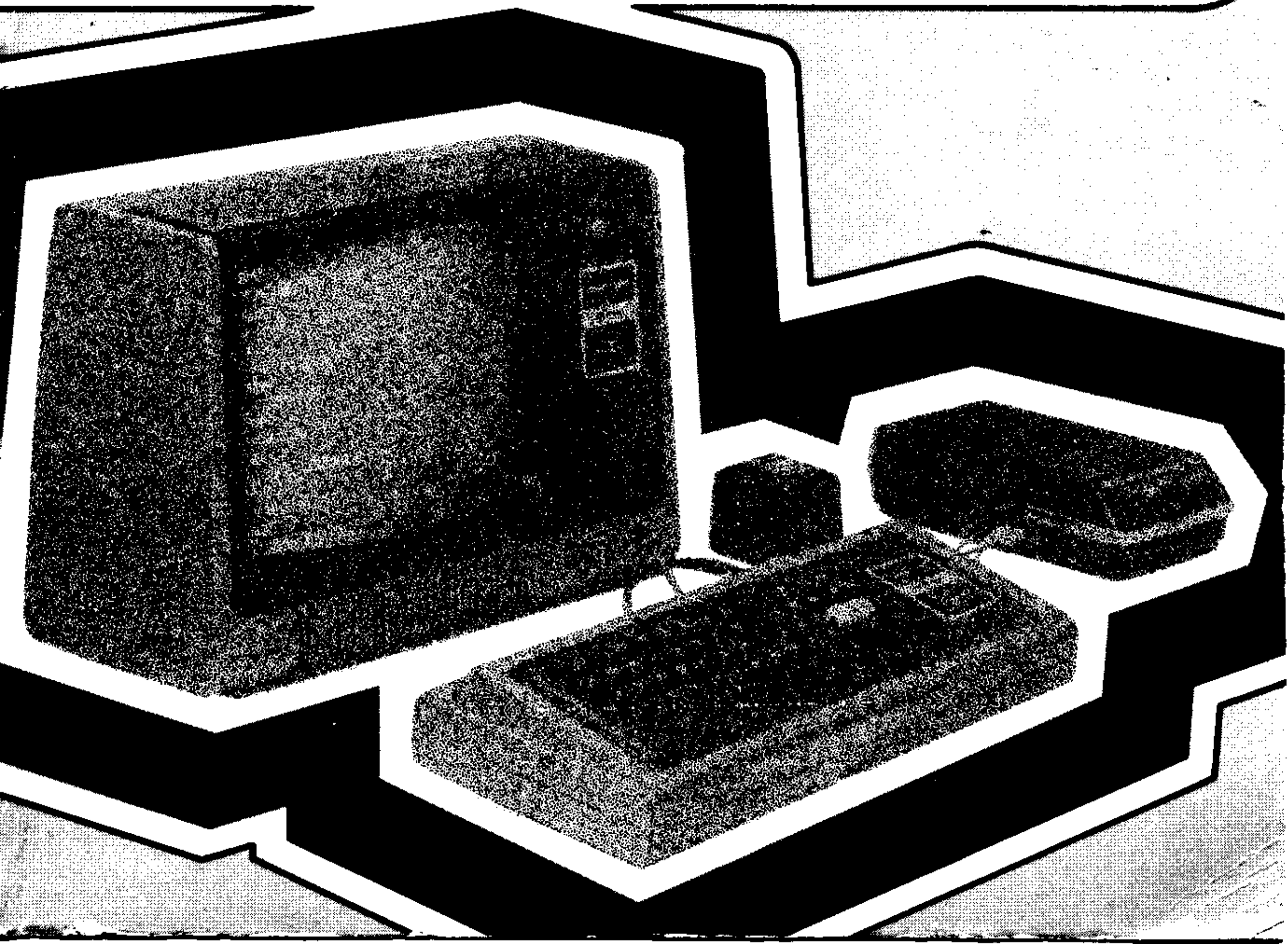
Radio Shack®

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

Copyright © 1979 by Radio Shack, a Tandy Corporation Company, Fort Worth, Texas 76107. Printed in the United States of America.

All rights reserved. Reproduction or use, without express permission, of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein.

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

TRS-80 AND Z-80 ARCHITECTURE 11

Functional Blocks—What Are All These Ones and Zeros—CPU, Memory, and I/O—The Z-80: A Chip Off the Old Block

CHAPTER 2

Z-80 INSTRUCTIONS 24

The Z-80 Family Tree—How Long Is an Instruction—Wait a Microsecond—Instruction Groups—Data Movement—Arithmetic, Logical, and Compare—Decision Making and Jumps—Stack Operations—Shifting and Bit Operations—I/O Operations—A Program of a Thousand Instructions Begins With the First Bit

CHAPTER 3

Z-80 ADDRESSING 41

Why Not One Addressing Mode—Implied Addressing: No Addressing at All—Immediate Addressing—Register Addressing—Register Indirect—Direct Addressing—Relative Addressing—A Special Type of Call—Indexed Addressing—Bit Addressing—Conclusion and Confusion

CHAPTER 4

ASSEMBLY-LANGUAGE PROGRAMMING 58

Machine-Language Coding—TRS-80 Editor/Assembler—Editing New Programs—Assembling—Loading—Assembler Formats—More Pseudo-Ops—A Mark II Version of the Store "1" Program—Further Editing and Assembling

CHAPTER 5

T-BUG AND DEBUGGING 75

Loading and Using T-BUG—T-BUG Commands—T-BUG Tape Formats—Standard Format in Following Chapters

Section II. Programming Methods

CHAPTER 6

MOVING DATA IN BYTES, WORDS, AND BLOCKS 87

Byte and Word Moves—Filling or Padding—An Unsophisticated Block Move—An Elegant Block Move—FILL Subroutine—MOVE Subroutine—Subroutine Format—Stack Operation

CHAPTER 7

ARITHMETIC AND COMPARE OPERATIONS 108

Number Formats: Absolutely and Positively—Signed Numbers—Adding and Subtracting 8-Bit Numbers—Adding and Subtracting 16-Bit Numbers—A Precision Instrument—Decimal Arithmetic—Compare Operations

CHAPTER 8

LOGICAL OPERATIONS, BIT OPERATIONS, AND SHIFTS 131

ANDs, ORs, and Exclusive ORs—Bit Instructions—Shiftless Computers—Rotates—Some Shifting Is Very Logical—Arithmetic Shifts—Software Multiply and Divide—Input and Output Conversions

CHAPTER 9

STRINGS AND TABLES 151

Assembler-Generated Strings — Generalized String Output — String Input—Block Compares—Table Searches—Unordered Tables—Ordered Tables

CHAPTER 10

I/O OPERATIONS 167

Memory Versus I/O—Keyboard Decoding—Display Programming—Mysteries of the Cassette Revealed—Real-World Interfacing—Discrete Inputs

CHAPTER 11

COMMON SUBROUTINE 189

FILL Subroutine—MOVE Subroutine—MULADD Subroutine — MULSUB Subroutine — COMPARE Subroutine — MUL16 Subroutine—DIV16 Subroutine—HEXCV Subroutine—SEARCH Subroutine—SET, RESET and TEST Subroutines

Section III. Appendices

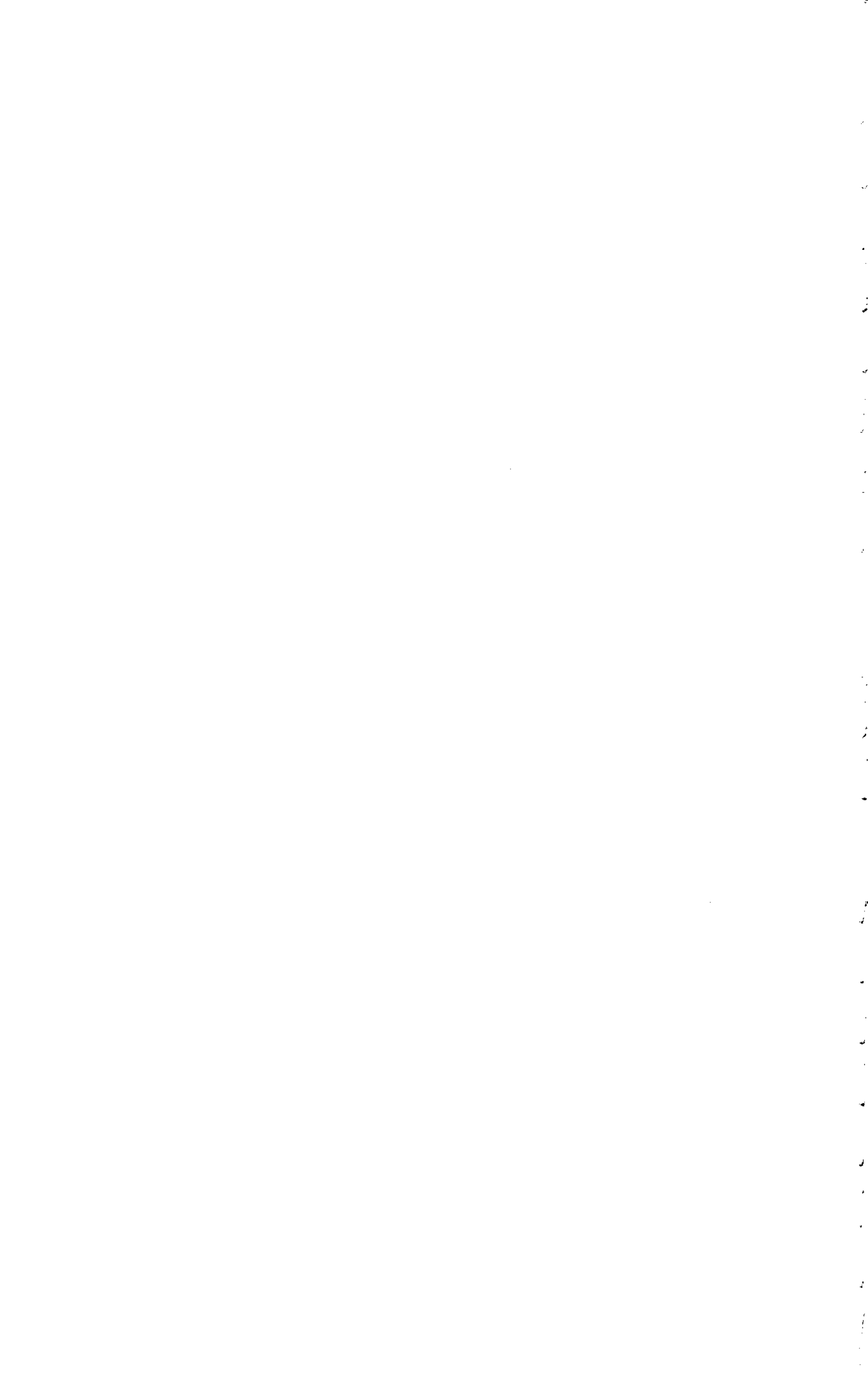
APPENDIX I

Z-80 INSTRUCTION SET 205

APPENDIX II

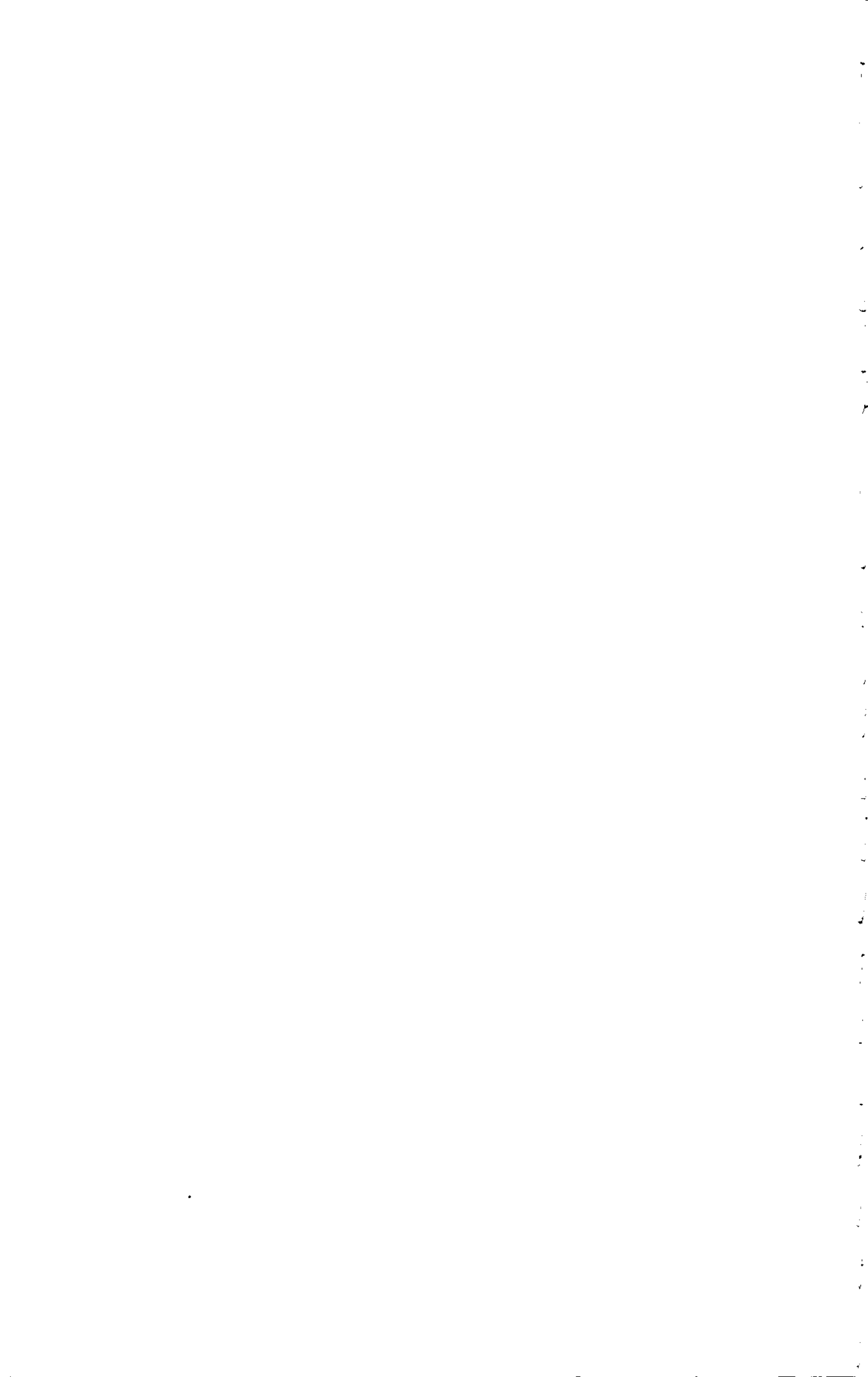
Z-80 OPERATION CODE LISTINGS 209

INDEX 221



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 *c*entral *p*rocessing *u*nit, 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 *i*nput/*o*utput 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,

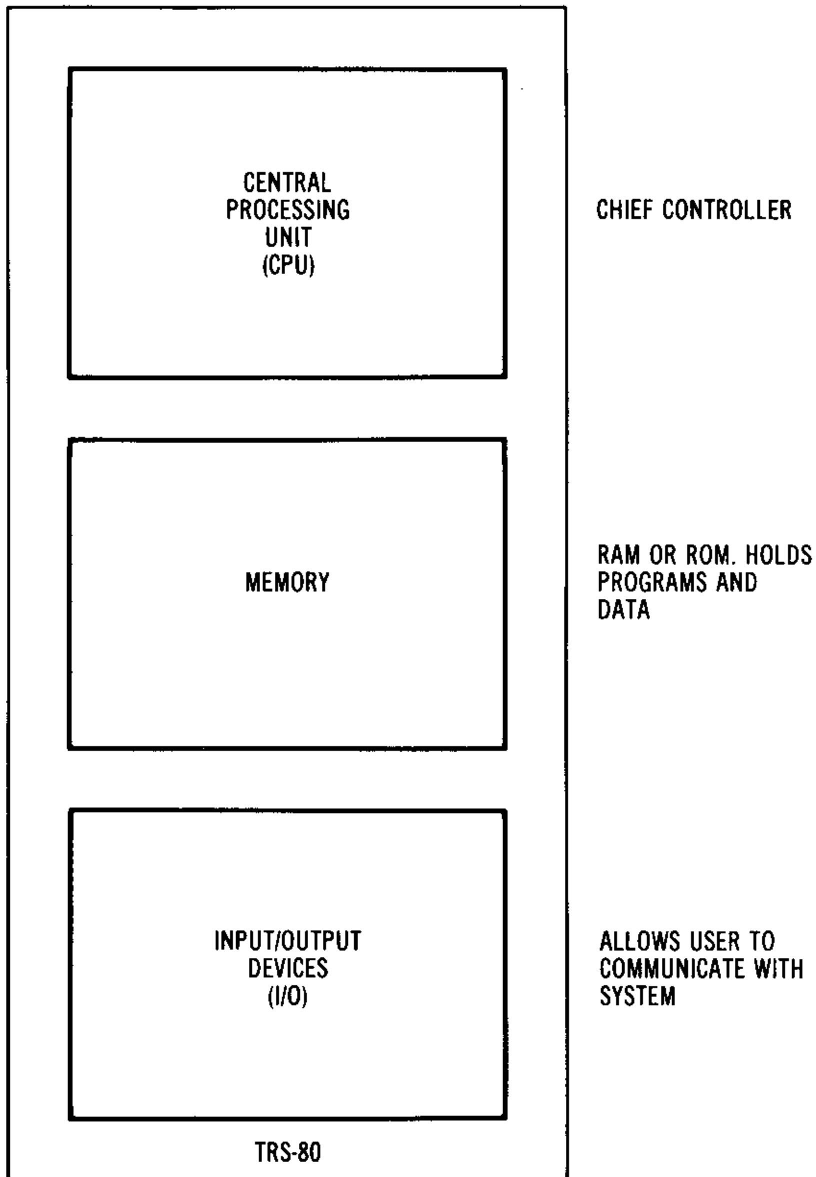


Fig. 1-1. Functional blocks of the TRS-80.

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,

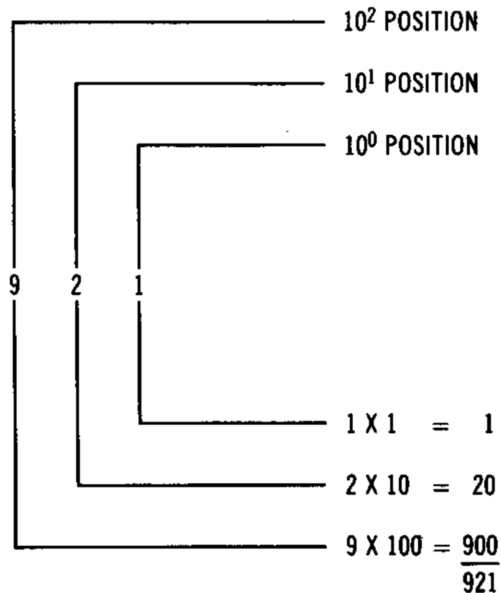


Fig. 1-2. Decimal notation.

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. *B*inary dig*it* 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

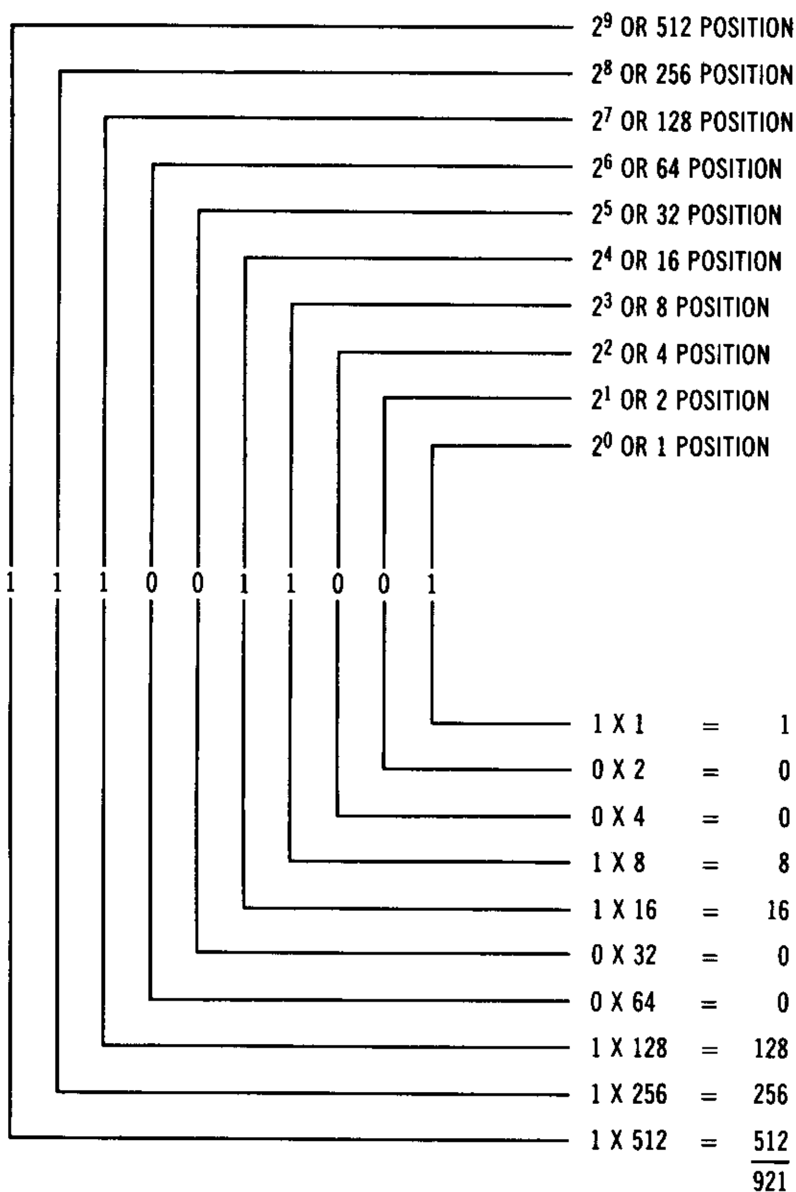


Fig. 1-3. Binary notation.

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*

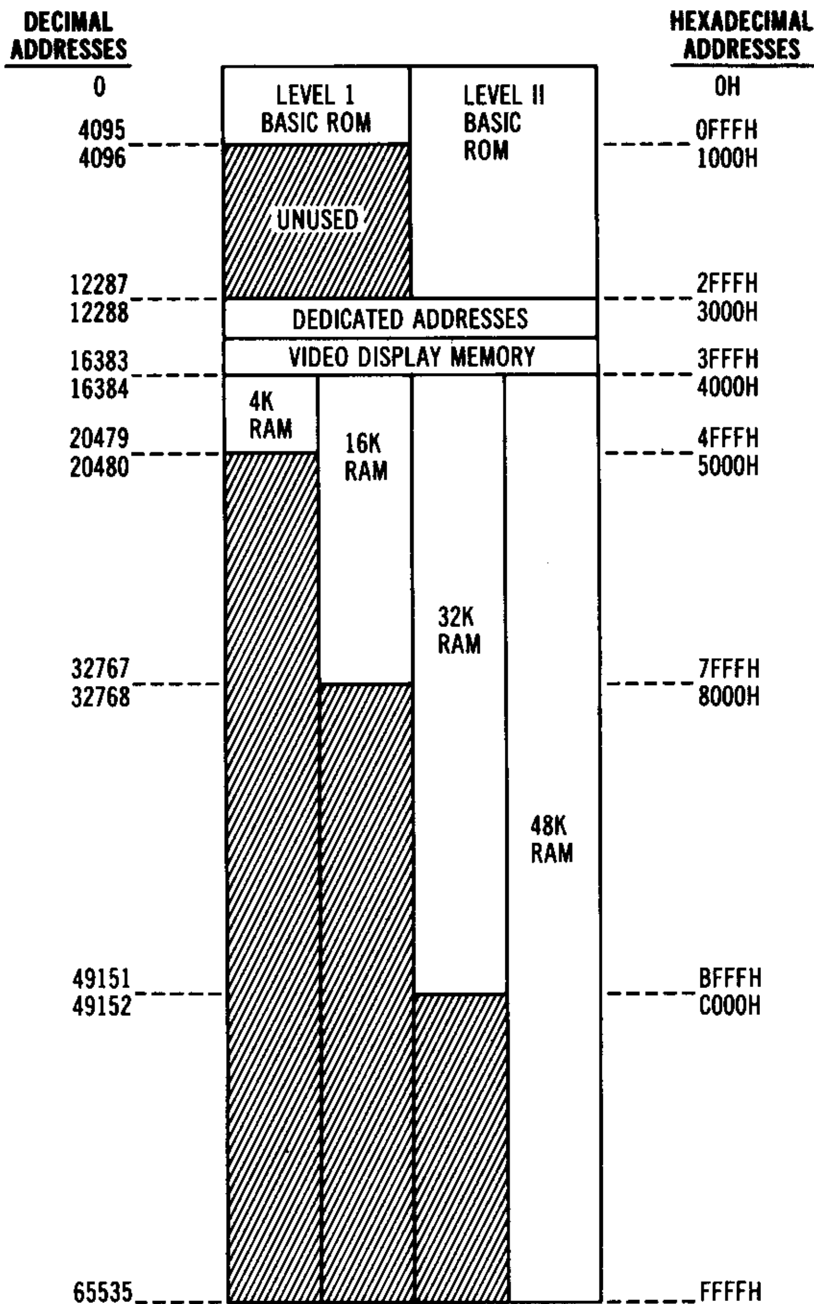


Fig. 1-4. TRS-80 memory mapping.

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

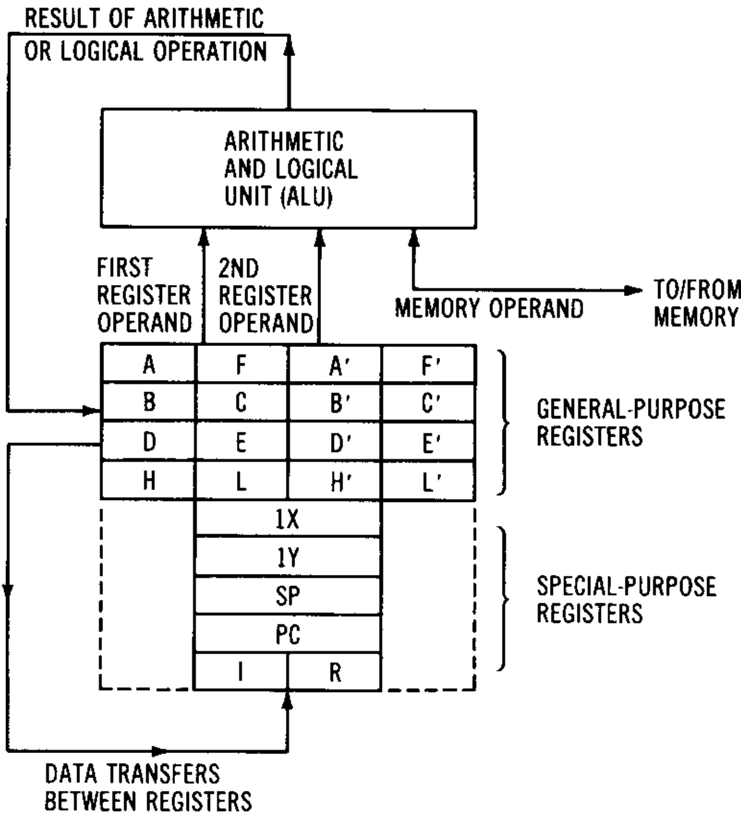


Fig. 1-5. Z-80 architecture.

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

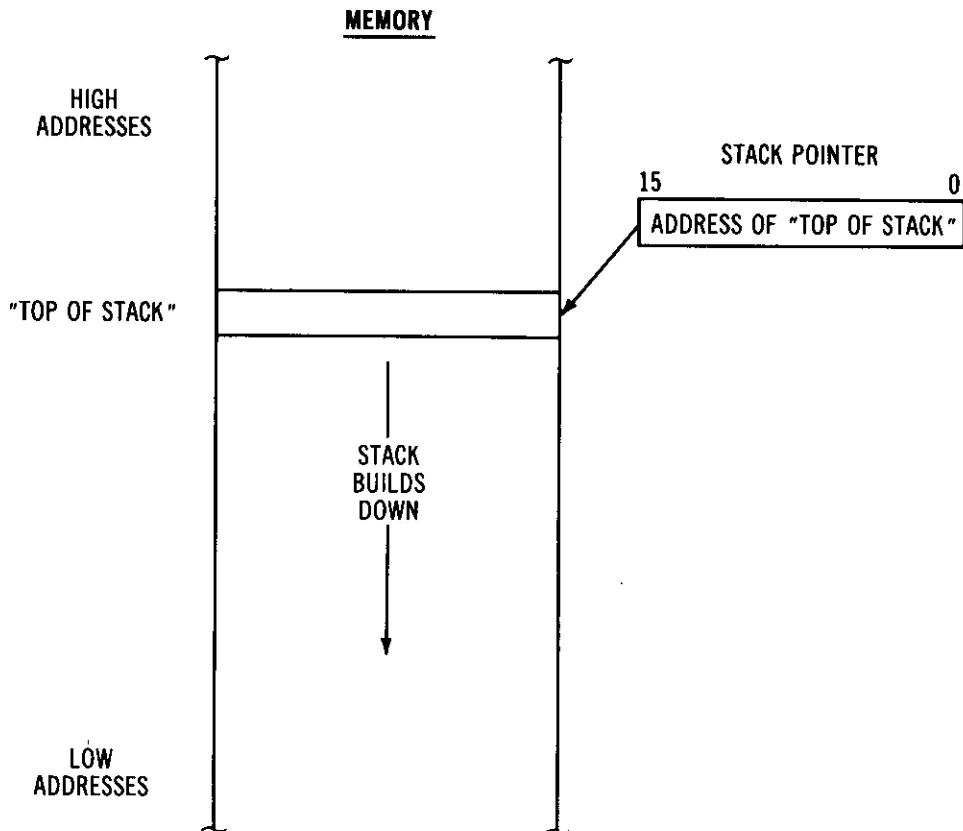


Fig. 1-7. Memory stack.

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-

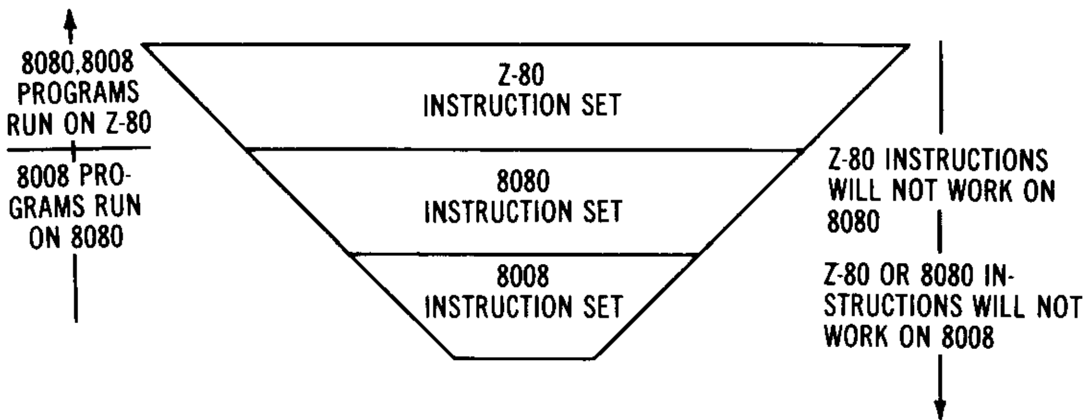


Fig. 2-1. The Z-80 family tree.

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
4A00          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. *LoaD*ing 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.

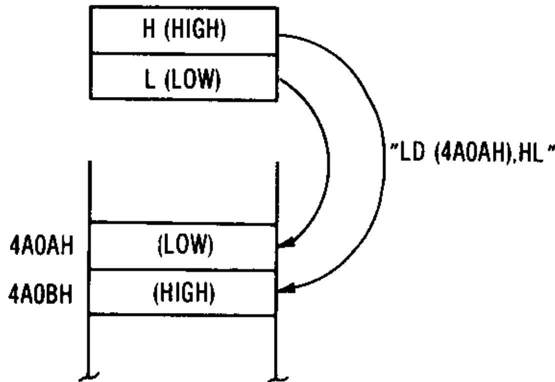


Fig. 2-4. Memory arrangement for 16-bit data.

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*

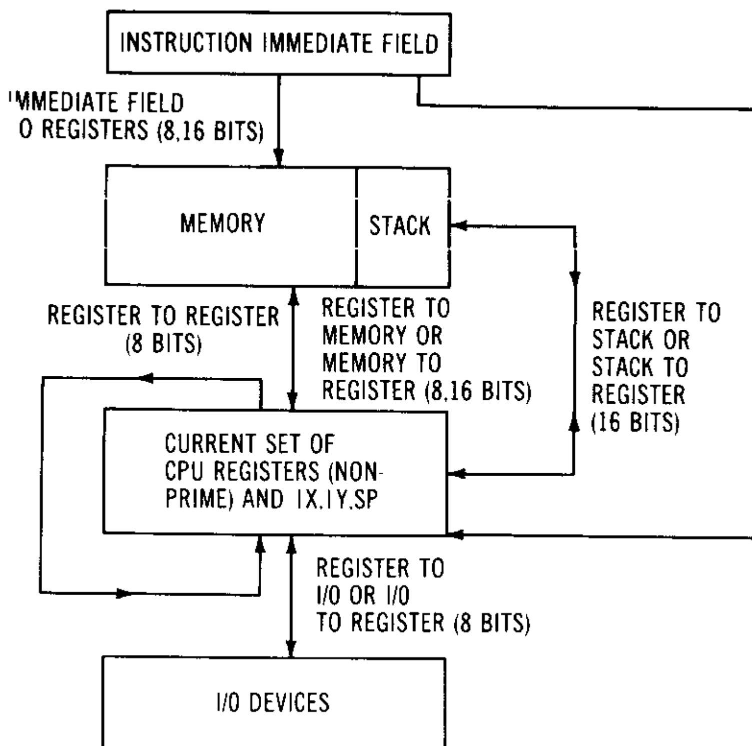


Fig. 2-5. Data transfer paths.

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

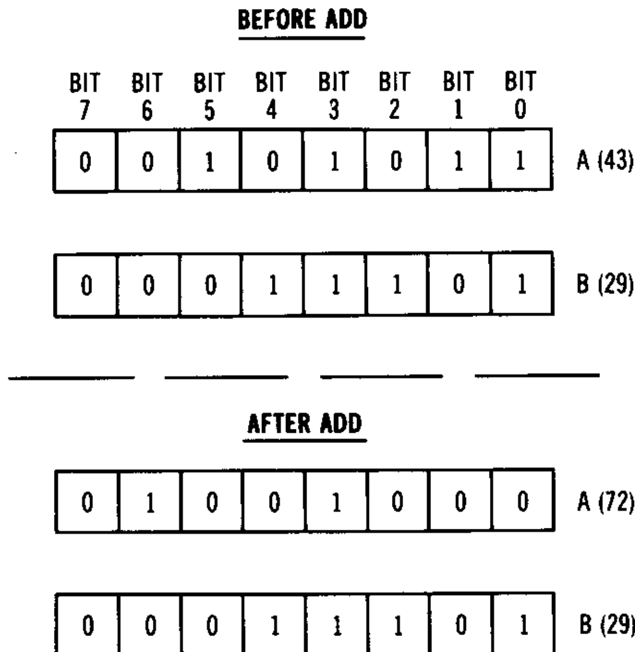
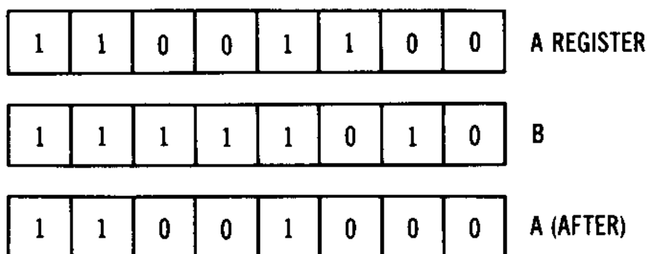


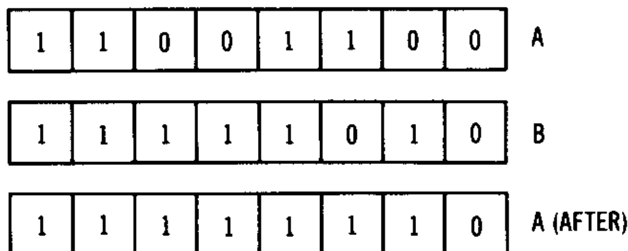
Fig. 2-6. Sample ADD operation.

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.

AND OPERATION



OR OPERATION



EXCLUSIVE OR OPERATION

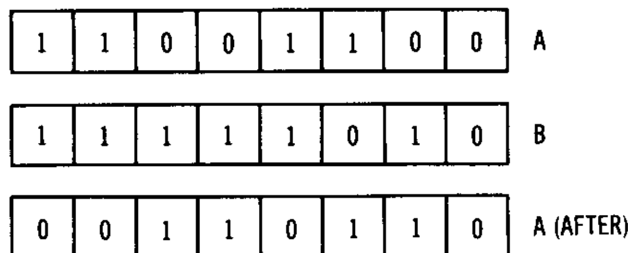


Fig. 2-7. Logical operations.

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.

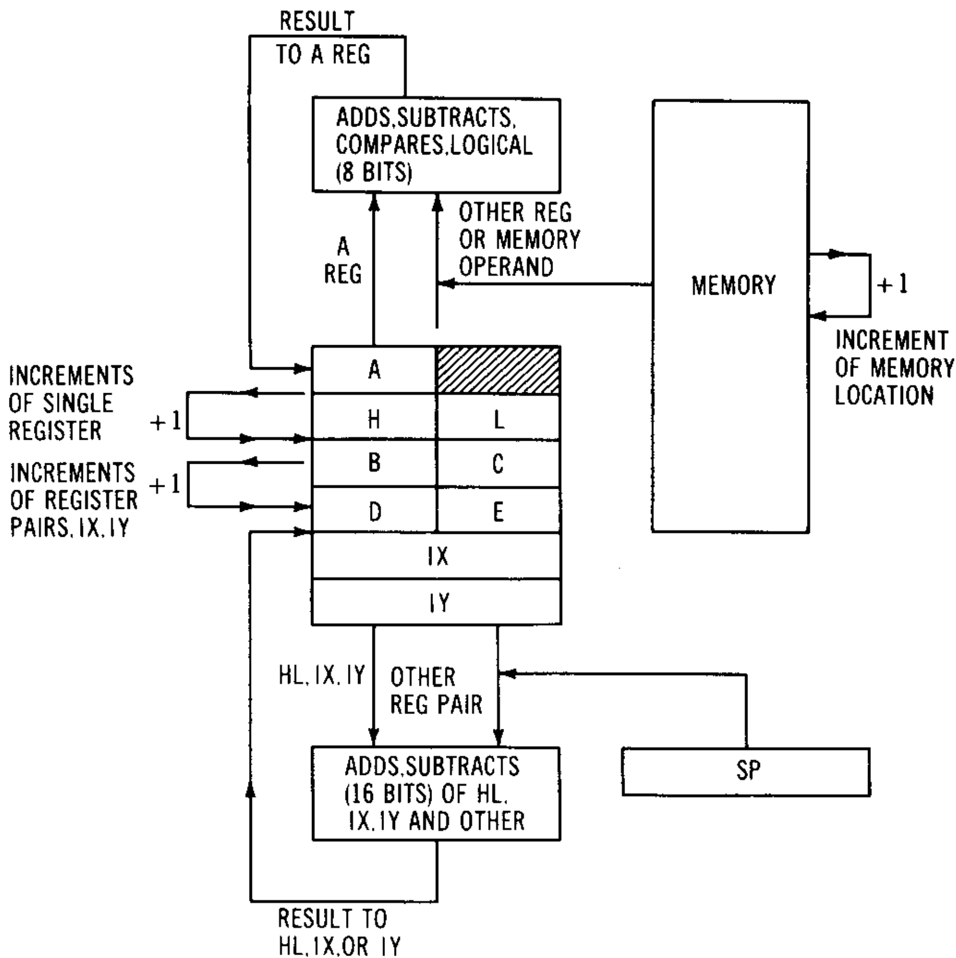


Fig. 2-8. Arithmetic, logical, and compare action.

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-

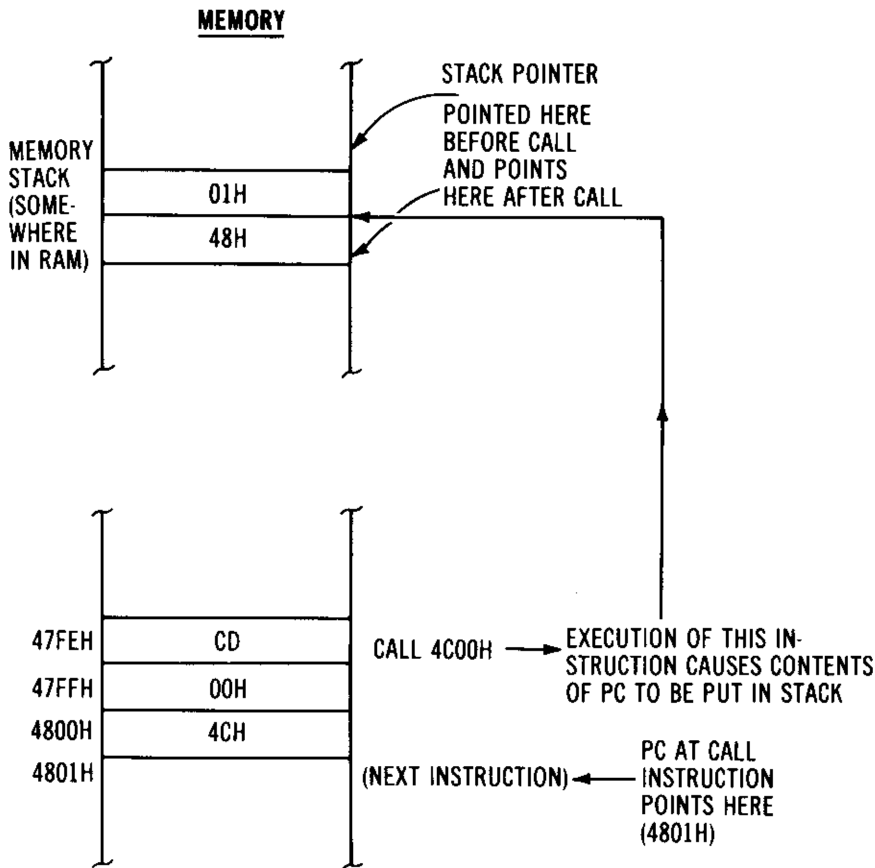


Fig. 2-9. CALL stack action.

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 instrucitions 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

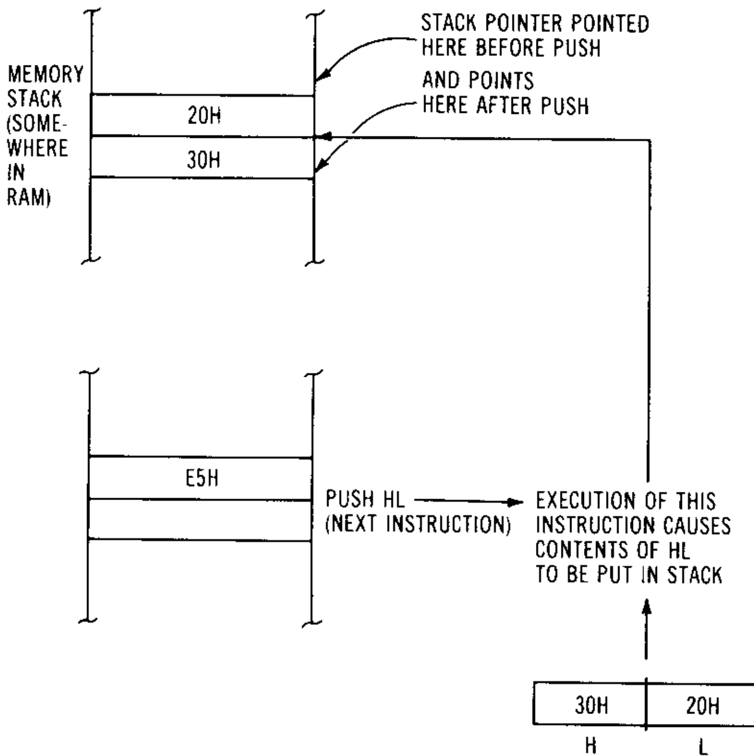


Fig. 2-10. PUSH stack action.

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.

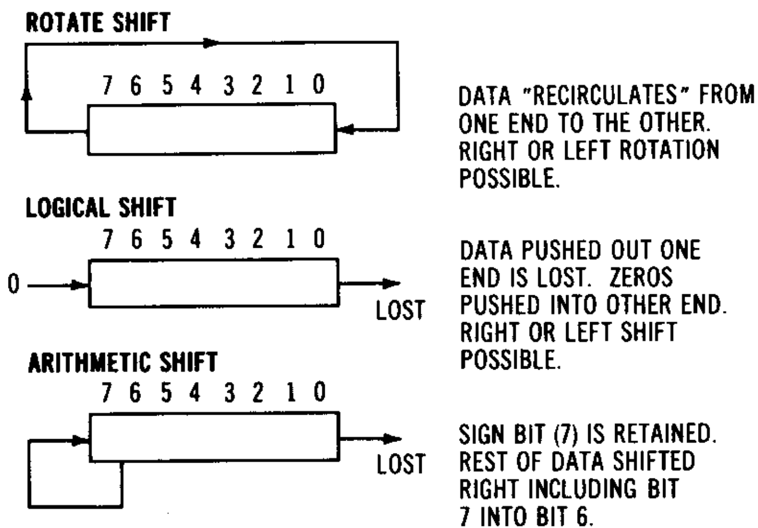


Fig. 2-11. Shifts in the Z-80.

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.

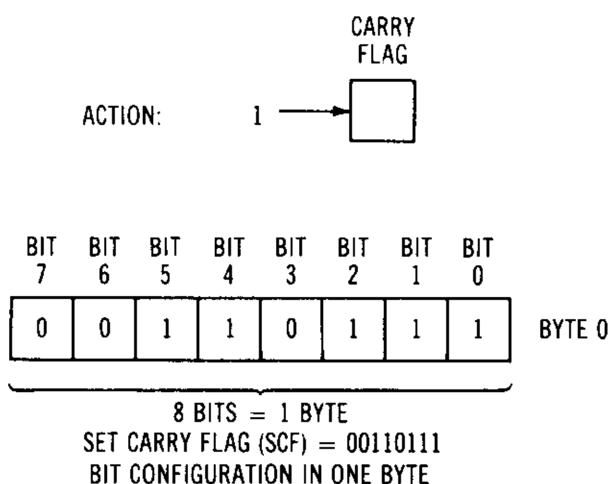


Fig. 3-1. Implied addressing.

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*

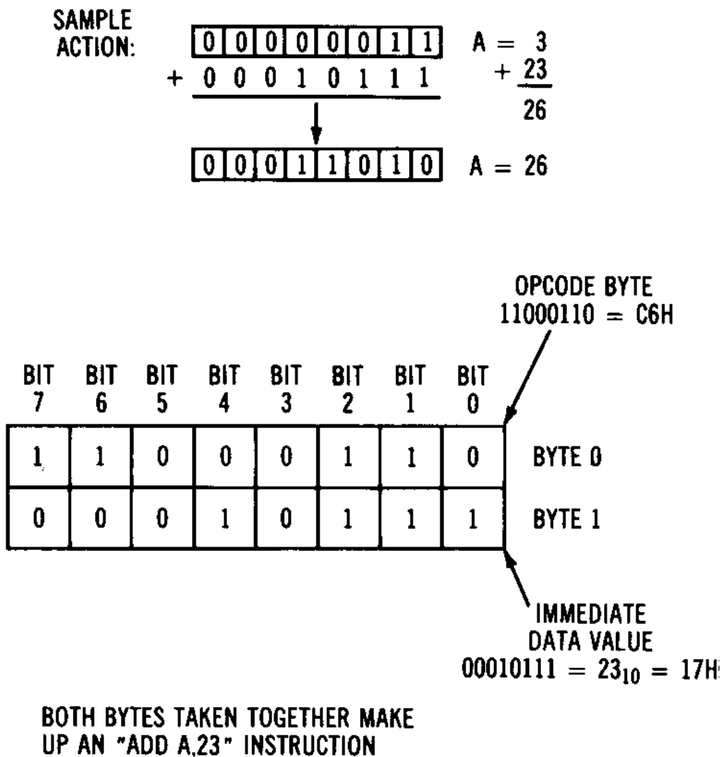


Fig. 3-2. Immediate addressing, 8 bits.

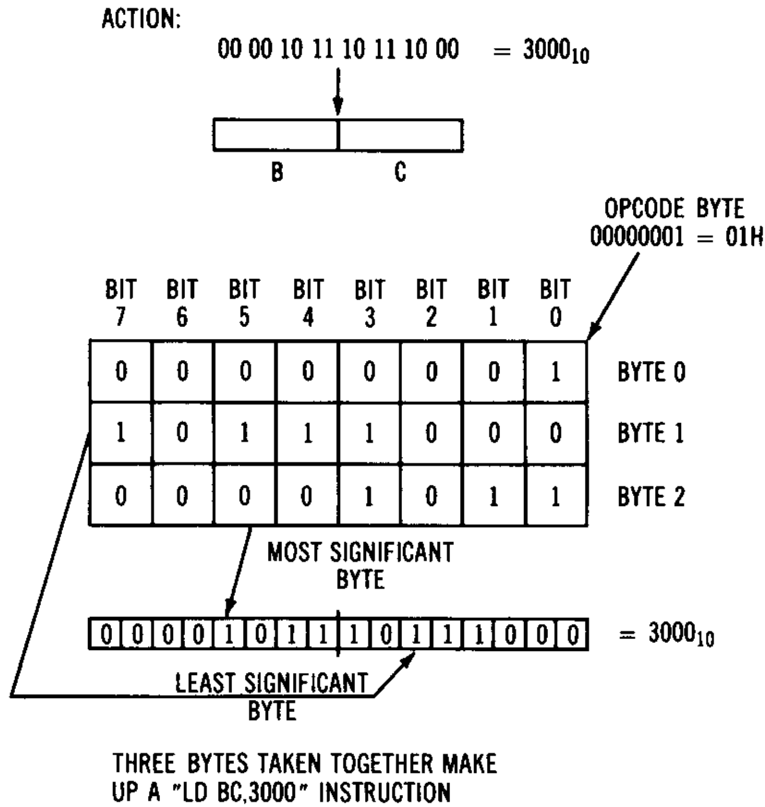


Fig. 3-3. Immediate addressing, 16 bits.

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.

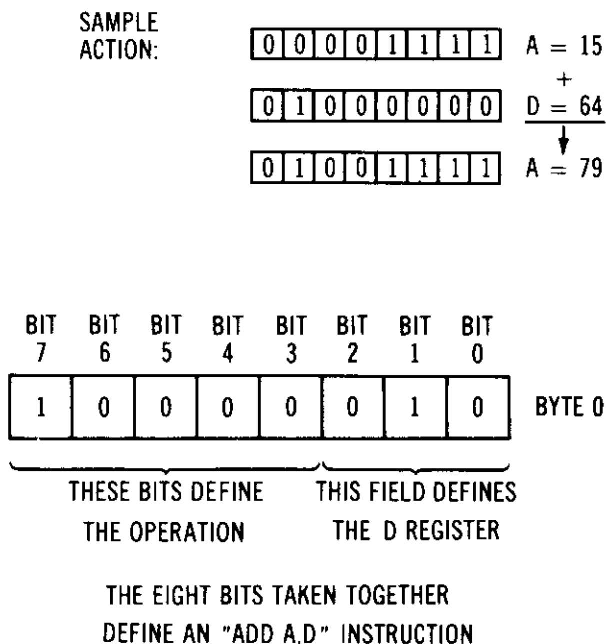


Fig. 3-4. Register addressing.

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.

SAMPLE ACTION:

00000001	00000000	IY = 256_{10} +
01000000	00000000	SP = 16384_{10}
01000001	00000000	IY = 16640_{10}

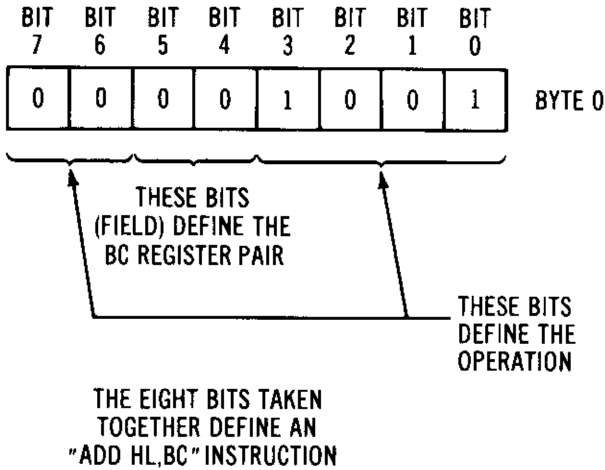
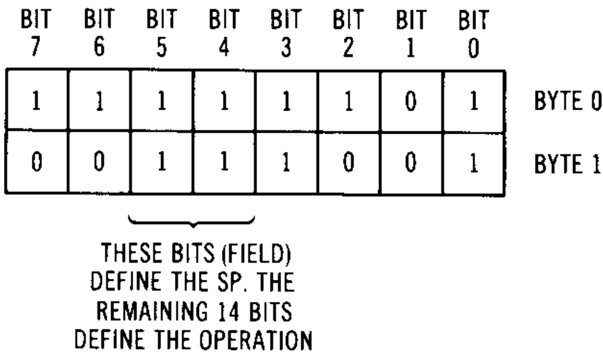


Fig. 3-5. Register pair addressing.

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.

SAMPLE ACTION:

00000001	00000000	IY = 256_{10} +
01000000	00000000	SP = 16384_{10}
01000001	00000000	IY = 16640_{10}



THE TWO BYTES TAKEN TOGETHER DEFINE AN "ADD IY, SP" INSTRUCTION

Fig. 3-6. Index register addressing.

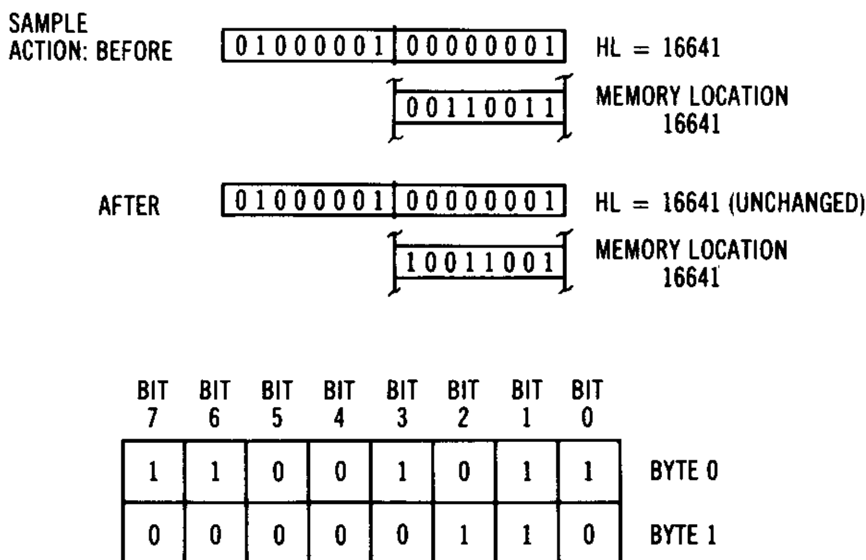
Register Indirect

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.



THE TWO BYTES TAKEN TOGETHER DEFINE AN "RRC (HL)" TYPE INSTRUCTION WHICH USES HL TO DEFINE A MEMORY LOCATION FOR A ROTATE.

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-

SAMPLE ACTION: MEMORY LOCATION 4000H [00011111] → A

"LD A,(4000H)"

THIS BYTE DEFINES THE OPCODE 3AH

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
BYTE 0	0	0	1	1	1	0	1	0
BYTE 1	0	0	0	0	0	0	0	0
BYTE 2	0	1	0	0	0	0	0	0

0100000000000000 $= 4000H = 16384_{10}$

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
BYTE 0	0	0	1	1	0	0	1	0
BYTE 1	0	0	0	0	0	0	0	0
BYTE 2	0	1	0	0	0	0	0	0

THIS BYTE DEFINES THE OPCODE 32H

"LD (4000H),A"

SAMPLE ACTION: A [00011111] → MEMORY LOCATION 4000H

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

MEMORY LOCATION	7 6 5 4 3 2 1 0		PROGRAM COUNTER AFTER BYTE FETCHED
42FEH	C6H	ADD A,5	42FFH
42FFH	05H		4300H
4300H	18H	JR 4350H	4301H
4301H	4EH		4302H
4302H	3AH	LD A,(5000H)	4303H
4303H	00H		4304H
4304H	50H		4305H
4305H	80H	ADD A,B	4306H
4306H	32H	LD (5000H),A	4307H
4307H	00H		4308H
4308H	50H		4309H
4309H	3AH	LD A,(6000H)	430AH
430AH	00H		430BH
430BH	60H		430CH

TO 4350H

PROGRAM COUNTER = 4302H
+ +
DISPLACEMENT = 4EH
EFFECTIVE ADDRESS = 4350H

Fig. 3-11. Relative jump action.

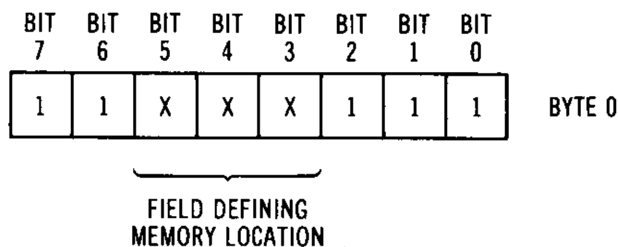
why we have computers!) You'll see in the next chapter that the instruction referenced may actually be given a name, much in the same fashion as a BASIC variable name, which the assembler will use in figuring out what the displacement should be.

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



Field	Address	
000	= 0000H	MEMORY LOCATION FOR CALL ACTION
001	= 0008H	
010	= 0010H	
011	= 0018H	
100	= 0020H	
101	= 0028H	
110	= 0030H	
111	= 0038H	

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-

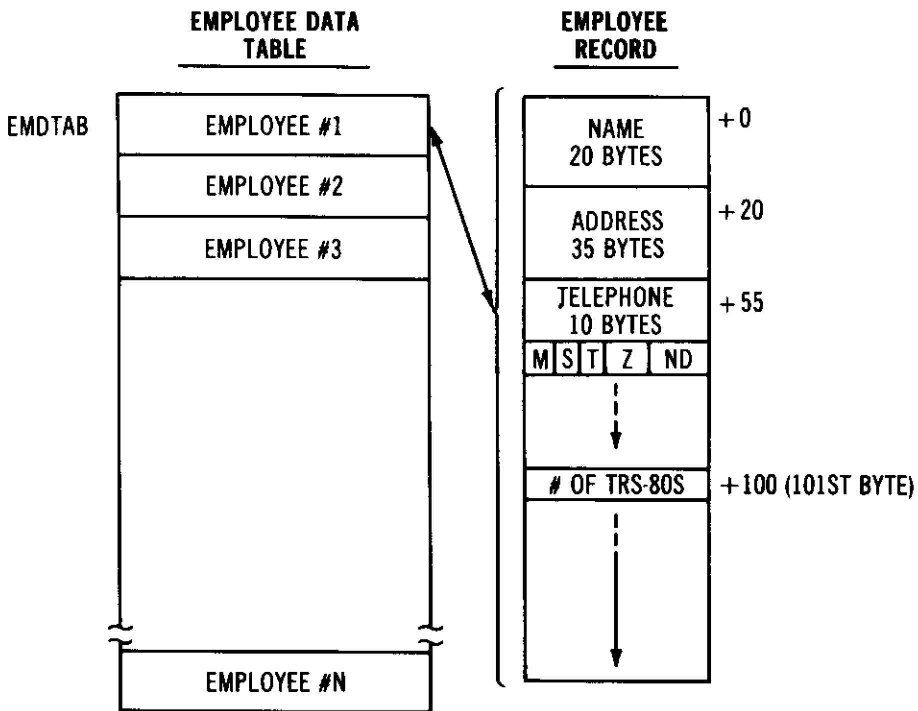


Fig. 3-13. Indexed-addressing table example.

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

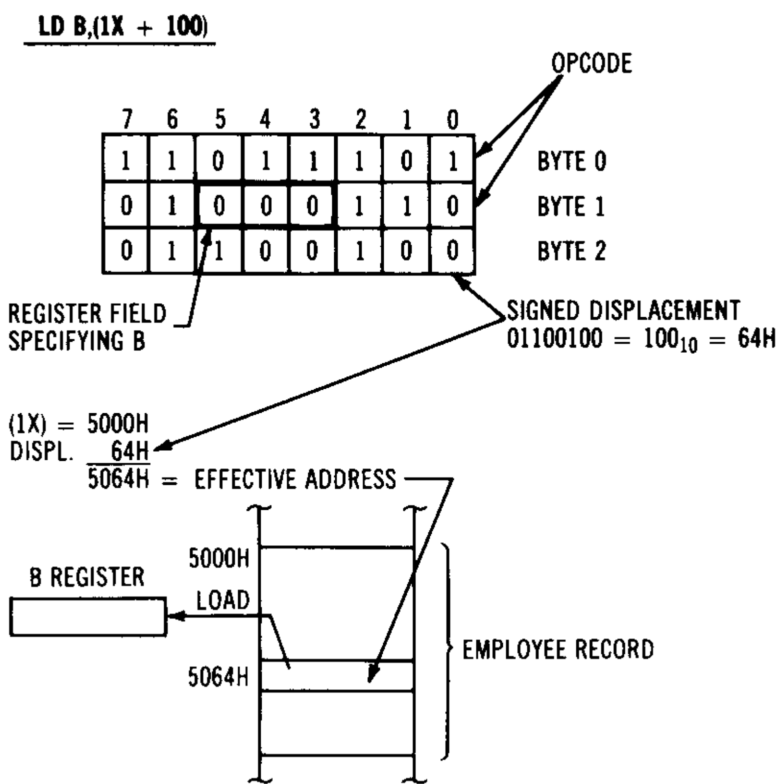


Fig. 3-14. Indexing into table.

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.

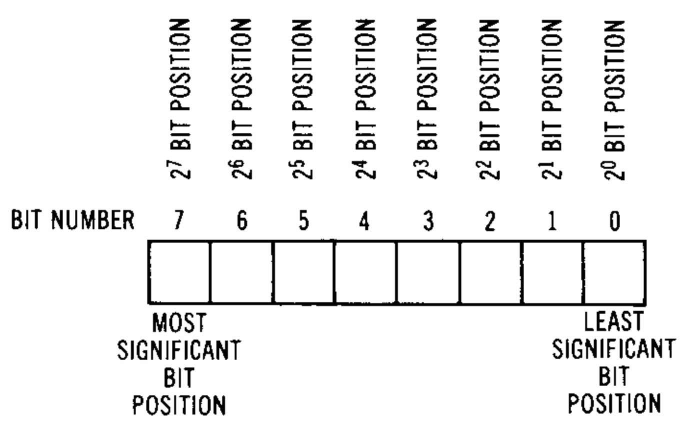


Fig. 3-15. Bit numbering.

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.

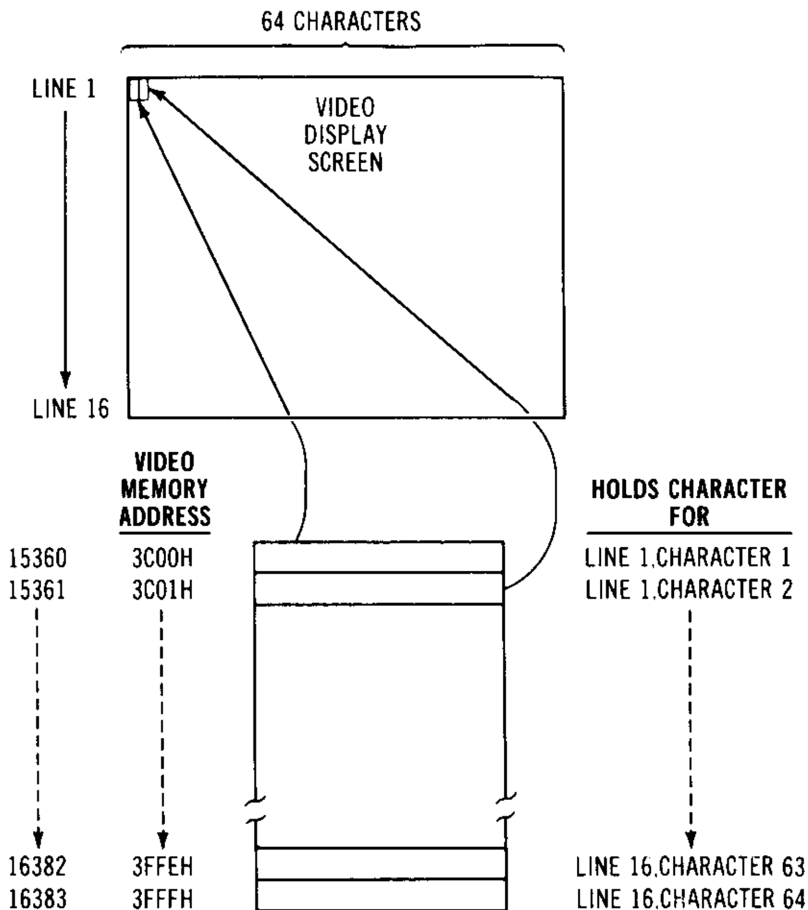


Fig. 4-1. Screen addressing.

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 (*ORiG*in) 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 3/4 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

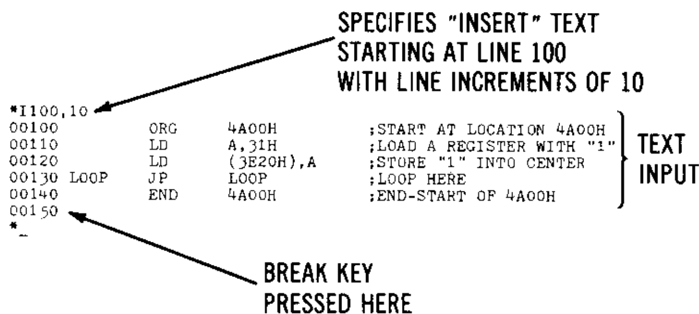


Fig. 4-2. Editing operations.

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
4A00            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)
                 v
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
4A00          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
4A00      00110       ORG     4A00H
4A00 00   00120       DEFB    0
4A01 11   00130       DEFB    11H
4A02 0A   00140       DEFB    0AH
4A03 1100 00150       DEFW    11H
4A05 0A00 00160       DEFW    0AH
4A07 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.

```
4A00      00100       ORG     4A00H
4A00 31   00110       DEFB    '1'            ;ASCII ONE
4A01 23   00120       DEFB    '#'            ;ASCII #
4A02 54   00130       DEFM    'THIS BEATS HAND ASSEMBLING'
4A03 48       4A04 49     4A05 53     4A06 20     4A07 42     4A08 45
  4A09 41     4A0A 54     4A0B 53     4A0C 20     4A0D 48     4A0E 4
1     4A0F 4E     4A10 44     4A11 20     4A12 41     4A13 53     4A
14 53     4A15 45     4A16 4D     4A17 42     4A18 4C     4A19 49
 4A1A 4E      4A1B 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

```
4A00          00100          ORG     4A00H
4C00          00110 BUFFER   EQU     4C00H          ;INPUT BUFFER
4A00 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.

```
4A00          00100          ORG     4A00H
0041          00110 ASCA     DEFL    'A'
              00120
              00130
4A00 3E41     00140          LD      A,ASCA         ;LOAD UPPER CASE
4A02 DD7700   00150          LD      (IX+0),A               ;STORE IN BUFFER
              00160
              00170
0061          00180 ASCA     DEFL    'A'+20H        ;CONVERT TO LOWER
4A05 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 ;
4A00            00120           ORG     4A00H
3C00            00130 VIDEO     EQU     3C00H           ;START OF SCREEN VIDEO
4A00 21184A     00140 START     LD      HL,MESSGE       ;SETUP MESSAGE PNTR
4A03 DD21003E   00150           LD      IX,VIDEO+512    ;POINT TO LINE 9
4A07 7E         00160 LOOP      LD      A,(HL)          ;GET NEXT CHARACTER
4A08 FE00       00170           CP      0               ;TEST FOR END
4A0A 2809       00180           JR      Z,DONE          ;GO IF DONE
4A0C DD7700     00190           LD      (IX),A          ;STORE IN VIDEO
4A0F 23         00200           INC     HL              ;ADD ONE FOR MESSAGE
4A10 DD23       00210           INC     IX              ;ADD ONE FOR VIDEO
4A12 C3074A     00220           JP      LOOP            ;CONTINUE
4A15 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.

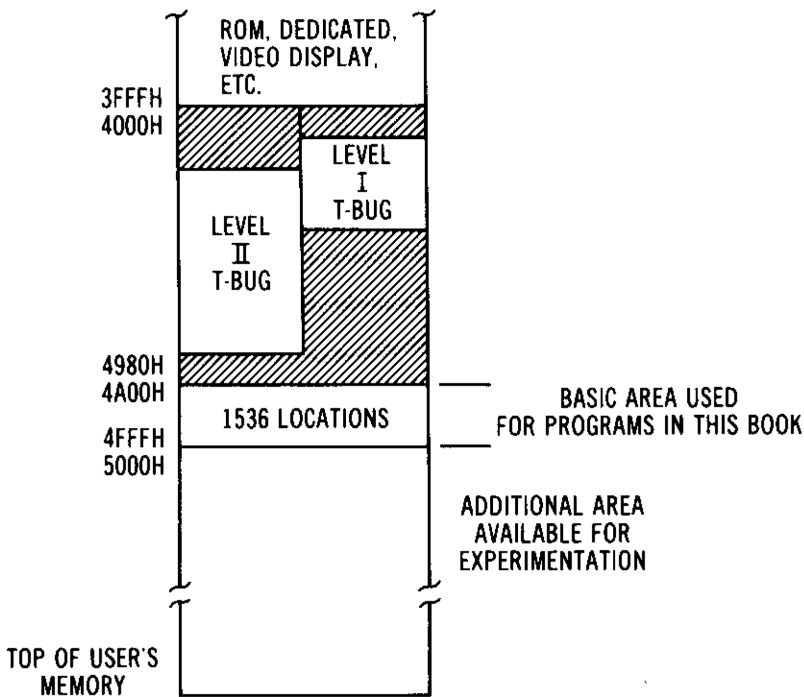


Fig. 5-1. Memory mapping with T-BUG present.

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
4A00          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

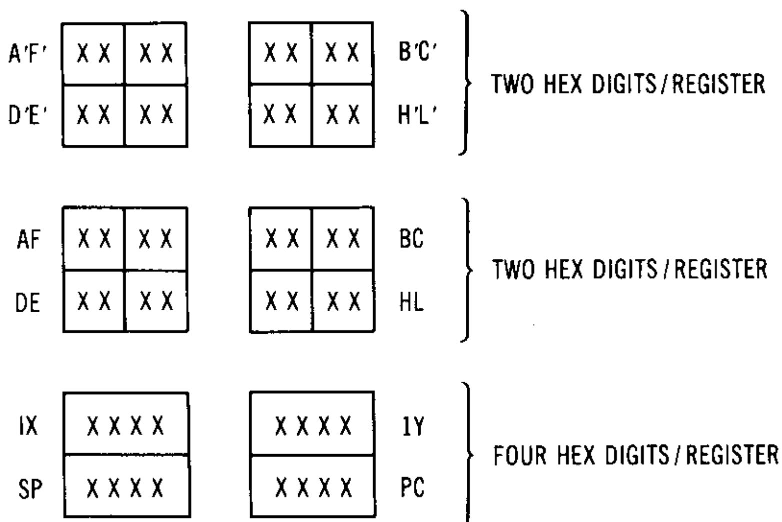


Fig. 5-2. T-BUG R command format.

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

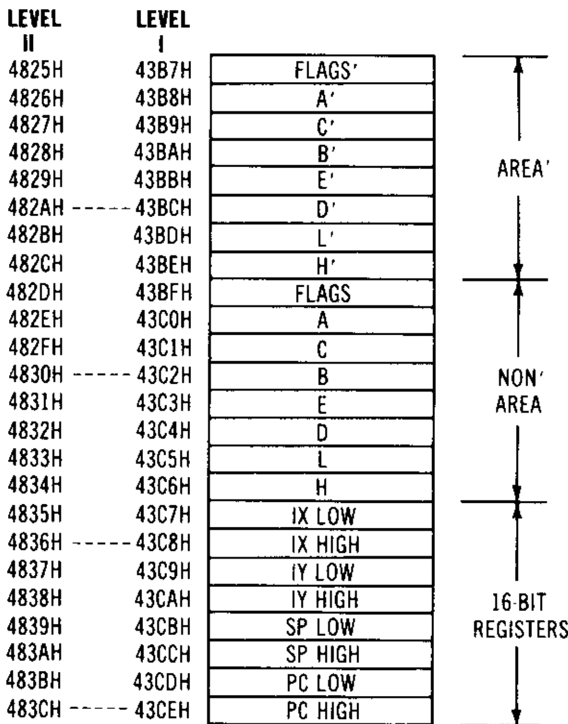


Fig. 5-3. T-BUG register locations.

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 MARKI     (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.

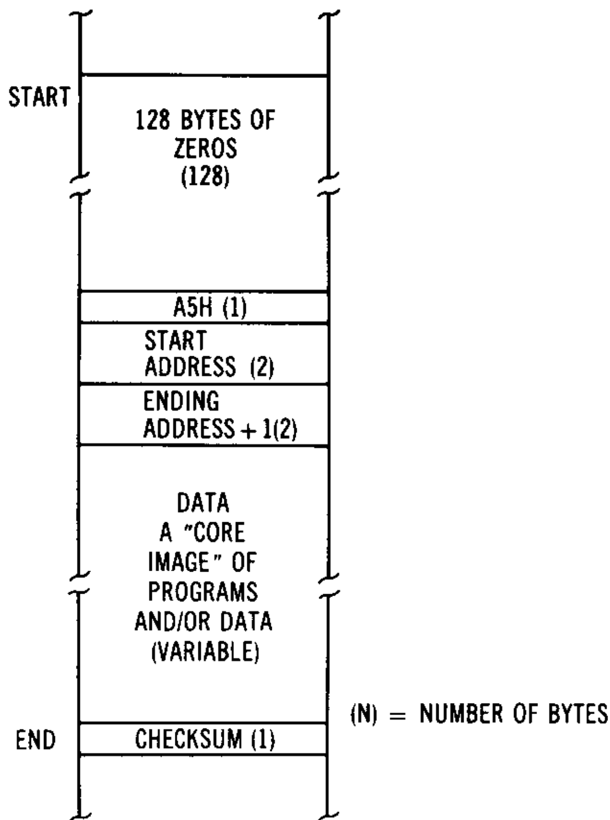


Fig. 5-4. T-BUG tape format.

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
4A00            00110       ORG    4A00H
4A00 3E09       00120 START LD     A,9
4A02 060B       00130       LD     B,11
4A04 0E0C       00140       LD     C,12
```

```
4A06 160D     00150       LD    D,13
4A08 1E0E     00160       LD    E,14
4A0A 260F     00170       LD    H,15
4A0C 2E10     00180       LD    L,16
4A0E 08       00190       EX    AF,AF'          ;SWITCH ACTIVE REGS
4A0F D9       00200       EXX                   ;SWITCH ACTIVE REGS
4A10 3E01     00210       LD    A,1
4A12 0603     00220       LD    B,3
4A14 0E04     00230       LD    C,4
4A16 1605     00240       LD    D,5
4A18 1E06     00250       LD    E,6
4A1A 2607     00260       LD    H,7
4A1C 2E08     00270       LD    L,8
4A1E 08       00280       EX    AF,AF'          ;SWITCH BACK
4A1F D9       00290       EXX                   ;SWITCH BACK
4A20 C3204A   00300 END   JP    END             ;LOOP HERE
0000          00310       END
00000 TOTAL ERRORS
END     4A20
START   4A00
```

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
4A00          00110       ORG   4A00H
4A00 3A0F4A   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 010201     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 BP
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

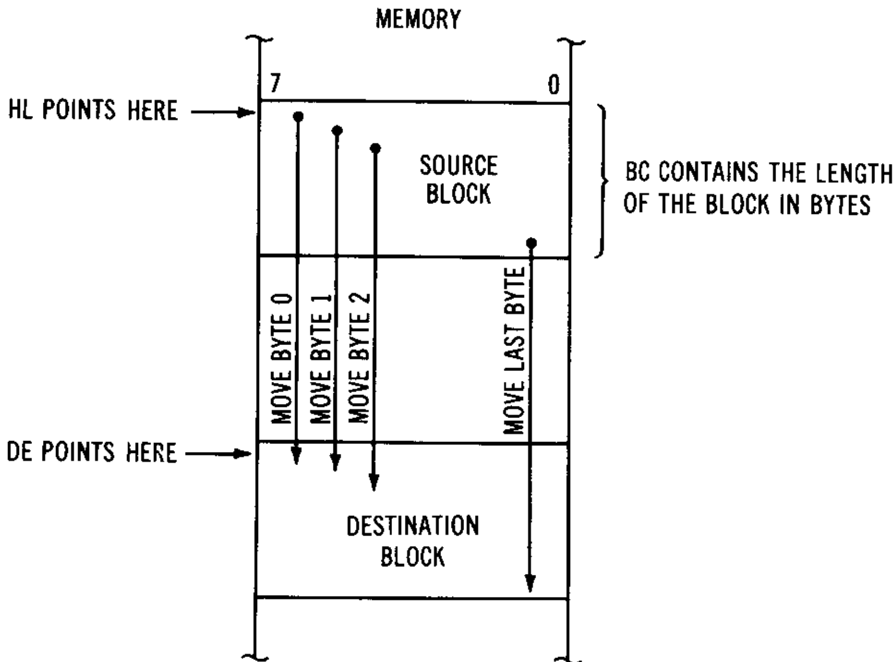


Fig. 6-1. An unsophisticated block move.

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 ;
4A00            00140         ORG     4A00H
4A00 210000     00150 START   LD      HL,0          ;SOURCE
4A03 11003C     00160         LD      DE,3C00H      ;DESTINATION
4A06 01E803     00170         LD      BC,1000       ;1000 BYTES
4A09 7E         00180 LOOP1   LD      A,(HL)        ;GET SOURCE BYTE
4A0A 12         00190         LD      (DE),A        ;STORE
4A0B 23         00200         INC     HL            ;POINT TO NEXT SOURCE
4A0C 13         00210         INC     DE            ;POINT TO NEXT DE
4A0D 0B         00220         DEC     BC            ;DECREMENT COUNT
4A0E 78         00230         LD      A,B           ;GET MS COUNT
4A0F B1         00240         OR      C             ;MERGE LS COUNT
4A10 20F7       00250         JR      NZ,LOOP1      ;GO IF COUNT NOT 0
4A12 18FE       00260 LOOP2   JR      LOOP2         ;LOOP HERE ON DONE
0000            00270         END
00000 TOTAL ERRORS
LOOP2   4A12
LOOP1   4A09
START   4A00
```

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."

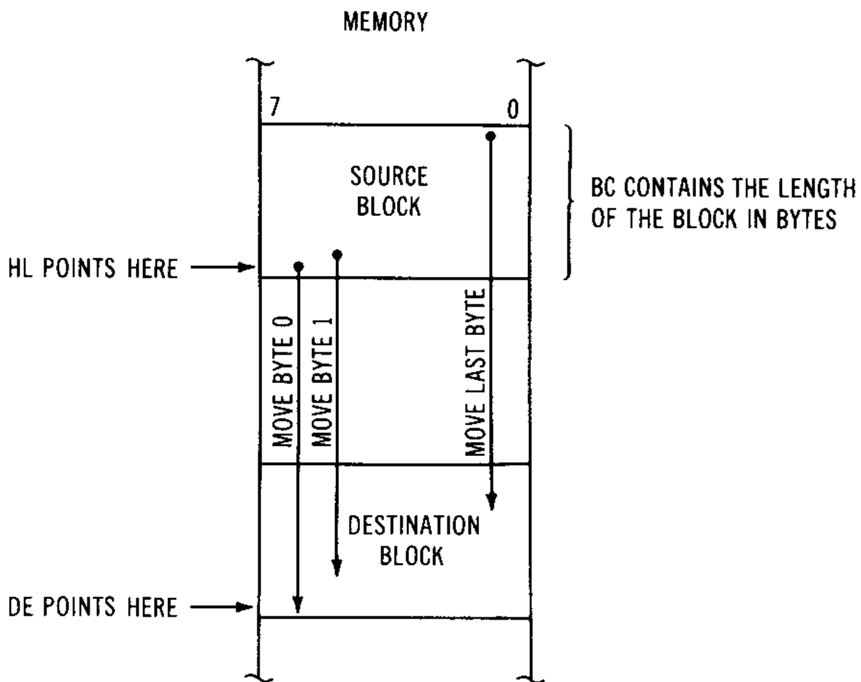


Fig. 6-2. Data transfer for an LDDR.

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 3E80       00270 SLOWLY  LD      A,80H        ;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 ;
```

```
4A00          00200       ORG   4A00H
4A00 72       00210 FILL  LD    (HL),D       ;STORE BYTE
4A01 23       00220       INC   HL           ;BUMP POINTER
4A02 0B       00230       DEC   BC           ;ADJUST COUNT
4A03 78       00240       LD    A,B          ;GET MS OF COUNT
4A04 B1       00250       OR    C            ;MERGE LS COUNT
4A05 20F9     00260       JR    NZ,FILL      ;CONTINUE IF DONE
4A07 C9       00270       RET                ;RETURN IF DONE
0000          00280       END
00000 TOTAL ERRORS
FILL    4A00
```

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 ;
4A00          00180       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 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.

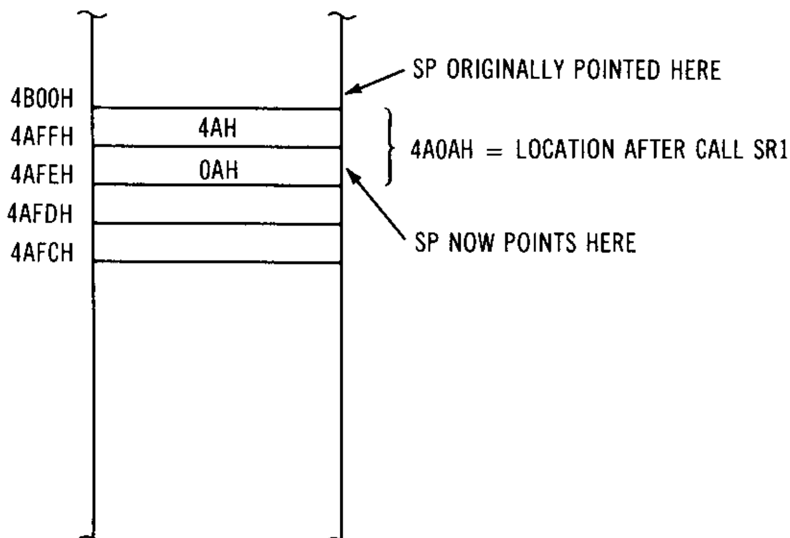


Fig. 6-3. Stack area Example 1.

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
```

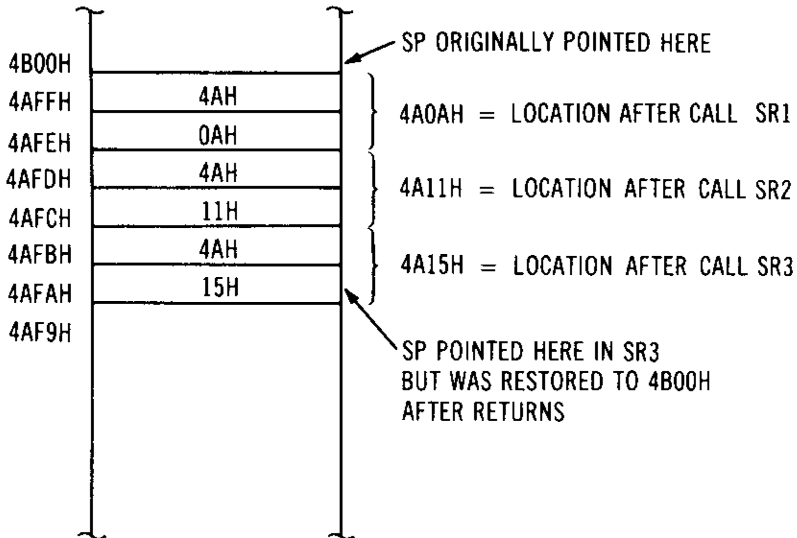


Fig. 6-4. Stack area Example 2.

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.

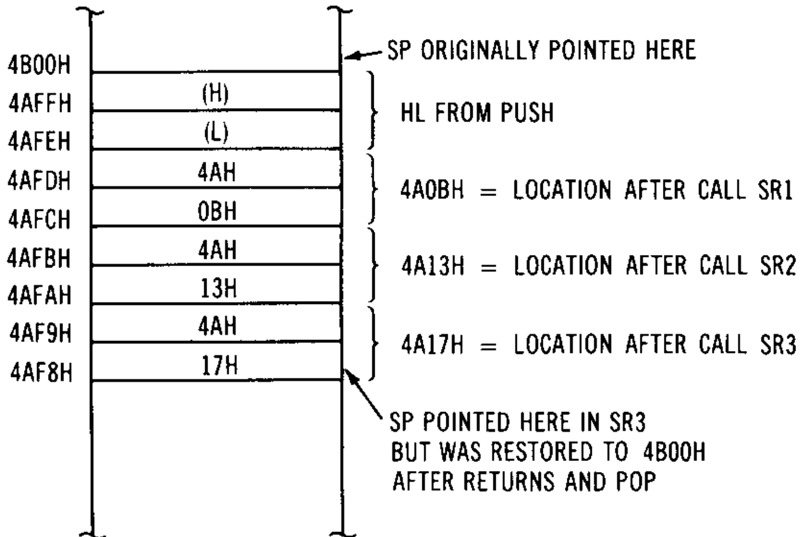


Fig. 6-5. Stack area Example 3.

```
                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.

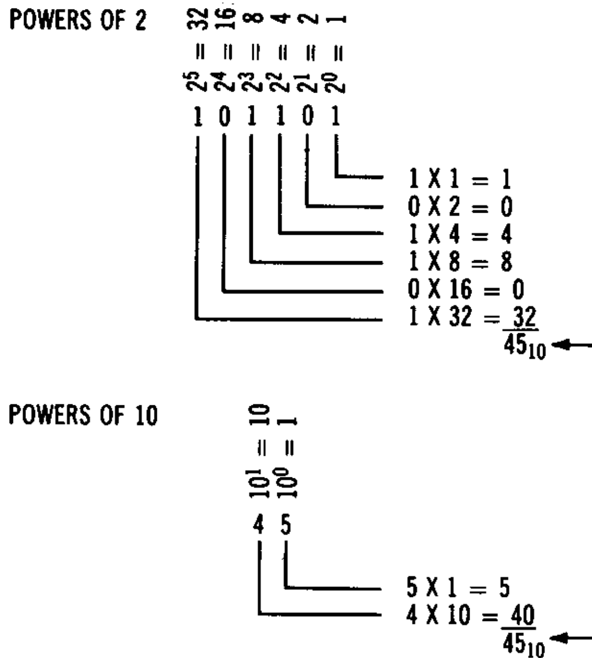


Fig. 7-1. Decimal versus binary numbers.

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.

TO CONVERT FROM BINARY TO DECIMAL

1. LIST POWERS OF TWO REPRESENTED
2. ADD TO FIND DECIMAL NUMBER.

```
0 1 0 0 0 1 0 1
  ↓       ↓   ↓
  64  +   4 + 1 = 69
```

TO CONVERT FROM DECIMAL TO BINARY

```
 105
 -64   (2^6)
  41
 -32   (2^5)     0 1 1 0 1 0 0 1
   9
  -8   (2^3)
   1   (2^0)
```

1. SUBTRACT LARGEST POSSIBLE POWER OF TWO.
2. PUT A BINARY 1 IN THE APPROPRIATE BIT POSITION.
3. CONTINUE UNTIL 0 REMAINS.
4. FILL IN REMAINDER OF BIT POSITIONS WITH ZEROS.

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

CONVERTING FROM BINARY TO DECIMAL AND BACK

1. GROUP BINARY # INTO 4-BIT GROUPS.

0101 1101 0011 0111

2. CHANGE EACH 4 BIT GROUP INTO A HEXADECIMAL DIGIT 0-9,A-F

5 D 3 7

REVERSE PROCESS TO CONVERT FROM HEX TO BINARY

CONVERTING FROM HEXADECIMAL TO DECIMAL

1. LIST POWERS OF 16 REPRESENTED.

$16^3 = 4096$ $16^2 = 256$ $16^1 = 16$ $16^0 = 1$

5 D 3 7

2. MULTIPLY BY DIGIT TO FIND DECIMAL.

4096 256 16 1

5 D 3 7

7 X 1 =	7
3 X 16 =	48
13 X 256 =	3328
5 X 4096 =	20480
	23863

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