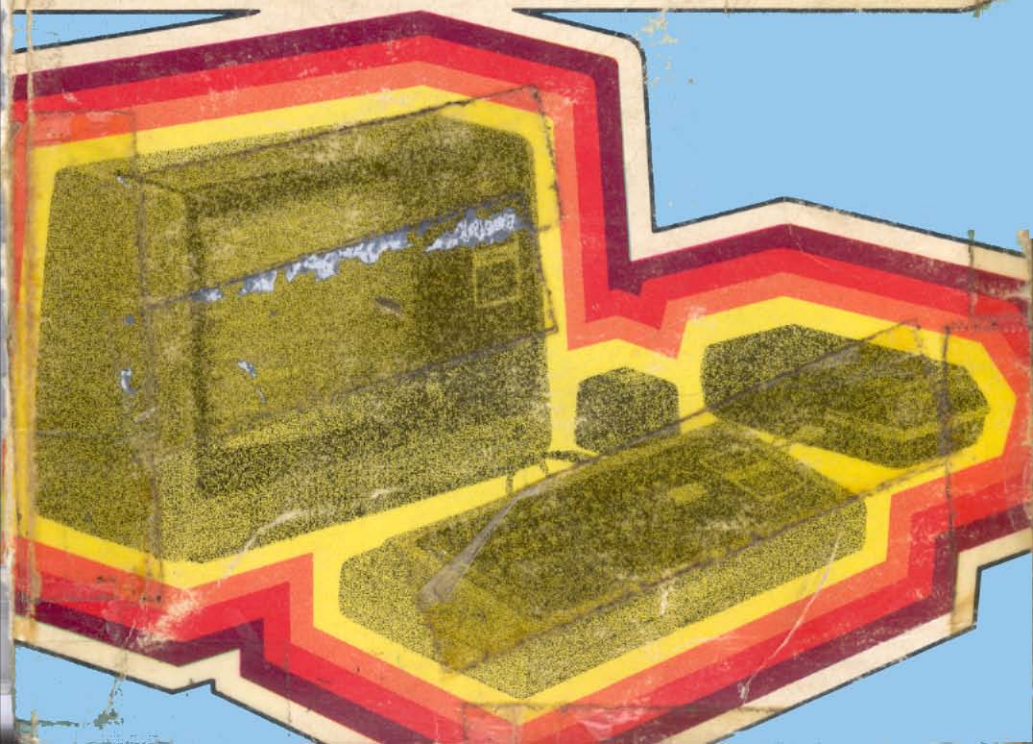
Radio Shack

THREE DOLLARS AND NINETY-FIVE CENTS

62-2006

TRS-80 Assembly-Language Programming



TRS-80 Assembly-Language Programming

by

William Barden, Jr.

Radio Shack®
A TANDY CORPORATION COMPANY

FIRST EDITION
THIRD PRINTING—1980

Library of Congress Catalog Card Number: 79-63607

Preface

Why study assembly language programming for the Radio Shack TRS-80? Why when I was a youngster all we had was Level I BASIC to work with and we did all right with that! Well, BASIC, whether it is Level I, Level II, or Disc, is still just as useful as ever. There are times, though, when the absolute fastest possible processing is called for. That is one case where assembly language reigns supreme. Programs run at assembly-language speeds are up to 300 times faster than their BASIC equivalents! Did you ever want to try your hand at the most elemental type of coding to see if you could construct a program in similar fashion to building electronic circuits from discrete components? Assembly-language will give you that challenge. How about your memory requirements? Do you find that you always require 4K bytes more than you have in RAM? Assembly language will enable you to run a program in 4K that requires 24K in BASIC. Did you ever have an urge to see what is going on in all of those routines in ROM or TRSDOS? You guessed it—assembly language again.

The goal of this book is to take a TRS-80 user familiar with some of the concepts of programming in BASIC and introduce him to TRS-80 assembly language. The text does not absolutely require a Radio Shack Editor/Assembler package, but it will help. If your system will not support an Editor/Assembler, then Radio Shack T-BUG can be used to key in all of the programs in this book without assembling—we've done that for you. We *have* designed the book to be highly interactive. There are many programs that can be assembled and loaded, or simply keyed in using T-BUG, and that illustrate the techniques of assembly-language programming as they relate to the TRS-80. We have routines to write data to the screen, to move patterns at high-speed, to graphically illustrate a bubble sort, and even a routine to play music by using the cassette output! Of course, you may also use the

book simply as a reference book for assembly-language routines. The last chapter has a dozen or so "standard" assembly-language routines that can be used in your own assembly-language coding.

Section I of this book covers the general concepts of TRS-80 assembly language. The TRS-80 uses a Z-80 microprocessor, and the architecture of both the TRS-80 and Z-80 are covered in Chapter 1. Chapter 2 talks about the instruction set of the Z-80. There are hundreds of actual instructions, but they can easily be grouped into a manageable number of types. Chapter 3 discusses the many addressing modes available for instructions in the Z-80. Assembly-language programming operations and formats are covered in Chapter 4, while Chapter 5 covers T-BUG and machine-language programming.

The second section of the book discusses various types of programming operations and provides many examples of each type. Chapter 6 shows how data is transferred within the TRS-80, between memory and central processing unit and between other parts of the system. Arithmetic and compare operations are covered in Chapter 7; this chapter describes how the Z-80 adds and subtracts, along with a description of different types of number formats. Chapter 8 gives examples of logical and bit operations and shifts, some of the most powerful instructions in the Z-80. Chapter 9 describes how assembly-language programs perform string manipulations and process data in tables. Chapter 10 talks about input/output operations, one of the most mysterious (unjustifiably so) areas of computer programming. The last chapter contains the previously mentioned common subroutines.

Two appendices provide a cross-reference of Z-80 operation codes and instruction set. Appendix I lists the Z-80 instruction set by function (add, subtract, etc.) while Appendix II provides a detailed alphabetized listing of all instructions.

If you suspect that assembly-language might be for you, then by all means give it a try. You have nothing to lose but your GOSUBs (and other BASIC statements). The author hopes that you have as much fun in sampling the programs in this book as he did in constructing them.

WILLIAM BARDEN, JR.

To Marguerite

Contents

Section I. General Concepts

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 10

CHAPTER 11

Section III. Appendices

APPENDIX I

APPENDIX II

SECTION I

General Concepts

CHAPTER 1

TRS-80 and Z-80 Architecture

This chapter will discuss the architecture of the TRS-80, with special consideration to the Z-80 microprocessor contained within the TRS-80. What is a microprocessor? What is a Z-80? Why do I need to know about it to program in assembly language? Why are we asking so many hypothetical questions? These and other questions will be answered in this chapter as we attempt to unravel the mysteries of the *architecture* or general functional blocks of the TRS-80 system. Stay tuned to this text. . . .

Functional Blocks

All computer systems are made up of three rather distinct parts shown in Figure 1-1. The *cpu*, or central processing unit, is the chief controller of the computer system. It fetches and executes instructions, does arithmetic calculations, moves data between the other parts of the system, and in general, controls all sequencing and timing of the system. The *memory* of the system holds a computer program or programs and various types of data. The *I/O*, or input/output devices of the system, allow a user to talk to the computer system in a manner in which he is familiar, such as a typewriter-style keyboard or display of characters on a crt screen.

As a TRS-80 user, you're undoubtedly familiar with these component parts. You have a nodding acquaintance with RAM memory from upgrading your system to 16K and perhaps more than just a casual relationship with an expansion interface and disc. To enable us to do assembly-language program-

ming properly, however, we are going to have to get more familiar with memory and I/O and (much to the dismay of our spouses, who are already computer widows or widowers) rather *intimately* involved with the cpu portion of the TRS-80 system. In addition, in later chapters, we're going to leave an old friend, BASIC, and strike up a relationship with assembly-language principles.

What Are All These Ones and Zeros?

Up to this point in your programming career, you have probably used decimal values for such things as constants,

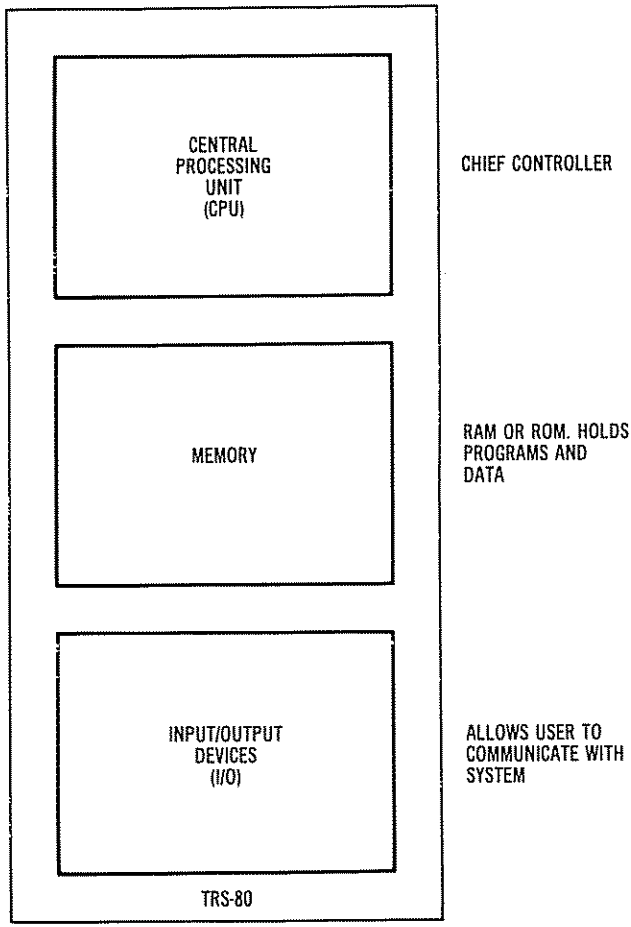


Fig. 1-1. Functional blocks of the TRS-80.

memory addresses, and POKEs. Assembly-language programming makes extensive use of *binary* data and *hexadecimal* data. Don't let these terms frighten you. They're really more simple than decimal data. Binary representation is a way of expressing numeric values using the binary digits of 0 and 1, rather than the decimal digits of 0 through 9. Binary digits represent an "on" or "off" condition. A wall switch is either on or off. An indicator light is either lighted or unlighted. In similar fashion, the transistors within the cpu portion of the TRS-80 are either on or off and hold binary values.

Now we know that in a decimal number such as 921 the 9 represents 9 hundreds, the 2 represents 2 tens, and the 1 represents 1 units, as shown in Figure 1-2. In a binary number,

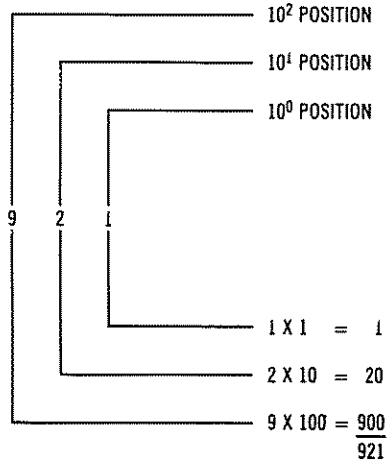


Fig. 1-2. Decimal notation.

the position of the digits represent powers of *two* rather than powers of *ten*. Instead of units, tens, hundreds, and other powers of ten, a binary number is made up of digits representing units, two, four, eight, sixteen, and other powers of two, as shown in Figure 1-3. Since there are only two binary digits, the digit at each position represents either 0 or 1 times the power of two for that position.

If the binary number is treated as groups of four binary digits, the binary number can be converted into a *hexadecimal* number. Hexadecimal means nothing more than powers of sixteen. The groups of four bits represent 0000 through 1111. Now, 0000 through 1001 correspond to the decimal digits 0 through 9, and the hexadecimal digits for 0000 through 1001 are similarly designated 0 through 9. This leaves the groups

of bits from 1010 through 1111. When the hexadecimal system was first proposed, one of the more obscure computer scientists proposed that the remaining six groups be designated actinium, barium, curium, dysprosium, erbium, and fernium. Cooler heads prevailed, however, and the digits were named A, B, C, D, E, and F.

In general we'll be working with groups of eight binary digits or sixteen binary digits within the TRS-80. *Binary digit* was long ago shortened to *bit* to prompt shorter lunches in the computer science cafeteria when researchers started talking shop. Whenever bit is used, then, it will mean one binary digit of either a 1 or 0. A group of four bits may be referred

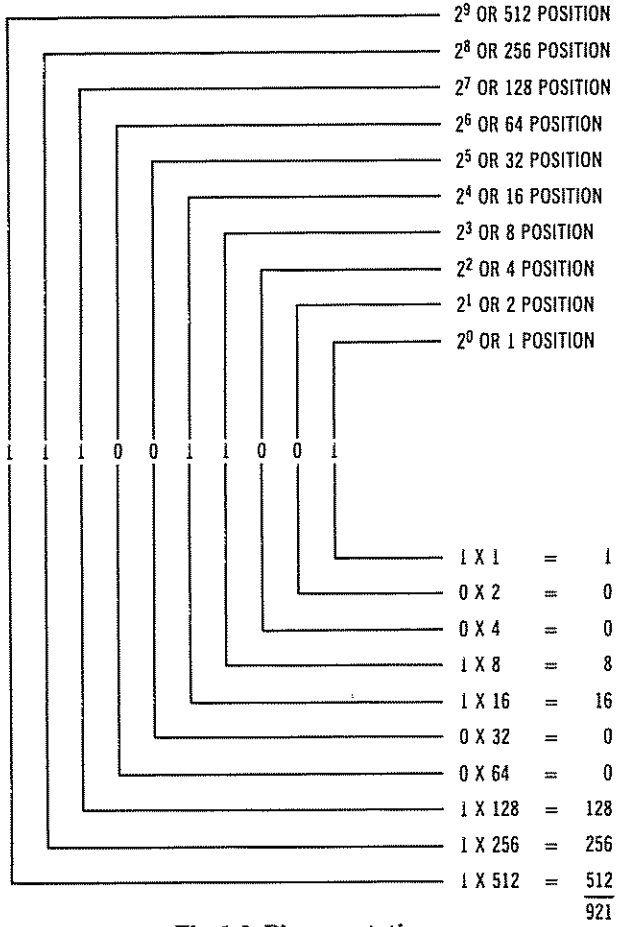


Fig. 1-3. Binary notation.

to as a hexadecimal digit of 0 through F. When this is done, the suffix *H* is added. The symbol EH, therefore, represents the hexadecimal digit E or the binary digits 1110. A group of eight bits is commonly called a *byte*. A byte is made up of two hexadecimal digits, since there are two groups of four bits.

Don't be too worried about the use of bits, bytes, and hexadecimal digits at this point. We'll reiterate some of these basic points as we go along in the text.

CPU, Memory, and I/O

Generally, all elements of the TRS-80 work with binary data. Each memory location, for example, is made up of eight bits, and can represent values from 00000000 through 11111111, or zero through 255 decimal. I/O devices such as cassette tape or floppy disc communicate with the cpu by transferring 8-bit bytes and converting between bytes of data and *bit streams*. The cpu is similarly a binary *digital* device, holding all data or control signals as discrete bits of information.

Let's talk a little bit (no pun intended) about the cpu. As we mentioned before, the cpu is primarily concerned with fetching and executing instructions. What are the types of instructions that the cpu can perform? Obviously, it would be very difficult to implement an instruction such as "if this is Friday blink the screen cursor on and off at location 512." It would be *possible* to implement this instruction, but as you might guess, it would be much more practical to implement a basic set of general-purpose instructions such as "add two numbers" or "compare the result with 67." As a matter of fact the *instruction set* of the TRS-80 at this cpu level is very similar to the instruction set of other microcomputers and the instruction sets of even very large computers. The instruction set of the TRS-80 allows for adding two *operands*, subtracting two operands, performing logical operations on two operands (such as AND or OR), transferring 8 or 16 bits of data between the cpu and memory or I/O devices, jumping to another portion of the program (similar to GOTO or IF . . . THEN), jumping to and returning from subroutines, and testing and manipulating bits.

Every application, including the Level I and II BASIC programs in ROM, and extending to such applications as high-speed video games and business payroll is made up of sequences of these rudimentary instructions such as adds, com-

pares, and jumps. As a matter of fact, *every* program, even those written in BASIC, ultimately resolves down to a sequence of these basic cpu instructions.

In older computer systems each of the component parts literally occupied rooms. Today, almost the entire logic of the cpu can be put on a single *microprocessor chip* about the size of a postage stamp. The microprocessor chosen for the TRS-80 was the Z-80, originally designed by Zilog, Inc. The Z-80 is a state-of-the-art (an engineering way of saying "modern") microprocessor with a good instruction set. Since the cpu portion of a microcomputer is now essentially its microprocessor we'll look in detail at the Z-80 architecture in this chapter, and at its instruction set in later chapters.

Memory within the TRS-80 system is made up of ROM, RAM, and *dedicated* memory addresses. We're all familiar with RAM memory. That's the memory that holds our programs and data, whether they are BASIC programs or SYSTEM types (assembly language). The minimum amount of RAM we can have is 4K, or 4096 bytes, and the maximum amount we can have is 48K, or 49152 bytes, for a system with an expansion interface. The term RAM stands for *R*andom-*A*ccess-*M*emory and simply means a memory that we can both read from and write into. ROM memory, on the other hand, is *R*ead-*O*nly *M*emory. ROM in the TRS-80 holds the Level I or Level II BASIC interpreter, and occupies 12288 bytes in the Level II case. Try as we might, we can't POKE into the ROM memory area. Each of the 61,440 locations of ROM and RAM can hold one byte, or 8 bits, of data. Each of these 61,440 locations is assigned a location number. ROM is assigned locations 0 through 12287, and RAM is assigned locations 16384 through 65535.

Yes? A question from the back of the room? The gentleman asks what locations 12288 through 16383 are used for? (These TRS-80 owners—you can't put anything over on them . . .) Locations 12288 through 16383 are not used for memory addresses in the conventional sense. These are dedicated locations that the cpu uses to address such things as the line printer, floppy disc, real-time clock and video screen. It turns out that the video display is indeed a RAM memory, but the remaining devices are only decoded as memory locations. We'll explain further in later chapters. Figure 1-4 shows the *memory mapping* for the TRS-80.

It's important to know that data in memory can be either an instruction for the cpu or data, such as a character for display. I see the same wise guy has his hand up! The cpu *doesn't*

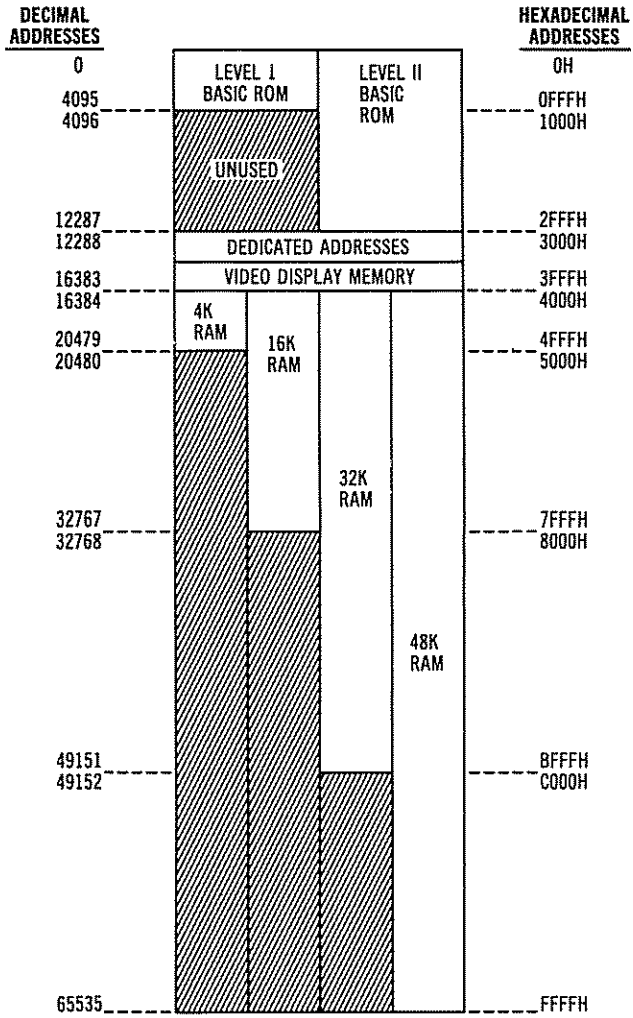


Fig. 1-4. TRS-80 memory mapping.

know which locations hold data and which hold instructions. The cpu blindly goes ahead and if a data byte is picked up instead of an instruction, it will attempt to execute the data as an instruction. The result will probably be catastrophic, and is a program *bug* (you're certainly familiar with bugs from your BASIC programs—in assembly language they are even more prolific). Data and programs are therefore intermixed in memory at the programmer's discretion (or indiscretion) and the program should know how to *jump* around the data.

I/O devices may be considered in two parts. Firstly, there is the physical I/O device, such as the cassette recorder, video display, keyboard, line printer, or floppy disc. Secondly, there is the *I/O device controller*. The I/O device controller performs an interfacing function between the cpu (microprocessor chip) and the I/O device. The controller matches the high rate of data transfers from the cpu (hundreds of thousands of bytes per second) to the I/O device (50 bytes per second for Level II tape cassette). The controller may also encode the data coming from the cpu into special format (video format for the display, for example) and provide a *handshake* function between the cpu and I/O device. (How are you, my name is Bernie. Do you have the next data byte for me?) The I/O device itself may be a device adapted to microcomputer use such as the cassette recorder or video display or one specifically made for a microcomputer environment, such as the line printer or floppy disc.

The Z-80: A Chip Off the Old Block

Now that we have an overview of the TRS-80, let's look at the internal workings of the Z-80, or at least those parts that

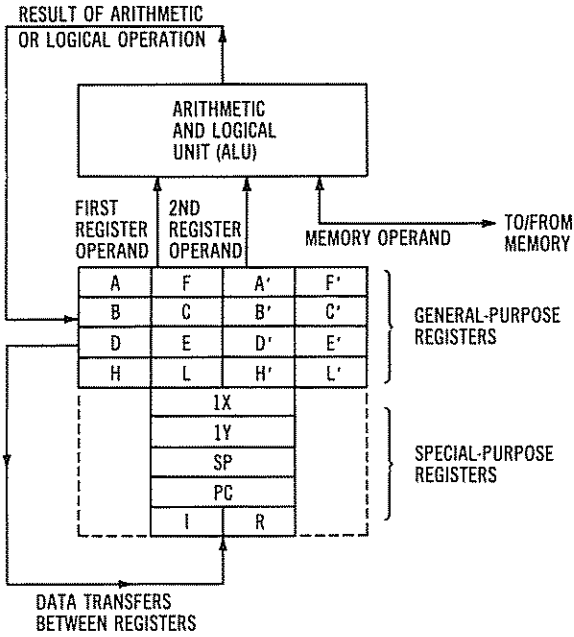


Fig. 1-5. Z-80 architecture.

we, as assembly language programmers, will want to be aware of. Figure 1-5 shows the cpu *register* arrangement, the ALU, or *a*rithmetic and *l*ogic *u*nit, and the data paths we should be concerned about.

In general, all data in the TRS-80 and most data within the Z-80 is handled in 8-bit, or one-byte segments. The Z-80 is called an "8-bit" microprocessor for this reason. The cpu (Z-80) registers are either 8 bits or 16 bits wide, and most manipulations within the cpu are done 8 bits at a time.

There are 14 general-purpose registers within the cpu, designated A, B, C, D, E, H, and L and the "primed" counterparts A′, B′, C′, D′, E′, H′, and L′. Many of the arithmetic and other instructions use the A register *contents* as one of the operands, with the other operand coming from memory or another register. For this reason, the "A" register can be thought of as the "accumulator" register, which is an old term that is still used today. In addition to being used separately as 8-bit registers, there are several sets of register "pairs" that form 16-bit registers when the 8-bit registers are used together. These are B/C, D/E, H/L, B′/C′, D′/′E, and H′/L′. The register pairs are used to perform limited 16-bit arithmetic, such as adding two 16-bit operands contained in two register pairs, or to specify a memory address.

At any time only one set of the registers, prime or non-prime, are active. Two Z-80 instructions select the current inactive set (prime) to become active and put the currently active (non-prime) into an inactive state. The instructions, therefore, are used to switch between the two sets as desired. A second set does not *have* to be used, but simply makes more register storage available if required.

The cpu registers are used to store temporary results, to hold data being transferred to memory or I/O, and in general to hold data that is being used for the current portion of the program that is being executed. Data changes within the cpu registers very rapidly as the program is being executed (tens of thousands of times per second) so the cpu registers may be thought of as a conveniently used, rapidly accessed, limited memory within the cpu itself that holds transient data.

In addition to the general-purpose registers within the cpu, there are special-purpose registers. The first of these is the PC, or *P*rogram *C*ounter. The PC is a 16-bit register that points to the current memory location holding the instruction to be executed. We mentioned previously that there were 65536 memory locations that could be used on the TRS-80. A 16-bit register may hold a range of values from 0000000000000000

through 1111111111111111, or decimal 0 through 65536 (hexadecimal 0000H through FFFFH). The PC can therefore *address* (point to) any memory location for the current instruction. Instructions to the cpu are *coded* into one, two, three, or four bytes and are generally arranged sequentially in memory, starting from "low" memory to "high" memory. Figure 1-6 shows a typical sequence of instructions. As each new instruction is "fetched" the PC is *updated* by adding the number of bytes in the instruction to the contents of the PC. The result points to the next instruction in sequence. When a "jump" is executed, the new memory location for the jump is forced into the PC, and replaces the previous value, so that the new instruction from a new segment of the program is accessed. If one could look at the PC in the TRS-80 as a program was running, the PC would be changing hundreds of thousands of times a second as sequences of instructions were executed and jumps were made to new sequences.

MEMORY LOCATION OF INSTRUCTION	CONTENTS	INSTRUCTION	PROGRAM COUNTER BEFORE EXECUTION
4A00H	06H	LD B,0	4A00H
4A01H	00H		
4A02H	B7H	OR A	4A02H
4A03H	EDH	SBC HL,DE	4A03H
4A04H	52H		
4A05H	FAH	JP M,DONE	4A05H
4A06H	0CH		
4A07H	4AH		
4A08H	04H	INC B	4A08H
4A09H	C3H	JP LOOP	4A09H
4A0AH	02H		
4A0BH	4AH		
4A0CH	19H	ADD HL,DE	4A0CH

Fig. 1-6. Typical sequence of instructions.

The SP, or *S*tack *P*ointer, is another 16-bit register that addresses memory (Figure 1-7). In this case, however, the SP addresses a memory *stack* area. The memory stack area is simply a portion of RAM used by the program as temporary storage of data and addresses of subroutines during subroutine calls. As the SP is 16 bits, any area of memory could conceivably be used, as long as it was RAM and not ROM. In practice, high areas of RAM memory are used, as the stack *builds down* from high memory to low memory. In a 16K RAM system, for example, the stack might start at 32767 (don't forget about that initial 16384 ROM and dedicated

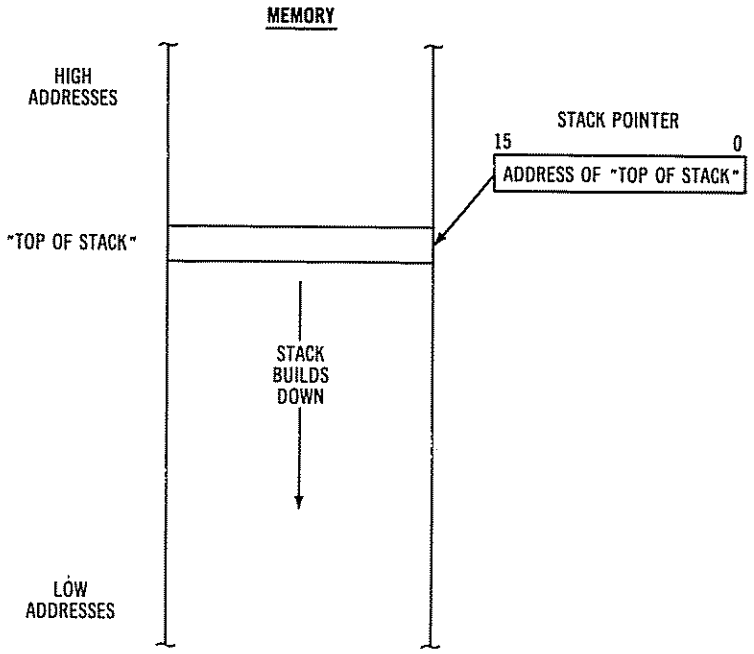


Fig. 1-7. Memory stack.

memory area) and build downward. Well, it appears that the programmer in the back wants an explanation of the stack action. We'll give a brief one here and give a more detailed one in a later chapter. The stack is a *LIFO* stack, which stands for "last-in-first-out." A good analogy is a dinner plate stacker found at some restaurants. The last dinner plate put on the stack is the first taken off. As more and more plates are put on the stack, the stack increases in size. If the reader can visualize data being put on the stack in this fashion, it will be somewhat similar to Z-80 stack action.

Two additional cpu registers, IX and IY, are used to modify the address in an instruction. This permits *indexing* operations which allow rapid access of data in tables. Indexing operations and the use of IX and IY will be discussed in detail in Chapter 3.

The I and R registers are two registers that the reader probably will not use in his TRS-80 system. The R register is continually used by TRS-80 *hardware* to *r*efresh the dynamic RAM memories used in the TRS-80 system. The 8-bit value in the R register is continually incremented by one to cycle the register from 0000000 through 1111111 and around

again to provide a *refresh count* for *dynamic memory refresh*, which restores the data in RAM. The 8-bit I register is used for a mode of interrupts not currently implemented in TRS-80 hardware.

There are other cpu registers, of course, but the foregoing registers are the only ones that are accessible by an assembly-language program. The other registers in the Z-80 cpu hold the instruction after it is fetched, buffer data as it is moved internally and transferred externally, and perform other actions required for instruction interpretation, instruction implementation, and system control.

The arithmetic and logic unit is the portion of the cpu that, as the name implies, performs the addition, subtraction, ANDing, ORing, exclusive ORing, and *shifting* of data from two operands. The result of these operations generally goes to a cpu register, although it may also go to memory in some cases. A set of *flags* are set on the results of the arithmetic or logical operation. For example, it is convenient to know when the result is zero after a subtract operation. A *zero flag* is set if this is the case. There are eight flag bits that are treated together as a cpu register, even though they are not used in the same fashion. The flag registers are called F and F'. When used in register pair operations the A and F or A' and F' registers would be grouped together. The flags will be further discussed in this section and in chapters dealing with specific sets of instructions. For the time being Table 1-1 shows the names and functions of the flags.

Table 1-1. CPU Flags

Name	Function
Sign(S)	Holds the sign of the result. 0 if positive, 1 if negative
Zero(Z)	Holds the zero status of the result 1 if zero, 0 if non-zero
Half-carry(H)	Holds the half-carry status of the result, 0 if no half-carry, 1 if half-carry. Not generally accessible by program.
Parity/ Overflow (P/V)	Holds the parity of the result or the overflow condition. If used as parity, P = 0 if the number of one bits in the result is odd, or P = 1 if the number is even. If used as overflow flag, V = 0 if no overflow or V = 1 if overflow.
Add/subtract(N)	Add or subtract condition for decimal instructions. Add = 0, subtract = 1. Not generally accessible by program.
Carry(C)	Holds the carry status of the result, 0 if no carry, 1 if carry

Data flow between the cpu and remaining TRS-80 system is shown in Figure 1-5. Almost all data within the system uses the cpu. As a program is being executed, the instruction bytes making up the program are continually being fetched from RAM memory and placed into the cpu instruction decoding logic. If an instruction is four bytes long, four separate memory fetches are made to RAM memory, with the PC pointing to each sequential byte in turn. Once the instruction is decoded, additional memory accesses may have to be made to get the operand to be used in the instruction. The instruction to add the contents of the A register and location 16400 (4010H) calls for the cpu to not only fetch the instruction, but to fetch the value found at location 16400 to be added to the value found in the A register. Similarly, the results of operations may be *stored* back into memory. In addition to transferring instruction bytes and operand data between itself and memory, the cpu also communicates with I/O devices such as the line printer and cassette. The cassette in Level II BASIC operates at 50 bytes per second. Each byte on a write (CSAVE) is held in a cpu register and written to the cassette interface logic one *bit* at a time. When a print operation on the system line printer is done (LPRINT), a byte of *status* from the line printer is read into a cpu register and checked. If the status indicates the line printer is *ready* to receive the next byte, the byte representing character data is transferred from a cpu register to the line printer. Note that in both the cassette and line printer cases the data may have been initially contained in a *buffer* in memory as a cassette program or print line, but that it is transferred from memory to the cpu register and from the cpu register to the I/O device a byte at a time. Although it is possible to bypass the cpu and transfer data between the I/O device and memory using a Z-80 technique called *d*irect-*m*emory-*a*ccess, or *DMA*, the TRS-80 does not currently use this method and we will not be describing it in this text.

In this chapter we've looked at the architecture of the TRS-80 and especially at the internal architecture of the Z-80 microprocessor used in the TRS-80. In the next two chapters we'll investigate two more topics closely associated with the Z-80, the Z-80 instruction set, and Z-80 addressing modes. After that we'll call a halt to theoretical discussions and get our hands dirty (figuratively, anyway, unless you code with a leaky pen) in learning how to use the assembler, editor, and T-BUG.

CHAPTER 2

Z-80 Instructions

In this chapter we will discuss the instruction set of the TRS-80 system. The instruction set of the TRS-80 at the assembly-language level is really the instruction set of the Z-80 microprocessor in the TRS-80 as we pointed out in the last chapter. If you have looked at the numeric list of the instruction set in the Radio Shack Editor/Assembler Manual (26-2002), you may have been one of the recent wave of trauma victims that have suddenly appeared all over the country. There are *many* different combinations of instructions! (There are well over five hundred, as a matter of fact!) This chapter, among other things, will attempt to prove that this massive, confusing list can be reduced to a tolerable number of basic instructions. It will take some effort to learn about the various instruction types, and a little more effort to learn about the addressing modes covered in the next chapter, but refuse to be intimidated! There are hundreds of thousands of assembly-language programmers in the country and there is no reason you cannot be another.

The Z-80 Family Tree

One of the things that we might mention in passing concerns the heritage of the Z-80 microprocessor. At many places in the discussion of the instruction set in this book, the reader may be prompted to say, "Why the devil did they do that?"

One of the reasons that there are many different ways of doing the same thing (say adding two operands) is related to the predecessor of the Z-80, the 8080A, and its predecessor, the 8008. The 8008 is the grandfather of the Z-80. The 8008 grew up in the early days of microcomputing, back in the early '70s (this century). The 8008 was the first microprocessor on a chip and had an instruction set of 58 instructions. Shortly after the 8008 was introduced, another microprocessor, the 8080, was developed. The 8080 was a faster, more powerful microprocessor than the 8008, and had an instruction set of 78 instructions. Recently, a third generation of microprocessor was developed—the Z-80. To compete in the hectic microprocessor marketplace, the 8080 included the 8008 in-

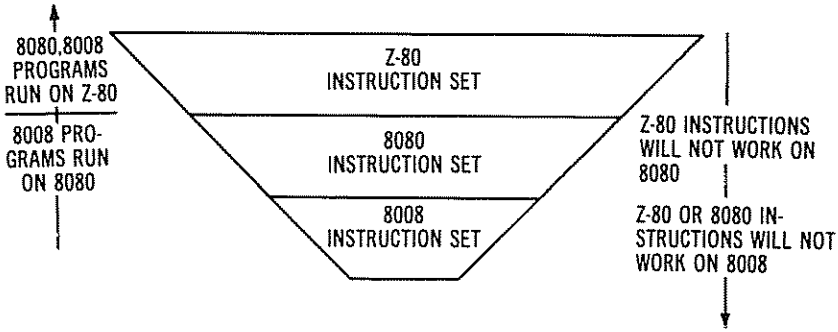


Fig. 2-1. The Z-80 family tree.

structions in its *repertoire*, and the Z-80 includes the 8080 instructions in *its* repertoire. The reason for this *downwards compatibility* is that existing programs can be executed on the newer generations of microprocessors, saving costs on software development. The situation for the instruction set of the Z-80 is shown in Figure 2-1. All programs written for both the 8008 and 8080 can be executed on the Z-80, assuming, of course, that the limitations of the system are equal (such as the same I/O device addresses, memory layout, and so forth).

In carrying through the instruction set of the 8008 and 8080, the Z-80 instruction set duplicates the architecture and general approach of its two predecessors, but adds many new instructions of its own. If the reader sees many ways of doing the same thing in future chapters, therefore, it is probably

related to the father's approach, or even the grandfather's. Which approach is best, the experience of age, or the innovation of youth? As in life, some of each.

How Long Is an Instruction?

The answer to this, of course, is "long enough to reach the memory." Z-80 instructions are, in fact, one to four bytes long with the average being about two bytes. This means that in 4096 bytes of memory we can hold about 2000 assembly-language instructions. This is quite a contrast to BASIC programs where each BASIC line probably takes up 40 characters or so, allowing perhaps 100 BASIC lines. (Each assembly-language instruction, of course, does much less than a BASIC instruction, but does it much faster.) Many of the older 8080-type instructions are one byte long, while the newer Z-80 type instructions are four bytes long. The assembler program automatically calculates the length of the instruction during the assembly process, so you do not need to be concerned with remembering instruction lengths.

```
4A00            00100          ORG     4A00H          ;START AT LOCATION 4A00H
4A00 3E31       00110          LD      A,31H          ;LOAD A REGISTER WITH "1"
4A02 32203E     00120          LD      (3E20H),A      ;STORE "1" INTO CENTER
4A05 C3054A     00130 LOOP     JP      LOOP           ;LOOP HERE
4A08            00140          END     4A00H          ;END-START OF 4A00H
00000 TOTAL ERRORS
LOOP   4A05
```

Fig. 2-2. Typical assembly-language listing.

To give the reader some feel for instruction lengths in a typical program we will look at a typical assembly-language *listing*, shown in Figure 2-2. The listing is the output display or printed output of the assembler portion of the Editor/Assembler after the assembly process.

The first column of the listing represents the location in RAM where the program is *to be* stored. The value "4A00," for example, indicates that the "LD A,31H" instruction will be put into memory locations 4A00H (18944 decimal) and 4A01H (it is a two-byte instruction). The next column is the machine-language code of the instruction itself. For the "LD

A,31H" this amounts to two bytes (16 bits or four hexadecimal digits). The "3E31" are the four hexadecimal digits representing the code. The next column is a line number for the assembly, which is identical to the BASIC line numbers with which you are familiar. The remaining columns represent the assembly-language line for the instruction code; the first column is a *label*, the next is the *operation code* (a shorthand representation of the instruction), the third is an *operand* (in this case 31H (49 decimal). This format will be discussed in detail in Chapter 4, so do not concern yourself with it at this point. *Do* note the second column, however, and observe how the instruction lengths vary from one to four bytes; each two hexadecimal digits are one byte.

Wait a Microsecond . . .

Another interesting attribute that we should discuss is instruction speed. Generally, the longer the instruction, the longer it takes. The reason for this is that for each byte of the instruction one *memory access* must be made. This amounts to the cpu transferring one byte of instruction data into an internal register for decoding. To make one memory access in the TRS-80 takes about .45 microsecond, or about ½ millionth of a second. Add to this time some additional overhead for executing the instruction and for obtaining operands from memory, and we find that TRS-80 instructions range from 2.3 microseconds to 13 microseconds, with the average being somewhere around 5 microseconds. To contrast assembly-language code with BASIC coding, consider this BASIC program

```
100 FOR I = 0  TO 255
200 NEXT I
```

The short loop above takes approximately ⅔ second to execute in BASIC. A corresponding assembly-language program

```
100 LOOP DEC  C       ;DECREMENT COUNT
200      JP   Z,LOOP  ;JUMP IF NOT ZERO
```

would take about 2 milliseconds, or two thousandths of a second, approximately 350 times as fast!

The extremely fast speed of assembly-language programs (when compared to higher-level languages such as BASIC) makes this type of programming excellent for such applications as *real-time* game simulations, fast business sorts, or

any task that might take prohibitive amounts of time with other methods.

Instruction Groups

Now that we've discussed some of the attributes of instructions, let's look at how we might whittle down that set of Z-80 instructions from sawmill size into at least a cord of wood. We'll do this by dividing the instruction set into six different groups:

Data Movement
Arithmetic, Logical, and Compare
Decision Making and Jumps
Stack Operations
Shifting and Bit Operations
I/O Operations

Data Movement: Loads, Stores, and Transfers

Much of the time in any program, whether it is BASIC or assembly language, is spent moving data from one place to another. In assembly-language programs the cpu registers are used for very temporary storage while RAM memory is used for data that may be somewhat less volatile. If one looks at the TRS-80 system components of cpu, memory, and I/O devices, one can say that data in the cpu is transient, data in memory is active for program usage, and data stored on audio cassette tape or floppy disc is most permanent. In any event, data is constantly being moved from cpu registers to other cpu registers, from cpu registers to memory, from memory to cpu registers, and from one memory area to another memory area.

The general term for moving data from memory to a cpu register is "load." Data is said to be *loaded* into a cpu register. Remember, now, that the data we are talking about is *operand* data rather than the data associated with the instruction operation itself. The data associated with the instruction itself is automatically brought into the cpu instruction decoding logic in the course of normal program execution as the PC (program counter) points to each instruction in turn. An example of this difference would be the instruction "LD B,101" which loads the value of 101 decimal into the cpu B register.

In many other microprocessors, the action of transferring data from a cpu register to memory is called a "store." In the

Z-80 microprocessor of the TRS-80, however, the term *load* is used to apply not only to transferring data from memory to a cpu register, but also in transferring data from a cpu register to memory. The instruction *mnemonic*, or shorthand symbol, for a load operation is "LD." Any time you see an "LD" on an assembly listing you know that data is being moved between cpu registers or between cpu registers and memory. The instruction

```
LD  A,B     ;LOAD A WITH B
```

for example, takes the contents of cpu register B and puts it into cpu register A, leaving the B register *unchanged*. This last point is an important one: All loads *copy* data, rather than transferring it. The *source* of the data remains unchanged, whether it is in memory or a cpu register.

Another example of a load is the instruction

```
LD (4234H),A     ;STORE A REGISTER INTO LOC 4234H
```

which takes the contents of the cpu A register and copies it into RAM memory location 4234H (16948 decimal).

We mentioned in Chapter 1 that the general-purpose cpu registers were 8 bits wide, but that sometimes they were grouped as *register pairs* of 16 bits. To refresh your memory (no pun intended), the register pairs were combinations of cpu registers B and C, D and E, and H and L. The load instructions give us the ability to move data one byte or two bytes at a time using single registers or register pairs.

When data is moved one byte at a time, the eight bits of the *source operand* are copied into the *destination* register or memory location. Of course the bits are copied with the same orientation. *LoaDing* the H register with 01100001 from the L register produces 01100001 in the H register, and not another arrangement of bits.

When data is moved two bytes at a time, the 16 bits are copied from one register pair to another, or between a register pair and *two* memory locations. Let's see how this works. Suppose that in register pair H,L we have the decimal value 1000. Now if we convert decimal 1000 into binary we have

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	
0	0	0	0	0	0	1	1	H (MOST SIGNIFICANT)
1	1	1	0	1	0	0	0	L (LEAST SIGNIFICANT)

Fig. 2-3. Register pair data arrangements.

0000 0011 1110 1000, after we have added the necessary leading zeros to make up 16 bits. Figure 2-3 shows how that value is arranged in the H and L registers. The upper 8 bits (one byte) is in H and the lower 8 bits is in L. The same arrangement holds true for B and C and D and E. B and D are always the upper, or *most significant*, registers, while C and E are always the lower or *least significant* registers.

That's easy enough to remember if you think of BC, DE, and HL and remember H(igh) and L(ow). And to answer that same heckler from the back of the room, yes, this was the reason for the 8008 designation of "H" and "L." But what happens in memory when a register pair is stored? When a register pair such as H and L are stored by the instruction

```
LD   (4A0AH),HL  ;STORE H AND L INTO 18954
```

the low or least significant register, in this case L, is stored in the memory location specified, in this case 4A0AH. The high, or most significant register, in this case H, is stored in the *next* memory location, in this case 4A0BH (18955). *This arrangement of low order byte followed by high order byte holds true for all types of data within a Z-80 assembly-language program.* As one would expect, data loaded from memory into a cpu register pair restores the register pair in the same fashion. Figure 2-4 shows the store described above.

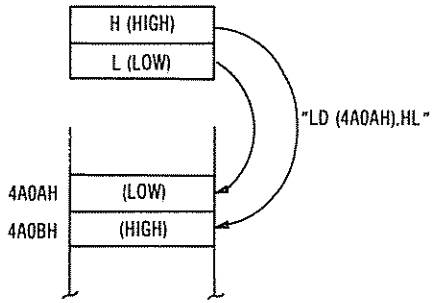


Fig. 2-4. Memory arrangement for 16-bit data.

A group of load instructions called the *block moves* enables from one to 65536 bytes to be moved in a single instruction, or in a very few instructions. These load instructions avoid the overhead of moving data in a long sequence of instructions and are a powerful feature of the Z-80. The four block moves will be discussed in detail in Chapter 6.

We won't attempt to list all of the possible loads in this section. Many of them are dependent upon the *addressing*

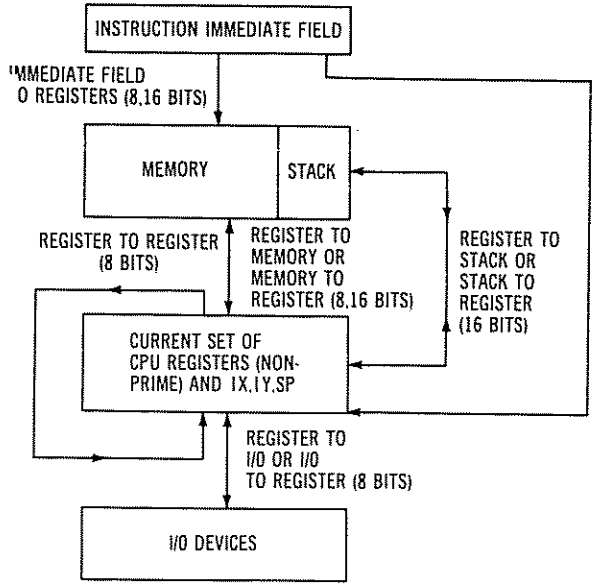


Fig. 2-5. Data transfer paths.

mode used in the instruction, which will be covered in Chapter 3. What we will do, however, is to illustrate the ways in which 8- or 16-bit data can be transferred from one part of the system to another by the use of LD instructions. Figure 2-5 shows the paths and indicates the types of instructions available in the Z-80 to perform the transfers.

Arithmetic, Logical, and Compare

The worst part in understanding this group is the pronunciation of "arithmetic." Contrary to what you learned in P.S. 49, the adjective is pronounced so that the last two syllables rhyme with the last two of "charismatic." Novice programmers have been dismissed on the spot for the use of the common pronunciation! This group includes instructions that add and subtract two operands, instructions that perform logical operations of ANDing, ORing, and exclusive ORing, and compare instructions, which are essentially subtracts.

The most common type of arithmetic is the simple ADD instruction. Suppose that we have two 8-bit operands (two one-byte operands) in cpu registers A and B, as shown in Figure 2-6. When the instruction

```
ADD   A,B      ;ADD REGISTER B TO REGISTER A
```

is executed in the program, the contents of register B (the *source* register) will be added to the contents of register A (the destination register) and the result will be put into the A register with the B register unchanged. All *8-bit* arithmetic and logical instructions operate in the same fashion; the result always goes to the A register, and one of the operands must have originally been in the A register. The instruction

```
SUB  (HL)    ;SUBTRACT LOCATION 4400H(HL) FROM A
```

takes the contents of location 4400H, subtracts it from the contents of the A register, and puts the result into the A register, leaving the contents of location 4400H unchanged.

When an arithmetic instruction such as an add or subtract is executed, the *flags* are set on the results of the instruction. If the result of the subtract were zero, for example, the "Z" flag would be set to a 1; if the result were non-zero, the Z flag would contain a 0. A decision could then be made by a jump-type instruction later in the program that would test the state of the zero and other flags. The flags will be further discussed in the appropriate material for the instruction group. Except for the two special adds and subtracts that add in the *carry* flag, that's about all there is to the 8-bit arithmetic group. As with the loads, there are many varieties of addressing modes that may be used, and these are discussed in the next chapter.

The logical instructions in this group work in similar fashion to the arithmetic instructions. An 8-bit operand from

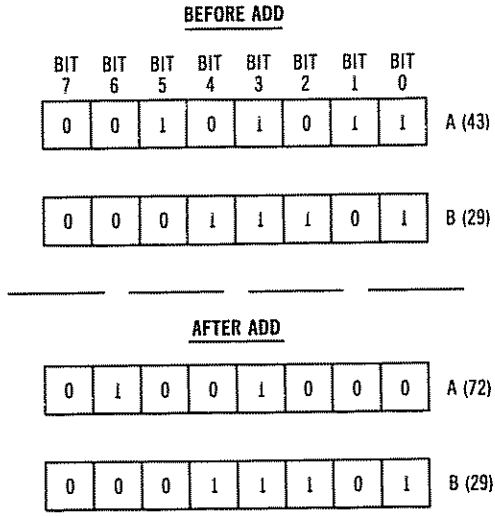


Fig. 2-6. Sample ADD operation.

memory or another register is used in conjunction with the contents of the A register. The result is put into the A register and appropriate flags are set. The functions that may be performed are ANDing, ORing, and exclusive ORing. You may be familiar with these functions from BASIC. When two bits are ANDed, the result bit is a one only if both operand bits are a one. When two bits are ORed, the result is a one if either bit or both bits are ones. When two bits are exclusive ORed, the result bit is a one if and only if one or the other bit is a one, but not both. For 8-bit operands, each *bit position* is considered one at a time, as shown in Figure 2-7. Here again there are many addressing modes possible.

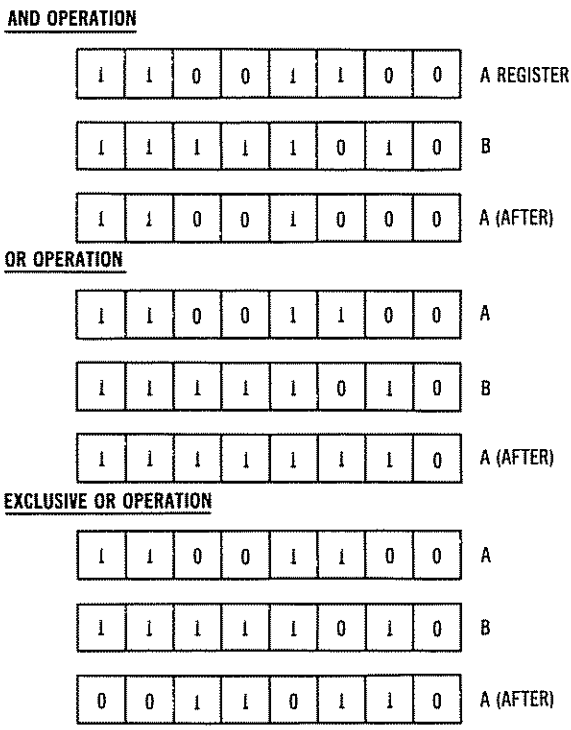


Fig. 2-7. Logical operations.

Compare instructions are very similar to subtracts. An operand from memory or another cpu register is subtracted from the contents of the A register. The flags are set as in the subtract. The result, however, does not go to the A register, but is discarded. A compare allows testing of an operand by

setting the flags without destroying the contents of the A register, a useful instruction. There is only one compare, the "CP" instruction, which again has several addressing modes.

In addition to the single compare instruction, there is a *block compare* set of instructions that allows an 8-bit compare of one operand to a specified block of memory locations. This is one of the most powerful features of the Z-80, as it is much faster than a software routine that does the same thing, as would have to be implemented in the 8080A. There are four block compare instructions and these will be discussed in detail in Chapter 6.

The instructions in the above discussion were 8-bit instructions; that is, they operated with two 8-bit operands. The A register was used in these instructions as an *accumulator* to hold the results of the operation. The Z-80 also allows a 16-bit add or subtract operation that uses the HL register pair in much the same way as the A register is used in 8-bit operations. In these adds and subtracts, a 16-bit operand from another register pair is added or subtracted from the contents of HL, with the result going to HL. The flags are set on the result of the add or subtract. The Z-80 also allows index register IX or IY (two 16-bit registers) to be used as the destination register in place of HL.

The remaining instructions in this group are the *increments* (INC) and decrements (DEC). These instructions are useful for adding one or subtracting one from the contents of a cpu register, a cpu register pair, or a memory location. Almost all assembly-language programs are continually incrementing or decrementing a count used as a loop control, index, or similar variable, and the INCs and DECs are more efficient than adding one or subtracting one by an ADD or SUB. Either single cpu registers, register pairs, or memory locations (8 bits) may be altered by these instructions.

Figure 2-8 illustrates the actions of the arithmetic, logical, and compare instructions and shows which cpu registers are used for operands and what types of instructions are available.

Decision Making and Jumps

There are only two ways to alter the path of execution of a program from BASIC, unconditionally or conditional upon some result, such as a variable being greater than a specified value. The Z-80 instructions "JP" and "JR" differ only in addressing mode and cause an unconditional jump to a specified location, exactly identical in concept to a BASIC GOTO.

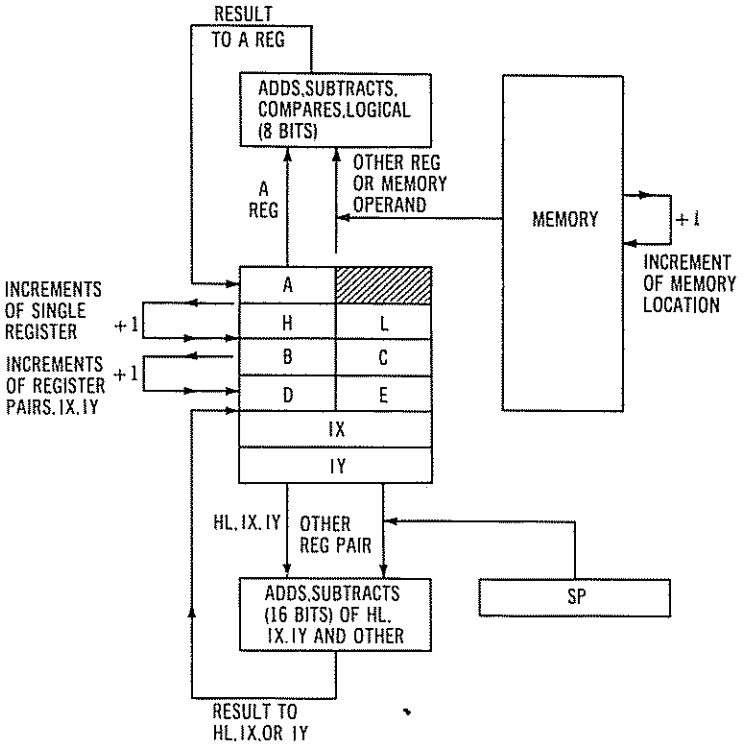


Fig. 2-8. Arithmetic, logical, and compare action.

Of course, in assembly-language jumps, a memory location is specified, rather than a line number. The instruction below will jump to Level I or II ROM

```
JP  066DH     ;JUMP TO ATTENTION
```

A similar type of jump can be made conditional upon the settings of the cpu *flags*. The flags, in turn, hold the conditions of an add, subtract, shift, or other previously executed instructions. These conditions are the conditions described in Chapter 1—zero (or non-zero) result, positive or negative result, two types of carry, parity (essentially a count of the number of "one" bits in the result), and overflow. The conditional jumps are the only way the program has of testing the results of an arithmetic or other operation, except for the conditional calls, which are very similar. Let's see how they work:

```
CP  100       ;COMPARE A REGISTER TO 100
JP  Z,42AAH   ;JUMP TO 17066 IF A = 100
```

The two instructions above cause the assembly-language program to jump to location 17066 (42AAH) if the contents of the A register are equal to 100. The CP (compare) instruction subtracts 100 from the contents of the A register. The zero flag is set if the result is zero, that is, if the A register holds 100 before the compare. If the A register does not contain 100, a value other than zero will result and the zero flag will not be set. The jump to 42AAH is made, therefore, only if the A register contained 100.

The Z-80 instruction set also has a number of instructions that are equivalent to BASIC GOSUBs. These are the CALL instructions. CALLs are used to conditionally go to a subroutine on the settings of the same flags used by jumps, or to unconditionally transfer control to a subroutine. When the transfer is made, the cpu remembers where the return point is in similar fashion to saving the next BASIC line number. The following instructions CALL a subroutine to calculate the number of TRS-80 systems (why not?) and to return at location 4801H

```
(47FE)  CALL  4C00H     ;CALCULATE NUMBER OF SYSTEMS
(4801)  ADD   2         ;ADD IN MINE AND URSULA'S
```

Note that in the above code the first instruction was located at location 47FEH, and that the next was located at 47FEH plus the length of the CALL (3 bytes), or 4801H (we'll get the reader used to hexadecimal yet!). While there are a few special jump instructions not mentioned, 99% of all jump and CALLS will be similar to those shown above.

Of course, as in BASIC, every CALL must have a RETurn. The Z-80 has two types of returns (that's correct!) conditional and unconditional. The unconditional RET always returns to the location following the CALL, while the conditional RET returns conditionally upon the flag settings. And that's about all there is to jumps, CALLs, and returns!

Stack Operations

The stack area of memory was mentioned in the first chapter. Recall that the stack area was used to store data and addresses on a temporary basis. The first use of the stack by Z-80 instructions has already been mentioned; CALLs automatically save the return address in the stack as the call is implemented. Let's look again at the last example, the CALL to location 4C00H instruction which was located at RAM memory location 47FE. When the CALL is made the PC (pro-

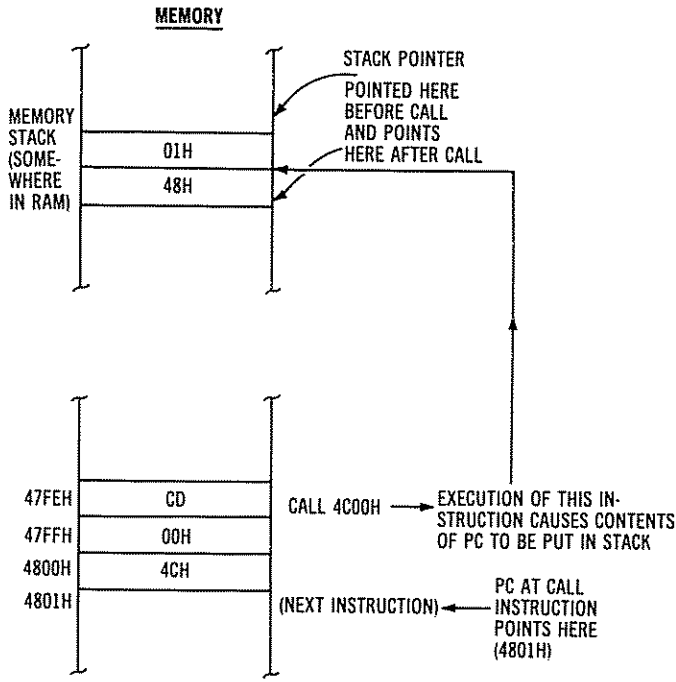


Fig. 2-9. CALL stack action.

gram counter) points to location 4801H, the next instruction (the PC is updated before the instruction is executed). As the CALL is implemented, the contents of the PC is pushed into the stack as shown in Figure 2-9. Each time the stack is used, of course, the SP (stack pointer) register is *decremented* to point to the next location to be used, or the *top of stack*. Why is the next location called the *top* of stack, when it looks like the *bottom* of stack? It's all in how one looks at it. The reader may optionally turn the book upside down to get a better picture of this action. When the RETurn associated with the CALL is executed later in the program, the return address is retrieved from the stack and put into the PC to effectively cause a jump to the return address as shown in the figure.

CALLs and RETs cause automatic stack action. The programmer may, however, temporarily store data in the stack by executing a *PUSH* instruction. PUSHes store a register pair into the stack area as shown in Figure 2-10. The data may be restored into the same or different register pair by a *POP* instruction. Of course the data comes off the stack when

a POP is executed in the same fashion it went in by the PUSH, with the most significant byte going to the high-order register (H, B, D, or the high-order portion of IX or IY) and the low-order byte going to the low-order register (L, C, E, or the low-order portion of IX or IY). The following two instructions PUSH the contents of register pair BC onto the stack, and then POP the data into register pair HL. This is a way of transferring data between BC and HL, as there is no other instruction that is able to perform this action.

```
PUSH  BC    ;CONTENTS OF BC TO STACK
POP   HL    ;HL NOW HAS CONTENTS OF BC
```

In addition to use of the stack by CALLs, RETs, PUSHes, and POPs, certain other instructions associated with interrupts and the interrupts themselves cause use of the stack. We will not be illustrating the use of interrupts in any detail, since they go beyond the scope of most assembly-language applications.

Shifting and Bit Operations

In the instructions discussed so far, we've covered a lot of ground. In fact, any computer program we want could be

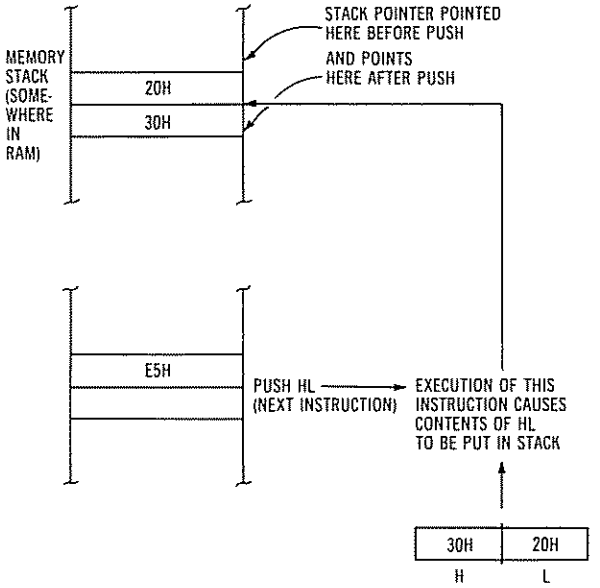


Fig. 2-10. PUSH stack action.

written in just those instructions (in fact, any computer program *could* be implemented in an eight or ten instruction machine, if it were carefully designed!). The instructions in this group, however, are niceties that make handling of *bits* and fields somewhat easier.

The shift instructions allow a single register to be *shifted* right or left. The shifting action can be visualized as pushing in another bit at the right or left end of a cpu register. As the cpu register can only hold 8 bits, a bit is "pushed out" from the other end of the register. When a zero is pushed into the end and the bit that is pushed out is discarded, the shift is said to be a "logical" shift. When the bit pushed out is carried around and pushed into the register from the other end, the shift is said to be a "circular" shift or a "rotate." The Z-80 has both logical shifts and rotates and also has a type called an "arithmetic" shift used for working with *signed numbers*. All of the shifts can be used with the A register, and some can be used with other cpu registers and with memory locations. Figure 2-11 shows some common shifts in the Z-80.

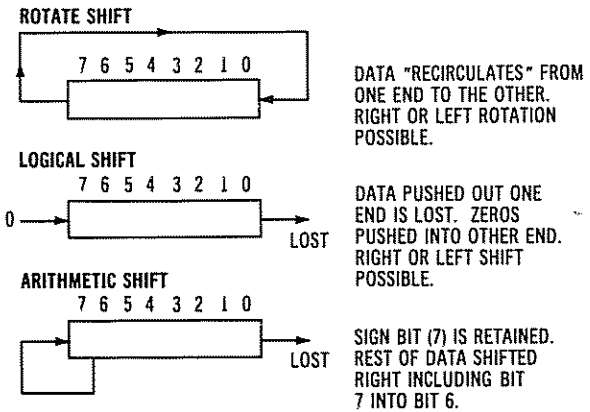


Fig. 2-11. Shifts in the Z-80.

Shifts may be used for a variety of reasons in computer programs including alignment of *fields* (subdivisions within bytes), multiplication and division, testing of individual bits, and computation of addresses. We'll say more about shifts in Chapter 8.

Bit operations allow any bit within a cpu register or memory location to be tested, set to a one, or set to a zero. As there are eight different *bit positions* that can be involved, many cpu registers, and many different ways of addressing

memory, it's easy to see why there are so many different *bit* instructions listed in the list of all Z-80 instructions. However, as with a lot of the instructions, they all resolve down to only three types, *BIT*, *SET*, and *RES*, which perform the test, set, and reset functions. These three are also covered in Chapter 8.

I/O Operations

The last group of instructions we'll discuss here are the I/O instructions. There are really only two in the Z-80, IN and OUT. All the IN does is to transfer one byte of data into a cpu register from an external device, such as cassette tape. The OUT *outputs* one byte of data from a cpu register to an external device. Although the original register used for these was the A register, the Z-80 added the use of other cpu registers as the source (OUT) or destination (IN) for the input/output operation. Another powerful feature the Z-80 added to the basic 8080A instruction set was the ability to perform a *block input/output* where the Z-80 will automatically transfer a block of data into an input area or output a block of data from an output area. The input "areas" in this type of operation are called *I/O buffers* or simply "buffers." More about input/output operations in Chapter 10.

A Program of a Thousand Locations Begins With the First Bit

The above homily was found inscribed on the first real digital computer, Babbage's Folly of a hundred years ago. It still holds true today. None of the instructions discussed here is *that* sophisticated; most are very easy to comprehend. If you will believe that and the idea that there are many ways to write a program that will do a specific task, you are prepared to advance into the ranks of assembly-language programmers. In the next chapter we will look at the last tedious description of the Z-80 instructions, their addressing modes. We will then be in a position to "lay down some code" and vindicate Babbage.

CHAPTER 3

Z-80 Addressing

The last chapter covered the types of instructions that are available in the Z-80 of the TRS-80. We warned the reader not to be intimidated by the many different instructions as they could really be grouped into a much smaller number. In this chapter we will talk about another factor that makes life interesting for Z-80 programmers— the wide variety of *addressing modes* that are available in the Z-80. Many instructions have several types of addressing modes, and the choice must be made of which one to use to do a certain task. Here again the reader shouldn't be frightened by the addressing modes available, as they are all readily understood.

Why Not One Addressing Mode?

If all instructions performed different functions, but worked with operands from the same place and operands of the same number, we could, in fact, have one addressing mode. However, we know from the last chapter that this is not true. We can add two operands from two cpu registers or one operand from a cpu register and one from memory. We can add two register pairs. Obviously the ADD instructions for these cases must be different, as they specify different locations for the operands. There are a few other instructions that we did not mention in Chapter 2 that require *no* operands. One example is SCF, which sets the carry flag. It would be foolhardy (or at

least ill advised) to attempt to make the instruction format for this type of instruction the same as the instruction format for an ADD.

To further complicate the addressing situation, we must consider the grandfather and father of the Z-80, the 8008 and 8080A, and their addressing modes. The 8008 had a very limited addressing capability. To address an operand in memory, the HL register pair had to be loaded with the 16-bit value representing the operand's location. If a load of the A register from memory location 20AAH (8362) was to be performed, the HL register pair was first loaded with 20AAH, and then a "LD (HL)" instruction was executed to perform the load. The HL register pair was used in this fashion as a *register pointer* to memory for most instructions involving an operand in memory. The 8080A, however, improved upon this type of addressing by allowing *direct addressing* of memory for certain instructions. With the 8080A, the instruction "LD A,(20AAH)" could be executed to directly load the A register with the contents of location 20AAH, without having to first point to that location with the HL register. Of course the 8080A retained the earlier addressing mode of the 8008. The Z-80 further expanded upon the 8008 and 8080A addressing capability by adding *indexed addressing* and other addressing modes, which permitted such operations as "LD A,(IX+123)" where index register IX points to the start of a table at 20AAH, and the "+123" refers to the 124th entry in the table.

"And that, Jimmy, is why we have the various addressing modes in the Z-80 today." "Gee, Mr. Computer Science, could we look at the Z-80 addressing modes in more detail now?" I thought he'd never ask . . .

Implied Addressing: No Addressing at All

The first of the addressing modes is *implied addressing*. This mode is used for simple instructions that require no operands, such as the SCF instruction which sets the carry flag. Other instructions of this type are *CCF*, *C*omplement *C*arry *F*lag, *DI*, *D*isable *I*nterrupts, *EI*, *E*nable *I*nterrupts, *HALT*, Halt CPU, and *NOP*, *N*o *O*peration, to name a few more. Because these specify a simple action and no operand, they can generally be held in an instruction of one byte, as is shown in Figure 3-1. Every time the cpu encounters the SCF instruction it will set the carry flag in the cpu and fetch no more bytes; the cpu knows the SCF instruction is only one byte long, as it knows the lengths of all other instructions.

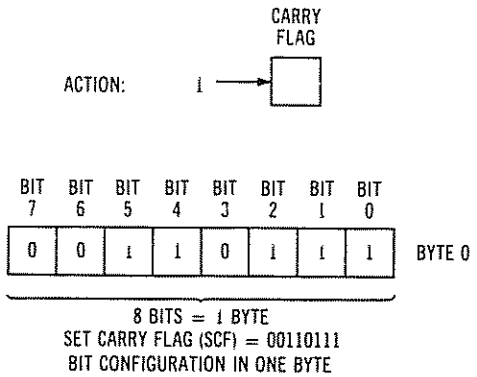


Fig. 3-1. Implied addressing.

Immediate Addressing

In immediate addressing the operand is contained within the instruction itself, rather than in a memory location. This type of addressing is used to load or perform arithmetic or logical operations with *constants*. Suppose we want to add 23 to the contents of the A register. One way to do this would be to have the value of 23 in a memory location and then perform the ADD as in

```
LD    B,A          ;MOVE A TO B
LD    A,(2111H)    ;2111H (8465) CONTAINS 23
ADD   A,B          ;ADD A REG AND B REG
```

If we had to use many constants throughout the program, however, the program would be *filled* with locations that held constants of various values, and we'd have to recall where each one was located.

Immediate addressing gets around this problem by allowing an instruction such as

```
ADD   A,23     ;ADD 23 TO THE A REGISTER
```

The actual appearance of the "ADD A,23" is shown in Figure 3-2. The first byte of the instruction is the *operation code* of the instruction, the code that tells the cpu what the instruction is and how long it is (the implied type of instructions really had a one-byte operation code). "Operation Code" has been shortened to *"opcode"* (those long cafeteria lunches again). In general, the *first* byte of an instruction in the Z-80 is the opcode, but some instructions have *two* bytes as opcodes. The second byte of the "ADD A,23" is the *immediate data* value of 23 decimal or 17H. The data value is *in the instruction it-*

self, rather than in another memory location located far away from the instruction.

Both 8-bit and 16-bit (one and two byte) immediate instructions are available in the Z-80. The one byte immediate instructions load a register or allow arithmetic or logical operations on the A register. Some samples are

```
LD    H,100     ;LOAD H REG WITH 100
LD    A,0FBH    ;LOAD A REG WITH -5
ADD   A,50H     ;ADD 50H (80) TO A REGISTER
AND   A,7       ;AND LOWER THREE BITS
```

The two byte immediate instructions in the Z-80 are used to load register pairs with constants. The instruction

```
LD  BC,3000  ;LOAD BC WITH 3000
```

loads register pair BC with a constant value of 3000 decimal. As two bytes are involved in the data, the immediate data value in the instruction is contained in bytes 2 and 3 of the instruction, as shown in Figure 3-3. Byte one is the opcode for a "LD BC" type instruction. Note that the hexadecimal representation of 3000, 0BB8H, is reordered least significant byte first in the instruction. As we mentioned earlier, all 16-bit data is handled in this manner in the Z-80. *If you are doing assembly-language programming, you will never have to*

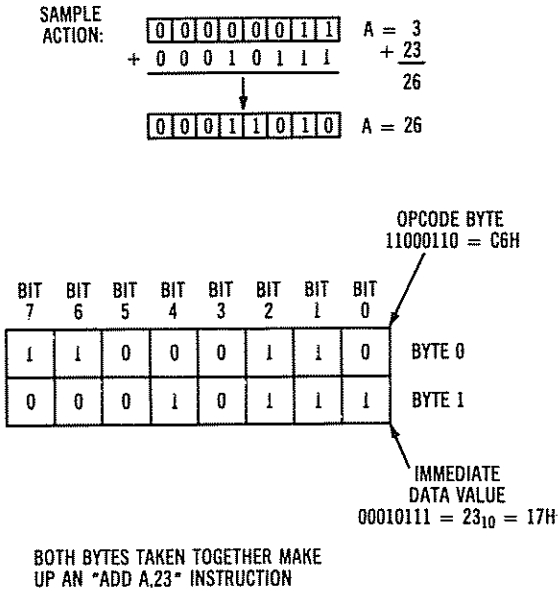


Fig. 3-2. Immediate addressing, 8 bits.

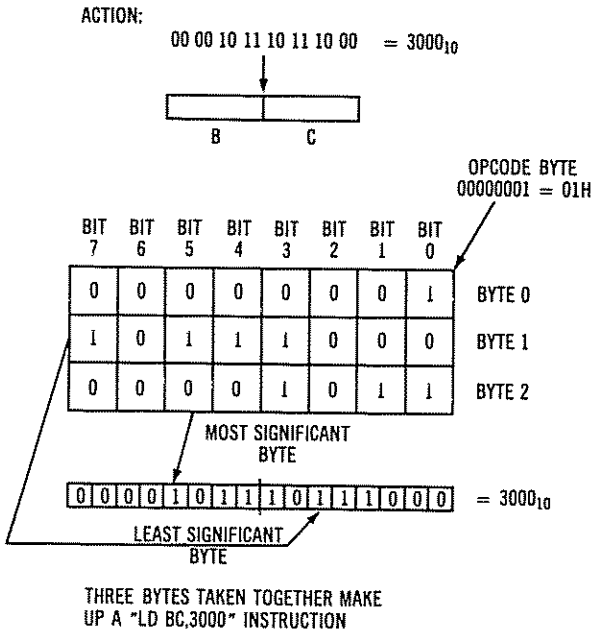


Fig. 3-3. Immediate addressing, 16 bits.

worry about putting data in the right order; the assembler program will do it for you. When the assembler sees the "LD BC,3000" it will generate a 3-byte instruction, with the data reversed in the second and third bytes. If you are "patching" code in machine instructions, however, or entering instructions in machine form (and there are some occasions when this must be done), you must be aware of this format.

Register Addressing

When a program adds two operands from cpu registers, the cpu knows that one of the operands (the destination) is in the A register. The location of the second operand (the source) must be coded in the instruction, however. Now, we have 14 general-purpose cpu registers, A, B, C, D, E, H, and L and their primed equivalents. As only one set, the primed or non-primed, is active at any given time, there are really only seven registers that may be used in an ADD operation with the A register. Does it sound reasonable to have a one-byte operation code, followed by two bytes indicating the code for the cpu register? Not at all. Since in three bits we can express the

numbers 0 through 7 (000 through 111 binary), we can in fact *code* those register names into a three-bit value contained within the instruction itself. This code is called a *field*, since it is smaller than a byte. Its use is shown in the "ADD A,D" instruction of Figure 3-4 which adds the D register to the A register. The *register field* value of 010 signifies that the D register will be used in the ADD. Note that the instruction is only one byte; that byte includes both opcode information and the register field information.

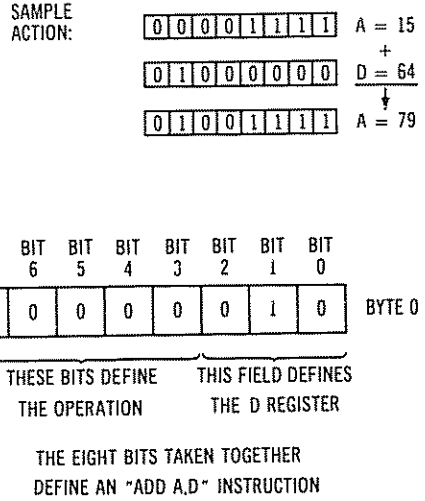


Fig. 3-4. Register addressing.

In addition to register fields that specify single cpu registers, certain instructions specify register pairs. There were originally four register pairs in the 8080A, A and flags, B and C, D and E, and H and L. Because of this many instructions will have a two-bit field (not a value judgment) that is used to specify one of the four original pairs. An example of this would be the "ADD HL,BC" instruction which adds register pair BC to register pair HL. As Figure 3-5 shows, a two-bit field within the two-byte instruction is used to specify a code of 00 for register pair BC.

With the expanded instruction set of the Z-80, however, fields must also specify the additional 16-bit registers of IX and IY, as shown in Figure 3-6. Here the instruction is an "ADD IY,SP", in which the contents of the 16-bit SP (stack pointer) register is added to the IY register.

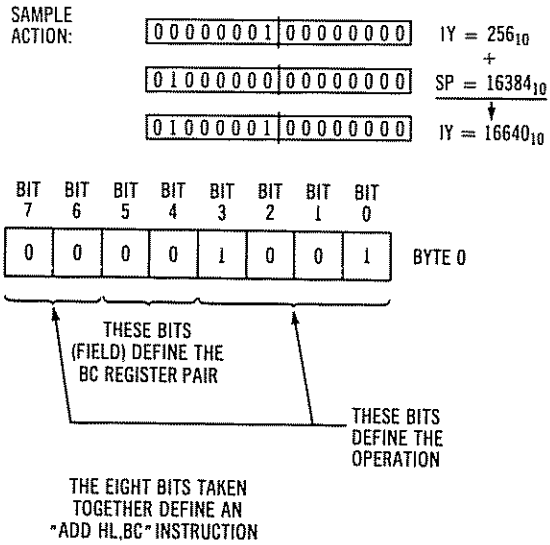


Fig. 3-5. Register pair addressing.

Once again, the assembly-language programmer need not be concerned with constructing the instruction with the proper codes in the fields, but may infrequently need to investigate the machine-language code spewed out by the assembler.

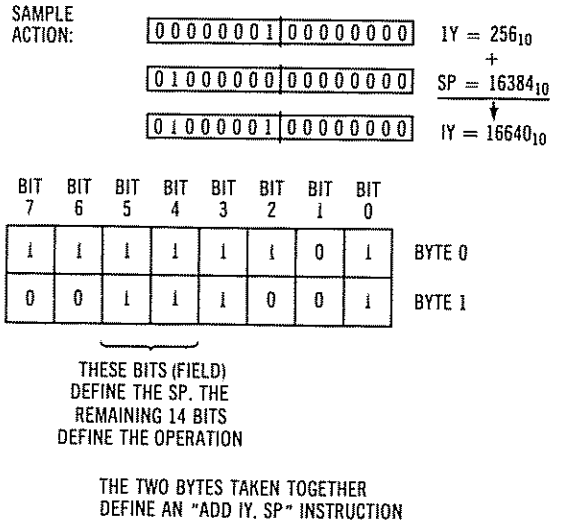


Fig. 3-6. Index register addressing.

Register Indirect

We mentioned this form of addressing earlier in the chapter. This was the main method of addressing memory in the 8008, and it used the HL register to point to the memory location of the operand. The 8080A added the capability to use BC and DE as "pointers" for loading the A register and storing the A register. You may be asking why this method should even be used in the Z-80. The answer is that many instruction types do not allow the operands to be addressed directly. While it is possible to load the A register from a memory location directly specified in an instruction [such as "LD A,(1234H)"], it is not possible to add a memory operand directly to the A register from memory [such as the invalid instruction "ADD A,(1234H)"]. It is possible, however, to set up the HL registers as a register pointer and *then* do an ADD, such as "ADD A,(HL)" or to set up the HL registers and do a variety of other things. In general, the only *direct* way into the cpu registers is through the A register. It alone is the only register (with two exceptions that permit the HL register pair to be loaded or stored) that can be loaded or stored by an instruction that specifies a direct memory address. Other registers in the cpu must use register indirect means to load or store data, or some form of *indexing* covered below. To show how this works, consider the following instructions which load the B, C, and D registers with the contents of memory locations 1000H, 2000H, and 3000H. Two ways of doing this are shown, one by loading the memory location into the A register, and then transferring it to the other cpu register, and the second by using the register indirect method.

```
(1)  LD   A,(1000H)   ;GET CONTENTS OF 1000H
     LD   B,A         ;TRANSFER TO B
     LD   A,(2000H)   ;GET CONTENTS OF 2000H
     LD   C,A         ;TRANSFER TO C
     LD   A,(3000H)   ;GET CONTENTS OF 3000H
     LD   D,A         ;TRANSFER TO D
(2)  LD   HL,1000H    ;SETUP POINTER REGISTER PAIR
     LD   B,(HL)      ;LOAD B WITH CONTENTS OF 1000H
     LD   HL,2000H    ;SETUP POINTER REGISTER PAIR
     LD   C,(HL)      ;LOAD C WITH CONTENTS OF 2000H
     LD   HL,3000H    ;SETUP POINTER REGISTER PAIR
     LD   D,(HL)      ;LOAD D WITH CONTENTS OF 3000H
```

The register indirect method of addressing is used for many different types of instructions including loads, arithmetic, logical, and shifts. It is *always* used with 8-bit (one byte) type of operations. Because it does not have to specify a memory

location, it is usually a one-byte instruction, and really comes close to being an implied addressing type. A typical register indirect instruction is shown in Figure 3-7 which shows a rotate-type of shift performed on the memory location addressed by the HL register pair used as the pointer.

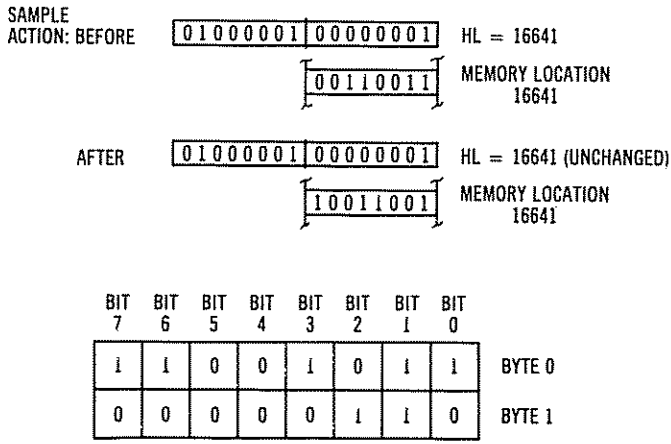


Fig. 3-7. Register indirect addressing.

Direct Addressing

Direct addressing is used with two general types of instructions, loads and jumps. We have been speaking of loading the A register directly and contrasting it with indirect means. When a direct instruction of this type is used, the second and third bytes of the instruction hold the 16-bit (two byte) memory address of the memory location to be used. The instructions "LD A,(4000H)" and "LD (4000H),A", which load A with the contents of location 4000H(16384) and store the contents of A into location 4000H, respectively, are shown in Figure 3-8. The two bytes representing the address are reversed, with the low order byte first, and the high-order second.

The HL register pair may also be stored or loaded directly with this type of addressing. In this case the register pair is stored in *two* memory locations as two bytes of data are in-

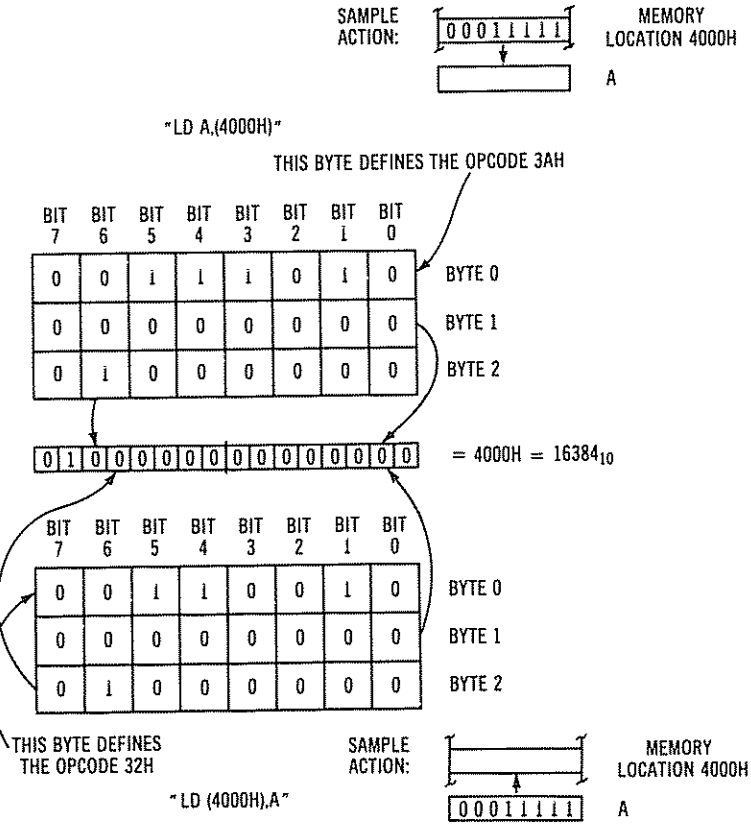


Fig. 3-8. Direct addressing.

volved. As usual, the first (lowest) holds the low-order byte and the next (highest) holds the high-order byte. The address used in the instruction itself points to the first byte of memory to be used. The instruction "LD HL,(5000H)" will load register L with the contents of memory location 5000H (20480) and register H with the contents of location 5001H (20481) as shown in Figure 3-9. Register pairs BC and DE and SP, IX, and IY may also be loaded or stored directly.

Direct addressing is also used with CALLs and jump instructions. All CALLs are direct addressing types, and all jumps are direct addressing except for the relative type of jumps covered later in this chapter. The format for CALLs and JPs is shown in Figure 3-10. The first byte specifies the opcode for the instruction and informs the cpu whether the instruction is conditional or unconditional and whether it is

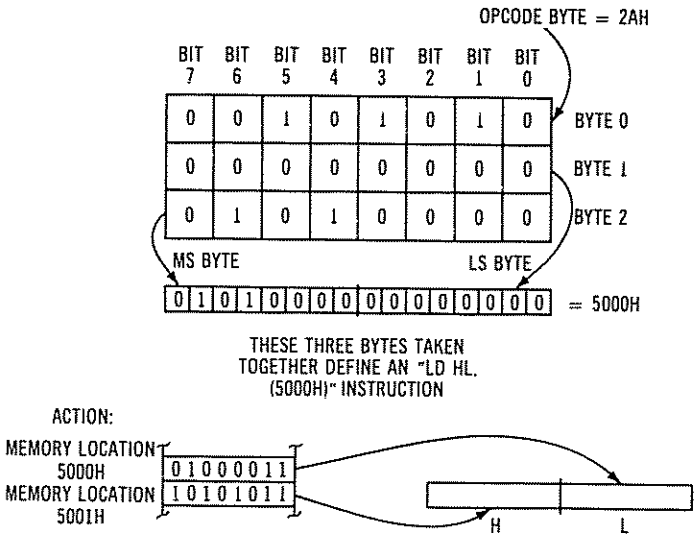


Fig. 3-9. Direct addressing involving HL.

a jump or CALL. The second and third bytes are the address of the jump location or CALL location. This data is not used to reference a memory location as with other types of instructions, but is simply jammed into the program counter to replace the "next instruction" address that was automatically calculated when the instruction was first accessed. The effective action is a jump or CALL to the location specified. As usual, the 16-bit address is in reverse order in the instruction.

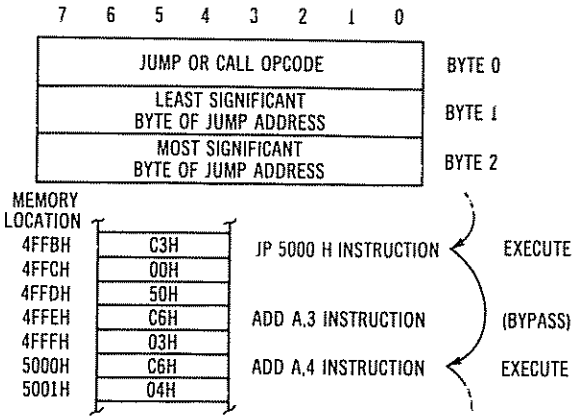


Fig. 3-10. Jump and CALL format.

Relative Addressing

Relative addressing is used only for *relative jump* instructions; no other types of instructions use the relative type of addressing, including CALLs. The relative jump uses two bytes to specify the instruction, one byte for the opcode, and one byte for the memory address. Oh, oh! There's that kid in the back of the class again. He's asked a very valid question—how can one byte specify a memory location when it takes 16 bits or two bytes to specify a memory location value of 0000H to FFFFH (0 to 65535). It would appear that we can't jump to anything other than locations 0 through 255, the values that can be held in one byte. Not true! What if we used that one byte to find the memory location by adding the contents of the program counter (PC) to the value found in the byte. The new address or *effective address* would be the address in the PC plus the value in the instruction byte. What's in the PC? Well, we know that the PC points to the next instruction after the jump. If we add the value in the instruction to the PC we get a value that points to the next instruction −128 through the next instruction plus 127, depending upon what was in the instruction byte *displacement*. In fact, with this type of instruction we can jump within a limited range of 256 bytes of the instruction itself. Since most of the jump destinations within a typical program are close to the jump instruction, this appears to be a valuable instruction, as it saves one byte of instruction length over a regular JP. Let's see how this works. Suppose that at location 4300 we have a jump to location 4350H. After the "JR 4350H" instruction has been fetched, the PC points to location 4302H, the next instruction. If we look at the second byte of the JR instruction, we find that the assembler has put a 4EH there. Adding the 4EH and 4302H we obtain 4350H, which is the jump address (effective address) that is jammed into the PC to cause the jump. This process is shown in Figure 3-11.

The second byte of the JR instruction actually holds an 8-bit *signed* value in this case. Rather than representing a range of binary values from 0 through 255, the displacement in the second byte represents a range of −128 through +127. Binary numbers in this two's complement form will be discussed further in Chapter 6, but for now just remember that the displacement may also be negative in a JR. Of course in the JR, as in other instructions, the programmer does not have to tediously compute the value to be put into the displacement byte; the assembler will automatically do it for him. (That's

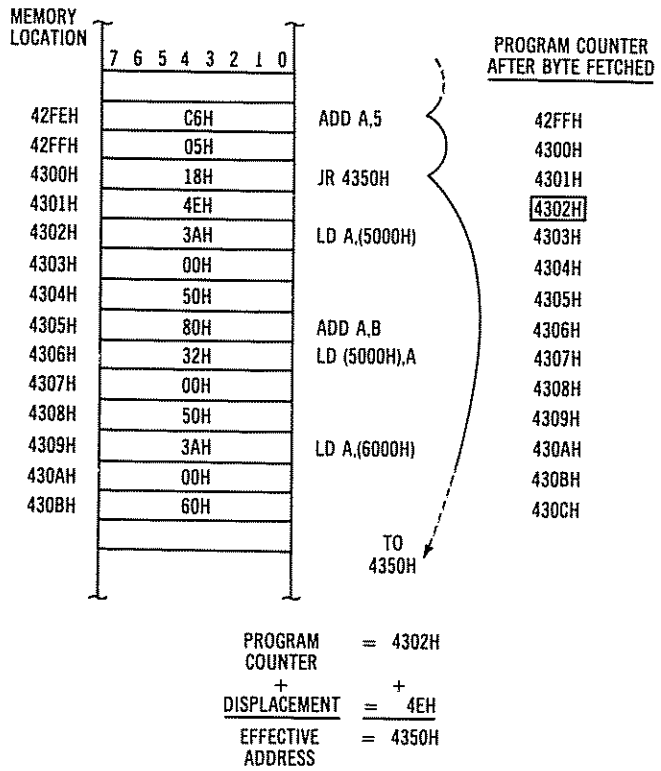


Fig. 3-11. Relative jump action.

why we have computers!) You'll see in the next chapter that the instruction referenced may actually be given a name, much in the same fashion as a BASIC variable name, which the assembler will use in figuring out what the displacement should be.

A Special Type of Call

The *RST*, or *Rest*art instruction, started out in the 8080A as an instruction geared for *interrupts* to the microprocessor, special signals to the cpu that signal external events such as typed characters or "line printed." In the TRS-80, however, the RST instruction is used for a second purpose, that of a "short" CALL, to call a subroutine. The RST permits a call to one of eight memory locations located at either 0000H, 0008H, 0010H, 0018H 0020H, 0028H, 0030H, or 0038H (decimal 0, 8, 16, 24, 32, 40, 48 or 56). As the RST is only one byte

long, it saves two bytes over a normal CALL instruction and is valuable for commonly used *subroutines* that would be frequently called in a program.

The appearance of an RST is shown in Figure 3-12. There is a three-bit field that specifies which of the eight locations is being CALLed as a subroutine. The actual location addressed is found by multiplying the contents of the 3-bit

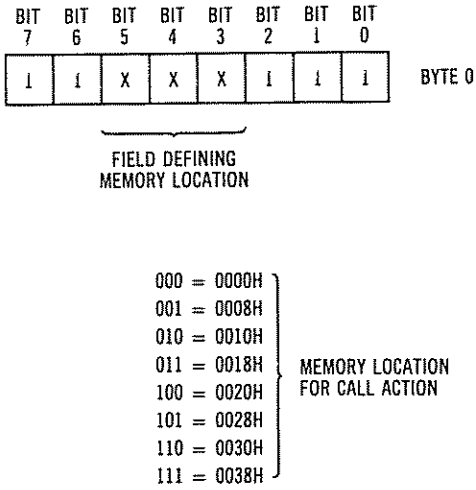


Fig. 3-12. Restart instruction.

field by eight. Naturally, the program does not have to do this dirty work, but simply specifies an

```
RST  18H     ;CALL ADDITION SUBROUTINE
```

or similar instruction to generate the instruction.

Indexed Addressing

This is one of the powerful addressing modes added to the base 8080A instructions by the Z-80. *Indexing* allows the assembly-language program to easily access data that is arranged in contiguous tables. Suppose, for example, that we have a table of employee data as shown in Figure 3-13. Each employee record has name, address, marital status, number of TRS-80 systems owned, and other relevant particulars. It

would be nice to have the capability to access data grouped around a particular employee record in the table. We know that we *could* do this by other addressing means, such as loading the A register directly but this is not an elegant way to do things, and we would like to consider ourselves sophisticated programmers. Take heart! The Z-80 indexed addressing capability affords an elegant solution (or at least a nice one . . . well, it's *pretty* good . . .).

Initially the program loads the value representing the address of the table entry into an index register, in this case IX, although IY could have been used as easily. Now, to access any data near the record, it's simply a case of using an in-

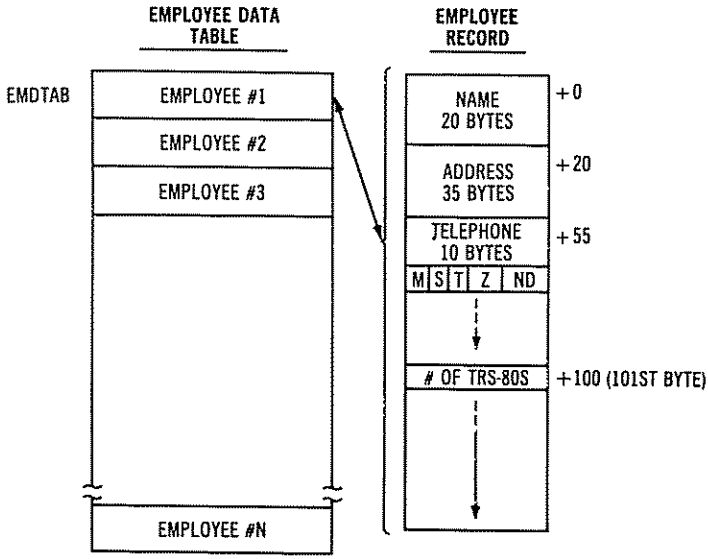


Fig. 3-13. Indexed-addressing table example.

dexed instruction. If the index register had been loaded with 5000H, the instruction

```
LD  B,(IX+100)  ;GET # OF TRS-80S
```

would load the 101st entry, the number of TRS-80 systems, into the cpu B register. In other words, the cpu creates an *effective address*, similar to the relative jump effective address, by adding the contents of the index register with a displacement byte from the instruction. In the case above, the instruction would appear as shown in Figure 3-14. The first two bytes are opcode and a register field that specifies the register

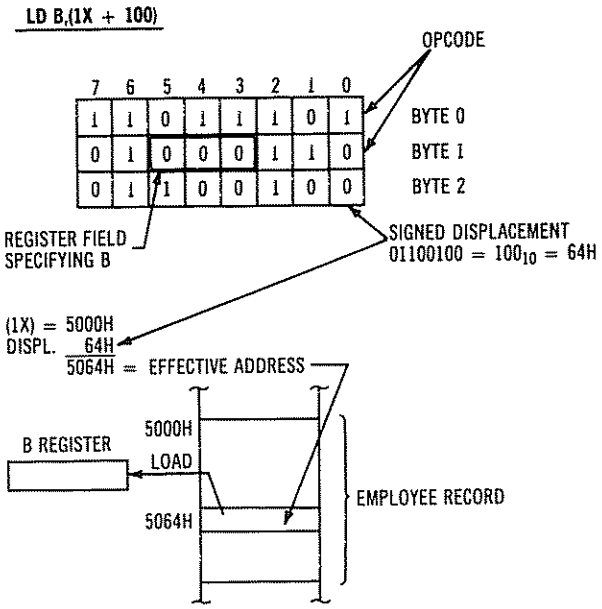


Fig. 3-14. Indexing into table.

to be loaded. The next byte is a *signed* displacement that is added to the IX register to form the effective address; in this case the displacement is 64H as shown. The effective address calculated for the access here is 5000H + 64H or 5064H, the memory address of the number of TRS-80 systems for employee number one.

Indexing using the IX or IY registers may be used for a variety of Z-80 instructions, but, of course, is always used when the address of a memory operand is used in IX or IY.

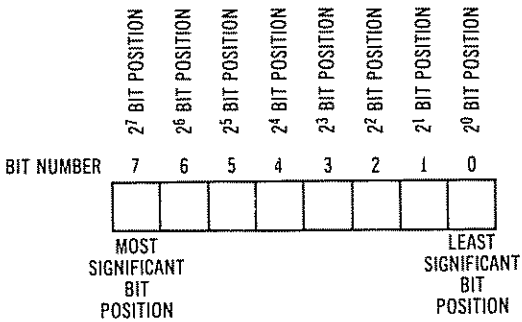


Fig. 3-15. Bit numbering.

We will speak more of indexing in Chapter 9, where table and other data structures are discussed.

Bit Addressing

All of the addressing done in the preceding sections referenced a memory location or cpu register byte. The *bit addressing* mode, used in the *bit* instructions, references a single bit somewhere in memory or a cpu register. The format of this addressing mode specifies a *bit position* from 7 through 0. The instruction

```
SET  6,(HL)     ;SET BIT 6 OF MEMORY BYTE
```

sets bit 6 of the memory location pointed to by the HL register pair pointer. Bits in memory, cpu registers, or other TRS-80 system components are *always* numbered as shown in Figure 3-15. The *most significant bit* (*msb*) is numbered bit 7, and the *least significant bit* (*lsb*) is numbered bit 0. These numbers correspond to the power of two represented by the bit position (bit 7 is 128, 6 is 64, etc.).

This addressing mode is used with the instructions of the bit instruction group only, the BIT, SET, and RES instructions. The bit addressing mode allows other addressing modes to be used in the instruction (as do other instructions, in fact), so that bit addressing may be used in conjunction with register indirect, indexed, or register addressing.

Conclusion and Confusion

This concludes the discussion of addressing modes used in the Z-80. The worst problem in the use of the addressing modes is not in understanding what they do, but in remembering which instructions use which addressing modes. I'm afraid that there is no magical solution to this except reference to Appendices I and II and experience. The saving grace is that there are always many ways to code a particular program, both in terms of which instructions to use and what their addressing types should be. There *is* no *one* correct solution to any programming problem, and there are very few "bad" programs either.

In the next chapter we will look into the use of TRS-80 Editor/Assembler and T-Bug packages and assembly-language and machine-language coding. If you have made it through these first few chapters, you have an excellent chance of becoming a certified TRS-80 assembly-language programmer!

CHAPTER 4

Assembly-Language Programming

Now that you have digested the necessary background information on the TRS-80 and Z-80 (hope it wasn't too filling), we are ready to assemble some assembly-language programs and run them. There are basically two ways to construct and implement machine-language programs for the TRS-80. The first way is by *machine-language* coding and the second is by *assembly-language* coding. In the first method, a program is written out, or coded, on paper and manual methods are used to construct the proper sequence of instructions for the Z-80; the program is actually coded in machine language. In the assembly-language method, the Editor/Assembler is used to *translate* a symbolic form of the instructions into machine-language, which is then loaded into the TRS-80 by the loader portion of the Editor/Assembler. Is the Editor/Assembler really necessary? For all programs over *one* instruction in length, the Editor/Assembler is almost a necessity for ease in editing, assembling, and loading programs. Machine-language can be employed in place of the Editor/Assembler, but only if the user likes to do tedious and exacting work. The exception to this is that *some* machine-language coding will give the TRS-80 user great insights into the way the Assembler constructs programs. Once he has this insight he then will probably want to do all of his coding in assembly language.

Machine-Language Coding

To show the reader how machine-language coding is done, let's write a program to write a "1" at the center of the video display. We know from BASIC that the video display has 1024 different character positions, 64 on the first line, 64 on the next, and 64 on each of the 16 lines making up the screen. We also might know that each character position has a video display memory location associated with it, starting at memory location 3C00H (15360) and ending at 3FFFH (16383). If we wish to display a "1" in the exact center of the screen, or as close as we can get, we would have to *store* that "1" in the memory location associated with line 9, 32 characters over. This will be location 15360 + 8 lines at 64 characters per line + 32 characters or 15360 + 512 + 32 = 15904 (3E20H). See Figure 4-1 for a diagram of the screen and memory associated with it.

Now that we know *where* to store the "1," *how* do we store it? The first thing that comes to mind is a *store* instruction.

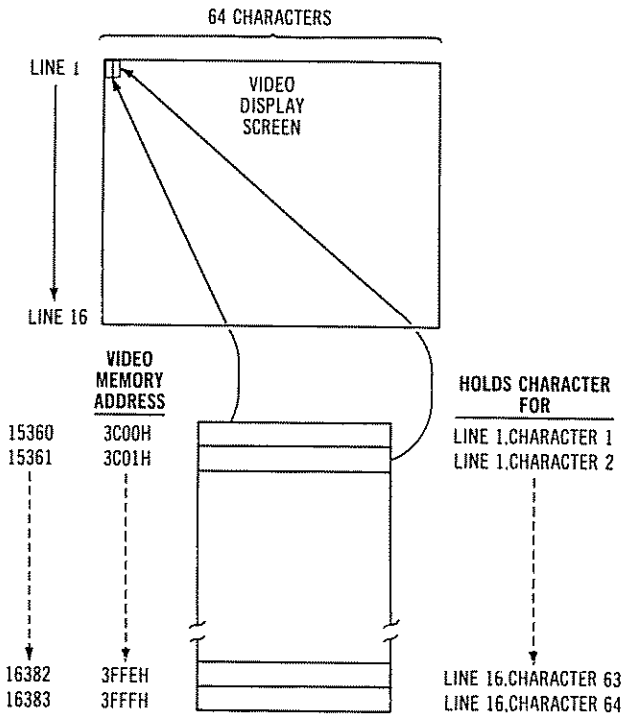


Fig. 4-1. Screen addressing.

We can *load* the "1" into a cpu register and then store it into location 3E20H. One question that comes to mind is the code for the "1." This is a 7-bit ASCII code representing the alphabetic characters, numeric values, and special characters as shown in the Level II or Editor/Assembler manuals. The code for a "1" is the value 00110001 or hexadecimal 31H (the 8th bit is set to a zero). The following instructions load the A register with the code for a one and then store it into location 3E20H.

```
LD  A,31H       ;LOAD A REGISTER WITH "1"
LD  (3E20H),A   ;STORE "1" INTO CENTER OF SCREEN
```

The first instruction in the above *program* is an *immediate* addressing type load which loads "1" from the immediate data in the instruction into the A register. The next instruction stores the A register into location 3E20H. The parentheses around the 3E20H indicate an *address* rather than a data value.

Well, it appears that this program should work. Our next task is to translate the mnemonics for the instructions into the actual opcodes, data fields, and addresses that can be input to the Z-80. We know from our discussion in the last chapter that the 8-bit immediate instructions have one byte for the opcode and one byte for the immediate value. If we look in the Editor/Assembler manual, we find that the opcode for the LD A is 00RRR110, where "RRR" represents the register code for the cpu register to be used. For an A register load, this field is 111, so we now have 00111110, or 3EH. Let's write the opcode down opposite the instruction.

```
3E 31  LD  A,31H       ;LOAD A REGISTER WITH "1"
       LD  (3E20H),A   ;STORE "1" INTO CENTER OF SCREEN
```

We've also written the immediate data value of 31H to be loaded into the A register. Now let's look at the second instruction. As this is a *direct* store, we know that it must contain a two-byte address for 3E20H. In this case the opcode is one byte long and is a 32H, with no fields. The address of 3E20H is put in reverse order into the second and third bytes of the instruction and the opcode is put into the first as follows

```
3E 31     LD  A,31H       ;LOAD A REGISTER WITH "1"
32 20 3E  LD  (3E20H),A   ;STORE "1" INTO CENTER OF SCREEN
```

About the only thing left in this program is to decide where in RAM it is to *reside*. In most programs we must know this before we start a manual or automatic assembly process, since many of the jump and CALL addresses are direct addresses

that refer to locations within the program. A good area for all systems, Level I or II, 4K and larger configurations would be near the end of the 4K RAM area or 18944 [the RAM area starts at 16384, and 18944 is 16384 plus 2560 or 18944 (4A00H)]. We will now assign the locations for the two instructions at 4A00H and 4A01H plus two bytes for the length of the LD A,31H or 4802H.

```
4A00  3E 31     LD  A,31H       ;LOAD A REGISTER WITH "1"
4A02  32 20 3E  LD  (3E20H),A   ;STORE "1" INTO CENTER OF SCREEN
```

In the process of hand-assembling the program above, we have had to do a number of things the Editor/Assembler could have done much more easily. We had to look up the opcodes for each of the instructions, insert the proper code for the A register in the first instruction, reverse the address and put it into the second instruction, find the length of the instructions and properly calculate the locations for each instruction, and find the code for the "1". All of these things *could* have been performed easily by the Editor/Assembler, leaving us free to concentrate on the *logic* of the program. In addition, the Assembler performs many other functions, such as data error checking, relative address range checking, checks on the number of operands, and so forth. For these reasons, we will be concentrating on use of the Editor/Assembler in the remainder of this book, although the reader may do his own machine-language coding from the instructions in the text, if he chooses to do so.

The TRS-80 Editor/Assembler

The TRS-80 Editor/Assembler is a program and documentation for 16K Level I or II systems. In the remainder of the book, we will assume that the reader has access to the Editor/Assembler and to the Editor/Assembler User Instruction Manual (#26-2002). The description in this chapter is meant to supplement the descriptions and operating procedures found in that manual.

As an example of edit and assembly of a new program, let's take the huge two-instruction program we did in machine language, edit and assemble it, load it, and execute it on the TRS-80. The two instructions we had originally were in *symbolic form*, that is we used symbols such as A to represent the A register code of 111.

```
LD  A,31H       ;LOAD A REGISTER WITH "1"
LD  (3E20H).A   ;STORE "1" INTO CENTER OF SCREEN
```

Before we begin editing and assembling, we need a few more things in this program and every program to put the program in proper format for the assembler. An "ORG" statement tells the assembler where the program will reside after it has been loaded. Without an ORG (*ORiGin*) the assembler could not assemble the direct addresses used in some of the instructions. We'll use the same origin as in our machine-language version, 4A00H, the ¾ point of 4K RAM.

```
ORG   4A00H       ;START AT LOCATION 4A00H
LD    A,31H       ;LOAD A REGISTER WITH "1"
LD    (3E20H),A   ;STORE "1" INTO CENTER OF SCREEN
END   4A00H       ;END-START OF 4A00H
```

As the ORG statement does not actually generate a machine language instruction as do the two LDs, it is called a *"pseudo-operation"* or *"pseudo-op."* In place of an opcode, pseudo-ops have *mnemonics* which tell the assembler what to do for program origin, end, and data. The ORG pseudo-op has one *operand* associated with it, a value indicating where the origin is to be.

Another pseudo-op that is an absolute necessity is the *END* pseudo-op. END, of course, tells the assembler that it has reached the end of the assembly-language program. It may or may not have an operand. If it does, the operand indicates the starting point for a program after the load. Here the starting point is 4A00H, so we have specified this value as an operand for the END.

Now before we enter this short program, we should really check over the logic of the program itself. This is called "desk checking," and saves reediting and reassembling the program several times. We may *still* have to change the program in the general case, but a good desk check will reduce the number of times that the program has to be edited and assembled. The only flaw in the program seems to be at the end. When the program is loaded and run the first LD will load the "1" into the A register and the next LD will store the "1" in A into location 3E20H, the center of the screen. What instruction is executed next? The one following the LD (3E20H),A. Since we have not specified another instruction after the second, however, there will be *no* third instruction, or if there is, it will be purely coincidental. This means that after executing the two instructions in the program, the cpu will go merrily on its way, attempting to execute what is referred to as *garbage*. We must terminate the program properly. One way to terminate the program would be with the addition of a third instruction that jumps to a known set of code, or a known

return point for Level I or Level II BASIC. We'll add a jump, but we'll simply jump *to* the jump itself to create an endless loop of jumping to the jump to avoid executing meaningless instructions that would cause unexpected results.

```
        ORG   4A00H      ;START AT LOCATION 4A00H
        LD    A,31H      ;LOAD A REGISTER WITH "1"
        LD    (3E20H),A  ;STORE "1"INTO CENTER OF SCREEN
LOOP    JP    LOOP       ;LOOP HERE
        END   4A00H      ;END-START OF 4A00H
```

We've introduced an important concept here. We did not have to calculate the location of the JP instruction ourselves. We gave the JP instruction a *label* of "LOOP," and let the assembler figure out that the "JP LOOP" is equivalent to "JP 4A05H." The reference to LOOP is a *symbolic address.*

Editing New Programs

We are now ready to use the Editor/Assembler. Load the Editor/Assembler using the procedures outlined in the Editor/Assembler manual. After a successful load, the program will display the prompt "*?" on the video. Now type *I100,10*, followed by an *ENTER*. This puts the Editor into the *I*nsert mode and allows us to enter a number of lines starting with line number 100, and incrementing by 10 for each line. Now type in the five lines. The → (right arrow) may be used to tab to the next column, the ← to backspace for error correction, and the ENTER must be used to indicate the end of each line. After you've entered the five lines, press BREAK and the program will return to the "*" prompt. The entire dialogue is shown in Figure 4-2.

The editing process is now complete. The Edit *buffer* has five lines of text duplicating what we have typed in. As a check on our input we can *P*rint the edit buffer by the command "*P#:*", which will display the entire text buffer of

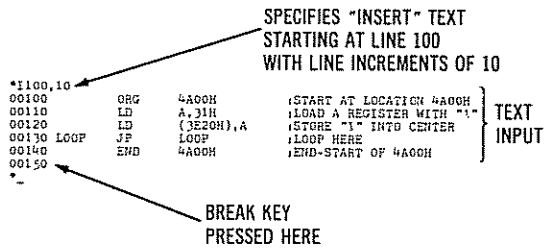


Fig. 4-2. Editing operations.

five lines. The editor does not check the text we are inputting, and will not catch any errors in *syntax* or mispellings (even TRS-80 programmers have been known to make occasional mistakes).

Assembling

Now we are ready to assemble. Type "*A" for assembly, and the Assmbler portion of the Editor-Assembler will assemble the five lines, displaying the assembled code on the screen as shown in Figure 4-3. The right-hand section of the *listing* is the *source code* that we have just entered; the left hand side of the listing is the machine code information that will be loaded into the TRS-80. The first column shows the locations for the instructions, starting at location 4A00H, and incrementing for each instruction, dependent upon instruction length. The next column shows the actual machine code for the instruction. This will range from one to four bytes, dependent upon the type of instruction and its addressing modes. The next column gives the line numbers, starting with line number 100, as we specified.

```
4A00            00100          ORG   4A00H       ;START AT LOCATION 4A00H
4A00 3E31       00110          LD    A,31H       ;LOAD A REGISTER WITH "1"
4A02 32203E     00120          LD    (3E20H),A   ;STORE "1" INTO CENTER
4A05 C3054A     00130 LOOP     JP    LOOP        ;LOOP HERE
4A08            00140          END   4A00H       ;END-START OF 4A00H
00000 TOTAL ERRORS
LOOP    4A05
```

Fig. 4-3. Assembly operations.

Where is the machine language code at this time? It is in a *buffer* ready to be written to cassette tape or disc. We will use cassette tape to make it more general for all TRS-80 users. After preparing the tape, press ENTER and the Assembler will write out the *machine code* to cassette. Notice that nothing has been executed in the program to this point. The actions so far have been analogous to hand assembling the instructions and writing down the machine code to be loaded and run. The cassette tape has been used to write a *file* of *object code* representing the machine code of our short program. There's that persistent kid from the back again. . . . The *object* code looks very similar to the machine code, except that it contains addi-

tional data about the origin, file header information (the name of the cassette file—"NONAME" in this case), and other information that will help in the loading of the program.

Loading

We've assembled, edited, and now we are ready to load the object code. After the load, the machine code will be at locations 4A00H through 4A07H and we can execute the program. At this point we are done with the Editor/Assembler and can go back to Level I or II BASIC. We must now use the SYSTEM mode to load the object program; the SYSTEM mode is inherent in Level II BASIC but must be implemented by loading a special SYSTEM tape in Level I BASIC (see the Editor/Assembler manual for directions). The SYSTEM mode is used to load assembler object programs, and to transfer control to the program after it has been loaded. After the SYSTEM prompt of "*?" type in "NONAME" to load the object file from cassette tape. If a successful load is performed, the prompt "*?" will again appear, indicating that the program is now in memory at locations 4A00H through 4A07H. All that remains now is to transfer control to the starting address of the program at 4A00H. We do this by typing in the decimal equivalent of 4A00H after a slash ("/") or simply by typing in a slash, as we have indicated the starting address of the program in the END statement, and this has been saved in the object program. The result should be a "1" displayed in line 8 at the middle of the screen. Not a very impressive beginning for a programmer who will revolutionize the field of assembly-language computing, eh? But by the end of the book. . . .

Assembler Formats

Now that we have successfully assembled our first program, let us discuss assembly-language formats in a little more detail. As we saw from the listing, the basic format of all assembly lines is an optional label, an opcode or pseudo-op mnemonic, operands to fit the instruction or pseudo-op, and optional comments, as shown below.

```
THERE  ADD  A,(IY+100)  ;THIS IS THERE
```

The label may be one to six characters, the first of which must be alphabetic. There are certain *reserved words* that cannot be used for labels, such as register names (IX) and flags (C). These are listed in the Editor/Assembler manual;

just remember to stay away from labels that are the same as flags or registers, such as HL (they will assemble with an error indication).

The opcode or pseudo-op must be one of the mnemonics given in Appendix II or the pseudo-ops given later in this chapter. These mnemonics follow the standard Zilog Z-80 mnemonics, and are also provided in the description of the instructions in the Editor/Assembler manual.

The operands are in the third column of an assembly-language line. The number of operands to use depends upon the instruction and addressing type. As we know from Chapters 2 and 3, some instructions have no operands, such as *SCF*, and others have one or two, such as "BIT 7,(HL). The operands may specify data, as in the immediate load

```
        LD   HL,3FFFH    ;LOAD HL WITH 3FFFH
```

or addresses, as in the load

```
        LD   HL,(3FFFH)  ;LOAD HL WITH CONTENTS OF 3FFFH
```

Note the difference in data and addresses. Except for jumps and CALLS *addresses are always enclosed by parentheses*, and data is never enclosed in this fashion. Jump and CALL operands are addresses not enclosed by parentheses. The formats for Z-80 instructions are given in Appendix II and in the instruction descriptions of the Editor/Assembler manual. In place of numeric operands for data or addresses, symbolic names may also be used. These names must have a corresponding label somewhere else in the program. An example of this is

```
        LD   A,(COUNT)  ;LOAD COUNT OF COUNTS
        LD   B,(DUKE)   ;LOAD COUNT OF DUKES
                        ;(other code)
COUNT   DEFB 0          ;LOCATION HOLDING COUNT OF COUNTS
DUKE    DEFB 0          ;LOCATION HOLDING COUNT OF DUKES
```

Some instructions refer to flags; for example, the conditional jumps that test a flag status for the jump. Certain mnemonics are used for the flag=0 and flag=1. These are "C" and "NC" for carry flag=1 and carry flag=0, "Z" for zero flag=1 and "NZ" for zero flag=0, "PE" for parity even (P/V flag=1) and "PO" for parity odd (P/V=0), and "M" for minus (S flag=1) and "P" for positive (S flag=0). These mnemonics are reserved for the use of flag references. An example of an assembler line using a flag reference is

```
ADD   A,37      ;ADD A AND 37
JP    M,OMMY    ;GO IF RESULT NEGATIVE
                ;RESULT POSITIVE HERE
```

The comments column is also optional. When you are *debugging* a program some dark and lonely night and wondering what you did in those ten instructions that have no comments, think back upon this advice: There are never too many comments. A comment may also be used in a line by itself as in the following code.

```
;THIS IS A ROUTINE FOR A STAND-UP COMIC
      LD  A,(JOKE)  ;GET JOKE FROM MEMORY
      LD  (LOC),A   ;DELIVER
```

Just as BASIC allows various expressions, combinations of symbolic variables and constants, so does the assembler allow limited use of expressions. These are detailed in the Editor/Assembler manual. Addition, subtraction, logical AND, and shifts are allowed. We will only be using addition and subtraction in this book, and leave the use of the others to your experimentation. Addition and subtraction are represented by "+" and "−", just as they are in BASIC. As an example of the use of expressions in assembly language coding, let's use the program we've been working with. We stored a "1" into the center of the video display, which was really in the center of the video memory area. We knew that the start of the video memory was at 3C00H, and that we wanted to store the "1" at the 512 + 32 character position on the screen. The following code will perform the store.

```
;STORE AN ASCII ONE NEAR CENTER OF SCREEN
;
      LD  A,31H                 ;ASCII ONE
      LD  (3C00H+512+32),A      ;CLOSE TO CENTER
```

The same technique could be used for subtracts, or with expressions consisting of symbolic labels and constants. Note that in the expression, hexadecimal data was intermixed with decimal data. Hexadecimal data is *always* suffixed by an "H" to mark it as hexadecimal. In addition, hexadecimal data must have a leading zero, if the hexadecimal value starts with A through F. The value of A000H will confuse the assembler and result in the assembler trying to find a label of A000H, rather than treating it as data. Decimal values may simply be values without either leading zeros or suffixes, as shown in the ex-

ample. We will use various expressions in the course of the chapters involved with programming examples in this book, so you will have a chance to see further use of them.

More Pseudo-Ops

When we wrote our first assembly-language program earlier in this chapter we used two pseudo-ops, the END and ORG pseudo-ops. The TRS-80 Editor/Assembler has six additional pseudo-ops, *DEFB*, *DEFW*, *DEFM*, *DEFS*, *EQU*, and *DEFL*. They are used to generate byte, word, and string data, to reserve memory, to equate a label, and to set a label.

The DEFB generates one byte of data rather than an instruction. Suppose that in the program we've been using we wanted to store the ASCII one in memory as a constant value, rather than loading it as an immediate value. The following code would do exactly that. A 31H, the ASCII 1, would be stored at location 4A09. When the program was executed, the instruction at 4A00 would load the contents of "ASCONE" into the A register.

```
                00100 ;CODE TO WRITE A ONE AT CENTER OF SCREEN
4A00            00110           ORG     4A00H           ;START AT LOCATION 4A00H
4A00 3A094A     00120           LD      A,(ASCONE)      ;LOAD A REG WITH "1"
4A03 32203E     00130           LD      (3E20H),A       ;STORE "1" INTO CENTER
4A06 C3064A     00140 LOOP      JP      LOOP            ;LOOP HERE
4A09 31         00150 ASCONE    DEFB    31H             ;ASCII ONE
4A0A            00160           END     4A00H           ;END-START OF 4A00H
00000 TOTAL ERRORS
LOOP    4A06
ASCONE  4A09
```

The DEFB can be used as many times as is necessary. Each time it appears, one byte of data is generated. The DEFW, on the other hand, *DEF*ines a *W*ord of data, or two bytes. As we are frequently working with 16-bit data for addresses or constants to be used with register pairs, the DEFW is handy. The following code generates both 8- and 16-bit constants by use of DEFB and DEFW.

Of course the 16-bit values generated are in the usual reverse order. The most significant byte is last and the least significant byte is first.

```
                00100 ;BUILD A TABLE OF DATA
4000            00110           ORG     4000H
4000 00         00120           DEFB    0
4001 11         00130           DEFB    11H
4002 0A         00140           DEFB    0AH
4003 1100       00150           DEFW    11H
4005 0A00       00160           DEFW    0AH
4007 DEFF       00170           DEFW    -34
0000            00180           END
00000 TOTAL ERRORS
```

The DEFB may also be used to generate a one byte ASCII value directly. This saves the programmer the trouble of looking up an ASCII equivalent code for character data. Not only does the assembler do this for *one* byte, but it also generates a whole *string* of characters to be used for messages or other purposes when a DEFM, or *DEF*ine *M*essage is encountered. The following code shows how the DEFB is used to generate one byte ASCII values and how the DEFM is used to generate a string of characters.

```
4000            00100           ORG     4000H
4000 31         00110           DEFB    '1'             ;ASCII ONE
4001 23         00120           DEFB    '#'             ;ASCII #
4002 54         00130           DEFM    'THIS BEATS HAND ASSEMBLING'
4003 48         4004 49         4005 53         4006 20         4007 42         4008 45
    4009 41         400A 54         400B 53         400C 20         400D 48         400E 4
1       400F 4E         4010 44         4011 20         4012 41         4013 53         40
14 53           4015 45         4016 4D         4017 42         4018 4C         4019 49
  401A 4E         401B 47         0000            00140           END
00000 TOTAL ERRORS
```

The resulting characters from DEFM are spread out over the print lines of the listing, along with the location for each. This makes it somewhat difficult to read, but it sure *does* beat assembling the corresponding messages or one byte ASCII data manually.

The DEFS pseudo-op is used to reserve *S*pace in the program. Many times a section of memory must be set aside to be used for a buffer, message area, matrix, or other reserved

```
4A00            00100           ORG     4A00H
4A00 C3CB4A     00110           JP      CONTNU          ;JUMP OVER BUFFER
00C8            00120 BUFFER    DEFS    200             ;BUFFER AREA
4ACB 3E0B       00130 CONTNU    LD      A,11            ;INITIALIZE COUNT
0000            00140           END
00000 TOTAL ERRORS
BUFFER  4A03
CONTNU  4ACB
```

area. Although we could use a series of DEFBs or DEFWs to generate the space by defining zeros or all ones, it is somewhat easier to use the DEFS pseudo-op. The DEFS in the above code reserves 200 bytes of storage between the JP and the LD instruction. (It would be very tedious to use 200 DEFBs or 100 DEFWs to do this, although we might do the same thing by a new ORG.)

The first instruction appears at 4A00H through 4A02H. Then 200 bytes (C8H) of reserved space are requested by the DEFS. The next instruction appears at 4A03H plus C8H, or 4ACBH.

The EQU or *EQU*ate pseudo-op is used to equate a label to a value. The label can then be used at any time, without knowing the value. If we haven't exhausted the usefulness of our first program, let us see how this works in a simple case. The code below *EQU*ates the label ASCONE to the value of ASCII one. Any time we wish to load or otherwise handle an ASCII one after the equate, we can simply use the label ASCONE, instead of having to remember the value or having to load the value from a constant location in memory.

```
4A00            00100           ORG     4A00H
0031            00110 ASCONE    EQU     31H             ;ASCII ONE
4A00 3E31       00120           LD      A,ASCONE        ;LOAD ASCII ONE
0000            00130           END
00000 TOTAL ERRORS
ASCONE  0031
```

Notice that when the ASCII one was referenced it was treated as an immediate value, rather than an address. The immediate load resulted in immediate data of one byte representing ASCONE, or 31H. The label may be an address as

```
4000            00100          ORG     4000H
4C00            00110 BUFFER   EQU     4C00H          ;INPUT BUFFER
4000 21004C     00120          LD      HL,BUFFER      ;POINT TO BUFFER
0000            00130          END
00000 TOTAL ERRORS
BUFFER  4C00
```

well, as in the case of the code above which loads the immediate data value of BUFFER, representing a 16-bit address value, into the HL register pair, thus causing the contents of HL to point to a buffer area.

The last pseudo-op is DEFL. DEFL is similar to EQU in that it sets a label equal to some value or expression. DEFL, however, can be used many times for the same label, while EQU may be used for a label only once in a program. As an example of this, consider the code below. An ASCII "A" has been defined by a DEFL as label ASCA with a value of 41H. In fact, this is an upper case ASCII A. By changing the 6th bit (bit *position* 5) from 0 to 1, the ASCII upper case A may be converted to a lower case A. We'll do this in the program by using DEFL to redefine the value of ASCA as required.

```
4000            00100          ORG     4000H
0041            00110 ASCA     DEFL    'A'
                00120
                00130
4000 3E41       00140          LD      A,ASCA         ;LOAD UPPER CASE
4002 DD7700     00150          LD      (IX+0),A             ;STORE IN BUFFER
                00160
                00170
0061            00180 ASCA     DEFL    'A'+20H        ;CONVERT TO LOWER
4005 3E61       00190          LD      A,ASCA         ;LOAD LOWER CASE
                00200
0000            00210          END
00000 TOTAL ERRORS
ASCA    0061
```

Notice in the above code that ASCA was first recorded by the assembler as a 41H, but when it was redefined by the second DEFL it appears at 61H. (The intervening blank lines represent other code in the program.)

A Mark II Version of the Store "1" Program

We've discussed a lot of concepts in this chapter. Let's try to clarify some of them by writing an expanded version of the program to write a "1" near the center of the screen. In the Mark II version we will write out an entire message to line 9 of the screen. From our earlier analysis, we know that the screen video starts at address 3C00H and ends at 3FFF. We want to start the message at line 9, which is 8*64 characters from the start of the screen memory, or 3C00H + 512. To simplify matters we will write out an entire line of 64 characters. We'll use register pair HL to point to each of the characters in the message and index register IX to point to the next byte of the video display memory. As we write out each character, we'll adjust HL and IX by adding one to point to the next character and next video memory address. To determine when we've reached the end of the message, we'll put a zero at the end and test for zero as we transfer each character. Zero (null) is not a valid ASCII character, so we will know when we have written 64 characters to the screen. The program for this is shown below.

```
                00100 ;MARK II VERSION. WRITES MESSAGE TO LINE 9
                00110 .
4000            00120       ORG     4000H
3C00            00130 VIDEO EQU     3C00H           ;START OF SCREEN VIDEO
4000 21184A     00140 START LD      HL,MESSGE       ;SETUP MESSAGE PNTR
4003 DD21003E   00150       LD      IX,VIDEO+512    ;POINT TO LINE 9
4007 7E         00160 LOOP  LD      A,(HL)          ;GET NEXT CHARACTER
4008 FE00       00170       CP      0               ;TEST FOR END
400A 2809       00180       JR      Z,DONE          ;GO IF DONE
400C DD7700     00190       LD      (IX),A          ;STORE IN VIDEO
400F 23         00200       INC     HL              ;ADD ONE FOR MESSAGE
4010 DD23       00210       INC     IX              ;ADD ONE FOR VIDEO
4012 C3074A     00220       JP      LOOP            ;CONTINUE
4015 C3154A     00230 DONE  JP      DONE            ;ENDLESS LOOP
```

```
4A18 54       00240 MESSGE  DEFM   'THIS IS THE MARK II VERSION

4A19 48       4A1A 49       4A1B 53       4A1C 20       4A1D 49       4A1E 53
   4A1F 20       4A20 54       4A21 48       4A22 45       4A23 20       4A24 4
D       4A25 41       4A26 52       4A27 4B       4A28 20       4A29 49       4A
2A 49       4A2B 20       4A2C 56       4A2D 45       4A2E 52       4A2F 53
  4A30 49       4A31 4F       4A32 4E       4A33 20       4A34 20       4A35 20
     4A36 20       4A37 20       4A38 20       4A39 20       4A3A 20       4A3B
20       4A3C 20       4A3D 20       4A3E 20       4A3F 20       4A40 20
4A41 20       4A42 20       4A43 20       4A44 20       4A45 20       4A46 20
   4A47 20       4A48 20       4A49 20       4A4A 20       4A4B 20       4A4C 2
0       4A4D 20       4A4E 20       4A4F 20       4A50 20       4A51 20       4A
52 20       4A53 20       4A54 20       4A55 20       4A56 20       4A57 20
  4A58 00       00250         DEFB   0
0000          00260         END
00000 TOTAL ERRORS
DONE    4A15
LOOP    4A07
MESSGE  4A18
START   4A00
VIDEO   3C00
```

The first statement puts the origin of the program at 4A00H, the location we have been using all along. VIDEO is equated to 3C00H, the start of the video display area. The next two instructions load HL with a 16-bit value representing the start of the message and load IX with the start of line 9 (3C00H+512). The instruction at LOOP loads the *next* character from the message. To begin with, this is the first letter of the message at 4A18H, but the contents of HL will be incremented by one with each storage of a character. The LD at loop uses register indirect addressing to load the memory location that HL points to. As each character is loaded, the instruction CP 0 tests for a zero byte. A zero has been put at the end of the message to indicate the terminating condition. Notice that no other ASCII character in MESSGE is zero. Normally, the JR Z,DONE will not transfer control to location DONE because the character will *not* be zero and the Z flag will not be set. In every case except the last character the program "falls through" to the instruction at 4A0CH which

stores the character in the location pointed to by the index register IX. In this case the displacement of the index register addressing is zero (the last byte) so the *effective address* is simply the contents of the index register itself. The next two instructions add one to the HL (message) pointer and IX (video) pointer. The jump at 4812H loops back to location LOOP where the process is repeated for the 64 characters of MESSGE. On the 65th character, the byte *is* zero, the Z flag is set on the compare, and the jump to DONE is taken. The instruction at DONE jumps to itself to create an endless loop.

There are many ways that this program could be implemented. Relative jumps could have been used in place of direct jumps in two places, for example, or the loop may have been made more efficient by using other types in instructions. However, as a second program, it is not a bad effort, and employs quite a few of the things we have been discussing in the last three chapters.

This program can be edited, assembled, loaded, and executed in the same manner as the first we discussed, and the reader is urged to do so.

Further Editing and Assembling

We have touched on a few basics in regard to editing and assembling. A complete description of editing modes is covered in the Editor/Assembler manual. The editing functions of the Editor/Assembler permit source lines to be deleted or modified either on a line or character basis and are similar to the EDIT mode of LEVEL II BASIC. There are additional capabilities of the assembler that we haven't discussed, primarily in regard to assembly options such as not producing object code, listings, waiting on errors, and so forth. We will attempt to fill in many of these as we give programming examples in the next chapter, but it would benefit the reader to review the first portion of the Editor/Assembler manual and run some practice examples in both the edit and assembly mode.

In the next chapter we'll cover T-BUG and debugging of programs. T-BUG is used to debug assembled and loaded assembly-language programs, but may also be used to hand assemble and load machine-language programs. If the reader still isn't convinced of the merits of the Editor/Assembler, if he has limited memory, or if he simply likes to do machine-language coding, he will find the chapter very useful.

CHAPTER 5

T-BUG and Debugging

In the past chapters we've learned about the architecture and instruction set of the TRS-80 and something about editing, assembly, and loading of an assembly-language program. The actual sequence of events for an assembly-language program is identical to BASIC programs. The program is first defined by some type of specification—what will the program do and how will the input and output look. The program is then coded. After a desk check, the program is assembled and reassembled if there are assembly errors. When an error-free assembly has been achieved, the resulting program is loaded and executed. Chances are the program will not run the first time, and may not run the fifth time. That's where *debugging* and a debug package, such as T-BUG comes in.

T-BUG is an assembly-language program that can be used to debug assembly-language code, or to enter machine-language code. T-BUG allows the assembly-language programmer to print the contents of locations, to modify locations, to print the register contents, and to debug small segments of code by breakpointing. It would be virtually impossible to debug an assembly-language program without some means to do these things, as each program would have to be completely error free before execution. There are very few programmers that have written a moderately large error-free assembly-language program that ran the first time!

Loading and Using T-BUG

T-BUG is loaded into Level I by the CLOAD command and into Level II by the SYSTEM command with a file name of

"TBUG". After T-BUG has been successfully loaded, a prompt sign of "#" will be present in the left hand corner of the display.

Since we will be debugging all of the programs in the remainder of the book using T-BUG, it is important to know where in RAM T-BUG resides, so that we may avoid that area of memory in assembling programs. Figure 5-1 shows the memory mapping when T-BUG is present. Level II T-BUG occupies 4380H through 4980H, or up to the first A00H (2560) locations of RAM. Level II T-BUG uses an "internal" stack area starting from 4980H, so that no RAM outside of the first 1024 locations will be used by any T-BUG function. We will be safe, then, in assembling our programs to run anywhere in the RAM area above 4A00H. We will use 4A00H as the starting location for all of our programs, giving us 600H (1536) bytes of memory for the reader with 4K of RAM (and who must program in machine language) up to a maximum of 44K for those readers with larger systems.

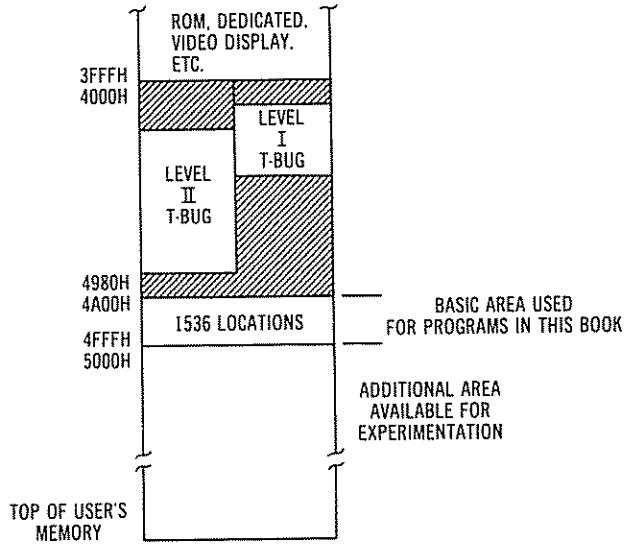


Fig. 5-1. Memory mapping with T-BUG present.

T-BUG Commands

T-BUG has nine commands, all specified by one character, M, X, R, P, L, B, J, F, and G. Some of the commands have arguments (data associated with the command) and some do not.

Let's load T-BUG and examine the commands available. After a successful load the first 16 columns of the video display are cleared and the program displays a "#" at the upper left column of the screen. The format of the M command is #M aaaa where aaaa is a hexadecimal value (can't get away from hexadecimal, can we!) representing the *M*emory location we wish to examine. After the last digit is entered, T-BUG will display a two digit hexadecimal value representing the contents of the memory address. Try the M command with a value of 4A00H. You will get a display of the contents of 4A00.

```
# M 4A00 21
```

Now hit the ENTER key, and you will find that the next location will also be displayed.

```
# M 4A00 21
4A01  FF
```

This process can be continued to display successive locations until either memory or you are exhausted. Hitting an "X" at any time terminates the memory display function and brings you back to the monitor prompt again.

```
# M 4A00 21
4A01  FF
4A02  FF
4A03  FF
#
```

Entering data in place of ENTER will change the memory location to the values entered. Two hexadecimal digits must be entered. Let's go back to the original (Mark I) version of the program to write a "1" near the center of the screen. The program is shown below.

```
4A00            00100          ORG     4A00H          ;START AT LOCATION 4A00H
4A00 3E31       00110          LD      A,31H          ;LOAD A REGISTER WITH "1"
4A02 32203E     00120          LD      (3E20H),A      ;STORE "1" INTO CENTER
4A05 C3054A     00130 LOOP     JP      LOOP           ;LOOP HERE
4A08            00140          END     4A00H          ;END-START OF 4A00H
00000 TOTAL ERRORS
LOOP    4A05
```

Starting at location 4A00H, let's enter the machine code for that program. The entry will look something like this at the end.

```
# M 4A00 3E
4A01  FF 31
4A02  FF 32
4A03  FF 20
4A04  FF 3E
4A05  FF C3
4A06  FF 05
4A07  FF 4A
#
```

We can now go back and check the locations by the M command to verify that all data has been entered correctly. This is really not so much a check on machine malfunction as it is on *operator* malfunction.

The J, or *J*ump, command in T-BUG allows the user to transfer control to a location for execution. Specifying J aaaa causes the monitor to jump to location aaaa, where aaaa is again a hexadecimal four-digit value. We could at this point perform a J 4A00 to execute the Mark I version of the program. If we do that, however, the program will be "hung up" in an endless loop to itself at location 4A05. The only way to get out of the loop in Level I is to reload T-BUG; in Level II T-BUG may be reentered by a SYSTEM transfer to 4380H (17280), but the recovery is still a nuisance.

The B command allows us to execute a program up to a point where control is returned to the T-BUG monitor, thus keeping the debugging from becoming a series of recovery procedures as it goes off into cloud cuckoo land. The B command establishes a *breakpoint*. At the breakpoint location control is returned to the monitor where locations or registers may be examined, a new breakpoint may be established a little further on, and the progress of the program may be checked.

Let us see how the *B*reakpoint operates. In this simple program, suppose that we want to stop at location 4A02 to verify that 31H did in fact get loaded into the A register. The B command would be

```
# B 4A02
```

Now we could execute the jump to 4A00H to start execution by

```
# J 4A00
```

The instruction at 4A00 would then be executed. After this instruction the breakpoint would be encountered at 4A02 (an instruction returning control to the T-BUG monitor) and *T-BUG would be reentered, with a display of the # in the* upper left-hand corner.

After the breakpoint, the first order of business is to execute an F command. Entering an F restores the instruction

that was temporarily replaced by the breakpoint. Entering a breakpoint address causes the monitor to place a CALL instruction to the breakpoint-handling routine in T-BUG. Since the CALL is three bytes long, it replaces the three bytes in the program at the breakpoint instruction. The three bytes of the program must be restored before proceeding and the F accomplishes this.

Before proceeding the user can now examine memory locations or cpu registers to see what program actions have occurred. About the only thing that can be verified here is that 31 was indeed loaded into the A register. We can examine the A registers and all cpu registers by using the T-BUG R command (*R*egister). The R command causes a display of all cpu registers in the format shown in Figure 5-2. In our case the display might look like the following.

```
# FFFF FFFF
  FFFF FFFF
  3142 00FD
  41E9 43E0
  FFFF FFFF
  4980 4A02
```

The 31 in the A register position indicates that the A register was properly loaded.

To continue from this point, another breakpoint must be put into the program a little further on. In our program the next breakpoint will be at 4A05 to prevent an endless loop. Rather than a J command to resume execution, however, a G (*G*o) should be used. The G command will cause resumption of the program at the breakpointed instruction (4A02), with

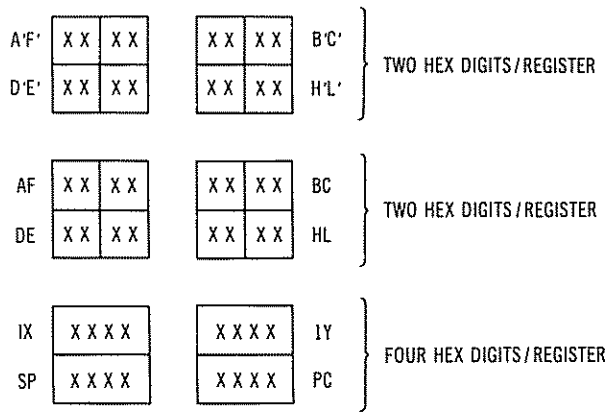


Fig. 5-2. T-BUG R command format.

all registers properly restored and no ill effects from having had the breakpoint. If the reader will execute the following sequence he will see the "1" written out to the center of the screen, followed by the "#" for the 4A05H breakpoint at the upper left hand corner of the screen.

```
# F          (to restore the 4A02 area)
#B 4A05      (to set a new breakpoint)
# G          (to resume execution)
```

What if the user had wanted to change the contents of a register location before proceeding from a breakpoint. This is certainly possible, and necessary in debugging. The procedure *is* somewhat complicated, however. To change the contents of a register, a memory location representing the current contents of that register must be changed. The memory locations representing all of the cpu registers are shown in Figure 5-3. To change the D register, for example, memory

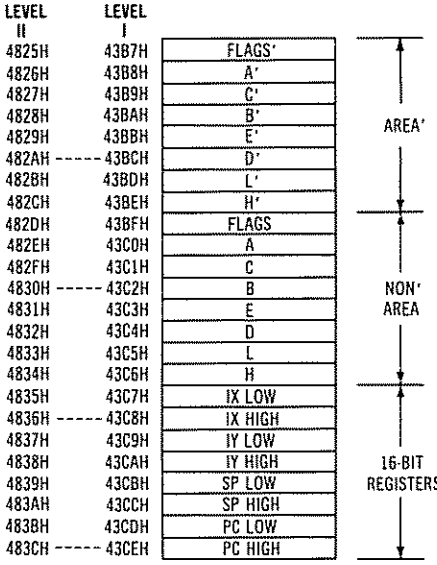


Fig. 5-3. T-BUG register locations.

location 43C4H must be examined by an M command and new data entered. To change a 16-bit register, two memory locations must be changed, representing the high-order byte and the low-order byte of that register, as shown in the figure. To change the IY register to 4FE3H, for example, memory location 43CAH must be changed to 4FH, and memory location 43C9H must be changed to E3H. After the change in one

or more registers, a J (jump) or G (go) command must be executed to effect the change.

T-BUG Tape Formats

As we mentioned earlier, debugging assembly-language programs is a major part of the assembly-language programming process. The object is to find as many bugs as possible before reassembling the program to reduce the time spent in editing and reassembling. As each bug is found it *may* be corrected in machine language, if the user knows the instruction formats and addressing modes (see, we told you that it would help to get a background in the instruction set). Of course the user can avoid this approach and simply reassemble the program each time bugs are found.

As a simple example of this *patching* technique, let's go back to the code we've entered for the Mark I version of the screen output routine. Suppose that we had found that instead of writing a "1" to the screen, we should have written an "*". Using T-BUG it is a simple matter to change the 31H in the second byte of the first instruction to the code for an asterisk, 2AH.

```
# M 4A01 31 2A        (hit X)
#
```

Suppose that we had wanted to *insert* code between two existing instructions. That is a little more difficult to patch, but still possible. If we had wanted to store one "1" in both the 32nd and *31st* character positions we could patch in the instruction to store in 3E1FH (31st position) by putting a jump to a *patch area* at 4A02, jumping out to the patch area, performing the store in 3E1F, performing the store in 3E20H (destroyed by the jump), and then jumping back to the instruction at 4A05. Of course, the patch area should be in an area of memory unused by our program or by T-BUG. The patches for this are shown below.

```
4A00  3E     (original LD A,31H)
4A01  31
4A02  C3     (patched JP 4B00)
4A03  00
4A04  4B
4A05  C3     (original JP 4A05)
4A06  05
4A07  4A
```

```
4B00  32    (restored store to 3E20H)
4B01  20
4B02  3E
4B03  32    (new store to 3E1FH)
4B04  1F
4B05  3E
4B06  C3    (return to program)
4B07  05
4B08  4A
```

Patching to correct errors can be done as often as required until it reaches the point where the programmer does not know which areas have been patched and which have not. The user can quickly determine his own requirements for reassembly of a patched program.

To provide a means to save patched programs, or to provide a means to save any machine-language program, T-BUG has two additional commands, P for *P*unch tape, and L for *L*oad tape. The P command writes any specified area in memory to cassette tape. The resulting tape format can be read by T-BUG or by the SYSTEM command in LEVEL II. To save locations 4A00H through 4B08H, for example, the command

```
# P 4A00 4B08
```

would be entered for LEVEL I. Level II requires two more arguments, one for the entry point (start) and one for the file name (up to six characters). The level II format might be

```
# P 4A00 4B08 4A00 MARK1      (ENTER)
```

After the command is entered, T-BUG writes out the specified area and includes the entry point and file name for Level II. The format used for Level I write is shown in Figure 5-4.

Once the T-BUG cassette tape has been written it may be loaded at any time by the L command (or the SYSTEM command in Level II). The L command has no arguments, and the tape will start loading after the L has been typed. Tape loading is indicated by the usual asterisk in the lower left hand corner. Successful loading is indicated by the "#" prompt; an error in loading the data will result in an "E" after the load command. The format used for assembly object output is the same as T-BUG's, so that T-BUG may be used to load object tapes produced during assembly.

The above describes the T-BUG commands and their typical use. The reader is urged to experiment with T-BUG as we will be using it in following chapters for debugging purposes. A reference list of T-BUG commands follows.

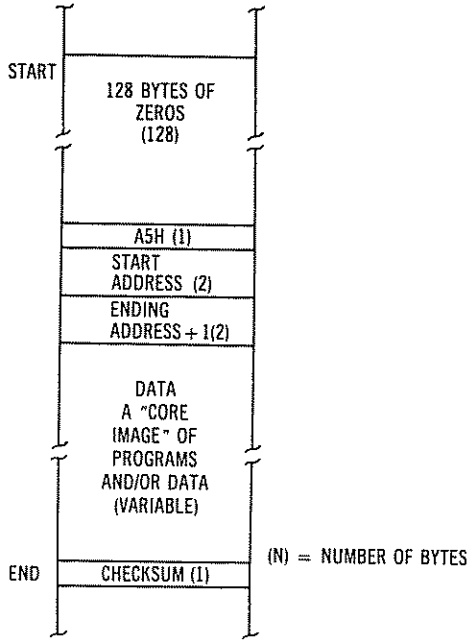


Fig. 5-4. T-BUG tape format.

Format	*Description*
# M aaaa	Display location aaaa
ENTER (after M)	display next location
X (after M, J, B, P)	Exit operation
# R	Display registers
# P aaaa bbbb (Level I)	Write cassette from aaaa through bbbb
# P aaaa bbbb cccc NAME (Level II)	Write cassette from aaaa through bbbb with starting address cccc and file name NAME
# L	Load a T-BUG tape
# B aaaa	Set breakpoint
# F	Restore instruction after breakpoint
# G	Continue from breakpoint
# J aaaa	Jump to location aaaa

Standard Format in Following Chapters

The program in the following chapters will illustrate the use of Z-80 instructions in accomplishing certain types of operations. All code will be assembled starting at location 4A00H, so that T-BUG may be used to debug or investigate the actions of the programs discussed. At the reader's option, these programs may be assembled and loaded using T-BUG, and then debugged, or the machine-language code for the program may be entered using T-BUG without assembly. The RAM area available for patching, buffers, or other use is located from 4A00H through 4FFF for minimum 4K RAM systems, or from 4A00H through "top of memory" for larger systems.

SECTION II

Programming Methods

CHAPTER 6

Moving Data in Bytes, Words, and Blocks

This chapter will discuss ways in which to move data from the cpu to memory, between cpu registers, from memory to cpu, and from one area of memory to another. At first glance this might not seem like such an exciting topic, but the addressing concepts practiced here can be applied to many of the other instructions covered in later chapters. In addition, the block move instructions are interesting instructions that are not found in other 8-bit microprocessors. They are some of the most powerful features of the Z-80.

Byte and Word Moves

We have already seen examples of loading and storing data into single or double registers in the Z-80. Eight-bit loads can be accomplished by immediate loads or by loading operands from memory. Suppose that we want to load *all* of the cpu registers except for the PC (program counter) with 8 bits of data. Remember that there are fourteen general purpose cpu registers, seven in each set of prime and non-prime registers, and three 16-bit or two-byte registers, the IX, IY, and SP. We'll ignore the I and R registers as these are not generally used except for interrupt handling and refresh operations.

Let's consider the 8-bit general-purpose registers first. We would like to write a program to load the registers as follows:

A register	Loaded with 9
B	11
C	12
D	13
E	14
H	15
L	16
A′	1
B′	3
C′	4
D′	5
E′	6
H′	7
L′	8

Loading these values into the cpu registers with immediate values is easy, because all cpu registers may be loaded by an immediate data value. The only trick here is swapping register sets. The swap is done by the EX AF,AF′ instruction, which swaps the two A registers and the flag, and the EXX instruction which swaps all other registers. The main question (raised from the back of the room again, I see) is which set is which? It is up to the programmer to keep track of which of the two sets of registers he is using. When the TRS-80 is powered up the non-prime set is *active;* performing one or both of the exchange instructions switches the cpu to the other set. The primed set is simply the set of registers that is not currently active, and the program must keep track of which set is being used, not unlike remembering which of two book ends you've hidden a ten-dollar bill under. The following program loads all general-purpose cpu registers with the indicated values above. Put a breakpoint at END, jump to START, and then display the registers by an R command in T-BUG, and you should see a sequence of 00 through 10H displayed for the general-purpose registers. The IX, IY, SP, and PC will hold meaningless values.

```
                00100 ;ROUTINE TO LOAD ALL REGISTERS
4000            00110           ORG     4000H
4000 3E09       00120 START     LD      A,9
4002 0608       00130           LD      B,11
4004 0E0C       00140           LD      C,12
```

```
4006 1600      00150        LD      D,13
4008 1E0E      00160        LD      E,14
400A 260F      00170        LD      H,15
400C 2E10      00180        LD      L,16
400E 08        00190        EX      AF,AF'          ;SWITCH ACTIVE REGS
400F D9        00200        EXX                     ;SWITCH ACTIVE REGS
4010 3E01      00210        LD      A,1
4012 0603      00220        LD      B,2
4014 0E04      00230        LD      C,4
4016 1605      00240        LD      D,5
4018 1E06      00250        LD      E,6
401A 2607      00260        LD      H,7
401C 2E08      00270        LD      L,8
401E 08        00280        EX      AF,AF'          ;SWITCH BACK
401F D9        00290        EXX                     ;SWITCH BACK
4020 C32040    00300 END    JP      END             ;LOOP HERE
0000           00310        END
00000 TOTAL ERRORS
END     4020
START   4000
```

Now suppose that we would like to load constants from memory instead of immediate values. (Don't ask why, kid, just do it!) There are two ways to handle this approach, as we explained in an earlier chapter. One way would be to set up HL, DE, BC, IX, or IY to point to the constants to be loaded and to then load in the values using either register indirect addressing or indexing. This would work fine if the data were grouped in a *contiguous* area, but would require setting up a new value in the pointer register for each load if the constants were scattered over different locations in memory. The second approach, which we'll implement in the following program uses the A register as a pipeline to channel data from the constants in memory to each of the cpu registers.

```
                00100 ;USING A AS A PIPELINE
4000            00110        ORG     4000H
4000 3A0F40     00120 START  LD      A,(ELEVEN)      ;11
```

```
4A03 47      00130        LD    B,A          ;MOVE TO B
4A04 3A104A  00140        LD    A,(TWELVE)   ;12
4A07 4F      00150        LD    C,A          ;MOVE TO C
4A08 3A114A  00160        LD    A,(THIRTN)   ;13P
4A0B 57      00170        LD    D,A          ;MOVE TO D
4A0C C30C4A  00180 LOOP   JP    LOOP         ;LOOP HERE
4A0F 0B      00190 ELEVEN DEFB  11
4A10 0C      00200 TWELVE DEFB  12
4A11 0D      00210 THIRTN DEFB  13
0000         00220        END
00000 TOTAL ERRORS
LOOP   4A0C
THIRTN 4A11
TWELVE 4A10
ELEVEN 4A0F
START  4A00
```

Storing 8-bit data works in pretty much the same fashion as loading cpu registers. The general registers can always be stored by using a register pair as an indirect pointer, but only the A register can be loaded directly from memory. If we were to store the contents of the cpu registers back into the constant locations in memory, the register pair or index register used as the pointer would have to be set up with the new location each time a store was performed, as shown in the program below. The reader may care to execute this program directly after the load program to verify that the registers have been stored. Zero ELEVEN, TWELVE and THIRTN after the load breakpoint, put in a new breakpoint at 4A1EH, and jump to 4A12H to perform the store. (Don't forget the "F" after each breakpoint to restore the instruction.)

```
                00100 ;CODE TO STORE REGISTERS
4A12            00110        ORG   4A12H        ;NEXT LOC AFTER LOAD CODE
4A12 7A         00120 START  LD    A,D          ;13
4A13 32114A     00130        LD    (THIRTN),A   ;RESTORE
4A16 79         00140        LD    A,C          ;12
4A17 32104A     00150        LD    (TWELVE),A   ;RESTORE
```

```
4A1A 78         00160           LD      A,B             ;11
4A1B 320F4A     00170           LD      (ELEVEN),A      ;RESTORE
4A1E C31E4A     00180 LOOP      JP      LOOP            ;LOOP HERE ON END
4A11            00190 THIRTN    EQU     4A11H
4A10            00200 TWELVE    EQU     4A10H
4A0F            00210 ELEVEN    EQU     4A0FH
0000            00220           END
00000 TOTAL ERRORS
LOOP    4A1E
ELEVEN  4A0F
TWELVE  4A10
THIRTN  4A11
START   4A12
```

Sixteen bits of data are somewhat harder to move around. Register pairs can be stored directly to memory, may be stored in the stack by PUSHes (covered in a later chapter), or may be transferred by using the HL register pair as a routing point. Storing the register pairs in memory is not generally something that is commonly required. Loading 16-bit data into register pairs can be handled by immediate loads for constants, by direct loading of register pairs, and by routing other 16-bit data through HL. A common trick in loading *two* single registers with two separate operands is to perform an immediate load of a register pair. This only works, of course, when the two single registers involved happen to be in the same register pair. The resulting instruction sequence is much shorter than 8-bit loads.

```
                00100 ;LOADING SINGLE REGISTERS WITH 16-BIT LOADS
4A00            00105           ORG     4A00H
4A00 01C201     00110 START     LD      BC,256+2        ;LOAD B WITH 1,C WITH 2
4A03 110403     00120           LD      DE,768+4        ;LOAD D WITH 3,E WITH 4
4A06 C3064A     00130 LOOP      JP      LOOP            ;LOOP HERE FOR EP
0000            00140           END
00000 TOTAL ERRORS
LOOP    4A06
START   4A00
```

Transferring data between two register pairs is almost always done by PUSHing the first register pair and POPping the second to transfer data from the first into the second. To load HL with the contents of BC, for example, the instructions

```
PUSH  BC    ;BC TO STACK
POP   HL    ;RETRIEVE BC, PUT IN HL
```

would be performed.

Filling or Padding

All of the foregoing is fairly abstract, even when T-BUG is being used to verify the results of the code. Get ready for some spectacular visual effects! Fortunately for us and especially that reader who keeps nodding off, moving identical data to fill buffer areas or to initialize tables may be observed on the display. After all, the display is simply additional memory dedicated to the 1024 characters or 6144 *pixels* of a display.

Let's illustrate two methods of addressing in a routine to *fill* data. In this routine, a specified data byte from 0 to 255 (OH-FFH) is written into a memory area from a starting address to an ending address. The fill function is frequently used to "zero" portions of memory, to fill tables with −1 (FFH), or to *pad* character lines with blanks (20H).

The fill character will be in the A register, while the HL register will be set up with the starting address of the memory area to be filled. We could specify either an ending address or the number of bytes to fill. Specifying an ending address would require that we have a 16-bit address that could be used to compare the current fill location with the ending address. The second approach would use a count in one of the registers that would be decremented with each filled byte. When the count reached zero the fill would be over. If a single register were used, the count could be 0 through 255. If we wanted to fill more than 255 bytes we would have to use a register pair, which could specify a fill count of 0 through 65535, which would certainly be adequate for a 64K system! In the following example of the fill we'll try the second approach; we'll put the fill count in a single register. The parameters will be in the registers before the fill starts as shown below.

```
(A)  = character to be filled
(HL) = starting address for the fill
(B)  = number of characters to be filled from 1 to 256
```

```
                00100 ;THIS IS A PROGRAM TO FILL MEMORY FROM
                00110 ;A STARTING ADDRESS FOR A NUMBER OF BYTES
                00120 ;  (A)=BYTE TO BE FILLED
                00130 ;  (HL)=STARTING ADDRESS
                00140 ;  (B)=NUMBER OF BYTES
                00150 ;
4A00            00160        ORG    4A00H       ;START OF PROGRAM
4A00 3E2A       00170 START  LD     A,'*'       ;FILL WITH ASTERISKS
4A02 21003C     00180        LD     HL,3C00H    ;START OF SCREEN
4A05 0600       00190        LD     B,0         ;FILL 256 BYTES
4A07 77         00200 LOOP1  LD     (HL),A      ;FILL BYTE
4A08 23         00210        INC    HL          ;INCREMENT POINTER
4A09 05         00220        DEC    B           ;DECREMENT COUNT
4A0A 20FB       00230        JR     NZ,LOOP1    ;GO IF NOT DONE
4A0C 18FE       00240 LOOP2  JR     LOOP2       ;LOOP HERE ON DONE
4A00            00250        END    START
00000 TOTAL ERRORS
LOOP2   4A0C
LOOP1   4A07
START   4A00
```

The first thing that is done in the program is to load A with the data (asterisk in this case) and to load the HL register pair with the starting address of the memory area to be loaded. To enable us to see the results we're using the start of the screen video at 3C00H. The B register is loaded with the number of bytes to be filled. If we had specified 1 through 255 bytes that number would have been filled with asterisks. Specifying zero, however, fills *256* bytes, as we shall see below. LOOP1 through the JR NZ,LOOP1 makes up the *main loop* in the program. For each *iteration* or pass through the loop one byte of data is filled. Initially the byte at 3C00H is filled. Each time through the loop, however, the HL register pair is incremented by one to point to the next memory byte, and the B register is decremented by one to count down. If the count in B has not reached zero, the Z flag is not set by the decrement, and the conditional branch at 4A0AH is taken. If the count has reached zero, the program falls through and the loop at LOOP2 is reached. Notice that the jumps here are

two-byte relative jumps. If we started with a count of zero, the count after the decrement of B is 11111111, as you will see if we subtract a one from eight zeros on paper. Starting with a count of zero, therefore, causes a fill of 256 bytes.

To run the program, assemble and load using SYSTEM or T-BUG. If no breakpoint is used, the program will fill the first four lines of the screen with asterisks. The reader may wish to try other values for the fill by changing the 2AH at 4A01H, or may change the fill area by changing the 3C00H at 4A03H and 4A04.

An Unsophisticated Block Move

Often it is necessary to move data from one block of memory to another block of memory. One example of this would be moving a string of characters that have been input to the screen display area. Another example might be inserting data in a table. The data below the inserted entry would have to be moved down to make room for the new data.

In the next program we'll be implementing some code to move one block of memory to another. We'll use register indirect addressing to accomplish this feat. Register pair HL will point to the *source* block and register pair DE will point to the *destination block*. Register pair BC will contain a count of the number of bytes to be moved. As BC may hold 0 through 65535, any size block up to maximum memory size may be

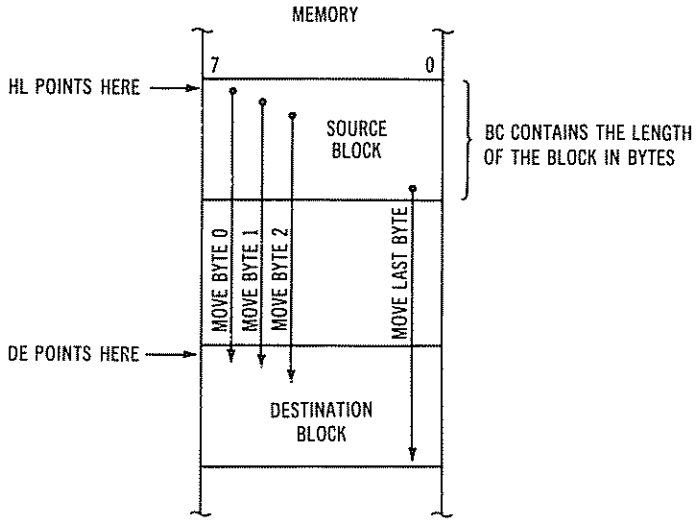


Fig. 6-1. An unsophisticated block move.

moved. Figure 6-1 shows the manner in which the move will be done.

We know that using the HL register pair as a pointer will work with any cpu register. Using BC or DE as a pointer is only useful for loading and storing the A register, however, so all data to be transferred must go through the A register.

```
                    00100 ;THIS PROGRAM WILL MOVE
                    00110 ;A BLOCK OF MEMORY FROM
                    00120 ;ONE AREA TO ANOTHER
                    00130 .
4900                00140          ORG     4900H
4900 210000         00150 START    LD      HL,0          ;SOURCE
4903 11003C         00160          LD      DE,3C00H      ;DESTINATION
4906 01E803         00170          LD      BC,1000       ;1000 BYTES
4909 7E             00180 LOOP1    LD      A,(HL)        ;GET SOURCE BYTE
490A 12             00190          LD      (DE),A        ;STORE
490B 23             00200          INC     HL            ;POINT TO NEXT SOURCE
490C 13             00210          INC     DE            ;POINT TO NEXT DE
490D 0B             00220          DEC     BC            ;DECREMENT COUNT
490E 78             00230          LD      A,B           ;GET MS COUNT
490F B1             00240          OR      C             ;MERGE LS COUNT
4910 20F7           00250          JR      NZ,LOOP1      ;GO IF COUNT NOT 0
4912 18FE           00260 LOOP2    JR      LOOP2         ;LOOP HERE ON DONE
0000                00270          END
00000 TOTAL ERRORS
LOOP2   4912
LOOP1   4909
START   4900
```

The resulting program is shown. Before the loop, HL is loaded with 0, the start of the source block, and DE is loaded with 3C00H, the start of the screen area for the destination. The BC register pair is loaded with the number of bytes to be transferred, in this case 1000. If the program works the way we want it to the first 1000 locations from 0H through 03E7H will be transferred from the ROM BASIC interpreter to the screen. What should we see on the screen? In a program

such as the BASIC interpreter there is a mix of relatively random data. Some of the data will coincidentally represent ASCII characters while some of the data will be actual BASIC messages, such as "MEMORY SIZE". Other data will represent (coincidentally) graphics data of different types. When we actually run the program, then, we'll largely see random patterns, but some messages.

The main loop of the program starts at LOOP1. The first thing that is done is to load a byte into A using HL as a register pointer. The source byte in A is then stored by using DE as the destination pointer. HL and DE are then incremented to point to the next source and destination byte. The count is then decremented by one. If the count is not zero, the program loops back to LOOP1, otherwise the program falls through to LOOP2. Now let's look at the way in which we test for a count of zero. While decrementing a *single* register sets the zero flag if the count decrements to zero, decrementing a register pair sets *no* flags. Why? That's just the way the instructions work. (Never try to be *too* logical with a given instruction set on any computer.) The BC register pair is tested for zero by effectively ORing the B and C registers together. Remember that the A register must be used for an OR operation, and that the OR of any two bits produces a one if either of the bits is a one. If *no* bits are a one then the result is zero in this case, and the zero flag is set. The only time no bits in either the B or C registers will be ones is when the count in BC is zero and hence we have our test.

Have you run the program yet? If you do, you'll find an interesting display of some of the secrets of the Radio Shack interpreter, displayed in living black and white on your TRS-80 screen. Try changing the source address, destination address, and byte count to display different areas of memory. Be careful not to overwrite the program itself or T-BUG, however. Keep the destination from pointing toward the 4000H through 4A00H area!

While the above program is perfectly fine for an 8080A (sniff!), one simply wouldn't want to run such a gaucherie on a Z-80.

An Elegant Block Move

The block move instructions on the Z-80 take the entire code from 4A09H through 4A11H in the above program and reduce it to one instruction! This *is* truly an elegant instruction. The Z-80 instruction for this is the LDIR instruction.

If we recode the program above to work with the LDIR, we come up with the program below.

```
                00100 ;THIS IS AN ELEGANT VERSION OF
                00110 ;A BLOCK MOVE
                00120 .
4A00            00130          ORG     4A00H
4A00 210000     00140 START    LD      HL,0            ;SOURCE
4A03 11003C     00150          LD      DE,3C00H        ;DESTINATION
4A06 01E803     00160          LD      BC,1000         ;1000 BYTES
4A09 EDB0       00170 LOOP1    LDIR                    ;OK,BUD,MOVE IT!
4A0B 18FE       00180 LOOP2    JR      LOOP2           ;LOOP HERE AT END
0000            00190          END
00000 TOTAL ERRORS
LOOP2   4A0B
LOOP1   4A09
START   4A00
```

As you can see in the program, the LDIR must have the HL, DE, and BC register pairs initialized to the source address, destination address, and byte count, respectively. Then it goes off looping to itself automatically until the byte count reaches zero. It would be interesting for the reader to examine the registers after the LDIR. We would find that HL and DE point to the last byte transferred *plus one* and that register pair BC contains 0.

The LDDR instruction works the same way as the LDIR instruction except that the register pairs are set up to the *end* of the source block, the *end* of the destination block, and the number of bytes to be transferred. Data is transferred from end to start in the LDDR, as shown in Figure 6-2.

There are two other block move instructions in the Z-80 instruction set, the LDI and the LDD. They operate exactly the same way as the LDIR and the LDDR, except that as each byte is transferred, the instruction pauses and the next instruction is executed. The program must check for the terminating condition of zero count in the BC register pair. The LDI instruction code that follows is identical to the operation of the LDIR, except that the test for BC=0 is done *externally* to the LDI.

```
START   LD    HL,0          ;SOURCE
        LD    DE,3C00H      ;DESTINATION
        LD    BC,1000       ;1000 BYTES
LOOP    LDI                 ;TRANSFER ONE BYTE
        JP    PE,LOOP       ;CONTINUE IF BC NOT 0
```

One would expect the Z flag to be set when the byte count in BC is decremented down to zero. This is not the case, however, in either the LDI or LDD. The parity/overflow flag is the one that is set after each transfer. When BC has reached zero, the parity/overflow flag will be reset (PO mnemonic), otherwise it will be set (PE mnemonic). The conditional jump, therefore, is done on overflow set, or "parity even."

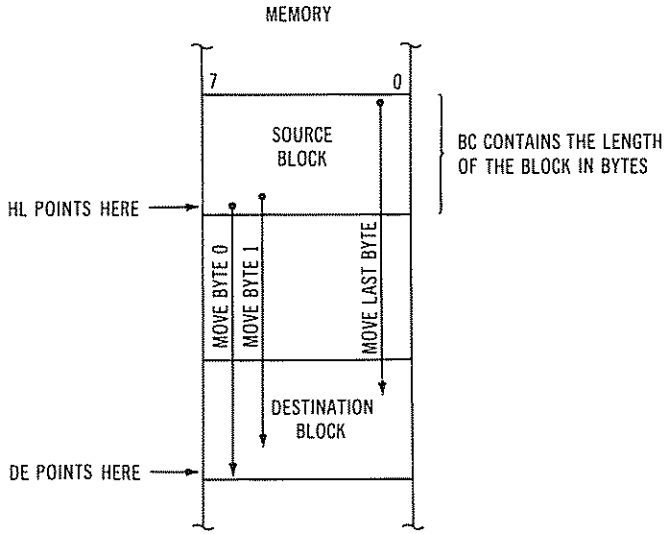


Fig. 6-2. Data transfer for an LDDR.

The LDI and LDD are used when the block transfer action is required, but when there must be intermediate processing between the transfer of individual bytes. Examples of this would be a transfer of a block of data *until* a terminating character such as line feed or null was reached, or transfer of data *except* for lower case characters.

To illustrate this intermediate processing, and to give the reader a graphic example of how the LDI, LDD, LDIR, and LDDR transfer data, we have coded the following program. This program *slowly* transfers a block to video memory in forward fashion, and then transfers another block in back-

ward fashion. Subroutine SLOWLY is used to slow down the transfer by a timing loop after each byte.

```
                    00100 ;A GRAPHIC EXAMPLE OF BLOCK MOVES
                    00110 .
4A00                00115       ORG     4A00H
4A00 210000         00120 START LD      HL,0            ;SOURCE
4A03 11003C         00130       LD      DE,3C00H        ;DESTINATION
4A06 010004         00140       LD      BC,1024         ;FULL SCREEN
4A09 EDA0           00150 LOOP1 LDI                     ;XFER ONE BYTE
4A0B E2144A         00160       JP      PO,NXT          ;GO IF OVER
4A0E CD284A         00170       CALL    SLOWLY          ;DELAY
4A11 C3094A         00180       JP      LOOP1           ;CONTINUE
4A14 1B             00190 NXT   DEC     DE              ;PNT TO LAST SCREEN
4A15 21FF07         00200       LD      HL,7FFH         ;NEW BLOCK
4A18 010004         00210       LD      BC,1024         ;STILL 1024 BYTES
4A1B EDA8           00220 LOOP2 LDD                     ;XFER BACKWARDS
4A1D E2264A         00230       JP      PO,DONE         ;GO IF DONE
4A20 CD284A         00240       CALL    SLOWLY          ;WHOA
4A23 C31B4A         00250       JP      LOOP2           ;CONTINUE
4A26 18FE           00260 DONE  JR      DONE            ;ENDLESS LOOP
4A28 3E99           00270 SLOWLY LD     A,99H           ;TIMING CNT
4A2A 3D             00280 SLOW10 DEC    A               ;A-1 TO A
4A2B 20FD           00290       JR      NZ,SLOW10       ;GO IF NOT DONE
4A2D C9             00300       RET                     ;RETURN
0000                00310       END
00000 TOTAL ERRORS
SLOW10  4A2A
DONE    4A26
LOOP2   4A1B
SLOWLY  4A28
NXT     4A14
LOOP1   4A09
START   4A00
```

The first four instructions of this routine are identical to the code above. If the P/V bit is set, subroutine SLOWLY is

called before the next byte is transferred. When the last byte has been transferred, the P/V bit is reset and the jump is taken to NXT. At NXT the DE register is decremented to point to the last screen location; it held 4000H before the decrement. The address of the last location in the second 1024 bytes of ROM (7FFH) is put into HL as the first source address. At LOOP2 an LDD is used to transfer the data from 7FFH through 400H to the screen video memory, with the SLOWLY delay between each byte. Subroutine SLOWLY simply sets the immediate value 80H (128) into A and then decrements the count, looping until the count reaches zero. *Register A was used to hold the count as all other registers were dedicated to functions used by the LDI or LDD.*

To tie together some of the concepts we have explored in this chapter, we'll conclude with two general-purpose routines, FILL, a routine to fill any character in any sized-block in memory, and MOVE, a routine to move any block in memory anywhere else.

FILL Subroutine

The FILL subroutine is modeled after the one discussed earlier in this chapter. It is CALLed with certain registers loaded with *parameters* to be used for the fill.

(D) = Byte to be filled, any value
(HL) = Start of memory area to be filled
(BC) = Number of bytes to fill

Upon return from the subroutine, the contents of BC are zero, the fill byte remains in D, and HL points to the last byte filled plus one. The contents of A have been zeroed.

```
00100 ;SUBROUTINE TO FILL DATA IN MEMORY
00110 ;   ENTRY:(D)=DATA TO BE FILLED
00120 ;         (HL)=START OF FILL AREA
00130 ;         (BC)=# OF BYTES TO FILL
00140 ;         CALL  FILL
00150 ;   EXIT: (D)=SAME
00160 ;         (HL)=END OF FILL+1
00170 ;         (BC)=0
00180 ;         (A)=0
00190 ;
```

```
4000            00200           ORG     4000H
4000 72         00210 FILL      LD      (HL),D          ;STORE BYTE
4001 23         00220           INC     HL              ;BUMP POINTER
4002 0B         00230           DEC     BC              ;ADJUST COUNT
4003 78         00240           LD      A,B             ;GET MS OF COUNT
4004 B1         00250           OR      C               ;MERGE LS COUNT
4005 20F9       00260           JR      NZ,FILL         ;CONTINUE IF DONE
4007 C9         00270           RET                     ;RETURN IF DONE
0000            00280           END
00000 TOTAL ERRORS
FILL    4000
```

MOVE Subroutine

The MOVE subroutine uses either an LDIR or an LDDR. The subroutine automatically checks to see whether the movement should be forward or backward. Ordinarily this is no problem, but when the source and destination blocks overlap, the reader can see that there is a conflict if the wrong direction is used; data will be destroyed before it has been moved to the new area. On entry into the subroutine, the following registers are set up.

(HL) = Start of source memory area
(DE) = Start of destination memory area
(BC) = Number of bytes to be moved

Upon return from the move, the contents of BC are zero, and the two other register pairs point to the last locations plus one.

```
                00100 ;SUBROUTINE TO MOVE MEMORY
                00110 ;   ENTRY:(HL)=SOURCE START
                00120 ;         (DE)=DESTINATION START
                00130 ;         (BC)=# OF BYTES TO MOVE
                00140 ;    EXIT:(HL)=SOURCE AREA+1
                00150 ;         (DE)=DEST AREA+1
                00160 ;         (BC)=0
                00170 ;
4000            00180           ORG     4000H           ;CHANGE ON REASSEMBLY
4000 E5         00190 MOVE      PUSH    HL              ;SAVE SOURCE PNTR
```

```
4A01 B7       00200          OR    A          ;CLEAR CARRY
4A02 ED52     00210          SBC   HL,DE      ;SOURCE-DEST PNTRS
4A04 E1       00220          POP   HL         ;RESTORE PNTR
4A05 DA0C4A   00230          JP    C,MOV10    ;GO IF MOVE BACK
4A08 EDB0     00240          LDIR             ;MOVE FORWARD
4A0A 1808     00250          JR    MOV20      ;GO TO RETURN
4A0C 09       00260 MOV10    ADD   HL,BC      ;POINT TO END+1
4A0D 2B       00270          DEC   HL         ;POINT TO END
4A0E EB       00280          EX    DE,HL      ;SWAP
4A0F 09       00290          ADD   HL,BC      ;POINT TO END+1
4A10 2B       00300          DEC   HL         ;POINT TO END
4A11 EB       00310          EX    DE,HL      ;SWAP BACK
4A12 EDB8     00320          LDDR             ;MOVE BACK
4A14 C9       00330 MOV20    RET              ;RETURN
0000          00340          END
00000 TOTAL ERRORS
MOV20  4A14
MOV10  4A0C
MOVE   4A00
```

Subroutine Format

FILL and MOVE follow the general format that will be used for subroutines in this book. All of the subroutines are assembled at 4A00H. To use them in other areas of memory it is generally mandatory to reassemble them with the proper ORG. Occasionally some of the subroutines will be *relocatable;* the subroutine would have identical machine code no matter what the origin. For this to be possible, the subroutine could not have direct addressing instructions such as JPs, CALLs, direct memory loads and stores, and so forth. In these cases the machine code could be moved without reassembly.

We will start building up a number of general-purpose subroutines in these chapters for the reader to use in his own programs. They'll be presented in the appropriate chapter and collected together in the last section of the book. FILL and MOVE are the first two of the lot.

Stack Operation

In the sample programs that we have been using up to this point we haven't been too concerned about the *stack*. The stack has been in use, however, and at this point it is best to pay some attention to it before it turns on us some day and devours some of our programs.

Every time we execute a CALL, RETurn, PUSH, or POP, we have been storing data into or removing data from the stack. For the sample programs here, we have been using the stack found in T-BUG, which is a short section of memory contained within the T-BUG program area. The stack *can* be located anywhere in RAM memory that we choose, however, as long as it does not conflict with any of our programs or data.

To recap what we learned about the stack in a previous chapter: The stack is an area of memory used to

Store return addresses for CALLs.
Store data when PUSHes are executed.
Store addresses when interrupts are active.

Addresses and data are *pushed* onto the stack, and the stack builds *downward* toward lower-numbered memory when this is done. A stack pointer register (SP) is adjusted to point to the *top of stack*, the location that has been used for the last CALL or PUSH storage. When a PUSH or CALL is performed, two bytes are pushed onto the stack and the SP register is decremented by two. When a POP or RETurn is performed, the two bytes are popped from the stack and the SP register is incremented by two to point to the next top of stack.

To see how this works, let us establish our own stack area and look at some of the stack actions. There is one instruction that loads the stack pointer with the first top of stack address, the LD SP,nnnn instruction. We will set aside 100 or so locations for the stack area starting at location 4AFFH, and building down to 4A9CH. The instruction to initialize this stack area is

```
LD  SP,4B00H  ;INITIALIZE STACK POINTER
```

The alert reader has discovered that *one more* than the actual top of stack address is used for initialization. The reason for this is that every PUSH or CALL first decrements the stack pointer *before* storing data. At any given time, then, the stack pointer points to the last byte stored, except for this case where no data has been stored at all.

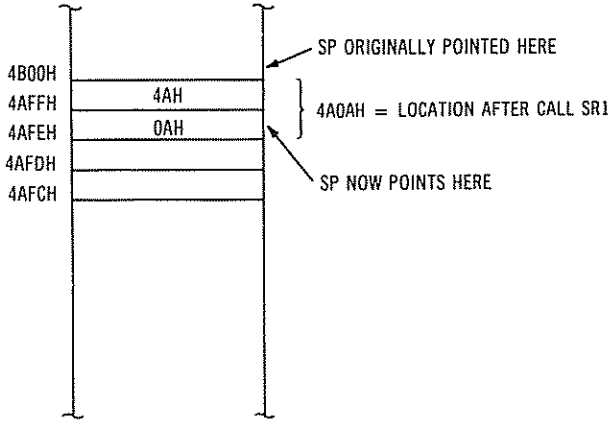


Fig. 6-3. Stack area Example 1.

When a CALL is executed, the address of the next instruction is stored in the stack with the most significant byte of the location stored in (SP)-1 and the least significant byte stored in (SP)-2. Let us illustrate this with a program. Key in the following code, set a breakpoint at LOOP, and then examine the stack area at 4AFFH down. You should see data as shown in Figure 6-3.

```
                00100 ;DEMONSTRATION OF STACK
                00110 .
4A00            00120           ORG     4A00H
4A00 210000     00130           LD      HL,0            ;CLEAR HL
4A03 39         00140           ADD     HL,SP           ;LOAD SP
4A04 31004B     00150           LD      SP,4B00H        ;INITIALIZE STACK
4A07 CD0E4A     00160           CALL    SR1             ;CALL SUBROUTINE
4A0A F9         00170           LD      SP,HL           ;RESTORE OLD SP
4A0B C30B4A     00180 LOOP      JP      LOOP            ;LOOP HERE FOR BP
4A0E C9         00190 SR1       RET                     ;A HUGE SUBROUTINE
0000            00200           END
00000 TOTAL ERRORS
LOOP    4A0B
SR1     4A0E
```

In the simple case above, the new stack area was used to store two bytes of the return address 4A and 0A in locations

4AFFH and 4AFEH, respectively. The stack pointer address used by T-BUG was saved in HL before the new stack area was initialized. When the short subroutine (the shortest possible subroutine) was executed and the RETurn made, the return address was retrieved from the stack and loaded into the program counter to cause the return to location 4A0AH. The LD SP,HL instruction restores the original stack pointer address used by T-BUG.

Nesting of subroutines can be used to any number of levels, just as GOSUBs in BASIC can cause nested subroutine action. As each new subroutine level is CALLed, the stack pointer is decremented further and further, and the return addresses are stored in lower and lower addresses in the stack. The program that follows shows how this works for four levels of subroutines. Breakpoint at LOOP, execute the program, and then examine the stack, starting at 4AFFH. It should correspond to Figure 6-4, and indicates that four separate return addresses were stored.

```
                00100 ;DEMONSTRATION OF STACK
                00110 .
4A00            00120           ORG     4A00H
4A00 210000     00130           LD      HL,0            ;CLEAR HL
4A03 39         00140           ADD     HL,SP           ;LOAD SP
4A04 31004B     00150           LD      SP,4B00H        ;INITIALIZE STACK
4A07 CD0E4A     00160           CALL    SR1             ;CALL SUBROUTINE
4A0A F9         00170           LD      SP,HL           ;RESTORE OLD SP
4A0B C30B4A     00180 LOOP      JP      LOOP            ;LOOP HERE FOR BP
4A0E CD124A     00190 SR1       CALL    SR2             ;SECOND LEVEL
4A11 C9         00200           RET
4A12 CD164A     00210 SR2       CALL    SR3             ;THIRD LEVEL
4A15 C9         00220           RET
4A16 C9         00230 SR3       RET                     ;FOURTH LEVEL
0000            00240           END
00000 TOTAL ERRORS
SR3     4A16
SR2     4A12
LOOP    4A0B
SR1     4A0E
```

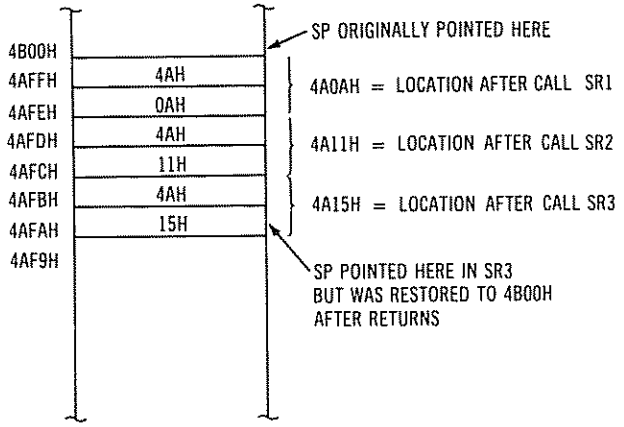


Fig. 6-4. Stack area Example 2.

When PUSHes or POPs are used, two bytes of data are also stored or retrieved in the stack, but the data represents data from cpu registers and not return addresses. When data is PUSHed, the high-order register is stored in (SP)-1 and the low-order register is stored in (SP)-2, in the same order that return addresses are stored (Figure 6-5). A third program following illustrates the storage action when CALLs and PUSHes are intermixed, as they will be in most programs.

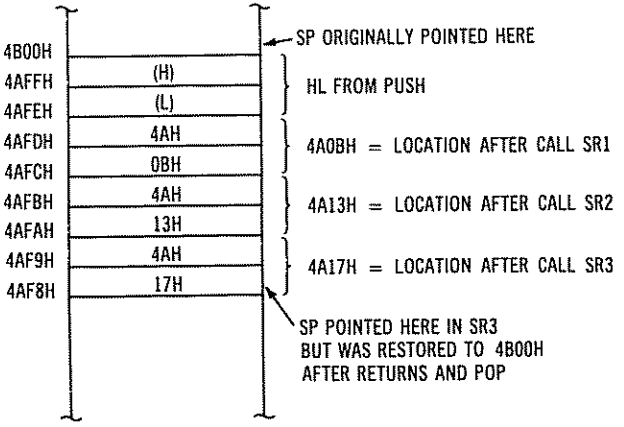


Fig. 6-5. Stack area Example 3.

```
                00100 ;DEMONSTRATION OF STACK
                00110 ;
4A00            00120           ORG     4A00H
4A00 210000     00130           LD      HL,0            ;CLEAR HL
4A03 39         00140           ADD     HL,SP           ;LOAD SP
4A04 31004B     00150           LD      SP,4B00H        ;INITIALIZE STACK
4A07 E5         00160           PUSH    HL              ;SAMPLE SAVE
4A08 CD104A     00170           CALL    SR1             ;CALL SUBROUTINE
4A0B D1         00180           POP     DE              ;SAMPLE RESTORE
4A0C F9         00190           LD      SP,HL           ;RESTORE OLD SP
4A0D C30D4A     00200 LOOP      JP      LOOP            ;LOOP HERE FOR BP
4A10 CD144A     00210 SR1       CALL    SR2             ;SECOND LEVEL
4A13 C9         00220           RET
4A14 CD184A     00230 SR2       CALL    SR3             ;THIRD LEVEL
4A17 C9         00240           RET
4A18 C9         00250 SR3       RET                     ;FOURTH LEVEL
0000            00260           END
00000 TOTAL ERRORS
SR3     4A18
SR2     4A14
LOOP    4A0D
SR1     4A10
```

Once the stack area has been defined by loading, the programmer need never worry about the stack and can indiscriminantly perform as many CALLs and PUSHes as he wishes, with a matching RETurn or POP for each CALL or PUSH. Generally, 30 or 40 bytes of RAM is large enough for even the most creative programmers; the number of nested subroutines is limited to 3 or 4 primarily by the problems in keeping the program in hand, just as in BASIC.

CHAPTER 7

Arithmetic and Compare Operations

This chapter will discuss the heart of any computer system —the ability to perform simple and complex arithmetic. In order to use the arithmetic capabilities of the Z-80, we will have to look in more detail at how numbers are represented in the architecture of the Z-80. After that chore, we'll build some routines to do adds and subtracts, decimal arithmetic, and other arithmetic-related processing.

Number Formats: Absolutely and Positively!

There are really three different ways to represent numbers in basic assembly-language routines used in the TRS-80, *absolute numbers*, *signed numbers*, and *binary-coded decimal*. (Another format, floating-point format, is too complex to describe in less than several chapters.) However, knowing the three formats just mentioned will enable the user to do virtually anything he wants in a TRS-80 processing routine.

In the previous chapters, we've been discussing numbers in absolute form, for the most part, although a few signed numbers have crept in when we discussed indexing and relative instructions. Absolute numbers are always positive; they can be looked at as "absolute-valued numbers." Earlier in the book we mentioned that in eight bits the binary values 00000000 through 11111111 could be held and that these represented 0 through 255 decimal. This still holds true (was there a collective sigh of relief?). Similarly, 16-bit numbers repre-

sent values from sixteen zeros to sixteen ones, or decimal 0 through 65535.

We also mentioned that binary numbers represented powers of two, and drew the parallel of the bit position in binary numbers representing powers of two, just as the decimal position in decimal numbers represents powers of ten. See Figure 7-1.

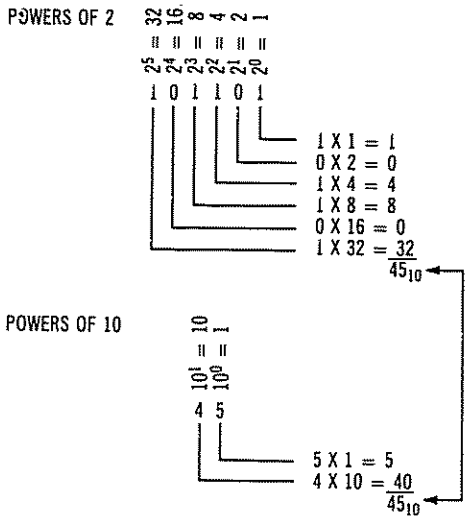


Fig. 7-1. Decimal versus binary numbers.

To convert any binary number to decimal, it is simply a matter of adding up all of the powers of two represented by one bits in the bit positions. Converting from decimal to binary can be done by inspection (what is the largest power of two that will go into this decimal number, what is the next, and so forth) or by reference to tables. See Figure 7-2.

We have been working with *hexadecimal* numbers, which are really a shorthand way of representing binary numbers that have been grouped in 4-bit groups. Converting from hexadecimal to decimal can be done in the same fashion as binary; that is, finding the weight of the power of 16 represented, or by reference to tables, as can conversion of decimal numbers to hexadecimal. See Figure 7-3.

Absolute numbers in binary (hexadecimal) can be used to represent memory addresses, counts, or any quantity that will never be negative. In register indirect addressing we've used absolute numbers to represent memory locations in the HL and other register pairs. We've also used absolute numbers

for counts or to represent the number of bytes to move in a block move. There are no negative numbers of bytes that must be moved, at least in this universe.

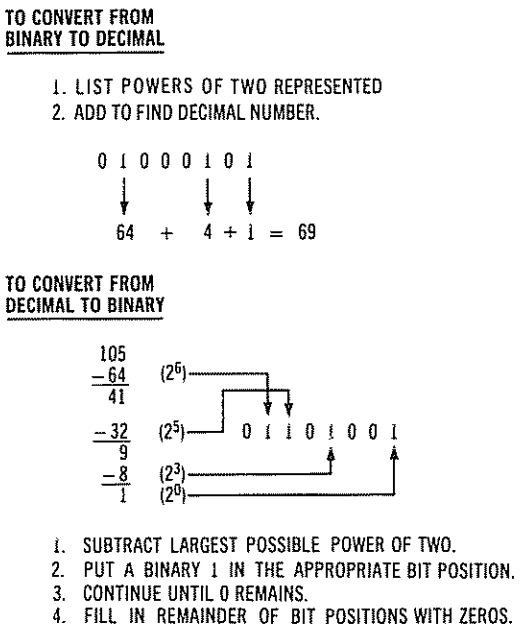


Fig. 7-2. Decimal/binary conversions.

Signed Numbers

The same registers and memory locations that are used to hold absolute addresses can hold *signed numbers*. Many different types of signed formats could be used, but the one that the Z-80 and TRS-80 uses is the same type that most other computers use, and it's called *two's complement notation.*

In two's complement notation, the most significant bit of eight bits or sixteen bits is used to represent the sign of the number. If the *sign bit* is a *zero*, then the remainder of the number *is the same as an absolute number*. For example, if we had the two's complement number 00001000, then that number would be an 8, the same as the absolute number 00001000. The difference between absolute numbers and positive two's complement numbers, is that the most significant bit is always the sign, and that means that the maximum positive number that can be held in 8-bit two's complement

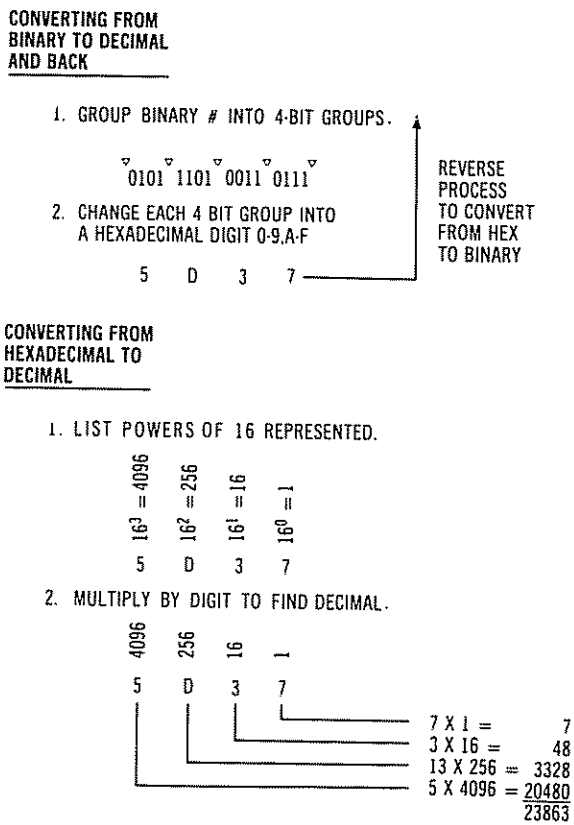


Fig. 7-3. Decimal/hexadecimal conversions.

notation is 01111111, or 127, about half of the maximum in absolute form (11111111 or 255). In sixteen bits the maximum positive number is 0111111111111111, or 32767 decimal.

Now here's the rub, as the Bard says in Much Ado About the TRS-80. When the sign bit is a one, the two's complement number represented is a negative number. When we see the two's complement number 10001000, we know from the sign bit that the number is negative. The question is, what negative number is it? The answer is *not* −8, even though it looks logical (all things in computers are not logical, in spite of the digital design). To find the actual negative number represented, we have to go through a purely rote procedure. It's not complicated, but it is tedious. In a negative two's complement number, to find the number represented, change all the ones to zeros, change all the zeros to ones, and add one. This process is demonstrated in Figure 7-4.

Why are negative numbers represented this way? To simplify hardware design. Next question . . . I'm afraid that's the way it is, TRS-80 programmers. Fortunately for us, the assembler takes care of constructing negative numbers and we generally don't have to be too concerned about manipulating them.

EXAMPLE 1: FIND TWO'S COMPLEMENT OF 10001000

```
10001000   NUMBER
01110111   CHANGE ALL ONES TO ZEROS
           ALL ZEROS TO ONES
   + 1     ADD ONE
01111000   THIS NUMBER NEGATED IS THE
           ACTUAL NUMBER. IN THIS CASE
           -120
```

EXAMPLE 2: FIND TWO'S COMPLEMENT OF 11110000

```
11110000   NUMBER
00001111   CHANGE ALL ONES TO ZEROS
           ALL ZEROS TO ONES
   + 1     ADD ONE
00010000   -16
```

EXAMPLE 3: FIND TWO'S COMPLEMENT OF 01111111

```
01111111   SIGN BIT IS + (0) AND NUMBER
           IS CORRECT AS IT STANDS (+127)
```

Fig. 7-4. Two's complement notation.

If we start applying this process of reconverting negative numbers, we find that the smallest number in two's complement notation is 10000000, or −128, while the largest negative number is 11111111, or −1, for 8-bit values. Similarly, the range of negative numbers for 16-bit values is −32768 (1000000000000000) through −1 (1111111111111111). So, the range of *all* signed numbers that can be held in 8 bits is +127 through −128 and in 16 bits +32767 through −32768.

The nice thing about two's complement notation is that the Z-80 will automatically handle addition and subtraction of any combination of signs. In the days of double-precision BASIC variables that can be processed in just about any manner this may raise some reader's eyebrows, but things in assembly language *are* at the most basic computational level. About the only requirement is that the programmer must know something about the range of numbers he will be handling. In 8 bits one can get +127 and no more, and in 16 bits

the maximum is +32767. If more precision is required, the program will have to handle longer strings of eight bits in a *multiple-precision scheme.*

Let's see how the assembler handles representation of signed numbers. The program that follows shows a data table of various types of signed numbers, eight bits (DEFB) and 16 bits (DEFW). Note how the assembler automatically computes the proper two's complement form. Might we even suggest the odious task of looking at the arguments, converting a few numbers yourself, and then checking them against the assembled value? Like chicken soup, it won't hurt!

```
                00100 ;TABLE OF CONSTANTS
                00110 .
4000            00120       ORG     4000H
4000 00         00130       DEFB    0
4001 01         00140       DEFB    1
4002 FF         00150       DEFB    0FFH
4003 FE         00160       DEFB    0FEH
4004 7F         00170       DEFB    127
4005 0000       00180       DEFW    0
4007 FFFF       00190       DEFW    -1
4009 0100       00200       DEFW    1
400B FF7F       00210       DEFW    +32767
400D 0080       00220       DEFW    -32768
                00230       END
00000 TOTAL ERRORS
```

Note that the 16-bit values are in standard Z-80 representation, reversed so that the most significant byte is last and the least significant byte is first.

Adding and Subtracting 8-Bit Numbers

There are several actions that occur when two 8-bit signed numbers are added in the Z-80. First, the instruction adds the two operands and puts the result in the A register (initially, as you will recall, one of the operands was in A). In the course of adding the numbers, the carry flag, half carry flag, overflow flag, zero flag, and sign flag are all affected according to the results of the add.

The zero flag is set if the result is zero. The two instructions

```
LD    A,23     ;LOAD 23 INTO A
ADD   A,-23    ;ADD -23
```

would result in an A register result of zero and the zero flag set to a one. The carry flag is set if there is a carry out of bit position 7 after the add, and the half carry is set if there is a carry out of bit position 3. These carries are equivalent to decimal carries during an addition of two decimal numbers. The carry out of bit position 3 is the "half-carry" and is used for decimal addition of binary-coded-decimal operands discussed later on in this chapter. The "carry" out of the high-order bit position occurs whenever a carry is generated for the add, as in the add of 23 and −23.

```
        00010111     23
carry   11101001    -23  (try the two's complement)
------------------------
  1     00000000      0  (zero result)
```

The carry flag can be used for adds of multiple bytes, for adds of bcd operands, or for certain types of compares.

The sign flag is really the duplication of the sign bit in the result after the add. If the result of the add is positive, the sign flag is reset (0), while if the result is negative, the sign flag is set (1). The sign flag can then be used for conditional jumps such as jump if result positive (JP P,aaaa) or jump if result negative (JP M,aaaa).

The overflow flag is used during adds and subtracts to detect *overflow* conditions. Overflow occurs when the result of the add is too large to fit into an 8-bit signed representation. Suppose that we are adding +127 and +50. We know that the maximum positive number that can fit in 8 bits is +127. What would the result be if we actually performed the add?

```
01111111  (+127)
00110010  (+ 50)
----------------
10110001  (- 79)   result — wrong!
```

As the reader can see from the example, the result of −79 is incorrect. If we had no way to detect the overflow, we might go merrily on our way printing a paycheck for an employee of $1,045,067.66, or an equally catastrophic action. Fortunately, the Z-80 *does* set overflow when the result is greater than +127 or less than −128.

When a subtract instead of an add is used, all of the above actions apply. The Z-80 performs the subtract just as you

would on paper, and then sets the flags according to the results of the subtract. There are really no fundamental differences between an add and subtract, as the reader can see if he considers adding +23 and −15 and then compares it to subtracting +15 from +23.

To illustrate the settings of the flag bits after an add or subtract, let's use T-BUG to execute some examples of arithmetic operations. Load T-BUG and key in the following program. Run the following examples by using T-BUG to change the operands in 4B00H and 4B01H, breakpoint at location 4A14H and then use the M command to look at the flags and results in locations 4B02H through 4B05H as shown in Table 7-1. In addition to the examples below the reader is urged to try his own values. The flags will have to be "decoded" from an 8-bit value to determine the state of the flags (it *is* some work, but you'll be a better programmer for it). The bit positions of the flag register are shown in Figure 7-5, and in Table 7-1.

```
                00100 ;PROGRAM TO ILLUSTRATE ARITHMETIC
                00110 .
4A00            00120       ORG   4A00H
4A00 3A014B     00130       LD    A,(4B01H)     ;GET SOURCE
4A03 47         00140       LD.   B,A           ;FOR OPERATION
4A04 3A004B     00150       LD    A,(4B00H)     ;GET DESTINATION
4A07 80         00160       ADD   A,B           ;ADD
4A08 F5         00170       PUSH  AF            ;TRANSFER FLAGS
4A09 E1         00180       POP   HL            ;GET RESULT FLAGS
4A0A 22024B     00190       LD    (4B02H),HL    ;STORE
4A0D 90         00200       SUB   B             ;RESTORE
4A0E 90         00210       SUB   B             ;SUBTRACT
4A0F F5         00220       PUSH  AF            ;TRANSFER FLAGS
4A10 E1         00230       POP   HL            ;GET RESULT, FLAGS
4A11 22044B     00240       LD    (4B04H),HL    ;STORE
4A14 C3144A     00245 LOOP  JP    LOOP          ;LOOP HERE FOR BP
0000            00250       END
00000 TOTAL ERRORS
LOOP   4A14
```

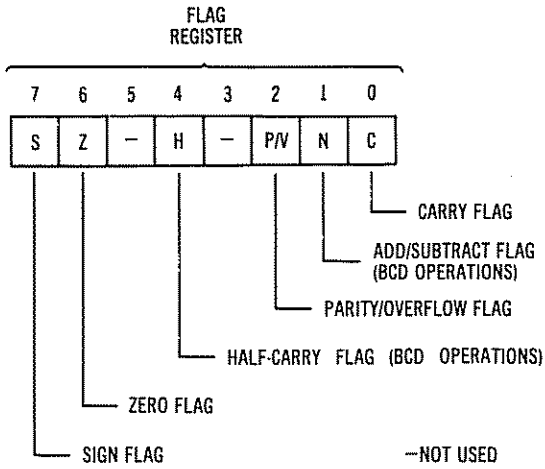


Fig. 7-5. Flag register bit positions.

Table 7-1. Examples of Add and Subtract Flag Bit

Location	Contents	Test Cases 1	2	3	4
4B00H	Dest Op	+33(21H)	−5(FBH)	−30(E2H)	120(78H)
4B01H	Source Op	+64(40H)	−30(E2H)	−5(FBH)	100(64H)
4B02H	Add Flags	00100000	10001001	10001001	10001100
4B03H	Add Result	+97(61H)	−35(DDH)	−35(DDH)	−24(DCH)
4B04H	Sub Flags	10100011	00001010	10110011	00000010
4B05H	Sub Result	−31(E1H)	+25(19H)	−25(E7H)	+20(14H)
		FLAGS S Z · H · P/V N C 7 6 5 4 3 2 1 0			

Adding and Subtracting 16-Bit Numbers

The Z-80 allows two 16-bit operands to be added, as we found in a previous chapter. One of the operands must be in the HL, IX, or IY registers, analogous to the A register in 16-bit arithmetic; the second operand must be in one of the other register pairs. When an add or subtract is performed 16 bits at a time, the flags are affected in various ways, depending upon which of the 16-bit arithmetic instructions is being used. When an add is done to the IX register, for example, the zero and sign flags are not affected, but when an "ADC" is done with the HL register, the sign and zero flags *are* affected. When in doubt about flag action, consult the

individual flag action listed under the instruction in question in the Editor/Assembler manual.

The advantage of the 16-bit adds, of course, is that much larger numbers can be handled, at the expense of addressing versatility. Since the HL, IX, and IY registers are generally used as memory printer registers, the 16-bit adds and subtracts using these registers can be used to advantage to calculate memory addresses. As an example of this memory address computation capability, let's use the following program. This program uses 16-bit adds and subtracts to calculate memory addresses for movement of a dot across the video screen.

```
                00100 ;ROUTINE TO MOVE A DOT
                00110 .
4000            00120           ORG     4000H           ;START
4000 21203C     00130           LD      HL,3C00H+32     ;START POSITION
4003 114000     00150           LD      DE,64           ;INCREMENT
4006 3E0F       00160           LD      A,15            ;NUMBER OF LINES
4008 010000     00170           LD      BC,0            ;DELAY COUNT
400B 360F       00180 LOOP1     LD      (HL),00FH       ;ALL ON
400D 05         00200 LOOP2     DEC     B               ;DELAY CNT-1
400E 20FD       00210           JR      NZ,LOOP2        ;GO IF NOT DONE
4010 0D         00220           DEC     C               ;DELAY COUNT
4011 20FA       00230           JR      NZ,LOOP2        ;GO IF NOT DONE
4013 3600       00240           LD      (HL),00H        ;ALL OFF
4015 19         00250           ADD     HL,DE           ;NEXT ROW
4016 3D         00260           DEC     A               ;#LINES-1
4017 20F2       00270           JR      NZ,LOOP1        ;CONTINUE
4019 18FE       00280 LOOP3     JR      LOOP3           ;LOOP HERE
0000            00290           END
00000 TOTAL ERRORS
LOOP3   4019
LOOP2   400D
LOOP1   400B
```

The program starts by loading HL with the first position of the dot, the screen memory plus one-half line. DE is loaded with 64, representing the number that must be added to move

the dot to the middle of the next line. A is loaded with 15, the number of lines that the dot will move. BC is loaded with a delay count of 0, representing a delay of 65536 counts when BC is decremented in the loop. The action of the loop from LOOP1 through 4A17H is this: The dot is initially set on by outputting the graphic character 0BFH. This character sets every one of the six *pixels* in the character position. Now the program delays about ½ second by means of a 4 instruction delay loop. BC has zero at the end of the loop. After the delay the pixels are turned off by outputting the graphics character 80H. Then the next address is computed by adding the 64 in DE to HL, the address pointer. The contents of A are decremented by one. If 15 lines have not been reached, the program loops back to LOOP1.

There are several interesting things in the above program. Because the assembly-language code is extremely fast, we had to delay each time a dot (actually six dots) was moved to a new position. The delay count in BC was initialized to 0, and decremented by decrementing B back to 0 again (256 loops) as an *inner loop* and by decrementing C from 0 back to 0 as an *outer loop*. The reader should realize that at 4A13H, the count in BC is 0, in preparation for the next delay loop. Another point is that there is no way to decrement BC and test for zero, as the flags are not affected by a DEC BC. Hence two decrements are used, each one checking one of the two registers for zero—a DEC B or DEC C *does* set the flags after the decrement.

To illustrate the 16-bit subtract, we'll rewrite the program above to make a single pixel move from the bottom of the screen to the top of the screen. This program will be identical to the one above except that the starting position will be 3C00H+992, the 32nd character position in line 16, the increment in DE will be −64, and the graphics codes will specify all on or all off for a single pixel (we'll be looking at the graphics codes in more detail in a later chapter).

```
                00100 ;ROUTINE TO MOVE A DOT (BACKWARDS)
                00110 .
4A00            00120       ORG     4A00H           ;START
4A00 21E03F     00130       LD      HL,3C00H+992    ;START POSITION
4A03 1140C0     00140       LD      DE,-64          ;INCREMENT
4A06 3E0F       00150       LD      A,15            ;NUMBER OF LINES
```

```
4A8B 010000     00160        LD      BC,0        ;DELAY COUNT
4A8E 3691       00170 LOOP1  LD      (HL),91H    ;ONE PIXEL ON
4A90 05         00180 LOOP2  DEC     B           ;DELAY COUNT - 1
4A91 20FD       00190        JR      NZ,LOOP2    ;GO IF NOT DONE
4A93 0D         00200        DEC     C           ;DELAY COUNT
4A94 20FA       00210        JR      NZ,LOOP2    ;GO IF NOT DONE
4A96 3600       00220        LD      (HL),00H    ;ALL OFF
4A98 B7         00230        OR      A           ;RESET CARRY
4A99 ED52       00240        SBC     HL,DE       ;NEXT ROW
4A9B 3D         00250        DEC     A           ;#LINES-1
4A9C 20F0       00260        JR      NZ,LOOP1    ;CONTINUE
4A9E 18FE       00270 LOOP3  JR      LOOP3       ;LOOP HERE
0000            00280        END
00000 TOTAL ERRORS
LOOP3  4A9E
LOOP2  4A90
LOOP1  4A8E
```

The subtract was performed by the SBC instruction which subtracted the increment value of 64 from the current video memory position in HL. Note that before the subtract, an OR A was done. *The only reason for performing the* OR *A was to reset the carry flag*. The two questions that may immediately come to the readers mind are why use an OR A to reset the carry, and *why* reset the carry? The OR A is used because it is a short (one byte) and fast instruction. We could have reset the carry flag by an SCF followed by a CCF (set carry, complement carry), but the OR A does not affect the contents of the A register (try ORing any value with itself) and it is efficient.

Why do we want to reset the carry before the SBC? Well the SBC is actually a Subtract with Carry type instruction that not only subtracts a second operand from the contents of HL, but also subtracts the current state of the carry. That means that one more count might be subtracted from HL if the carry is set before the subtract. Since the carry is set and reset with many instructions, we have no way of knowing whether the carry will be set or reset before the SBC, and therefore *must* clear the carry to avoid subtracting a possible one from the result.

A Precision Instrument

The reason that the carry enters into some adds and subtracts on the TRS-80 is that the Z-80, like other microprocessors, is able to handle *multiple-precision* adds and subtracts. Remember that the maximum value that can be held in 8 bits is 255 and that the maximum value that can be held in 16 bits is 65535. What happens if we want more *precision* and want to hold larger numbers for adds and subtracts? How could we add 32-bit (four byte) numbers, for example, allowing us to work with values up to 4 billion or so (2^{32})?

Larger numbers are held in multiple-precision representation, which is simply a method for representing the numbers in as many bytes as required. If we know, for example, that a billion or so is the largest number we'll be working with, we can conveniently work with four-byte numbers in the Z-80. Suppose that we wanted to add two four-byte operands of +344,050 and +500,000, as shown in Figure 7-6. The numbers

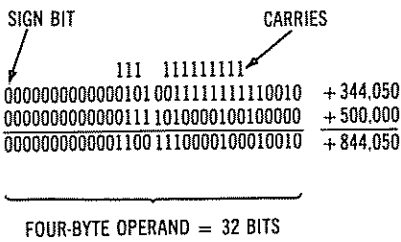


Fig. 7-6. Multiple-precision adds by manual methods.

are *signed* 32-bit operands, with the most significant bit representing the sign of plus (0) or minus (1) just as in the case of 8- or 16-bit operands. To add them with pencil and paper, we simply add the ones and zeros, and any carry from the lower bit positions as shown in the figure.

To add the numbers in the Z-80, we have a bit of a problem (32 bits of a problem, to be precise). We can add up to 16-bit operands, but how can we add 32 bits at a time? The answer is that the adds must be either four 8-bit adds or two 16-bit adds. Each of the adds must *add in* any carry from the last byte or two bytes, just as we do on pencil and paper operations. The following program adds two four-byte operands, representing the above values. The operands are in memory locations 4B00H-4B03H and 4B10H-4B13H and the result is stored in location 4B00H-4B03H. Key in the program using

T-BUG (or assemble and load), execute at 4A00H after breakpointing, and then check the result at 4B00H-4B03H. It should correspond with the result shown in Figure 7-7.

```
                00100 ;FOUR BYTE ADD ROUTINE
                00110 ;
4000            00120           ORG     4000H
4000 DD21004B   00130 START     LD      IX,4B00H        ;DESTINATION
4004 FD21104B   00140           LD      IY,4B10H        ;SOURCE
4008 DD7E03     00150           LD      A,(IX+3)        ;GET BYTE 0
400B FD8603     00160           ADD     A,(IY+3)        ;ADD SOURCE
400E DD7703     00170           LD      (IX+3),A        ;STORE RESULT
4011 DD7E02     00180           LD      A,(IX+2)        ;GET BYTE 1
4014 FD8E01     00190           ADC     A,(IY+1)        ;ADD SOURCE
4017 DD7702     00200           LD      (IX+2),A        ;STORE RESULT
401A DD7E01     00210           LD      A,(IX+1)        ;GET BYTE 2
401D FD8E02     00220           ADC     A,(IY+2)        ;ADD SOURCE
4020 DD7701     00230           LD      (IX+1),A        ;STORE RESULT
4023 DD7E00     00240           LD      A,(IX)          ;GET BYTE 3
4026 FD8E00     00250           ADC     A,(IY)          ;ADD SOURCE
4029 DD7700     00260           LD      (IX),A          ;STORE RESULT
402C C32C40     00270 LOOP      JP      LOOP            ;LOOP HERE FOR BP
4B00            00280           ORG     4B00H           ;DESTINATION AREA
4B00 00         00290           DEFB    0               ;+344,650
4B01 05         00300           DEFB    5
4B02 3F         00310           DEFB    3FH
4B03 F2         00320           DEFB    0F2H
4B10            00330           ORG     4B10H           ;SOURCE AREA
4B10 00         00340           DEFB    0
4B11 07         00350           DEFB    7
4B12 A1         00360           DEFB    0A1H
4B13 20         00370           DEFB    20H
0000            00380           END
00000 TOTAL ERRORS
LOOP    402C
START   4000
```

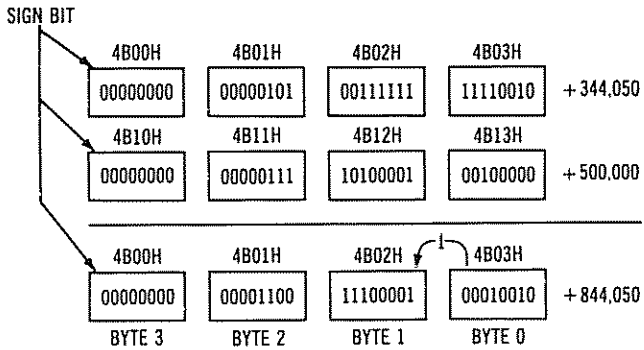


Fig. 7-7. Multiple-precision adds by machine.

The program uses indexed addressing with both IX and IY. The IX register points to the *destination* operand in 4B00H through 4B03H. Note that the most significant byte is at 4B00H and the last significant byte is at 4B03H. The IY register points to the *source* operand at 4B10H. Although the four adds could have been done in a loop, the in-line code in the program clearly shows the steps that must be taken for the adds. The first add adds (IX+3) and (IY+3), the least significant byte, and stores the result in 4B03H. After the ADD, the carry flag is set or reset dependent upon the carry from bit position 7, *which is not a sign bit, but just another bit position.* The next add (ADC) adds not only the two bytes from (IX+2) and (IY+2), but the *carry* from the previous add, which is undisturbed, as loads do not affect the carry or other flags. The next add adds in the carry from the second byte, and the last add adds in the carry from the third byte. All adds, except the first, added in a possible carry from the lower order byte. In the first add there was no preceding carry to be added in.

The program shows the general approach to add any number of bytes. There is no limit on the maximum number of bytes that could be used, but working with 32-byte operands might get somewhat tedious after a while. Floating-point format allows a more compact representation of large numbers, at the sacrifice of the number of significant digits, and is widely used in cases where very large, very small, or mixed numbers must be used.

Subtraction of multiple-precision numbers is handled in similar fashion. The first subtract would be an SUB without the carry, but the remaining three would be SBCs, which use the borrow from the preceding lower-order byte. A portion of this code is shown below.

```
LD    A,(IX+2)  ;SECOND BYTE
SBC   A,(IY+2)  ;SUBTRACT SOURCE
LD    (IX+2),A  ;STORE RESULT
```

There is no reason that 16-bit adds and subtracts couldn't be used, as long as the total number of bytes was a multiple of two. In the general case, 8-bit adds and subtracts are somewhat easier to work with, as they allow for an odd number of bytes and permit a direct add or subtract of the source operand (through HL, IX, or IY). The two programs shown below are general-purpose subroutines for multiple-precision adds and subtracts. They will handle any number of bytes required. Upon entry, IX and IY point to the first (most significant) bytes of the destination and source, respectively. The B register contains the number of bytes in the operands (both operands must have the same number of bytes). The subroutines add or subtract the source operand from the destination operand and put the result in the destination operand memory locations. Upon return from the subroutine IX and IY are unchanged and the contents of B are zero.

```
                00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION ADDS
                00110 ;   ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                00120 ;        (IY)=POINTS TO MS BYTE OF SOURCE
                00130 ;        (B)=# OF BYTES IN OPERANDS
                00140 ;        CALL  MULADD
                00150 ;        (RETURN)
                00160 ;   EXIT:(IX)=UNCHANGED
                00170 ;        (IY)=UNCHANGED
                00180 ;        (A)=DESTROYED
                00190 ;        (B)=0
                00200 ;
4A00            00210         ORG   4A00H         ;CHANGE ON REASSEMBLY
4A00 D5         00220 MULADD  PUSH  DE            ;SAVE DE
4A01 58         00230         LD    E,B           ;#BYTES TO E
4A02 1600       00240         LD    D,0           ;DE NOW HAS #
4A04 1B         00250         DEC   DE            ;DE NOW HAS #-1
4A05 DD19       00260         ADD   IX,DE         ;POINT TO LS BYTE
4A07 FD19       00270         ADD   IY,DE         ;POINT TO LS BYTE
```

```
4009 D1        00280          POP     DE              ;RESTORE ORIGINAL
400A AF        00290          XOR     A               ;RESET CARRY
400B DD7E00    00300 LOOP     LD      A,(IX)          ;GET DESTINATION
400E FD9E00    00310          ADC     A,(IY)          ;ADD SOURCE
4011 DD7700    00320          LD      (IX),A          ;STORE RESULT
4014 1001      00330          DJNZ    LOOP1           ;GO IF NOT DONE
4016 C9        00340          RET                     ;RETURN
4017 DD2B      00350 LOOP1    DEC     IX              ;PNT TO NEXT HIGHER
4019 FD2B      00360          DEC     IY              ;PNT TO NEXT HIGHER
401B C30B40    00370          JP      LOOP            ;CONTINUE
0000           00380          END
00000 TOTAL ERRORS
LOOP1  4017
LOOP   400B
MULSUB 4000
```

```
                00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION SUBTRACTS
                00110 ;    ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                00120 ;          (IY)=POINTS TO MS BYTE OF SOURCE
                00130 ;          (B)=# OF BYTES IN OPERANDS
                00140 ;          CALL  MULSUB
                00150 ;          (RETURN)
                00160 ;    EXIT:(IX)=UNCHANGED
                00170 ;         (IY)=UNCHANGED
                00180 ;         (A)=DESTROYED
                00190 ;         (B)=0
                00200 ;
4000            00210          ORG     4000H           ;CHANGE ON REASSEMBLY
4000 D5         00220 MULSUB   PUSH    DE              ;SAVE DE
4001 58         00230          LD      E,B             ;#BYTES TO E
4002 1600       00240          LD      D,0             ;DE NOW HAS #
4004 1B         00250          DEC     DE              ;DE NOW HAS #-1
4005 DD19       00260          ADD     IX,DE           ;POINT TO LS BYTE
4007 FD19       00270          ADD     IY,DE           ;POINT TO LS BYTE
4009 D1         00280          POP     DE              ;RESTORE ORIGINAL
```

```
4000 AF         00290          XOR     A          ;RESET CARRY
4001 DD7E00     00300 LOOP     LD      A,(IX)     ;GET DESTINATION
4004 FD9E00     00310          SBC     A,(IY)     ;SUBTRACT SOURCE
4007 DD7700     00320          LD      (IX),A     ;STORE RESULT
400A 1001       00330          DJNZ    LOOP1      ;GO IF NOT DONE
400C C9         00340          RET                ;RETURN
400D DD2B       00350 LOOP1    DEC     IX         ;PNT TO NEXT HIGHER
400F FD2B       00360          DEC     IY         ;PNT TO NEXT HIGHER
4011 C30040     00370          JP      LOOP       ;CONTINUE
0000            00380          END
00000 TOTAL ERRORS
LOOP1   400D
LOOP    4001
MULSUB  4000
```

Decimal Arithmetic

Up to this point we've been doing arithmetic operations with absolute and two's complement numbers. As we mentioned earlier in the chapter, there is a third type of arithmetic that is possible in the Z-80 and many other microprocessors, *binary-coded-decimal* (*bcd*) arithmetic. The bcd representation is a more direct translation from decimal than binary. To convert a decimal number into bcd, change each decimal digit into its 4-bit binary equivalent. Some exam-

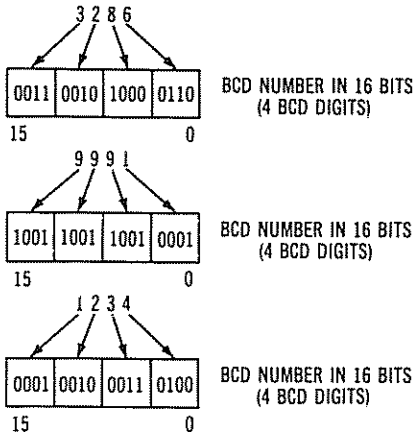


Fig. 7-8. The bcd representation.

ples of this are shown in Figure 7-8. After the conversion we're left with a *binarylike* number whose length equals four times the number of decimal digits, or to put it another way, two bcd digits in each 8-bit segment as shown in the figure.

The bcd representation is used for a variety of purposes. Much instrumentation uses bcd, especially instrumentation that displays digits in digital readout form, such as digital voltmeters and digital frequency counters. We could, of course, convert from bcd to binary, perform arithmetic operations in binary, and reconvert to bcd, but it is convenient to be able to directly add or subtract bcd values in the Z-80.

Adding or subtracting bcd is *not* the same as adding or subtracting binary numbers. Since the binary groups of 1010 through 1111 are not permitted in bcd (there is no bcd equivalent), operations in binary produce erroneous results, as

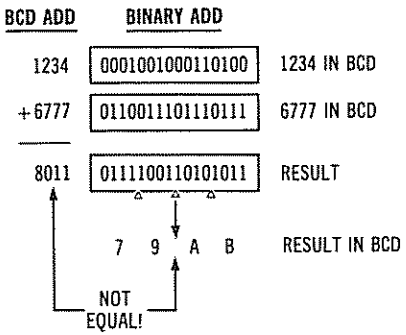


Fig. 7-9. A bcd add with erroneous result.

shown in Figure 7-9, where the bcd add of 1234 and 6777 produces 8011H and the binary add of the two numbers produces 79ABH. It turns out that to convert a binary result of the add of two bcd operands into bcd, it is only necessary to

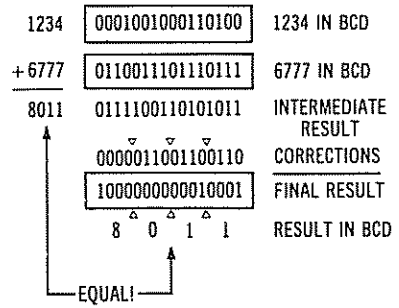


Fig. 7-10. Bcd corrections.

look at each of the groups of four bits to see whether or not a *correction* is required. If a 4-bit group in the result contains 1010, 1011, 1100, 1101, 1110, or 1111, or if a carry *from* the group resulted, then 0110 is added to the group to adjust the binary result to a bcd result. As every byte holds two bcd digits, two such checks are necessary for each binary byte. The process is shown in Figure 7-10, where corrections are made to the operands shown in Figure 7-9.

Bcd subtractions require the same adjustment, but in this case six is *subtracted* if necessary from a bcd digit in the result. It's relatively easy to implement a program to look at each bcd digit and test to see if an add or subtract adjustment is necessary, but the Z-80 does it all in one instruction, the DAA, or *D*ecimal *A*djust *A*ccumulator instruction. When bcd operands are being added or subtracted, the DAA is executed directly after the add or subtract to automatically (aren't computers wonderful) adjust the binary result to a bcd result. To see how this works, we'll write a program to count in bcd for 00 to 99 and compare the results with values stored from 00H through 63H in binary. The following program stores the bcd values from 00 through 99 into a buffer starting at 4B00H and stores a corresponding count from 0 through 99 in binary into a second buffer at 4C00H. Enter the program by assembling and loading or by using T-BUG to enter in machine language, breakpoint at END, and then compare the results in the two buffers. By "dumping" the bcd buffer using the M command in T-BUG (with carriage return), you will see a sequence of bcd numbers from 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, up to 99.

```
                00100 ;PROGRAM TO DEMONSTRATE BCD
                00110 .
4000            00120       ORG     4000H
4000 110000     00130 START LD      DE,0        ;D=BCD;E=BINARY
4003 DD21004B   00140       LD      IX,4B00H    ;BCD BUFFER
4007 FD21004C   00150       LD      IY,4C00H    ;BINARY BUFFER
400B 0664       00160       LD      B,100       ;COUNT
400D DD7200     00170 LOOP  LD      (IX),D      ;STORE BCD
4010 FD7300     00180       LD      (IY),E      ;STORE BINARY
4013 DD23       00190       INC     IX          ;BUMP BCD POINTER
4015 FD23       00200       INC     IY          ;BUMP BINARY POINTER
4017 7A         00210       LD      A,D         ;GET BCD
```

```
4018 C601       00220          ADD     A,1         ;ADD 1
401A 27         00230          DAA                 ;DECIMAL ADJUST
401B 57         00240          LD      D,A         ;SAVE BCD VALUE
401C 7B         00250          LD      A,E         ;GET BINARY
401D C601       00260          ADD     A,1         ;ADD 1
401F 5F         00270          LD      E,A         ;SAVE BINARY VALUE
4020 10EB       00280          DJNZ    LOOP        ;GO IF NOT 100
4022 18FE       00290 LOOP1    JR      LOOP1       ;LOOP HERE IF DONE
0000            00300          END
00000 TOTAL ERRORS
LOOP1   4022
LOOP    4008
START   4000
```

Compare Operations

As we described in an earlier chapter, compares are essentially subtracts, where the result of the subtraction is only used to set the cpu flags and is not put into the destination register. Unlike subtracts, compares only operate with 8-bit operands, and one of the operands must be in the A register. Compares and subtracts may be used to test two operands for the same states as BASIC comparisons—tests for an operand greater than another, greater or equal, equal, not equal, less than or equal, or less than. Some of these tests are directly handled by the zero and sign flags, while others must use the carry flag.

The test for equality or non-equality is simple and uses the zero flag. In the following code a branch is made to NOTEQ if the contents of the A register are not equal to the contents of the B register and to EQUAL if the two registers are the same.

```
TEST  CP   B          ;TEST BY A-B
      JP   Z,EQUAL    ;GO IF A=B
      JP   NOTEQ      ;MUST BE A NE B HERE
```

When the two numbers to be compared are absolute (unsigned) numbers, the carry flag will be set after the compare if the contents of A are less than the second operand. If A holds 128 and the C register holds 130, for example, the branch to LESSTH will be taken in the code below.

```
TEST    CP   C          ;TEST A-C
        JP   C,LESSTH   ;GO IF A LESS THAN C
        JP   Z,EQUAL    ;GO IF A=C
GTHAN   ...             ;A GREATER THAN C HERE
```

When the two numbers to be compared are signed numbers, then the carry flag logic gets rather confusing. For this reason we present a general-purpose subroutine that compares two signed numbers and jumps to one of three locations based on a comparison of the operands. By making the branch locations identical, any combination of equality conditions may be constructed. If a branch is to be made on greater or equal, for example, the greater than branch will be to GTEQU and the equal branch will also be to GTEQU, with the less than branch to some other location.

```
                00100 ;SUBROUTINE TO COMPARE TWO 8-BIT SIGNED OPERANDS
                00110 ;    ENTRY:(A)=OPERAND 1
                00120 ;          (B)=OPERAND 2
                00130 ;          CALL  COMPARE        ;CALL IR
                00140 ;          ;RTN FOR A LT B)     ;PUT JP  LESST HERE
                00150 ;          ;RTN FOR A=B)        ;PUT JP  EQUAL HERE
                00160 ;          ;RTN FOR A GT B)     ;PUT JP  GREATR HERE
                00170 ;    EXIT: (A)=UNCHANGED
                00180 ;          (B)=UNCHANGED
                00190 ;          (HL)=DESTROYED
                00200 ;
4A00            00210          ORG   4A00H          ;CHANGE ON REASSEMBLY
4A00 E1         00220 COMPARE  POP   HL             ;GET RTN ADDRESS
4A01 D5         00230          PUSH  DE             ;SAVE DE
4A02 110300     00240          LD    DE,3           ;ADDRESS INCREMENT
4A05 B8         00250          CP    B              ;COMPARE A:B
4A06 280A       00260          JR    Z,EQUAL        ;GO IF EQUAL
4A08 F5         00270          PUSH  AF             ;SAVE FLAGS
4A09 A8         00280          XOR   B              ;TEST SIGN BITS
4A0A 17         00290          RLA                  ;XOR TO C
4A0B D4154A     00300          JP    C,DIFFER       ;GO IF DIFFERENT SIGNS
4A0E F1         00310          POP   AF             ;RESTORE FLAGS
4A0F 3802       00320          JR    C,LESST        ;GO IF A LT B
4A11 19         00330 GREATR   ADD   HL,DE          ;BUMP RTN BY 3
```

```
4A1E 19       00340 EQUAL   ADD     HL,DE         ;BUMP RTN BY 3
4A1F D1       00350 LESST   POP     DE            ;RESTORE DE
4A20 E9       00360         JP      (HL)          ;RTN TO 0,3,6
4A21 F1       00370 DIFFER  POP     AF            ;RESTORE FLAGS
4A22 DA114A   00380         JP      C,GREATR      ;GO IF A GT B
4A25 C3134A   00390         JP      LESST         ;A LT B
0000          00400         END
00000 TOTAL ERRORS
GREATR  4A11
LESST   4A13
DIFFER  4A15
EQUAL   4A12
COMPARE 4A00
```

The block compare is used in string searches and will be discussed in Chapter 9 when we look at strings and tables.

CHAPTER 8

Logical Operations, Bit Operations, and Shifts

The operations in this chapter differ from the arithmetic operations in the last chapter in that the operations here are all concerned with subdivisions of bytes, either *fields* of a byte or down to the individual bit level. The logical instructions are used to retrieve or store information in segments less than a byte in length, the bit instructions manipulate individual bits in memory or register bytes, and the shifts align fields or manipulate individual bits.

AND, ORs, and Exclusive ORs

The AND instruction is used primarily to *mask out* unwanted data in bytes. Suppose, for example, that in each byte of data in a table in memory we had an ASCII character representing the digits of 0 through 9. Now it turns out that the ASCII representation of those digits follows a rather logical order as the reader can see from Table 8-1. The ASCII representation of 0 is 30H, 1 is 31H, and so on up to 39H for 9. To convert one ASCII digit of 30H through 39H into a binary value equivalent to the ASCII character, it is only necessary to get rid of the *bias* of 30H. This could be done by subtraction, but an equivalent alternative would be to mask out the "3" portion of the ASCII by an AND.

```
LD    A,ASCII  ;GET ASCII VALUE
AND   0FH      ;GET LAST FOUR BITS
```

When the ASCII values are masked by the immediate value 0FH (00001111), only the last four bits fall through, and since the least significant four bits are 0 through 9 in this case, the result is the equivalent binary value.

Table 8-1. ASCII Representation of Decimal and Hexadecimal

	Digit	ASCII Code
Decimal	0	30H
	1	31H
	2	32H
	3	33H
	4	34H
	5	35H
	6	36H
	7	37H
	8	38H
	9	39H
Hexadecimal	A	41H
	B	42H
	C	43H
	D	44H
	E	45H
	F	46H

Conversely, a binary value of 0 through 9 could be converted into an equivalent ASCII value for output by setting the "3" bits. Although an add could be used, the ASCII values could also be generated by an OR instruction.

```
LD   A,(BINARY)  ;GET BINARY VALUE
OR   30H         ;CONVERT TO ASCII
```

In both of the preceding cases we have assumed that only valid ASCII characters of 0 through 9 are involved, and that the binary values will be 0 through 9. As a simple illustration of this conversion, let's write out the screen line number 0 through 9, for the first ten lines of the screen. The following program does this by counting for 0 through 9 and ORing in the "3" value to make an ASCII digit out of the count.

```
                00100 ;WRITE OUT LINES 0-9 IN ASCII
                00110 ;
4000            00120       ORG     4000H
4000 213C0C     00130       LD      HL,3C00H+32     ;MIDDLE OF 1ST LINE
```

```
4003 0600       00140        LD      B,0          ;INITIALIZE COUNT
4005 0E39       00150        LD      C,39H        ;LAST ASCII
4007 114000     00170        LD      DE,64        ;LINE INCREMENT
400A 78         00180 LOOP   LD      A,B          ;GET CURRENT COUNT
400B F630       00190        OR      30H          ;CONVERT TO ASCII
400D 77         00200        LD      (HL),A       ;STORE ON SCREEN
400E 19         00210        ADD     HL,DE        ;BUMP LINE PNTR
400F 04         00220        INC     B            ;BUMP COUNT
4010 B9         00230        CP      C            ;TEST FOR END
4011 C20A40     00240        JP      NZ,LOOP      ;GO IF NOT DONE
4014 C31440     00250 LOOP1  JP      LOOP1        ;LOOP HERE AT END
0000            00260        END
00000 TOTAL ERRORS
LOOP1   4014
LOOP    400A
```

The exclusive OR does not find as much use as the AND and OR instructions. Recall that the exclusive OR generates a one bit in the result if there is a single one bit but not two one bits in the bit positions of the two operands. The most common use of the exclusive OR in the TRS-80 is to zero the accumulator by the efficient instruction.

```
XOR  A  ;ZERO A REGISTER AND CARRY
```

Another use of the exclusive OR is to *toggle* a counter from 0 to 1 and back again as in

```
        LD    A,1       ;SET TOGGLE TO ONE
LOOP    XOR   1         ;TOGGLE
        JP    Z,ZERO    ;GO IF ZERO
        JP    ONE       ;ONE ACTION
```

One of the more common operations in the Z-80 and other computers is to set or reset a bit in a memory byte or register byte. To set a bit in memory in many computers, the following three instructions must be executed

```
LD   A,(HL)  ;LOAD THE MEMORY BYTE
OR   A,4     ;SET BIT 2
LD   (HL),A  ;STORE BYTE WITH BIT SET
```

Similarly, resetting any of the eight bits of a memory byte calls for

```
LD    A,(HL)    ;LOAD THE MEMORY BYTE
AND   A,0FBH    ;RESET BIT 2
LD    (HL),A    ;STORE BYTE WITH BIT RESET
```

Lastly, testing a bit of a memory location requires a load and test, usually an AND

```
LD    A,(HL)    ;LOAD THE MEMORY BYTE
AND   A,4       ;TEST BIT 2
JP    Z,ZERO    ;GO IF BIT 2 = 0
JP    ONE       ;BIT 2 = 1
```

Bit Instructions

In the Z-80 only one instruction is required to set, reset, or test any one bit of a memory or cpu register bit. The instruction SET 2,(HL) takes the place of the three instructions for setting a bit, RES 2,(HL) causes a reset of bit 2, and BIT 2,(HL) sets the zero flag to the condition of the bit. Since these sets, resets, and tests are continually being done in assembly language programming, the bit instructions are quite powerful.

Shiftless Computers

It is possible to perform the actions of aligning data, dividing and multiplying by powers of two, and bit testing without shift instructions, but the Z-80 shifts are much more efficient than other shiftless instruction sets, and make these common operations much easier to perform.

Often shifts are used to align data, that is, to move fields within bytes to a desired location. The Z-80 shift instructions for data alignment are the *Rotate* instructions. Rotates are either 8-bit rotates or 9-bit rotates. The 8-bit rotates move the 8 bits within a register or memory location out one end and in the other, as shown in Figure 8-1. The 9-bit rotates rotate the carry along with the 8 register or memory data bits. Both types of rotates have their uses.

Rotates

As an example of use of rotate, let's write a routine that will output the contents of a block of memory locations in binary. Each memory location has eight bits, of course, and we must convert each bit to an ASCII one or zero for display. The following code outputs locations 0 through 0FH to the screen in binary ASCII.

```
                00100 ;ROUTINE TO DUMP IN BINARY
                00110 .
4A00            00120          ORG     4A00H
4A00 DD21283C   00130 START    LD      IX,3C00H+32     ;MIDDLE OF LINE 0
4A04 FD21004B   00140          LD      IY,4B00H        ;START OF DUMP LOC
4A08 113000     00150          LD      DE,56           ;LINE INCREMENT
4A0B 0610       00160          LD      B,16            ;LINE COUNT
4A0D D9         00170 LOOP1    EXX                     ;SWITCH REGISTERS
4A0E 0608       00180          LD      B,8             ;BIT COUNT
4A10 3E30       00190 LOOP2    LD      A,30H           ;ASCII 0
4A12 FDCB0006   00200          RLC     (IY)            ;ROTATE LEFT
4A16 3001       00210          JR      NC,LOOP3        ;GO IF 0
4A18 3C         00220          INC     A               ;CHANGE 0 TO 1
4A19 DD7700     00230 LOOP3    LD      (IX),A          ;STORE 0 OR 1
4A1C DD23       00235          INC     IX              ;NEXT CHARACTER POSTN
4A1E 10F0       00240          DJNZ    LOOP2           ;GO IF NOT 8 BITS
4A20 D9         00250          EXX                     ;SWITCH BACK
4A21 FD23       00260          INC     IY              ;BUMP LOCATION PNTR
4A23 DD19       00270          ADD     IX,DE           ;POINT TO NEXT LINE
4A25 10E6       00280          DJNZ    LOOP1           ;GO IF NOT 16 LOOPS
4A27 18FE       00290 LOOP4    JR      LOOP4           ;LOOP HERE ON DONE
0000            00300          END
00000 TOTAL ERRORS
LOOP4   4A27
LOOP3   4A19
LOOP2   4A10
LOOP1   4A0D
START   4A00
```

This is our most complicated program thus far, and it bears some detailed study. The IX register is used to point to the current screen line, starting at the middle of the first line. The IY register is used to point to the location to be *dumped*, in this case starting at 4B00. DE holds the line increment to be added to IX to point to the next display line. Since we're going to be writing out 8 ASCII bytes on each line, the increment on

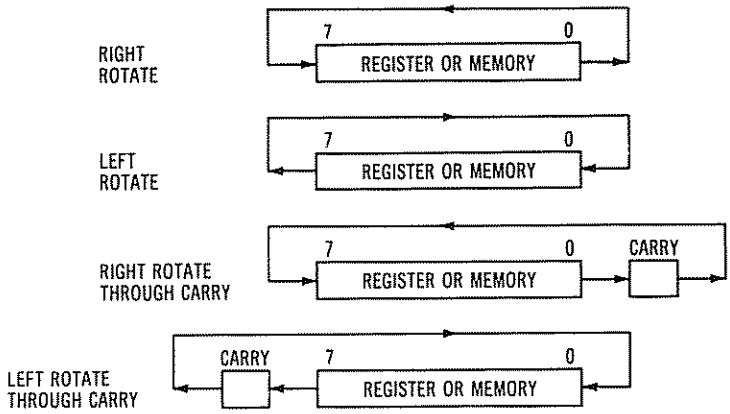


Fig. 8-1. Rotate operation.

this is (64-8) or 56. B is initialized with the number of locations to be dumped or 16.

The main loop in the code starts at LOOP1. The first instruction swaps the inactive and active set of cpu registers B through L. This is done to enable us to use more cpu registers since just about every one is in use. Now we can use B of the second set to hold a bit count for the inner loop of 4A10 through 4A1F that looks at the 8 bits and outputs them in ASCII to the screen. The inner loop rotates the location pointed to by IY. As each rotate is done, the leftmost bit is rotated both to the carry and around to the right-hand side of the memory location. The carry is tested to store either a 30H, for an ASCII zero, or 31H, for an ASCII one. Eight rotates are done, and at the end the memory location in the 4B00H area has been rotated completely around.

For each store of an ASCII one or zero, IX is incremented to point to the next character position on the line. When the count in B is decremented down to zero, an EXX switches back the cpu registers, restoring the original count in B for number of lines. IY is incremented to point to the next memory location in the 4B00H area, and IX is incremented by 56 to point to the next line for display. If 16 locations have not been dumped, the next location is stored as eight ASCII characters.

A program that has several nested loops such as this can be confusing to a programmer seeing it for the first time. It can also be confusing to the programmer who wrote it when he picks it up several months later! One convenient way to get a clear picture of what is going on in a program such as this is to "play computer." On a sheet of paper, make columns rep-

resenting the registers that are in use in the program. Then step through the program one instruction at a time, filling in the proper values in the registers. It isn't necessary to loop all the way through some of the loops (65536 loops makes for a lot of writing), but it does make many programs very clear. See Figure 8-2.

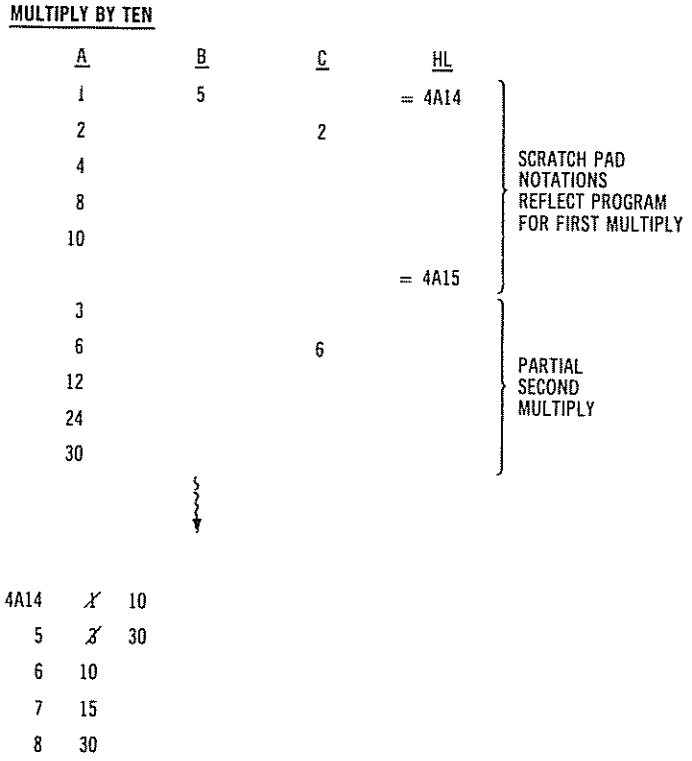
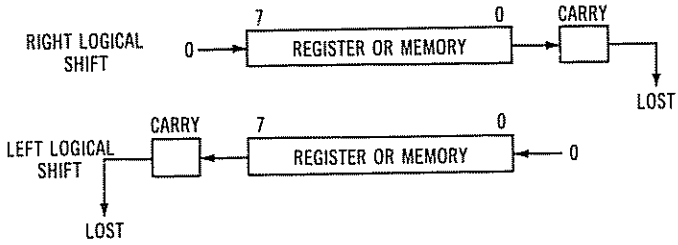


Fig. 8-2. Playing computer.

Some Shifting Is Very Logical

Logical shifts differ from rotates in that data shifted off the end of the register or memory location is lost. Zeros are used to shift into the byte from the other end, as shown in Figure 8-3. Logical shifts are used to align data as in the rotate case, and to divide or multiply by two. If an 8-bit value is shifted left one bit, the effect is to multiply the original value by two while a shift right of one bit position divides the original value by two and discards the remainder.



	SAMPLE RIGHT LOGICAL SHIFT		SAMPLE LEFT LOGICAL SHIFT	
ORIGINAL NUMBER	01010000	(80_{10})	00001011	(11_{10})
1 SHIFT	00101000	(40_{10})	00010110	(22_{10})
2 SHIFTS	00010100	(20_{10})	00101100	(44_{10})
3 SHIFTS	00001010	(10_{10})	01011000	(88_{10})
4 SHIFTS	00000101	(5_{10})	10110000	(-80_{10})!
5 SHIFTS	00000010	(2_{10})	01100000	(96_{10})!
6 SHIFTS	00000001	(1_{10})	11000000	(-64_{10})!
7 SHIFTS	00000000	(0)	10000000	(-128_{10})!

Fig. 8-3. Logical shift operation.

All rotates, logical shifts, and arithmetic shifts in the Z-80 operate only one bit at a time, so that a shift of four bit positions requires four separate shifts. To show how shifts may be used to multiply, consider the following code. Multiplication by ten is a common problem in many programs. For example, keyboard values may be input in ASCII and represent a string of decimal digits, such as 567.89, that must be converted to binary values for arithmetic manipulations within the program. MULTEN takes an 8-bit value from memory, multiplies it by ten, and stores it back into the memory location.

```
                    00100 ;MULTIPLY BY TEN ROUTINE
                    00110 .
4000                00120       ORG   4A00H
4A00 21154A         00130 MULTEN LD   HL,DATA      ;TABLE OF DATA
4A03 0605           00140       LD    B,5          ;FOR FIVE VALUES
4A05 7E             00150 LOOP  LD    A,(HL)       ;GET VALUE
4A06 CB27           00160       SLA   A            ;VALUE*2
4A08 4F             00170       LD    C,A          ;SAVE VALUE*2
```

```
4A09 CB27     00180        SLA   A          ;VALUE*4
4A0B CB27     00190        SLA   A          ;VALUE*8
4A0D 81       00200        ADD   A,C        ;VALUE*10
4A0E 77       00210        LD    (HL),A     ;RESTORE
4A0F 23       00220        INC   HL         ;POINT TO NEXT VALUE
4A10 10F3     00230        DJNZ  LOOP       ;CONTINUE
4A12 C3124A   00240 LOOP1  JP    LOOP1      ;LOOP HERE IF DONE
4A15 01       00250 DATA   DEFB  1
4A16 03       00260        DEFB  3
4A17 0A       00270        DEFB  10
4A18 0F       00280        DEFB  15
4A19 1E       00290        DEFB  30
0000          00300        END
00000 TOTAL ERRORS
LOOP1  4A12
LOOP   4A05
DATA   4A15
MULTEN 4A00
```

After the value is loaded into the A register it is shifted left by the SLA A to multiply the value by two. This value is then saved in the C register. Now the A register is shifted left two more times to multiply the original value by four and eight. Now the value in the C register, which represents the original value times two, is added to the value times eight to give a result of the value times ten. Execute the program with a breakpoint at 4A12 and then look at the table locations to see the results. Note that the multiply was an unsigned (absolute) multiply, and that in one case (30), the result was too large for the 8-bit memory location. In this case only the lower-order eight bits of the result are in the memory location!

Arithmetic Shifts

The TRS-80 has one shift that is an arithmetic-type shift (even though the mnemonic for the SLA is *S*hift *L*eft *A*rithmetic it is really a logical shift). The SRA (*S*hift *R*ight *A*rithmetic) always retains the *sign* of the operand to be shifted as shown in Figure 8-4. The bit in bit 7 is shifted right to bit 6,

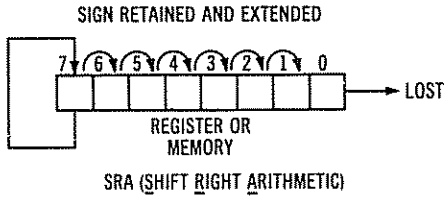


Fig. 8-4. Arithmetic shift operation.

but also goes back into bit 7 as the sign. The process is called *sign extension* as the sign is extended to the right. The SRA may be used to divide a signed 8-bit operand by two. The operation for a value of −37 is shown in Figure 8-5.

Software Multiply and Divide

What! No multiply and divide instructions in the Z-80! That's right, and no current 8-bit microprocessor has them either. Before you pull out that weathered four-function calculator, let's see how multiply and divide can be implemented in software.

There are a number of approaches in writing a multiply routine for any computer. The easiest is repetitive addition. Mulplying 63 by 15 is really only adding 63 to itself 14 times

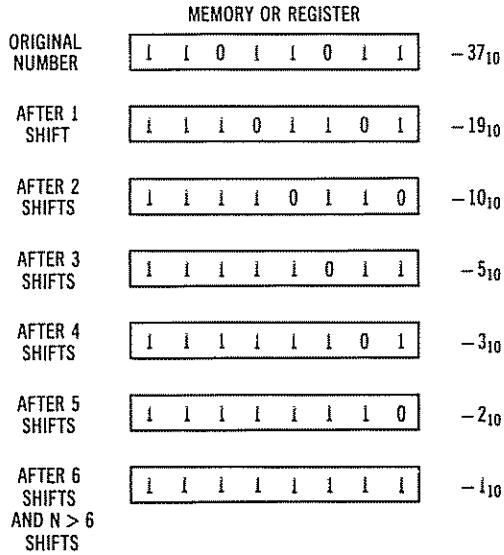


Fig. 8-5. Arithmetic shift example.

(or adding 15 63 times), and that's very easy to implement in the Z-80. The following routine uses this approach to multiply the 16-bit absolute value in DE by an 8-bit *multiplier* in B, which is also unsigned or absolute. The product is in HL at completion (use the R command to see the product).

```
                00100 ;REPETITIVE ADDITION MULTIPLY
                00110 .
4A00            00120           ORG     4A00H
4A00 ED5B144A   00130 START     LD      DE,(ARG1)       ;LOAD MULTIPLICAND
4A04 3A134A     00140           LD      A,(ARG2)        ;LOAD MULTIPLIER
4A07 47         00150           LD      B,A             ;TRANSFER TO B
4A08 210000     00160           LD      HL,0            ;CLEAR PARTIAL PRODUCT
4A0B 19         00170 LOOP      ADD     HL,DE           ;ADD MULTIPLICAND
4A0C 10FD       00180           DJNZ    LOOP            ;GO IF NOT DONE
4A0E C30E4A     00190 LOOP1     JP      LOOP1           ;LOOP HERE ON DONE
4A11 E803       00200 ARG1      DEFW    1000            ;PUT MULTIPLICAND HERE
4A13 1400       00210 ARG2      DEFB    20              ;PUT MULTIPLIER HERE
0000            00220           END
00000 TOTAL ERRORS
LOOP1   4A0E
LOOP    4A0B
ARG2    4A13
ARG1    4A11
START   4A00
```

As short and sweet as this routine is, it does have a serious disadvantage. It is horrendously slow, compared to other ways in which the multiply could be implemented. Use this approach only when the multiplier is small. It is efficient when multipliers of ten or less will be used.

The usual way of implementing a software multiply is to use the same approach as the pencil and paper method for decimal numbers. In this approach a shifted multiplicand multiplied by the digit in the multiplier is added to other partial products to get the final product as shown in Figure 8-6. Binary multiplication using this technique is fairly simple as the value to be added can *only be* the multiplicand or zero, depending upon the value of the multiplier bit. The following

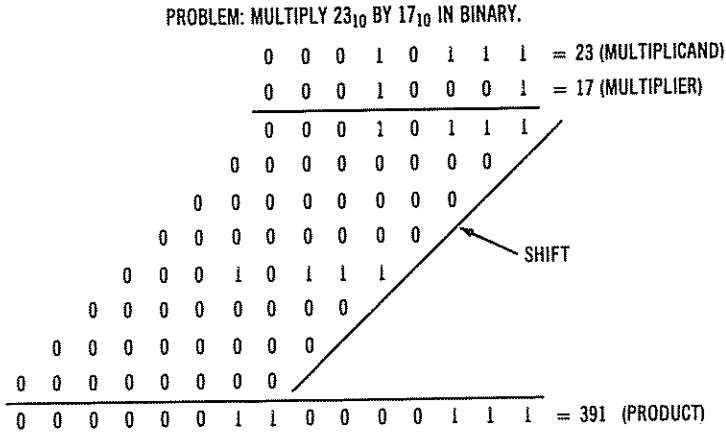


Fig. 8-6. Multiplication methods.

routine is one of the "standard" routines that might be helpful in the user's programs. It multiplies an *unsigned* 16-bit value in DE by an *unsigned* 8-bit value in the B register and returns the product in HL. The B register contents is zero upon return.

```
                00100 ;SUBROUTINE TO MULTIPLY 16 BY 8
                00110 ;   ENTRY:(DE)=MULTIPLICAND,UNSIGNED
                00120 ;         (B)=MULTIPLIER,UNSIGNED
                00130 ;         CALL  MUL16
                00140 ;   EXIT: (HL)=PRODUCT
                00150 ;         (DE)=DESTROYED
                00160 ;         (B)=0
                00170 ;
4000            00180           ORG   4000H       ;CHANGE ON REASSEMBLY
4000 210000     00190 MUL16     LD    HL,0        ;CLEAR PARTIAL PRODUCT
4003 CB38       00200 LOOP      SRL   B           ;SHIFT OUT M'IER BIT
4005 3001       00210           JR    NC,CONT     ;GO IF NO CARRY (1 BIT)
4007 19         00220           ADD   HL,DE       ;ADD MULTIPLICAND
4008 C8         00230 CONT      RET   Z           ;GO IF M'IER
4009 EB         00240           EX    DE,HL       ;MULTIPLICAND TO HL
400A 29         00250           ADD   HL,HL       ;SHIFT MULTIPLICAND
400B EB         00260           EX    DE,HL       ;SWAP BACK
```

```
49BC C3B34A   00270        JP      LOOP          ;CONTINUE
0000          00280        END
00000 TOTAL ERRORS
CONT   4988
LOOP   4983
ML16   4980
```

Note that in the above routine the ADD HL,DE *does not* affect the zero flag, allowing it to be used for a check on the shifted result in B *after* the add.

Divide routines implemented in software are not nearly so neat. Experienced programmers have been known to wail and gnash their teeth while trying to implement an efficient divide routine on certain computers. Successive subtraction may be used, but it is as slow as a multiply routine using this approach, and should be used only with operations resulting in small quotients. The following code divides the contents of the HL register, an *unsigned* 16-bit number by the contents of DE, an unsigned 16-bit *divisor*. Both numbers must be less than 32,768. The quotient is in B at the end, and any remainder is in HL. If the quotient is larger than 255 overflow will result.

```
              00100 ;DIVIDE BY SUCCESSIVE SUBTRACTION
              00110 .
4980          00120        ORG     4980H
4980 2A1A4A   00130 START  LD      HL,(ARG2)     ;GET DIVISOR
4983 E5       00140        PUSH    HL            ;TRANSFER TO DE
4984 D1       00150        POP     DE
4985 2A184A   00160        LD      HL,(ARG1)     ;DIVIDEND
4988 0600     00170        LD      B,0           ;CLEAR QUOTIENT
498A B7       00180 LOOP   OR      A             ;CLEAR CARRY FOR SUBTR
498B ED52     00190        SBC     HL,DE         ;DIVIDEND-DIVISOR
498D FA1A49   00200        JP      M,DONE        ;GO IF DONE
4810 04       00210        INC     B             ;BUMP QUOTIENT
4811 C38A49   00220        JP      LOOP          ;CONTINUE
4814 19       00230 DONE   ADD     HL,DE         ;FIND TRUE REMAINDER
4815 C3154A   00240 LOOP1  JP      LOOP1         ;LOOP HERE ON DONE
4818 204E     00250 ARG1   DEFW    20000         ;ARG1/ARG2
481A C800     00260 ARG2   DEFW    200
```

```
0000        00270       END
00000 TOTAL ERRORS
LOOP1  4015
DONE   4014
LOOP   4000
ORG1   4018
ORG2   401A
START  4000
```

In the routine the divisor is repeatedly subtracted from the dividend until the dividend goes negative. When this occurs, the *residue* is changed to a true remainder by adding back the divisor. Each time the subtraction can be successfully made the contents of B are incremented by one to show the quotient. This method exactly emulates what can be done with pencil and paper.

A more general-purpose divide for an unsigned 16-bit dividend and unsigned 8-bit divisor is shown in the following "standard" subroutine. Here the division is a *restoring* type similar to a paper and pencil approach. Instead of asking itself "Does the divisor go into the next group of digits," however, the computer in this case blindly goes ahead and attempts the divide. If the divisor doesn't go, then the previous residue is *restored* by adding back the shifted dividend, similar to what was done in the successive subtraction case.

```
                00100 ;SUBROUTINE TO DIVIDE 16 BY 8
                00110 .   ENTRY:(HL)=DIVIDEND 16 BITS
                00120 .         (D)=DIVISOR 8 BITS
                00125 .         CALL  DIV16
                00130 .   EXIT: (IX)=QUOTIENT 16 BITS
                00140 .         (H)=REMAINDER 8 BITS
                00150 .         (L)=DESTROYED
                00160 .         (D)=UNCHANGED
                00170 .         (E)=0
                00180 .         (A)=DESTROYED
                00190 .
4000            00200       ORG     4000H          ;CHANGE ON REASSEMBLY
4000 7D         00210 DIV16 LD      A,L            ;LS BYTE DIVDND
4001 6C         00220       LD      L,H            ;MS BYTE DIVDND
```

```
4A02 2600      00230        LD      H,0          ;CLEAR FOR SUBT
4A04 1E00      00240        LD      E,0          ;SETUP FOR SUBTRACT
4A06 0610      00250        LD      B,16         ;16 ITERATIONS
4A08 DD210000  00260        LD      IX,0         ;INITIALIZE QUOTIENT
4A0C 29        00270 LOOP   ADD     HL,HL        ;SHIFT DIVD LEFT
4A0D 17        00280        RLA                  ;SHIFT 8 LS BITS
4A0E D2124A    00290        JP      NC,LOOP1     ;GO IF 0 BIT
4A11 2C        00300        INC     L            ;SHIFT TO HL
4A12 DD29      00310 LOOP1  ADD     IX,IX        ;SHIFT QUOTIENT LEFT
4A14 DD23      00320        INC     IX           ;0 BIT=1
4A16 B7        00330        OR      A            ;CLEAR CARRY FOR SUB
4A17 ED52      00340        SBC     HL,DE        ;TRY SUBTRACT
4A19 D21F4A    00350        JP      NC,CONT      ;GO IF IT WENT
4A1C 19        00360        ADD     HL,DE        ;RESTORE
4A1D DD2B      00370        DEC     IX           ;SET 0 BIT=0
4A1F 10EB      00380 CONT   DJNZ    LOOP         ;GO IF NOT 16
4A21 C9        00390        RET                  ;RETURN
0000           00400        END
00000 TOTAL ERRORS
CONT    4A1F
LOOP1   4A12
LOOP    4A0C
DIV16   4A00
```

The register setup before and after the divide is shown in Figure 8-7. The divisor in D is repetitively subtracted from the residue of the dividend in HL. The residue is shifted over one bit position for every iteration just the way it is done by the paper and pencil method. If the subtract for any *iteration* is successful, a one bit is left in the quotient; if the subtract is not successful a zero bit is put in the quotient. The quotient is shifted left one bit for every iteration as less and less significant subtracts are made. After 16 bits the IX register holds the possible 16-bit quotient, the H register holds an 8-bit remainder, D holds the original divisor, and E is zeroed. One interesting point is that both the HL and IX registers are effectively shifted left one bit position in a logical shift by adding HL or IX to themselves. It may benefit the reader to actually play computer on this routine and step through the 16 iterations of the divide while using actual numeric values.

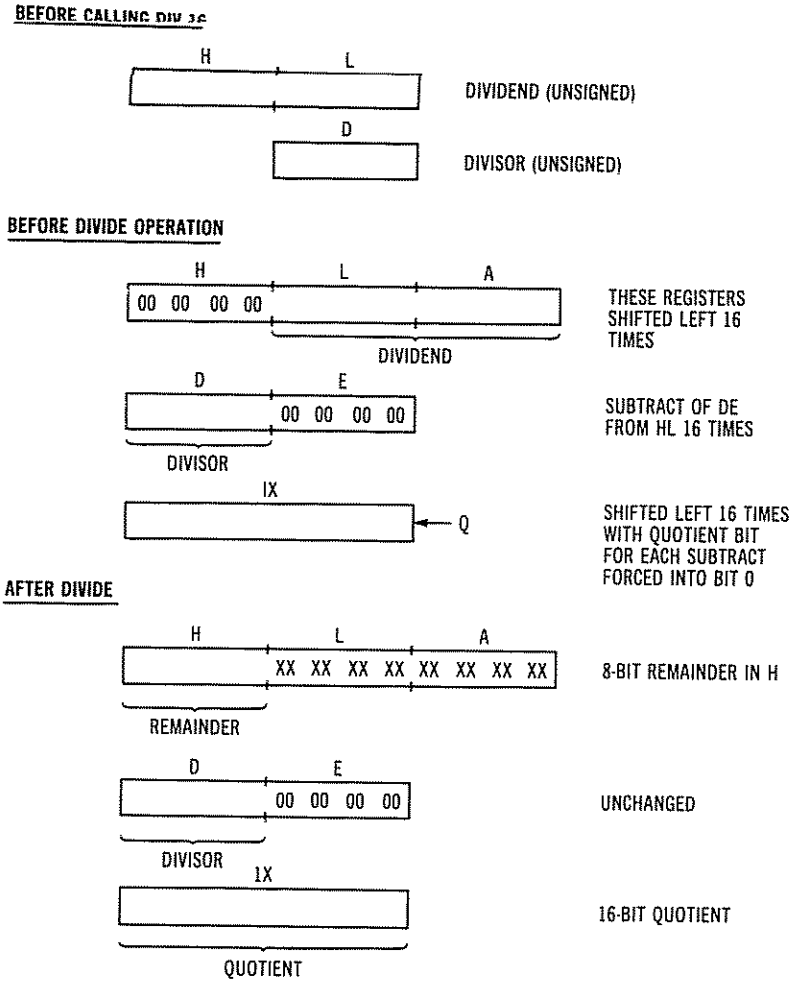


Fig. 8-7. Divide register setup.

At the end you may vow never to do it again, but it will give some insight into this type of operation.

The preceding multiply and divide routines are *unsigned* multiplies and divides. It is possible to implement *signed* multiplies and divides, but they are not as neatly packaged as the unsigned. The unsigned routines may be used to implement a signed multiply or divide if the operands are changed to their absolute values and the results changed again to their proper signs. However, watch for overflow conditions when this approach is used, such as multiplying −128 by −128!

Input and Output Conversions

The techniques of shifting, multiplications, and divides that we covered in this chapter are very useful in conversion between internal data representation and ASCII. Most programs require some type of input of ASCII data from keyboard, usually in decimal, and that string of decimal digits must be converted into an eight, sixteen, or larger number of bits so that the program can process the data. Sometimes, as in the case of T-BUG, there must be a way of converting from a *hexadecimal* ASCII input to eight or sixteen bits, and infrequently, a way of converting ASCII *binary* into internal data values. Similarly, once the data has been processed, it must be displayed in a more convenient form, which usually means ASCII decimal, but which may also be hexadecimal (in the T-BUG case) or binary.

We have already covered one conversion in an earlier program in this chapter, the conversion of eight bits into equivalent ASCII ones or zeros for display. The conversion of

```
                00100 ;SUBROUTINE TO CONVERT FROM HEX TO ASCII
                00110 ;
                00120 ;   ENTRY:(A)=8-BIT VALUE TO BE CONVERTED
                00130 ;         CALL  HEXCV
                00140 ;         (RETURN)
                00150 ;   EXIT: (HL)=TWO ASCII VALUES, HIGH AND LOW
                00160 ;         (A)=DESTROYED
                00170 ;         (C)=DESTROYED
                00180 ;
4000            00190         ORG   4000H         ;CHANGE ON REASSEMBLY
4000 4F         00200 HEXCV   LD    C,A           ;SAVE TWO HEX DIGITS
4001 CB3F       00210         SRL   A             ;ALIGN HIGH DIGIT
4003 CB3F       00220         SRL   A
4005 CB3F       00230         SRL   A
4007 CB3F       00240         SRL   A
4009 CD154A     00250         CALL  TEST          ;CONVERT TO ASCII
400C 67         00260         LD    H,A           ;SAVE FOR RTN
400D 79         00270         LD    A,C           ;RESTORE ORIGINAL
400E E60F       00280         AND   0FH           ;GET LOW DIGIT
```

```
4A10 CD154A   00290        CALL   TEST        ;CONVERT TO ASCII
4A13 6F       00300        LD     L,A         ;SAVE FOR RTN
4A14 C9       00310        RET
4A15 C630     00320 TEST   ADD    A,30H       ;CONVERSION FACTOR
4A17 FE3A     00330        CP     3AH         ;TEST FOR 0-9
4A19 FA1E4A   00340        JP     M,TEST1     ;GO IF 0-9
4A1C C607     00350        ADD    A,7         ;CORRECT FOR A-F
4A1E C9       00360 TEST1  RET                ;RETURN
0000          00370        END
00000 TOTAL ERRORS
TEST1  4A1E
TEST   4A15
HEXCV  4A00
```

hexadecimal values to ASCII digits of 0 through 9 and A through F is a similar problem. Let's write a program to convert any number in the A register into two hexadecimal ASCII digits. We will also write a simple *driver* to use the program and display some data.

Program HEXCV is the general-purpose routine to perform the conversion. The first four bits, representing the first hexadecimal digit are shifted 4 bit positions right in the A register. They are now aligned in the A register, and the register holds a value of from 0000 through 1111 (the upper four bits are zero), representing hexadecimal 0 through F. The ASCII equivalents for 0 through F are shown in Table 8-1. Unfortunately, there is a "gap" between the digits 0 through 9 and the characters A through F. If there were no gap, 30H could be added to the four bits to compute the ASCII value for the character. Since there is a gap, however, there must be a test for the hexadecimal letter digits, and this is done in the compare. If the conversion resulted in a result greater than 39H, then the ASCII character must be a letter, and 7 is added to obtain the letter value. For the second (least significant) hexadecimal digit, the A register is restored, the upper four bits are *masked out* (the lower four are already aligned) and the same conversion is made. Upon completion, HL holds the two ASCII characters representing the hexadecimal digits.

A simple *driver* to test this routine could be constructed from something similar to the following code.

```
        LD      B,8         ;8 LINES
        LD      IX,3C00H    ;FIRST LINE
        LD      DE,xxxx     ;LOCATIONS TO DISPLAY
LOOP    LD      A,(DE)      ;GET LOCATION
        CALL    HEXCV       ;CONVERT
        LD      (IX),H      ;STORE 1ST CHARACTER
        LD      (IX+1),L    ;STORE 2ND CHARACTER
        INC     IX          ;BUMP POINTER
        INC     IX
        INC     DE          ;BUMP LOCATION POINTER
        DJNZ    LOOP        ;CONTINUE IF NOT 8
```

A driver in this case means a routine to test and exercise the HEXCV routine. This driver displays 8 locations on line 1 of the video display in a string of 16 hexadecimal digits. The reader can undoubtedly see how different formats could be constructed to display hexadecimal data in a more convenient format by using HEXCV and other types of drivers.

Converting input data from ASCII to binary or hexadecimal is about as easy as the output conversion. For binary, the ASCII character representing a binary one or zero is converted to a true binary one or zero by subtracting 30H. This bit is then aligned and merged with other bits representing the 8- or 16-bit input value. Hexadecimal ASCII characters are adjusted by subtraction of 30H. If the result is greater than 9, a second subtract of 7 is performed to convert the letter digit to A through F in hexadecimal. The 4-bit result is merged with a second result or three other results to produce an 8-bit or 16-bit value.

Conversion of decimal data is the most difficult of the three types of cenversions. It is not simply a case of shifting bits as is the case in binary and hexadecimal.

For a conversion of decimal input data, each ASCII character represents a decimal digit from 0 through 9 (30H through 39H). The ASCII character is changed to four bits of bcd by subtracting 30H. Now this result must be multiplied by the power of ten it represents. For example, if the ASCII string was 123, the one would be converted to a bcd 1 and multiplied by 100, the 2 would be converted and multiplied by 10, and the 3 would only be converted. In practice decimal input conversion routines work with five digits as 65,535 can be held in 16 bits, and use a combination of conversion of each of the digits and multiplication of the result by ten for five iterations to convert the data.

For output conversions, an 8- or 16-bit value is converted by division by ten, the resulting remainders adjusted to an ASCII character by addition of 30H, and the result

stored in an intermediate buffer before output. Another approach is to use successive subtractions of the powers of ten, starting with 10000 (for a 16-bit value) to convert the number into decimal values which can then be converted by addition of 30H to ASCII outputs.

CHAPTER 9

Strings and Tables

This chapter discusses two important aspects of assembly-language programs, strings and tables. Strings are generally strings of text characters, just as in BASIC programs. Many assembly-language programs are concerned with separating segments of the string into various fields representing subdivisions of the string data such as names, addresses, mnemonics, and so forth. The Z-80 has a powerful block search capability to help in handling strings. Tables are generally one-dimensional arrays that represent such diverse things as addresses for jumps, sine values, and withholding tax percentages. The Z-80 has many features that permit the assembly-language programmer to work with tables, such as indexing.

Assembler-Generated Strings

We have seen in an earlier chapter how the assembler automatically generates a text string when the DEFM pseudo-op is used. Generally, this pseudo-op is used to produce messages which are output to the display or printer. The code below, for example, outputs a message to the middle of the screen, after the message has been converted from a symbolic source line into ASCII by the assembler.

```
                00100 ;ROUTINE TO OUTPUT MESSAGE
                00110 .
4000            00120       ORG    4000H
```

```
4A00 210E4A   00130 START  LD      HL,MESS        ;LOAD ADDRESS OF MESS
4A03 11203E   00140        LD      DE,3C00H+544   ;MIDDLE LINE+32
4A06 011100   00150        LD      BC,MESSL       ;LENGTH OF MESS
4A09 EDB0     00160        LDIR                   ;OUTPUT TO SCREEN
4A0B C30B4A   00170 LOOP   JP      LOOP           ;LOOP HERE ON DONE
4A0E 41       00180 MESS   DEFM    'ANOTHER FINE MESS'
4A0F 4E       4A10 4F      4A11 54      4A12 48      4A13 45      4A14 52
   4A15 20      4A16 46      4A17 49      4A18 4E      4A19 45      4A1A 2
0      4A1B 4D      4A1C 45      4A1D 53      4A1E 53      0011         00
165 MESSL  EQU    $-MESS
0000         00190        END
00000 TOTAL ERRORS
LOOP   4A0B
MESSL  0011
MESS   4A0E
START  4A00
```

The program uses the block move LDIR after setting up the register pairs for the parameters of the move. Note that the length of the message has been *generated by the assembler* by equating an assembly variable MESSL to the next assembler location minus the start of the message. When the BC register pair is loaded with MESSL, the assembler loads the immediate field of the load instruction with the length of 11H.

Generalized String Output

In the case above, the message could be moved to the output device in a block, as the output device was really a memory area. If your system has a printer that operates through a parallel or serial port on the TRS-80, the way that an output string is sent to the printer is somewhat different. Let's suppose that subroutine OUTPUT actually communicates with the printer (we'll talk about that communication in the next chapter). The subroutine below CALLs OUTPUT with the next ASCII character to be transmitted to the printer. The problem here is to determine when to stop. Initially, the MESSGE subroutine is called with HL holding the start of the message area. However, we need not only the start of the message area, but the end of the message area, the number of

bytes in the message, or some other means to signal the MESSGE subroutine that the message has come to an end. MESSGE here uses a *terminator* approach to detect the end of the message. The next character is sent to the OUTPUT subroutine as long as a *null* (all zeros) character is not detected. If a null is detected, MESSGE knows that the message area has come to an end and returns to the calling subroutine. A length could have been specified to MESSGE, but the terminator approach is used quite frequently.

```
                00100 ;MESSAGE OUTPUT ROUTINE
                00110 .
4000            00120       ORG     4000H
4000 7E         00130 START LD      A,(HL)      ;LOAD NEXT CHARACTER
4001 B7         00140       OR      A           ;TEST FOR NULL
4002 C8         00150       RET     Z           ;RETURN ON ZERO
4003 CD0050     00160       CALL    OUTPUT      ;OUTPUT TO PRINTER
4006 23         00170       INC     HL          ;POINT TO NEXT CHAR
4007 18F7       00180       JR      START       ;CONTINUE
5000            00190 OUTPUT EQU    5000H       ;PRINTER OUT ROUTINE
0000            00200       END
00000 TOTAL ERRORS
OUTPUT  5000
START   4000
```

In many cases, the message to be output to the screen or I/O device must first be assembled during program execution. In these cases, a *message buffer* area is allocated, and the component parts of the message are moved into the area, and the message is then printed. The approach is valuable for printing variable data that cannot be defined beforehand, and for saving memory when a large number of messages must be printed. In the code below, a message buffer for a mailing list has been defined. The *fields* of the buffer are defined by symbolic names and the execution time assembly can be done by transferring ASCII data to the proper fields.

```
                00100 ;MAILING LIST PRINT LINE
                00110 .
4000            00115       ORG     4000H
```

```
4000            00117 LABEL   EQU   $       ;START OF LABEL BUFFER
4000            00120 NAME    EQU   $       ;20 CHAR NAME HERE
0014            00130         DEFS  20      ;RESERVE 20
4014            00140 STREET  EQU   $       ;22 CHAR STREET HERE
0016            00150         DEFS  22      ;RESERVE 22
402A            00160 CITY    EQU   $       ;15 CHAR CITY HERE
000F            00170         DEFS  15      ;RESERVE 15
4039            00180 STATE   EQU   $       ;2 CHAR STATE HERE
0002            00190         DEFS  2       ;RESERVE 2
403B            00200 ZIP     EQU   $       ;5 CHAR ZIP HERE
0005            00210         DEFS  5       ;RESERVE 5
4040 00         00220         DEFB  0       ;NULL TERMINATOR
0000            00230         END
00000 TOTAL ERRORS
ZIP     403B
STATE   4039
CITY    402A
STREET  4014
NAME    4000
LABEL   4000
```

String Input

When strings are input from either the TRS-80 keyboard or from another type of I/O device, an *input buffer* is allocated to hold the string of characters in much the same way as the output message buffer is defined at assembly time. The problem with input of strings is not how to detect the end of the string, but to limit the number of input characters so that the space allocated for the input buffer is not exceeded. In the code below, the subroutine INPUT is called to input one character from an external keyboard. INPUT handles all of the communication between the TRS-80 in regard to *status* and transmission of the character. The input text string in ASCII is stored into INMESS, starting at 4B00H. The INPTMS routine is exited when either a carriage return (0DH) or 64 characters has been input. Terminating the routine at 64 characters guarantees that the message buffer will not overflow, possibly overwriting program code adjacent to it.

```
                00100 ;MESSAGE INPUT ROUTINE
                00110 ;
4A00            00120          ORG     4A00H
4A00 211B4A     00130 INPTMS   LD      HL,INMESS       ;START OF INPUT BUFFER
4A03 0640       00140          LD      B,64            ;MAXIMUM # OF CHARACTERS
4A05 CD004B     00150 LOOP     CALL    INPUT           ;GET ONE CHARACTER
4A08 FE0D       00160          CP      0DH             ;TEST FOR CARRIAGE RTN
4A0A C8         00170          RET     Z               ;RETURN IF CR
4A0B 77         00180          LD      (HL),A          ;STORE IN BUFFER
4A0C 23         00190          INC     HL              ;BUMP POINTER
4A0D 10F6       00200          DJNZ    LOOP            ;CONTINUE IF NOT 64
4A0F C9         00210          RET                     ;64 CHARACTERS
0040            00212 INMESS   DEFS    64
4B00            00213 INPUT    EQU     4B00H           ;TERMINAL INPUT
0000            00220          END
00000 TOTAL ERRORS
INPUT   4B00
LOOP    4A05
INMESS  4A1B
INPTMS  4A00
```

Once the string has been stored in the input buffer, of course, it must be separated into fields representing different types of data, as in the case of the mailing list line defined earlier. Conversion from ASCII data into decimal, hexadecimal, and other number representations must be performed. We've covered some of the conversion techniques for numbers earlier, but let us look at processing of the text strings that will be in the input message and may be carried through the entire processing of the program without being reformatted. The block move instructions allow shuffling of the strings from one place in memory to another, but the block search instructions perform an equally important task, comparison of one text string to another.

Block Compares

The block compare instructions, CPD, CPI, CPIR, and CPDR, search a block of memory (string) for a given char-

acter. If the character is found, the *location* of the character is returned. Since the search can be done in one instruction for the CPIR and CPDR, the search process is much faster on the Z-80 than on equivalent microprocessors. Let us see how the block compares operate. Suppose that we have just input a line of mailing list information using the INPTMS routine. The information input was in the format

JOHN J. PROGRAMMER/32768 OVERFLOW ST./COMPUTERTON/CA/92677

Here the fields of the mailing list information were separated by special characters called *delimiters*, which could have been any character normally not used in the text. To use the CPIR to search the input line for the next delimiter, the HL register pair is set up with the start of the message area, the BC register pair is set up with the number of bytes to be searched, and the A register is loaded with the character for which the search is to be done. The code below shows the initialization and the CPIR.

```
LD    HL,INMESS   ;INPUT MESSAGE START
LD    BC,64       ;64 CHARACTERS TO BE SCANNED
LD    A,'/'       ;SEARCH FOR SLASH
CPIR              ;PERFORM SEARCH
```

At the end of the search, the Z flag will be set if the character has been found, or reset if the character was not found in the entire block of memory. If the character *was* found, the HL register points to the location of the character *plus one*, and the HL register must, therefore, be decremented to point to the actual character. An actual example of this search would be the code below. Assemble and load using T-BUG, or key in using T-BUG, execute the program, and then display the registers using the R command. The Z flag should be set, and the HL register pair should contain 4A11H, the location of the slash plus one.

```
                00100 ;ROUTINE TO SEARCH FOR SLASH
                00110 ,
4000            00120       ORG   4000H
4000 21044A     00130 START LD    HL,INMESS   ;START OF MESSAGE AREA
4003 010600     00140       LD    BC,6        ;# OF CHARACTERS TO SCAN
4006 3E2F       00150       LD    A,'/'       ;SEARCH CHARACTER
4008 EDB1       00160       CPIR              ;SEARCH
400A C30040     00170 LOOP  JP    LOOP        ;LOOP HERE ON DONE
400D 31         00180 INMESS DEFM '12/ME'     ;MESSAGE
```

```
4AAE 32      4AAF 33      4AB0 3F      4AB1 4D      4AB2 45      0000
  00190        END
00000 TOTAL ERRORS
LOOP   4A00
INMESS 4A0D
START  4A00
```

The CPID works similarly to the CPIR, except that the CPID searches the string from end to beginning. In this case the HL register pair points to the character found minus one byte for the location. The HL register pair must be set up to the end of the string area in the CPID case.

```
LD      HL,INMESS+63   ;INPUT MESSAGE END
LD      BC,64          ;64 CHARACTERS TO BE SCANNED
LD      A,'/'          ;SEARCH FOR SLASH
CPDR                   ;SEARCH FOR SDRAWKCAB
JP      Z,FOUND        ;GO IF FOUND
...                    ;NOT FOUND HERE
```

The CPI and CPD instructions require the same setup as the CPIR and CPID, respectively. They operate in similar fashion to the block move instructions in that only one iteration is done at a time. The instruction then pauses so that additional operations can be performed. Suppose, for example, we wished to search for two characters in the search. The following code would do that by a CPI-type search. After each iteration the Z flag would be set if the search character was found, and the P/V flag would be set if the byte count in BC was counted down to zero and the search was over. In this case, if the Z flag is set the first character was found and a check is made for the second character, as the HL register pair now points to a location one past the found character. If the second character does not match, then the search is continued until the end. Upon completion the HL register pair should point to 4A1EH in this case.

```
                00100 ;ROUTINE TO SEARCH FOR '/*'
                00110 .
4A00            00120          ORG    4A00H
4A00 211B4A     00130 START    LD     HL,INMESS     ;START OF MESSAGE
4A03 010600     00140          LD     BC,6          ;# OF BYTES TO SEARCH
4A06 3E2F       00150 LOOPA    LD     A,'/'         ;SLASH FOR FIRST CHAR
4A08 EDA1       00160 LOOP     CPI                  ;SEARCH ONE BYTE
```

```
4000 2006     00170        JR     Z,MAYBE        ;GO IF FIRST FOUND
4002 EA0B4A   00180        JP     PE,LOOP        ;GO IF NOT DONE
4005 C3054A   00190 LOOP1  JP     LOOP1          ;LOOP HERE NOT FOUND
4008 3E2A     00200 MAYBE  LD     A,'*'          ;SECOND CHAR
400A BE       00210        CP     (HL)           ;COMPARE
400B C2064A   00220        JP     NZ,LOOPA       ;NO MATCH
400E C3184A   00230 LOOP2  JP     LOOP2          ;LOOP HERE IF FOUND
4011 23       00240 MESS   DEFM   '#$/*()'       ;MESSAGE
4012 24       4013 2F      4014 2A      4015 28      4016 29      0000
   00250       END
0000 TOTAL ERRORS
LOOP2  4018
LOOP1  4005
MAYBE  4008
LOOP   4002
LOOPA  4006
MESS   4011
START  4000
```

Searches for greater than one character may be done in this manner by searching for the first character using the search character in A for the CPI or CPD, and then searching the remainder of the string one byte at a time if there is a match on the first byte.

Table Searches

Tables are used extensively in all types of assembly-language programs. One of the simplest table types is a table of unordered or random data. The table is searched for a specific piece of data and the position in the table, or its *index*, is then used to access other information or simply as data itself.

Suppose, for example, that we have a table consisting of one-letter commands for T-BUG as shown in Figure 9-1. (In fact, this table is a kind of text string, as it is made up of ASCII characters.) We would like to see if we can find a given one letter command that has been input from the TRS-80 keyboard, match it up with a table entry, find the index, and then use that index to get the address of the routine to process that command in T-BUG.

The first thing that we must do is a table search, which in this case is exactly the same as the string search we performed under the string operations.

```
START   LD    HL,TABLE    ;TABLE START
        LD    BC,9        ;# OF BYTES
        LD    A,(INPUT)   ;GET INPUT CHARACTER
        CPIR              ;SEARCH
```

In the above code A was loaded with the input character from the keyboard, a one-letter ASCII command. At the end of the LDIR search Z will be set if the character was found and HL will then point to the character in the table plus one location. If the table is set up as in Figure 9-1, then HL will contain

TABLE	7 0	
4A10H	'B'	BREAKPOINT
4A11	'F'	RESTORE
4A12	'G'	CONTINUE
4A13	'J'	JUMP
4A14	'L'	LOAD CASSETTE
4A15	'M'	MEMORY DISPLAY
4A16	'P'	WRITE CASSETTE
4A17	'R'	DISPLAY REGISTERS
4A18	'X'	EXIT

Fig. 9-1. Sample table of T-BUG commands.

location 4A11H through 4A19H if the character was found and location 4A19H (with zero reset) if the character was not found. We can find the *index* of the command in the table by subtracting the value of table from the value in HL if the character was found.

```
JP    NZ,NFND   ;GO IF CHARACTER NOT FOUND
LD    BC,TABLE  ;START OF TABLE
OR    A         ;CLEAR CARRY FOR SUBTRACT
SBC   HL,BC     ;FIND INDEX
```

At the end of the code above, L will contain the index of 1 through 9. If "INPUT" was a G, for example, L will contain a 3, indicating that G was the third entry in the table, counting from the *zeroth* entry. Now that we have the index, what do we do with it? Well, we can now use that index to *index into* another table of jumps corresponding to the routines that process each of the T-BUG commands. The relationships of the two tables are shown in Figure 9-2.

In the case of the first command table, the *entries* of the table were one byte long, each byte being an ASCII character representing the command. In the address table, however, each

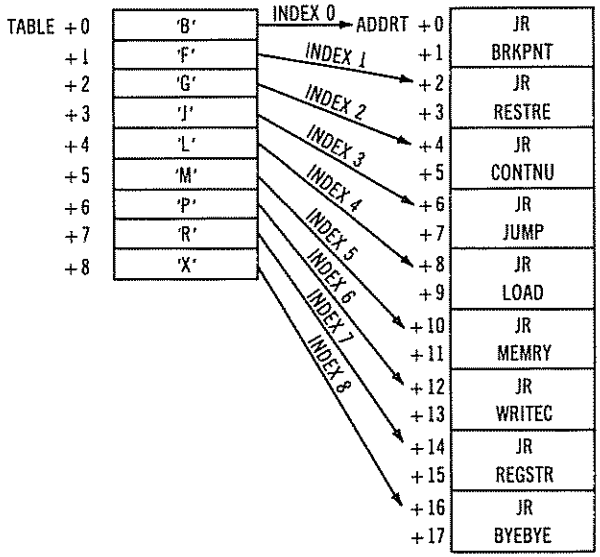


Fig. 9-2. Indexing into tables.

entry is two bytes long, since a relative jump must be represented. We now need to change that index from the first table into a *displacement* value that will pick up the right address table entry, the displacement being the number of physical bytes from the beginning of the address table. (The displacement for the first table was one times the index, but the displacement in the second table is two times the index.) The following code accomplishes this after first decrementing the index to adjust for the way the CPIR leaves the HL register.

```
DEC   HL        ;FIND TRUE INDEX
SLA   L         ;INDEX TIMES TWO
EX    DE,HL     ;SWAP DE AND HL
LD    HL,ADDRT  ;JUMP TABLE LOCATION
ADD   HL,DE     ;HL NOW HAS LOCATION OF JUMP
JP    (HL)      ;JUMP OUT TO JUMP
```

In the short tables here this code is not the most efficient (it took about 14 instructions to get to the routine), but the reader can see that this is a good approach for very long tables that are used in this fashion.

To recap the table structure, once again, a general table (see Figure 9-3) has a number of *entries*, each a certain *entry length*, and each having a displacement from the start of the table of entry length times # of entry.

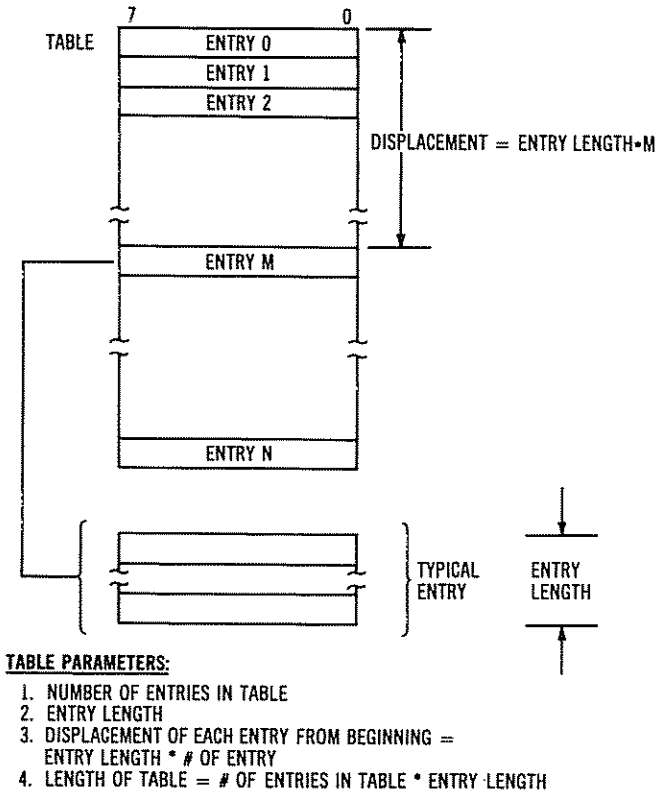


Fig. 9-3. General table structure.

Another method of using tables is to include the data associated with the *search key* in the entry itself, rather than in a separate table. Figure 9-4 shows this type of table. Each entry consists of a disc file name of 1 to 8 characters, a track number, and a sector number. The track and sector number always occupy the ninth and tenth bytes of each entry.

This table could be used to locate a specific *file* on disc by first searching the entire table for the correct file name, and then picking up the location of the file by the associated track and sector number when the file is found.

Unordered Tables

Tables in which the key entries are in random fashion are said to be unordered. When tables of this type are searched

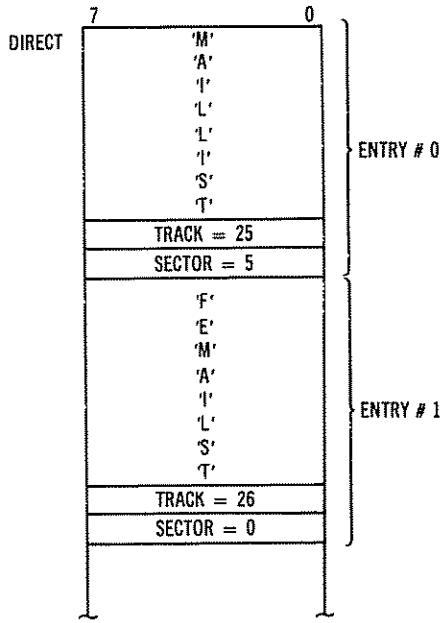


Fig. 9-4. Sample table of disc files.

for a specific entry, the minimum search occurs when the first entry is the desired entry and the maximum search occurs when the last entry is the one sought. The average number of entries that must be searched in this type of table is one-half the number of entries in the table. This type of table is fine for a small number of entries, but when the table must be continually searched and it holds a large number of entries, then a table with ordered entries could be used to a greater advantage.

The following program is another "standard" subroutine that the reader might find useful. It searches an unordered table from beginning to end for an 8-bit search key. Before the subroutine is called, A must be loaded with the search key, HL must be loaded with the start of the table, DE must be loaded with the length of each entry, and C must be loaded with the number of entries. If the entry is found, HL points to the entry upon return and the Z flag is set. If the entry is not found, the Z flag is not set upon return. The key in each entry is assumed to be the first byte.

```
                00100 ;SUBROUTINE FOR TABLE SEARCH
                00110 ;   ENTRY:(A)=KEY
                00120 ;         (HL)=TABLE START
                00130 ;         (DE)=LENGTH OF EACH ENTRY IN BYTES
                00140 ;         (C)=# OF ENTRIES IN TABLE
                00150 ;         CALL  SEARCH
                00160 ;   EXIT: Z FLAG SET IF FOUND,NOT SET IF NOT FOUND
                00170 ;         (HL)=LOCATION OF MATCH IF FOUND
                00180 ;         (BC)=CURRENT # LEFT
                00190 ;         (DE)=UNCHANGED
                00200 ;
4000            00210       ORG   4000H       ;CHANGE ON REASSEMBLY
4000 0600       00220 SEARCH LD   B,0         ;BC NOW HAS #
4002 EDA1       00230 LOOP   CPI              ;COMPARE A WITH (HL)
4004 CA0E40     00240        JP   Z,FOUND     ;GO IF FOUND
4007 E20F40     00250        JP   PO,NFND     ;AT END AND NOT FND
400A 19         00260        ADD  HL,DE       ;CURRENT+LENGTH+1
400B 2B         00270        DEC  HL          ;CURRENT +LENGTH
400C 18F4       00280        JR   LOOP        ;TRY AGAIN
400E 2B         00290 FOUND  DEC  HL          ;ADJUST TO FOUND LOC
400F C9         00300 NFND   RET              ;RETURN
0000            00310       END
00000 TOTAL ERRORS
NFND    400F
FOUND   400E
LOOP    4002
SEARCH  4000
```

Ordered Tables

Tables may be ordered in many different ways. The order may be *ascending* as in the sequence 1,3,5,6,7,10, . . . or descending as in the sequence 101,99,97,5,1,0. The keys used for ordering may be one byte or larger straight numeric values, or ASCII text strings. Tables that are ordered invariably require that new data must be merged into the existing order, existing entries deleted or modified, or that the entries

should be resorted. There have been literally thousands of books and articles written about the problems and approaches of sorting (ordering data), searching (finding data), and merging (merging in new data), and we may not cover all of it in this chapter. We will present *one* of the approaches to ordering data in a *list* of items, the *bubble sort*. Becoming familiar with the 16,387 other methods will be left up to the reader as an exercise.

The bubble sort orders data by comparing each entry in a list with the next entry of the list. If the next entry is a lower value, then the two entries are swapped. The next entry is then compared, and so on, until the end of the list is reached. If there has been *at least* one set of items swapped during the search of the list, then another pass is made, starting from the beginning. Passes continue until there have been no swaps made during the last pass, signifying that the list has been ordered. The code for the sort takes advantage of the indexing capability to swap the items, and is shown below.

```
                00100 ;BUBBLE SORT
                00110 .
4000            00120           ORG     4000H
4000 DD212B40   00130 LOOP      LD      IX,TABLE        ;TABLE START
4004 060F       00150           LD      B,15            ;NO OF LINES
4006 0E00       00160           LD      C,0             ;CHANGE FLAG
4008 DD7E00     00170 LOOP1     LD      A,(IX)          ;GET ENTRY
400B DDBE01     00180           CP      (IX+1)          ;TEST NEXT
400E CA1F40     00185           JP      Z,NOSWAP        ;GO IF EQUAL
4011 DA1F40     00190           JP      C,NOSWAP        ;GO IF NEXT LARGER
4014 DD4E01     00200           LD      C,(IX+1)        ;GET NEXT TO C
4017 DD7701     00210           LD      (IX+1),A        ;STORE CURRENT
401A DD7100     00220           LD      (IX),C          ;STORE NEXT
401D 0E01       00270           LD      C,1             ;SET CHANGE FLAG
401F DD23       00280 NOSWAP    INC     IX              ;POINT TO NEXT
4021 10E5       00290           DJNZ    LOOP1           ;DECREMENT LN CNT
4023 CB41       00300           BIT     0,C             ;TEST CHANGE
4025 C20040     00310           JP      NZ,LOOP         ;GO IF CHANGE
4028 C32840     00320 LOOP2     JP      LOOP2           ;DONE HERE
402B            00325 TABLE     EQU     $               ;PUT 16 ITEMS HERE
```

```
0000          00130          END
00000 TOTAL ERRORS
LOOP2   4026
NOSWAP  401F
LOOP1   4008
TABLE   4028
LOOP    4000
```

Assemble and load the program using T-BUG, or key in the program using T-BUG. TABLE can be filled with any number of data items that the reader desires, in any order. When a breakpoint at LOOP2 is reached, the table will have been reordered so that it is in ascending order, and the bubbles will have done an effective job in cleaning some of that RAM memory area. The reader may wish to breakpoint at the JP NZ,LOOP before LOOP2 to investigate the intermediate sorting after each pass. Use an "F" command and a "G" after looking at the table data, if breakpointing.

For another display of the bubble sort, use the program below. First use the M command in T-BUG to fill screen memory locations 3C20, 3C60, 3CA0, 3CE0, 3D20, 3D60 . . . 3FE0 with alphabetic or other characters in random order. A suggested sequence is shown in Table 9-1. You will see the characters appear in the middle of the screen as you fill them in. Now run the program, and you will see a literal graphic display of the bubble sort implementation.

```
                00100 ;BUBBLE SORT TO DISPLAY
                00110 .
4000            00120          ORG     4000H
4000 DD212C3C   00130 LOOP     LD      IX,3C60H+32     ;FIRST LINE, MIDDLE
4004 114000     00140          LD      DE,64           ;LINE INCREMENT
4007 060F       00150          LD      B,15            ;NO OF LINES
4009 0E00       00160          LD      C,0             ;CHANGE FLAG
400B DD7E00     00170 LOOP1    LD      A,(IX)          ;GET ENTRY
400E DDBE40     00180          CP      (IX+64)         ;TEST NEXT
4011 CA2840     00185          JP      Z,NOSWAP        ;GO IF EQUAL
4014 D22840     00190          JP      C,NOSWAP        ;GO IF NEXT LARGER
```

```
4017 DD4E40    00200        LD    C,(IX+64)     ;GET NEXT TO C
401A DD7740    00210        LD    (IX+64),A     ;STORE CURRENT
401D DD7100    00220        LD    (IX),C        ;STORE NEXT
4020 210000    00230        LD    HL,0          ;DELAY
4023 23        00240 LOOPD  INC   HL
4024 CB7C      00250        BIT   7,H           ;TEST FOR COUNTDOWN
4026 CA234A    00260        JP    Z,LOOPD       ;GO FOR DELAY
4029 0E01      00270        LD    C,1           ;SET CHANGE FLAG
402B DD19      00280 NOSWAP ADD   IX,DE         ;POINT TO NEXT LN
402D 100C      00290        DJNZ  LOOP1         ;DECREMENT LN CNT
402F CB41      00300        BIT   0,C           ;TEST CHANGE
4031 C2004A    00310        JP    NZ,LOOP       ;GO IF CHANGE
4034 C3344A    00320 LOOP2  JP    LOOP2         ;DONE HERE
0000           00330        END
00000 TOTAL ERRORS
LOOP2  4034
LOOPD  4023
NOSWAP 402B
LOOP1  400B
LOOP   4000
```

Table 9-1. Bubble Sort Sample Data

Display Memory Location	Contents
3C20H	46H
3C60	45
3CA0	44
3CE0	43
3D20	42
3D60	41
3DA0	39
3DE0	38
3E20	37
3E60	36
3EA0	35
3EE0	34
3F20	33
3F60	32
3FA0	31
3FE0	30

CHAPTER 10

I/O Operations

In this chapter we will rush in where many programmers fear to tread and describe some simple I/O operations in the TRS-80. I/O programming is intimately tied to the hardware configuration of a system, and for that reason some people are somewhat afraid of it, but we hope that the reader will find at the end of the chapter that it is really not that difficult. To lay the groundwork to discuss I/O programming we will review the *memory and I/O* mapping of the TRS-80. Then we will discuss the keyboard, display, cassette, and real-world applications, such as controlling the lawn sprinklers or your electric toothbrush.

Memory Versus I/O

In the first part of the book we talked somewhat about the architecture of the TRS-80. We mentioned that the TRS-80 has 64K or 65,536 bytes of memory available to it and explained how the memory was broken down into ROM, dedicated I/O addresses, and RAM as shown in Figure 10-1. The area that we will be considering in this chapter will be the central area of the figure, the dedicated I/O addresses, together with 256 I/O *ports*.

Let us expand that dedicated I/O address area and see what I/O devices are involved. Figure 10-2 shows that most of the area is devoted to display memory. Anytime that locations 3C00H through 3FFFH are addressed we are communicating with display memory, and that memory looks very similar to

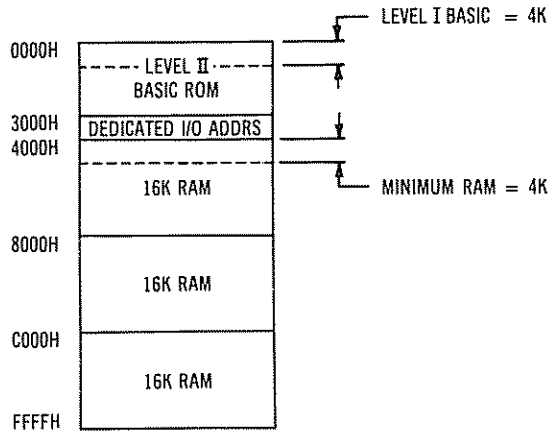


Fig. 10-1. Memory mapping with I/O addresses.

other RAM. We have been using display memory for many of the programs in previous chapters, and the reader should be very familiar with display memory at this point.

The section of dedicated memory from 3800H through 3BFFH is devoted to *keyboard addressing*. In this area memory does not exist, as it does for the display. When a location in this area is addressed, the keys of the TRS-80 keyboard are actually addressed. Addressing location 3801H addresses

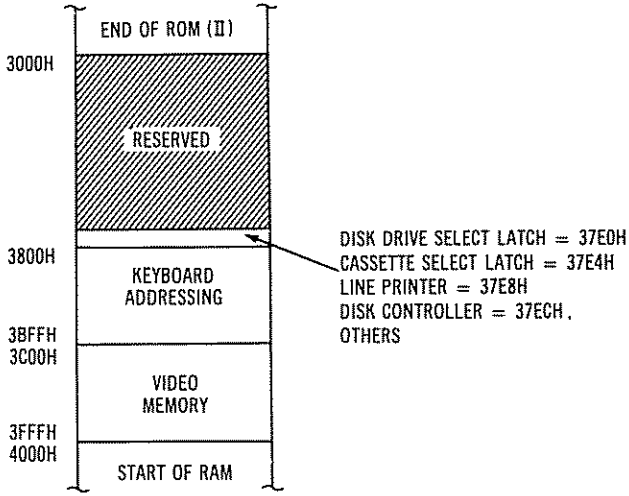
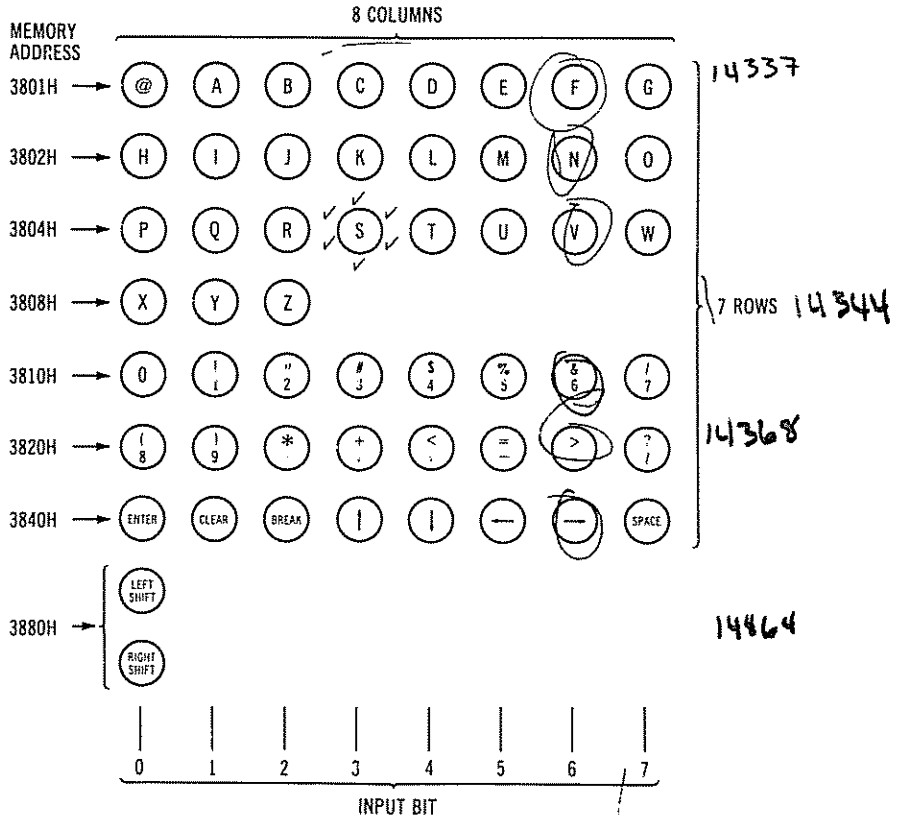


Fig. 10-2. Dedicated memory addresses.

the first row of keys, from "@" to "G", addressing location 3802H addresses the second row of keys from "H" to "O," and so forth, as shown in Figure 10-3. It turns out that there are eight addresses that address the keyboard, and they are 3801H, 3802H, 3804H, 3808H, 3810H, 3820H, 3840H, and 3880H. Every time a load is performed with one of these addresses 8 bits from the columns are loaded into the cpu register, as shown in Figure 10-3. These bits represent keys being pressed (1 bit) or not pressed (0 bit). We will discuss keyboard I/O a little later.



EXAMPLE: IF "S" IS PRESSED INPUT BYTE WILL BE 08H FOR ADDRESSING LOCATION 3804H. ALL OTHER INPUTS WILL YIELD 00H FOR INPUT BYTE.

Fig. 10-3. Keyboard addressing.

The remaining area of the dedicated memory addresses are used for such things as the line printer, floppy disc controller, and cassette select. Most of this area is reserved for future use (3000H through 37DDH). Addressing locations in the addresses above 37DDH enable communications with appropriate I/O devices. Loading a register from "memory" location 37E8H, for example, actually loads the register with eight bits of *status* for the system line printer, if one is attached. The status is a byte that is transmitted by the line printer that indicates whether the line printer is *ready* for the next character, whether it is *on-line*, and whether it has enough paper. Storing a register to location 37E8H actually transmits a byte of data, assumed to be an ASCII character, to the line printer for printing, in exactly the way a character is sent to a normal memory location to be stored.

For all intents and purposes, then, there is no practical difference in addressing a memory location in RAM or display memory and addressing an I/O device, as long as the I/O device is connected in such a manner as to look for that address and respond in the same manner that a memory location would respond.

Along with the memory addressing area devoted to system I/O devices, the TRS-80 has 256 other addresses that are devoted to I/O. These are the addresses used when an I/O instruction is executed. They differ from a memory address in that a signal goes out to all parts of the system that essentially says "here is an *I/O* address of 00000000 through 11111111." That signal is *not* present when a memory address is used (instead another signal goes out that says "here is a memory address of 16 bits").

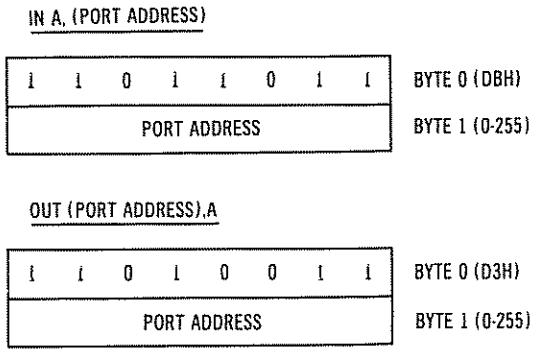


Fig. 10-4. I/O instruction format.

The general form of the I/O instruction is shown in Figure 10-4. There are several other formats, but we will be using these two in the rest of this chapter. The second byte of the instruction is the *port address* of 0 through 255. When an OUT instruction is executed, 8 bits of data from the A register are sent out to the system along with a signal that says "here is an I/O address" and the actual 8 bits of the port address itself. In a large system there could be many devices attached to the system *bus* (collection of data, address, and control signals), and they would all be continually looking for the I/O signal, *their* unique address (one of the 256), and the data to be received (or sent). See Figure 10-5.

In most configurations of the TRS-80, the only device that is attached in this port fashion is the cassette recorder. Logic on the cpu board is continually looking for port address FFH and the I/O signal indicating that an I/O instruction is being executed. If the instruction is an input (IN), the cassette

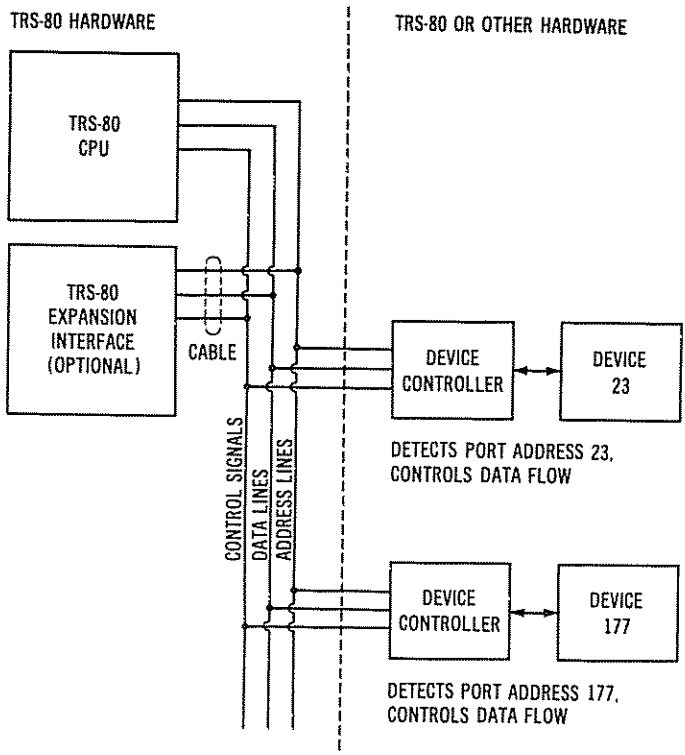


Fig. 10-5. I/O ports and port addressing.

logic will send a byte of data to the A register. Seven of those bits will be zeros, with only the most significant bit being active. If the instruction is an output (OUT), the contents of the A register will be sent to the cassette logic. Only the four least significant bits will cause actions in the logic.

The TRS-80 is expandable so that additional ports can be used by external devices, as long as the port addresses do not conflict with FFH or other port addresses used by TRS-80 devices. Since there are 256 total port addresses, however, there is a great deal of room for expansion, and conceivably the TRS-80 could be used to control dozens of functions such as home heating and lighting, burglar alarms, and others limited only by the user's imagination (and bank account).

Keyboard Decoding

Refer back to Figure 10-3. The keyboard is set up in eight rows and eight columns as shown in the figure. If a key is pressed, then the corresponding bit for that column becomes a one, and if the associated row address is read by a load instruction, then the column *byte* that is loaded will contain a one bit for the column of the key. As the program knows which row of the eight is being addressed when the one bit appears, it knows the key junction from the row and column. This type of I/O operation is called *matrix* decoding as the keyboard forms an eight-by-eight matrix.

The following program continually *scans* the keyboard and waits for a one bit to appear for the first and second rows (characters @, A through 0). When a one does appear, the row and column is computed to give an index of 0 through 15. This index is then used to *look up* the corresponding character in a sixteen-byte *look-up table*. The character is then printed on the screen.

```
                00100 ;KEYBOARD SCAN ROUTINE FOR FIRST TWO ROWS
                00110 ;
4A00            00120          ORG     4A00H
4A00 0E00       00130 KEYSCN   LD      C,0             ;FOR FIRST ROW
4A02 3A0138     00140          LD      A,(3801H)       ;1ST ROW ADDRESS
4A05 B7         00150          OR      A               ;TEST ZERO OR NON-ZERO
4A06 C2124A     00160          JP      NZ,KEY10        ;GO IF KEY PRESSED
4A09 3A0238     00170          LD      A,(3802H)       ;2ND ROW ADDRESS
4A0C B7         00180          OR      A               ;TEST ZERO OR NON-ZERO
```

```
4A0D C2004A   00190        JP    Z,KEYSCN      ;GO IF NO KEY
4A10 0E08     00200        LD    C,8           ;FOR 2ND ROW
4A12 06FF     00210 KEY10  LD    B,0FFH        ;INDEX
4A14 04       00220 KEY20  INC   B             ;DECREMENT INDEX
4A15 CB3F     00230        SRL   A             ;SHIFT 'TIL ZERO
4A17 C2144A   00240        JP    NZ,KEY20      ;GO IF NOT ZERO
4A1A 78       00250        LD    A,B           ;GET # 0-7
4A1B 81       00260        ADD   A,C           ;ADD ROW #
4A1C 4F       00270        LD    C,A           ;INDEX 0-15 TO C
4A1D 0600     00280        LD    B,0           ;ZERO B FOR ADDR
4A1F 21344A   00290        LD    HL,TABLE      ;TABLE OF CHARACTERS
4A22 09       00300        ADD   HL,BC         ;COMPUTE DISPLACEMENT
4A23 7E       00310        LD    A,(HL)        ;GET CHARACTER
4A24 32003E   00320        LD    (3C00H+512+32),A  ;DISPLAY
4A27 0E0A     00330        LD    C,10
4A29 0600     00340 LOOP   LD    B,0           ;DELAY ABOUT 17 MILLISEC
4A2B 10FE     00350 LOOP1  DJNZ  LOOP1
4A2D 0D       00360        DEC   C
4A2E C2294A   00370        JP    NZ,LOOP
4A31 C3004A   00380        JP    KEYSCN        ;GO FOR NEXT KEY
4A34 40       00390 TABLE  DEFM  '@ABCDEFGHIJKLMNO'
4A35 41       4A36 42      4A37 43.     4A38 44      4A39 45      4A3A 46
   4A3B 47      4A3C 48      4A3D 49      4A3E 4A      4A3F 4B      4A40 4
C      4A41 4D      4A42 4E      4A43 4F      0000         00400        EN
D
00000 TOTAL ERRORS
LOOP1   4A2B
LOOP    4A29
TABLE   4A34
KEY20   4A14
KEY10   4A12
KEYSCN  4A00
```

The A register is loaded with the contents of row 1 by addressing 3801H. If this is zero, the next row, 3802H, is addressed. If either row has at least one bit, the rest of the

program is executed, otherwise the program loops back to KEYSCN to scan the rows again. If a one bit has been detected, the C register holds either 0 for row one or 8 for row 2. The A register holds the column bit corresponding to the key column. As this is a power of two (80H, 40H, 20H, 10H, 8H, 4H, 2H, or 1H) it must be converted to a number representing the column of 0 through 7. This is done by shifting A until it becomes zero, and keeping a count of the number of shifts. 80H will require 7 shifts, for example, before A becomes 0. At the end of the shifting B holds the column number. This is added to the row number of 0 or 8 to produce an index of 0 through 15. This index is then added to the address of TABLE to point to the corresponding character in the table. This character is picked up and displayed on the center of the screen. LOOP is a *timing loop* to *debounce* the key so that the program does not loop back to the *same* key depression and output a spurious character (the same character twice or a number of times).

Although this program works only with the first two rows of keys, the reader can see how it can be expanded to work with *all* keys on the keyboard, and he will find a similar program in Level I or II BASIC.

Display Programming

We have used programs that output both ASCII and graphics characters to the screen, but have not discussed the graphics capabilities of the TRS-80 in any detail. The display memory is similar to normal RAM memory, except that each address of the 1024 bytes of display memory is made up of seven instead of eight bits, as shown in Figure 10-6. As the reader knows from his BASIC experiences, the display can display upper case alphanumeric and special characters *or* graphics characters, intermixed in any combination. The most significant bit of the 7-bit display memory is used to mark a graphics character. If this bit is a zero, then the remaining six bits define an alphanumeric or special character. If the most significant bit is a one, then the other six bits define a graphics character. The ASCII codes for alphanumeric and special characters are defined in the Editor/Assembler manual or the TRS-80 BASIC manual.

The graphics codes define a six-element graphics character that occupies one character position on the screen. As there are 1024 character positions (64 characters per line and 16 lines), there are 6144 graphics elements on the screen, ar-

ranged in a 128 by 48 matrix. The question arises of how one sets or resets a single element. There is no corresponding assembly-language SET or RESET command as there is in BASIC.

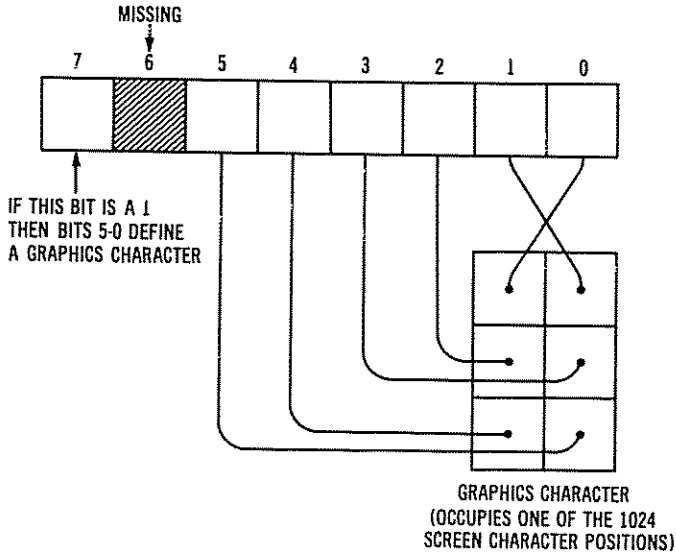


Fig. 10-6. Display memory format.

The following code attempts to solve the problem of converting an x,y coordinate into the proper bit position in the graphics memory cell. There are three entry points in the routine. The first entry point sets the *pixel* (element) corresponding to the given x,y (horizontal, vertical) position. The second entry point resets the pixel corresponding to the given x,y position, and the third entry point tests the current on/off status of the pixel, returning the zero/non-zero status in the zero flag. The three entry points of SET, RESET, and TEST all converge to a common location at TEST10. The store at TEST10 stores the second byte or a SET, RES, or BIT instruction at location INST+1. The first byte of all three instructions are the same, a CBH. The second byte is complete except for a three-bit field defining the bit to be set, reset, or tested. This will be calculated in the main body of the routine, along with the location in screen memory to be used, which will be put into HL. All three instructions use HL as a register pointer. See Figure 10-7.

The main body of the code converts an x,y location into a screen memory location and bit position. The bit position is merged into INST+1 to set the proper field. The memory location is retained in HL for the instruction. The actual *algorithm* works like this: The y position of 0-47 is converted to a line number by dividing by 3 to give 0 through 15. The remainder is saved. The x position is divided by 2 to give the character position along the line. We now have a line number of 0 through 15 and a character position of 0 through 63. If the line number is multiplied by 64 and the character position added to it, we will have the *byte displacement* from the start of screen memory, as shown in Figure 10-8. The actual location can then be found by adding 3C00H, the start of display memory.

The only remaining task is to find the bit position of the pixel to be set, reset, or tested. This is given by the remainder of the Y/3 operation times 2 plus the remainder of the X/2 operation. This value is stored in the bit position field of the instruction at INST+1. As a last step, bit 7 is set to ensure that all character positions processed will be graphics characters.

The code for this problem is somewhat complex and it may help the reader to "play computer" by actually using some values of x and y and working through the routine to find

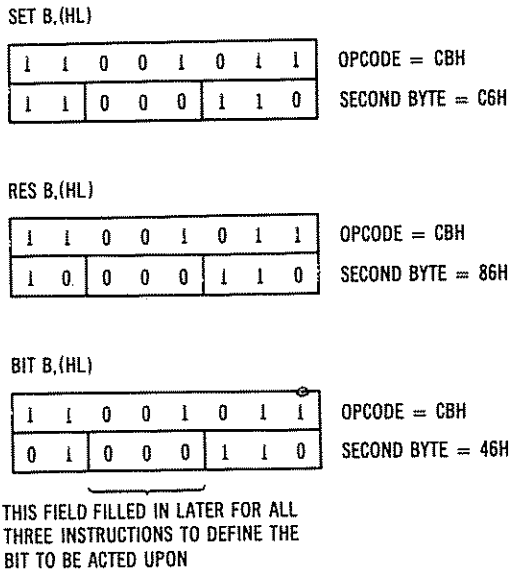
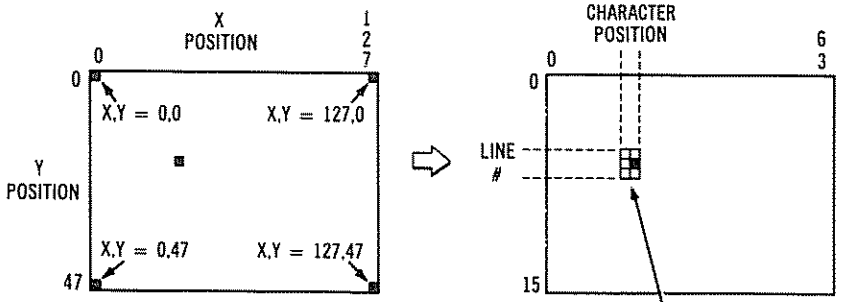


Fig. 10-7. Modifying instructions.



FOR ANY X,Y:

1. LINE # $= \frac{Y}{3}$ QUOTIENT
2. ROW # OF GRAPHICS CHARACTER $= \frac{Y}{3}$ REMAINDER
3. CHARACTER POSITION $= \frac{X}{2}$ QUOTIENT
4. COLUMN # OF GRAPHICS CHARACTER $= \frac{X}{2}$ REMAINDER
5. BYTE DISPLACEMENT FROM START OF SCREEN MEMORY = (LINE #) *64 + CHARACTER POSITION
6. ACTUAL LOCATION IN MEMORY = (LINE #) *64 + CHARACTER POSITION +3C00H
7. BIT POSITION WITHIN GRAPHICS CHARACTER = (ROW #) *2 + COLUMN NUMBER

Fig. 10-8. Screen coordinate algorithm.

out how it works. An interesting point is that the instruction at INST has been treated as another piece of data to be processed and modified. It is not a good practice to do this in some types of programming (for example, where interrupts are involved), but it is perfectly permissible in many *stand-alone* programs of this type.

```
00100 ;SUBROUTINE TO CONVERT SCREEN COORDINATES
00110 .   ENTRY: (DE)=Y,X COORDINATES OF POINT
00120 .          X=0 TO 127; Y=0 TO 47
00130 .       CALL  SET      ;SETS POINT
00140 .       CALL  RESET    ;RESETS POINT
00150 .       CALL  TEST     ;TESTS POINT RETURNS Z FLAG
00160 .   EXIT: (A THROUGH L)=DESTROYED
```

```
                00170 ;         Z FLAG SET IF TEST
                00180 .
4A00            00190           ORG     4A00H
4A00 3EC6       00200 SET       LD      A,0C6H          ;SET B,(HL) INSTRUCTION
4A02 1806       00210           JR      TEST10          ;GO TO STORE
4A04 3E86       00220 RESET     LD      A,86H           ;RES B,(HL) INSTRUCTION
4A06 1802       00230           JR      TEST10          ;GO TO STORE
4A08 3E46       00240 TEST      LD      A,46H           ;BIT B,(HL) INSTRUCTION
4A0A 32D4A      00250 TEST10    LD      (INST+1),A      ;STORE 2ND BYTE
4A0D 7A         00260 ADDRES    LD      A,D             ;GET Y
4A0E 06FF       00270           LD      B,0FFH          .-1
4A10 04         00280 LOOP      INC     B
4A11 D603       00290           SUB     3               .  BY THREE
4A13 F2104A     00300           JP      P,LOOP          ;GO IF NOT MINUS
4A16 C603       00310           ADD     A,3             ;YQ IN B, YR IN A
4A18 CB27       00320           SLA     A               ;YR*2
4A1A 4F         00330           LD      C,A             ;SAVE YR*2
4A1B 68         00340           LD      L,B             ;YQ TO L
4A1C 2600       00350           LD      H,0             ;YQ IN HL
4A1E 0606       00360           LD      B,6             ;CNT FOR MULTIPLY BY 64
4A20 29         00370 LOOP1     ADD     HL,HL           ;YQ*2
4A21 10FD       00380           DJNZ    LOOP1           ;GO IF NOT YQ*64
4A23 1600       00390           LD      D,0             ;DE NOW HAS X
4A25 CB3B       00400           SRL     E               ;XQ
4A27 3001       00410           JR      NC,CONT         ;GO IF XR NE 1
4A29 0C         00420           INC     C               ;C NOW HAS YR*2+XR
4A2A 19         00430 CONT      ADD     HL,DE           ;HL NOW HAS YQ*2+XQ
4A2B 113D3C     00440           LD      DE,3C00H        ;START OF DISPLAY
4A2E 19         00450           ADD     HL,DE           ;HL NOW HAS DISPLACE
4A2F CB21       00460           SLA     C               ;ALIGN TO FIELD
4A31 CB21       00470           SLA     C
4A33 CB21       00480           SLA     C
4A35 3AD04A     00490           LD      A,(INST+1)      ;GET INSTRUCTION
4A38 81         00500           ADD     A,C             ;SET FIELD
```

```
4A29 32204A   00510        LD      (INST+1),A      ;STORE
4A2C CB       00520 INST   DEFB    0CBH            ;PERFORM BIT,SET,RES
4A2D 00       00530        DEFB    0               ;WILL BE FILLED IN
4A2E C0FE     00540        SET     7,(HL)          ;FOR GRAPHICS
4A40 C9       00550        RET
0000          00560        END
00000 TOTAL ERRORS
CONT    4A2A
LOOP1   4A20
LOOP    4A10
ADDRES  4A00
INST    4A2C
TEST    4A00
RESET   4A04
TEST10  4A0A
SET     4A00
```

Mysteries of the Cassette Revealed

The cassette of a one cassette system is controlled by three bits of a 4-bit *latch* in the cpu. The latch is simply another type of memory, which happens to be four bits wide instead of the usual eight. When the cassette is addressed by performing an OUT instruction to port address 0FFH, the cassette latch is loaded with four bits of data as shown in Figure 10-9.

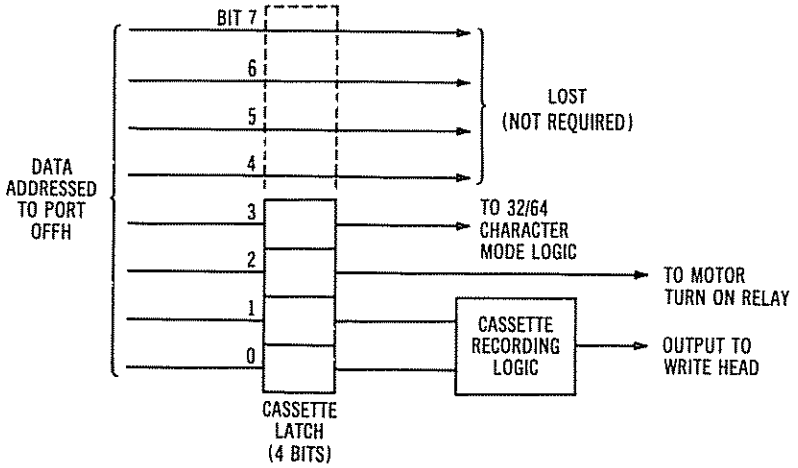


Fig. 10-9. Cassette latch transfers.

The other four bits of data, bits 7 through 4 are discarded into the bit bucket on the floor near the TRS-80.

Bit number 3 of the latch controls the 32- and 64-character mode of the TRS-80. Outputting a one to this bit will set the display into 32 character mode; outputting a zero will reset the display into the normal 64-character mode.

```
LD    A,8       ;BIT 3 IS SET
OUT   0FFH,A    ;SET 32-CHAR MODE
```

Bit number 2 of the cassette latch is the cassette motor on/off bit. Setting this bit by an OUT 0FFH will turn the cassette motor on, and resetting the bit will turn the cassette motor off. This action is produced by a *small* relay in the TRS-80 cpu, and it would be wise to quench all thoughts about controlling that four-ton air conditioner with this one small control device!

```
LD    A,4       ;BIT 2 IS SET
OUT   0FFH,A    ;SET MOTOR ON
```

Bits number 1 and 0 in the cassette latch are used to write data to the cassette tape. As you probably know from reading your TRS-80 Technical Reference Handbook, data on cassette is arranged serially, and everything is represented by a stream of bits. In the implementation on the TRS-80, cassette data is written by setting bit 0 of the cassette latch, then by setting

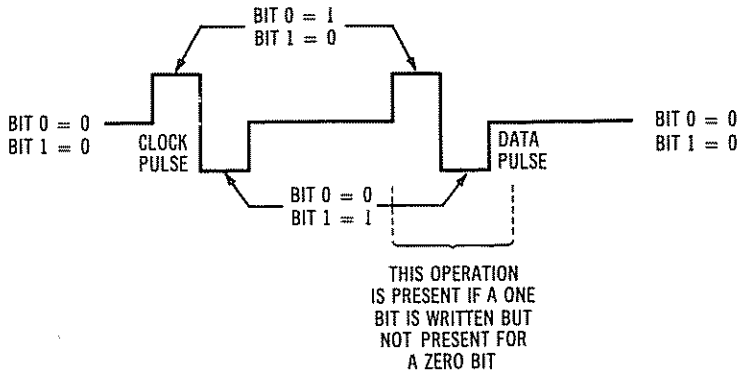


Fig. 10-10. Cassette data waveform.

bit 1 of the cassette latch, and then by resetting both bits. When this is done for both a *clock pulse* and *data pulse* the waveform appears as shown in Figure 10-10.

To illustrate how this works, let us write a program to record some music on cassette. It might be nice to try a little Bach or Beethoven, but perhaps we'll try something a little simpler. First of all, it is necessary to know how to produce *any* tone on the cassette. A simple tone has the appearance of the *sine* wave of Figure 10-11. We can produce a *square wave* on the cassette by turning the cassette output bits on and off rapidly as shown in the figure.

We know how to turn the cassette signal to the recording head on (01) and off (10), but what about the time delay to produce the tone? If we look in the Editor/Assembler Manual we find instruction times under "4MHZ E.T." This is the execution time in microseconds for a Z-80 microprocessor running at a *clock frequency* of 4 megahertz (4 million cycles

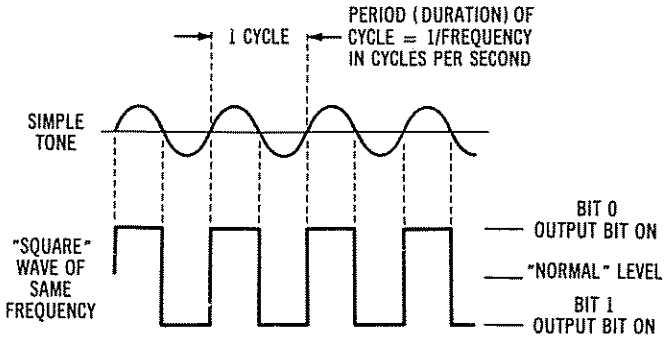


Fig. 10-11. Square wave tones.

per second). The TRS-80 clock frequency is about 1.774 megahertz, so to get the actual *execution times* of TRS-80 instructions we must multiply the 4 MHZ E.T. by 2.26. Let us see how long a simple loop would take. If we have a value of 1 through 255 in the B register, then the simple loop

```
LOOP  DJNZ  LOOP      LOOP HERE FOR 1 TO 255 TIMES
```

would take 3.25 microseconds (4 MHZ E.T.) * 2.255 * count in B, or 7.32 microseconds * count in B. This gives us a range of frequencies from about 535 Hz through 136,612 Hz. (The frequency of the tone can be found by dividing one by the time in microseconds, for example, 500 microseconds would produce a tone of 1/500E-06 or 2000 Hz.)

As the complete cycle would be determined by a timing delay to turn the write head on one direction and off the other, the actual tones that could be produced are 267 Hz through

68,306 Hz. If we stay on the lower end of that range we should be able to get a nice range of notes.

The routine to play a note with a given value in B follows:

```
PLAYN   LD      C,(DURTN)   ;GET DURATION
CONT    LD      B,(FREQ)    ;GET FREQUENCY
        LD      A,1
        OUT     (0FFH),A    ;TURN ON 1/2 CYCLE
LOOP1   DJNZ    LOOP1       ;DELAY FOR FREQUENCY
        LD      B,(FREQ)    ;GET FREQUENCY
        LD      A,2
        OUT     (0FFH),A    ;TURN ON OTHER 1/2 CYCLE
LOOP2   DJNZ    LOOP2       ;DELAY FOR FREQUENCY
        DEC     C           ;DECREMENT DURATION
        JP      NZ,CONT     ;CONTINUE IF NOT DONE
```

The additional count in C is used to adjust the length of time that the note plays. The value of D is related to the value of the frequency count to make all notes a quarter note duration, or approximately so (what did you expect, the New York Philharmonic?). The entire code required to play the TRS-80 concerto is given below. A table of delay values defines the duration and notes, and is terminated by a zero.

```
                00100 ;PROG 1 THE TRS-80 CONCERTO
                00110 .
4A00            00120       ORG     4A00H
4A00 DD21344A   00130 START LD      IX,TABLE    ;START OF MUSIC TABLE
4A04 DD4E00     00140 CONT1 LD      C,(IX)      ;DURATION
4A07 79         00150       LD      A,C         ;MOVE TO A FOR TEST
4A08 B7         00160       OR      A           ;TEST FOR 0
4A09 CA0D4A     00170 LOOP  JP      Z,LOOP      ;LOOP HERE ON DONE
4A0C DD4601     00180 CONT2 LD      B,(IX+1)    ;GET DELAY COUNT
4A0F 3E01       00190       LD      A,1
4A11 D3FF       00200       OUT     (0FFH),A    ;TURN ON 1/2 CYCLE
4A13 10FE       00210 LOOP1 DJNZ    LOOP1       ;DELAY FOR FREQ
4A15 DD4601     00220       LD      B,(IX+1)    ;GET DELAY AGAIN
4A18 3E02       00230       LD      A,2
4A1A D3FF       00240       OUT     (0FFH),A    ;TURN ON OTHER 1/2 CYCLE
4A1C 10FE       00250 LOOP2 DJNZ    LOOP2       ;DELAY FOR FREQ
4A1E 0D         00260 LOOP3 DEC     C           ;DECREMENT DURATION
4A1F C2004A     00270       JP      NZ,CONT2    ;GO IF NOT DONE
```

```
4A22 DD23      00280          INC     IX              ;POINT TO NEXT NOTE
4A24 DD23      00290          INC     IX
4A26 01FFFF    00300          LD      BC,-1           ;INCREMENT VALUE
4A29 213000    00310          LD      HL,30H          ;INITIAL DELAY VALUE
4A2C 09        00320 LOOP4    ADD     HL,BC           ;DELAY FOR INTERVAL
4A2D DA2C4A    00330          JP      C,LOOP4         ;  BETWEEN NOTES
4A30 C3044A    00340          JP      CONT1           ;CONTINUE
4A33 A89A      00350 TABLE    DEFW    9AA8H           ;TABLE OF NOTES
4A35 3FA2      00360          DEFW    0A23FH          ;EACH ENTRY IS MADE UP
4A37 5DAC      00370          DEFW    0AC5DH          ;OF TWO BYTES.  FIRST
4A39 6090      00380          DEFW    9060H           ;BYTE IS DURATION OF
4A3B A090      00390          DEFW    90A0H           ;NOTE.  SECOND BYTE IS
4A3D 4890      00400          DEFW    9048H           ;FREQUENCY VALUE
4A3F 7890      00410          DEFW    9078H
4A41 F890      00420          DEFW    90F8H
4A43 5DA2      00430          DEFW    0A25DH
4A45 5DAD      00440          DEFW    0AD5DH
4A47 6090      00450          DEFW    9060H
4A49 E86B      00460          DEFW    6BE8H
4A4B 485F      00470          DEFW    5F48H
4A4D FF54      00480          DEFW    54FFH
4A4F 00        00490          DEFB    0               ;TERMINATOR
0000           00500          END     4A0004
00000 TOTAL ERRORS
LOOP4   4A2C
LOOP3   4A1E
LOOP2   4A1C
LOOP1   4A13
CONT2   4A0C
LOOP    4A09
CONT1   4A04
TABLE   4A33
START   4A00
```

Arrangement compliments of James Garon

Real-World Interfacing

Is it possible to use the TRS-80 to control real-world events? An emphatic yes! But here's the catch. It does take some *hardware*. In this section, we will discuss how real-world control is done. We will be talking about some simple hardware, but you should find it interesting. (Just think about that TRS-80 controlled robot mowing the grass while you sleep in! But seriously...)

First of all, let us talk about what types of control can be provided to the external world with the TRS-80. Things externally are controlled by on/off conditions in a large number of cases. Such things as garage door openers, burglar alarms triggered by a switch being opened, sprinkler valves being turned on by a time switch—these are all events controlled by an on/off state. This class of functions can be controlled by *discrete* inputs and outputs to the TRS-80. One bit of an output or input can control or detect the operation, as only an on or off state is involved.

A second class of things in the external world are those events that are not controlled in binary fashion. The temperature of a room, windspeed, dampness of the soil, and lighting intensity are but a few items that have a range of values and cannot be represented by a single binary one or zero. These physical quantities require many bits to represent them, but they *can* be represented. There are many available devices that convert external world quantities into voltage, current, or resistance *analogs* that are then converted into binary form by an *analog-to-digital* converter. The resulting digital form, whether it is 8 bits or 24 can then be read into a computer such as a TRS-80 and processed.

Discrete Inputs

Suppose that we want to input a set of eight bits into the A register. These bits represent eight different discrete inputs that are either on or off. A good example would be a set of inputs from burglar alarm switches in eight rooms of a house. The bits are either a one (switch closed) or a zero (switch open), and we would like to read these eight inputs once a second or so to find out whether a switch that is normally closed is open, or a switch that is normally open is closed. How do we go about designing interface circuitry to do this, and what programming steps are required?

Earlier we discussed I/O ports. If we set up our burglar alarm inputs for a particular I/O port, then that port must have the following capability:

1. It must be able to recognize its address when it is sent over the system *address lines.*
2. It must be able to tell when an I/O instruction is being executed.

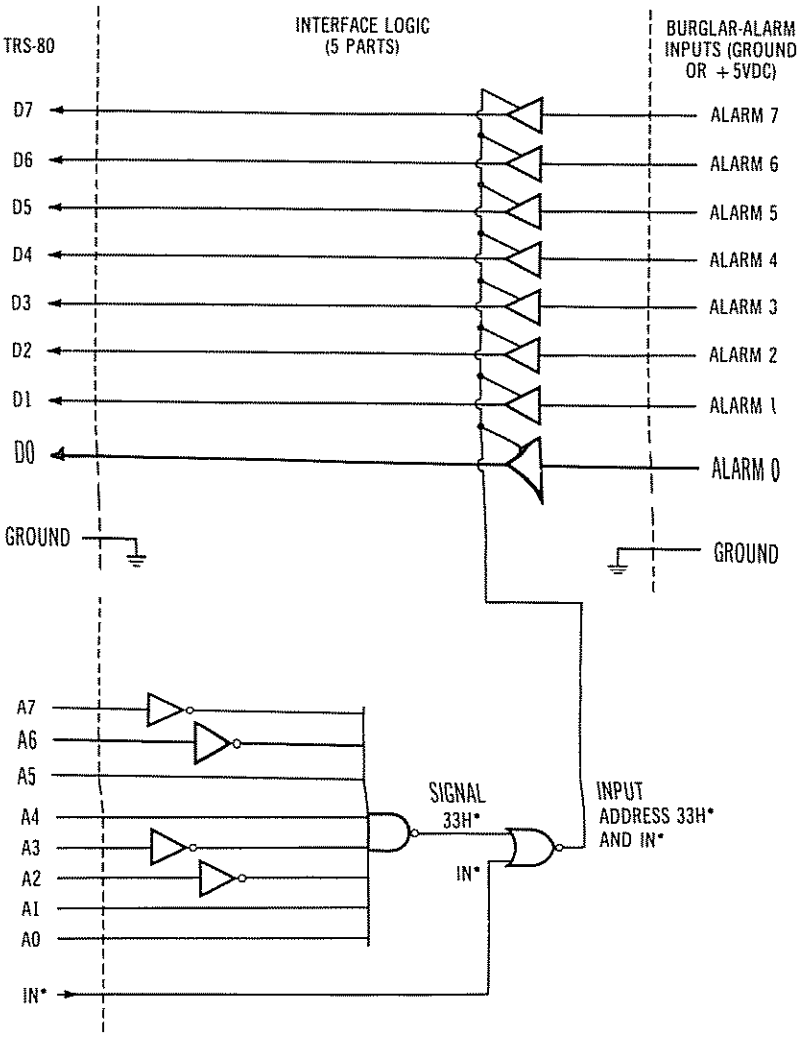


Fig. 10-12. Inputting external data.

3. It must be able to pass the eight bits of data to the cpu over the system *data lines.*

The circuitry for performing these tasks is shown in Figure 10-12. When signal RD* is active (this is the signal on pin 15 of the cpu or interface 40-pin connector), address lines A7 through A0 contain the port address from the IN instruction (address lines A7 through A0 are on various pins of the 40-pin connector). If, for example, we have defined the address of the port as 33H, executing an IN A,(33H) instruction would cause signal RD* to become active and simultaneously put 00110011 on address lines A7 through A0. The circuitry shown in the figure outputs a one for signal INPUT when both RD* and address 33H are present. This will only occur for the IN A,(33H) instruction. Signal INPUT allows the burglar alarm inputs to be *gated* (transmitted) from the eight lines onto the system data bus lines D7 through D0 (on various pins of the connector). During the execution of the IN A,(33H) the cpu will take the contents of the data bus and store it in the A register, completing the execution of the IN instruction. Now the data from the eight inputs can be processed, which might go something like this

```
LOOP  IN    A,(33H)   ;GET INPUTS
      XOR   0B3H      ;TEST 7,5,4,1,0 ON;6,3,2 OFF
      JP    NZ,HELP   ;GO IF BURGLAR
      JP    LOOP      ;TRY AGAIN
```

As the entire input, test, and loop takes under 20 millionths of a second (!) the constraint of one test every second is indeed met. As a matter of fact, there is more than enough time to do all kinds of other processing or control applications and still meet the *poll* of the burglar alarms every second.

This implementation is one of the more simple real-world applications. However, an output of discrete values is not much more complicated. The signal decoded in this case is analogous to the IN* signal, and, strangely enough, is called OUT*. The output operation works as follows: When signal OUT* is active a port address is present on the address bus lines A7 through A0. If the port address matches the built-in address of the hardware, then there is data for the port on data lines D7 through D0. If this is the case, the data lines are written into a memory *latch* similar to that used for the cassette. When the data disappears (it is only present for a few microseconds), the latch will retain the bit configuration and transmit it to the outside world. This circuitry is shown in Figure 10-13.

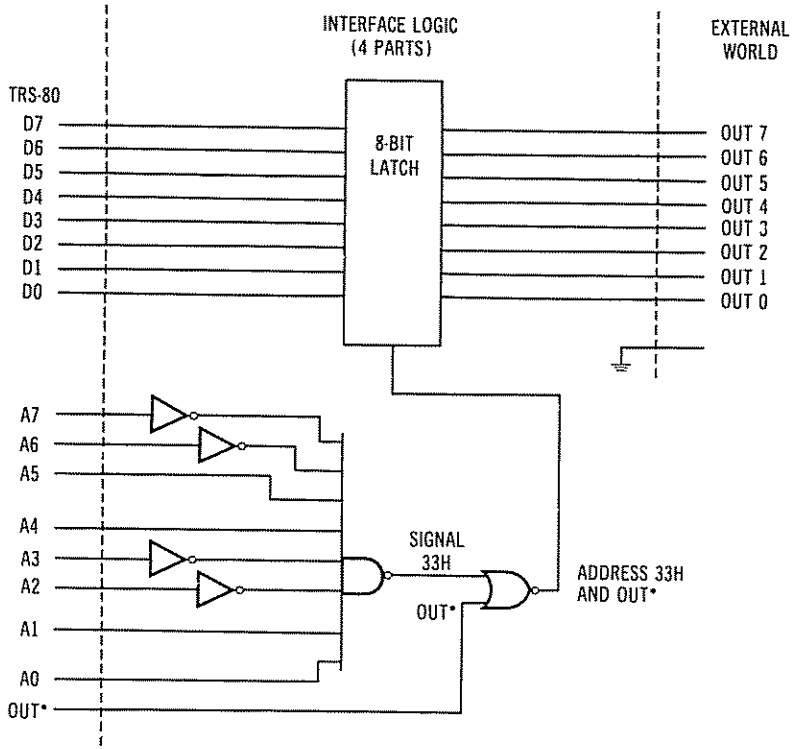


Fig. 10-13. Outputting data to the external world.

Suppose we have a set of lawn sprinkler valves that must be turned on at certain times. Location TIME holds the time in increments of one minute. The time at 12:00 noon is represented by a count of 720 in the two-byte variable. To turn sprinklers 3 and 5 on at noon, the interface in Figure 10-13 is used, along with the code below.

```
LD    HL,TIME     ;GET CURRENT TIME IN MINUTES
LD    BC,720      ;NOON
OR    A           ;RESET CARRY
SBC   HL,BC       ;TEST FOR NOON
JP    NZ,OTHER    ;CONTINUE WITH OTHER PROCESSING
LD    A,28H       ;SPRINKLERS 3 AND 5
OUT   (033H),A    ;TURN THEM ON
```

Another method for implementing discrete input and outputs involves a memory-mapped approach similar to that used for the line printer and other devices. Here IN and OUT instructions are not used, but the external logic is treated as

memory locations and loads and stores are used instead. This is also a valid approach but requires slightly different implementation. The problem of reading in other than discrete inputs or of outputting other than discrete bit patterns is similar to the methods described previously. The major consideration in this case is the conversion between analog values and digital 8-bit values required. The logic required for reading or writing the digital values between the cpu registers and the I/O port, however, is exactly the same as described above.

We hope that these very simple examples will give you some insight into the nature of external-world interfacing. With a little bit of external logic, the TRS-80 can indeed be used to control any number of things around the home or in industry, and with assembly-language programming, this control can be fast indeed.

CHAPTER 11

Common Subroutines

The subroutines presented in this section are the "common" subroutines described elsewhere in the book. Many of them are used continually in larger programs, and they are given here so that the reader may incorporate them into his own programs if he desires. All of them are *subroutines* and must be CALLed from the reader's code. All are assembled at 4A00H, and must be reassembled by incorporating the source code into the reader's source code, or by separate reassembly with a new *ORiGin*. A brief description of each routine is given below, with the assembly following.

FILL Subroutine

The FILL subroutine is used to fill a block of memory with a given 8-bit value. FILL could be used, for example, to zero a buffer, or to fill the video display area with blank characters. On entry, the D register contains the character to be filled, HL points to the start of the fill area, and BC contains the number of bytes to fill, 1 through 65535. An entry with BC=0 is treated as 65536 bytes to fill. On exit HL points to the last byte filled plus one, D is unchanged, and A and BC are zeroed.

```
0100 ;SUBROUTINE TO FILL DATA IN MEMORY
0110 ;   ENTRY:(D)=DATA TO BE FILLED
0120 ;         (HL)=START OF FILL AREA
0130 ;         (BC)=# OF BYTES TO FILL
```

```
                00140 .          CALL   FILL
                00150 .   EXIT: (D)=SAME
                00160 .          (HL)=END OF FILL+1
                00170 .          (BC)=0
                00180 .          (A)=0
                00190 .
4000            00200          ORG    4000H
4000 72         00210 FILL     LD     (HL),D          ;STORE BYTE
4001 23         00220          INC    HL              ;BUMP POINTER
4002 0B         00230          DEC    BC              ;ADJUST COUNT
4003 78         00240          LD     A,B             ;GET MS OF COUNT
4004 B1         00250          OR     C               ;MERGE LS COUNT
4005 20F9       00260          JR     NZ,FILL         ;CONTINUE IF DONE
4007 C9         00270          RET                    ;RETURN IF DONE
0000            00280          END
00000 TOTAL ERRORS
FILL    4000
```

MOVE Subroutine

The MOVE subroutine is used to move one block of memory to another area in memory. The blocks may be *overlapping* without conflict; the program is "smart" enough to calculate the direction of the move based on the type of overlap. On entry HL, DE, and BC are set up as in the block move instructions—HL points to the source block, DE points to the destination block, and BC holds the number of bytes to move. On exit, HL and DE point either to the block areas plus one, or to the block areas minus one, based on the direction of the move. BC contains zero.

```
00100 ;SUBROUTINE TO MOVE MEMORY
00110 .    ENTRY:(HL)=SOURCE START
00120 .          (DE)=DESTINATION START
00130 .          (BC)=# OF BYTES TO MOVE
00140 .     EXIT:(HL)=SOURCE AREA+1
00150 .          (DE)=DEST AREA+1
00160 .          (BC)=0
00170 .
```

```
4000            00100           ORG     4A00H           ;CHANGE ON REASSEMBLY
4A00 E5         00190 MOVE      PUSH    HL              ;SAVE SOURCE PNTR
4A01 B7         00200           OR      A               ;CLEAR CARRY
4A02 ED52       00210           SBC     HL,DE           ;SOURCE-DEST PNTRS
4A04 E1         00220           POP     HL              ;RESTORE PNTR
4A05 DA0C4A     00230           JP      C,MOV10         ;GO IF MOVE BACK
4A08 EDB0       00240           LDIR                    ;MOVE FORWARD
4A0A 1806       00250           JR      MOV20           ;GO TO RETURN
4A0C 09         00260 MOV10     ADD     HL,BC           ;POINT TO END+1
4A0D 2B         00270           DEC     HL              ;POINT TO END
4A0E EB         00280           EX      DE,HL           ;SWAP
4A0F 09         00290           ADD     HL,BC           ;POINT TO END+1
4A10 2B         00300           DEC     HL              ;POINT TO END
4A11 EB         00310           EX      DE,HL           ;SWAP BACK
4A12 EDB8       00320           LDDR                    ;MOVE BACK
4A14 C9         00330 MOV20     RET                     ;RETURN
0000            00340           END
00000 TOTAL ERRORS
MOV20   4A14
MOV10   4A0C
MOVE    4A00
```

MULADD Subroutine

MULADD is a subroutine to perform multiple-precision adds. Two multiple-precision operands from one to 256 bytes in length are added to each other, and the result is put into the destination operand. The source operand remains unchanged. The operands are located anywhere in memory desired, with the data arranged most significant byte through least significant byte from low memory through high memory. On entry the IX register points to the first byte (most significant) of the destination operand and IY points to the first byte (most significant) of the source operand. Both operands are treated as the same length. The B register contains the number of bytes in each operand from 1 through 255. An entry of B=0 is treated as a length of 256 bytes. On exit, the destination operand contains the result of the add. IX and IY are un-

changed. The B register is zeroed, and the A register is destroyed.

```
                00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION ADDS
                00110 .   ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                00120 .         (IY)=POINTS TO MS BYTE OF SOURCE
                00130 .         (B)=# OF BYTES IN OPERANDS
                00140 .         CALL  MULADD
                00150 .         (RETURN)
                00160 .   EXIT:(IX)=UNCHANGED
                00170 .        (IY)=UNCHANGED
                00180 .        (A)=DESTROYED
                00190 .        (B)=0
                00200 .
4A00            00210         ORG   4A00H       ;CHANGE ON REASSEMBLY
4A00 D5         00220 MULADD  PUSH  DE          ;SAVE DE
4A01 58         00230         LD    E,B         ;#BYTES TO E
4A02 1600       00240         LD    D,0         ;DE NOW HAS #
4A04 1B         00250         DEC   DE          ;DE NOW HAS #-1
4A05 DD19       00260         ADD   IX,DE       ;POINT TO LS BYTE
4A07 FD19       00270         ADD   IY,DE       ;POINT TO LS BYTE
4A09 D1         00280         POP   DE          ;RESTORE ORIGINAL
4A0A AF         00290         XOR   A           ;RESET CARRY
4A0B DD7E00     00300 LOOP    LD    A,(IX)      ;GET DESTINATION
4A0E FD8E00     00310         ADC   A,(IY)      ;ADD SOURCE
4A11 DD7700     00320         LD    (IX),A      ;STORE RESULT
4A14 1001       00330         DJNZ  LOOP1       ;GO IF NOT DONE
4A16 C9         00340         RET               ;RETURN
4A17 DD2B       00350 LOOP1   DEC   IX          ;PNT TO NEXT HIGHER
4A19 FD2B       00360         DEC   IY          ;PNT TO NEXT HIGHER
4A1B C30B4A     00370         JP    LOOP        ;CONTINUE
0000            00380         END
00000 TOTAL ERRORS
LOOP1   4A17
LOOP    4A0B
MULSUB  4A00
```

MULSUB Subroutine

The MULSUB subroutine performs multiple-precision subtracts. Two multiple-precision operands from one byte to 256 bytes in length are subtracted from each other, and the result is put into the destination operand memory locations. The source operand remains unchanged. The operands are located anywhere in memory desired, with the data arranged most significant byte through least significant byte from low memory through high memory. On entry the IX register points to the first byte (most significant) of the destination operand and IY points to the first byte (most significant) of the source operand. Both operands are treated as the same length. The B register contains the number of bytes in each operand from 1 through 255. An entry of B=0 is treated as a length of 256 bytes. On exit, the destination operand contains the result of the subtract. IX and IY are unchanged. The B register is zeroed, and the A register is destroyed.

```
                        00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION SUBTRACTS
                        00110 ;     ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                        00120 ;           (IY)=POINTS TO MS BYTE OF SOURCE
                        00130 ;           (B)=# OF BYTES IN OPERANDS
                        00140 ;           CALL MULSUB
                        00150 ;           (RETURN)
                        00160 ;     EXIT:(IX)=UNCHANGED
                        00170 ;          (IY)=UNCHANGED
                        00180 ;          (A)=DESTROYED
                        00190 ;          (B)=0
                        00200 ;
4A00                    00210           ORG     4A00H           ;CHANGE ON REASSEMBLY
4A00 D5                 00220 MULSUB    PUSH    DE              ;SAVE DE
4A01 58                 00230           LD      E,B             ;#BYTES TO E
4A02 1600               00240           LD      D,0             ;DE NOW HAS #
4A04 1B                 00250           DEC     DE              ;DE NOW HAS #-1
4A05 DD19               00260           ADD     IX,DE           ;POINT TO LS BYTE
4A07 FD19               00270           ADD     IY,DE           ;POINT TO LS BYTE
4A09 D1                 00280           POP     DE              ;RESTORE ORIGINAL
4A0A AF                 00290           XOR     A               ;RESET CARRY
4A0B DD7E00             00300 LOOP      LD      A,(IX)          ;GET DESTINATION
```

```
4A0E FD9E00    00310        SBC   A,(IY)      ;SUBTRACT SOURCE
4A11 DD7700    00320        LD    (IX),A      ;STORE RESULT
4A14 1001      00330        DJNZ  LOOP1       ;GO IF NOT DONE
4A16 C9        00340        RET               ;RETURN
4A17 DD2B      00350 LOOP1  DEC   IX          ;PNT TO NEXT HIGHER
4A19 FD2B      00360        DEC   IY          ;PNT TO NEXT HIGHER
4A1B C3084A    00370        JP    LOOP        ;CONTINUE
0000           00380        END
00000 TOTAL ERRORS
LOOP1  4A17
LOOP   4A08
MULSUB 4000
```

CMPARE Subroutine

The CMPARE subroutine compares two 8-bit operands in true algebraic fashion, that is, a −5 is less than a −1, and so forth. Three return points must be provided by the user after the CALL to CMPARE. Each return point must have a jump instruction of three bytes. The first return point is the return made when the A operand is less than the B operand. The second return point is the return made when the two operands are equal. The third return point is the return made when operand A is greater than operand B. By putting in jumps to the same areas, any combination of equalities may be constructed. For example, if the three return points have

```
JP  ONE   ;JUMP TO ONE ON LESS THAN
JP  ONE   ;JUMP TO ONE ON EQUAL
JP  TWO   ;JUMP TO TWO ON GREATER THAN
```

a jump will be made to location "ONE" if A is less than or equal to B, and a jump to location "TWO" will be made if A is greater than B.

On entry, the A register contains the first operand, and the B register contains the second. On exit, the return point is based on the comparison of A to B. A remains unchanged along with B, and the HL register is destroyed.

```
00000 TOTAL ERRORS
GREATR  4A11
LESST   4A13
DIFFER  4A15
```

```
                00100 ;SUBROUTINE TO COMPARE TWO 8-BIT SIGNED OPERANDS
                00110 ;   ENTRY:(A)=OPERAND 1
                00120 ;         (B)=OPERAND 2
                00130 ;         CALL   COMPARE        ;CALL SR
                00140 ;         (RTN FOR A LT B)      ;PUT JP  LESST HERE
                00150 ;         (RRTN FOR A=B)        ;PUT JP  EQUAL HERE
                00160 ;         (RTN FOR A GT B)      ;PUT JP  GREATR HERE
                00170 ;   EXIT: (A)=UNCHANGED
                00180 ;         (B)=UNCHANGED
                00190 ;         (HL)=DESTROYED
                00200 ;
4A00            00210          ORG    4A00H          ;CHANGE ON REASSEMBLY
4A00 E1         00220 COMPARE  POP    HL             ;GET RTN ADDRESS
4A01 D5         00230          PUSH   DE             ;SAVE DE
4A02 110300     00240          LD     DE,3           ;ADDRESS INCREMENT
4A05 B8         00250          CP     B              ;COMPARE A:B
4A06 2808       00260          JR     Z,EQUAL        ;GO IF EQUAL
4A08 F5         00270          PUSH   AF             ;SAVE FLAGS
4A09 A8         00280          XOR    B              ;TEST SIGN BITS
4A0A 17         00290          RLA                   ;XOR TO C
4A0B DA154A     00300          JP     C,DIFFER       ;GO IF DIFFERENT SIGNS
4A0E F1         00310          POP    AF             ;RESTORE FLAGS
4A0F 3802       00320          JR     C,LESST        ;GO IF A LT B
4A11 19         00330 GREATR   ADD    HL,DE          ;BUMP RTN BY 3
4A12 19         00340 EQUAL    ADD    HL,DE          ;BUMP RTN BY 3
4A13 D1         00350 LESST    POP    DE             ;RESTORE DE
4A14 E9         00360          JP     (HL)           ;RTN TO 0,3,6
4A15 F1         00370 DIFFER   POP    AF             ;RESTORE FLAGS
4A16 DA114A     00380          JP     C,GREATR       ;GO IF A GT B
4A19 C3134A     00390          JP     LESST          ;A LT B
0000            00400          END
EQUAL   4A12
COMPARE 4A00
```

MUL16 Subroutine

The MUL16 multiplies an unsigned 16-bit number in the DE register by an unsigned 8-bit number in the B register,

putting the result in the HL register. As the numbers are unsigned, DE may hold from 0 through 65535 and B may hold from 0 through 255. Overflow may result if the product is too large to be held in 16 bits. There is no check on overflow. On entry, DE contains the 16-bit multiplicand and B contains the 8-bit multiplier. On exit, HL contains the 16-bit product, DE is destroyed, and B contains 0.

```
                00100 ;SUBROUTINE TO MULTIPLY 16 BY 8
                00110 ;   ENTRY:(DE)=MULTIPLICAND,UNSIGNED
                00120 ;         (B)=MULTIPLIER,UNSIGNED
                00130 ;         CALL  MUL16
                00140 ;   EXIT: (HL)=PRODUCT
                00150 ;         (DE)=DESTROYED
                00160 ;         (B)=0
                00170 ;
4000            00180          ORG     4000H          ;CHANGE ON REASSEMBLY
4000 210000     00190 MUL16    LD      HL,0           ;CLEAR PARTIAL PRODUCT
4003 CB38       00200 LOOP     SRL     B              ;SHIFT OUT M'IER BIT
4005 3001       00210          JR      NC,CONT        ;GO IF NO CARRY (1 BIT)
4007 19         00220          ADD     HL,DE          ;ADD MULTIPLICAND
4008 C8         00230 CONT     RET     Z              ;GO IF M'IER
4009 EB         00240          EX      DE,HL          ;MULTIPLICAND TO HL
400A 29         00250          ADD     HL,HL          ;SHIFT MULTIPLICAND
400B EB         00260          EX      DE,HL          ;SWAP BACK
400C C30340     00270          JP      LOOP           ;CONTINUE
0000            00280          END
00000 TOTAL ERRORS
CONT    4008
LOOP    4003
MUL16   4000
```

DIV16 Subroutine

The DIV16 subroutine divides an unsigned 16-bit number in the HL register pair by an unsigned 8-bit number in the D register. The quotient is placed in the IX register and the remainder is left in H. As the numbers are unsigned, the dividend in HL may be 0 through 65535 and the divisor in D may

be 0 through 255. Overflow will result on division by zero. The remainder is less than the divisor. On entry, HL contains the 16-bit dividend and D contains the 8-bit divisor. On exit, IX contains the 16-bit quotient and H contains the 8-bit remainder. The L and A registers are destroyed, E is zeroed, and the D register is unchanged.

```
                00100 ;SUBROUTINE TO DIVIDE 16 BY 8
                00110 .   ENTRY:(HL)=DIVIDEND 16 BITS
                00120 .         (D)=DIVISOR 8 BITS
                00125 .         CALL  DIV16
                00130 .   EXIT: (IX)=QUOTIENT 16 BITS
                00140 .         (H)=REMAINDER 8 BITS
                00150 .         (L)=DESTROYED
                00160 .         (D)=UNCHANGED
                00170 .         (E)=0
                00180 .         (A)=DESTROYED
                00190 .
4A00            00200         ORG   4A00H       ;CHANGE ON REASSEMBLY
4A00 7D         00210 DIV16   LD    A,L         ;LS BYTE DIVDND
4A01 6C         00220         LD    L,H         ;MS BYTE DIVDND
4A02 2600       00230         LD    H,0         ;CLEAR FOR SUBT
4A04 1E00       00240         LD    E,0         ;SETUP FOR SUBTRACT
4A06 0610       00250         LD    B,16        ;16 ITERATIONS
4A08 DD210000   00260         LD    IX,0        ;INITIALIZE QUOTIENT
4A0C 29         00270 LOOP    ADD   HL,HL       ;SHIFT DIVD LEFT
4A0D 17         00280         RLA               ;SHIFT 8 LS BITS
4A0E D2124A     00290         JP    NC,LOOP1    ;GO IF 0 BIT
4A11 2C         00300         INC   L           ;SHIFT TO HL
4A12 DD29       00310 LOOP1   ADD   IX,IX       ;SHIFT QUOTIENT LEFT
4A14 DD23       00320         INC   IX          ;0 BIT=1
4A16 B7         00330         OR    A           ;CLEAR CARRY FOR SUB
4A17 ED52       00340         SBC   HL,DE       ;TRY SUBTRACT
4A19 D21F4A     00350         JP    NC,CONT     ;GO IF IT WENT
4A1C 19         00360         ADD   HL,DE       ;RESTORE
4A1D DD2B       00370         DEC   IX .        ;SET Q BIT=0
4A1F 10EB       00380 CONT    DJNZ  LOOP        ;GO IF NOT 16
```

```
4A21 C9        00390        RET                    RETURN
0000           00400        END
00000 TOTAL ERRORS
CONT    4A1F
LOOP1   4A12
LOOP    4A0C
DIV16   4A00
```

HEXCV Subroutine

HEXCV is a subroutine to convert an 8-bit value (two hexadecimal digits) into two ASCII characters representing the hexadecimal characters 0 through 9 and A through F. On entry, the A register contains the 8-bit value to be converted. On exit, the H register contains an ASCII character representing the hex character for the four high-order bits and the L register contains an ASCII character representing the hex character for the four low-order bits. The A and C registers are destroyed.

```
                00100 ;SUBROUTINE TO CONVERT FROM HEX TO ASCII
                00110 ;
                00120 ;   ENTRY:(A)=8-BIT VALUE TO BE CONVERTED
                00130 ;         CALL   HEXCV
                00140 ;         (RETURN)
                00150 ;   EXIT: (HL)=TWO ASCII VALUES, HIGH AND LOW
                00160 ;         (A)=DESTROYED
                00170 ;         (C)=DESTROYED
                00180 ;
4A00            00190        ORG    4A00H          ;CHANGE ON REASSEMBLY
4A00 4F         00200 HEXCV  LD     C,A            ;SAVE TWO HEX DIGITS
4A01 CB3F       00210        SRL    A              ;ALIGN HIGH DIGIT
4A03 CB3F       00220        SRL    A
4A05 CB3F       00230        SRL    A
4A07 CB3F       00240        SRL    A
4A09 CD154A     00250        CALL   TEST           ;CONVERT TO ASCII
4A0C 67         00260        LD     H,A            ;SAVE FOR RTN
4A0D 79         00270        LD     A,C            ;RESTORE ORIGINAL
4A0E E60F       00280        AND    0FH            ;GET LOW DIGIT
```

```
4A10 CD154A   00290        CALL   TEST        ;CONVERT TO ASCII
4A13 6F       00300        LD     L,A         ;SAVE FOR RTN
4A14 C9       00310        RET
4A15 C630     00320 TEST   ADD    A,30H       ;CONVERSION FACTOR
4A17 FE3A     00330        CP     3AH         ;TEST FOR 0-9
4A19 FA1E4A   00340        JP     M,TEST1     ;GO IF 0-9
4A1C C607     00350        ADD    A,7         ;CORRECT FOR A-F
4A1E C9       00360 TEST1  RET                ;RETURN
0000          00370        END
00000 TOTAL ERRORS
TEST1  4A1E
TEST   4A15
HEXCV  4A00
```

SEARCH Subroutine

The SEARCH subroutine searches a table for a given *key* value of 8 bits. The table may be any number of entries from 1 through 256, with each entry a *fixed-length* of one to n bytes. The table may be located anywhere in memory. The key value is assumed to be the first byte in each entry.

On entry, the A register holds the 8-bit key of 0 through 255. HL points to the start of the table in memory. DE contains the length of each entry in bytes. The C register holds the number of entries in the table, from 1 through 255. On exit, the Z flag is set if the key has been found in the table, and the HL register points to the entry containing the matching value in this case. If the key is not found, the Z flag is not set upon return. If the key is found, BC contains the current number of entries *left* in the table. In this case the subroutine may be called again to search for another occurrence of the key, without changing the contents of HL, DE, BC, or A.

SET, RESET, and TEST Subroutines

These subroutines are used to set, reset, and test a point on the screen in similar fashion to SET, RESET, and POINT in BASIC. The screen coordinate values given are converted into corresponding memory locations in video memory, which are

```
                00100 ;SUBROUTINE FOR TABLE SEARCH
                00110 ;   ENTRY:(A)=KEY
                00120 ;         (HL)=TABLE START
                00130 ;         (DE)=LENGTH OF EACH ENTRY IN BYTES
                00140 ;         (C)=# OF ENTRIES IN TABLE
                00150 ;         CALL  SEARCH
                00160 ;   EXIT: Z FLAG SET IF FOUND, NOT SET IF NOT FOUND
                00170 ;         (HL)=LOCATION OF MATCH IF FOUND
                00180 ;         (BC)=CURRENT # LEFT
                00190 ;         (DE)=UNCHANGED
                00200 ;
4A00            00210         ORG   4A00H       ;CHANGE ON REASSEMBLY
4A00 0600       00220 SEARCH  LD    B,0         ;BC NOW HAS #
4A02 EDA1       00230 LOOP    CPI               ;COMPARE A WITH (HL)
4A04 CA0E4A     00240         JP    Z,FOUND     ;GO IF FOUND
4A07 E20F4A     00250         JP    PO,NFND     ;AT END AND NOT FND
4A0A 19         00260         ADD   HL,DE       ;CURRENT+LENGTH+1
4A0B 2B         00270         DEC   HL          ;CURRENT +LENGTH
4A0C 18F4       00280         JR    LOOP        ;TRY AGAIN
4A0E 2B         00290 FOUND   DEC   HL          ;ADJUST TO FOUND LOC
4A0F C9         00300 NFND    RET               ;RETURN
0000            00310         END
00000 TOTAL ERRORS
NFND    4A0F
FOUND   4A0E
LOOP    4A02
SEARCH  4A00
```

then processed. The high-order bit of each memory location is set when any of the three subroutines is called, on the assumption that the coordinates addressed represent graphics points.

On entry, DE contains the y,x coordinates. The D register contains the Y value of 0 through 47, while the E register holds the X value of 0 through 127. A CALL is made to SET, RESET, or TEST to set, reset, or test the coordinate specified.

On exit, the A, B, C, D, E, H, and L registers are destroyed. If a test was involved, the Z flag is set if the point was a zero and reset if the point was a one.

Care must be taken in using this subroutine to make certain that the x and y values are in the range given, as the subroutine may wreak havoc if invalid values are input.

```
00100 ;SUBROUTINE TO CONVERT SCREEN COORDINATES
00110 .   ENTRY: (DE)=Y,X COORDINATES OF POINT
00120 .              X=0 TO 127; Y=0 TO 47
00130 .         CALL  SET      ;SETS POINT
00140 .         CALL  RESET    ;RESETS POINT
00150 .         CALL  TEST     ;TESTS POINT RETURNS Z FLAG
00160 .   EXIT: (A THROUGH L)=DESTROYED
00170 .         Z FLAG SET IF TEST
00180 .
4A00          00190         ORG   4A00H
4A00 3ED6     00200 SET     LD    A,0D6H     ;SET B,(HL) INSTRUCTION
4A02 1806     00210         JR    TEST10     ;GO TO STORE
4A04 3E86     00220 RESET   LD    A,86H      ;RES B,(HL) INSTRUCTION
4A06 1802     00230         JR    TEST10     ;GO TO STORE
4A08 3E46     00240 TEST    LD    A,46H      ;BIT B,(HL) INSTRUCTION
4A0A 3204A    00250 TEST10  LD    (INST+1),A ;STORE 2ND BYTE
4A0D 7A       00260 ADDRES  LD    A,D        ;GET Y
4A0E 06FF     00270         LD    B,0FFH     ;-1
4A10 04       00280 LOOP    INC   B          ;SUCCESIVE SUBS FOR DIV
4A11 D603     00290         SUB   3          ;  BY THREE
4A13 F2104A   00300         JP    P,LOOP     ;GO IF NOT MINUS
4A16 C603     00310         ADD   A,3        ;Y0 IN B, YR IN A
4A18 CB27     00320         SLA   A          ;YR*2
4A1A 4F       00330         LD    C,A        ;SAVE YR*2
4A1B 68       00340         LD    L,B        ;Y0 TO L
4A1C 2600     00350         LD    H,0        ;Y0 IN HL
4A1E 0606     00360         LD    B,6        ;CNT FOR MULTIPLY BY 64
4A20 29       00370 LOOP1   ADD   HL,HL      ;Y0*2
```

```
4021 16FD       00380          DJNZ   LOOP1          ;GO IF NOT Y0=64
4023 1600       00390          LD     D,0             ;DE NOW HAS X
4025 CB3B       00400          SRL    E               ;X0
4027 3001       00410          JR     NC,CONT         ;GO IF XR NE 1
4029 0C         00420          INC    C               ;C NOW HAS YR*2+XR
402A 19         00430 CONT     ADD    HL,DE           ;HL NOW HAS Y0*2+X0
402B 11003C     00440          LD     DE,3C00H        ;START OF DISPLAY
402E 19         00450          ADD    HL,DE           ;HL NOW HAS DISPLACE
402F CB21       00460          SLA    C               ;ALIGN TO FIELD
4031 CB21       00470          SLA    C
4033 CB21       00480          SLA    C
4035 3A3D4A     00490          LD     A,(INST+1)      ;GET INSTRUCTION
4038 81         00500          ADD    A,C             ;SET FIELD
4039 323D4A     00510          LD     (INST+1),A      ;STORE
403C CB         00520 INST     DEFB   0CBH            ;PERFORM BIT,SET,RES
403D 00         00530          DEFB   0               ;WILL BE FILLED IN
403E CBFE       00540          SET    7,(HL)          ;FOR GRAPHICS
4040 C9         00550          RET
0000            00560          END
00000 TOTAL ERRORS
CONT    402A
LOOP1   4020
LOOP    4010
MOORES  4000
INST    403C
TEST    4008
RESET   4004
TEST10  4009
SET     4000
```

SECTION III

Appendices

APPENDIX I

Z-80 Instruction Set

A Register Operations

Complement CPL
Decimal DAA
Negate NEG

Adding/Subtracting Two 8-Bit Numbers

A and Another Register
ADC A,r SBC A,r
ADD A,r SUB A,r
A and Immediate Operand
ADC A,n SBC A,n
ADD A,n SUB A,n
A and Memory Operand

ADC A,(HL)	ADD A,(HL)	SBC (HL)	SUB (HL)
ADC A,(IX+d)	ADD A,(IX+d)	SBC (IX+d)	SUB (IX+d)
ADC A,(IY+d)	ADD A,(IY+d)	SBC (IY+d)	SUB (IY+d)

Adding/Subtracting Two 16-Bit Numbers

HL and Another Register Pair
ADC HL,ss ADD HL,ss SBC HL,ss
IX and Another Register Pair
ADD IX,pp ADD IY,rr

Bit Instructions

Test Bit
Register BIT b,r
Memory BIT b,(HL) BIT b,(IX+d) BIT b,(IY+)
Reset Bit
Register RES b,r
Memory RES b,(HL) RES b,(IX+d) RES b,(IY+d)
Set Bit
Register SET b,r
Memory SET b,(HL) SET b,(IX+d) SET b,(IY+d)

Carry Flag

Complement CCF
Set SCF

Compare Two 8-Bit Operands

A and Another Register CP r
A and Immediate Operand CP n
A and Memory Operand
CP (HL) CP (IX+d) CP (IY+d)
Block Compare
CPD,CPDR,CPI,CPIR

Decrements and Increments

Single Register
DEC r INC r DEC IX DEC IY INC
Register Pair
DEC ss INC ss DEC IX DEC IY INC IX DEC IY
Memory
DEC HL DEC (IX+d) DEC (IY+d)

Exchanges

DE and HL EX DE,HL
Top of Stack
EX (SP),HL EX (SP),IX EX (SP),IY

Input/Output

I/O To/From A and Port
IN A,(n) OUT (n),A
I/O To/From Register and Port
IN r,(C) OUT (C),r
Block
IND,INDR,INR,INIR,OTDR,OTIR,OUTD,OUTI

Interrupts

Disable DI
Enable EI
Interrupt Mode
IM 0 IM 1 IM 2
Return From Interrupt
RETI RETN

Jumps

Unconditional
JP (HL) JP (IX) JP (IY) JP (nn) JR e
Conditional
JP cc,nn JR C,e JR NZ,e JR Z,e
Special Conditional
DJNZ e

Loads

A Load Memory Operand
LD A,(BC) LD A,(DE) LD A,(nn)

A and Other Registers
LD A,I LD A,R LD I,A LD R,A
Between Registers, 8-Bit
LD r,r'
Immediate 8-Bit
LD r,n
Immediate 16-Bit
LD dd,nn LD IX,nn LD IY,nn
Register Pairs From Other Register Pairs
LD SP,HL LD SP,IX LD SP,IY
From Memory, 8-Bits
LD r,(HL) LD r,(IX+d) LD r,(IY+d)
From Memory, 16-Bits
LD HL,(nn) LD IX,(nn) LD IY,(nn) LD dd,(nn)
Block
LDD,LDDR,LDI,LDIR

Logical Operations 8 Bits With A

A and Another Register
AND r OR r XOR r
A and Immediate Operand
AND n OR n XOR n
A and Memory Operand

AND (HL)	OR (HL)	XOR (HL)
AND (IX+d)	OR (IX+d)	XOR (IX+d)
AND (IY+d)	OR (IY+d)	XOR (IY+d)

Miscellaneous

Halt HALT
No Operation NOP

Prime/Non-Prime

Switch AF
EX AF,AF'
Switch Others
EXX

Shifts

Circular (Rotate)
A Only RLA, RLCA, RRA, RRCA
All Registers RL r RLC r RR r RRC r
Memory

RL (HL)	RLC (HL)	RR (HL)	RRC (HL)
RL (IX+d)	RLC (IX+d)	RR (IX+d)	RRC (IX+d)
RL (IY+d)	RLC (IY+d)	RR (IY+d)	RRC (IY+d)

Logical
Registers SRL r
Memory SRL (HL) SRL (IX+d) SRL (IY+d)
Arithmetic
Registers SLA r SRA r
Memory

SLA (HL)	SRA (HL)
SLA (IX+d)	SRA (IX+d)
SLA (IY+d)	SRA (IY+d)

Stack Operations

PUSH IX PUSH IY PUSH qq POP IX POP IY POP qq

Stores

Of A Only
LD (BC),A LD (DE),A LD (HL),A LD (nn),A
All Registers
LD (HL),r LD (IX+d),r LD (IY+d),r
Immediate Data
LD (HL),n LD (IX+d),n LD (IY+d),n
16-Bit Registers
LD ((nn),dd LD (nn),IX LD (nn),IY

Subroutine Action

Conditional CALLs CALL cc,nn
Unconditional CALLs CALL nn
Conditional Return RET cc
Unconditional Return RET cc
Special CALL RST p

APPENDIX II

Z-80 Operation Code Listings

Mnemonic	Format			Description	S	Z	P/V	C
ADC HL,ss	11101101	01ss1010		HL+ss+CY to HL	●	●	●	●
ADC A,r	10001 r			A+r+CY to A	●	●	●	●
ADC A,n	11001110	n		A+n+CY to A	●	●	●	●
ADC A,(HL)	10001110			A+(HL)+CY to A	●	●	●	●
ADC A,(IX+d)	11011101	10001110	d	A+(IX+d)+CY to A	●	●	●	●
ADC A,(IY+d)	11111101	10001110	d	A+(IY+d)+CY to A	●	●	●	●
ADD A,n	11000110	n		A+n to A	●	●	●	●
ADD A,r	10000 r			A+r to A	●	●	●	●
ADD A,(HL)	10000110			A+(HL) to A	●	●	●	●
ADD A,(IX+d)	11011101	10000110	d	A+(IX+d) to A	●	●	●	●
ADD A,(IY+d)	11111101	10000110	d	A+(IY+d) to A	●	●	●	●
ADD HL,ss	00ss1001			HL+ss to HL				●
ADD IX,pp	11011101	00pp1001		IX+pp to IX				●
ADD IY,rr	11111101	00rr1001		IY+rr to IY				●
AND r	10100 r			A AND r to A	●	●	●	0
AND n	11100110	n		A AND n to A	●	●	●	0
AND (HL)	10100110			A AND (HL) to A	●	●	●	0
AND (IX+d)	11011101	10100110	d	A AND (IX+d) to A	●	●	●	0
AND (IY+d)	11111101	10100110	d	A AND (IY+d) to A	●	●	●	0

BIT b,r	11001011	01 b r			Test bit b of r	●	●	●	
BIT b,(HL)	11001011	01 b 110			Test bit b of (HL)	●	●	●	
BIT b,(IX+d)	11011101	11001011	d	01 b 110	Test bit b of (IX+d)	●	●	●	
BIT b,(IY+d)	11111101	11001011	d	01 b 110	Test bit b of (IY+d)	●	●	●	
CALL cc,nn	11 c 100	n	n		CALL subroutine at nn if cc				
CALL nn	11001101	n	n		Unconditionally CALL nn				
CCF	00111111				Complement carry flag				●
CP r	10111 r				Compare A:r	●	●	●	●
CP n	11111110	n			Compare A:n	●	●	●	●
CP (HL)	10111110				Compare A:(HL)	●	●	●	●
CP (IX+d)	11011101	10111110	d		Compare A:(IX+d)	●	●	●	●
CP (IY+d)	11111101	10111110	d		Compare A:(IY+d)	●	●	●	●
CPD	11101101	10101001			Block Compare, no repeat	●	●	●	
CPDR	11101101	10111001			Block Compare, repeat	●	●	●	
CPI	11101101	10100001			Block Compare, no repeat	●	●	●	
CPIR	11101101	10110001			Block Compare, repeat	●	●	●	
CPL	00101111				Complement A (1's comple)				
DAA	00100111				Decimal Adjust A	●	●	●	
DEC r	00 r 101				Decrement r by one	●	●	●	
DEC (HL)	00110101				Decrement (HL) by one	●	●	●	

Mnemonic	Format			Description	S	Z	P/V	C
DEC (IX+d)	11011101	00110101	d	Decrement (IX+d) by one	●	●	●	
DEC (IY+d)	11111101	00110101	d	Decrement (IY+d) by one	●	●	●	
DEC IX	11011101	00101011		Decrement IX by one				
DEC IY	11111101	00101011		Decrement IY by one				
DEC ss	00ss1011			Decrement register pair				
DI	11110011			Disable interrupts				
DJNZ e	00010000	e-2		Decrement B and JR if B≠0				
EI	11111011			Enable interrupts				
EX (SP),HL	11100011			Exchange (SP) and HL				
EX (SP),IX	11011101	11100011		Exchange (SP) and IX				
EX (SP),IY	11111101	11100011		Exchange (SP) and IY				
EX AF,AF'	00001000			Set prime AF active				
EX DE,HL	11101011			Exchange DE and HL				
EXX	11011001			Set prime B-L active				
HALT	01110110			Halt				
IM 0	11101101	01000110		Set interrupt mode 0				
IM 1	11101101	01010110		Set interrupt mode 1				
IM 2	11101101	01011110		Set interrupt mode 2				
IN A,(n)	11011011	n		Load A with input from n				

IN r,(C)	11101101	01 r 000		Load r with input from (C)	●	●	●
INC r	00 r 100			Increment r by one	●	●	●
INC (HL)	00110100			Increment (HL) by one	●	●	●
INC (IX+d)	11011101	00110100	d	Increment (IX+d) by one	●	●	●
INC (IY+d)	11111101	00110100	d	Increment (IY+d) by one	●	●	●
INC IX	11011101	00100011		Increment IX by one			
INC IY	11111101	00100011		Increment IY by one			
INC ss	00ss0011			Increment register pair			
IND	11101101	10101010		Block I/O input from (C)	●	●	●
INDR	11101101	10111010		Block I/0 input, repeat	●	●	●
INI	11101101	10100010		Block I/O input from (C)	●	●	●
INIR	11101101	10110010		Block I/0 input, repeat	●	●	●
JP (HL)	11101001			Unconditional jump to (HL)			
JP (IX)	11011101	11101001		Unconditional jump to (IX)			
JP (IY)	11111101	11101001		Unconditional jump to (IY)			
JP cc,nn	11 c 010	n	n	Jump to nn if cc			
JP nn	11000011	n	n	Unconditional jump to nn			
JR C,e	00111000	e-2		Jump relative if carry			
JR e	00011000	e-2		Unconditional jump relative			
JR NC,e	00110000	e-2		Jump relative if no carry			

Mnemonic	Format	Description	S	Z	P/V	C
JR NZ,e	00100000 | e-2	Jump relative if non-zero				
JR Z,e	00101000 | e-2	Jump relative if zero				
LD A,(BC)	00001010	Load A with (BC)				
LD A,(DE)	00011010	Load A with (DE)				
LD A,I	11101101 | 01010111	Load A with I	●	●	●	
LD A,(nn)	00111010 | n | n	Load A with location nn				
LD A,R	11101101 | 01011111	Load A with R	●	●	●	
LD (BC),A	00000010	Store A to (BC)				
LD (DE),A	00010010	Store A to (DE)				
LD (HL),n	00110110 | n	Store n to (HL)				
LD dd,nn	00dd0001 | n | n	Load register pair with nn				
LD dd,(nn)	11101101 | 01dd1011 | n | n	Load register pair with location nn				
LD HL,(nn)	00101010 | n | n	Load HL with location nn				
LD (HL),r	01110 r	Store r to (HL)				
LD I,A	11101011 | 01000111	Load I with A				
LD IX,(nn)	11011101 | 00101010 | n | n	Load IX with nn				
LD IX,nn	11011101 | 00100001 | n | n	Load IX with location nn				
LD (IX+d),n	11011101 | 00110110 | d | n	Store n to (IX+d)				
LD (IX+d),r	11011101 | 01110 r | d	Store r to (IX+d)				

LD IY,nn	11111101	00100001	n	n	Load IY with nn	
LD IY,(nn)	11111101	00101010	n	n	Load IY with location nn	
LD (IY+d),n	11111101	00110110	d	n	Store n to (IY+d)	
LD (IY+d),r	11111101	01110 r	d		Store r to (IY+d)	
LD (nn),A	00110010	n	n		Store A to location nn	
LD (nn),dd	11101101	01dd0011	n	n	Store register pair to loc'n nn	
LD (nn),HL	00100010	n	n		Store HL to location nn	
LD (nn),IX	11011101	00100010	n	n	Store IX to location nn	
LD (nn),IY	11111101	00100010	n	n	Store IY to location nn	
LD R,A	11101101	01001111			Load R with A	
LD r,r'	01 r r'				Load r with r'	
LD r,n	00 r 110	n			Load r with n	
LD r,(HL)	01 r 110				Load r with (HL)	
LD r,(IX+d)	11011101	01 r 110	d		Load r with (IX+d)	
LD r,(IY+d)	11111101	01 r 110	d		Load rf with (IY+d)	
LD SP,HL	11111001				Load SP with HL	
LD SP,IX	11011101	11111001			Load SP with IX	
LD SP,IY	11111101	11111001			Load SP with IY	
LDD	11101101	10101000			Block load, f'ward, no repeat	●
LDDR	11101101	10111000			Block load, f'ward, repeat	0

Mnemonic	Format			Description	S	Z	P/V	C
LDI	11101101	10100000		Block load, b'ward, no repeat			●	
LDIR	11101101	10110000		Block load b'ward, repeat			0	
NEG	11101101	01000100		Negate A (two's complement)	●	●	●	●
NOP	00000000			No operation				
OR r	10110 r			A OR r to A	●	●	●	0
OR n	11110110	n		A OR n to A	●	●	●	0
OR (HL)	10110110			A OR (HL) to A	●	●	●	0
OR (IX+d)	11011101	10110110	d	A OR (IX+d) to A	●	●	●	0
OR (IY+d)	11111101	10110110	d	A OR (IY+d) to A	●	●	●	0
OTDR	11101101	10111011		Block output, b'ward, repeat	●	●	●	
OTIR	11101101	10110011		Block output, f'ward, repeat	●	●	●	
OUT (C),r	11101101	01 r 001		Output r to (C)				
OUT (n),A	11010011	n		Output A to port n				
OUTD	11101101	10101011		Block output, b'ward, no rpt	●	●	●	
OUTI	11101101	10100011		Block output, f'ward, no rpt	●	●	●	
POP IX	11011101	11100001		Pop IX from stack				
POP IY	11111101	11100001		Pop IY from stack				
POP qq	11qq0001			Pop qq from stack				
PUSH IX	11011101	11100101		Push IX onto stack				

PUSH IY	11111101	11100101			Push IY onto stack				
PUSH qq	11qq0101				Push qq onto stack				
RES b,r	11001011	10 b r			Reset bit b of r				
RES b,(HL)	11001011	10 b 110			Reset bit b of (HL)				
RES b,(IX+d)	11011101	11001011	d	10 b 110	Reset bit b of (IX+d)				
RES b,(IY+d)	11111101	11001011	d	10 b 110	Reset bit b of (IY+d)				
RET	11001001				Return from subroutine				
RET cc	11 c 000				Return from subroutine if cc				
RETI	11101101	01001101			Return from interrupt				
RETN	11101101	01000101			Return from non-maskable int				
RL r	11001011	00010 r			Rotate left thru carry r	●	●	●	●
RL (HL)	11001011	00010110			Rotate left thru carry (HL)	●	●	●	●
RL (IX+d)	11011101	11001011	d	00010110	Rotate left thru carry (IY+d)	●	●	●	●
RL (IY+d)	11010101	11001011	d	00000110	Rotate left thru carry (IY+d)	●	●	●	●
RLA	00010111				Rotate A left thru carry				●
RLC r	11001011	00000 r			Rotate left circular r	●	●	●	●
RLC (HL)	11001011	00000110			Rotate left circular (HL)	●	●	●	●
RLC (IX+d)	11011101	11001011	d	00000110	Rotate left circular (IX+d)	●	●	●	●
RLC (IY+d)	11111101	11001011	d	00000110	Rotate left circular (IY+d)	●	●	●	●
RLCA	00000111				Rotate left circular A				●

Mnemonic	Format				Description	S	Z	P/V	C
RLD	11101101	01101111			Rotate bcd digit left (HL)	●	●	●	
RR r	11001011	00011 r			Rotate right thru carry r	●	●	●	●
RR (HL)	11001011	00011110			Rotate right thru carry (HL)	●	●	●	●
RR (IX+d)	11011101	11001011	d	00011110	Rotate right thru cy (IX+d)	●	●	●	●
RR (IY+d)	00011110	11001011	d	00011110	Rotate left thru cy (IY+d)	●	●	●	●
RRA	00011111				Rotate A right thru carry				●
RRC r	11001011	00001 r			Rotate r right circular	●	●	●	●
RRC (HL)	11001011	00001110			Rotate (HL) right circular	●	●	●	●
RRC (IX+d)	11011101	11001011	d	00001110	Rotate (IX+d) right circular	●	●	●	●
RRC (IY+d)	11111101	11001011	d	00001110	Rotate (IY+d) right circular	●	●	●	●
RRCA	00001111				Rotate A right circular				●
RRD	11101101	01100111			Rotate bcd digit right (HL)	●	●	●	
RST p	11 t 110				Restart to location p				
SBC A,r	10011 r				A-r-CY to A	●	●	●	●
SBC A,n	11011110	n			A-n-CY to A	●	●	●	●
SBC A,(HL)	10011110				A-(HL)-CY to A	●	●	●	●
SBC A,(IX+d)	11011101	10011110	d		A-(IX+d)-CY to A	●	●	●	●
SBC A,(IY+d)	11111101	10011110	d		A-(IY+d)-CY to A	●	●	●	●
SBC HL,ss	11101101	01ss0010			HL-ss-CY to HL	●	●	●	●

SCF	00110111								Set carry flag						1
SET b,(HL)	11001011	11 b 110							Set bit b of (HL)						
SET b,(IX+d)	11011101	11001011	d	11 b 110					Set bit b of (IX+d)						
SET b,(IY+d)	11111101	11001011	d	11 b 110					Set bit b of (IY+d)						
SET b,r	11001011	11 b r							Set bit b of r						
SLA r	11001011	00100 r							Shift r left arithmetic	●	●	●	●		
SLA (HL)	11001011	00100110							Shift (HL) left arithmetic	●	●	●	●		
SLA (IX+d)	11011101	11001011	d	00100110					Shift (IX+d) left arithmetic	●	●	●	●		
SLA (IY+d)	11111101	11001011	d	00100110					Shift (IY+d) left arithmetic	●	●	●	●		
SRA r	11001011	00101 r							Shift r right arithmetic	●	●	●	●		
SRA (HL)	11001011	00101110							Shift (HL) right arithmetic	●	●	●	●		
SRA IX+d)	11011101	11001011	d	00101110					Shift (IX+d) right arithmetic	●	●	●	●		
SRA (IY+d)	11111101	11001011	d	00101110					Shift (IY+d) right arithmetic	●	●	●	●		
SRL r	11001011	00111 r							Shift r right logical	●	●	●	●		
SRL (HL)	11001011	00111110							Shift (HL) right arithmetic	●	●	●	●		
SRL (IX+d)	11011101	11001011	d	00111110					Shift (IX+d) right arithmetic	●	●	●	●		
SRL (IY+d)	11111101	11001011	d	00111110					Shift (IY+d) right arithmetic	●	●	●	●		
SUB r	10010 r								A-r to A	●	●	●	●		
SUB n	11010110	n							A-n to A	●	●	●	●		
SUB (HL)	10010110								A-(HL) to A	●	●	●	●		

Mnemonic	Format			Description	S	Z	P/V	C
SUB (IX+d)	11011101	10010110	d	A-(IX+d) to A	●	●	●	●
SUB (IY+d)	11111101	10010110	d	A-(IY+d) to A	●	●	●	●
XOR r	10101 r			A EXCLUSIVE OR r to A	●	●	●	0
XOR n	11101110	n		A EXCLUSIVE OR n to A	●	●	●	0
XOR (HL)	10101110			A EXCLUSIVE OR (HL) to A	●	●	●	0
XOR (IX+d)	11011101	10101110	d	A EXCLUSIVE OR (IX+d) to A	●	●	●	0
XOR (IY+d)	11111101	10101110	d	A EXCLUSIVE OR (IY+d) to A	●	●	●	0

Key:

Condition Codes:
- ● = affected
- 0 = reset
- 1 = set
- = unaffected

Instruction Fields:
- b bit field 0-7
- c condition field 0=NZ, 1=Z, 2=NC, 3=C, 4=PO, 5=PE, 6=P, 7=M
- d indexing displacement +127 to −128
- dd register pair: 0=BC, 1=DE, 2=HL, 3=SP
- e relative jump displacement +127 to −128
- n immediate or address value
- pp register pair: 0=BC, 1=DE, 2=IX, 3=SP
- qq register pair: 0=BC, 1=DE, 2=IY, 3=SP
- r register: 0=B, 1=C, 2=D, 3=E, 4=H, 5=L, 7=A
- r' register: same as r
- ss register pair: 0=BC, 1=DE, 2=HL, 3=SP
- t RST field: Location=t*8

Index

Radio Shack®

A TANDY CORPORATION COMPANY