address of that byte is automatically incremented by one after loading. This provides a simple way to load bytes from sequential memory locations, one after another into D.

The two store instructions also require a register set to an address, in this case pointing to the destination where bytes are to go. STR causes D to be placed into memory, and like LDN and LDX, the addressing register is not changed. STXD stores bytes using the X register to select in this indirect manner one of the 16 registers to be the pointer. The D of STXD means "decrement" and, following execution of this instruction, the value of the specified register will be one less. Thus, bytes may be stored sequentially at decreasing memory addresses with a single byte instruction.

Loading and storing bytes in memory as a way to preserve results of operations is an obvious use of this instruction sub set. Blocks of data may be examined by loading values into D, video graphics displays may be animated by changing selected bytes, and generally anything requiring the manipulation of memory may be performed by these instructions.

Not so obvious are programming techniques such as stack control using this instruction group. A stack is a block of memory set aside to contain values that the programmer wants to retrieve later. By using the LDX, LDXA, STR, and STXD instructions, bytes may be put onto the stack (pushed) then returned to D (popped) when needed. In a stack, the last byte pushed is the first one to be received in a way that requires strict discipline in the programming of code. We'll see more of the all-important stack in a later section.

Load-and-store instructions—in combination with others—are often used in a fashion that gives the structure of data in memory an importance perhaps equal to the data itself. There are many possibilities: tree structures in which each element of data contains a pointer, an address to the next element, as well as the data itself; sorted lists in which data is rearranged into numerical order; other types of linked lists; and buffers whose jobs are to serve as halfway houses for data on its way in or out of a computer. In all of these processes, the load-and-store instructions will play major roles.

Load-and-store instructions may also be used to relocate other instruction codes, which, except when being executed by the computer, may be viewed simply as data. This is an advanced technique employed in operating systems to use available memory efficiently during booting of disk control routines and other fancy stuff. For such a method to operate, however, the code needs to be designed in a relocatable way, and this may be easier to do on some computers than others.

A word of advice needs to be included on another possible use for a store instruction. The use is the design and programming of self-modifying code. The advice is "don't."

A self-modifying program is one that actually creates, through various operations, the instruction codes that are eventually to be executed along the line. Store instructions place this code into the proper memory locations. At least that's the theory. In practice, this can be a most difficult and frustrating technique to manage. Algorithms taking on a self-modifying structure as their basic building block may lead to some brain-fusing "chicken or egg" debugging walk-throughs for the programmer.

Simulator programs that duplicate their own microprocessor on which they are written may make extensive use of self-modifying code. These programs are useful when debugging software and usually allow single stepping and register displays as well. Note that any self-modifying routine must exist in RAM for it to operate.

With the admonishment never, never to use self-modifying code, here's one way to write such a demon. (Actually, self-modifying routines can be fun to experiment with—it's exciting to think that a program can help to create itself—but be prepared for the worst if you plan to try some of this forbidden fruit!)

The following demonstration takes an instruction from the low part of register RF and inserts it into itself for execution. R3 is the program counter and is used as the pointer to store the instruction for execution just before the computer gets to the memory location where the stored instruction is to be obtained (gulp!).

```
MADMOD: GLO  RF   ;Get instruction from RF
        STR  R3   ;Store at the following memory location
        ??        ;Execute mystery instruction
        LDI  $23  ;Load immediate a 23 (DEC R3) byte
        STR  R3   ;Store at the following memory location
        ??        ;Stop by continually decrementing R3
```

This routine has as much right to be in this book as the devil has to appear in the Bible. I hope I've made my point.

Operations on Internal Registers and Miscellaneous

Throughout the book we have been using the 1802's scratch pad registers as address pointers and have been suggesting that data may be held by a register for later use. Remember that a register is similar to a location in memory except that it exists inside the microprocessor electronics.

The 1802 is fortunate to have sixteen 16-bit registers—some microprocessors have only three or four eight-bit registers. In those computers, memory locations are usually set aside to function as the processor's registers, something a bit awkward to accomplish on the 1802.

Setting registers to known values is basic to computer programming. For example, when storing the value of D in memory, a register

is used to specify the address where that value is to go. The programmer needs a way to load a register with that address. Similarly, because of the all-important D register, values in the scratch pad registers may be pulled into D for manipulation, testing, or outputting to somewhere else.

1802 registers are split down the middle into two halves, upper and lower. Each half is eight bits long, and it is possible for the programmer to manipulate and make use of any register either as a whole 16-bit entity or as two individual eight-bit units.

Each half of a register is specified using the following notation.

—8 bits high—	—8 bits low—
RN.1	RN.0

REGISTER RN

"R" of course means "register" and "N" stands in place of a hex digit specifying which particular register, 0–F. R4 would refer to register 4, RA to register A, etc.

The period that may follow RN is in turn followed by either a 1 or 0 specifying either the high or the low portion of register RN. RB.1 would mean the high or leftmost eight bits of register RN; R0.0 means the low or rightmost eight bits of register R0, etc. If no qualification is attached to the register, then the entire 16 bits of that register are being referred to.

The distinction between the high and low portions of a register are important to a programmer's allocation of registers to be used during a program run. RE.1, for example, may be intended as a holder for a specific value during a subroutine, while its partner half, RE.0, is relegated to an altogether different use, perhaps as a loop counter. Or the entire register RE may be used to hold a 16-bit address that will point to a block of data the program will need to access.

But the qualification is for the programmer's notes only. The computer understands which register half is to be acted on by virtue of whatever instruction is being executed.

In assemblers that allow labels to be used instead of addresses, the distinction between the high part and low part of an address may also be specified in the same way. Still the notation is not directly understood by the computer; rather, it is a convenient way to instruct the assembler to which half of the 16-bit value it is to use. ASMBLR.1 would mean the page or high eight bits of the address where that routine is to be found. With many computer instruction sets, unlike the 1802, this notation becomes less important and in many assemblers it is not even used.

There are four 1802 instructions used to transfer eight-bit values to and from any of the 16-bit registers. These are GLO, PLO, GHI, and PHI. The action of transferring is similar to the load-and-store instruc-

tions with the D register either accepting a value from a register or holding a value that is to be inserted into a register.

Each of these four may only operate on a single eight-bit half of a register. Therefore, in order to set a 16-bit register to some value, at least two instructions will have to be executed.

PLO and PHI transfer (PUT) the value of D into the low or the high part, respectively, of register RN. "N" is specified as the second digit of these instructions' op codes. The op code for a PLO R2 would be A2, for example, while A9 would specify the same PLO instruction but for register R9. The same holds true for the rest of the instructions in this group, but with the use of an assembler, the programmer may not have to be aware of this process.

GLO and GHI transfer (GET) bytes out of the low or high parts of registers placing the value into D. Again, only one-half of any register may be brought into D at a time.

Just as with load-and-store operations, these four register instructions do not alter values in their original locations. Putting D into a register does not change D. Bringing a register value into D leaves that value unchanged in the register. "Copy" would again be a more descriptive word for the action of these instructions.

In the loop examples in the section on branching, a GLO RF instruction was used to bring the low half of register RF into D to test if those eight bits were equal to zero. The action of the loops in those examples is dependent on the ability of the program to know at just what point RF goes to zero.

Another use for registers is to serve as holders of intermediate values that will be used by other parts of the program. The value could be stored in memory, but many times it will be faster to simply put a number into an unused register half, then get it back again later when the value is needed. In this way registers may act as instantly accessible variables. Frequently, subroutines will pass values back to other routines by putting the results of their operations into a register or registers.

When a register is to be used to hold the address of a memory location, that address is put into the register most commonly in the following way.

```
SETBUF:  LDI   $04   ;Load immediate value $04 into D
         PHI   RB    ;Put value of D in RB.1
         LDI   $00   ;Load immediate value $00 into D
         PLO   RB    ;Put value of D in RB.0
```

1802 MICROPROCESSOR INSTRUCTIONS

ADC	74	IDL	00	ORI	F9
ADCI	7C	INC	1N	OUT	6N, 1≤N≤7
ADD	F4	INP	6N, 9≤N≤F	PHI	BN
ADI	FC	IRX	60	PLO	AN
AND	F2	LBDF	C3	REQ	7A
ANI	FA	LBNF	CB	RET	70
B1	34	LBNQ	C9	RSHL	*7E
B2	35	LBNZ	CA	RSHR	*76
B3	36	LBQ	C1	SAV	78
B4	37	LBR	C0	SD	F5
BDF	*33	LBZ	C2	SDB	75
BGE	*33	LDA	4N	SDBI	7D
BL	*3B	LDI	F8	SDI	FD
BM	*3B	LDN	0N	SEP	DN
BN1	3C	LDX	F0	SEQ	7B
BN2	3D	LDXA	72	SEX	EN
BN3	3E	LSDF	CF	SHL	FE
BN4	3F	LSIE	CC	SHLC	*7E
BNF	*3B	LSKP	*C8	SHR	F6
BNQ	39	LSNF	C7	SHRC	*76
BNZ	3A	LSNQ	C5	SKP	*38
BPZ	*33	LSNZ	C6	SM	F7
BQ	31	LSQ	CD	SMB	77
BR	30	LSZ	CE	SMBI	7F
BZ	32	MARK	79	SMI	FF
DEC	2N	NBR	*38	STR	5N
DIS	71	NLBR	*C8	STXD	73
GHI	9N	NOP	C4	XOR	F3
GLO	8N	OR	F1	XRI	FB

*These instructions have duplicate mnemonics.

The effect of these four instructions is to set register RB to the 16-bit value 0400, an address in memory. Some future operation may execute, for example, an STR RB or an LDA RB, which would transfer bytes to and from location 0400 via register RB.

When many registers need to be set at the beginning of a program or a subroutine (programmers say "initialize the registers"), if some registers are to have the same values, some space may be saved in the following way:

```
INIT: LDI  $06  ;Load immediate value $06 into D
      PHI  R8   ;Put value of D into R8.1
      PHI  R9   ;Also in R9.1. (R8.1 = R9.1 = 06)
      LDI  $00  ;Load immediate value $00 into D
      PLO  R8   ;Put value of D into R8.0 (R8 = 0600)
      LDI  $20  ;Load immediate value $20 into D
      PLO  R9   ;Put value of D into R9.0 (R9 = 0620)
```

Because the value of D does not change when it is copied into a register, as many registers as needed may be set to D's current value by executing successive PHI RN instructions.

To discuss all the possible uses for transferring bytes to and from registers would be monumental (and extremely exhausting). Many examples of these instructions exist in the assembler in Chapter 4, and the reader is encouraged to examine the listing for a better understanding of their actions.

Two other instructions in the 1802 repertoire act directly on the 16-bit scratch pad registers. They are extremely simple to use and understand.

INC adds (increments) 1 to the 16-bit value in register RN.
DEC subtracts (decrements) 1 from the 16-bit value in register RN.

You now know everything there is to know abount INC and DEC!

Note the definitions specify the "16-bit value of RN." These two instructions act without regard for the division of each register into two eight-bit halves. Incrementing and decrementing are performed on the *entire* register, providing an ability to count by one all the way up to (or down from) FFFF if desired. (Not so difficult for a computer to do, of course.)

The following subroutine demonstrates one way to take advantage of this 16-bit action.

```
TIMER:  LDI   $FF    ;Load immediate value $FF into D
        PHI   RF     ;Put D in RF.1
        PLO   RF     ;And in RF.0 (RF = FFFF)
LOOP:   DEC   RF     ;Decrement RF by 1 over 16 bits
        GHI   RF     ;Get RF.1 to test
        BNZ   LOOP   ;If RF.1 ≠ 0, branch to LOOP
        RETN         ;Return from subroutine
```

This timing loop works by first setting RF to its highest possible value (but any value could be used for shorter timing intervals). By decrementing RF, then testing the *high* eight bits of the register rather than the low eight bits (RF.0) as normal for loops, the maximum number of loops are performed before the subroutine ends. To understand better how the timer operates, work out the values of RF and D on paper as you

1802 HEX CODES

00	IDL	6N, 9≤N≤F	INP	C6	LSNZ
0N	LDN	70	RET	C7	LSNF
1N	INC	71	DIS	C8*	LSKP
2N	DEC	72	LDXA	C8*	NLBR
30	BR	73	STXD	C9	LBNQ
31	BQ	74	ADC	CA	LBNZ
32	BZ	75	SDB	CB	LBNF
33*	BDF	76*	RSHR	CC	LSIE
33*	BGE	76*	SHRC	CD	LSQ
33*	BPZ	77	SMB	CE	LSZ
34	B1	78	SAV	CF	LSDF
35	B2	79	MARK	DN	SEP
36	B3	7A	REQ	EN	SEX
37	B4	7B	SEQ	F0	LDX
38*	NBR	7C	ADCI	F1	OR
38*	SKP	7D	SDBI	F2	AND
39	BNQ	7E*	RSHL	F3	XOR
3A	BNZ	7E*	SHLC	F4	ADD
3B*	BL	7F	SMBI	F5	SD
3B*	BM	8N	GLO	F6	SHR
3B*	BNF	9N	GHI	F7	SM
3C	BN1	AN	PLO	F8	LDI
3D	BN2	BN	PHI	F9	ORI
3E	BN3	C0	LBR	FA	ANI
3F	BN4	C1	LBQ	FB	XRI
4N	LDA	C2	LBZ	FC	ADI
5N	STR	C3	LBDF	FD	SDI
60	IRX	C4	NOP	FE	SHL
6N, 1≤N≤7	OUT	C5	LSNQ	FF	SMI

*These instructions have duplicate mnemonics.

"walk through" the subroutine. (Warning—you'll need a *lot* of paper to go to the end and I don't suggest you try to finish this project unless you have a few months of time to kill!)

One further note about INC and DEC. Decrementing the value 0000 results in FFFF, and incrementing FFFF will produce 0000. Thus, the registers act in "wraparound" fashion. The overflow DF register is not affected by these two conditions—in fact there is no way to tell anything at all about the values in registers without first bringing those values with GLO and GHI into D.

Miscellaneous

There are a few 1802 instructions that refuse to be placed into categories. (Stubborn little guys.) Like most processors, the 1802 has several commands that take advantage of their computer's unique structure.

Except for NOP, the rest of the instructions discussed in this section are probably unique to the 1802. NOP does nothing, and believe it

or not will be used quite often. Most computers contain an NOP (no operation) instruction that will simply hold a place in a program for a later instruction, or replace an instruction of questionable action during debugging. Fill a computer with NOPs and run the program and absolutely nothing will happen. Most beginners to programming are surprised to find such an instruction and then are equally surprised by how many times they need to use it.

IDL is another 1802 specialty command (see Chapter 3). It causes the program to wait for an interrupt to occur, an important feature of programming we'll see in a moment.

Four unique 1802 commands make use of a different bank of registers that exist in the 1802. These registers, X, P, T, and Q, exhaust the register supply in the processor and, unlike the general-purpose scratch pad registers, may be used only in restricted and defined ways.

Actually, Q is not truly an internal register—it is rather thought of as a flip-flop, an electronic switch that can be turned on or off. Viewed as a register, Q many contain the single bit values 1 or 0. Viewed as a switch, Q may be connected to external devices providing a way to switch those devices on or off under software control.

The instructions affecting Q are SEQ and REQ, which turn Q on and off, respectively (see Chapter 3 for more details on Q).

The X and P registers are two of the most important in the 1802 structure. Each is four bits wide and each may be set to contain one of 16 binary values from 0000 to 1111. This corresponds to the hex digits 0–F, which in turn correspond to each of the sixteen 16-bit general-purpose scratch pad registers.

The P register designates which of the scratch pad registers is to be used as the program counter. If P is equal to three, then the program will run at the address in register R3. If R4 contains a different address, setting P to four would cause the program to begin executing at that address. Then setting P equal to 3 again would take control back to where R3 pointed to when P was changed to 4.

Because switching program counters leaves the previous counter pointing to wherever P was changed, setting P provides an acceptable means for jumping around in memory at will. It is acceptable, as compared with long branching, because now there exists a way to get back to where the jump occurred where with LBR this would not be possible.

The 1802 instruction associated with the P register is SEP. Performing SEP R3 causes R3 to become the program counter. SEP RC would start the program running at the address in register RC.

R0 is the normal 1802 program counter when starting a run. However, since R0 is usually reserved for other uses, one of the first operations of any 1802 program is to switch to another program counter register. R3 is most often chosen as the new program counter, and the

reader is advised to stay with this convention or else software that is not compatible with other 1802 systems may result. R3 is the program counter for the assembler in Chapter 4.

The following routine at the beginning of an 1802 program will change program counters from R0 to R3.

```
0000  90          GHI   R0       ;Get R0.1 (=00 as P=0 here)
0001  B3          PHI   R3       ;Put in R3.1 to be new PC
0002  F8 06       LDI   START.0  ;Load low byte start address
0004  A3          PLO   R3       ;Put in R3.0 (R3=0006)
0005  D3          SEP   R3       ;Set P=3 and begin running
0006  —     START:—              ;     the program at 0006 with
                                 ;     R3 as the program counter
```

Following this routine, R0 is free for other uses.

Register X has the same four-bit length as P and is also used to specify one of the 16 scratch pad registers. The 1802 instruction that sets X to any hex digit from 0 to F is SEX, and after thinking about it for a while, the author decided to refrain from any comments on the designer's inclinations in the course of choosing this eye-catching mnemonic.

The X register is normally set to 2 with a SEX R2 instruction usually early in the program. R2 is set to address the bottom of a block on memory reserved as a stack. A stack in memory resembles a stack of dishes in a spring-loaded cafeteria well—the last dish put onto the stack must always be the first one to come off the top. R2 is used in most 1802 software to address the topmost free location in the stack, which grows and shrinks as needed by the program.

When X is used in this way, it is said to "designate the stack pointer," indicating which register will be used for stack control.

Except for the "immediate" type of arithmetic instructions—those that take as the second operand the byte immediately following the instruction op code—X is used to select a pointer to the second operand, which could be anywhere in memory. Supposing that we want to add nine to an unknown value stored at location ANSWER. All we know about the value of ANSWER is that it is stored at location 0620 and that it needs to have nine added to it. The following code will accomplish this.

```
0000  F8 06   BEGIN: LDI   $6     ;Load 06 into D
0002  B8             PHI   R8     ;Put in R8.1
0003  F8 20          LDI   $20    ;Load 20 into D
0005  A8             PLO   R8     ;Put in R8.0 (R8 = 0620)
0006  F8             SEX   R8     ;Set X=8 to reference register R8
```

```
0007  F8 09          LDI   $9     ;Load 09 into D
0009  F4             ADD          ;Add D + byte at 0620 via X
000A  58             STR   R8     ;Store result in D at 0620 via R8
000B  30 0B   STOP:  BR    STOP   ;Branch to here to halt program
```

Although we have neglected overflow, the byte at 0620 has been increased by nine then stored back at 0620. The other math instructions ADC, SM, SMB, SD, and SDB also operate with one byte in D and the other at a location pointed to by register RN. Setting X to N activates that register for use by these math operators.

Loops can frequently be shortened by using the LDXA and STXD instructions. These instructions also require X to designate the addressing register. Not only may bytes be loaded or stored with these two instructions, but the register-addressing memory indicated by X is automatically advanced or decremented. Compare the following routines for an example. Assume R5 to be set to the end on some memory location where we want the next eight bytes up to be set to zero.

Routine A

```
SETZER:  LDI   $8
         PLO   RF
SET1:    LDI   $0
         STR   R5
         DEC   R5
         DEC   RF
         GLO   RF
         BNZ   SET1
```

Routine B

```
SETZER:  LDI   $8
         PLO   RF
         SEX   R5
SET1:    LDI   $0
         STXD
         DEC   RF
         GLO   RF
         BNZ   SET1
```

No comments have been added so that you may more easily compare the routines side by side. The two are the same length and perform the same job. Both use RF.0 as a loop counter set to eight in the way explained earlier.

Routine A uses an STR R5 followed by a DEC R5 pair to sequentially store the value 00 in D at eight decreasing memory locations. Routine B first sets X equal to 5, then executes the STXD instruction to store D (00) at R5, designated by X, while decrementing R5 automatically.

Although each routine is the same apparent length, Routine B is faster because the inner loop is shorter. Removing the one instruction from inside the loop means *eight* fewer instruction executions because the loop will be performed eight times. On loops that execute many more times, removing an instruction may have drastic results. If a loop will be performed 255 times, for example, removing just a single instruction would be the same as shortening a nonloop program by a full memory page!

Another use for the X register is to permit different registers to be referenced by the same subroutines. On one execution X may be set

PROGRAMMER'S 1802 WORKSHEET PAGE ____ OF ____

PROGRAM TITLE _____ DATE ____/____/____

[ ] PAGE ADDRESS PROGRAMMED BY _____

ADDRESS				OPCODE				LABEL	MNEMONIC	REF	COMMENT

to 2, then on the next, X may be set to E, allowing the same routine to perform first via R2, then via RE. Like self-modifying code, however, this may be difficult to manage.

Two other instructions depending on the X register to select a scratch pad register are the IRX and the LDX. The LDX is a load instruction the same as LDN except that X selects the active register holding an address from which a byte is to be loaded. After LDX, the

register address is not changed. IRX increments just as INC but again needs X to tell the computer which register to increment. If X equals six, for example, IRX would do the same job as INC R6. Note that there is no X equivalent for STR. The only store instruction that operates through X is STXD.

□ □ □

Four unique 1802 instructions, SAV, MARK, RET, and DIS are explained fully in Chapter 3. For the most part, these four will be seen in interrupt handling, a procedure to be discussed in the next section.

To understand these, however, you need to be aware of yet one more 1802 register, the eight-bit T (for Temporary) register. The only time T is used is during SAV and MARK when it holds the packed values of the X and P registers. The T register may not otherwise be set or used by the programmer.

Input/Output

Because this book is concerned with programming the 1802, not much attention will be given to input and output, since this is necessarily dependent on hardware. Terminals, keyboards, music synthesizers, and devices of all sorts may be controlled by the 1802, and software may be written to do the controlling. But first the devices must be somehow wired to the computer. The following discussion, therefore, is intended as a general reference to techniques typically employed in writing such software, not for hooking up "machines" to the "machine."

The two 1802 input/output instructions, INP and OUT (what else?), are supplemented by four flag lines, EF1, EF2, EF3, and EF4, as well as by three control lines, N0, N1, and N2. The Q flip-flop may also play a part during input/output, since it can be hooked up to an external device as well.

In all computers, some sort of "bus" will be constructed to carry signals on their way in and out of the processor, sometimes bypassing the computer circuits and going directly to memory. With eight-bit microcomputers, this bus consists of eight separate lines or electrical pathways that are "bidirectional." That is, data may travel on the bus in either direction—this is something like being able to flip the backs of the bus's seats down when riding the other way. Data may, however, go only one way at a time.

The computer uses this bus internally as an eight-lane highway to and from memory devices for instructions that require a reference to memory. Transfers of data, etc., are all handled (without the programmer's help) via the bidirectional data bus.

The 1802 input/output instructions in combination with hardware may be used to energize the bus to a particular value (OUT) or capture the signals on the bus (INP) as a binary value, storing that value in memory. These instructions are discussed in Chapter 3 along with the coincident action of the three "N" lines. Latching circuits at the end of the data bus are usually given the descriptive titles, input and output "ports."

Flag lines are channels that may be used to signal (or flag) the computer that a condition exists outside of the computer. These lines, which may be set only by external devices, may also be used to input serial data. Serial data is information that comes in a bit at a time as compared with parallel data, which is transferred one byte (or many bits) at a time. The data bus carries parallel data, for example. For operating with serial information, the computer needs a program to interpret each bit as it comes in, packing bytes with those successive bits forming whole byte values from the serially fed data.

The short branch instructions that conditionally jump depending on the state of flag line may be used in conjunction with INP to accept data from the outside would. A typical case is an ASCII keyboard. ASCII stands for "American Standard Code for Information Interchange" and is a standard binary code for representing alphabetic characters, numbers, and punctuation. Also, "control characters," those that instruct the computer to do something, are represented in ASCII. A carriage return, hex OD for example, tells the computer to do just that on the printer or video screen. An entire communications system may be constructed with ASCII as a basic building block.

ASCII has become so popular that it is just about unnecessary to mention other codes such as Baudot. The assembler in Chapter 4 is written to understand ASCII and is therefore compatible with most keyboards, text editors, etc.

When a key is pressed on an ASCII keyboard, its circuits are wired to produce a unique character code in the hex range of 00 to 7F. This parallel output from the keyboard is sent directly to the processor's data bus for input to the processor and memory via an INP command.

However, the computer needs to know that a key has been depressed, and for this reason, along with the ASCII data a strobe or single pulse is sent each time *any* key is pressed. The separate strobe line is connected to one of the 1802 flag lines. A routine using conditional branching may then test for the action of a key being depressed. When a key press is sensed, the program would be directed to take the necessary steps to input the ASCII code.

The following routine would do just this. EF4 is the flag line that is hooked to the keyboard's output strobe.

```
KEYIN:  SEX     RB      ;X = B. Byte input via X
        BN4     KEYIN   ;If no keypress, wait in loop
        INP     DEVB    ;Input to D and via RB (because X = B)
        BN4     KEYIN   ;If no keypress, repeat (bounce)
HERE:   B4      HERE    ;If still keypress, wait till turned off
        RETN            ;Return
```

Alternatively, the following routine could be used but without the feature of checking for a solid (debounced) key press as above.

```
KEYIN:  SEX     RB      ;X = B
        BN4     KEYIN   ;Wait for keypress
        INP     DEVB    ;Input
HERE:   B4      HERE    ;Wait for key release
        RETN            ;Return
```

Bouncing is the action of two electrical contacts that come so close together that electricity is conducted in erratic pulses even though the switch may not actually be closed. Unless something is done about this bounce, or arcing, between the contacts, fast computer routines such as these may read the bounce to be tens or hundreds of valid key presses instead of only one.

Usually keyboards and other circuits should be debounced in hardware as a feature of their design. However, *sometimes* a software debounce will help, although it is difficult to construct a truly reliable debounce routine.

Note that a device number is specified with INP instruction. Most often a latch circuit is used between the device, the keyboard, or other input device and the computer. The DEVB designation is to activate device B, the latch circuit holding the information to be accepted as input.

For both INP and OUT, different devices may be activated through hardware decoding of the 1802's "N" control lines. These three lines mirror the last three bits of the op code either for INP or for OUT. Seven input or seven output devices may be easily controlled, and an unlimited number by the addition of extra decoding circuitry.

Output works in a fashion almost the reverse of input. (But see Chapter 3 for important differences between INP and OUT.) Bytes are transferred to the computer's bus by an INP instruction. Again, some sort of latch circuit is connected to the eight bus paths to "catch" bytes intended for output. The three "N" lines are used to select this latching circuitry that will then hold the value placed on the data bus by an OUT instruction.

Next, the computer will usually need to signal an external device that a piece of data awaits in the latch. The Q line may do this by being set, then immediately reset, to provide an "output strobe," which will send the data from the latch on toward its final destination.

The next routine demonstrates a way to output a byte of information to a device, in this case a printer, hooked to a latch, which in turn is hooked to the computer's parallel data bus. The 1802 Q line is hooked to the *device's* flag line much in the way the example keyboard earlier was hooked to the *computer's* flag line. (This should suggest to you a simple way to hook two computers together in a sort of binary hermaphroditic arrangement.)

```
PRNOUT: SEX     RB      ;X = B. RB addresses byte for output
        OUT     DEV3    ;Output to latch. RB is advanced
        SEQ             ;Set Q = 1 then Q = 0 to strobe
        REQ             ;     the output on to the printer
        RETN            ;Return
```

The printer would have to possess the ability to understand parallel data usually in ASCII code form. Video devices exist on the market that could accept data for printing on a CRT (a cathode-ray-tube, a television monitor). This routine could be used to print a character addressed by RB on the monitor screen. Note again that a device number is associated with OUT. In the example, the device selected is the output latch or port connected to the data bus.

A higher sophistication of control is possible by using data itself to select one device over another. In that case the data would be decoded by circuits to activate the proper device. Some computers also use address lines to output data. Simply referencing a particular address will cause a device to be activated.

One particularly important technique usually associated with input/output (but not necessarily with the instructions INP and OUT) is the handling of interrupt requests and DMA, meaning "direct memory access."

An interrupt is just that. The 1802 (and other computers too, usually) has an interrupt line that may be connected to an external device. When a signal is applied to this line, whatever the program is doing is interrupted and control passes to a special routine called, not surprisingly, the interrupt routine.

You may want to view an interrupt as a request to the computer to drop everything, remember where you are and what you are doing, run the interrupt routine, then, when that is done, go back and continue where you left off.

In computer jargon, this sequence is called "servicing an interrupt request." Inside the 1802 there is an internal flag, the interrupt enable

(IE) flag which may be set, like the Q flip-flop, to 1 or 0 but only indirectly through the instructions RET and DIS. When IE is set to equal 1, interrupt requests will be acknowledged by the processor. If IE is equal to 0, however, any interrupt requests are ignored. When an interrupt request has been acknowledged, the IE flag is automatically set to 0 so that any future interrupts won't interrupt the interrupt currently being serviced. It is the program's responsibility to reset IE to 1 at the end of the interrupt routine (usually) to enable future servicing of new interrupt requests.

The actions of an interrupt are strictly defined in all computers and in the 1802, the following events occur on receiving an interrupt request.

1. If IE = 1, then set IE← 0 and go to 2. Otherwise continue, ignoring the request.
2. Pack the values of the X, P registers into the T register.
3. Set P← 1, X← 2. Interrupt routine begins running with register R1 as the program counter.

Before any interrupts will be sent to the computer, the program must have set register R1 to the address of the start of the interrupt routine. When the interrupt line is used in 1802 systems, R1 will not usually be used for anything else. For example, in RCA's VIP computer and with many of the Elf computers, video output is controlled by an interrupt routine that formats and directs output requested via an interrupt by the 1861 video controller chip. When activated, the 1861 chip requests interrupts *60 times a second*, during which time memory bytes will be sent to the chip for further processing and output to the video screen. R1 *must* be dedicated as the interrupt program counter on these computers to make use of this video output circuitry.

Following is an example of what an interrupt routine may look like. You could use this format to construct any interrupt routine.

```
EXIT:   IRX             ;Point to saved data on stack
        LDXA            ;Pop stack
        SHR             ;Shift right to restore old DF
        LDXA            ;Pop stack to restore old D
        RET             ;Return. Restore X, P. Set IE = 1
INTRPT: DEC     R2      ;Decrement stack pointer—Entry point
        SAV             ;Push T register (holding X,P) onto stack
        DEC     R2      ;Decrement stack pointer
        STXD            ;Push value of D to save
        SHLC            ;Shift DF left into D
        STXD            ;Push to save old DF
        ( )             ;    Interrupt routine goes
        ( )             ;    here. Must not change
        ( )             ;    X without resetting to 2
        BR      EXIT    ;Branch to exit above (X must = 2)
```

Registers D, DF, X, and P are preserved by this technique so that the interrupt could occur during any part of the main program's execution without disturbing the program at that point. On return from the interrupt, all previous conditions will be restored, allowing the program to continue as if there had been no interruption (except, of course, for the effect on the interrupt routine itself). Some of the code may be left out. For example, if the interrupt routine will not change DF (i.e., does not execute any shifts or arithmetic instructions) then DF does not have to be saved on the stack. Also, if registers will be needed, they may have to be saved on the stack then restored along with D, X, and P.

Important to the operation is the placing of the exit portion of the routine on *top* of the entry. To exit the routine, a branch is executed to the very top of the code. This leaves the program counter R1 addressing the entry point after execution of the RET instruction. Thus, another interrupt may immediately occur without having to reset the address in R1.

DMA, direct memory access, is not an 1802 instruction but rather a description of a process that bypasses the processor's D register. An external device, such as a disk or again the 1861 video chip, signals the processor that it wants to open a channel directly to memory.

The action may occur in either direction, directly placing or retrieving bytes in the computer's memory. The processor serves only as a blind controller for the memory address set in register R0.

With DMA-IN, bytes are transferred through the eight-line data bus directly to the memory address in register R0. Following each transfer, R0 is automatically incremented so that successive bytes may be input. Such action is extremely efficient and quick when blocks of data need to be accepted.

DMA-OUT also requires R0 to contain an address in memory. For each DMA-OUT request applied externally to the 1802, a byte addressed by R0 will be sent to the data bus. Following DMA-OUT, R0 is automatically incremented. In this way, blocks of data may be sent to the outside world.

One good use for an interrupt routine is to control the DMA action by performing necessary settings of R0. It could also test R0 to terminate the action when the transfer is complete.

Depending on the application, DMA action may require highly accurate timing on the part of both the software and the hardware circuitry. The reader is directed to RCA's literature for suggestions on connecting devices requiring DMA.

The input/output port of a computer, where hardware meets software, is the programmer's inlet from the bay to the ocean. Without the ability for external devices to communicate freely with the computer, a program could operate only in a severely restricted and land-locked environment.

Computers can be purchased with most of the essential hardware already designed and connected. It is not necessary for a programmer to be an electronics engineer to operate external devices and write programs for input/output. Designers will appreciate the simplicity of input/output on the 1802, and programmers will appreciate and make heavy use of their work. Like the ports of the globe, the ports of a computer are where worlds meet.

Taming the Wild Animal

Learning a processor's instruction set is only one element in the study of machine language programming. The author remembers his first experience with a machine's code. The feeling was similar to memorizing large numbers of foreign words, arriving in the country where that language is spoken, and being frustratingly unable to utter a single comprehensible sentence. The analogy, by the way, is not a fictitious invention.

Expressing solutions to problems using ingredients that will lead to computer programs is a difficult hurdle for all beginners. The language BASIC, in fact, was designed as a means for students to develop the abstract concepts of computer programming. The concepts, not the forms, of programming are the most difficult to come by.

A good way to build confidence in applying and working with computers is the programming of games. Setting down the rules of play to a game forces one to consider all the possibilities and express them clearly. Teaching a friend the rules of a game is not unlike programming a computer to perform a task.

The limitations of a computer are as important as its capabilities. Both must be considered in the construction of programs that work. Expert programmers, when approaching a solution that will be attempted on a computer, keep their own thoughts within the realm of computer operations.

It is not adequate to state that a checker piece should probably capture an opponent if it can do so safely, then for the programmer to sit down at the keyboard and write a checkers strategy for a computer. For one thing, the word "probably" is not a good one to use when expressing what a computer is to do. True, a random number generator

may be able to inject the element of chance into a decision point, but the computer cannot be expected to make any rational judgments about the course it is to take.

But the computer *can* be programmed to make reasonable judgments based on conditions that exist or do not exist. It is the reasoning of the programmer, not the computer, though, that fuels the ability of a computer and a computer program to operate. Computers are extensions of the human mind, tools into which we may dismantle the thoughts in our heads for further processing. Without a program, a computer is no more than a big, dumb (but lovable) electronic animal waiting for someone to come along and tell it what to do.

For some people, a computer will prove to be an unpredictable untamed beast. Others will find an obedient companion willing to listen patiently and help in whatever way it is able. Taming the computer, the wild animal, is happily a skill that may be learned by anyone. Only those people who approach programming with arrogance will fail in their efforts and lose patience. The disciplined, determined person who has a little imagination will soon find that computers are not so difficult to use and understand. It is we who are complex and unfathomable, not the machines we work with.

A good way to start any program is to begin by turning off the computer. Pencil and paper are ingredients one and two for writing a good program.

Write down what it is you want the computer to accomplish for you. Ask yourself if the computer is physically capable of performing the task. (Having a machine to mow the lawn would be wonderful no doubt, but one can hardly expect the old memory chips to march outside on command and start munching blades of grass.) In other words, does the hardware fit the intended application? That's number one.

Is the project something you will be able to complete? Do *you* have the knowledge needed to instruct the computer? This is one of the most common sources of frustration among beginning programmers. Don't try to program the game of chess unless you are an expert in the game yourself. This is not to discourage you from choosing challenging projects. Just pay attention to your own limitations as well as the computer's.

Now that the idea is down on paper, prepare a preliminary list of all the parts and pieces that you feel will be needed for the program to operate. Go through the list several times breaking down those items that may still be too complex.

For example, take any card game. We may begin with the partial list:

Go Fish—a Card Game

1. Shuffle deck
2. Deal hands
3. Give instructions
4. Play hand

The list does not need to be in order, but it should contain all the needed elements, which our simple example certainly does not do. Looking again at the list, we see how the entries may be further broken down into subsections perhaps as follows:

1. Shuffle deck
   a. Generate random number
   b. Exchange random cards
2. Deal hands
   a. Get a card
   b. Transfer card to player hand (etc.)

You will eventually find on your desk a list of all the most basic elements that, when combined in specific ways, will play, in this example, a rousing game of Go Fish.

These basic elements may now be programmed. Provided the functions have been clearly defined, routines may be constructed to perform the tasks. For example, GETCRD will return the next card in the deck. That's *all* it will do. The routine may be tested thoroughly almost without regard for how it will eventually fit into the whole. When this is done, the programmer may forget about the machinations of the routine. In fact, the rest of the programming may proceed as if the instruction GETCRD had been added to the computer's instruction set.

Eventually, all these modules will begin to suggest the form the final program will take. Many times an omission will now be discovered, leading to a rewrite of the whole program from a different attack. Don't be discouraged if this happens and don't be afraid to use another piece of hardware as much as necessary—the wastebasket. The assembler in Chapter 4 was born of two previous unsatisfactory (yet satisfying) attempts.

This approach, called "bottom up programming," will often lead to the best design, producing very readable and understandable code. Furthermore, the basic building blocks of a program may be general enough to be used in other projects. Most programmers have closely guarded libraries of those building blocks from which to build new programs.

□ □ □

Inherent in the design of modular programs is the idea of subroutines. A subroutine is a preferably simple, one-function program segment that produces an effect needed by various other parts of the whole. Program modules are usually written as subroutines so that they may be utilized—"called"—from any part of the main program.

When a subroutine is called from another part of the program, control passes to that routine. After the subroutine is finished performing its duty, control "returns" to the point at which the subroutine was originally called. Graphically, subroutine action may be represented in the way illustrated in Fig. 2–2.

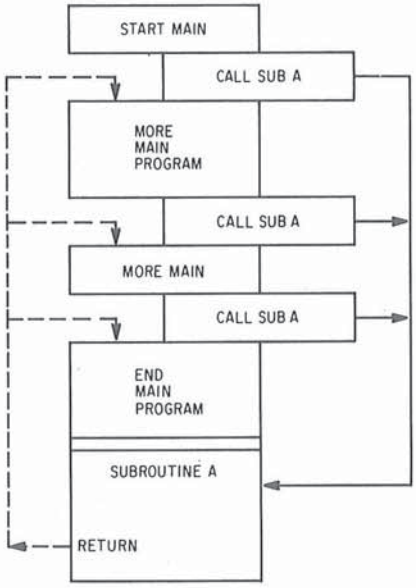


Fig. 2-2.

Subroutine A is called from three separate places in the main program. When subroutine A is finished, control returns to the point just following the call to the subroutine. The subroutine does not need to know from where it will be called, but when the call is made, that location needs to be preserved so that eventually the program will be able to find its way back.

There are three ways to make use of subroutines in 1802 software. Without a question the standard call and return technique (SCRT) is the best. This is fully described in RCA's 1802 user's manual MPM-210A. It is the most complex of subroutine controllers and needs three regis-

ters, usually R4, R5, and R6, dedicated to their jobs of calling and returning from subroutines.

The MARK SEP technique may also be used. Like SCRT, it is employed when subroutines themselves may call yet other subroutines that may in turn call still others. Such a condition is called nesting, and each subroutine call is said to add one deeper level of nesting to the structure of the program.

A less common subroutine technique impossible with many other processors is suggested by the 1802's ability to use any register as the program counter.

Using the same example in Fig. 2–2, suppose the main routine is set to run with register R3 as the program counter. The address of subroutine A is placed in register R4 and a SEP R4 is executed to call the subroutine, which will then begin running with register R4 as the program counter. This leaves register R3 pointing back to the main program so that with a SEP R3 instruction at the end of subroutine A, the main program will continue to run from where it left off, once again with R3 as the program counter.

Subroutines to be called in this manner are best constructed with the return at the top in the way the interrupt routine was written in the last section. For example, here is a typical arrangement:

```
RETA:   SEP     R3      ;Return to main
SUBA:   (       )       ;Begin here—
        (       )       ;    Subroutine code goes in
        (       )       ;    these locations
        BR      RETA    ;Branch to exit
```

When the SEP R3 instruction is executed upon branching to location RETA, SUBA's program counter R4 will be left pointing once again to the entry address of the subroutine so that future calls may be made with SEP R4 instructions without having to reset the address in R4 each time. This also has the advantage of assuring that R4 will always be equal to some known fixed address, except of course while the subroutine SUBA is actually running.

A disadvantage of the "SEP technique" is the inability to nest subroutines. In order to get back from a call, the subroutine must know which register was the program counter of the calling routine. In other words, if a second subroutine running in R6 were to call the above subroutine, SEP R3 would return control not to the address in R6 but wrongly to the address in R3.

One little-explored technique is to use two return commands, say SEP R4 followed by SEP R3. Now the subroutine may run in *either* R4 or R3, returning via the opposite register automatically. (Obviously, setting P to the current program counter register has no effect so that either the SEP R3 or SEP R4 will function as a NOP depending on which

register is the program counter at the time.) The author has used this technique to construct "co-routines," which differ from subroutines in that one is never able to say which is the caller and which is the "callee."

Stacks have been mentioned earlier as defined areas in memory used to store information that will be needed later. Keeping the return addresses on a stack following subroutine calls is a perfect use for this structure. The SCRT subroutine controllers use the stack in this way to "remember" how to re-thread the way back to the original main calling point.

Railroad cars on sidings, piles of dishes, bricks, and other more or less stackable items have been used to describe the structure of a stack in computers. To be original, the author adds here his own analogy, although it resembles the railroad car explanation, one of the best.

Picture the long, one-car-wide driveway of a typical suburban house whose occupants have invited several of their friends over for a party. One by one the guests arrive in their automobiles and one by one each arrival pulls his vehicle into the driveway immediately behind a predecessor if there was one. Obviously (driving over the lawn is forbidden), the last car to pull in will have to be the first one to pull out. Need I mention who will decide to leave the party early?

In a computer, a stack is controlled by a memory pointer and in the 1802 processor register R2 is almost always devoted to this function. The area in memory must be set aside for use as the stack, and the base address of the area, the endmost memory location, must be set in R2 sometime early in the program. The stack grows and shrinks as needed by decrementing and incrementing the stack pointer.

When X equals 2, the instruction STXD may be used to "push" a byte onto the stack. R2 is decremented by this instruction so that it points to a free location where another byte may be pushed. To "pop" or remove the last entry from the stack, an IRX is first performed followed by LDX or LDXA.

Note carefully the distinction in popping bytes off the stack. When successive bytes are to be popped, LDXA will automatically advance R2 (as long as X = 2) to the *next* element on the stack. The last pop, however, must be LDX, leaving R2 addressing the memory location where a byte may next be pushed. When bytes are popped off the stack, they are usually considered to be gone even though the values do not change in memory.

By following this convention, R2 is assured of always addressing the empty topmost byte of the stack. It may be used by all routines

provided each routine is responsible for removing bytes it had pushed onto the stack. Sometimes, STR R2 may be used to place a byte temporarily onto the stack. Usually one operand may be set up for an arithmetic operation in this way. However, unless the stack pointer is decremented, the top of the stack is subject to change on the execution of the next STXD instruction. Remember, to avoid a runaway stack, what goes on the stack *must* come off—and no more.

Two important but surprisingly ignored features of stack control are overflow and underflow. Although the RCA manual does not include tests for these conditions in discussing SCRT subroutines, this would be a highly desirable feature to include in an operating system. When the stack is full, overflow occurs on attempting to push another byte onto the stack. Underflow happens when the attempt is made to pop an empty stack. Both conditions are easily tested for by watching the address in register R2 (or other register if chosen), being sure that it remains within the defined memory area for the stack. (A large system with more than, say, 8K of memory may benefit from a full memory page dedicated to the stack. This would permit extensive future expansion of the system.)

Other stack-like structures are the queue ("cue") and the deque ("deck"). A queue allows entries from one end but exits from the other. Let's say another party is being held at a different house, this one with a semicircular drive with both ends opening onto the street. Restricting the cars to forward motion resembles the action of a queue or a "first-in, first-out" stack. Whoever arrived first will be the first to pull out of the drive. *Now* guess who wants to leave early!

An obvious improvement is the deque, where entries and exits may occur at both ends. Again the action resembles the circular drive but with no restrictions on direction of movement. Of course, in order to get the car out of the garage in the center . . .

In the Appendix, a full control package for handling a deque is included. Unlike the analogy of cars, bytes do not have to be shifted up and down to accept new data and to give up stored information. Instead, two pointers are kept to the ends of the stored data. These pointers will proceed to run away from and go toward each other as data enters and leaves the deque. Overflow and underflow notices are given when appropriate.

Earlier, a basic loop structure was presented. Such loops will find their way into most programs. Repeating the same sequences over and over is one of the things a computer does best, and it will probably take less memory space to use a loop than to write out the sequence in a straight line the number of times that function is to be performed.

However, loops *always* run more slowly than a program written redundantly in a straightforward manner. Practical limitations will restrict the programmer to using loops, but sometimes a loop may be unrolled to gain speed. For instance, an earlier example used a loop to store zeros in eight successive memory locations. We could just as well have written eight STXD instructions in a row (with X set to the addressing register). This executes faster than the loop and in fact takes up no more room if the mechanics to control the loop are included! Be especially careful not to use a loop when a straightforward sequence will be *shorter*. That would be a waste of both space and time.

Loop structures may also become more complex by programming loops within loops. As with subroutines that call other subroutines, this is called "nesting." Proper nesting is essential to these complex loop structures. For example, Fig. 2–3 shows the right and the wrong way to nest three loops together.

Corresponding to Fig. 2–3, the following program demonstrates how a loop structure three levels deep could be written. RD.0, RE.0, and RF.0 are used as the loop counters. A good rule to follow is to test all loop counters at exit points in the reverse order they are set at the entries to the loops. Also be careful not to branch back to the setup points of the loop. This is quite a common error to watch for. The result is usually an endless loop.

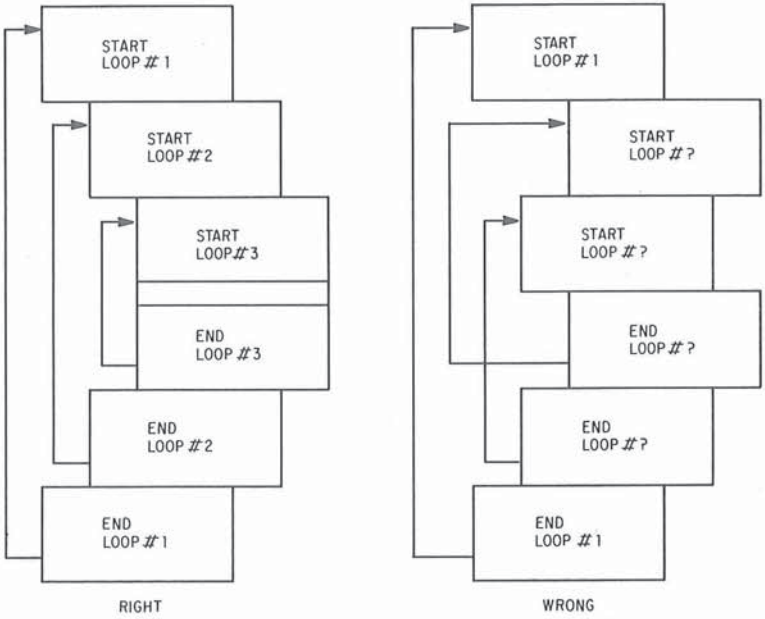


Fig. 2-3. Two ways to construct a loop. One works and the other one doesn't.

```
        LDI     $4                      ;Load 04 into D
        PLO     RD                      ;RD.0 = 4 = Loop #1
L1:     (       )                       ;      code in Loop #1
        (       )                       ;      may go here
                LDI     $10             ;Load 10 into D
                PLO     RE              ;RE.0 = 10 = Loop #2
L2:             (       )               ;      code in Loop #2
                (       )               ;      may go here
                        LDI     $FF     ;Load FF into D
                        PLO     RF      ;RF.0 = FF = Loop #3
L3:                     (       )       ;      all code in Loop #3
                        (       )       ;      may go here
                        DEC     RF      ;Decrement loop count #3
                        GLO     RF      ;Test loop count #3
                        BNZ     L3      ;If ≠ 0, branch to Loop L3
                (       )               ;      more code in Loop #2
                (       )               ;      may go here
                DEC     RE              ;Decrement loop count #2
                GLO     RE              ;Test loop count #2
                BNZ     L2              ;If ≠ 0, branch to Loop L3
        (       )                       ;      more code in Loop #1
        (       )                       ;      may go here
        DEC     RD                      ;Decrement loop count #1
        GLO     RD                      ;Test loop count #1
        BNZ     L1                      ;If ≠ 0, branch to Loop L1
                                        ;(continue on RD.0 = RE.0 =
                                        ;      RF.0 = 00)
```

Each of the nested levels has been indented to show the structure of the nesting. You may want to write out your loops in this way including the arrows to avoid accidentally crossing from one loop up into another.

Even more complex structures are possible, of course, although you may be limited by the number of available registers for loop counters. Usually, however, the nesting will not need to go very deep.

An advantage of this loop construction is that the loop counter will always equal zero at the end as long as the loop is allowed to terminate normally. If the loop is terminated early, the loop count is assured *not* to equal zero. This may provide a handy automatic "flag" to another program section, indicating that some condition caused the loop to terminate before running its course. That part of the program would need only to test the loop count in the register to know what happened.

Note also that only RD.0, RE.0, and RF.0 are altered by the loop "housekeeping" chores. Their high eight-bit counterparts RD.1, RE.1, and RF.1 *do not change* with this type of looping, and the programmer may trust these eight-bit halves to hold the results of operations, perhaps the very results generated inside the loops themselves. Other loop con-

tructions may not allow this, particularly if the loop count goes *past* zero before the loop is terminated.

To use the entire 16 bits of a register as a loop counter, each half of that register must subsequently be brought into D for testing. The following routine demonstrates this type of looping by erasing 4096 memory locations from 2000 to 2FFF.

```
                              ;Load the address 2FFF
        LDI     $2F           ;    into register RE
        PHI     RE            ;
        LDI     $FF           ;
        PLO     RE            ;
        LDI     $10           ;Load loop count of 1000 hex
        PHI     RF            ;    (=4096 decimal) into RF
        LDI     $0            ;
        PLO     RF            ;
        SEX     RE            ;Set X = E
LOOP:   LDI     $0            ;Load 00 byte into D
        STXD                  ;Store at RE & decrement
        DEC     RF            ;Count # loops
        GLO     RF            ;Get loop count low
        BNZ     LOOP          ;If ≠ 0, branch to continue
        GHI     RF            ;Get loop count high
        BNZ     LOOP          ;If ≠ 0, branch to continue
        RETN                  ;Return from subroutine
```

There are better ways to do this. Can you think of one? (See Answers to Exercises for a suggestion, page 140.) If the above routine is followed, these conditions exist: RE = 1FFF and RF = 0000. Note that testing RF.0 before RF.1 keeps the loop as short as possible.

Interpreters are programs that run other programs. They belong to the class of "higher-level languages," a description of the distance between the programmer and the machine. With machine language, the programmer instructs the computer on a one-to-one level, as if the captain of a ship were firing his own boiler. Interpreters move the programmer away (higher) from the machine's level of understanding.

An interpreter translates instructions into calls to the necessary routines that perform desired functions. Using the same maritime analogy, interpreters more closely resemble real life. The captain says "ship out," and the *first mate* tells the boiler room crew to start shoveling.

With very few exceptions, interpreters are written in machine language. All computer languages, however, must eventually filter down to the machine's level for execution. It may appear that a computer un-

derstands a GOTO instruction in the higher-level language BASIC, for instance, but it is really the interpreter *program* that understands and in turn tells the computer what to do.

The 1802 microprocessor has a structure that blends in well with the idea of interpretive languages. An advanced application may be approached by actually designing a higher-level language to perform various tasks and then using that language to program the computer.

Such an interpreter will be highly modular, with each function carefully designed and programmed. Instead of a main routine to control the calling of each subroutine module, the addresses of the subroutines themselves are kept in a list in the order the programmer wants them to be executed. Then all that needs to be done is to take each address, insert it in a register, and execute an SEP instruction to call the proper routine.

The program that accesses the address list and calls the subroutines is functioning as an interpreter. It oversees the entire operation of the program. The biggest advantage is the ability the programmer has to make changes without altering anything more than the order of the subroutine address list.

One popular 1802 language that uses this technique is named Chip-8 and was developed by Joe Weisbecker for the Cosmac VIP computer. Chip-8 contains 31 graphics and game-related instructions with which hundreds of computer games have been written. The entire Chip-8 interpreter fits in less than 512 bytes—two memory pages—and even a small 2K computer may be programmed to execute extremely sophisticated programs. By comparison, a small BASIC interpreter may require a minimum of 8K of memory.

When many applications will use the same structure, an interpreter may simplify the approach, allowing new programs to be written that are dissimilar only in output. In many cases, these programs will be shorter—and therefore cheaper in terms of memory—than comparable solutions. The compression results from the elimination of subroutine calls and the high degree of modularity imposed on the design.

Machine language programming on the 1802 is easy to grasp and fun to work with. Inexpensive computers that use the 1802 processor are opening the world of computing to a public that is just beginning to discard a Wizard of Oz view of programmers and their machines.

Soon millions of people will be programming and using home computers for business, personal education, and just for fun. Programming, even in the ominous-looking machine language, need not be an obstacle to anyone with the simple desire to learn. Understanding the workings of a computer is understanding the forces that will shape the future.

It's a future whose history has just begun.

3

1802 Instruction Set

The Instruction Set

To become fluent in any language, you must study and memorize the vocabulary. To program in a computer's machine language, you must have more than a simple familiarity with the computer's instruction set.

The following section details each of the 1802's instructions. The commands are presented in various forms, and to understand how to study the instructions, these forms are defined here.

1. *Op code*—Hexadecimal value, an eight-bit binary byte. The actual instruction as it would be stored in a computer's memory.
2. *Mnemonic*—A two-, three-, or four-letter code that describes the instruction's operation. Usually this is a contraction formed from the *definition.*
3. *Definition*—A short description of the instruction's operation. The pronunciation of the *mnemonic.*
4. *Symbolic action*—A mathematical-like representation of the operation.
5. *Discussion*—Details, hints, and further description of the operation.
6. *Programming example*—An example of how the instruction would appear, and its action in an actual program example.

Memorizing the instruction set and all of its forms is not enough for proficiency in the machine's language. If there is anything you do not understand about an instruction, the best way to discover its intricacies

1802 Instruction Codes

Mnemonic	Op code	No. cycles	Page No.	Mnemonic	Op code	No. cycles	Page No.
ADC	74	2	91	LDXA	72	2	90
ADCI	7C	2	95	LSDF	CF	3	104
ADD	F4	2	107	LSIE	CC	3	103
ADI	FC	2	110	*LSKP	C8	3	101
AND	F2	2	106	LSNF	C7	3	101
ANI	FA	2	109	LSNQ	C5	3	100
B1	34	2	81	LSNZ	C6	3	101
B2	35	2	83	LSQ	CD	3	103
B3	36	2	83	LSZ	CE	3	104
B4	37	2	83	MARK	79	2	93
*BDF	33	2	81	*NBR	38	2	83
*BGE	33	2	81	*NLBR	C8	3	101
*BL	3B	2	85	NOP	C4	2	100
*BM	3B	2	85	OR	F1	2	106
BN1	3C	2	85	ORI	F9	2	109
BN2	3D	2	85	OUT	6N	2	87
BN3	3E	2	85	PHI	BN	2	98
BN4	3F	2	85	PLO	AN	2	98
*BNF	3B	2	85	REQ	7A	2	94
BNQ	39	2	84	RET	70	2	89
BNZ	3A	2	84	*RSHL	76	2	96
*BPZ	33	2	81	*RSHR	7E	2	92
BQ	31	2	80	SAV	78	2	93
BR	30	2	80	SD	F5	2	108
BZ	32	2	81	SDB	75	2	91
DEC	2N	2	79	SDBI	7D	2	95
DIS	71	2	89	SDI	FD	2	111
GHI	9N	2	97	SEP	DN	2	104
GLO	8N	2	97	SEQ	7B	2	94
IDL	00	2	77	SEX	EN	2	105
INC	1N	2	79	SHL	FE	2	111
INP	6N	2	87	*SHLC	7E	2	96
IRX	60	2	87	SHR	F6	2	108
LBDF	C3	3	100	*SHRC	76	2	92
LBNF	CB	3	103	*SKP	38	2	83
LBNQ	C9	3	102	SM	F7	2	108
LBNZ	CA	3	102	SMB	77	2	92
LBQ	C1	3	99	SMBI	7F	2	96
LBR	C0	3	98	SMI	FF	2	111
LBZ	C2	3	99	STR	5N	2	86
LDA	4N	2	86	STXD	73	2	90
LDI	F8	2	109	XOR	F3	2	107
LDN	ON	2	77	XRI	FB	2	110
LDX	FO	2	106				

for yourself is to write a short program using the command. The simpler the better. For example, count up to two and stop, or logically "AND" two values together and stop. Comparing what actually happens to what you think should happen will often give you a clue to the instruction's action. Some *debugging* tools and *single-stepping* programs exist on the market that could make such experimentation easier.

To many beginners, the *symbolic action* of an instruction seems the most difficult to grasp. However, when discussing a machine language all computer books use pseudomathematical representations such as these. The equations do look confusing, but serious programmers realize the importance of presenting the action of an instruction in an unambiguous and concise form.

Unfortunately, there is little agreement on the syntax used in this symbolic representation and each machine code may be described using symbols that do not necessarily conform to those used for a different processor. Even RCA departs from the norm on the programming card supplied with their Cosmac VIP computer products. The following section sticks as closely as possible to recognized symbolic representation even though this may differ from RCA's supplied version. All mnemonics, op codes, and most definitions are those recommended by the manufacturer. The discussion is original and has not appeared in print elsewhere. An attempt has also been made to use real program examples such as those frequently encountered in software.

The Syntax

P—four-bit register designating which of the 16 general-purpose registers is to be used as the program counter.

Q—single-bit internal flip-flop. May be on (=1) or off (=0).

X—four-bit register designating which of the 16 general-purpose registers is to be used as the stack pointer.

T—eight-bit register used during interrupts to preserve the values of the X and P registers.

D—the accumulator; the data register.

IE—interrupt enable flag. If IE=1, then interrupts are enabled. If IE=0, then no interrupts will be acknowledged.

N—a hexadecimal digit. Ex: 5N would include the values 50–5F.

( )—indirect reference. Ex: R(X) = the register specified by X.

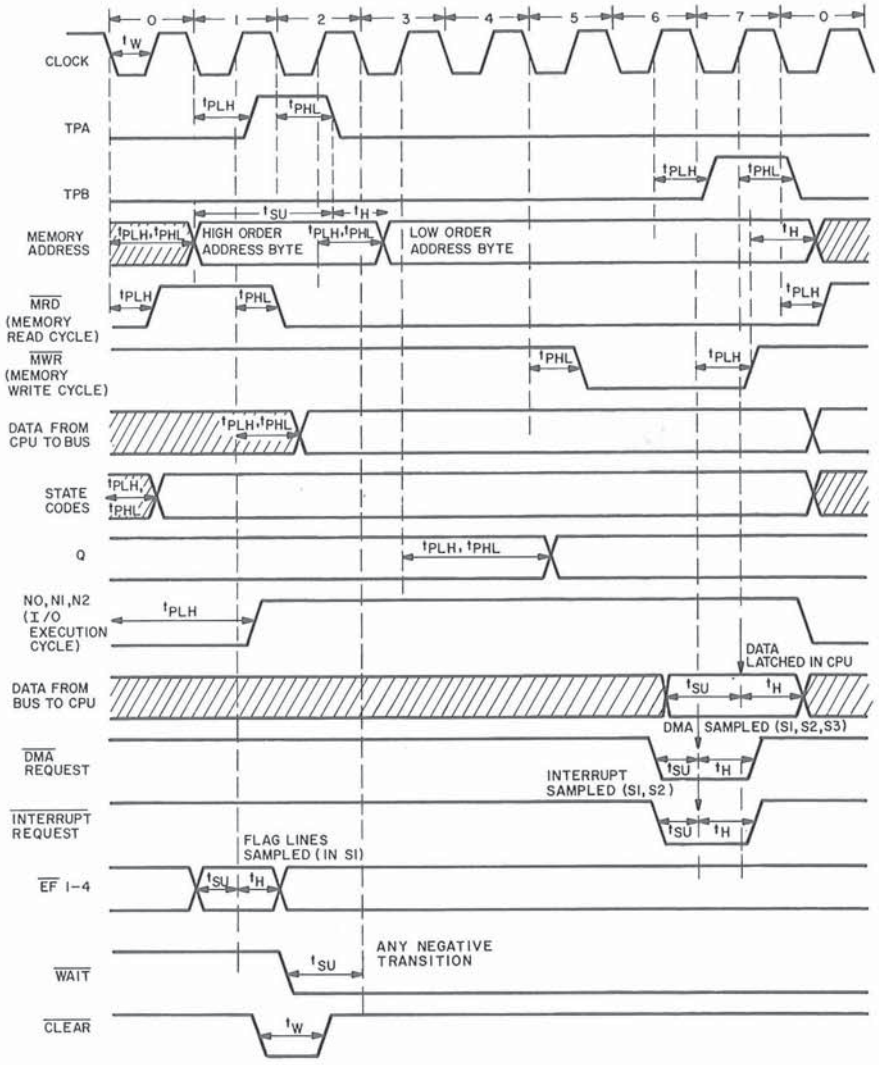
M—memory reference. Ex: M(R(X)) = byte addressed by the register designated by X. M(R(P)) = byte addressed by the program counter register.

+—the usual math operator.

−—the usual math operator.

=—the usual math operator.

←—arrow. The object to the left of the arrow accepts or becomes the object to the right. Note that RCA reverses the direction of the



NOTES:
1. THIS TIMING DIAGRAM IS USED TO SHOW SIGNAL RELATIONSHIPS ONLY AND DOES NOT REPRESENT ANY SPECIFIC MACHINE CYCLE
2. ALL MEASUREMENTS ARE REFERENCED TO 50% POINT OF THE WAVEFORMS
3. SHADED AREAS INDICATE "DON'T CARE" OR UNDEFINED STATE; MULTIPLE TRANSITIONS MAY OCCUR DURING THIS PERIOD

CDP1802 INSTRUCTION EXECUTION TIMES

1-Byte instructions require 2 Machine Cycles: 1 Fetch, 1 Execute.

2-Byte instructions also require 2 Machine Cycles: 1 Fetch, 1 Execute.

3-Byte instructions require 3 Machine Cycles: 1 Fetch, 2 Executes.

INSTRUCTION SIZE			% OF TOTAL REPERTOIRE	INSTRUCTION TIME @ f =		
Bytes of Op. Code	Bytes of Address or Data	Total Bytes of Memory		6.4 MHz	3.2 MHz	2 MHz
1	0	1	93	2.5 μs	5 μs	8 μs
1	1	2				
1	2	3	7	3.75 μs	7.5 μs	12 μs
Average Instruction Execution Time			–	2.6 μs	5.2 μs	8.3 μs

CDP1802 Timing Diagram (*Courtesy* RCA Corporation)

arrow. It is important to realize that the arrow symbolizes an action taking place. It should not be replaced by the equals sign as it is in the higher level language BASIC, for instance. "Q ← 0" and "Q = 0" are *not* the same thing. The first describes the action, "Q becomes zero," and the second makes a statement about the condition of Q. Note that the second would be a valid description of Q *following* the action of "Q ← 0."

.N—the period. When N=0, this denotes the least significant eight bits of a 16-bit register. When N=1, the expression denotes the most significant eight bits of a 16-bit register. Ex: R(2).1 = the high or leftmost eight bits of register 2.

MSB/LSB—most/least significant bit.

BUS—eight bidirectional data lines used for transferring data to and from the processor.

;—separates possibly unrelated actions occurring with the same instruction. Read as "also." In programming examples, signifies that a comment follows.

Argument—byte or bytes immediately following an instruction op code. Usually data or an address to be used by the instruction.

EFN—one of the 1802's four flag lines, EF1, EF2, EF3, and EF4, which may be set or reset by external sources only.

00	IDL

Idle or wait for interrupt or DMA request

Symbolic Action

R(P) ← R(P)−1; Bus ← M(R(0)) until an interrupt or DMA request

Discussion

When executed, this instruction halts the program, changing no registers or memory bytes. Upon sensing an external request for an interrupt or DMA action (direct memory action), the program continues with the value of the memory byte addressed by register R(O) placed on the data bus.

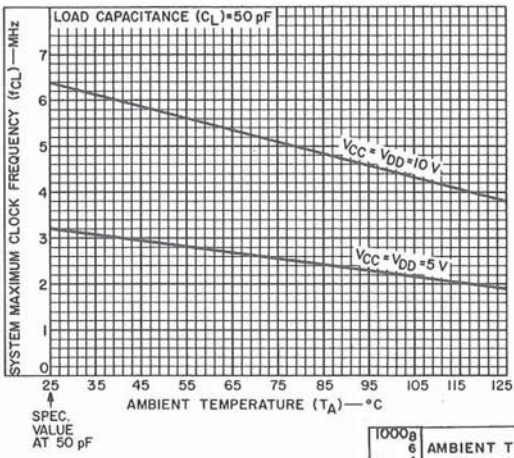
A typical use of IDL is to sync the output of a memory refresh block to those times when the information in the block is in a stable or final form. In a system that will not make use of interrupt or DMA action, the IDL may be used as a HALT instruction.

Programming Example

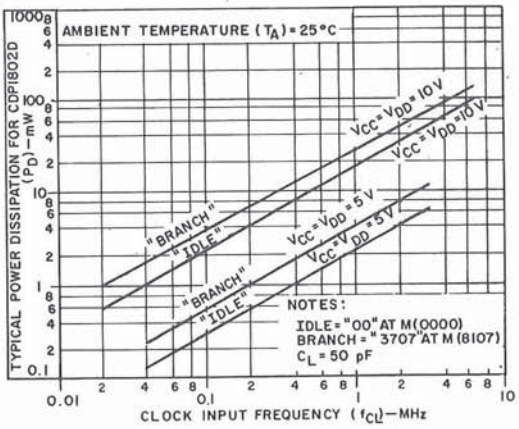
```
0200  00  IDL          ;Wait at this address for an interrupt or
                        DMA, then continue
```

0N	LDN

Load via N



Typical maximum clock frequency as a function of temperature.



Typical power dissipation as a function of clock frequency for BRANCH instruction and IDLE instruction for CDP1802.

CHARACTERISTIC	CONDITIONS V_{CC}[1] (V)	CONDITIONS V_{DD} (V)	VALUES AT 25°C CDP1802	VALUES AT 25°C CDP1802C	UNITS
Supply-Voltage Range	–	–	4 to 10.5	4 to 6.5	V
Input Voltage Range	–	–	V_{SS} to V_{CC}	V_{SS} to V_{CC}	V
Maximum Clock Input Rise or Fall Time, t_r or t_f	–	–	1	1	μs
Instruction Time[2]	5	5	5	5	μs
	5	10	4	–	
	10	10	2.5	–	
Maximum DMA Transfer Rate	5	5	400	400	KBytes/sec
	5	10	500	–	
	10	10	800	–	
Maximum Clock Input Frequency, f_{CL}[3]	5	5	DC – 3.2	DC – 3.2	MHz
	5	10	DC – 4	–	
	10	10	DC – 6.4	–	

NOTES:
1. $V_{CC} \leq V_{DD}$; for CDP1802C V_{CC} = V_{DD} = 5 volts.
2. Equals 2 machine cycles—one Fetch and one Execute operation for all instructions except Long Branch and Long Skip, which require 3 machine cycles—one Fetch and two Execute operations.
3. Load Capacitance (C_L) = 50 pF.

Operating Conditions (*Courtesy* RCA Corporation)

Symbolic Action

D ← M(R(N)) for N ≠ 0

Discussion

The memory byte at the address specified by register N is placed in the D register. The old value of D is destroyed by this action. The memory byte loaded and the register that addresses this byte are not changed in any way. Bytes addressed by registers 1 through F may be loaded into the D register by this action. Note that the op code 00 is an IDL instruction, *not* a "Load via R(0)" command.

Programming Example

```
0201  05  LDN  R5        ;Load D register with byte addressed by
                          register R(5)
```

1N	INC

Increment register N

Symbolic Action

R(N) ← R(N)+1

Discussion

Register N is increased by the value of 1. The entire 16-bit register is affected by the action of this instruction without the programmer's having to worry about overflow or carries between the lower and higher eight-bit parts of register N.

This is one of the few direct 16-bit operations of the 1802 processor providing a way either to count from 0000 to FFFF hexadecimal (0–65535 decimal) or to address lists anywhere in programmable memory.

INC is frequently used to count the occurrences of some operation, address sequential values in memory, or form an essential part of a timing loop that would delay the program until register N was equal to some specific value.

Registers act in a "wraparound" fashion when acted on by the INC instruction. Incrementing any register past the highest possible value of hexadecimal FFFF will result in 0000 in the specified register. Note that the one-bit overflow register DF is *never* affected by the INC instruction.

Programming Example

```
0202  1F  INC  RF         ;Add 1 to 16-bit value in register R(F)
```

2N	DEC

Decrement register N

Symbolic Action

R(N) ← R(N)−1

Discussion

Register N is decreased by the value of 1. The action is similar to INC except that counting past 0000 will now result in FFFF hexadecimal in the specified register. Again, all 16 bits of the register may be affected by the DEC instruction, and the overflow register DF is never affected by the operation.

Programming Example

```
0203  2A   DEC   RA          ;Subtract 1 from the 16-bit value in reg-
                              ister R(A)
```

30	BR

Short branch

Symbolic Action

R(P).0← M(R(P))

Discussion

Requires a one byte argument. The byte immediately following the instruction's op code replaces the lower eight bits of the program counter register. This will cause an unconditional jump to occur to the specified address.

Using the BR restricts branches to within the same memory page indicated by the high eight bits of the program counter, which will never change as a result of this instruction.

Allows the design of "page relocatable code," a program section that will operate regardless of which memory page it happens to reside in.

Programming Example

```
0204  30   6C   BR          ;Branch to memory location 026C
```

31	BQ

Short branch on Q=1

Symbolic Action

If Q=1 then R(P).0 ← M(R(P)) Else R(P) ← R(P)+1

Discussion

Requires a single-byte argument. Branch will be taken or not taken depending on the value of the Q flip-flop register. If Q=1, then the branch will be taken exactly as if the instruction were a BN. If Q=0, however, the program continues as if the BQ instruction were not there. Q is not changed.

Programming Example

```
0206  31   D0   BQ          ;If Q= 1 then branch to memory location
                              02D0
```

32	BZ

Short branch on D=0

Symbolic Action

If D=0 then R(P).0 ← M(R(P)) Else R(P) ← R(P)+1

Discussion

Requires a single-byte argument. Branch will be taken or not taken depending on the value of the D register. If the D register is equal to zero, then the branch will occur. If D is not zero, then the program will continue as if the BZ instruction were not there. D is not changed.

Programming Example

```
0208  32  01  BZ        ;If D=0 then branch to memory location
                          0201
```

33	BDF
33	BPZ
33	BGE

Short branch on DF=1
Short branch if positive or zero
Short branch if greater or equal

Symbolic Action

If DF=1 then R(P).0 ← M(R(P)) Else R(P) ← R(P)+1

Discussion

Requires a single-byte argument. If the value of the one-bit overflow/carry register DF is equal to 1, then the branch will be taken to the specified address. If DF is equal to 0, then the program continues as if the BDF instruction were not there. DF is not changed.

Usually this instruction will follow an arithmetic operation of some kind, and program flow will be altered or not depending on an overflow occurring from that operation. BPZ and BGE, while functionally identical, are potentially ambiguous and should be used with care.

Programming Example

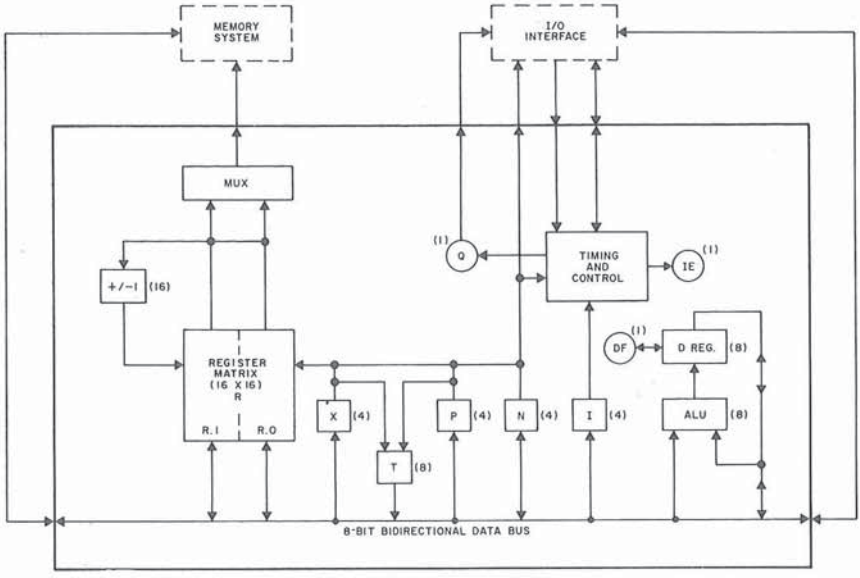
```
020A  33  00  BDF         ;If DF=1 then branch to memory loca-
                           tion 0200
```

34	B1

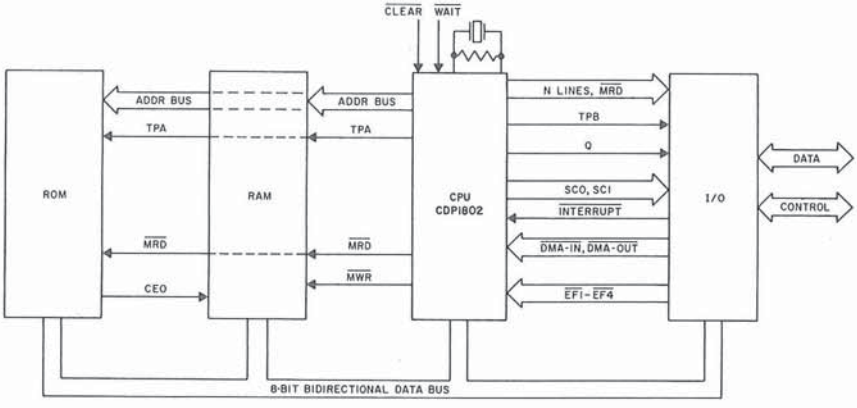
Short branch on EF1=1

Symbolic Action

If EF1=1 then R(P).0 ← M(R(P)) Else R(P) ← R(P)+1



CDP1802 COSMAC Architecture



Typical CDP1802-based Microcomputer System (*Courtesy* RCA Corporation)

Discussion

Requires a single-byte argument. If the flag line EF1= 1, then the branch will be taken to the specified address. Otherwise, the program continues on to the next instruction.

This and the following similar flag line conditional branch instructions allow program flow to be altered depending on the state of an

external signal. If a flag line is = 1, then a signal is being applied to that line by an external source. If no signal is being applied, then the flag line = 0. The flags do not retain their values after a signal has been applied and so do not have to be reset. However, the sensing of the signal must occur simultaneously with that signal—a flag line can tell if a condition exists but can describe nothing about conditions that may have existed and stopped before.

Programming Example

```
020C  34  FE  B1        ;If EF1=1 then branch to memory location 02FE
```

35	B2	Short branch on EF2=1
36	B3	Short branch on EF3=1
37	B4	Short branch on EF4=1

Symbolic Action

If EFN=1 then R(P).0 ← M(R(P)) for N=2,3,4 Else R(P) ← R(P)+1

Discussion

Identical to B1, short branch on EF1 except that flag lines 2, 3, or 4 are those tested.

Programming Examples

```
020E  35  68  B2        ;If EF2=1 then branch to memory location 0268
0210  36  01  B3        ;If EF3=1 then branch to memory location 0201
0212  37  0E  B4        ;If EF4=1 then branch to memory location 020E
```

38	NBR	No short branch
38	SKP	Skip next byte

Symbolic Action

R(P) ← R(P)+1

Discussion

When executed, causes the program to skip over the next byte. Note that there are two mnemonics associated with this instruction, but the action in both cases is identical. The different mnemonics refer to the intended use of the instruction.

NBR assumes that the following byte is an address the branch to which is never taken. This could be used to replace a different branch instruction, removing its effect from the program while having to make only a single-byte change.

SKP assumes nothing about the following byte and is frequently used to begin a loop by skipping into that section of code. In that case, the byte skipped would most likely be an instruction that is not to be performed on the first pass of the loop.

The distinction between SKP and NBR becomes more significant when using an assembler program where the output would be different depending on which mnemonic was used. The action of both instructions is always identical, however.

Programming Examples

```
0214  38  77  NBR      ;Do not branch to address 0277
0216  38      SKP      ;Skip the next byte, probably an instruc-
                        tion
```

39	BNQ

Short branch on Q=0

Symbolic Action

If Q=0 then R(P).0 ← M(R(P)) Else R(P) ← R(P)+1

Discussion

Requires a single-byte argument. Opposite in action to a BQ instruction. In this case, the branch is taken only if the Q flip-flop is set to 0 (off). If Q=1, then the next instruction in line will be performed. Q is not changed by this instruction.

Programming Example

```
0217  39  17  BNQ      ;If Q=0 then branch to memory location
                        0217. (On Q= 0, this example would halt
                        the program!)
```

3A	BNZ

Short branch on D≠0

Symbolic Action

If D≠0 then R(P).0 ← M(R(P)) Else R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The BNZ is opposite to the BZ instruction in effect.

If the value in the D register is not zero, regardless of what that value is, then the branch will be taken to the specified address. Other-

wise, the next instruction in line will be performed as if the BNZ instruction were not there.

The value of D is not changed.

Programming Example

```
0219  3A  17  BNZ      ;If D≠0 then branch to memory location
                        0217
```

3B	BNF	Short branch on DF=0 ("Not" DF)
3B	BM	Short branch if minus
3B	BL	Short branch if less

Symbolic Action

If DF=0 then R(P).0 ← M(R(P)) Else R(P) ← R(P)+1

Discussion

Requires a single-byte argument. Opposite to BDF instruction. If the value of DF is zero, then branch to the specified address. Otherwise, perform the next instruction in line as if the BNF instruction were not there.

The value of DF is not affected by this instruction.

BM and BL, while functionally identical, are potentially ambiguous and should be used with care.

Programming Example

```
021B  3B  FF  BNF      ;If DF=0 then branch to memory location
                        02FF
```

3C	BN1	Short branch on EF1=0
3D	BN2	Short branch on EF2=0
3E	BN3	Short branch on EF3=0
3F	BN4	Short branch on EF4=0

Symbolic Action

If EFN=0 then R(P).0 ← M(R(P)) for N=1,2,3,4 Else R(P) ← R(P)+1

Discussion

Require single-byte arguments. Opposites to B1, B2, B3, and B4 instructions.

If the specified flag line is =0 (i.e., *no* signal is being applied externally to that line), then take the branch to the specified address. Otherwise, perform the next instruction in line.

The flag lines are not changed in any way by these instructions.

Programming Examples

```
021D  3C  00  BN1     ;If EF1=0 then branch to memory location 0200
021F  3D  01  BN2     ;If EF2=0 then branch to memory location 0201
0221  3E  02  BN3     ;If EF3=0 then branch to memory location 0202
0223  3F  1D  BN4     ;If EF4=0 then branch to memory location 021D
```

4N	LDA

Load D via N; advance N

Symbolic Action

$D \leftarrow M(R(N))$; $R(N) \leftarrow R(N)+1$

Discussion

The byte at the memory location addressed by register R(N) is loaded into the D register. Following this, the 16-bit value in register R(N) is incremented by 1. Except for the increment of R(N), this instruction is identical to LDN. However, in this case a byte may be loaded into D via R(0) where no such capability exists for LDN.

An important use of the LDA instruction is moving blocks of memory or sequentially examining bytes in a list. With the single-byte instruction, bytes may be loaded into D via an address in any register and that register is sure to point to the next byte in the list following the LDA.

The following two instructions are *identical* in effect to the LDA programming example below.

```
0225  0C  LDN  RC  ;Load D via register R(C)
0226  1C  INC  RC  ;Increment register R(C)
```

Programming Example

```
0225  4C  LDA  RC     ;Load D via register R(C); advance register R(C) by 1
```

5N	STR

Store via N

Symbolic Action

$M(R(N)) \leftarrow D$

Discussion

The value of the D register is stored at the memory location addressed by the 16-bit value in register R(N). Whatever was at this memory location is now replaced by the current value in the D register. The

value in the D register is not changed nor is the address in register R(N) altered in any way.

You may view this operation as the opposite in effect of the load instruction LDN. However, all 16 registers may be used with STR.

An important use of STR is to set certain memory bytes to some required value or to store the value of the D register somewhere for use later on. STR is also used frequently in stack operations in which the programmer does not want to change the value of the addressing register.

Programming Example

```
0226  50  STR  R0        ;Store value of D at memory location
                          specified in register R(0)
```

60	IRX

Increment register X

Symbolic Action

R(X) ← R(X)+1

Discussion

Whichever 16-bit register is specified by X is incremented by the value of 1. If X=4 for example, register R(4) would be incremented by 1. In that case, the instruction INC R4 would have an identical effect.

IRX is most often used during stack operations. It is also handy in situations in which the register to be incremented may not be the same one during each execution of the IRX.

Programming Example

```
0227  60  IRX             ;Increment register R(X) by 1 (If X=2,
                          then R(2)=R(2)+1)
```

6N	OUT
6N	INP

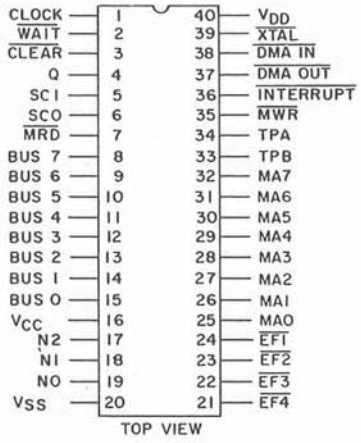
Output for N=1 to 7
Input for N=9 to F

Symbolic Action

For N=1 to 7: BUS ← M(R(X)); R(X) ← R(X)+1
For N=9 to F: M(R(X)) ← BUS; D ← BUS
Also: the 1802 "N" lines N0, N1, N2 are set equal to the last 3 bits of the N in the code 6N.

Discussion

These two instructions are presented together to emphasize the not so obvious differences between the two. Instructions 61 through 67 are used to output bytes *from* the computer, while 69 through 6F are in-



Terminal assignment diagram for the CDP1802 COSMAC Microprocessor. (*Courtesy* RCA Corporation)

structions that will input bytes *to* the computer. They are important links to the outside world, allowing transferral of data to and from communications devices, keyboards, or terminals, possibly to and from other computers. These devices would be hooked up to the computer's data bus, used in this case as a highway between device and computer.

OUT places the byte addressed by the register specified by X onto the data bus. That register is then automatically incremented by 1 before the next instruction is executed.

INP obtains whatever value happens to be on the data bus and stores that value at the memory location specified by the register designated by X. The input value is also placed into the D register replacing whatever value happened to be there before. R(X) is *not* changed by INP.

Please note that the two instructions are definitely not opposites of each other, although their actions seem to be simple changes in direction. Realizing the differences between OUT and INP may happily save you from some awful debugging sessions that have awed the author.

Also note that the op code 68 is missing from this book. No valid instruction exists for this op code.

Besides the transferral of data in and out of the computer, INP and OUT cause three special lines in the 1802 to be set equal to the value of the last three bits of the N in 6N. In this way, many devices may use the same data bus for sending and receiving bytes. The configuration of the three N lines may be used to select one device over another for the input or output and are sometimes used to simply toggle a device into action.

Programming Examples

```
0228  63    OUT          ;Output byte addressed by R(X). In-
                           crease R(X) by 1. N0=1, N1=1, N2=0
```

```
0229  6A   INP        ;Input byte to address specified by R(X).
                       Also place this byte in D. N0=0, N1=1,
                       N2=0.
```

70	RET

Return

Symbolic Action

(X,P) ← M(R(X)); R(X) ← R(X)+1; IE ← 1

Discussion

The eight-bit byte addressed by R(X) is broken into two parts; the MSB (four bits) being set into the X register and the LSB (four bits) set into the P register. Following this, R(X) is advanced by 1, and the interrupt enable flag IE is set to 1, allowing interrupts to be acknowledged.

Although RET may be used as a subroutine control instruction, it is most often seen as a return from an interrupt routine. Presumably, the interrupt's first instruction placed X and P at the byte addressed by R(X) by a MARK or SAV. The subsequent use of RET would then restore the values of X and P preserved in this way.

Note that an acknowledged interrupt automatically sets IE=0 (disabled).

Programming Example

```
022A  70   RET        ;Return from interrupt. Restore X,P,
                       advance R(X), and set IE=1 to allow
                       future interrupts to occur.
```

71	DIS

Disable

Symbolic Action

(X,P) ← M(R(X)); R(X) ← R(X)+1; IE ← 0

Discussion

Identical in every way to RET except for the condition of the interrupt enable flag IE. When DIS is executed, this flag is set to 0, and future interrupts will not be acknowledged until the flag is again set equal to 1.

Although it is possible to use DIS as a subroutine control instruction, it will more often be seen (as RET) in interrupt handling routines. Note that X and P are restored from the top of the stack addressed by R(X). This assumes that a previous use of the MARK or SAV instruction had been used.

Programming Example

```
022B  71  DIS        ;Return from interrupt, restore X,P,
                      then advance R(X) and disable future
                      interrupt action by setting IE=0.
```

72	LDXA

Load via X and advance

Symbolic Action

D ← M(R(X)); R(X) ← R(X)+1

Discussion

Used in stack handling and as a general-purpose load instruction, LDXA finds its way into just about any program.

X is assumed to be set to the hex number of any one of the 16 general-purpose registers. Execution of the LDXA will cause the byte addressed by that register designated by X to be loaded into the D register. Following this action, that 16-bit register value is incremented by 1. This provides a way to sequentially load values into the D register and to "pop" values from a stack where the exact location of the value needed is not known.

The old value in D is destroyed by the LDXA. X is not changed nor is the byte that was addressed by R(X) before that register was incremented. However, in standard stack procedures, once a byte is "popped" off the top of the stack, it should be considered undefined even though the LDXA did not actually change this value.

Programming Example

```
022C  72  LDXA       ;Load via X and advance—pop stack
                      into D
```

73	STXD

Store via X and decrement

Symbolic Action

M(R(X)) ← D; R(X) ← R(X)−1

Discussion

Opposite in effect to LDXA, the instruction STXD is normally used to place values in a memory area known as a stack. X is set to a hex number 0–F designating one of the sixteen 16-bit general-purpose registers. That register contains an address of a memory location. On execution of the STXD, the current value of the D register is stored at that memory location. Then the 16-bit address in the register is decremented by 1.

The value in D is not changed by the STXD. X is also not altered. Only the byte at the memory location and the register value addressing that location will change. During stack handling, an STXD instruction is said to "push" a value onto the stack.

Programming Example

```
022D  73  STXD        ;Push D; Save the value of D at the
                       memory location addressed by R(X),
                       then decrement R(X).
```

74	ADC

Add with carry

Symbolic Action

DF, D ← M(R(X))+D+DF

Discussion

The byte addressed by the register designated by X is added to the value of the D register. The value of the DF overflow/carry flag is also added at the same time. The result of the addition is stored in the D register, replacing its previous contents. If overflow occurs, DF will be set equal to 1; otherwise DF will be set equal to 0.

The byte added to D is unchanged in its memory location. X is not altered, nor is the address in the register designated by X changed in any way. Only D and DF are changed by the ADC.

Programming Example

```
022E  74  ADC         ;Add byte at M(R(X)) plus DF to D reg-
                       ister, storing the result in D and setting
                       DF to indicate overflow if the result is
                       larger than eight bits.
```

75	SDB

Subtract D with borrow

Symbolic Action

D, DF ← M(R(X))−D−$\overline{\text{DF}}$

Discussion

The value of the D register is subtracted *from* the byte addressed by the register designated by X. The complemented value of DF (the line above DF means "NOT DF") is also subtracted at the same time and is presumably the result of a borrow or no borrow from a previous subtraction. The result of the subtraction is placed in the D register, replacing its previous contents. DF indicates whether or not a borrow was needed.

The byte addressed by R(X) is not changed, nor is the address in the register R(X) changed in any way. Only D and DF are altered by the SDB.

Programming Example

```
022F  75  SDB      ;Subtract D from the byte at M(R(X)),
                    taking the value of DF as a possible bor-
                    row from a previous subtract. Store the
                    answer in D. Set DF to indicate if a bor-
                    row occurred.
```

76	SHRC
76	RSHR

Shift right with carry
Ring shift right

Symbolic Action

Shift D right one bit; MSB(D) $\leftarrow$ DF; DF $\leftarrow$ LSB(D)

Discussion

The value in D is shifted right so that each of the eight bits in D moves one bit position to the right. The previous value of DF moves to the leftmost bit position in D. The rightmost bit of D moves into DF. All of these operations occur simultaneously. No bits are changed by SHRC, only their positions.

Note that two mnemonics are associated with this instruction. Both operations are identical but the intended use may differ.

In double-precision shifting, where DF contains a bit value from a previous shift operation, an SHRC may take part in a simple integer divide by 2.

Programming Examples

```
0230  76  SHRC     ;Shift right, shifting old DF in at left and
                    shifting old LSB into DF. Finish divide
                    by 2.
0231  76  RSHR     ;Shift right circularly. Move LSB of D
                    into DF for testing while preserving old
                    DF value in MSB of D.
```

77	SMB

Subtract memory with borrow

Symbolic Action

D, DF $\leftarrow$ D$-$M(R(X))$-\overline{DF}$

Discussion

The byte addressed by the register designated by X is subtracted *from* the value in the D register. The complemented value of DF is also subtracted at the same time and is presumably the result of a borrow or no borrow from a previous subtraction. The result of the subtract is placed in the D register, and DF indicates whether or not a borrow was needed.

The byte addressed by R(X) is not changed. Register R(X) is also unaltered by SMB. Only D and DF will change from the use of this instruction.

Programming Example

```
0232  77  SMB        ;Subtract M(R(X)) from D, taking the
                      value of DF as a possible borrow from
                      a previous subtract. Store the answer
                      in D and set DF to indicate if a borrow
                      has occurred.
```

78	SAV

Save

Symbolic Action

M(R(X)) ← T

Discussion

The eight-bit value of the T register (temporary register) is stored at the memory location addressed by the register designated by X. The value of T contains the packed hex values of X and P in a single byte; thus, this action is used to save the values of X and P for later restoration or to test their values during debugging. T is automatically set to the values of X and P when an interrupt occurs.

No registers are altered by the use of SAV. R(X) is normally used as a stack pointer, although note that it is *not* decremented.

Programming Example

```
0233  78  SAV          ;Push T onto stack saving X and P
```

79	MARK

Push X,P; mark subroutine call

Symbolic Action

T ← X,P; M(R(2)) ← T; X ← P; R(2) ← R(2)−1

Discussion

Four separate things occur on execution of a MARK instruction. The current four-bit values of X and P are packed into a single eight-bit

value contained in the T register. This byte is then stored at the memory location addressed by register R(2). After this is done, the value of the P register indicating the program counter is copied into X. Finally R(2) is decremented by 1. Note that register R(2) is the normal stack pointer for most 1802 software.

The reason for copying P into X is to provide a way for passing bytes to a subroutine. Following the MARK instruction will usually be a SEP subroutine call (see SEP). Following this could be a string of bytes needed by the subroutine. These bytes may be obtained by executing LDXA instructions from within the subroutine as X points "back" to the calling point.

Programming Example

```
0234  79   MARK        ;Push XP. SET X← P. Prepare for sub call
  35  D4   SEP  R(4)   ;Call subroutine with R(4)=program counter
  36  01                ;    Three data bytes which the
  37  02                ;    subroutine obtains by executing three
  38  FF                ;    LDXA instructions
  39 (Main program continues on return from subroutine)
```

7A	REQ

Reset Q=0

Symbolic Action

Q ← 0

Discussion

The flip-flop Q line is set equal to 0. No other registers, values, or conditions are changed.

See SEQ for more information on using the versatile Q line.

Programming Example

```
023A  7A   REQ             ;Set flip-flop Q line=0 (turn off)
```

7B	SEQ

Set Q=1

Symbolic Action

Q ← 1

Discussion

The flip-flop Q line is set equal to 1. No other registers, values, or conditions are changed.

The Q line provides a simple way to interface the 1802 processor to the outside world. It may be connected to numerous devices and is

frequently used as a "strobe" or pulse to tell another device that some data is ready for intercepting on the data bus.

By setting and resetting Q at selected intervals, bits may be output in a serial fashion—that is, one after the other down a single-bit line. With some additional complications, Q may be used to store programs or data on tape in the way of the Cosmac VIP computer.

In that same computer, Q is also used to trigger an oscillating circuit that is connected to a speaker. By setting Q=1, a tone is turned on. Setting Q=0 turns the tone off. Additionally, Q may be hooked directly to a small amplifier/speaker circuit providing software control of the speaker's diaphragm movements.

Truly the Q line is a simple but versatile feature of the 1802 microprocessor.

Programming Example

```
023B  7B  SEQ          ;Set flip-flop Q line=1 (turn on)
```

7C	ADCI

Add with carry immediate

Symbolic Action

DF,D ← M(R(P))+D+DF; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The byte immediately following the ADCI instruction is added to the value in the D register. The overflow/carry flag DF is also added at the same time. The result of the addition is placed in the D register destroying its previous contents. DF indicates whether a carry or overflow out of D occurred following the addition.

Programming Example

```
023C  7C  06  ADCI     ;Add D register +DF+06, storing answer in D, and set DF to indicate if overflow occurred.
```

7D	SDBI

Subtract D with borrow immediate

Symbolic Action

DF,D ← M(R(P))−D−$\overline{DF}$; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The value of the D register is subtracted *from* the byte immediately following the op code for the SDBI

instruction. At the same time, the complement of DF is subtracted, then the final answer is stored in the D register. Presumably DF contained the result of a borrow or no borrow from a previous subtraction. Following the SDBI, DF is set to indicate whether or not a borrow has occurred.

Programming Example

```
023E  7D  00  SDBI      ;Subtract the value of the D register
                        from 00, taking DF into consideration.
                        Store the answer in the D register and
                        set DF to indicate if a borrow has oc-
                        curred. (This example would negate D.)
```

7E	SHLC	Shift left with carry
7E	RSHL	Ring shift left

Symbolic Action

Shift D left one bit; LSB(D) ← DF; DF ← MSB(D)

Discussion

The value in D is shifted left one bit so that each of the eight bits in D moves one bit position to the left. The previous value of DF moves to the rightmost bit position in D. The leftmost bit of D moves into DF. All of these operations occur simultaneously. No bits are changed by SHLC, only their positions.

Note that two mnemonics are associated with this instruction. Both operations are identical but the intended use may differ.

In double-precision shifting, where DF contains a bit value from a previous shift operation, an SHLC may take part in a simple multiply by 2.

Programming Examples

```
0240  7E  SHLC      ;Shift D left. Move DF into the LSB of
                    D while moving the MSB of D into DF.
                    Finish a double-precision multiply by 2.
0241  7E  RSHL      ;Ring shift left. Move DF into the LSB
                    of D to preserve the bit while moving
                    MSB of D into DF for testing that bit.
```

7F	SMBI	Subtract memory with borrow, immediate

Symbolic Action

DF, D ← D−M(R(P))−$\overline{DF}$; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The byte immediately following the SMBI op code is subtracted from the value in the D register. DF,

presumably the result of a borrow or no borrow from a previous subtract, is taken into consideration at the same time. The result of the subtraction is stored in the D register, and DF indicates whether or not a borrow was needed. This is identical in action to SDBI but with the operands reversed.

Programming Example

```
0242  7F  01  SMBI        ;Subtract 01 from D with the possible
                           borrow of a previous subtract in DF.
                           Store answer in D and set DF to indi-
                           cate if a borrow occurred.
```

8N	GLO

Get low register N

Symbolic Action

D ← R(N).0

Discussion

The lower least significant eight bits of the 16-bit register N are brought into the D register by the GLO instruction. At that time, the state of those bits may be tested, stored, compared, or used in whatever fashion fits the program design.

The original contents of the specified register are not changed by using the GLO.

Programming Example

```
0244  8A  GLO  RA          ;Get low eight bits of register R(A),
                           placing those bits in the D register for
                           examination.
```

9N	GHI

Get high register N

Symbolic Action

D ← R(N).1

Discussion

Identical to GLO except that the high, most significant eight bits of register N are brought into the D register. Again, the contents of the specified register are not changed by the use of GHI.

Programming Example

```
0245  92  GHI  R2          ;Get high eight bits of register R(2),
                           placing those bits in the D register for
                           examination.
```

AN	PLO

Put low register N

Symbolic Action

R(N).0 ← D

Discussion

Opposite to GLO, the PLO instruction places the contents of the D register into the low, least significant eight bits of the specified register. The value in D is not changed by the use of this instruction, nor is the other half of register N.

The primary use of PLO (also see PHI) is to set any of the sixteen 16-bit registers to a known value. Also, when a fast temporary storage cell for D is needed, that value may be saved in a register by executing a PLO for later retrieval through the use of a GLO instruction.

Programming Example

```
0246  A9  PLO  R9      ;Put the value of D into the low eight
                        bits of register R(9)
```

BN	PHI

Put high register N

Symbolic Action

R(N).1 ← D

Discussion

Opposite to GHI, the PHI instruction functions identically to PLO except that the contents of the D register are placed in the high, most significant eight bits of register N. The value of D is not changed nor is the other half of register N altered in any way.

Programming Example

```
0247  BF  PHI  RF      ;Put the value of D into the high eight
                        bits of register R(F).
```

CO	LBR

Long branch

Symbolic Action

R(P).1 ← M(R(P)); R(P).0 ← M(R(P)+1)

Discussion

Requires a two-byte argument specifying a branch address where the program is to continue. This address may be any four-digit hexadecimal (16-bit binary) address from 0000 to FFFF. The byte immediately

following the op code of LBR is the high two hex digits (eight bits) of the address. The next and last byte of this three-byte instruction specifies the low two hex digits (eight bits) of the address. Both of these argument bytes are loaded into whichever register has been designated the program counter by the value set into P at some previous time. Then the program begins execution at that new address.

LBR causes an unconditional jump to occur. Regardless of any conditions or values, the jump will always be taken.

Programming Example

```
0248  CO  04  48  LBR     ;Branch unconditionally to memory lo-
                           cation 0448, in this example 512 bytes
                           ahead.
```

C1	LBQ

Long branch on Q=1

Symbolic Action

If Q=1 then R(P).1 ← M(R(P)); R(P).0 ← M(R(P)+1)
Else R(P) ← R(P)+2

Discussion

Requires a two-byte argument as for LBR. If Q flip-flop is set to 1, then the branch will be taken to the specified address. Otherwise, the program counter will be incremented by 2 to continue the program as if the LBQ were not there.

Programming Example

```
024B  C1  00  50  LBQ     ;If Q=1 then branch to location 0050
```

C2	LBZ

Long branch on D=0

Symbolic Action

If D=0 then R(P).1 ← M(R(P)); R(P).0 ← M(R(P)+1
Else R(P) ← R(P)+2

Discussion

Requires a two-byte argument as for LBR. If the D register is equal to 0, then the branch will be taken to the specified address. Otherwise, the program counter will be incremented by 2 to continue the program as if the LBZ were not there.

Programming Example

```
024E  C2  F0  16  LBZ     ;If D=0 then branch to memory location
                           F016
```

C3	LBDF	Long branch on DF=1

Symbolic Action

If DF=1 then R(P).1 ← M(R(P)); R(P).0 ← M(R(P)+1)
Else R(P) ← R(P)+2

Discussion

Requires a two-byte argument as for LBR. If the overflow/carry flag DF is equal to 1, then the branch to the specified address will be taken. Otherwise, the program counter will be incremented by 2 to continue the program as if the LBDF were not there.

Programming Example

```
0251  C3  A1  00  LBDF    ;If DF=1 then branch to memory loca-
                           tion A100
```

C4	NOP	No operation

Symbolic Action

R(P) ← R(P)+1

Discussion

The program continues as though the no-operation instruction were not there. Absolutely nothing is changed by executing an NOP.

The usefulness of an NOP is surprisingly high. It may replace a questionable command during debugging or hold a place for a future instruction.

Note that the C4 NOP takes three machine cycles for execution and that other instructions may serve as NOPs when a two-cycle version is needed. Setting X to its current value would be the equivalent of an NOP, for example.

Programming Example

```
0254  C4  NOP             ;No operation. Continue program
```

C5	LSNQ	Long skip on Q=0

Symbolic Action

If Q=0 then R(P) ← R(P)+2 Else continue

Discussion

If and only if the Q flip-flop is set to 0, then the next two bytes will be skipped. Otherwise, the instruction(s) formed by those two bytes will be performed.

Programming Example

```
0255  C5  LSNQ          ;If Q=0 (not on) then skip next two
                         bytes
  56  12  INC  R2       ;Increment R2 × 2 on Q=1
  57  12  INC  R2
  58  (Continue here)
```

C6	LSNZ

Long skip on D≠0

Symbolic Action

If D≠0 then R(P) ← R(P)+2 Else continue

Discussion

If the D register holds any value other than 0, then the next two bytes will be skipped. Otherwise, if D=0, then the instruction(s) immediately following the op code for LSNZ will be performed.

Programming Example

```
0259  C6  LSNZ          ;If D≠0 then skip over the next two
                         bytes
```

C7	LSNF

Long skip on DF=0

Symbolic Action

If DF=0, then R(P) ← R(P)+2 Else continue

Discussion

If the overflow/carry flag DF=0, then skip the next two bytes. Otherwise, if DF=1, then the following instruction(s) after the op code for LSNF will be performed.

Programming Example

```
025A  C7  LSNF          ;If DF=0 then skip over the next two
                         bytes
```

C8	LSKP
C8	NLBR

Long skip
No long branch

Symbolic Action

R(P) ← R(P)+2

Discussion

The next two bytes are skipped regardless of any conditions that may or may not exist. Note that two mnemonics are used. LSKP assumes

that the following two bytes form an instruction or two instructions. NLBR would be assembled as if the following two bytes form an address. The distinction is important only during assembly, since execution is identical for both mnemonic representations.

Programming Examples

```
025B  C8  LSKP          ;Skip the following two bytes unconditionally
025C  C8  01  00  NLBR  ;Do not branch to 0100
```

C9	LBNQ

Long branch on Q=0

Symbolic Action

If Q=0 then R(P).1 ← M(R(P)); R(P).0 ← M(R(P)+1)
Else R(P) ← R(P)+2

Discussion

Requires a two-byte argument as for LBR. If the Q flip-flop is equal to 0 (off), then the branch to the specified address will be taken. Otherwise, the program counter will be incremented by 2 to continue the program as if the LBNQ were not there. Q is not changed by execution of the LBNQ.

Programming Example

```
025F  C9  C0  01  LBNQ  ;If Q=0 then branch to memory location C001
```

CA	LBNZ

Long branch on D≠0

Symbolic Action

If D≠0 then R(P).1 ← M(R(P)); R(P).0 ← M(R(P)+1)
Else R(P) ← R(P)+2

Discussion

Requires a two-byte argument as for LBR. If the value in the D register is not equal to 0, then the branch will be taken to the specified address. Otherwise, if D=0, then the program counter will be incremented by 2 to continue the program as if the LBNZ instruction were not there. D is not changed by execution of the LBNZ.

Programming Example

```
0262  CA  00  02  LBNZ  ;If D≠0 then branch to location 0002
```

CB	LBNF

Long branch on DF=0

Symbolic Action

If DF=0 then R(P).1 ← M(R(P)); R(P).0 ← M(R(P)+1)
Else R(P) ← R(P)+2

Discussion

Requires a two-byte argument as for LBR. If the overflow/carry flag DF=0, then the branch will be taken to the specified address. Otherwise, the program counter will be incremented by 2 to continue the program as if the LBNF instruction were not there.

Programming Example

```
0265  CB  7F  00  LBNF    ;If DF=0 then branch to memory location 7F00
```

CC	LSIE

Long skip on IE=1

Symbolic Action

If IE=1 then R(P) ← R(P)+2 Else continue

Discussion

If the interrupt enable flag is set equal to 1 (i.e., interrupts will be acknowledged), then skip the next two bytes. Otherwise, if IE=0, the instruction(s) formed by the two bytes following the op code for LSIE will be executed.

Note that this is the only 1802 instruction available for conditionally altering program flow on the value of the interrupt enable flag. In combination with branch instructions, other conditional responses to IE may be constructed.

Programming Example

```
0268   CC  LSIE          ;If IE=1 then skip next two bytes
  69   30  00  BN        ;If IE=0 then branch to 0200
  6B   (more code)       ;Continue here if IE=1 (set)
```

CD	LSQ

Long skip on Q=1

Symbolic Action

If Q=1 then R(P) ← R(P)+2 Else continue

Discussion

If the Q flip-flop is set equal to 1 (on), then the next two bytes will be skipped. Otherwise, if Q=0, then the instruction(s) formed by the two bytes immediately following the op code for LSQ will be executed. Q is not changed by the use of LSQ.

Programming Example

```
026C  CD  LSQ          ;If Q=1 then skip the next two bytes
```

CE	LSZ	Long skip on D=0

Symbolic Action

If D=0 then R(P) ← R(P)+2 Else continue

Discussion

If the value in the D register is equal to 0, then the next two bytes will be skipped. Otherwise, if D is any value other than 0, the instruction(s) formed by the two bytes following the op code for LSZ will be executed. D is not changed by the use of LSZ.

Programming Example

```
026D  CE  LSZ          ;If D=0 then skip the next two bytes
```

CF	LSDF	Long skip on DF=1

Symbolic Action

If DF=1 then R(P) ← R(P)+2 Else continue

Discussion

If the overflow/carry flag DF is set equal to 1, then the next two bytes will be skipped. Otherwise, if DF is equal to 0, the instruction(s) formed by the two bytes immediately following the op code for LSDF will be executed. DF is not changed by the use of LSDF.

Programming Example

```
026E  CF  LSDF         ;If DF=1 (i.e., on overflow) skip next
                        two bytes
```

DN	SEP	Set P

Symbolic Action

P ← N; R(N) becomes the program counter

Discussion

P is set to the hex digit specified by N. This may be any of the 16 digits 0 through F.

The instruction SEP is one that helps make the 1802 processor unique. Any single one of the sixteen 16-bit general-purpose registers may be designated as the program counter merely by setting P to the number of that register. Upon execution of the SEP instruction, the program will begin running at the address specified in register R(N), leaving the previous program counter addressing the byte immediately following the op code for the SEP instruction. Such means for switching program counters offers an extremely fast and simple method for calling and returning from subroutines.

Programming Example

```
026F   D4   SEP   R4        ;Set P=4. R(4) becomes the program
                             counter. Call sub whose address is in
                             R(4)
```

EN	SEX	Set X

Symbolic Action

X ← N; Register R(N) becomes reference for all "X" type instructions

Discussion

This wholesome instruction resembles SEP in operation except that X is set to the hex digit specified by N. That hex digit may be any one of 16, 0 through F.

Normally, a register (most often R(2)) is selected to be the stack pointer by executing the instruction E2 SEX R2, causing X to become equal to 2. Future commands such as STXD and LDXA would then reference the bytes addressed by R(2). In addition, most arithmetic instructions use the register designated by X to point to an operand located in memory. SEX is often followed by an arithmetic procedure.

Also, SEX will be used in place of the three-cycle NOP,C4. If X is already equal to 2, then the instruction E2 SEX R2 would cause nothing to happen. Since SEX is a two-cycle instruction, this technique is frequently used when a three-cycle command would interfere with critical timing requirements of an 1802-based system.

Programming Example

```
0270   EE   SEX   RE        ;Set X=E. R(E) is reference register for
                             future "X" type instructions.
```

F0	LDX	Load via X

Symbolic Action

D ← M(R(X))

Discussion

The byte addressed by the register designated by X is loaded into the D register. In its original location, the value of that byte remains the same. Also the register R(X) is not changed by the LDX instruction.

When used in stack handling, LDX is said to "pop" a byte off the top of the stack.

Programming Example

```
0271  F0  LDX          ;Pop stack. Place value in D register.
```

F1	OR	Logical OR

Symbolic Action

D ← M(R(X)) OR D

Discussion

The byte addressed by the register designated by X is combined by the rules of the logical OR with the value in the D register. The resulting byte is placed in the D register. ORing occurs on a bit-by-bit basis, adjacent bits in the bytes having no effect on their neighbors.

R(X) is not changed, nor is the byte addressed by R(X) altered in any way.

Programming Example

```
0272  E4  SEX  R4      ;X=4
  73  F1  OR          ;Logically OR D with the byte addressed
                       by R(4) and place the result in D
```

F2	AND	Logical AND

Symbolic Action

D ← M(R(X)) AND D

Discussion

The byte addressed by the register designated by X is combined by the rules of the logical AND with the value in the D register. The

resulting byte is placed in the D register. ANDing occurs on a bit-by-bit basis, with adjacent bits having no effect on their neighbors.

R(X) is not changed, nor is the byte addressed by R(X) altered in any way.

Programming Example

```
0274  E5  SEX  R5      ;X=5
  75  F2  AND         ;Logically AND D with the byte ad-
                       dressed by R(5) and place the result
                       in D
```

F3	XOR

Logical exclusive OR

Symbolic Action

D ← M(R(X)) XOR D

Discussion

The byte addressed by the register designated by X is combined by the rules of the logical exclusive OR with the value in the D register. The resulting byte is placed in the D register. Exclusive ORing occurs on a bit-by-bit basis with adjacent bits having no effect on their neighbors.

R(X) is not changed, nor is the byte addressed by R(X) altered in any way.

Programming Example

```
0276  EF  SEX  RF      ;X=F
  77  F3  XOR         ;Logically exclusive OR D with the byte
                       addressed by R(F) and place the result
                       in D
```

F4	ADD

Add

Symbolic Action

DF, D ← M(R(X))+D

Discussion

The byte addressed by the register designated by X is added to the value in the D register. The result of the addition is placed in the D register and the overflow/carry flag indicates whether or not overflow occurred.

R(X) does not change, nor does the byte addressed by R(X). The previous value of DF has no bearing on the addition.

Programming Example

```
0278  F4  ADD          ;Add D to M(R(X)) and place the answer
                        in D. DF indicates if overflow occurred.
```

F5	SD

Subtract D

Symbolic Action

DF, D ← M(R(X))−D

Discussion

The value in the D register is subtracted from the byte addressed by the register designated by X. The result of the subtraction is placed in the D register and the overflow/carry flag DF indicates whether or not a borrow was needed.

R(X) does not change nor does the byte addressed by R(X). The previous value of DF has no bearing on the subtraction.

Programming Example

```
0279  F5  SD           ;Subtract D from M(R(X)) and place the
                        answer in D. DF Indicates borrow.
```

F6	SHR

Shift right

Symbolic Action

Shift D right one bit; MSB(D) ← 0; DF ← LSB(D)

Discussion

The value in the D register is shifted right so that each of the eight bits in D moves one bit position to the right. The least significant bit of D moves into DF. A 0 bit is in turn shifted into the most significant bit of D. All of these actions occur simultaneously. The old value of DF is lost and is of no consequence to the result of SHR.

Programming Example

```
027A  F6  SHR          ;Shift D to the right one bit position.
                        MSB(D)=0 and DF= old LSB of D.
```

F7	SM

Subtract memory

Symbolic Action

DF, D ← D−M(R(X))

Discussion

The byte addressed by the register designated by X is subtracted from the value in the D register. The result of the subtraction is placed in D and the overflow/carry flag DF indicates whether or not a borrow was needed.

R(X) does not change nor does the byte addressed by R(X). The previous value of DF has no bearing on the result of the subtraction.

Programming Example

```
027B   F7   SM          ;Subtract M(R(X)) from D. Place the re-
                         sult in D and set DF to indicate if a
                         borrow occurred.
```

F8	LDI

Load immediate

Symbolic Action

D ← M(R(P)); R(P) ← R(P)+1

Discussion

Requires a single-byte argument. Execution of the LDI instruction causes the value of that argument to be loaded into the D register.

Programming Example

```
027C   F8   00   LDI        ;Load D register with the value 00
```

F9	ORI

Logical OR immediate

Symbolic Action

D ← M(R(P)) OR D; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The byte immediately following the op code for ORI is combined by the rules of the logical OR with the value in the D register. The resulting byte is placed in the D register.

Programming Example

```
027E   F9   30   ORI        ;Logically OR D with hexadecimal 30
                             and store the result in the D register
```

FA	ANI

Logical AND immediate

Symbolic Action

D ← M(R(P)) AND D; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The byte immediately following the op code for ANI is combined by the rules of the logical AND. The result is placed in the D register.

Programming Example

```
0280  FA  F0  ANI      ;Logically AND the D register with hex-
                        adecimal FO. Place the result in the D
                        register.
```

FB	XRI

Logical exclusive OR immediate

Symbolic Action

D ← M(R(P)) XOR D; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The byte immediately following the op code for XRI is combined by the rules of the logical exclusive OR. The result is placed in the D register.

Programming Example

```
0281  FB  FF  XRI      ;Logically exclusive OR the D register
                        with the hexadecimal value FF. Place
                        the result in the D register. (This ex-
                        ample would complement the value
                        in D.)
```

FC	ADI

Add immediate

Symbolic Action

DF, D ← M(R(P))+D; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The byte immediately following the op code for ADI is added to the value in the D register. The result of the addition is placed in D. DF is set to indicate whether or not overflow occurred.

The previous value of DF has no bearing on the result of the addition.

Programming Example

```
0283  FC  01  ADI      ;Add 01 to the D register. Set DF to
                        indicate if overflow occurred.
```

FD	SDI

Subtract D immediate

Symbolic Action

DF, D ← M(R(P))−D; R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The value in the D register is subtracted from the byte immediately following the op code for SDI. The result of the subtraction is placed in the D register. DF indicates whether or not a borrow was needed.

The previous value of DF has no bearing on the result of the subtraction.

Programming Example

```
0285  FD  21  SDI      ;Subtract the D register from hexadecimal 21. Place the result in D and set DF to indicate if a borrow occurred.
```

FE	SHL

Shift left

Symbolic Action

Shift D left one bit; LSB(D) ← 0; DF ← MSB(D)

Discussion

The value in the D register is shifted left so that each of the eight bits in D moves one bit position to the left. The most significant bit of D moves into DF. A 0 bit is in turn shifted into the least significant bit of D. All of these actions occur simultaneously. The old value of DF is lost and is of no consequence to the result of SHL.

Programming Example

```
0287  FE  SHL          ;Shift D to the left one bit position. LSB(D)=0 and DF= old MSB of D
```

FF	SMI

Subtract memory immediate

Symbolic Action

DF, D ← D−M(R(P)); R(P) ← R(P)+1

Discussion

Requires a single-byte argument. The byte immediately following the op code for SMI is subtracted from the value in the D register. The

result of the subtraction is placed in the D register. DF indicates whether or not a borrow was needed.

The previous value of DF has no bearing on the result of the subtraction.

Programming Example

```
0288  FF  02  SMI        ;Subtract 02 from the D register. Place
                          the result in D and set DF to indicate
                          if a borrow occurred.
```

4

□ □ □

The Assembler

If you have done much programming in absolute hex, loading instruction codes by hand into a computer, you may have wondered if a better way didn't exist. The following program will free you from the tedious job of translating mnemonics from a table into machine language hex codes, a process that not surprisingly is a common source of many frustrating bugs.

Instruction validity is checked as you type, eliminating errors that otherwise could go undetected until run time. The assembler is capable of assembling code anywhere in programmable memory. Once you have it running, the program may be used to expand its own capabilities. In fact, some of the following code was written in this manner on the author's system (a 4K Cosmac VIP) using the sections that had been completed to finish the rest and to experiment with controllers responsible for running the assembler.

Any 1802 system may use this assembler. All routines are modular, capable of functioning alone without complications. No assumptions have been made about your operating system except that it must be able to accept ASCII keyboard input and have the ability to print ASCII encoded information in some fashion.

Register use has been purposely restricted to avoid competing with your system's requirements. R0 and R1 have not been used at all, these registers being reserved for interrupt usage common in many 1802 computers. R2 must be set by you to an area designated as a stack with a minimum of 11 bytes of headroom available before calling ASMBLR with a SEP R4 D40600 instruction.

Register R3 is the program counter controlled by the standard call and return technique (SCRT) exactly as described in RCA's 1802 user's manual. The assembler's use of SCRT assumes X to be set to 2 upon execution of a CALL, and that at all times R2 addresses a free byte at the top of the stack. The CALL routine runs in R4, the RETN in R5 with R6 functioning as a pointer to the return address and/or arguments to be passed. You must provide these call and return subs and dedicate

R4, R5, and R6 to their use as subroutine controllers. Provided you follow RCA's suggested techniques, this should give you no problem.

Registers R7, R8, and R9 are not required by the assembler nor are their values changed by any of its routines.

Registers RA, RB, RC, RD, RE, and RF are used by the assembler. If your system also uses these registers, you will need to push onto the stack any values you want to preserve, then restore those values on return from the assembler. Be sure your stack has enough room for these bytes.

Register RA needs some special care. It will be used by the assembler as a pointer to the final assembly. In other words, RA tells the assembler at what address to assemble instructions into memory.

It will be more convenient if your system does not change RA between calls to ASMBLR. Then you may program sequentially, having to specify a starting address only once. If your system needs RA, then you will have to preserve and restore values (swap) on both the CALL and RETN to and from assembly.

The suggested handler flow chart should help you overcome any difficulties presented by these restrictions. This flow chart assumes the worst case—that your system needs all of the registers mentioned above. Even though this will result in the most complex handler for the assembler, you will still be able to run the program.

Register RB.1 is used to hold the display page address in some VIP systems. This is not allowed concurrently with the use of the assembler if video refresh is being controlled via an interrupt routine. Two possible solutions: (1) turn off the video, set RB.1 to the input buffer for the assembler, then restore RB.1 before turning on the video after assembly; or (2) write a new interrupt routine that loads the display page as an immediate value instead of from a register. The second solution is preferred.

Instructions for Using

Input to the assembler follows an unstructured format. You must adhere to a few rules, but the program is very forgiving. A 32-byte input buffer has been reserved at 0400–041F, although you may elect to have input (and subsequent output from the disassembler) appear in any memory location indicated by RB.1. This buffer must begin on a page boundary (the buffer low address starting at 00). You are responsible for setting RB.1 to 04 if you plan to use the reserved buffer at 0400.

Into this buffer go the ASCII codes representing the instruction you want to assemble into memory. The very first thing you must specify in the buffer is an address where you want the output to go. Failure to specify a starting address may crash the assembler if it is loaded into RAM.

To specify any hex value for the address or an instruction argument, type the dollar sign ("$") followed by the hex value. No spaces may follow the $ and the hex digits. Values may be up to four digits long. Leading zeros do not need to be written. If you specify too many digits (>4) an error will result. If you specify too many digits for an instruction requiring a single-byte argument, only the last two digits will be used. Zero is the default value if only the $ is typed.

Mnemonics are entered as listed in RCA literature (and in this book, of course). Entering a nonexistent mnemonic will produce an error message. Once you have entered a starting address (followed by an instruction), subsequent instructions may be entered without specifying a new address.

If the instruction requires an argument, this must be entered in hexadecimal (preceded by the $ sign) following the mnemonic. The assembler accepts only hexadecimal arguments.

If the instruction requires a register reference, this must be entered following the mnemonic. Practically any form will work: R4, REG4, 4, $4 will all assemble as a reference to register R4. The only restriction is that whatever follows the mnemonic must contain no spaces and must end with the specified hex digit. This may be of use with 6N input/output instructions allowing a device number (for example DEV-N) to be specified. It will disassemble as RN, however.

The assembler ignores anything in the line following a valid instruction. Comments could thus be kept without needing to precede them with a semicolon, although this assembler will more often be used in a direct or immediate assembly mode rather than in assembling a long source listing.

Note that a carriage return is not required to terminate the input in the buffer. If a carriage return does not terminate the input, however, the last string (the mnemonic, the register reference, or the argument) must be followed by at least one space. In other words, only spaces (ASCII 20) or a CR (ASCII OD) will terminate input strings.

Provided the input is acceptable to the assembler, the instruction op code and any required arguments will be placed into memory via RA. If an error is detected during the assembly process, the assembly will be aborted, leaving the address in RA unchanged. (Therefore, if an error is received, you may reenter the instruction without having to respecify the address.) The byte addressed by RA may or may not be changed, depending on when the error occurred.

When the assembly of any instruction is complete, RA is left pointing to the assembled instruction *not* at a new position past the instruction as may be expected. At this point in the process, the assembler calls a disassembler routine (DISASM), which will advance RA.

DISASM is called to provide a confirmation echo of the assembled instruction. The same input buffer (now serving as an output buffer) is

first set to all ASCII spaces (20 hex). The instruction is then disassembled in ASCII into the buffer ending with a carriage return at the 17th character position. On return from the assembler, the ASCII code in the buffer should be printed by your operating system to echo the instruction you just entered. This provides a sure confirmation that the code was assembled as expected. On a video display it will look good to have the output overwrite the previously typed input, then scroll up a line or two.

If you wish, you may call DISASM separately subject to the same register restrictions and requirements for ASMBLR. The instruction addressed by RA will be disassembled into the buffer, and RA will address the next instruction in memory on return.

Debugging Hints

All possible errors are not recognized by ASMBLR, although most common ones will be caught. It is possible to assemble the illegal instruction LDN RO, for example, although this will disassemble as the IDL command. Also, you may assemble the illegal INP D8 or OUT D8 or OUT D0, but again these will disassemble correctly as the illegal 68 "---" instruction and the legal 60 "IRX" code. Since the buffer provides immediate confirmation at assembly time, these unusual errors certainly won't go unnoticed by the programmer.

DISASM always uses the first instruction mnemonic it finds in the table. Therefore, if you input a BGE $00, it may disassemble as BDF $00, which is the same thing. Other duplicate mnemonics will respond similarly.

If you receive the "---" output for an op code that you think should be correct, you have a problem in the mnemonic table and need to check that the table was loaded correctly. This debug "bounce" was included in DISASM as an aid to implementation. Likewise, if a known mnemonic won't assemble (but others will), the problem is most likely in the table.

When using DISASM alone, be careful not to address the middle of an instruction with RA (i.e., the second or third byte of a two- or three-byte instruction). Disassembly could then appear quite foreign to what it should. Be sure RA addresses a valid 1802 instruction before calling DISASM. This cannot happen when DISASM is called from ASMBLR.

You may obtain a hex dump with a call to address $078A. Set RB (RB.1 and RB.0 here) to the top of your buffer cleared to spaces, and the number of bytes +1 to output in RC.0. The bytes addressed by RA will be dumped in ASCII into the buffer followed by a CR at position 17. Be especially careful not to overwrite the end of the buffer by dumping

too many bytes at once! No checks are made against this happening. If you want to output the address in RA with the hex dump you may call $0763 with the same parameters with RC.0 set to exactly the number of bytes to output. This may produce a strange output on RC.0 = 02, however.

The output of DISASM is configured to a 16-character line, meaning that arguments for three-byte instructions such as LBR just don't have room to be printed following the mnemonic. If you are fortunate to have a longer line, the code has been written to allow easy expansion. Each section of output is TABed to a start address in the buffer even when this meant programming additional code into DISASM. You may set the tabs to any values you want, thus formatting the output to your tastes. If you use a buffer greater than 32 bytes, you must also modify the code at the beginning of DISASM to clear the additional buffer space to all ASCII 20s.

Physical layout of the subroutines is purposely wasteful. Although all routines reside in a 1K (4-page) memory area, 363 bytes are available for expansion within that area. The main loops have been separated to their own pages (ASMBLR @ 0600, DISASM @ 0700) so that you may easily construct patches if you decide to add capabilities. (One idea that comes to mind would be to print out specific error messages depending on the error that occurred. Perhaps this could be a first project using the assembler to construct the routines!)

The mnemonic table may be expanded to include entries of your own design. Each entry is six bytes long in a format explained in the listing. The optional CALL and RETN may be added at the suggested addresses (replacing NBR and NLBR, respectively) or simply tacked on to the end of the table. The very last byte in the table *must* be an FF hex byte or the assembler is almost sure to fail. The table may be any length.

All of the routines may be placed in ROM, although you will have to change call addresses to relocate the routines. No long branches, long skips, or three-cycle instructions have been used. The input buffer must, of course, reside in RAM.

A "trick" the author has discovered on using ASMBLR is to call the program with *only* an address (preceded by $ as described) in the input buffer. The effect is an error; however, RA will still be set to the address before the error occurs. You may want to use this technique to set RA before calling DISASM, perhaps looping several times to output successive instructions.

Operation Summary

Input examples (ASMBLR)			Output examples (DISASM)			
$A00	LDN	R6	0A00	06	LDN	R6
BNZ	$		0A01	3A00	BNZ	00
GLO	REG-F		0A03	8F	GLO	RF
PLO	E		0A04	AE	PLO	RE
$07E0	LDI	$1234	07E0	F834	LDI	34
SEP	3		07E2	D3	SEP	R3
SEX	2		07E3	E2	SEX	R2
CALL	$600		07E4	D40600	CALL	
RETN			07E7	D5	RETN	
INP	D-0		07E8	60	IRX	
LDN	R0		07E9	00	IDL	
OUT	D-8		07EA	68	---	
0900	LDP		ERROR			

The above examples should answer some format questions you may have. The column on the left shows how input may appear in the buffer before calling ASMBLR. The column on the right shows how DISASM will disassemble the input. Some experimentation on your part will reveal what the assembler likes and doesn't like more explicitly than is possible to present here.

To Call ASMBLR

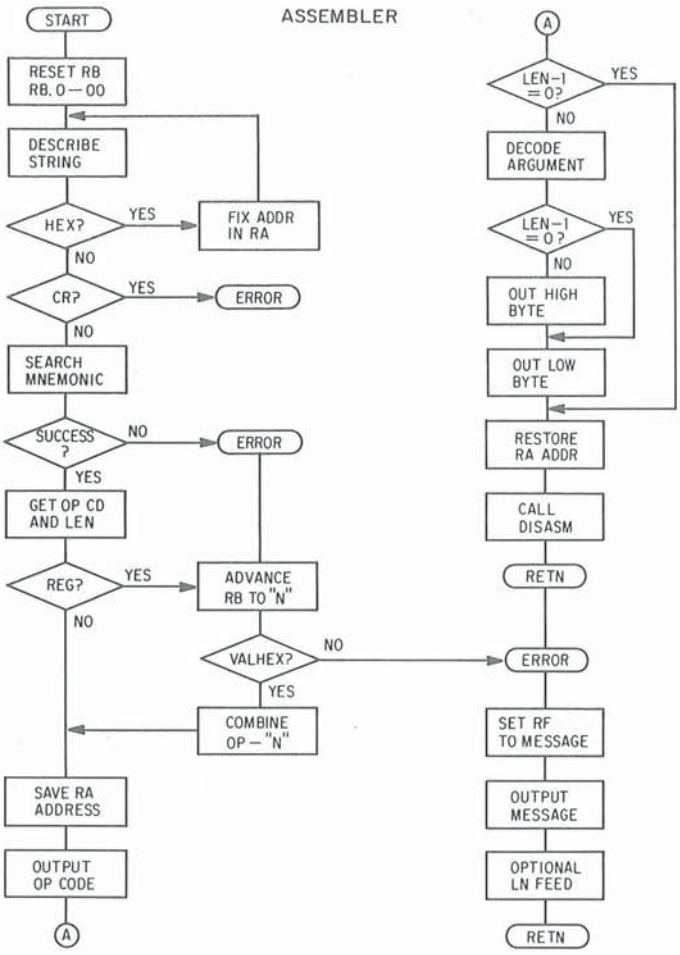
1. Put ASCII input into buffer
2. Save registers (as needed)
3. Set RB.1 to address buffer
4. Call ASMBLR (D40600)
5. Restore saved registers
6. Print echo in buffer

To Call DISASM (Optional)

1. Save registers (as needed)
2. Address op code with RA
3. Set RB.1 to address buffer
4. Call DISASM (D40700)
5. Restore saved registers
6. Print buffer

Notes on the Listing

The following program listing does not strictly follow assembly language formats since it is assumed that most users will be keying it in

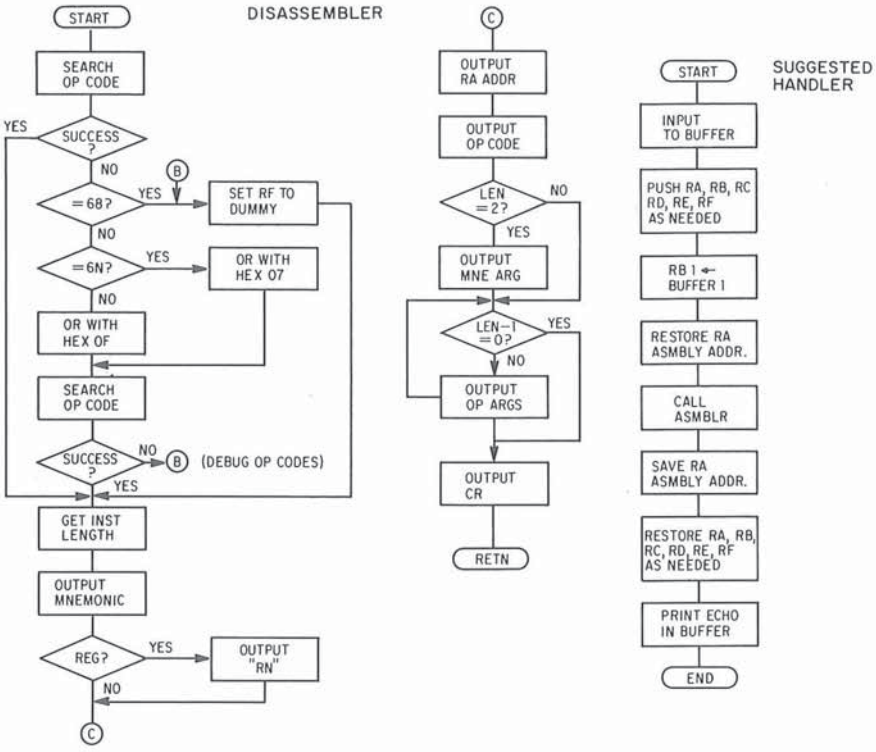


using absolute hex entry. CALL is SEP R4 followed by an address and RETN is SEP R5 with no argument.

On the other hand, all subroutine entries have been labeled and all branch destinations marked. This will be handy for those who wish to relocate the code elsewhere in memory.

Local jumps within a routine use local labels that begin with a number. The label 1H (the H means "here") marks a location that may be jumped to with a branch 1F ("forward") or 1B ("back"). Note that 1H may be repeated several times. A jump always proceeds in the indicated direction to the nearest local label of the same number. (Thanks go to D. Knuth, *The Art of Computer Programming*, for his explanation of this remarkably simple assembly technique, although he does not claim to be the originator of the idea.)

At location 0682 in the error handler of ASMBLR a call is made to your operating system's line feed routine. This would protect the bad input on display from being overwritten by the ERROR message, allowing examination of what went wrong. The call is optional, and you may terminate the routine with a D5 SEP R5 at 0682 if you wish.



REGISTER ASSIGNMENT

```
R0 - DMA Pointer - not used or changed
R1 - Interrupt PC - not used or changed
R2 - Stack Pointer - to be fixed by user
R3 - Program Counter
R4 - Call routine PC - to be fixed by user
R5 - Return routine PC - to be fixed by user
R6 - Pointer to return - to be fixed by user CALL/RETN routines
R7 - Undefined - not used or changed
R8 - Undefined - not used or changed
R9 - Undefined - not used or changed
RA - Pointer to final assembly - set by assembler - must not
     be changed by user!
RB - Pointer to Input Buffer - RB.1 set by user - RB.0 = 00 =
     top of buffer
RC - Utility - loops, flags, data passing, etc...
RD -    "          "       "       "       "       "
RE -    "          "       "       "       "       "
RF -    "          "       "       "       "       "
```

```
                    MEMORY ALLOCATION

0400 - 041F - Input Buffer (optional)
0420 - 07FF - Program
0800 - 09FF - Mnemonic Look up Table (ends with FF)
      (363 bytes available for expansion)

                    SUBROUTINES

0400 - 041F - 32 byte Input Buffer
0420 - 0428 - ASCHEX
0429 - 043D - ASCDIG
043E - 0450 - HEXASC
0451 - 045E - HEXDIG
045F - 0477 - DESCRB (Entry @ 0460)
0478 - 048D - LEN
048E - 04C1 - MNESCH
04C2 - 04E2 - OPSCH
04E3 - 04EF - REGTAB
04F0 - 04F5 - DUMMY
04F6 - 04FF - AVAILABLE (10 bytes)
0500 - 0528 - NUMBER
0529 - 0541 - VALHEX
0542 - 0555 - REGSCH
0556 - 055E - FIXRC
055F - 056A - ADVRB (Entry @ 0560)
056B - 0576 - OUTBUF (056D = OUTBUF2)
0577 - 0583 - ERRPNT
0584 - 05FF - AVAILABLE (124 bytes)

0600 - 0685 - ASMBLR (Main loop)
0686 - 06FF - AVAILABLE (122 bytes)

0700 - 0794 - DISASM (Main loop)
0795 - 07FF - AVAILABLE (107 bytes)

0800 - 09FF - MNEMONIC TABLE
```

```
;*****************************************************************
; ASCHEX:         ASCII/HEX CONVERSION
;*****************************************************************
; INPUT:      --  RE.1 = first ASCII digit (valid range)
;                 RE.0 = second ASCII digit (valid range)
;
; OUTPUT:     --  RF.1 = hexadecimal equivalent
;
; CALLS:      --  HEXDIG
;
; CALLED BY:  --  NUMBER
;
; CHANGES:    --  RE
;*****************************************************************

0420  ASCHEX:  D4  SEP   R4   ;Call ASCDIG with RE.1 holding
  21           04             ;   ASCII code for converting
  22           29
  23           8E  GLO   RE   ;Get second half ASCII code
  24           BE  PHI   RE   ;Put in RE.1 for passing to sub
  25           D4  SEP   R4   ;Call ASCDIG with RE.1 holding
  26           04             ;   ASCII code for converting
  27           29
  28           D5  SEP   R5   ;Return - hex value in RF.1
```

```
;*****************************************************************
; ASCDIG:          ASCII DIGIT CONVERSION
;*****************************************************************
; INPUT:      --   RE.1 = one ASCII digit
;
; OUTPUT:     --   RF.1 4 MSBs ← RF.1 4 LSBs;
;                  RF.1 4 LSBs ← one hex digit
;
; CALLED BY: --    ASCHEX
;
; CHANGES:    --   RE.1  RF.1
;*****************************************************************

0429  ASCDIG:  9E  GHI   RE    ;Get passed ASCII code from RE.1
  2A           FD  SDI   $39   ;Subtract from 39 hex -- if negative
  2B           39             ;    then ASCII > 39
  2C           33  BPZ   1F    ;Branch if positive or zero --
  2D           32             ;    ASCII ≤ 39 and is numerical
  2E           9E  GHI   RE    ;Get same ASCII code ( > 39)
  2F           FC  ADI   $09   ;Add 9 to convert all

0430           09             ;    alphabetical characters to range
  31           BE  PHI   RE    ;Replace in RE.1 then continue

0432  1H:      9E  GHI   RE    ;Get possibly adjusted ASCII code
  33           FA  ANI   $0F   ;AND with 0F hex to retain
  34           0F             ;    only the last four bits
  35           52  STR   R2    ;Push this value onto the stack
  36           9F  GHI   RF    ;Get RF.1 - possibly ½ hex value
  37           FE  SHL         ;Shift left 4 times to preserve
  38           FE  SHL         ;    the original low 4 bits and
  39           FE  SHL         ;    to open up space for
  3A           FE  SHL         ;    inserting a new hex digit
  3B           F1  OR          ;OR in new hex digit
  3C           BF  PHI   RF    ;Place value in RF.1
  3D           D5  SEP   R5    ;Return


;*****************************************************************
; HEXASC:          HEX/ASCII CONVERSION
;*****************************************************************
; INPUT:      --   RF.1 = hex value
;
; OUTPUT:     --   RE.1 = first ASCII digit
;                  RE.0 = second ASCII digit
;
; CALLS:      --   ASCDIG
;
; CALLED BY: --    DISASM  OUTBUF
;
; CHANGES:    --   RE
;*****************************************************************

043E  HEXASC:  9F  GHI   RF    ;Get hex byte from RF.1
  3F           F6  SHR         ;Shift right 4 times to move

0440           F6  SHR         ;    4 MSBs to LSB position
  41           F6  SHR         ;    (sub will convert one hex
  42           F6  SHR         ;    digit at a time)
  43           AE  PLO   RE    ;Put in RE.0 to pass to sub
  44           D4  SEP   R4    ;Call HEXDIG with hex digit in
  45           04              ;    RE.0 - returns ASCII equivalent
```

```
  46              51                      ;    in same register
  47              8E  GLO   RE            ;Transfer converted digit from
  48              BE  PHI   RE            ;    RE.0 to RE.1
  49              9F  GHI   RF            ;Get same hex byte from RF.1
  4A              FA  ANI   $0F           ;AND with 0F to retain only
  4B              0F                      ;    the low 4 bits (2nd digit)
  4C              AE  PLO   RE            ;Put in RE.0 for passing to sub
  4D              D4  SEP   R4            ;Call HEXDIG with hex digit in
  4E              04                      ;    RE.0 - returns ASCII equivalent
  4F              51                      ;    in same register
0450              D5  SEP   R5            ;Return - 2 byte ASCII in RE

;*********************************************************************
; HEXDIG:         HEX DIGIT CONVERSION
;*********************************************************************
; INPUT:      --  RE.0 = single hex digit in low 4 bits
;
; OUTPUT:     --  RE.0 = ASCII equivalent
;
; CALLED BY:  --  HEXASC
;
; CHANGES:    --  RE.0
;*********************************************************************

0451  HEXDIG:  8E  GLO   RE            ;Get hex digit passed in RE.0
  52            FD  SDI   $09           ;Subtract from 9 -- if negative
  53            09                      ;    then digit > 9, a letter
  54            33  BPZ   1F            ;Branch if positive or zero
  55            5A                      ;    digit is ≤ 9, a number
  56            8E  GLO   RE            ;Get same hex digit (>9)
  57            FC  ADI   $07           ;Add 7 to convert all letters
  58            07                      ;    to range
  59            AE  PLO   RE            ;Put in RE.0, then continue
  5A  1H:       8E  GLO   RE            ;Get digit, possibly adjusted
  5B            FC  ADI   $30           ;Add in ASCII code information
  5C            30                      ;    to complete conversion
  5D            AE  PLO   RE            ;Put in RE.0 to return ASCII code
  5E            D5  SEP   R5            ;Return

;*********************************************************************
; DESCRB:         DESCRIBE STRING
;*********************************************************************
; INPUT:      --  RB addresses input buffer
;
; OUTPUT:     --  RB addresses first character of string
;                 RF.0 = 1 = Mnemonic (probable)
;                 RF.0 = 2 = Numerical ($ sign)
;                 RF.0 = 3 = Carriage return
;
; CALLED BY:  --  ASMBLR
;
; CHANGES:    --  RB  RF.0
;*********************************************************************

045F  1H:       1B  INC   RB            ;Increment RB past spaces
  60  DESCRB:   0B  LDN   RB            ;Get character from input buffer
  61            FB  XRI   $20           ;Compare with ASCII for a
  62            20                      ;    space
  63            32  BZ    1B            ;If a space, branch to increment
  64            5F                      ;    RB past all spaces
```

```
0465            F8  LDI   $03  ;Load D with 03 byte
  66            03
  67            AF  PLO   RF   ;Set RF.0 = 03 to initialize  register
  68            0B  LDN   RB   ;Get same character from buffer
  69            FB  XRI   $0D  ;Compare with ASCII for a
  6A            0D             ;    carriage return
  6B            3A  BNZ   1F   ;If ≠ 0D, branch to skip over
  6C            6E             ;    next instruction
  6D            D5  SEP   R5   ;Return, RF.0 = 03 = carriage return
  6E  1H:       2F  DEC   RF   ;Decrement RF (RF.0 = 02)
  6F            0B  LDN   RB   ;Get same character from buffer

0470            FB  XRI   $24  ;Compare with ASCII for
  71            24             ;    $ sign (hexadecimal)
  72            3A  BNZ   1F   ;Branch on ≠ $ -- not a
  73            76             ;    hexadecimal
  74            1B  INC   RB   ;Advance RB past the $ sign
  75            D5  SEP   R5   ;Return.  RF.0 = 02 = numerical
  76  1H:       2F  DEC   RF   ;Decrement RF (RF.0 = 01)
  77            D5  SEP   R5   ;Return - probable mnemonic


;*****************************************************************
; LEN:            LENGTH STRING
;*****************************************************************
; INPUT:      --  RB addresses any string in buffer
;
; OUTPUT:     --  RE.1 and RE.0 = # bytes to next CR or space
;                 RB addresses same string (unchanged)
;
; CALLED BY:  --  MNESCH  NUMBER
;
; CHANGES:    --  RE
;*****************************************************************

0478  LEN:    8B  GLO   RB   ;Get RB.0 buffer pointer address low
  79          52  STR   R2   ;Push to save on stack (no decrement)
  7A          F8  LDI        ;Load 00 byte into D register
  7B          00
  7C          AE  PLO   RE   ;Put in RE.0 to initialize count
  7D          38  SKP        ;Skip into routine
  7E  1H:     1E  INC   RE   ;Increment count by 1 for each
                              character
  7F          0B  LDN   RB   ;Load a character @ M(R(B))

0480            FB  XRI   $20  ;Compare with ASCII for a
  81            20             ;    space
  82            32  BZ    2F   ;If = 20, then branch to exit --
  83            89             ;    RB is past end of string

0484            4B  LDA   RB   ;Get same non-space character (RB+1)
  85            FB  XRI   $0D  ;Compare with ASCII for a
  86            0D             ;    carriage return
  87            3A  BNZ   1B   ;If ≠ 0D (or 20) then branch to
  88            7E             ;    count & test next character
  89  2H:       8E  GLO   RE   ;Get count in RE.0
  8A            BE  PHI   RE   ;Put in RE.1 to pass back from sub
  8B            F0  LDX        ;Pop saved buffer address
  8C            AB  PLO   RB   ;Restore RB.0 to address the string
  8D            D5  SEP   R5   ;Return with length in RE.1/RE.0
```

```
;*************************************************************
; MNESCH:        MNEMONIC SEARCH
;*************************************************************
; INPUT:     --  RB addresses mnemonic ending with CR or space
;
; OUTPUT:    --  RD.0 = 0 = success/RD ≠ 0 = failure
;                RF addresses mnemonic in table on success
;
; CALLS:     --  LEN
;
; CALLED BY: --  ASMBLR
;
; CHANGES:   --  RD.0  RE  RF  (RB not changed)
;*************************************************************

048E  MNESCH:  F8  LDI        ;Load high address of mnemonic
  8F           08             ;   table into D

0490           BF  PHI   RF   ;Put in RF.1 (RF.1 = 08)
  91           F8  LDI        ;Load low address of mnemonic
  92           00             ;   table into D
  93           AF  PLO   RF   ;Put in RF.0 (RF = 0800)
  94           D4  SEP   R4   ;Call LEN. Returns length of
  95           04             ;   string @ M(R(B)) in RE.1
  96           78
  97  1H:      9E  GHI   RE   ;Get length from RE.1
  98           AE  PLO   RE   ;Put in RE.0 as a loop counter
  99           9F  GHI   RF   ;Transfer RF into RC.  This will
  9A           BC  PHI   RC   ;   preserve the value of RF.  RC
  9B           8F  GLO   RF   ;   will be used to make the
  9C           AC  PLO   RC   ;   comparison with bytes @ M(R(B))
  9D           8B  GLO   RB   ;Get low address of buffer pointer
  9E           52  STR   R2   ;Push to save start address of string
  9F           EC  SEX   RC   ;X = C.  M(R(X)) will be used
                               for comparing

04A0  2H:      4B  LDA   RB   ;Get a byte of string.  Advance RB
  A1           F3  XOR        ;Compare with byte in table @ M(R(C))
  A2           3A  BNZ   2F   ;If ≠, then comparison fails.  Branch
  A3           B0             ;   to set RD.0 ≠ 0
  A4           1C  INC   RC   ;Advance pointer RC to next table
                               character
  A5           2E  DEC   RE   ;Decrement count in RE.0 (# characters)
  A6           8E  GLO   RE   ;Test the count in RE.0
  A7           3A  BNZ   2B   ;If ≠ 0, then branch to continue
  A8           A0             ;   comparison.  Else comparison
                               succeeds
  A9           0C  LDN   RC   ;Get byte @ M(R(C)) on success
  AA           FD  SDI   $20  ;Subtract 20 - M(R(C)) to see if
  AB           20             ;   entire mnemonic was checked
  AC           3B  BM    2F   ;If negative, then M(R(C))>20 and
  AD           B0             ;   comparison fails.  (Also D≠0 here)
  AE           F8  LDI        ;Load 00 byte into D for marking
  AF           00             ;   the successful comparison

04B0  2H:      AD  PLO   RD   ;Put 00 or not 00 into RD.0 as flag
  B1           02  LDN   R2   ;Pop stack (X≠2 so LDX is improper)
  B2           AB  PLO   RB   ;Restore RB.0 buffer pointer address
  B3           8D  GLO   RD   ;Get success/fail flag in RD.0
  B4           3A  BNZ   2F   ;If ≠ 0 then comparison had failed
```

```
  B5              B7                ;   and test will continue
  B6              D5  SEP   R5      ;Return.  RD.0 = 00 indicates success
  B7  2H:         1F  INC   RF      ;Increment RF 6 times to address
  B8              1F  INC   RF      ;   next entry in the mnemonic table.
  B9              1F  INC   RF      ;   Though apparently redundant,
  BA              1F  INC   RF      ;   this is the fastest, shortest
  BB              1F  INC   RF      ;   way to increase RF x 6!
  BC              1F  INC   RF      ;
  BD              0F  LDN   RF      ;Load byte @ M(R(F))
  BE              FE  SHL           ;Shift left to test if MSB = 1
  BF              3B  BNF   1B      ;If DF ≠ 1 then branch to continue

04C0              97                ;   search.  (FF marks end of table)
  C1              D5  SEP   R5      ;Return.  RD.0 ≠ 0 marks failure

;*****************************************************************
; OPSCH:          OP CODE SEARCH
;*****************************************************************
; INPUT:      --  RE.1 holds op code (possibly modified)
;
; OUTPUT:     --  Same as MNESCH
;
; CALLED BY:  --  DISASM
;
; CHANGES:    --  RD.0  RF
;*****************************************************************

04C2  OPSCH:    EF  SEX   RF      ;X = F.  R(F) will address table entries
  C3            F8  LDI           ;Load 08 byte into D register
  C4            08
  C5            BF  PHI   RF      ;Put in RF.1
  C6            F8  LDI           ;Load 00 byte into D register
  C7            00
  C8            AF  PLO   RF      ;Put in RF.0.  RF=0800=top of table
  C9  1H:       4F  LDA   RF      ;Load byte from table, advance RF
  CA            FB  XRI   $FF     ;Compare with FF end of table
  CB            FF                ;   marker
  CC            32  BZ    2F      ;If =, branch to exit - search
  CD            DF                ;   has failed to find op code
  CE            1F  INC   RF      ;Increment RF x 4 to address the
  CF            1F  INC   RF      ;   op code field in table entries

04D0            1F  INC   RF      ;   "          "       "       "
  D1            1F  INC   RF      ;   "          "       "       "
  D2            9E  GHI   RE      ;Get op code being searched from RE.1
  D3            F3  XOR           ;Compare with op code in table @ M(R(X))
  D4            1F  INC   RF      ;RF ← RF+1 addresses next field
  D5            3A  BNZ   1B      ;Branch if comparison fails, otherwise
  D6            C9                ;   continue - op code found
  D7            AD  PLO   RD      ;RD.0 ← 00.  Mark successful search
  D8            2F  DEC   RF      ;Decrement RF x 6 to address
  D9            2F  DEC   RF      ;   the mnemonic of the
  DA            2F  DEC   RF      ;   field in which the
  DB            2F  DEC   RF      ;   op codes matched
  DC            2F  DEC   RF
  DD            2F  DEC   RF
  DE            D5  SEP   R5      ;Return - success
  DF  2H:       F8  LDI           ;Load 01 byte into D register

04E0            01
  E1            AD  PLO   RD      ;Put in RD.0.  Flag the failure
  E2            D5  SEP   R5      ;Return - failure
```

```
;*****************************************************************
; REGTAB:         REGISTER OP TABLE
;*****************************************************************

04E3  REGTAB:  0F              ;Each of these bytes
  E4           1F              ;   represents an instruction
  E5           2F              ;   group that requires a
  E6           4F              ;   register to be specified
  E7           5F              ;   in the second hex digit
  E8           6F              ;This table is used by both
  E9           8F              ;   the assembler and disassembler
  EA           9F
  EB           AF              ;Note that the hex digit F is
  EC           BF              ;   used here to hold the place
  ED           DF              ;   where the register digit will go
  EE           EF
  EF           67              ;Entries match mnemonic table op codes


;*****************************************************************
; DUMMY:          DUMMY MNEMONIC ENTRY
;*****************************************************************

04F0  DUMMY:   2D              ;Dummy entry for illegal
  F1           2D              ;   op code 68.  Simulates
  F2           2D              ;   an entry in the mnemonic/
  F3           20              ;   op code table and will
  F4           01              ;   cause "---" to be output
  F5           68              ;   to the buffer.


;*****************************************************************
; ERRMSG:         ERROR MESSAGE
;*****************************************************************

04F6  ERRMSG:  45              ;ASCII codes for "ERROR"
  F7           52              ;   ending with a null (00)
  F8           52              ;   byte.
  F9           4F              ;  "          "         "
  FA           52              ;  "          "         "
  FB           00              ;  "          "         "


;*****************************************************************
; NUMBER:         CONVERT NUMBERS
;*****************************************************************
; INPUT:     --  RB addresses hex string ending with CR or space
;
; OUTPUT:    --  RD = 16 bit equivalent of that string
;                RF.0 = 0 = no errors
;                RF.0 ≠ 0 = string > 4 digits/also RD=0000 if RF.0≠0
;                RB addresses byte after string unless RF.0≠0 then
;                   RB unchanged
;
; CALLS:     --  ASCHEX  LEN
;
; CALLED BY: --  ASMBLR
;
; CHANGES:   --  RB  RD  RE  RF
;*****************************************************************

0500  NUMBER:  F8  LDI         ;Load 00 into D register
  01           00
```

```
  02              BD  PHI   RD    ;Put in RD.1
  03              AD  PLO   RD    ;Put in RD.0 -- initialize result
                                   = 0000
  04              D4  SEP   R4    ;Call LEN -- determine length of
  05              04              ;   numerical string @ M(R(B))--
  06              78              ;   returns byte count in RE.1 & RE.0
  07              9E  GHI   RE    ;Get length returned in RE.1
  08              AF  PLO   RF    ;Put in RF.0
  09              FD  SDI   $4    ;Subtract 4 - LEN.  If negative
  0A              04              ;   then LEN > 4 and is illegal
  0B              33  BPZ   1F    ;If positive (LEN ≤ 4) then  branch
  0C              0E              ;   to continue NUMBER sub
  0D              D5  SEP   R5    ;Return.  RD=0000 default value for
                                   illegal #
  0E  1H:         F8  LDI         ;Load D with ASCII code for
  0F              30              ;   a zero

0510              BE  PHI   RE    ;Put in RE.1
  11              AE  PLO   RE    ;Put in RE.0.  RE=3030 to initialize
  12              8F  GLO   RF    ;Get string length
  13              32  BZ    3F    ;If = 0, branch directly to process
  14              1E              ;   ASCII codes in RE
  15              F6  SHR         ;Shift right to test length even or odd
  16              33  BDF   2F    ;If DF=1 then length is odd.  Branch
  17              1B              ;   to do only the single hex digit
  18              4B  LDA   RB    ;Get a character (hex) from buffer
  19              BE  PHI   RE    ;Put in RE.1 to pass to ASCHEX sub
  1A              2F  DEC   RF    ;Decrement string length
  1B  2H:         4B  LDA   RB    ;Get a character (hex) from buffer

051C              AE  PLO   RE    ;Put in RE.0 to pass to ASCHEX sub
  1D              2F  DEC   RF    ;Decrement string length
  1E  3H:         D4  SEP   R4    ;Call ASCHEX to process the
  1F              04              ;   ASCII codes in RE.1 & RE.0

0520              20
  21              8D  GLO   RD    ;Get low 8 bits of RD
  22              BD  PHI   RD    ;Put in RD.1.  Make room for answer
  23              9F  GHI   RF    ;Get hex value returned in RF.1
  24              AD  PLO   RD    ;Put in RD.0
  25              8F  GLO   RF    ;Test length to see if done yet
  26              3A  BNZ   1B    ;If ≠ 0. then branch back to
  27              0E              ;   do another byte
  28              D5  SEP   R5    ;Return.  Value in RD over 16 bits

;*****************************************************************
; VALHEX:         VALID HEX TEST
;*****************************************************************
; INPUT:      --  RE.1 = ASCII code for testing
;
; OUTPUT:     --  RF.0 = 0 = legal hex number 0-9, A-F
;                 RF.0 ≠ 0 = not legal hex number
;
; CALLED BY:  --  ASMBLR
;
; CHANGES:    --  RF.0
;*****************************************************************

0529  VALHEX:  F8  LDI         ;Load D with 00 byte
  2A              00
  2B              AF  PLO   RF    ;Put in RF.0 to initialize flag
```

```
  2C              9E  GHI   RE    ;Get ASCII code from RE.1
  2D              FD  SDI   $46   ;Subtract 46 - ASCII.  If positive
  2E              46              ;   then ASCII ≤ 46
  2F              3B  BM    1F    ;Branch on minus to signal

0530              40              ;   error. (47 ≤ ASCII ≤ FF)
  31              9E  GHI   RE    ;Get same ASCII code from RE.1
  32              FD  SDI   $40   ;Subtract 40 - ASCII.  If positive
  33              40              ;   then ASCII ≤ 40
  34              3B  BM    2F    ;Branch on minus to signal
  35              41              ;   success. (41 ≤ ASCII ≤ 46)
  36              9E  GHI   RE    ;Get same ASCII code from RE.1
  37              FD  SDI   $39   ;Subtract 39 - ASCII.  If positive
  38              39              ;   then ASCII ≤ 39
  39              3B  BM    1F    ;Branch on minus to signal
  3A              40              ;   error. (3A ≤ ASCII ≤ 40)
  3B              9E  GHI   RE    ;Get same ASCII code from RE.1

053C              FD  SDI   $2F   ;Subtract 2F - ASCII.  If positive
  3D              2F              ;   then ASCII ≤ 2F
  3E              3B  BM    2F    ;Branch on minus to signal
  3F              41              ;   success. (30 ≤ ASCII ≤ 39)

0540  1H:         1F  INC   RF    ;RF+1 signals failure (RF.0 ≠ 0)
  41  2H:         D5  SEP   R5    ;Return  (RF.0 = 0 = success/
                                   RF.0 ≠ 0 = failure)

;*********************************************************************
; REGSCH:         REGISTER OP CODE TABLE SEARCH
;*********************************************************************
; INPUT:      --  RE.1 = ASCII code for searching
;
; OUTPUT:     --  RF.0 = 0 = failure -- not in table
;                 RF.0 ≠ 0 = success -- in table
;
; CALLED BY:  --  ASMBLR  DISASM
;
; CHANGES:    --  RD  RF.0
;*********************************************************************

0542  REGSCH:   F8  LDI   $04   ;Load high address REGTAB
  43              04
  44              BD  PHI   RD  ;Put in RD.1
  45              F8  LDI   $E3 ;Load low address REGTAB
  46              E3
  47              AD  PLO   RD  ;Put in RD.0
  48              F8  LDI   $0D ;Load table size (# bytes)
  49              0D
  4A              AF  PLO   RF  ;Put in RF.0 as a loop counter
  4B              9E  GHI   RE  ;Get the code being searched
  4C              52  STR   R2  ;Push onto stack for the comparison
  4D  1H:         4D  LDA   RD  ;Get a byte from table
  4E              F3  XOR       ;Compare with byte on stack
  4F              32  BZ    2F  ;If =, branch to exit

0550              55              ;   match found
  51              2F  DEC   RF  ;Decrement loop count
  52              8F  GLO   RF  ;Test count in RF.0
  53              3A  BNZ   1B  ;If ≠ 0, branch to continue
  54              4D              ;   searching
  55  2H:         D5  SEP   R5  ;Return.  (RF.0 ≠ 0 = success/
                                   RF.0 = 0 = failure)
```

```
;*****************************************************************
; FIXRC:          FIX OP CODE/LENGTH IN RC
;*****************************************************************
; INPUT:      --  RF addresses mnemonic field in table
;
; OUTPUT:     --  RC.1 = op code from table
;                 RC.0 = length from table
;                 RF addresses last byte of entry (op code)
;
; CALLED BY: --  ASMBLR  DISASM
;
; CHANGES:    --  RC  RF
;*****************************************************************

0556  FIXRC:   1F  INC   RF    ;Increment RF to length field
  57           1F  INC   RF    ;   of mnemonic table entry
  58           1F  INC   RF    ;  "       "      "      "
  59           1F  INC   RF    ;  "       "      "      "
  5A  FIXRC2:  4F  LDA   RF    ;Load length byte from table
  5B           AC  PLO   RC    ;Put in RC.0 (= # bytes this instruction)
  5C           0F  LDN   RF    ;Load op code byte from table
  5D           BC  PHI   RC    ;Put in RC.1
  5E           D5  SEP   R5    ;Return  (RF addresses op code)

;*****************************************************************
; ADVRB:          ADVANCE RB
;*****************************************************************
; INPUT:      --  RB addresses string ending with space of CR
;
; OUTPUT:     --  RB addresses byte following the string
;
; CALLED BY: --  ASMBLR
;
; CHANGES:    --  RB
;*****************************************************************
055F  1H:       1B  INC   RB    ;RB← RB+1 to advance
  60  ADVRB:    0B  LDN   RB    ;Load byte @ M(R(B))-no advance
  61            FB  XRI   $20   ;Compare with ASCII for space
  62            20
  63            32  BZ    2F    ;If RB addresses a space, branch
  64            6A              ;   to exit
  65            0B  LDN   RB    ;Load byte @ M(R(B))-no advance
  66            FB  XRI   $0D   ;Compare with ASCII for
  67            0D              ;   carriage return
  68            3A  BNZ   1B    ;If RB does not address a carriage
  69            5F              ;   return, then loop to advance RB
  6A  2H:       D5  SEP   R5    ;Return.  RB advanced past string
;*****************************************************************
; OUTBUF:         OUTPUT TO BUFFER
;*****************************************************************
; INPUT:      --  RB addresses buffer where output is to go
;                 RA addresses byte to output
;
; OUTPUT:     --  ASCII equivalent (2 bytes) @ M(R(B))
;                 RB← RB+2/RA ← RA+1
;
; CALLS:      --  HEXASC
;
; CALLED BY: --  DISASM
;
; CHANGES:    --  RA  RB  RE  RF
;*****************************************************************
```

```
056B  OUTBUF:  4A  LDA   RA    ;Get M(R(A)).  Advance RA
  6C           BF  PHI   RF    ;Put in RF.1 to pass to sub
  6D  OUTBUF2: D4  SEP   R4    ;Call HEXASC.  Returns ASCII
  6E           04              ;   equivalent of byte in RF.1
  6F           3E              ;   in RE.1 and RE.0

0570           9E  GHI   RE    ;Get first digit in RE.1
  71           5B  STR   RB    ;Store in buffer @ M(R(B))
  72           1B  INC   RB    ;Advance RB
  73           8E  GLO   RE    ;Get second digit in RE.0
  74           5B  STR   RB    ;Store in buffer @ M(R(B))
  75           1B  INC   RB    ;Advance RB
  76           D5  SEP   R5    ;Return


;*****************************************************************
; ERRPNT:          PRINT ERROR MESSAGE
;*****************************************************************
; INPUT:     --  RF addresses message in ASCII ending with 00
;
; OUTPUT:    --  Message transferred to buffer via RB ending with 0D
;
; CALLED BY: --  ASMBLR
;
; CHANGES:   --  RB  RF
;*****************************************************************

0577  ERRPNT:  F8  LDI   $0   ;Load 00 byte into D
  78           00
  79           AB  PLO   RB    ;Put in RB.0 to reset buffer pointer
  7A  1H:      4F  LDA   RF    ;Load a character @ M(R(F))
  7B           5B  STR   RB    ;Store in buffer @  M(R(B))
  7C           1B  INC   RB    ;Increment buffer pointer

057D           3A  BNZ   1B    ;If byte stored was not 00, then
  7E           7A              ;   branch to store another
  7F           2B  DEC   RB    ;Else decrement RB to the 00 byte

0580           F8  LDI   $0D   ;Load ASCII for a carriage
  81           0D              ;   return
  82           5B  STR   RB    ;Store in buffer as end of string
  83           D5  SEP   R5    ;Return


;*****************************************************************
; ASMBLR:          ASSEMBLER MAIN LOOP
;*****************************************************************
; INPUT:     --  RA not altered by user
;                RB.1 set to input buffer page (00=top)
;                RC  RD  RE  RF free for use
;
; OUTPUT:    --  Input in buffer assembled @ M(R(A))
;
; CALLS:     --  ADVRB  DESCRB  DISASM  ERRPNT  FIXRC  LINFD(optional)
;                MNESCH  NUMBER  REGSCH  VALHEX
;
; CALLED BY: --  USER ROUTINE
;
; CHANGES:   --  RA (if address is input) RB.0  RC  RD  RE  RF
;*****************************************************************
```

```
0600 ASMBLR: F8 LDI  $00  ;Load 00 byte into D
  01         00
  02         AB PLO  RB   ;Put in RB.0.  Reset buffer register
  03 1H:     D4 SEP  R4   ;Call DESCRB.  Returns RF.0
  04         04           ;   with type of string
  05         60           ;   found
  06         8F GLO  RF   ;Get type of string
  07         FB XRI  $02  ;Compare with code for
  08         02           ;   numerical hex string
  09         3A BNZ  2F   ;If ≠ 2 then not a hex string
  0A         17           ;   branch to skip address fixing
  0B         D4 SEP  R4   ;Call NUMBER.  Returns RD
  0C         05           ;   with binary equivalent
  0D         00           ;   and advance RB past string
  0E         8F GLO  RF   ;Test error flag on return
  0F         3A BNZ  ERR2 ;If ≠ 0, then branch to output

0610         79           ;   error message and end
  11         9D GHI  RD   ;Transfer RD to RA.  RA will
  12         BA PHI  RA   ;   address final assembly
  13         8D GLO  RD   ;   area.  RA should not be
  14         AA PLO  RA   ;   changed by user program

0615         30 BN   1B   ;Loop back to start again.
  16         03           ;   Address fixed in RA
  17 2H:     8F GLO  RF   ;Get type of string (≠02) from RF.0
  18         FB XRI       ;Compare with code for a
  19         03           ;   carriage return
  1A         32 BZ   ERR2 ;If = CR, then branch to error.
  1B         79           ;   Must have mnemonic here
  1C         D4 SEP  R4   ;Call MNESCH to locate mnemonic
  1D         04           ;   @ M(R(B)) in mnemonic table
  1E         8E
  1F         8D GLO  RD   ;Get success/fail flag in RD.0

0620         3A BNZ  ERR2 ;If ≠ 0, then branch to error.  Search
  21         79           ;   was not successful
  22 2H:     D4 SEP  R4   ;Call ADVRB.  RB is advanced
  23         05           ;   past the mnemonic string ending
  24         60           ;   with either a space or CR
  25         D4 SEP  R4   ;Call FIXRC.  Returns op code in RC.1
  26         05           ;   and length in RC.0 from table
  27         56           ;   entry addressed by RF
  28         9C GHI  RC   ;Get op code in RC.1
  29         BE PHI  RE   ;Put in RE.1 to pass to sub
  2A         D4 SEP  R4   ;Call REGSCH.  Check if this op code
  2B         05           ;   will require a register specified
  2C         42           ;   in the 2nd hex digit
  2D         8F GLO  RF   ;Test success/fail flag returned
  2E         32 BZ   2F   ;If RF.0=0 then branch.  Op code
  2F         49           ;   is OK in present form

0630         D4 SEP  R4   ;Call DESCRB to advance RB
  31         04           ;   to next string (presumably a
  32         60           ;   RN or N desgination)
  33         D4 SEP  R4   ;Call ADVRB to advance RB past
  34         05           ;   the register designation (thus
  35         60           ;   disregarding the R or DEV or
                             whatever
  36         2B DEC  RB   ;RB addresses last byte of string
```

```
37          0B  LDN   RB    ;Get that byte (in ASCII form)
38          BE  PHI   RE    ;Put in RE.1 for testing
39          D4  SEP   R4    ;Call VALHEX to check if this
3A          05              ;   is a valid hex digit.  At this
3B          29              ;   point it must be a hex digit
3C          8F  GLO   RF    ;Get success/fail flag returned in RF.
3D          3A  BNZ   ERR2  ;If RF.0 ≠ 0 then branch to
3E          79              ;   error.  Must have a valid
                            ;   register specification here
3F          D4  SEP   R4    ;Call NUMBER.  Returns RD as 16 bit

0640        05              ;   equivalent of string @ M(R(B))
41          00              ;   and advances RB.
42          8D  GLO   RD    ;Get single digit (0N) from RD.0
43          52  STR   R2    ;Push for combining with op code
44          9C  GHI   RC    ;Get the op code from RC.1
45          FA  ANI   $F0   ;Logically AND with F0 hex thus
46          F0              ;   stripping off the last digit
47          F1  OR          ;OR in the new 2nd digit on stack
48          BC  PHI   RC    ;Put new op code back in RC.1
49  2H:     9A  GHI   RA    ;Get assembly address RA.1
4A          73  STXD        ;Push to save on stack (R(X)←R(X)-1)
4B          8A  GLO   RA    ;Get assembly address RA.0
4C          73  STXD        ;Push to save on stack (R(X)←R(X)-1)
4D          9C  GHI   RC    ;Get final op code in RC.1
4E          5A  STR   RA    ;Store in memory @ M(R(A))
4F          1A  INC   RA    ;Advance RA

0650        2C  DEC   RC    ;Decrement RC.  RC.0 has instruction
                             length
51          8C  GLO   RC    ;Test RC.0
52          32  BZ    3F    ;If RC.0 = 0 then assembly is complete
53          6B              ;   Branch to restore RA ? continue
54          D4  SEP   R4    ;Call DESCRB.  Advances RB to
55          04              ;   next string which at this point
56          60              ;   must be a hexadecimal
57          8F  GLO   RF    ;Get string type returned in RF.0
58          FB  XRI   $02   ;Compare with type code for
59          02              ;   a hexadecimal (02)
5A          3A  BNZ   ERR1  ;If not hexadecimal ($) indicator,
5B          74              ;   branch to error
5C          D4  SEP   R4    ;Call NUMBER.  Returns equivalent
5D          05              ;   16 bit value in RD and advances
5E          00              ;   RB past the string
5F          8F  GLO   RF    ;Get success/fail flag in RF.0

0660        3A  BNZ   ERR1  ;If RF.0 ≠ 0 then string too large.
61          74              ;   Branch to error (> 4 digits)
62          2C  DEC   RC    ;Decrement instruction length in RC.0
63          8C  GLO   RC    ;Test length
64          32  BZ    2F    ;If = 0 then only single byte
65          69              ;   argument required.  Branch
66          9D  GHI   RD    ;Get high byte of argument in RD.1
67          5A  STR   RA    ;Store in memory @ M(R(A))
68          1A  INC   RA    ;Advance RA
69  2H:     8D  GLO   RD    ;Get low byte of argument in RD.0
6A          5A  STR   RA    ;Store in memory @M(R(A))-no increment
6B  3H:     60  IRX         ;Point R(X) to saved RA on stack
6C          72  LDXA        ;Pop RA.0
6D          AA  PLO   RA    ;Restore old RA.0 address
```

```
066E            F0  LDX         ;Pop RA.1
  6F            BA  PHI   RA    ;Restore old RA.1 address

0670            D4  SEP   R4    ;Call DISASM to echo the
  71            07              ;   assembly for verification
  72            00              ;   printout. (Optional)
  73            D5  SEP   R5    ;Return. End ASMBLR Main Loop


                          ERROR HANDLER

0674  ERR1:     60  IRX         ;Point to saved RA.0 on stack
  75            72  LDXA        ;Pop RA.0
  76            AA  PLO   RA    ;Restore assembly address RA.0
  77            F0  LDX         ;Pop RA.1
  78            BA  PHI   RA    ;Restore assembly address RA.1
  79  ERR2:     F8  LDI         ;Load high address of
  7A            04              ;   error message
  7B            BF  PHI   RF    ;Put in RF.1
  7C            F8  LDI         ;Load low address of
  7D            F6              ;   error message
  7E            AF  PLO   RF    ;Put in RF.0
  7F            D4  SEP   R4    ;Call ERRPNT.  Output message

0680            05              ;   to buffer for printing
  81            77
  82            D4  SEP   R4    ;Call LINFD.  Optional call
  83            ??              ;   to be used if your system
  84            ??              ;   has such a subroutine
  85            D5  SEP   R5    ;Return.  Assembly aborted
```

```
;*****************************************************************
; DISASM:         DISASSEMBLER MAIN LOOP
;*****************************************************************
; INPUT:      --  RA addresses any 1802 instruction
;                 RB.1 set to input buffer page (00=top)
;                 RC  RD  RE  RF  free for use
;
; OUTPUT:     --  M(R(A)) disassembled into buffer ending
;                   with CR ($0D)
;                 RA advanced to next instruction
;
; CALLS:      --  FIXRC  HEXASC  OPSCH  OUTBUF  OUTBUF2  REGSCH
;
; CALLED BY:  --  ASMBLR  USER ROUTINE (optional)
;
; CHANGES:    --  RA  RB.0  RC  RD  RE  RF
;*****************************************************************

0700  DISASM:   F8  LDI   $1F   ;Load address of buffer end
  01            1F              ;   into D register (1F)
  02            AB  PLO   RB    ;Put in RB.0.  Initialize pointer
                                  to end
  03            EB  SEX   RB    ;X = B to keep the coming loop short
  04  1H:       F8  LDI   $20   ;Load 20 byte  (ASCII for
  05            20              ;   a space)
  06            73  STXD        ;Store @ M(R(B)).  RB←RB-1
  07            8B  GLO   RB    ;Test RB.0
  08            3A  BNZ   1B    ;If ≠ 00 yet, loop to fill buffer
  09            04              ;   with "20" bytes (except @RB.0=00)
  0A            0A  LDN   RA    ;Get instruction @ M(R(A))
  0B            BE  PHI   RE    ;Put in RE.1 to pass to OPSCH
```

```
  0C            D4  SEP   R4    ;Call OPSCH -- search op codes
  0D            04              ;   returning RF with address of
  0E            C2              ;   entry if found (also X=2 again)
  0F            8D  GLO   RD    ;Get success/fail flag from RD.0

0710            32  BZ    3F    ;If = 0 then branch.  Op code
  11            32              ;   found in table
  12            F8  LDI   $0F   ;Load 0F byte into D
  13            0F
  14            52  STR   R2    ;Push for later combining with op code
  15            9E  GHI   RE    ;Get op code from RE.1
  16            FB  XRI   $68   ;Test if = 68 the only illegal
  17            68              ;   1802 op code
  18            3A  BNZ   2F    ;If not, branch to skip the
  19            22              ;   next special handling
  1A  1H:       F8  LDI         ;Load address of dummy
  1B            04              ;   table entry
  1C            BF  PHI   RF    ;Put in RF so to fake a

071D            F8  LDI         ;   successful search for the
  1E            F0              ;   68 code or debug error
  1F            AF  PLO   RF    ;

0720            30  BN    3F    ;Branch to simulate a successful
  21            32              ;   search with RF @ dummy entry
  22  2H:       FA  ANI   $F0   ;Strip last digit of code (≠68)
  23            F0              ;   to test first digit = 6
  24            3A  BNZ   2F    ;If ≠ 0 then first digit ≠ 6
  25            29              ;   branch to skip next part
  26            F8  LDI   $07   ;Load 07 byte
  27            07
  28            52  STR   R2    ;Push in place of 0F on stack
  29  2H:       9E  GHI   RE    ;Get op code held in RE.1
  2A            F1  OR          ;OR to form NF or N7 byte
  2B            BE  PHI   RE    ;Put back in RE.1
  2C            D4  SEP   R4    ;Call OPSCH.  If it fails
  2D            04              ;   this time then there is a
  2E            C2              ;   problem with the op code table
  2F            8D  GLO   RD    ;Get success/fail flag in RD.0

0730            3A  BNZ   1B    ;If ≠ 0 then search failed.  Debug
  31            1A              ;   op codes in table
  32  3H:       D4  SEP   R4    ;Call FIXRC.  Returns raw
  33            05              ;   op code in RC.1 (not needed)
  34            56              ;   and length in RC.0
  35            2F  DEC   RF    ;Decrement RF 5 times to reset to
  36            2F  DEC   RF    ;   address of mnemonic in
  37            2F  DEC   RF    ;   table
  38            2F  DEC   RF    ;  "        "      "        "
  39            2F  DEC   RF    ;  "        "      "        "
  3A            F8  LDI         ;Load tab address in buffer for
  3B            0A              ;   normal mnemonics
  3C            AB  PLO   RB    ;Put in RB.0
  3D            8C  GLO   RC    ;Get length code from RC.0
  3E            FB  XRI   $3    ;Test if this is a 3 byte
  3F            03              ;   instruction code

0740            3A  BNZ   2F    ;If not, then branch to skip
  41            45              ;   next tab (needed for 16 char. line)
  42            F8  LDI         ;Load tab address in buffer for
  43            0C              ;   3 byte instruction mnemonics
  44            AB  PLO   RB    ;Put in RB.0 (long lines may need
                                   no adjustment)
```

```
45  2H:     F8  LDI   04   ;Load 04 byte into D register
46          04
47          AE  PLO   RE   ;Put in RE.0 as loop counter
48  2H:     4F  LDA   RF   ;Load a character @ M(R(F))
49          5B  STR   RB   ;Store in buffer @ M(R(B))

074A            1B  INC   RB   ;Increment RB buffer pointer
 4B             2E  DEC   RE   ;Count # characters transferred
 4C             8E  GLO   RE   ;Test count in RE.0
 4D             3A  BNZ   2B   ;If ≠ 0 branch to transfer 4
 4E             48             ;   characters always
 4F             D4  SEP   R4   ;Call REGSCH.  RE.1 holds the

0750            05             ;   op code as found in table.
 51             42             ;   Check to see if RN needed.
 52             8F  GLO   RF   ;Get success/fail flag RF.0
 53             32  BZ    2F   ;If = 0 then RN not needed.  Branch
 54             63             ;   to skip RN output
 55             F8  LDI   $0E  ;Load tab for RN output
 56             0E
 57             AB  PLO   RB   ;Put in RB.0 buffer pointer
 58             F8  LDI   $52  ;Load ASCII code for R character
 59             52
 5A             5B  STR   RB   ;Store in buffer @ M(R(B))
 5B             1B  INC   RB   ;Increment RB to next byte
 5C             0A  LDN   RA   ;Get instruction @ M(R(A))
 5D             BF  PHI   RF   ;Put in RF.1 to pass to sub
 5E             D4  SEP   R4   ;Call HEXASC to convert digit to
 5F             04             ;   ASCII.  Note, RE changed

0760            3E             ;   by this call.
 61             8E  GLO   RE   ;Get the digit from RE.0
 62             5B  STR   RB   ;Store in buffer just after R
 63  2H:        F8  LDI        ;Load tab address for the address
 64             00             ;   output (should not be changed)
 65             AB  PLO   RB   ;Put in RB.0 buffer pointer
 66             9A  GHI   RA   ;Get high byte of address
 67             BF  PHI   RF   ;Put in RF.1 to pass to sub
 68             D4  SEP   R4   ;Call OUTBUF2 to output
 69             05             ;   RA.1 address
 6A             6D
 6B             8A  GLO   RA   ;Get low byte of address
 6C             BF  PHI   RF   ;Put in RF.1 to pass to sub
 6D             D4  SEP   R4   ;Call OUTBUF2 to output
 6E             05             ;   RA.0 address
 6F             6D

0770            F8  LDI        ;Load tab for printing the
 71             05             ;   op codes
 72             AB  PLO   RB   ;Put in RB.0 buffer pointer
 73             D4  SEP   R4   ;Call OUTBUF to print the
 74             05             ;   op code
 75             6B
 76             8C  GLO   RC   ;Get byte count in RC.0
 77             FB  XRI   $2   ;Test if = 2 in which case a

0778            02             ;   single byte argument is needed
 79             3A  BNZ   2F   ;If ≠ 2 then branch to skip
 7A             82             ;   outputting the argument
```

```
  7B            F8  LDI   $0E  ;Load tab for single byte
  7C            0E            ;   argument (just after mnemonic)
  7D            AB  PLO   RB  ;Put in RB.0 buffer pointer
  7E            D4  SEP   R4  ;Call OUTBUF to print the
  7F            05            ;   argument

0780            6B
  81            2A  DEC   RA  ;Reset RA to address same byte
  82  2H:       F8  LDI   $07 ;Load tab for printing
  83            07            ;   arguments after op codes
  84            AB  PLO   RB  ;Put in RB.0 buffer pointer
  85            30  BN    2F  ;Branch into the next loop
  86            8A            ;   to start
  87  3H:       D4  SEP   R4  ;Call OUTBUF.  Output
  88            05            ;   byte @ M(R(A)) to buffer
  89            6B            ;   as long as RC.0 ≠ 0
  8A  2H:       2C  DEC   RC  ;Count # bytes output
  8B            8C  GLO   RC  ;Get count in RC.0
  8C            3A  BNZ   3B  ;If ≠ 0 branch to output
  8D            87            ;   a byte to buffer
  8E            F8  LDI   $10 ;Load tab for carriage
  8F            10            ;   return

0790            AB  PLO   RB  ;Put in RB.0 buffer pointer
  91            F8  LDI   $0D ;Load ASCII for a
  92            0D            ;   carriage return
  93            5B  STR   RB  ;Store @ M(R(B)) in buffer
  94            D5  SEP   R5  ;Return.  Disassembly complete

;*****************************************************************
; MNEMONIC OP CODE TABLE
;*****************************************************************
; FORMAT:     --  / 1 / 2 / 3 / 4 // 5 / 6 /
;                   Mnemonic    Lng OpCode
; MNEMONIC:   --  Bytes 1 - 4.  In ASCII with spaces (hex 20)
;                 to fill to 4 bytes if needed.
;
; LENGTH:     --  In hex.  Number of bytes required by this
;                 instruction.  For example, short branches = 2
;
; OP CODE:    --  1802 operation code in hex.  Register
;                 reference types ("RN") have hex F as second digit.
;
; NOTES:      --  End of table must be hex FF byte.  Add the
;                 following two entries to enable CALL and RETN
;                   084E  43 41 4C 4C 03 D4 CALL
;                   094A  52 45 54 4E 01 D5 RETN
;                 These replace NBR and NLBR but may be added
;                 to the end of the table alternatively.  Also,
;                 RSHR and RSHL are missing.  These too may be
;                 added to the end of the table of by replacing
;                 other duplicate mnemonic entries.
;
;                   0MMM  52 53 48 52 01 76 RSHR (=SHRC)
;                   0MMM  52 53 48 4C 01 7E RSHL (=SHLC)
;
;                 These omissions keep the table to a convenient
;                 2 page length, but it may be expanded to any
;                 size to accommodate the full mnemonic set.
;*****************************************************************
```

```
0800    49 44 4C 20 01 00    ;IDL
0806    4C 44 4E 20 01 0F    ;LDN
080C    49 4E 43 20 01 1F    ;INC
0812    44 45 43 20 01 2F    ;DEC
0818    42 52 20 20 02 30    ;BR
081E    42 51 20 20 02 31    ;BQ
0824    42 5A 20 20 02 32    ;BZ
082A    42 44 46 20 02 33    ;BDF
0830    42 31 20 20 02 34    ;B1
0836    42 32 20 20 02 35    ;B2

083C    42 33 20 20 02 36    ;B3
0842    42 34 20 20 02 37    ;B4
0848    53 4B 50 20 01 38    ;SKP
084E    4E 42 52 20 02 38    ;NBR
0854    42 4E 51 20 02 39    ;BNQ
085A    42 4E 5A 20 02 3A    ;BNZ
0860    42 4E 46 20 02 3B    ;BNF
0866    42 4E 31 20 02 3C    ;BN1

086C    42 4E 32 20 02 3D    ;BN2
0872    42 4E 33 20 02 3E    ;BN3
0878    42 4E 34 20 02 3F    ;BN4
087E    4C 44 41 20 01 4F    ;LDA
0884    53 54 52 20 01 5F    ;STR
088A    49 52 58 20 01 60    ;IRX
0890    4F 55 54 20 01 67    ;OUT
0896    49 4E 50 20 01 6F    ;INP
089C    52 45 54 20 01 70    ;RET
08A2    44 49 53 20 01 71    ;DIS

08A8    4C 44 58 20 01 F0    ;LDX
08AE    53 54 58 44 01 73    ;STXD
08B4    41 44 43 20 01 74    ;ADC
08BA    53 44 20 20 01 F5    ;SD
08C0    53 48 52 20 01 F6    ;SHR
08C6    53 4D 20 20 01 F7    ;SM
08CC    53 41 56 20 01 78    ;SAV
08D2    4D 41 52 4B 01 79    ;MARK
08D8    52 45 51 20 01 7A    ;REQ
08DE    53 45 51 20 01 7B    ;SEQ

08E4    41 44 43 49 02 7C    ;ADCI
08EA    53 44 42 20 01 75    ;SDB
08F0    53 48 4C 20 01 FE    ;SHL
08F6    53 4D 42 20 01 77    ;SMB
08FC    47 4C 4F 20 01 8F    ;GLO
0902    47 48 49 20 01 9F    ;GHI
0908    50 4C 4F 20 01 AF    ;PLO
090E    50 48 49 20 01 BF    ;PHI
0914    4C 42 52 20 03 C0    ;LBR
091A    4C 42 51 20 03 C1    ;LBQ

0920    4C 42 5A 20 03 C2    ;LBZ
0926    4C 42 44 46 03 C3    ;LBDF
092C    4E 4F 50 20 01 C4    ;NOP
0932    4C 53 4E 51 01 C5    ;LSNQ
0938    4C 53 4E 5A 01 C6    ;LSNZ
093E    4C 53 4E 46 C1 C7    ;LSNF
0944    4C 53 4B 50 01 C8    ;LSKP
094A    4E 4C 42 52 03 C8    ;NLBR
0950    4C 42 4E 51 03 C9    ;LBNQ
0956    4C 42 4E 5A 03 CA    ;LBNZ
```

```
095C    4C 42 4E 46 03 CB   ;LBNF
0962    4C 53 49 45 01 CC   ;LSIE
0968    4C 53 51 20 01 CD   ;LSQ
096E    4C 53 5A 20 01 CE   ;LSZ
0974    4C 53 44 46 01 CF   ;LSDF
097A    53 45 50 20 01 DF   ;SEP
0980    53 45 58 20 01 EF   ;SEX
0986    4C 44 58 41 01 72   ;LDXA
098C    4F 52 20 20 01 F1   ;OR
0992    41 4E 44 20 01 F2   ;AND

0998    58 4F 52 20 01 F3   ;XOR
099E    41 44 44 20 01 F4   ;ADD
09A4    53 44 42 49 02 7D   ;SDBI
09AA    53 48 52 43 01 76   ;SHRC
09B0    53 4D 42 49 02 7F   ;SMBI
09B6    4C 44 49 20 02 F8   ;LDI
09BC    4F 52 49 20 02 F9   ;ORI
09C2    41 4E 49 20 02 FA   ;ANI
09C8    58 52 49 20 02 FB   ;XRI
09CE    41 44 49 20 02 FC   ;ADI

09D4    53 44 49 20 02 FD   ;SDI
09DA    53 48 4C 43 01 7E   ;SHLC
09E0    53 4D 49 20 02 FF   ;SMI
09E6    42 50 5A 20 02 33   ;BPZ
09EC    42 47 45 20 02 33   ;BGE
09F2    42 4D 20 20 02 3B   ;BM
09F8    42 4C 20 20 02 3B   ;BL
09FE    FF                  ;End of table marker
```

□ □ □

Answers to Exercises

1010 Little Indians

1. Because of the ease with which the binary number system coefficients 1 and 0 may be represented electrically.
2. a) 15 b) 11 c) 219 d) 230
3. a) 0110 0100 b) 0100 0000 c) 1111 1001 d) 0101 0111

Binary Arithmetic

1 and 2.

a)
$$\begin{array}{r} 1011\ 1100 = 188 \\ +\ 0110\ 0110 = 102 \\ \hline 1\ 0010\ 0010 = 290 \end{array}$$

b)
$$\begin{array}{r} 110\ 0000 = 96 \\ +\ 010\ 1111 = 47 \\ \hline 1000\ 1111 = 143 \end{array}$$

c)
$$\begin{array}{r} 0111\ 1111 = 127 \\ +\ 0000\ 0001 = 1 \\ \hline 1000\ 0000 = 128 \end{array}$$

d)
$$\begin{array}{r} 1101 = 13 \\ 0101 = 5 \\ 0110 = 6 \\ 1001 = 9 \\ \hline 10\ 0001 = 33 \end{array}$$

e)
$$\begin{array}{r} 1111 = 15 \\ 0010 = 2 \\ 0101 = 5 \\ 1000 = 8 \\ \hline 1\ 1110 = 30 \end{array}$$

f)
$$\begin{array}{r} 1011 = 11 \\ 1101 = 13 \\ 1110 = 14 \\ 0111 = 7 \\ \hline 10\ 1101 = 45 \end{array}$$

3. The unary number system base 1 would contain only one coefficient. If we specify its only symbol to be a 1, then the decimal value 7 would look like this:

$$1111111 \text{ base } 1 = 7 \text{ base } 10$$

Here is the proof:

$$\begin{array}{rll} 1111111_1 = & 1 \times 1^6 = 1 \times (1 \times 1 \times 1 \times 1 \times 1 \times 1) & = 1 \\ & 1 \times 1^5 = 1 \times (1 \times 1 \times 1 \times 1 \times 1) & = 1 \\ & 1 \times 1^4 = 1 \times (1 \times 1 \times 1 \times 1) & = 1 \\ & 1 \times 1^3 = 1 \times (1 \times 1 \times 1) & = 1 \\ & 1 \times 1^2 = 1 \times (1 \times 1) & = 1 \\ & 1 \times 1^1 = 1 \times (1) & = 1 \\ & 1 \times 1^0 = 1 \times 1 & = 1 \\ & & \hline & & 7_{10} \end{array}$$

4.

a) 1111 1111 = 255
− 0000 0001 = 1
1111 1110 = 254

b) 1010 1010 = 170
− 0101 0101 = 85
0101 0101 = 85

c) 1101 1011 = 219
− 1011 1101 = 189
0001 1110 = 30

The Hexadecimal Number System

1. a) 1101 1110 1010 1111 b) 1100 1010 1111 1110
   c) 1111 1010 1100 1110 d) 1011 1110 1010 1101
2. a) C6 b) F5 c) ACE6 d) 89D2
3. a) 7FB6 b) EA92 c) 2B68 d) 9EF4
4. a) 498 b) 10,751 c) 33,900 d) 61,936
5. a) $5{,}199_{10}$/$144F_h$/0001 0100 0100 1111_2
   b) $61{,}134_{10}$/$EECE_h$/1110 1110 1100 1110_2
   c) $53{,}142_{10}$/$CF96_h$/1100 1111 1001 0110_2

Fundamentals of Assembly Language

Alternate to Chapter 2 Erase 4096 bytes subroutine

```
        LDI     $2F     ;Load the address 2FFF
        PHI     RE      ;     into register RE
        LDI     $FF
        PLO     RE
        SEX     RE      ;Set X = E
LOOP:   LDI     $0      ;Load 00 byte
        STXD            ;Store at RE 7 decrement
        GHI     RE      ;Test RE.1 by comparing with
        XRI     $1F     ;  hex 1F
        BNZ     LOOP    ;If ≠ 1F then loop
        RETN            ;Else return (RE = 1FFF)
```

APPENDIX

□ □ □

A Mini Library

The idea of writing this 1802 manual led to the writing of the assembler in Chapter 4. The assembler led in turn to the following subroutine library.

Some of these subroutines are quite simple—ADDIT and SUBTR, for example—and have been included both for their usefulness and for their instructive value. Many of the programming concepts discussed in previous pages are utilized by these subroutines.

The more advanced routines such as DEQUE and INSORT may take some time to study and understand.

Each routine is well documented and each has been thoroughly tested using the assembler in Chapter 4 running on the author's 4K Cosmac VIP computer with an extremely makeshift, rather flimsy keyboard donated by a friend. Experimenting with computers does not necessarily require expensive, fancy equipment.

Note that some of the routines call others, but that each performs only one carefully defined function. This modularity is highly desired in a subroutine library so that many programs may make use of the same routines. Also, register use has been limited as much as possible for compatibility with operating systems software. The programmer may, of course, use whichever registers are available.

A library such as this will cut down programming time, and serious programmers will want to add their own routines as these are developed. Then, the next time a block of memory needs to be sorted, the *debugged, tested* subroutine from the library may be trusted to do the job.

```
;*********************************************************************
; ADDIT:          DOUBLE PRECISION ADDITION
;*********************************************************************
; INPUT:      --  RE = operand #1
;                 RF = operand #2
;
; OUTPUT:     --  RE<- RE + RF using double precision
;                 DF indicates if overflow occurred
;
; CHANGES:    --  RE
;*********************************************************************
```

```
ADDIT:  GLO   RE          ;Get low byte operand #1
        STR   R2          ;Push onto stack @ M(R(2))
        GLO   RF          ;Get low byte operand #2
        ADD               ;Add to byte on stack via R(X)
        PLO   RE          ;Put result in RE.0
        GHI   RE          ;Get high byte operand #1
        STR   R2          ;Push onto stack @ M(R(2))
        GHI   RF          ;Get high byte operand #2
        ADC               ;Add with possible carry to M(R(X))
        PHI   RE          ;Put result in RE.1
        RETN              ;Return from subroutine
;*************************************************************
; SUBTR:        DOUBLE PRECISION SUBTRACTION
;*************************************************************
; INPUT:    --  RE = minuend
;               RF = subtrahend
;
; OUTPUT:   --  RE <- RE - RF using double precision
;               DF indicates if underflow (borrow) occurred
;
; CHANGES:  --  RE
;*************************************************************
SUBTR:  GLO   RE          ;Get low byte of minuend
        STR   R2          ;Push onto stack @ M(R(2))
        GLO   RF          ;Get low byte of subtrahend
        SD                ;Subtract from byte on stack
        PLO   RE          ;Put result in RE.0
        GHI   RE          ;Get high byte of minuend
        STR   R2          ;Push onto stack @ M(R(2))
        GHI   RF          ;Get high byte of subtrahend
        SDB               ;Subtract with possible borrow
                          ;from M(R(X))
        PHI   RE          ;Put result in RE.1
        RETN              ;Return from subroutine

;*************************************************************
; MULT1:        MULTIPLY ROUTINE #1 (simple - slow)
;*************************************************************
; INPUT:    --  RF.1 = multiplicand
;               RF.0 = multiplier
;
; OUTPUT:   --  RE = RF.1 x RF.0 using successive addition
;
; CHANGES:  --  RE  RF.0
;*************************************************************
MULT1:  LDI   $0          ;Load 00 byte into D
        PHI   RE          ;Put in RE.1 and RE.0 to initialize
        PLO   RE          ;    the answer to zero
        GHI   RF          ;Get multiplicand from RF.1
        STR   R2          ;Push onto stack @ M(R(2))
        BR    2F          ;Jump to begin loop
1H:     GLO   RE          ;Get low byte of answer in RE.0
        ADD               ;Add to RF.1 value on stack
        PLO   RE          ;Replace result in RE.0
        GHI   RE          ;Get high part of answer in RE.1
        ADCI  $0          ;Add possible carry by adding DF + 00
        PHI   RE          ;Replace result in RE.1
        DEC   RF          ;Decrement multiplier used as loop count
2H:     GLO   RF          ;Get value of RF.0 multiplier
        BNZ   1B          ;If not yet = 00, loop to continue
        RETN              ;Return.  16 bit answer in RE
```

```
;*****************************************************************
; MULT2:          MULTIPLY ROUTINE #2 (advanced - fast)
;*****************************************************************
; INPUT:      --  RF.1 = multiplier
;                 RF.0 = multiplicand
;
; OUTPUT:     --  RE = RF.0 x RF.1 using bit shifting
;
; CHANGES:    --  RE  RF
;*****************************************************************

MULT2:    LDI   $0            ;Load 00 byte into D
          PHI   RE            ;Put in RE.1 to initialize answer
          GLO   RF            ;Get multiplicand in RF.0
          STR   R2            ;Push onto stack @ M(R(2))
          LDI   $8            ;Load 08 into D
          PLO   RF            ;Put in RF.0 as a loop count (old
                              ;RF.0 on stack)
1H:       GHI   RF            ;Get multiplier from RF.1
          SHR                 ;Shift LSB into DF

          PHI   RF            ;Put shifted value back in RF.1
          GHI   RE            ;Get high byte of answer into D
          BNF   2F            ;If DF = 0 then jump to skip
                              ;next instruction
          ADD                 ;Add multiplicand on stack to
                              ;D on DF =1
2H:       SHRC                ;Shift D right with carry
          PHI   RE            ;Put in RE.2, high byte of answer
          GLO   RE            ;Get low byte of answer from RE.0
          SHRC                ;Shift with possible carry
          PLO   RE            ;Put in RE.0 now double precision
                              ;shifted right
          DEC   RF            ;Decrement loop count in RF.0
          GLO   RF            ;Get loop count to test value
          BNZ   1B            ;If ≠ 0, loop to test all eight
                              ;bits of multiplier
          RETN                ;Return.  16 bit answer in RE

;*****************************************************************
; DIVIDE:         DIVISION
;*****************************************************************
; INPUT:      --  RE = dividend (16 bits)
;                 RF = divisor (16 bits)
;
; OUTPUT:     --  RD = RE/RF by successive subtraction
;                 RE = remainder
;                 Divide by zero produces zero answer rather
;                   than an error
;
; CALLS:      --  ADDIT SUBTR
;
; CHANGES:    --  RD  RE
;*****************************************************************

DIVIDE:   LDI   $0            ;Load 00 byte into D
          PLO   RD            ;Put in RD.0 and RD.1 to
          PHI   RD            ;   initialize answer = 0000
          GLO   RF            ;Test low byte of divisor
          BNZ   2F            ;If ≠ 0, branch to continue
          GHI   RF            ;Else test high byte of divisor
          BNZ   2F            ;If ≠ 0, branch to continue
          RETN                ;Return on divide by 0.  Answer = 0
```

```
1H:     INC   RD          ;Add 1 to answer on loop
2H:     CALL  SUBTR       ;Subtract RF from RE
        BDF   1B          ;Loop on positive result.  DF = 1
        CALL  ADDIT       ;Add RF to RE on subtracting to
                          ;negative
        RETN              ;Return from subroutine

;*************************************************************
; RANDM1:        RANDOM SEQUENCE GENERATOR
;*************************************************************
; INPUT:     --  R9.1 = seed for start (any value)
;
; OUTPUT:    --  Repeatable random sequences of 0-255 in R9.1.
;                Binary bits are more random toward the MSB's.
;                Each number occurs once before sequence
;                  repeats (period 256)
;
; CALLS:     --  MULT2  (optional MULT1)
;
; CHANGES:   --  R9.1  RE  RF
;*************************************************************

RANDM1: GHI   R9          ;Get seed from R9.1
        PHI   RF          ;Put in RF.1 to pass to MULT2
        LDI   $65         ;Load constant for multiplication
        PLO   RF          ;Put in RF.0 to pass to MULT2
        CALL  MULT2       ;Multiply RE ← RF.1 x RF.0
        GLO   RE          ;Get low byte of result in RE.0
        ADI   $35         ;Add constant
        PHI   R9          ;Put in R9.1 as next # in sequence
        RETN              ;Return from subroutine

;*************************************************************
; CLEAR: FILL:   CLEAR / FILL MEMORY
;*************************************************************
; INPUT:     --  RE = end address
;                RF = start address (for RF ≤ RE)
;                RD.0 = byte to fill on call to FILL
;
; OUTPUT:    --  Memory cleared or filled with bytes in RD.0
;                Aborts on RF > RE
;
; CALLS:     --  SUBTR
;
; CHANGES:   --  RD.0 (Clear only)  RE  RF
;*************************************************************

CLEAR:  LDI   $0          ;Load 00 byte into D
        PLO   RD          ;Put in RD.0 to hold erase byte
FILL:   CALL  SUBTR       ;Subtract RE - RF.  RE = # bytes
                          ;to clear
        BM    2F          ;Branch on negative.  RF > RE.
                          ;Abort sub
        INC   RE          ;At least 1 byte will be cleared
1B:     GLO   RD          ;Get byte from RD.0
        STR   RF          ;Store @ M(R(F))
        INC   RF          ;Add one to RF pointer to advance
        DEC   RE          ;Decrement byte count in RE
        GLO   RE          ;Test RE.0 count
        BNZ   1B          ;If ≠ 0, branch to do another byte
        GHI   RE          ;Test RE.1 count on RE.0 = 0
        BNZ   1B          ;If ≠ 0, branch to do more bytes
2H:     RETN              ;Return from subroutine
```

```
;*****************************************************************
; MATIND:        MATRIX ARRAY INDEXING FOR X,Y ARRAYS
;*****************************************************************
; INPUT:     --  RD = base address (first element location) Matrix
;                RE.1 = X index
;                RF.1 = Y index
;                RF.0 = maximum X elements ((0,0) = first element
;                  so an X dimension of 4 equals 5 maximum X
;                  elements)
;
; OUTPUT:    --  RE = address element (X,Y)  Each element is a
;                  2 byte double precision entry unless using
;                  single precision option
;
; CALLS:     --  MULT2  (optional MULT1)  ADDIT
;
; CHANGES:   --  RE  RF
;
; NOTES:     --  For single byte arrays skip multiply by 2.
;                  String arrays possible by storing the address
;                  of the string location as an array element.
;                To "DIMension" an array, call with max X,Y for
;                  address of last element.  RE + 2 is next
;                  array base address.
;*****************************************************************

MATIND: GHI   RE          ;Get the X index in RE.1
        STXD              ;Push to save.  Decrement R(X)
        CALL  MULT2       ;Multiply RF.1 x RF.0 to create
                          ;row index
        IRX               ;Point to saved X index (old RE.1)
        GLO   RE          ;Get low row index from multiplication
        ADD               ;Add X index on stack
        PLO   RE          ;Put in RE.0
        GHI   RE          ;Then add possible carry to
        ADCI  $0          ;   high row index in RE.1
        PHI   RE          ;    "          "
        GLO   RE          ;Shift RE left one bit position
        SHL               ;   using double precision to
        PLO   RE          ;   multiply RE x 2.  If single
        GHI   RE          ;   byte element array desired, this
        SHLC              ;   shift should be eliminated
        PHI   RE          ;    "          "
        GHI   RD          ;Transfer base address in RD to
        PHI   RF          ;   RF to pass to ADDIT sub
        GLO   RD          ;   which will return final indexed
        PLO   RF          ;   array address in RE by adding
                          ;   RE + RF
        CALL  ADDIT       ;Add base address to index
        RETN              ;Return from subroutine
```

```
;*****************************************************************
; JMPSUB:         SCRT INDEXED SUB JUMP
;*****************************************************************
; INPUT:      --  RE.0 = sub index number 00-0F (i.e. 1 of
;                   16 possibilities)
;                 Sub addresses in SUBTAB jump sub table
;
; OUTPUT:     --  Sub # N called -- must end with RETN
;
; CHANGES:    --  RE
;
; NOTES:      --  SUBTAB may be expanded but must exist on same
;                   page as JMPSUB.  Advantage of this technique
;                   is its ROM-ability.  Returns from subroutines
;                   are from called sub to main calling routine,
;                   not back to here!  So JMPSUB must be called
;                   as a subroutine itself
;*****************************************************************

JMPSUB:   GLO   RE          ;Get sub index number from RE.0
          ANI   $0F         ;Limit to 00-0F range to avoid crash
          SHL               ;Multiply x 2.  Each entry in table
                            ;is 2 bytes.
          ADI   SUBTAB.0    ;Add base address of jump sub table
          PLO   RE          ;Put in RE.0 forming address in table
          GHI   R3          ;Get high address of this page from
                            ;the PC register
          PHI   RE          ;   and place in RE.1 to complete
                            ;   the address
          GHI   R6          ;Push the return (to main) address
          STXD              ;   onto the stack thus creating an
          GLO   R6          ;   additional return address there
          STXD              ;   pointing the way back to the caller
          LDA   RE          ;Transfer the address in SUBTAB pointed
          PHI   R6          ;   to by RE into R6 which on the next
          LDN   RE          ;   RETN will cause a false call to the
          PLO   R6          ;   desired sub
1H:       RETN              ;Go to sub via RETN - false call &
                            ;default entry
SUBTAB:   .WRD  SUB0        ;Addresses of subroutines go in here
          .WRD  SUB1        ;   Note that entries which are not
          .WRD  SUB2        ;   implemented contain the
          .WRD  SUB3        ;   address of the return instruction
          .WRD  SUB4        ;   in JMPSUB.  This will cause a
          .WRD  SUB5        ;   default execution of the RETN
          .WRD  1B          ;   two times eliminating the
          .WRD  1B          ;   possibility of a crash caused
          .WRD  SUB8        ;   by this sub's operation.
```

```
          .WRD  SUB9          ;
          .WRD  SUBA          ;Additionally, the entire table must
          .WRD  SUBB          ;   reside on the same page as
          .WRD  SUBC          ;   JMPSUB unless the code is changed
          .WRD  1B            ;   to use double precision to form the
          .WRD  SUBE          ;   indexed address.
          .WRD  SUBF
;*****************************************************************
; INSORT:        INSERTION SORT
;*****************************************************************
; INPUT:     --  RE.1 = number bytes from 2 to 255 (hex FF)
;                  RE.1 = 0 or 1 is illegal
;                RF = base address of bytes to be sorted
;
; OUTPUT:    --  RE.1 bytes sorted at RF in ascending order
;
; CHANGES:   --  RC.0  RD  RE  RF
;
; NOTES:     --  Fast, about 2 seconds for 255 bytes.  If display
;                  is memory mapped, sorting the display refresh
;                  page makes an interesting visable test.
;*****************************************************************

INSORT:   DEC   RF            ;RF addresses byte before first
          LDI   $1            ;Load starting j index into D
          PLO   RE            ;Put in RE.0 (will do for j = 2
                              ;to N bytes)
1H:       INC   RE            ;Increment j.  (= 2 on start)
          GLO   RE            ;Get j index from RE.0
          STR   R2            ;Push for adding to base address
          PLO   RC            ;Also put in RC.0, the i index
          DEC   RC            ;Decrement RC.
          GLO   RF            ;Add address in RF to the i index
          ADD                 ;   which is on the stack.  Use
          PLO   RD            ;   double precision.  Sort is a
          GHI   RF            ;   little faster if page boundaries
          ADCI  $0            ;   won't be crossed in which case
          PHI   RD            ;   double precision is not required.
          LDN   RD            ;Get byte to be inserted
          STR   R2            ;Push.  Routine "looks" for proper
                              ;location
2H:       DEC   RD            ;Search toward base address for insert
                              ;location
          LDN   RD            ;Get byte at RD
          SD                  ;Compare by subtracting from byte
                              ;on stack
          BPZ   3F            ;If ≥, then branch to insert
          LDA   RD            ;Move byte at RD up one location
          STR   RD            ;   At this point the proper location for
          DEC   RD            ;   the stacked byte has not been found
          DEC   RC            ;Decrement the i index
          GLO   RC            ;Test index in RC.0
          BNZ   2B            ;If ≠ 0, branch.  Else at end of list
          SKP                 ;Skip next instruction.  Byte goes to
                              ;M(R(D))
3H:       INC   RD            ;Byte goes "ahead" of the one found to
                              ;be less
          LDX                 ;Pop byte off stack
          STR   RD            ;Insert in list
          GLO   RE            ;Get j index from RE.0
          STR   R2            ;Push for comparing
          GHI   RE            ;Get # bytes from RE.1 (N)
          XOR                 ;Compare by exclusive OR
```

```
        BNZ   $4              ;If ≠, branch to continue sorting
        INC   RF              ;Reset RF address (optional)
        RETN                  ;Return from subroutine

;*************************************************************
; DEQUE:          DOUBLE ENDED QUEUE ROUTINES
;*************************************************************
; The following five subroutines permit any page of memory to
; serve as a double ended queue, a deque (pronounced "deck"),
; which is like a stack except that bytes may be inserted or
; removed from either end of the structure.  R7 and R8 must be
; initialized to the same page and same low address before
; calling for the first time.  RE.1 passes values to and from
; the deque and RF.1 signals if overflow or underflow resulted
; from trying to insert a byte into a full deque or remove one
; from an empty deque.  Up to 255 entries may go into the deque
; before overflow.  FRONT and REAR eventually become meaningless
; in terms of memory addresses due to the wrap around of the
; pointers at page ends.  But the deque looks something like this.
;
; Min address   FRONT/   /   /   /REAR      Max address
;                      — ENTRIES —
;
; Using only the INSR and DELF subs will cause the storage area
; to function as a simple queue or a first in first out (FIFO)
; stack.  Note that the subs are not just simple compliments of
; each other but carefully govern the action of the pointers.
; Because of the action of OVRNDR, the subs INSR, DELF, INSF,
; and DELR must be called as subroutines or underflow of the
; system stack may result.
;*************************************************************

;*************************************************************
; OVRNDR:         OVERFLOW / UNDERFLOW TEST
;*************************************************************
; INPUT:     --   R7 = Front pointer
;                 R8 = Rear pointer
;
; OUTPUT:    --   RF.1 ≠ 0 = no overflow / underflow
;                 RF.1 = 0 = underflow (on deletes) overflow
;                   (on inserts)
;                 Also, stack popped into R6, and RETN to main
;                   executed on RF.1 = 0
;
; CHANGES:   --   RF.1
;*************************************************************

OVRNDR: GLO   R7              ;Get R7.0 front pointer low address
        STR   R2              ;Push onto stack for comparing
        GLO   R8              ;Get R8.0 front pointer high address
        XOR                   ;Compare the two addresses
        PHI   RF              ;Put result of comparison in RF.1
                              ;as flag
        BNZ   1F              ;If ≠, then branch to RETN to
                              ;insert or delete
        IRX                   ;If addresses were equal, then pop
        LDXA                  ;   the main return address from the
        PLO   R6              ;   stack into R6 thus cancelling
        LDX                   ;   return to insert or delete caller
        PHI   R6              ;   and returning (with RF.1 = 0) to
                              ;main routine
1H:     RETN                  ;Return to insert / delete on
                              ;R7.0 ≠ R8.0
                              ;Return to main on R7.0 = R8.0
```

```
;*****************************************************************
; INSR:          INSERT AT REAR
;*****************************************************************
; INPUT:     --  RE.1 = byte to insert
;
; OUTPUT:    --  Byte inserted at rear of deque (@ M(R(8))
;                Abort on overflow -- deque full -- RF.1 = 0
;
; CALLS:     --  OVRNDR
;
; CHANGES:   --  R8.0
;*****************************************************************

INSR:     GLO   R8            ;Get low address rear pointer
          ADI   $1            ;Add 1 to move down to free location
          PLO   R8            ;Put back in R8.0 (auto wrap around
                              ;@ page end)
          CALL  OVRNDR        ;Test if R7.0 = R8.0 & abort on overflow
          GHI   RE            ;Get byte for entering at rear
          STR   R8            ;Store @ M(R(8))
          RETN                ;Return.  RF.1 ≠ 0 = successful insertion


;*****************************************************************
; DELF:          DELETE FROM FRONT
;*****************************************************************
; INPUT:     --  None
;
; OUTPUT:    --  RE.1 = byte at front of deque @ M(R(7))
;                Abort on underflow -- deque empty -- RF.1 = 0
;
; CALLS:     --  OVRNDR
;
; CHANGES:   --  R7.0
;*****************************************************************

DELF:     CALL  OVRNDR        ;Test if R7.0 = R8.0 & abort on
                              ;underflow

          GLO   R7            ;Get low address front pointer
          ADI   $1            ;Add 1 to move down to frontmost entry
          PLO   R7            ;Put back in R7.0 (auto wrap around
                              ;@ page end)
          LDN   R7            ;Load entry at front @ M(R(7))
          PHI   RE            ;Put in RE.1 to pass back to caller
          RETN                ;Return.  RF.1 ≠ 0 = successful deletion


;*****************************************************************
; INSF:          INSERT AT FRONT
;*****************************************************************
; INPUT:     --  RE.1 = byte to insert
;
; OUTPUT:    --  Byte inserted at front of deque (@ M(R(7))
;                Abort on overflow -- deque full -- RF.1 = 0
;
; CALLS:     --  OVRNDR
;
; CHANGES:   --  R7.0
;*****************************************************************

INSF:     GHI   RE            ;Get byte to insert from RE.1
          STR   R7            ;Store at front of deque @ M(R(7))
```

```
          GLO   R7            ;Get low address front pointer
          SMI   $1            ;Subtract 1 to move up to free location
          PLO   R7            ;Put in R7.0
          CALL  OVRNDR        ;Test if R7.0 = R8.0 & abort on overflow
          RETN                ;Return.  RF.1 ≠ 0 successful insertion

;*****************************************************************
; DELR:           DELETE FROM REAR
;*****************************************************************
; INPUT:     --   None
;
; OUTPUT:    --   RE.1 = byte at rear of deque (@ M(R(8))
;                 Abort on underflow -- deque empty -- RF.1 = 0
;
; CALLS:     --   OVRNDR
;
; CHANGES:   --   R8.0
;*****************************************************************

DELR:     CALL  OVRNDR        ;Test if R7.0 = R8.0 & abort on underflow
          LDN   R8            ;Load entry at rear @ M(R(8))
          PHI   RE            ;Put in RE.1 to pass back to caller
          GLO   R8            ;Get low address rear pointer
          SMI   $1            ;Subtract 1 to move up to rearmost entry
          PLO   R8            ;Put in R8.0
          RETN                ;Return.  RF.1 ≠ 0 = successful deletion
```

Index

5183-2
$7.95

PROGRAMMER'S GUIDE TO THE 1802 (with an Assembler for Your Machine)

Tom Swan

Here's an assembly language primer that has an assembler! Coverage includes everything from the binary number system and the fundamentals of machine language to the development of a working 1802 assembler. Simply written in nontechnical language, the text is intended for the beginner but contains information that will be appreciated by experts.

Other Books of Interest . . .

Z-80 AND 8080 ASSEMBLY LANGUAGE PROGRAMMING

Kathe Spracklen

Provides everything the applications programmer needs to know about his or her machine. Programming techniques are presented along with the instructions. Numerous diagrams and examples are provided. Exercises, with answers, are included with each chapter. #5167-0, paper, 192 pages

LOGIC ANALYZERS FOR MICROPROCESSORS

John Kneen

The most up-to-date information available on diagnostic test equipment for digital system troubleshooting. Describes the current second-generation set of logic analyzers for bus analysis problems, VLSI, digital system measurements, and much more. #0953-4, paper, 128 pages

SMALL COMPUTER SYSTEMS HANDBOOK

Sol Libes

Enables the small-computer user to purchase, assemble, and interconnect components and to program the microcomputer. Little knowledge of electronics is required to use this book. #5678-8, paper, 208 pages

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

ISBN 0-8104-5183-2