8080/Z80 Assembly Language

Techniques For Improved Programming

Alan R. Miller

THE 8080/Z-80 ASSEMBLY LANGUAGE

TECHNIQUES FOR IMPROVED PROGRAMMING

ALAN R. MILLER
Professor of Metallurgy
New Mexico Institute of Mining and Technology
Socorro, New Mexico

Software Editor, Interface Age
Cerritos, California

John Wiley & Sons, Inc., Publishers
New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Judy Wilson
Production Manager: Ken Burke
Editorial Supervision: Winn Kalmon
Line Artist: Carl Brown
Page Makeup: Meredythe

Library of Congress Cataloging in Publication Data

Miller, Alan R. 1932-
8080/Z-80 assembly language.

Includes index.
1. INTEL 8080 (Computer)—Programming. 2. Zilog model Z-80 (Computer)—Programming. 3. Assembler language (Computer program language) I. Title.
QA76.8.I28 M53 001.64'2 80-21492
ISBN 0-471-08124-8

Printed in the United States of America

81 82 10 9 8 7 6 5

Preface

On first thought, it might seem strange that another book on the 8080 and Z-80 should appear at this time. Z-80 CPU cards generally became available in 1977 and the 8080 CPU is even older. But the Z-80 computer seems to become more popular with time. For example, the TRS-80 Model II announced recently by Radio Shack, and Heath's H-89 both use the CPU. High-level languages such as Pascal, APL, BASIC, FORTRAN, and C are now run on the 8080 and Z-80. Furthermore, Microsoft has available a Z-80 CPU card that can be easily inserted into the Apple II computer. There should be an increasing interest in the 8080 and Z-80 CPUs in the coming years, and I believe, a great increase in the number of 8080 and Z-80 programmers. So, there is a growing need for a book that covers programming for the 8080 and Z-80 assembly languages.

The combination of 8080 and Z-80 programming concepts into a single work is quite natural. The Z-80 CPU is upward compatible from the 8080 so that all commercially available 8080 software will run on the Z-80. Furthermore, 8080 assemblers, such as ASM provided with CP/M, can be used to create programs that will run on either an 8080 system or a Z-80 system.

The purpose of this book is twofold. First, I want to provide a single reference source for both 8080 and Z-80 assembly language programmers. The appendixes are designed with this goal in mind. They begin with the ASCII character set and a 64K memory map. These two appendixes are as useful to those using higher level languages as they are to assembly language programmers.

The 8080 and Z-80 instruction sets are listed both alphabetically and numerically in the next four appendixes. This is followed by a cross reference between the 8080 and the Z-80 mnemonics. An appendix describing each instruction in detail then follows. Common acronyms are identified

next in Appendix I, and some undocumented Z-80 instructions are discussed in the final appendix. Collectively, the appendixes contain all of the reference material needed to write 8080 or Z-80 assembly language programs.

The second purpose of this work is to demonstrate some useful techniques of assembly language programming. As an editor for Interface Age, I have seen numerous examples of inefficient or improper programming. General principles of assembly language programming are discussed in Chapters One through Five; specific programming examples are given in Chapters Five through Ten. The reader can actually assemble the programs and try them out.

The organization and operation of the 8080 and Z-80 CPUs is covered in Chapter One. This includes a discussion of the general-purpose registers, the flag registers, logical operations, branching, double-register operations, rotation and shifting. The concepts of hexadecimal, octal, and binary numbers, one's and two's complement arithmetic, and the use of logical operations are presented in Chapter Two.

Stack operations with PUSH, POP, CALL and RET commands and the passing of data between calling program and subroutine are given in Chapter Three. Chapter Four is devoted to input and output techniques, including an interrupt-dirven keyboard routine and a telephone transmission program. Assembler macros are discussed in Chapter Five. Examples show how to generate Z-80 instructions with an 8080 macro assembler, and how to emulate Z-80 instructions on an 8080 CPU.

The reader can develop a small, powerful monitor in Chapter Six using the top-down programming method. The monitor contains the usual commands of dump, load, and go. In addition, there is a memory test, a routine to search for one or two hex bytes or ASCII characters, a routine to replace all occurrences of one byte with another, and a routine to perform input and output through any port.

In Chapter Seven the monitor is converted to Z-80 instructions and some additional features are added. Assembly-language subroutines for interconverting between binary numbers and ASCII characters coded in one of the common number bases are given in Chapter Eight. These routines perform all of the input and output through the system monitor developed in Chapter Six. Paper tape and magnetic tape routines are given in Chapter Nine. This method of data transfer is still very popular. I frequently am asked to read information on paper tape into our Z-80 computer so that it can be transmitted over the telephone line to our campus Dec-20 computer.

CP/M is currently the most popular 8080/Z-80 operating system. Chapter Ten demonstrates how assembly language programs can utilize CP/M for all input and output by presenting three programs. One of these programs allows the user to branch to any address from the system level. Nevertheless, the use of CP/M is not the subject of this book. More information on the use of the CP/M operating system can be obtained from *Using CP/M: A Self-Teaching Guide* by Judi Fernandez and Ruth Ashley (John Wiley and Sons, Inc., 1980).

The assembly language programs in this book have all been assembled on an Altair 8800, with an Ithaca Z-80 CPU card and North Star double-density disks. The Lifeboat 2.0 version of CP/M was used as the operating system. The system monitor given in Chapter Six was additionally programmed to run on a TRS-80 Model II, using a Lifeboart 2.2 version CP/M operating system. The alternate version of the input and output routines was used in this case. The Digital Research assembler MAC was used for the 8080 instructions and the Microsoft assembler MACRO-80 was used for the Z-80 code. All of the assembly listings have been reproduced directly from the original computer printouts. The manuscript was created and edited with MicroPro's Word-Master and formatted with Organic Software's Textwriter.

Thanks to Heidi for typing the manuscript. Also, I should like to acknowledge the programmers at Microsoft, Digital Research, and Lifeboat Associates for the many things they have taught me about programming.

Alan R. Miller
June 1980

Contents

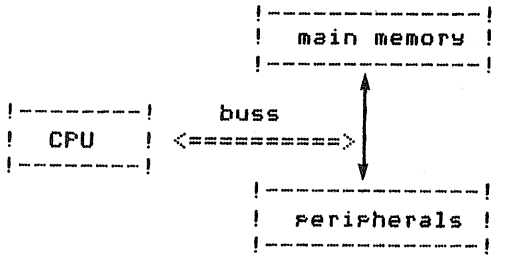
CHAPTER ONE

Introduction

There was a time when computers were gigantic machines containing racks upon racks of vacuum tubes. The invention of the transistor and the development of the integrated circuit (IC) changed all that. Today, it is possible to place tens of thousands of transistors on a single "chip" of silicon that is smaller than a quarter of an inch square. As a result of this technology. computers have become smaller and cheaper.

Computers are commonly classified into three categories, based on size and capability. The largest are known as *main frame computers*, the middle-sized ones are called *minicomputers*, and the smallest are termed *microcomputers*. A computer consists of three parts: the *central processing unit* (CPU), the *main memory*, and the *peripherals*.

The CPU directs the activities of the computer by interpreting a set of *instructions* called *operation codes*, or *op codes* for short. These instructions are located in the main memory. The memory is also used for the storage of data.



The CPU communicates with the user through such peripherals as the console, the printer, the disks, and so on. There are several electrical lines which are used to connect the CPU to the memory and to the peripherals. These lines are collectively known as the *buss*, or *bus*.

The CPU contains a set of *registers*, which are internal memory locations used for data storage and manipulation. One of these is a special register

called the *accumulator*. It receives the results of certain CPU operations. The CPU will also have a *status* register to indicate the nature of a previous operation, e.g., whether the result is zero or negative or positive. It will also indicate whether a carry or a borrow occurred during the operation.

Additional registers are used for auxiliary storage. They may contain general information such as a number that is about to be added to the accumulator. Alternately, a register may contain a number that refers to an *address* in the main memory. The value is called a *memory pointer* in this case. A special portion of main memory may be set aside for storing data. This area is called a *stack*. A special register called a *stack pointer* refers to this region. Another register, the *program counter*, tells the CPU where to find the next instruction in memory.

Computer operations are controlled by a computer *program*. Those programs which are used to solve engineering and physics problems are called *application* programs. On the other hand, computer programs which deal with the operation of the computer's own peripherals are known as *systems* programs.

The instruction set used by the CPU can be very large and difficult to use. Consequently, symbolic programming languages are commonly used instead. An application program may be written in a language such as BASIC, FORTRAN, or Pascal. This is called a *source* program. Then a separate processor program called a *compiler* or an *interpreter* is used to convert the user's source program into an *object* program that corresponds to the instructions needed by the computer.

A microcomputer's instruction set is relatively small compared to that of a larger computer. But even so, it is more convenient to write systems programs in a symbolic language called *assembly language*, rather than in the machine language of the computer. A processor program, called an *assembler*, is then utilized to translate the source program into the corresponding instructions of the computer. A major difference between assembly language and *higher-level* languages such as Pascal is that each line of an assembly language program represents one computer instruction. By contrast, one line of a Pascal source program might represent many computer instructions.

A line of an assembly language program can contain up to four elements: the *label*, the *mnemonic*, one or two *operands*, and a comment.

```
Label      Mnemonic   Operand     Comment
START:       CALL       FIRST   ;initialize data
```

The label, which is usually terminated by a colon, is used to transfer control from one portion of the source program to another. The mnemonic represents the desired CPU instruction. The operand might reference a CPU register, a memory location, or simply a constant. Finally, a comment, preceded by a semicolon, can be used to explain the instruction. The comment, of course, is ignored by the assembler.

The remainder of this chapter is devoted to a general discussion of some of the features of the 8080 and Z-80 CPUs. The complete instruction sets

for these CPUs are listed in the appendix. Specific details of each instruction are given in Appendix H. If you are already familiar with these instruction sets, then you might want to go on to the next chapter.

THE 8080 CPU

The 8080 CPU is an integrated circuit that has 40 pins (legs). It requires three power supply voltages—12 V, 5 V, and −5 V, and a two-phase clock that runs at 2 Megahertz (MHz). There is an accumulator, a flag register, six general-purpose registers, a stack pointer, and a program counter. The accumulator is sometimes known as register A. The flag register is usually called the PSW (the letters being an acronym for *program status word*). The general-purpose registers are designated by the letters B, C, D, E, H, and L. Sometimes the registers are paired into 16-bit double registers known as BC, DE, and HL. The accumulator and flag register may also be paired. There are 78 different instruction types that produce a total of 245 different op codes.

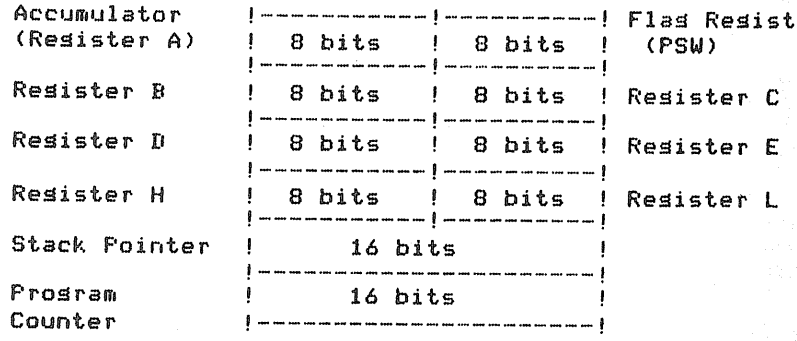


Figure 1.1. The 8080 CPU registers.

Some of the 8080 instructions explicitly refer to the accumulator or to one of the general-purpose registers (B, C, D, E, H, and L).

```
mnemonic   operand     comment
  INR        A        ;increment accumulator
  DCR        B        ;decrement register B
  MOV        H,D      ;move contents of D to H
  MVI        C,4      ;put value of 4 into C
```

When there are two operands, data moves from the right operand (the *source*) into the left operand (the *destination*). There are additional 8080 commands that implicitly refer to the accumulator.

```
mnemonic   operand       comment
  RAR                  ; rotate accumulator right
  RAL                  ; rotate accumulator left
  IN          0        ; input a byte to A from port 0
  OUT         1        ; output a byte from A to port 1
  ANI         7        ; logical AND with A and 7
  ORI         3        ; logical OR with A and 3
```

For certain 8-bit operations, the accumulator is implicitly one of the source registers and will contain the result of the operation.

```
mnemonic operand      comment
  ADD      C        ; add C to A
  SUB      D        ; subtract D from A
  ANA      H        ; logical AND of A with H
  ORA      B        ; logical OR of A and B
```

Other instructions refer to coupled pairs of 8-bit registers. These extended operations treat the BC, the DE, and the HL register pairs as 16-bit entities. Sometimes the stack pointer and program counter are included in these instructions. The X symbol in the mnemonic refers to these extended 16-bit operations.

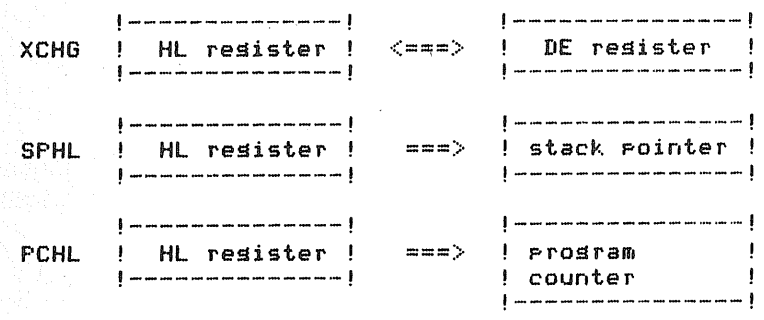
```
mnemonic  operand      comment
  INX       H        ; increment HL register pair
  DCX       SP       ; decrement stack pointer
  LXI       D,0      ; load zero into DE pair
```

Additional instructions deal specifically with the HL register pair. The following two instructions move two bytes of data between memory and the HL double register.

```
mnemonic  operand   comment
  LHLD      3     ; addr 3, 4 to L and H
  SHLD      3     ; L,H to addr 3, 4
```

The LHLD instruction copies the value at memory location 3 into the L register and the value at location 4 into the H register. The SHLD operation reverses the process.

The XCHG operation interchanges the 16-bit HL register pair with the 16-bit DE register pair.



The SPHL command copies the HL register into the stack pointer register. The PCHL instruction copies the HL register pair into the program counter register.

There are several instructions that perform double-register addition. The number in one of the 16-bit registers is added to the number in HL. The sum appears in the HL register pair.

```
DAD     D       ;add DE to HL
DAD     SP      ;stack pointer + HL
```

THE MEMORY REGISTER

There is another 8-bit register for the 8080 that is not shown in Figure 1.1. It is located in main memory. The 16-bit address contained in HL defines the location of this memory register, i.e., HL is a memory pointer. The instruction

```
MOV     M,E     ;move E to memory
```

will copy the contents of register E into the memory location pointed to by the HL register pair. The instruction

```
INR     M       ;increment memory
```

will increment this byte in memory.

THE FLAG REGISTER

Four bits of the PSW register can be used to control program flow. The bits or *flags* are used in conjunction with conditional jump, conditional call, and conditional return instructions. We say that a flag is *set* if it has a value of 1 or is *reset* if it has a value of zero.

The CPU sets the sign flag (S) if the result of a previous operation is positive; the flag is reset if the result is negative. The CPU sets a second flag, the zero flag (Z), if the result is zero; it is reset if not zero. A third flag, the carry flag (C), is set if there is a carry on addition or borrow on subtraction; it is reset otherwise. A fourth flag, the parity flag (P), indicates the parity of the result. Parity is even if there is an even number of ones (or zeros) and odd otherwise.

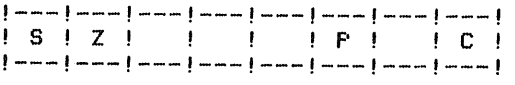


Figure 1.2. The PSW (flag) register.

The use of the flag register can be demonstrated with a simple routine. Suppose that a group of instructions is to be executed eight times. The following code will do this.

```
        MVI     B,8     ;Put 8 into B
LOOP:   . . .
        . . .
        DCR     B       ;decrement B
        JNZ     LOOP    ;loop if not zero
```

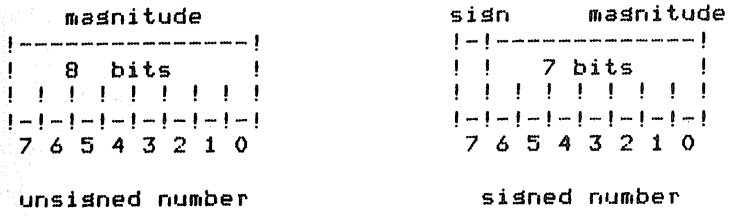
The B register is initialized to the value of 8. The DCR B instruction near the end of the loop decrements the B register each time the loop is executed. This will reset the zero flag on each of the first seven passes through the loop since B has not reached zero. The following conditional jump instruction, JNZ LOOP, causes the CPU to return to the line labeled LOOP in this case.

On the eighth pass through the loop the original value of 8 in the B register will have been decremented to zero. Now the zero flag will be set and the conditional jump instruction will not cause a branch. The instruction immediately following the jump will be executed instead.

FLAGS AND ARITHMETIC OPERATIONS

The results of addition and subtraction operations can be characterized from the PSW flags. Three of the flags are of interest here: the carry flag, the zero flag, and the sign flag. If the sum of two numbers exceeds 255 (1 less than 2 to the eighth power), then the result is too large to fit into the 8-bit accumulator. The carry flag will be set to reflect this overflow. During subtraction, the carry flag is set when a larger number is subtracted from a smaller one. In this case, the flag becomes an indication that borrowing has taken place.

Sometimes all eight bits of a register or memory location are used to represent a number. This is then an *unsigned* number. At other times it is convenient to utilize only the low-order seven bits (bits 0-6) for the magnitude of a number. The remaining high-order bit (bit 7) is then used to indicate the sign.



Numbers represented in this way are known as *signed* numbers. A 0 in bit 7 means that the number is positive and a 1 means that the number is negative. An 8-bit signed number can range in magnitude from −128 to 127, whereas an unsigned 8-bit number can range from 0 to 255.

The sign flag is set after certain operations if the value of bit 7 is 1 and it is reset if bit 7 is 0. If the sum of two numbers is exactly 256, the result in the 8-bit accumulator will be a zero. This occurs because 256 is 1 greater than the largest 8-bit number (255). The zero flag will be set in this case. In addition, the carry flag will be set because there is an overflow. The parity

flag will be set, since there is an even number of ones. (Zero is an even number.) Finally, the sign flag will be reset because bit 7 is a zero.

FLAGS AND LOGICAL OPERATIONS

In the case of arithmetic operations such as addition and subtraction, there can be a carry or borrow from one bit to another. But the logical operations AND, OR, and XOR (exclusive OR) operate on each bit separately; there is never a carry from one bit to the next. These logical operations, therefore, always reset the carry flag. The zero and parity flags, however, will be set or reset according to the result of the particular operation. We will discuss logical operations more fully in Chapter 2.

A value in the accumulator can be compared to a value in another register or to the byte immediately following the instruction byte in memory. The CPU performs the comparison by subtracting the value of the operand from the value in the accumulator. In the case of a regular subtraction, the difference is placed in the accumulator. For example, the arithmetic instruction

```
SUB     C
```

subtracts the value in register C from the accumulator and places the difference into the accumulator. The logical comparison operation

```
CMP     C
```

also subtracts the value in register C from the value in the accumulator. However, unlike the regular subtraction operation, the difference in this case is not actually saved. The flags, of course, will reflect the result of the operation. If the value in C is equal to the value in the accumulator the difference between them will be zero. In this case the zero flag is set indicating the equality. The carry flag will be reset since there was no borrow during the subtraction.

If the two values are not equal, then A is either larger or smaller. If A is larger, the comparison operation will reset the carry flag (and, of course, the zero flag). If A is smaller, then the carry flag will be set, because a larger number has been subtracted from a smaller one. Thus, if the carry flag has been set after a comparison, then the value originally in the accumulator must have been smaller than the value with which it was compared.

The following instructions can be used to determine if the value in register C is less than, greater than, or equal to the value in the accumulator.

```
CMP     C       ; subtract A from C
JZ      ZERO    ; if A equals C
JC      LESS    ; if A less than C
. . .           ; if A greater than C
```

The comparison instruction is executed first. This operation subtracts the value of C from the value in the accumulator. If the two numbers are equal, then their difference is zero. In this case, the zero flag is set and the JZ instruction causes a branch to the label ZERO. Otherwise, the next instruction is executed. Another possibility is that the value in C is greater than the value in the accumulator. The subtraction in this case requires a borrow so the carry (borrow) flag is set. The JC instruction then causes a branch to the label LESS. The last possibility is that the value in the accumulator is larger than that in register C. For this case, both the zero and the carry flags are reset, and the program continues.

INCREMENT, DECREMENT, AND ROTATE INSTRUCTIONS

The 8-bit increment and decrement instructions present an interesting case. Mathematically, the increment operation simply adds 1 to the current value in a register. Likewise, the decrement operation subtracts 1 from the present value. Thus, the two instructions

```
INR     A       and
ADI     1
```

both increase the value in the accumulator by 1 and the operations

```
DCR     A       and
SUI     1
```

both decrease the value in the accumulator by 1. The zero, parity, and sign flags correctly reflect the result in all cases.

The carry flag, however, responds differently for the two cases. The flag correctly reflects the result of the operation in the case of addition, but it is unaffected in the case of an increment or decrement operation. Thus, if you need to increment or decrement a value without disturbing the carry flag, then you should use the INR or DCR instructions. On the other hand, if you need to know whether a carry or borrow occurred during an increment or decrement, then use an add or subtract operation.

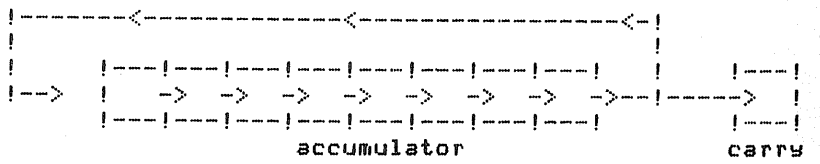
The instructions following the label GETCH in Listing 6.1 (in Chapter 6) are used to set ASCII characters from the console input buffer. As each character is obtained, the count of the remaining characters is decremented. When the count has been decremented past 0, then the routine is finished. Subtracting 1 from 0 requires a borrow so the carry flag should be set. But since the regular decrement operation doesn't alter the carry flag, the subtract instruction must be used instead.

ROTATION OF BITS IN THE ACCUMULATOR

There are four 8080 instructions that rotate the bits in the accumulator. The operations move each bit by one position. Two instructions rotate the bits to the right and two rotate them to the left. The right circular rotation

```
RRC
```

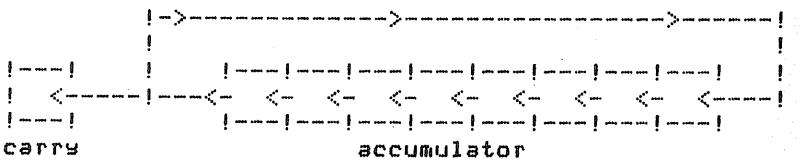
moves each bit one position to the right. The rightmost (low-order) bit is moved to the high-order bit and into the carry flag.



The left circular rotation

```
RLC
```

moves the bits the other way.

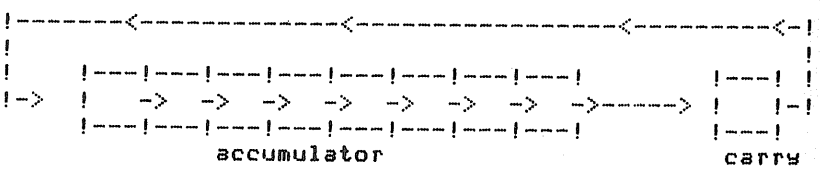


Each bit is moved one position to the left. The high-order bit goes to both the low-order bit and to the carry flag.

The rotate accumulator right instruction

```
RAR
```

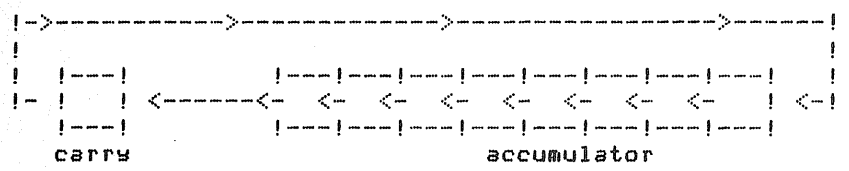
moves each bit one position to the right. But this time, the carry flag moves into the high-order bit and the low-order bit moves into the carry flag.



The instruction

```
RAL
```

moves each bit one position to the left. The carry flag moves into the low-order bit and the high-order bit moves into the carry flag.



FLAGS AND DOUBLE-REGISTER OPERATIONS

Double-register, or extended, operations involving HL, DE, and BC affect the flags very differently from the single-register operations. We saw that single-register increment and decrement operations did not alter the carry flag. The extended increment and decrement commands never alter any of the flags. This means that if a program is to loop until a double register has been decremented to zero, the following set of instructions will not work.

```
LOOP:   . . .
        . . .
        DCX     H       ;16-bit decrement
        JNZ     LOOP    ;if not zero
```

The proper procedure is to compare the two 8-bit halves with each other. This can be done by moving one of the registers to the accumulator.

```
MOV     A,L
```

Then the accumulator is compared to the other half by performing a logical OR. The result of this operation will set the zero flag only if both halves are zero. The complete operation looks like this.

```
LOOP:   . . .
        . . .
        DCX     H       ;16-bit decrement
        MOV     A,L     ;move L to A
        ORA     H       ;OR with H
        JNZ     LOOP    ;if not zero
```

The double-register add instruction correctly sets the carry flag if there is an overflow from the 16 bits, but zero, parity, and sign flags are not altered.

THE Z-80 CPU

The Z-80 CPU is a 40-pin IC just like the 8080. All of the 8080 instructions are common to the Z-80, thus we say the Z-80 is upward compatible from the 8080. In general, any program that runs on an 8080 will also run on a

Z-80. The one exception is that the 8080 parity flag is affected by arithmetic operations, while the Z-80 parity flag is not. Thus, one can use an 8080 assembler to generate 8080 code on a Z-80.

The Z-80 requires only a single 5-volt power supply and a single-phase clock that can run as fast as 6 MHz. There are 158 instruction types that give a very large number of total commands with all variations. These are given briefly in Appendixes E and F and in more detail in Appendix H. The Z-80 contains all of the 8080 general-purpose registers, plus an alternate set for easy interrupt processing. The alternate set is indicated with a prime symbol: A′, B′, and so on. Only one of the two general sets of registers can be used at any time, therefore, data cannot be transferred directly from one set to the other. There are also two 16-bit index registers called IX and IY, an 8-bit interrupt register (I), and an 8-bit refresh register (R).

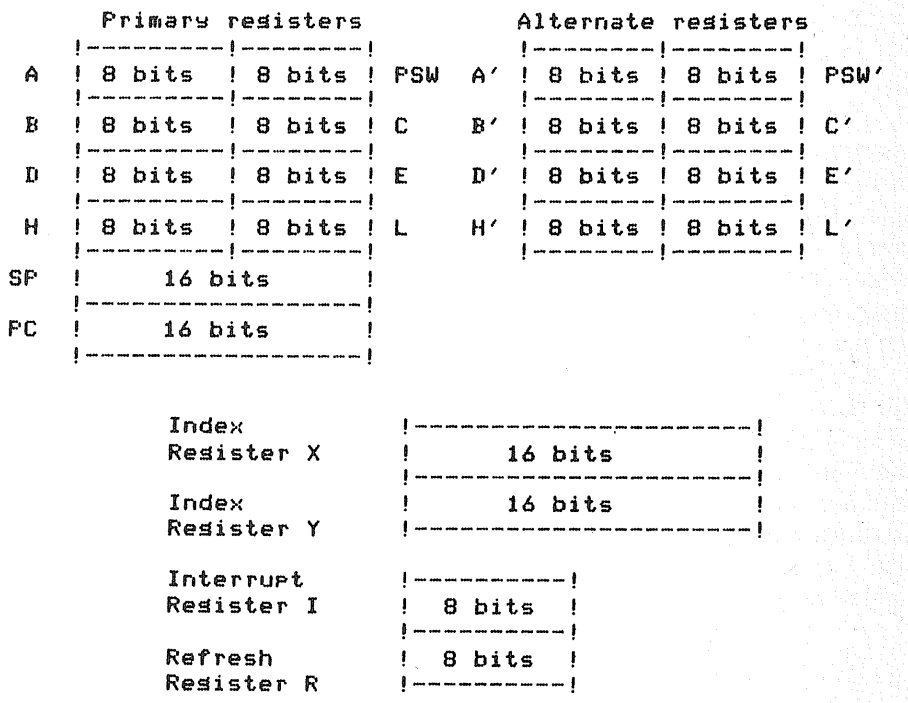


Figure 1.3. The Z-80 CPU registers.

An operand for an assembly-language instruction may consist of a value that is used directly, or it may refer to a location that contains the value. For example, the command

```
LD      A,6
```

instructs the CPU to place the value of 6 into register A. Similarly, the instruction

```
LD      A,D
```

will move the contents of register D into register A. Alternately, the operand may be a pointer to another location. Thus the command

```
LD      A,(6)
```

will move the byte located at address 6 into the accumulator. Similarly, the instruction

```
LD      A,(HL)
```

tells the CPU to move the byte pointed to by the HL register into the accumulator. The Z-80 mnemonics clearly differentiate a pointer by means of the parentheses, whereas the corresponding 8080 mnemonics do not make such a clear distinction.

Z-80 RELATIVE JUMPS

Computer instructions are generally executed in order, one after the other. But it is sometimes necessary to branch out of the normal sequence of statements. Branching statements can be classified as either *conditional* or *unconditional.* An unconditional or absolute branch always causes the computer to execute instructions at a new location, out of the normal flow. Conditional branching, on the other hand, is based upon the condition of one of the flags.

Programs utilizing the Z-80 instruction set can be significantly shorter than those written with 8080 operation codes, especially if the relative jump instructions are used. Relative jumps are performed by branching forward or backward relative to the present position. Absolute jumps, on the other hand, are made to a specific memory location. Furthermore, there are both unconditional and conditional branch instructions. The absolute, unconditional jump op code and the conditional jump codes based on the state of the zero and parity flags are all three-byte instructions.

```
JP      ADDR1       ; unconditional jump
JP      Z,ADDR2     ; jump if zero flag set
JP      NZ,ADDR3    ; jump if zero flag reset
JP      C,ADDR4     ; jump if carry flag set
JP      NC,ADDR5    ; jump if carry flag reset
```

The above instructions are available on both the 8080 and the Z-80 CPUs. In addition, the Z-80 has a relative, unconditional jump and five relative, conditional jumps.

```
JR      ADDR        ; unconditional jump
JR      Z,ADDR6     ; zero
JR      NZ,ADDR7    ; not zero
JR      C,ADDR8     ; carry
JR      NC,ADDR9    ; not carry
DJNZ    ADDR10      ; decr, jump not zero
```

The relative jumps are only two bytes long as opposed to three bytes for the regular jumps, but the relative jump is limited to a displacement of less than 126 bytes forward or 128 bytes backward from the address of the current instruction. These numbers derive from the magnitude of the signed 8-bit displacement. Bit 7 is used for the sign of the number. A 0 in bit position 7 means a forward or positive displacement, a 1 in this bit position means a backward or negative displacement. The remaining seven bits are used for the magnitude of the jump.

Absolute jumps are specified with a 16-bit address that gives the new location. Relative jumps on the other hand are position-independent. The resulting code can be placed anywhere in memory. The last operation above, DJNZ, is a combination of two instructions. The B register is decremented. If the result is not zero, then there is a relative jump to the given argument ADDR10. This two-byte instruction requires four bytes on an 8080 CPU.

Z-80 DOUBLE-REGISTER OPERATIONS

While some of the Z-80 instructions appear to be shorter than their 8080 counterparts, they may not actually reduce the program size. Suppose, for example, that we want to move a block of data from one memory location to another. There is a single Z-80 instruction for accomplishing this task. The problem is that no verification is performed during the move. Thus, if there were no memory at the new location, or if the memory were defective, this fact would not immediately be discovered. If you want to check each location as the data are moved, then the Z-80 block-move instruction cannot be used.

A better way to move data is to define the beginning of the original memory block with HL and the end with DE. The BC register defines the beginning of the new block. We can work our way through the original block by incrementing HL and BC at each step along the way.

The end of the block can be detected when HL exceeds DE. We subtract the two 16-bit registers and observe the carry flag. The HL register pair will initially be less than the DE pair. Therefore, if we subtract DE from HL, we will set the carry (borrow) flag.

Eventually, the number in the HL register will equal the value in the DE pair. This time, the subtraction will not set the carry flag and the task will be completed. Since the 8080 doesn't have a 16-bit subtract instruction, the routine might look like this.

```
LOOP:   . . .               ; 8080 version
        . . .
        MOVE    A,L         ; GET L
        SUB     E           ; SUBTRACT E
        MOV     A,H         ; GET H
        SBB     D           ; SUBTRACT D AND BORROW
        JC      LOOP        ; IF NOT DONE
        RET                 ; DONE
```

As long as HL is less than DE, the subtraction will set the carry flag and the loop will be repeated. But as soon as HL equals DE, the carry flag will be reset and the subroutine is finished.

The Z-80 has a 16-bit subtract instruction that can simplify the operation. But since the result of the subtraction is placed in the HL register pair rather than in the accumulator, the data originally present in HL will have to be saved somewhere else, say, on the stack. The Z-80 code is:

```
LOOP:  . . .              ; Z-80 version
       OR       A         ; RESET CARRY
       PUSH     HL        ; SAVE HL ON STACK
       SBC      HL,DE     ; SUBTRACT DE FROM HL
       POP      HL        ; RESTORE  ORIGINAL HL PAIR
       JR       C,LOOP    ; IF NOT DONE
       RET
```

The necessary Z-80 instructions require just as many bytes as the corresponding 8080 code. And if the carry flag on the Z-80 has not been reset by a previous instruction, it will have to be reset at the beginning with a logical OR instruction. This latter problem occurs because the Z-80 16-bit subtraction includes the carry flag in its calculations.

Z-80 INPUT AND OUTPUT (I/O) INSTRUCTIONS

A useful pair of Z-80 instructions deals with input and output, i.e., the transfer of data between the CPU and peripherals such as the console, the printer, and the disk. The 8080 can only input and output data from the accumulator, and the address of the peripheral device must immediately follow the IN or OUT instruction in memory.

```
OUT      10
IN       11
```

This usually means that for *read-only memory* (ROM), there must be separate input and output routines for each peripheral.

In contrast, the Z-80 can input or output a byte from any of the general-purpose registers when the peripheral address is in the C register. In this case, it may be possible to use a single set of I/O routines for all peripherals. This approach is discussed more fully in Chapter 4.

SHIFTING BITS

The Z-80 CPU extends the four 8080 rotate instructions to the general-purpose registers B, C, D, E, H, and L. The memory byte referenced by HL, IX, and IY is also included.

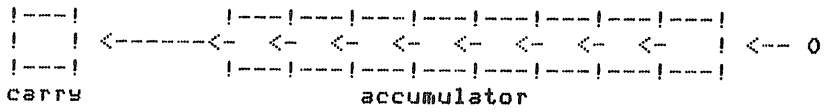
The Z-80 instruction set includes three shift operations. Shifts are similar to rotations since each bit moves one position and the bit that is

shifted out of the register is moved into the carry flag. The difference is in the bit that is shifted into the register.

The arithmetic shift left

```
SLA
```

shifts all bits to the left. A zero bit moves into the low-order bit.



The operation doubles the original 8-bit value. This operation can be performed on the accumulator of an 8080 by using an

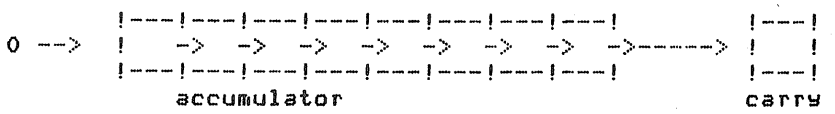
```
ADD   A
```

instruction.

A logical shift right

```
SRL
```

is the inverse of the arithmetic shift left operation. Each bit shifts one position to the right. A zero bit is shifted into the high-order position.

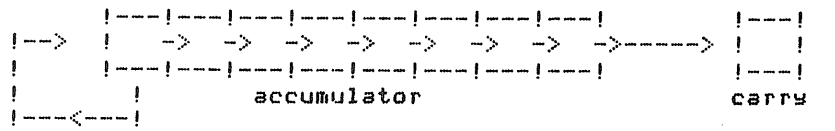


This operation halves the original 8-bit value. The carry flag is set if the original value was odd, that is, if there is a remainder from the division.

The arithmetic shift right

```
SRA
```

shifts each bit one position to the right, but the original high-order bit is unchanged.

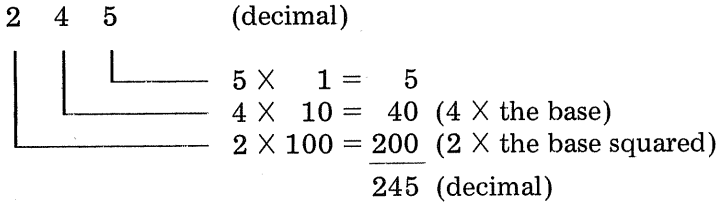


This operation can be used to divide a signed number in half. The high-order bit, the sign bit, is unchanged. As with the logical shift right, the carry flag is set if the original number was odd.

CHAPTER TWO

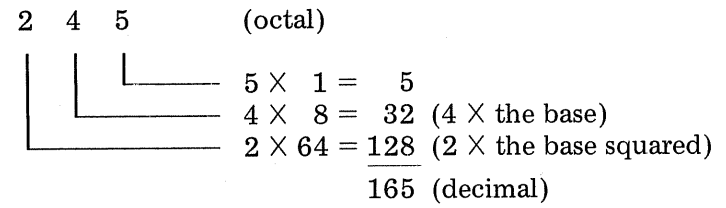
Number Bases and Logical Operations

In this chapter we will consider how numbers are stored in a computer. We will also look at some of the operations that can be performed on these numbers. But first we will review the representation of numbers in general. When we write a number such as 245, we usually mean the quantity 5 plus 40 plus 200.



This is the ordinary decimal or base-10 representation of a number. The rightmost digit gives the number of units. The digit immediately to the left is the number of tens (the base). The next digit to the left is the number of 100s (the base squared).

In assembly language programs it is sometimes convenient to represent numbers with a base of 2, 8, or 16. In the octal, or base-8, system, for example, the number 245 is equivalent to the decimal number 165 (5 plus 32 plus 128).

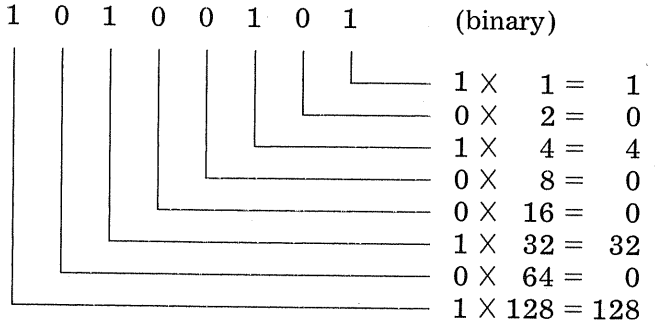


This example demonstrates how to convert numbers from other bases into the decimal representation by adding up the decimal equivalent of each digit.

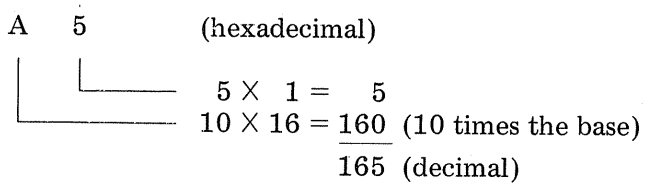
In the binary, or base-2, system, only the digits 0 and 1 are used. The individual digits are called *bits*, an acronym for binary digits. The rightmost bit represents the units. The bit immediately to the left is the number of 2s (the base). The next bit to the left is the number of 4s (the base squared). We continue in this way through all of the bits. For example, the binary number

10100101

is equivalent to the decimal number 165. The conversion is obtained in the following way.



We have seen that the decimal system utilizes ten different digits (0-9). The octal system, however, utilizes only eight digits (0-7), and the binary system uses only two (0-1). The hexadecimal, or base-16, system is also commonly used in computer programs. With this method, we need 16 different digits. The problem is that if we use all of the digits (0-9) from the decimal system, we will still be six digits short. The solution is to use the letters A through F to represent the digits beyond 9. Thus, the hexadecimal number A5 is equivalent to the decimal number 165. We can convert a hexadecimal number into decimal in the usual way if we remember that A stands for decimal 10, B for 11, and so on.



The first 16 integers of the decimal, binary, octal, and hexadecimal systems are shown in Table 2.1.

Table 2.2. The first 16 integers represented in various number systems.

decimal	binary	octal	hex
0	0000	000	00
1	0001	001	01
2	0010	002	02
3	0011	003	03
4	0100	004	04
5	0101	005	05
6	0110	006	06
7	0111	007	07
8	1000	010	08
9	1001	011	09
10	1010	012	0A
11	1011	013	0B
12	1100	014	0C
13	1101	015	0D
14	1110	016	0E
15	1111	017	0F

Table 2.1 shows the common practice of displaying leading zeros on numbers expressed in bases other than 10. Thus we write 5 for a decimal number, but we may write 005 if it is an octal number or 05 if it is a hexadecimal number. We may explicitly represent the base by a suffix. In books, for example, we typically utilize a subscript in smaller size type. Thus we will write:

1010_2 (binary)
17_8 (octal)
17_{10} (decimal)
17_{16} (hexadecimal)

Alternately, we use suffixes of B, Q, D, and H to designate, respectively, binary, octal, decimal, or hexadecimal mode in computer programs where subscripts are not available.

1010B (binary)
17Q (octal)
17D (decimal)
17H (hexadecimal)

(The letter Q is used instead of an O for an octal number to avoid confusion with zero.)

Binary numbers such as

011001101111

can be difficult to read, so it is common practice to represent them in octal or hexadecimal form. Conversion to octal is easy if the bits are grouped by threes.

```
011   001   101   111     (binary)
 3     1     5     7      (octal)
```

Grouping by fours facilitates the conversion to hexadecimal.

```
0110  0110  1111    (binary)
 6     6     F      (hexadecimal)
```

NUMBER REPRESENTATION IN BINARY, BCD, AND ASCII

All information is ultimately stored in computers as a series of binary digits. There are, however, several different coding schemes for representing the original data. The simplest method is to use straight binary coding, as shown in Table 2.1. Notice that we might choose to represent a binary value in decimal, octal, or hexadecimal notation. The number itself is unchanged by this. The decimal number 12, for example, is stored as the binary number 1100.

A different method of representing data is called *binary coded decimal* (BCD). Actually, there are two types of BCD: unpacked and packed. With unpacked BCD, each byte contains a single decimal digit from 0 to 9. Packed BCD can have one or two decimal digits in each byte. Thus, a packed BCD number can range from 0 to 99. By comparison, an 8-bit binary number can range from 0 to 255. Table 2.2 shows the first 16 integers in BCD. The first column gives the decimal equivalent, the second column the corresponding bit pattern.

Table 2.2. The first 16 integers represented in decimal and binary-coded decimal (BCD).

decimal	BCD
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100
5	0000 0101
6	0000 0110
7	0000 0111
8	0000 1000
9	0000 1001
10	0001 0000
11	0001 0001
12	0001 0010
13	0001 0011
14	0001 0100
15	0001 0101

Notice that the binary representation for the decimal numbers 0 through 9 is the same for both binary and for BCD.

A third method for encoding data is called ASCII. This scheme is commonly used with peripherals such as printers and video terminals. When the key labeled 2 of an ASCII console is pressed, the bit pattern

0011 0010

is generated. Table 2.3 gives the bit patterns for the ASCII digits 0-9 in binary and hexadecimal notation.

Table 2.3. The bit pattern for the ASCII digits 0-9.

digit	binary	hexadecimal
0	0011 0000	30
1	0011 0001	31
2	0011 0010	32
3	0011 0011	33
4	0011 0100	34
5	0011 0101	35
6	0011 0110	36
7	0011 0111	37
8	0011 1000	38
9	0011 1001	39

LOGICAL OPERATIONS

The fundamental operations of a computer involve electrical signals that can have only one of two values. The two voltage levels might be zero and 5 volts, for example, or they might be something else. The actual value is unimportant at this point. Instead, we refer to the two allowable states as TRUE and FALSE. The TRUE state is also called a logical 1, or high state, and the FALSE state is also known as a logical 0, or low state.

TRUE = 1 (high)
FALSE = 0 (low)

Computers store numbers in binary form as a series of 1s and 0s. These two possible values correspond to the two possible voltage levels of the electronic circuitry. We can therefore utilize the expressions TRUE and FALSE to describe the state of each bit.

The collection of transistors, resistors, and so forth that makes up the physical computer is called the *hardware*. The computer program used to direct the activities of the computer is termed the *software*. In this sense, the hardware and software are distinctly different. But sometimes we use these terms a little differently.

Consider, for example, one of the major differences between minicomputers and microcomputers. Minicomputers contain electronic circuitry for the multiplication of two numbers. Since microcomputers do not contain such circuitry, multiplication is performed instead by executing a special computer program. We say that minicomputers perform multiplication by hardware, but that microcomputers must do multiplication by software.

Hardware operations are performed by electronic devices called *gates*. The internal structure of the gate is unimportant if we are only interested in the logic of its operation. There are input signal lines that are sampled by the gate, and there is an output signal that is generated by the gate. When we consider the logical operations that are performed by a computer, we can imagine that they are accomplished either by hardware or by software. The answer is the same.

A common logical operation is the complement or inversion of a binary digit. The complement of 0 is 1 and the complement of 1 is 0. The hardware complement is performed with an inverter or NOT gate. The electronic symbol for this gate, shown in Figure 2.1, is a triangle with one apex to the right (usually) or to the left (sometimes). A small circle or triangle at this apex completes the symbol.

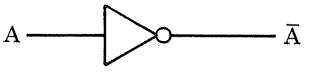


Figure 2.1. The electronic symbol for the NOT or inverter gate.

Letters of the alphabet are used to designate input or output signals. These binary signals can have one of two states, termed TRUE (1) or FALSE (0). The letter A with a bar over it ($\bar{A}$) represents the complement of A and is called NOT A. A truth table is used to summarize the possible states.

A	$\bar{A}$	or	A	$\bar{A}$
0	1		FALSE	TRUE
1	0		TRUE	FALSE

THE TWO'S COMPLEMENT

If each bit of an 8-bit byte is complemented, we produce a result that is termed the one's complement of the byte.

```
0000 1001 = 9
1111 0110 = one's complement of 9
```

Both the 8080 and the Z-80 CPUs provide an operation code for complementing the accumulator. A slightly different operation is the two's complement. It is obtained by incrementing (adding 1 to) the one's complement of a number. For example:

```
  0000 1001 = 9

  1111 0110 = one's complement of 9
+ 0000 0001   add one
-----------
  1111 0111 = two's complement of 9
```

It is interesting to note that the sum of a number and its two's complement is zero.

```
  1010 1010 = 170

  0101 0101 = one's complement of 170
+ 0000 0001   add one
-----------
  0101 0110 = two's complement of 170

  0101 0110 = two's complement of 170
+ 1010 1010 = 170
-----------
  0000 0000   sum
```

Adding the two's complement of a number produces the same result as subtracting the number itself. For example, we can subtract 170 from 223 by adding the two's complement of 170. The result is the same.

```
  1101 1111 = 223
- 1010 1010 = 170
-----------
  0011 0101 = 53
```

or

```
  1101 1111 = 223
+ 0101 0110 = 2's complement of 170
-----------
  0011 0101 = 53
```

The 8080 CPU can perform both addition and subtraction with 8-bit numbers and it can add 16-bit numbers, but there is no 16-bit subtraction operation. We can effectively perform a 16-bit subtraction, however, by adding the two's complement. Suppose that the HL register pair contains the decimal value 10,005 and we want to subtract 10,000 from it. The difference between 10,005 and 10,000 can be obtained by adding the two's complement. Consider the bit pattern for the number 10,000.

```
0010 0111 0001 0000 = 10,000
```

We first form the one's complement, then increment the result to form the two's complement.

```
  1101 1000 1110 1111  =  one's complement of 10,000
+ 0000 0000 0000 0001     add one
-------------------
  1101 1000 1111 0000  =  two's complement of 10,000
```

Finally, we add this two's complement to the value in HL.

```
  0010 0111 0001 0101  =  10,005 in HL
+ 1101 1000 1111 0000  =  two's complement of 10,000
-------------------
  0000 0000 0000 0101     difference (sum) is 5
```

When an assembler encounters a negative argument, it will automatically calculate the corresponding two's complement. Thus the 8080 expression

```
LXI  =  D,-10000
```

will place the bit pattern

```
1101 1000 1111 0000
```

in the DE register pair. The instruction

```
DAD     D
```

will then effectively perform a 16-bit subtraction on the number in HL.

LOGICAL OR AND LOGICAL AND

In the previous section, we considered the logical operation of NOT. Two other important logical operations are OR and AND. Both of these operations reflect the usual English meaning. The logical OR of two bits results in a value of TRUE (1) if either or both the original values are TRUE. The result is FALSE otherwise. The logical AND of two values gives an answer of TRUE (1) if and only if both of the original values are TRUE. If either or both the original values are FALSE, then the answer is FALSE.

Equations of logical operations can be written using the appropriate symbols. Two OR operators are in common use: a plus symbol and a V-shaped symbol. The AND operator is either a dot or an inverted V. The schematic representations of the OR and AND gates are shown with their corresponding mathematical representations in Figure 2.2.

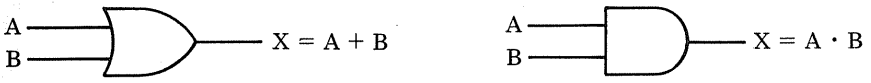


Figure 2.2. The OR and the AND gates.

The truth table is

A	B	(OR) A+B	(AND) A·B
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

where zero means FALSE and 1 means TRUE. The origin of the + symbol for the OR operation and · symbol for the AND operation can be seen from the truth table. Logical operations are performed separately on each bit, and there is never a carry. The logical OR (sum) of A and B gives zero if both bits are zero, but 1 otherwise. (Binary digits can't be larger than 1.) The logical AND (product) of A and B gives zero if either or both bits are zero and unity otherwise.

SETTING A BIT WITH LOGICAL OR

Sometimes, we need to set one or more bits of the accumulator. We can use the logical OR operation for this purpose. From the truth table in the previous section, we can see that a logical OR of 1 with either a 0 or a 1 will give a result of 1.

A	B	A+B
1	0	1
1	1	1

Thus, a logical OR of any bit with a 1 will set that bit. On the other hand, a logical OR of 0 and another bit gives the result of that other bit.

A	B	A+B
0	0	0
0	1	1

In this case, the second bit is not changed.

Suppose that the accumulator contains a binary 5 and we want to convert it to an ASCII 5.

```
0000 0101 = binary 5
0011 0101 = ASCII 5
```

If we compare the two bit patterns, we can see that they are the same except for bits 4 and 5. These bits can be set by executing a logical OR with an ASCII zero.

```
     0000 0101  =  binary 5
OR   0011 0000  =  ASCII zero
     ---------
     0011 0101  =  ASCII 5
```

The OR operation has set the bit corresponding to the location of the 1, but it has left the other bits unchanged.

A logical OR of a register with itself does not change the value.

```
     0101 1010  =  5A hex
OR   0101 1010  =  5A hex
     ---------
     0101 1010  =  5A hex
```

But this operation can be used to set the flags. In this example, the zero, carry, and sign flags are reset and the parity flag is set.

RESETTING A BIT WITH LOGICAL AND

A logical AND operation can be used to reset any particular bit of the accumulator; the truth table shows how. A logical AND of 0 and either a 0 or a 1 will always give a result of 0.

A	B	A·B
0	0	0
0	1	0

Thus, the bit is reset. On the other hand, a logical AND of 1 and another bit will give the value of the other bit.

A	B	A·B
1	0	0
1	1	1

Thus the AND instruction can be used to reset or "turn off" particular bits. This step is sometimes called a *masking* AND operation.

When the CPU reads an ASCII character from the console, it gets an 8-bit byte. But since the ASCII code contains only 7 bits, the high-order bit is not needed. The console-input routine typically resets this bit by performing a masking AND operation. Suppose that the console transmitted an ASCII 5 with the high-order bit set. The bit pattern looks like this.

```
1011 0101
```

The high-order bit can be reset with an AND operation.

```
        1011 0101  (original byte)
AND     0111 1111  (mask)
        ---------
        0011 0101  (ASCII 5)
```

LOGICAL EXCLUSIVE OR

The ordinary OR operation is sometimes called an inclusive-or operation to distinguish it from the exclusive OR (XOR) operation. For this latter operation, the result is TRUE only if the corresponding bits of both values are different. Either A or B must be TRUE, but not both. The XOR operation is represented by a plus symbol surrounded by a circle. The complement of the XOR is the exclusive NOR or XNOR. It can be used as a comparator. The hardware implementation is sometimes used in circuitry to enable memory boards. The result is TRUE if and only if both corresponding bits are identical. The result is FALSE otherwise. The truth table is:

A	B	$A \oplus B$	$\overline{A \oplus B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

The exclusive OR of a bit with itself will always be FALSE. Therefore the XOR of the accumulator with itself will set it to zero.

```
        0111 1100  = 7C hex
XOR     0111 1100  = 7C hex
        ---------
        0000 0000  = zero
```

The corresponding electronic symbols for the hardware implementation of the XOR and XNOR are shown in Figure 2.3.

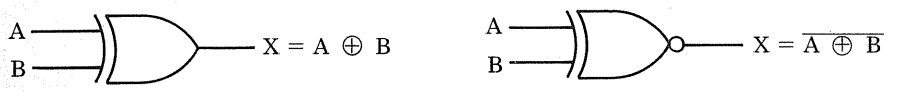


Figure 2.3. The exclusive or (XOR) and comparator (XNOR) gates.

LOGICAL NAND AND NOR GATES

By combining an inverter gate in series with the AND and OR gates, a new set of gates is formed. The NOT AND gate is called a NAND gate; it is shown

in Figure 2.4. The NOT OR gate is known as a NOR gate; it is shown in Figure 2.5.

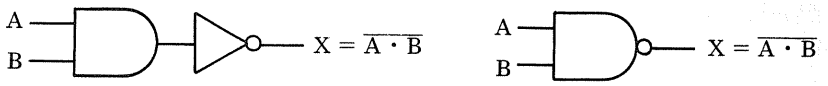


Figure 2.4. The NAND gate can be produced from an AND gate and a NOT gate.

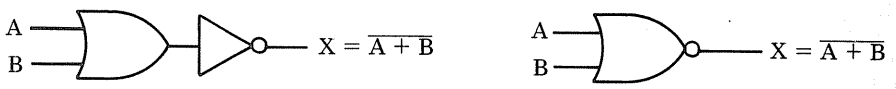


Figure 2.5. The NOR gate can be formed from the OR gate and the NOT gate.

From the truth table, it can be seen that the outputs of the NOR and NAND gates are the inverse of the corresponding OR and AND gates.

A	B	$\overline{A+B}$	$\overline{A \cdot B}$
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0

If both inputs of the NOR gate are connected together, then the gate behaves like a NOT gate. The same is true for the NAND gate. This can be seen by comparing the first and last rows of the truth tables. In this way, two NOR gates can be combined serially to produce an OR gate. The result is a NOT NOT OR gate that is equivalent to an OR gate. This is shown in Figure 2.6. In a similar way, two NAND gates can be used to make an AND gate as shown in Figure 2.7. Since OR and AND gates cannot be similarly combined to produce the NOR and NAND gates, we will find that NAND and NOR gates are more common.

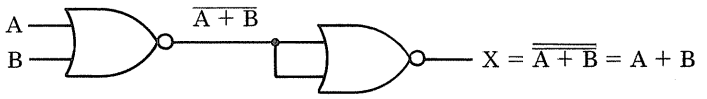


Figure 2.6. An OR gate is formed from two NOR gates.

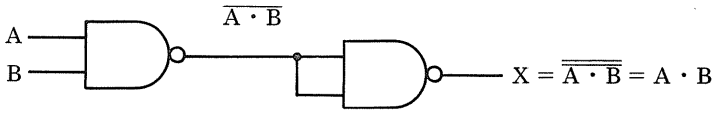


Figure 2.7. Two NAND gates are combined to produce an AND gate.

MAKING OTHER GATES

NOR and NAND gates are very versatile. NOR gates or NAND gates can be combined to produce all of the other gates. This can be seen from the following truth table.

A	B	$\overline{A}$	$\overline{B}$	A+B	A·B	$\overline{A}+\overline{B}$	$\overline{A}\cdot\overline{B}$	$\overline{A+B}$	$\overline{A\cdot B}$
0	0	1	1	0	0	1	1	1	1
0	1	1	0	1	0	1	0	0	1
1	0	0	1	1	0	1	0	0	1
1	1	0	0	1	1	0	0	0	0

Notice that column 7 of the truth table has the same values as the last column. Similarly, columns 8 and 9 are identical. These relations follow De Morgan's theorem, which can be expressed mathematically as:

$$\overline{A} + \overline{B} = \overline{A \cdot B} \quad \text{and}$$
$$\overline{A} \cdot \overline{B} = \overline{A + B}$$

The corresponding digital gates are shown in Figures 2.8 and 2.9.

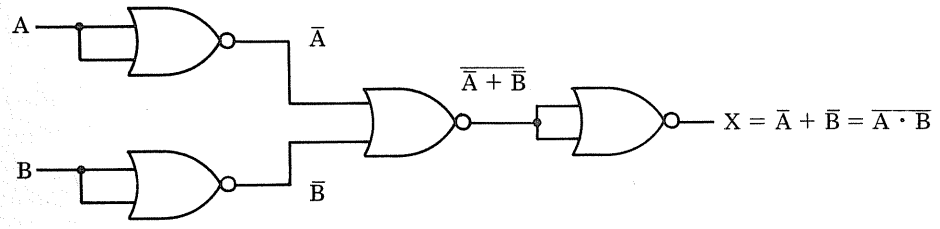


Figure 2.8. A NAND gate is formed from four NOR gates.

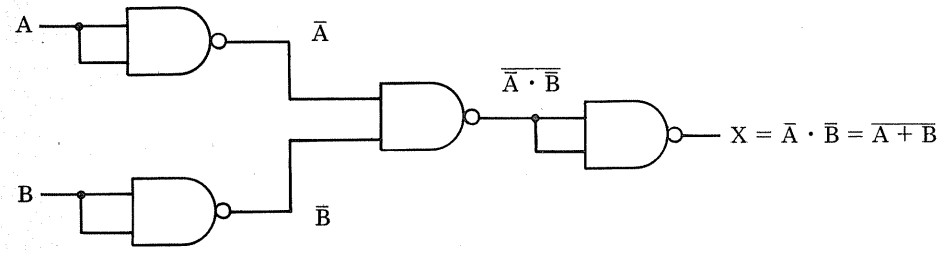


Figure 2.9. A NOR gate is obtained from four NAND gates.

The use of a small circle to represent inverted output brings up another approach to the understanding of digital logic gates. In the more commonly used system, the small circles are used only on the output side of the gate.

Another approach, however, is to always connect active-high outputs to active-high inputs, and active-low outputs to active-low inputs. For this latter system, NAND gates will sometimes appear as OR gates with inverted inputs, and NOR gates will sometimes appear as AND gates with inverted

inputs. According to De Morgan's theorem, the NAND gate is equivalent to the OR gate with inverted input signals. This is demonstrated in Figure 2.10. The circuit shown is logically the same as the one shown in Figure 2.9. Notice that the active-low outputs of the first NAND gates are connected to the active-low inputs of the next OR gate. That is, there are small circles on the outputs of the first gates and on the inputs of the second gate.

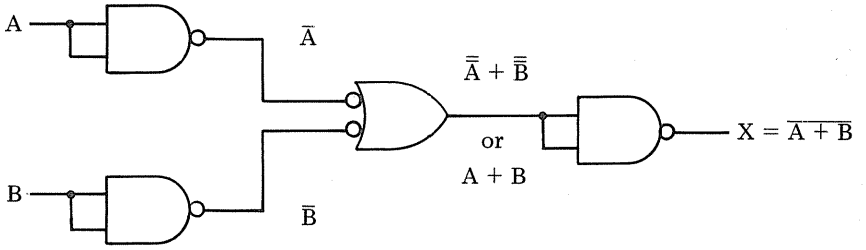


Figure 2.10. A NOR gate is produced from four NAND gates. The middle NAND gate is shown in its alternate representation.

CHAPTER THREE

The Stack

When main memory is used to store a collection of data, each member of the data set is individually accessible. This type of storage is termed *random access memory* (RAM). Magnetic tape storage, by contrast, is serial or *sequential access memory.* In this latter case, only one item of the set is available at any one time. There are two ways of storing and retrieving the items in a serial memory buffer: one is by means of a first-in, first-out (FIFO) buffer, and the other is by means of a last-in, first-out (LIFO) buffer. We can visualize the serial buffer as a long string of information. With the FIFO buffer, items are added at one end and removed from the other. This buffer is analogous to an escalator: the people who ride the escalator are like the data—those who get on first, get off first.

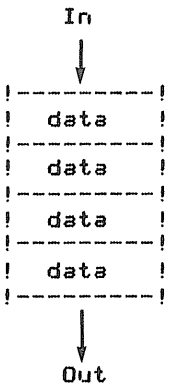


Figure 3.1. The first-in, first-out (FIFO) buffer.

With the LIFO buffer, on the other hand, the data are added and removed at the same place. This arrangement is analogous to a very long, narrow elevator. Those who get on first, have to wait until everyone else is

off before they can get off. It can be seen that magnetic tape is a FIFO medium.

Figure 3.2. The last-in, first-out (LIFO) buffer.

Sometimes, a special area of main memory is designated as a LIFO buffer even though each member of the buffer is individually accessible. This region is known as a *stack*. As an example, Hewlett-Packard calculators utilize a very short LIFO stack, consisting of registers known by the letters Y, Z, and T. An item in any of the registers is individually accessible, yet the stack as a whole can be manipulated. As data is entered from the keyboard, it is placed into the X register. This information can then be transferred to the stack (register Y in this case) by pressing the ENTER key. We say that the contents of the X register are *pushed* onto the stack. Items can be retrieved from the stack and placed in the X register with the roll-down (R) key. We say that data are *popped* from the stack into the X register by this means. Another stack operation is performed by the EXCHANGE key which is used to swap the contents of the X and Y registers.

STORING DATA ON THE STACK

We have seen in the previous chapters that the 8080 and Z-80 microprocessors incorporate general-purpose registers for the storage of information. But these registers are limited in number. Consequently, a special area of main memory is designated for the additional storage of information. This area, called the stack, is implemented on the Z-80 and 8080 as a last-in, first-out serial buffer even though each item in the stack is individually accessible. One of the CPU registers, the stack pointer, references the current location in memory. This is the address of the most recently added item. The stack pointer is decremented as items are added and incremented as items are removed. The programmer may place the stack anywhere in memory by loading the stack pointer with the desired address. For example, the instruction

```
LD      SP,4000H        (Z-80)    or
LXI     SP,4000H        (8080)
```

initializes the stack to location 4000 hex.

Data can be placed on the stack with one of the PUSH operations. A command of

```
(Z-80)              (8080)
PUSH     HL         PUSH     H
```

will move a copy of HL to the stack. Since main memory is addressed eight bits at a time, the PUSH operation is actually performed in two stages. The stack pointer is decremented, then the H register (the high half) is copied to the stack. The stack pointer is decremented a second time and the L register (the low half) is copied to the stack. The stack pointer register now contains the address of the low byte. Figure 3.3 demonstrates the action of a PUSH HL command. The region of memory devoted to the stack is shown with higher memory upward. The arrow represents the stack pointer.

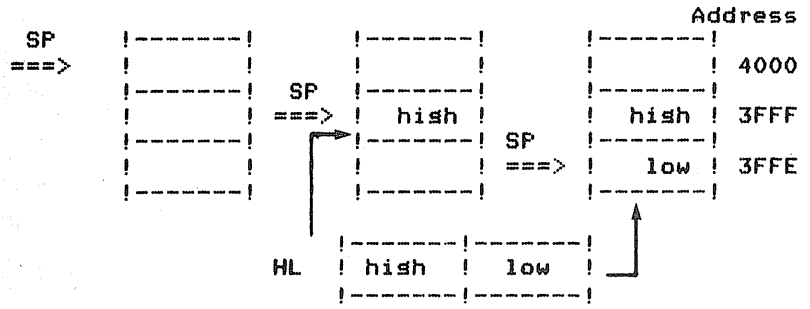


Figure 3.3. The HL register is pushed onto the stack.

The POP instruction reverses the PUSH process. For example, a POP DE command copies 16 bits from the stack into the DE register. Because the stack operates in a LIFQ manner, the most recently added byte is removed first. This is placed into register E (the low half of the DE pair). The stack pointer is automatically incremented and the next byte is transferred from memory to register D (the high half). The stack pointer is then incremented a second time. Figure 3.4 demonstrates the operation. Notice that the data originally pushed onto the stack is still present.

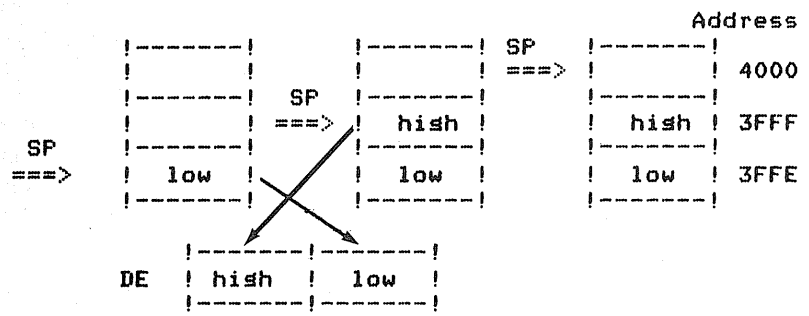


Figure 3.4. Two bytes are popped from the stack into DE.

It can be seen that the stack grows downward in memory as data are pushed into it, and it moves back up as data are popped off. For this reason, it is common practice to initialize the stack pointer to the top of usable memory. Actually, the stack pointer can start at one address above the top of memory since the stack pointer is always decremented before use.

If the general-purpose registers contain important information but they are needed for a calculation, it will be necessary to save the original data. This can be easily done by pushing the contents onto the stack. The registers are restored at the end of the calculation with the three corresponding POP commands. The operation goes like this.

```
PUSH    HL      ;save HL
PUSH    DE      ;save DE
PUSH    BC      ;save BC
 .  .  .
 .  .  .        ;do the calculation
 .  .  .
POP     BC      ;restore BC
POP     DE      ;restore DE
POP     HL      ;restore HL
```

Notice that the order of the POP commands is reversed from that of the PUSH sequence. This is necessary because of the stack's LIFO operation.

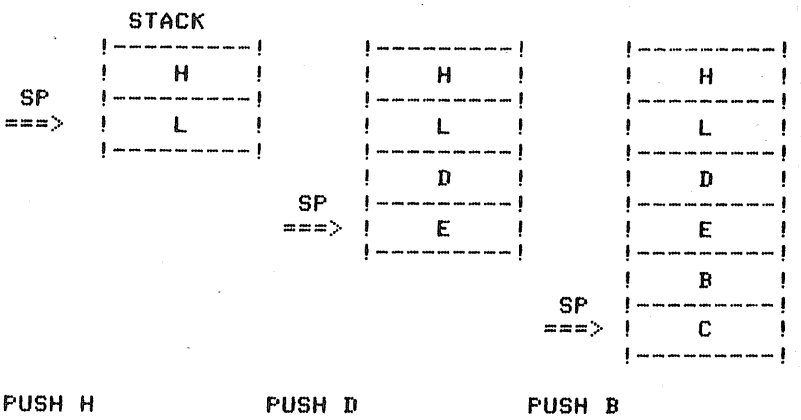


Figure 3.5. The contents of the general-purpose registers are saved on the stack.

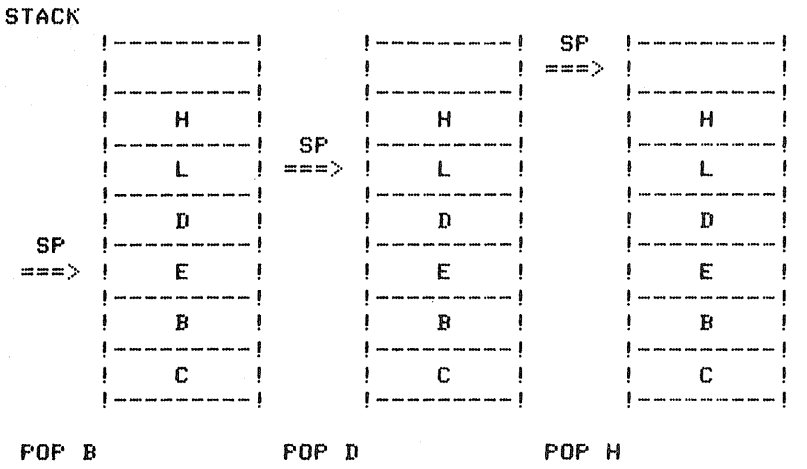


Figure 3.6. The original contents of the general-purpose registers are restored from the stack.

THE ACCUMULATOR AND PSW AS A DOUBLE REGISTER

The 8-bit accumulator and the 8-bit flag register are treated as a 16-bit double register for the PUSH AF and POP AF instructions. In this case, the accumulator is treated like the high byte since it is pushed onto the stack first. The flag register is pushed onto the stack second. Figure 3.7 demonstrates this.

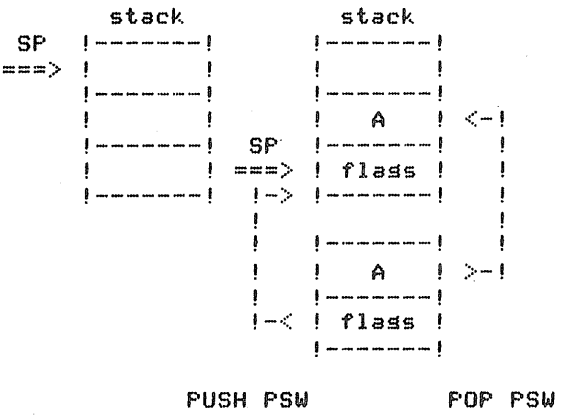


Figure 3.7. Contents of the accumulator and flag registers are pushed onto the stack.

Data can be moved from one register pair to another by using a PUSH/POP combination. For example, the two 8080 commands

```
PUSH  H
POP   D
```

will move H to D and L to E. This is not the most efficient way to accomplish the move, however. The sequence requires access to main memory and so is slower than the direct register moves

```
MOV   D,H
MOV   E,L
```

Z-80 INDEX REGISTERS

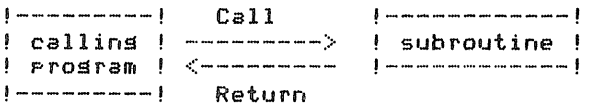
The Z-80 has two, 16-bit index registers that can participate in the PUSH and POP operations. However, the instructions each require two bytes compared to the other PUSH and POP instructions which only require one byte each. As a result, the execution time is slower than the other PUSH and POP instructions. There are no official instructions for moving data between the index registers and the general-purpose registers. This transfer can be performed, however, by use of the PUSH and POP commands. The two instructions

```
PUSH  IX
POP   BC
```

will copy the IX register into the BC register.

SUBROUTINE CALLS

We have seen that the PUSH instructions can be used by the programmer to store data on the stack. The 8080 and Z-80 CPUs use the stack for a second purpose: storing the return address when a subroutine is called. Subroutines are used to efficiently code a set of instructions needed at several different places in a computer program. A subroutine is called by using the assembly-language mnemonic CALL. At the end of the subroutine, indicated by the return statement, control is automatically returned to the calling program.



The input and output routines which control the console may be needed at several locations in a program. Consequently, they are coded as

subroutines. The 8080 assembly language subroutine for the console might look like this.

```
OUTPUT: IN      STATUS  ; CHECK STATUS
        ANI     INMSK   ; INPUT MASK
        JZ      OUTPUT  ; NOT READY
        MOV     A,B     ; GET DATA
        OUT     DATA    ; SEND DATA
        RET             ; DONE
```

Data can be output from anywhere in a program by placing the byte in the B register and calling the output subroutine. The following examples of 8080 assembly language mnemonics show how a question mark and a colon can be printed by calling the console output routine.

```
WHAT:   MVI     B,'?'   ;OUTPUT A ?
        CALL    OUTPUT
. . .
        . . .
COLON:  MVI     B,':'   ;OUTPUT A COLON
        CALL    OUTPUT
        . . .
```

The above examples utilize the unconditional subroutine call and unconditional return instructions. Conditional call and return instructions are also available. These commands perform the appropriate call or return only if the referenced PSW flag is in the desired state. The four flags—zero, sign, carry, and parity—give rise to eight conditions.

zero
not zero
plus
minus
carry
not carry
parity even
parity odd

These instructions are discussed in more detail in Appendix H.

The stack provides the mechanism for subroutine operation. When a CALL instruction is encountered, the address immediately following the CALL statement is automatically pushed onto the stack. The subroutine address is then loaded into the program counter register. The program counter tells the CPU which instruction to execute next. Since a subroutine CALL uses the stack, the programmer must be sure that the stack is properly defined prior to a subroutine CALL. When a return instruction is subsequently encountered, the return address is popped off the stack and placed into the program counter. After return from a subroutine, program execution continues with the instruction following the CALL statement.

PASSING DATA IMPROPERLY TO A SUBROUTINE

Since the stack can be used for storing both data and subroutine return addresses, the programmer must ensure that there are no conflicts. First, there should normally be as many POP instructions as PUSH instructions. Second, one must be careful not to PUSH data onto the stack, CALL a subroutine, then POP data off the stack. The LIFO nature of the stack will cause trouble in this case.

```
        PUSH    H
        CALL    ORDER >---!
        . . .             !
        . . .             !
ORDER:  . . .         <---!
        POP     H
        . . .
        RET     >----> ???? CRASH !
```

Figure 3.8 shows an example of improper mixing of data and the return address on the stack. Higher memory is upward and lower memory is downward. The arrow indicates the current stack pointer position.

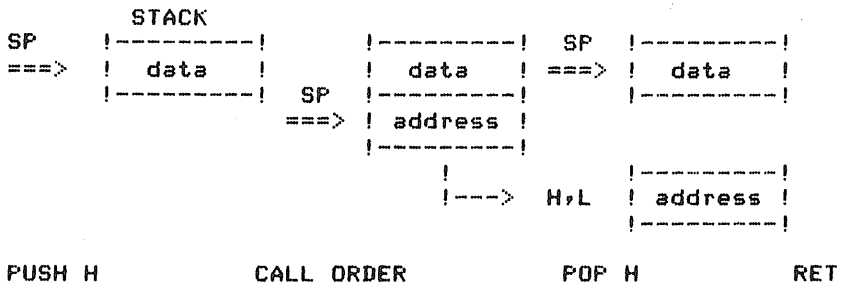


Figure 3.8. Improper mixing of data and the return address on the stack.

In this example, the data is first pushed onto the stack while in the main program. The return address is then pushed onto the stack next, when the CALL instruction is encountered. The POP instruction in the subroutine will actually load the HL register pair with the subroutine return address rather than the data that was expected. This occurs because the data was pushed onto the stack before the return address. Worse yet, the RET instruction will load the program counter with the data, rather than with a useful address. Strange things are likely to happen when the CPU attempts to execute instructions at an address defined by the data.

PASSING DATA PROPERLY TO A SUBROUTINE

This section demonstrates a proper way to pass data into a subroutine by using the stack. The task can be accomplished with the 8080 XTHL instruction

or the Z-80 EX (SP),HL instruction. This operation exchanges the HL register pair with the two bytes at the current stack position. The instruction is analogous to the X/Y EXCHANGE key on an HP calculator.

The method works in the following way. The data is pushed onto the stack while control is in the calling program. When the subroutine is called, the return address is pushed onto the stack, just after the data. A POP instruction, executed in the subroutine, delivers the return address to the HL register. Now, the XTHL instruction exchanges the HL register with the stack. The desired data is now in HL and the return address is on the stack. Finally, a return instruction will correctly return control to the calling program.

```
        PUSH    H       ; main program
        CALL    ORDER   ; call subroutine
        . . .
        . . .
ORDER:  . . .           ; start of subroutine
        POP     H       ; get return address
        XTHL            ; exchange with data
        . . .
        RET             ; return to main program
```

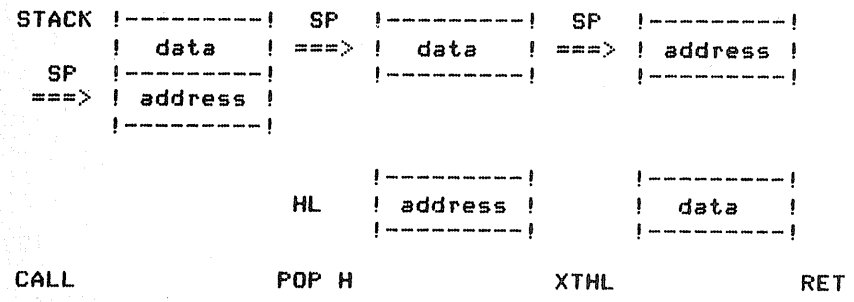


Figure 3.9. Proper mixing of data and return address on the stack.

It is important to note that the XTHL command only works with the HL register. There is no equivalent instruction for the DE or BC registers.

PASSING DATA BACK FROM A SUBROUTINE

A variation of the XTHL technique is also possible. Data can be pushed onto the stack from within a subroutine, then retrieved after returning to the calling program.

```
        CALL    FETCH
        POP     H       ;GET THE DATA
        . . .
        . . .
FETCH:  . . .
        LXI     H,DATA  ;PUT IN H,L
        XTHL            ;SWITCH STACK
        PUSH    H       ;RET ADDR
        . . .
        RET
```

DATA in this case is predefined and is part of the LXI instruction.

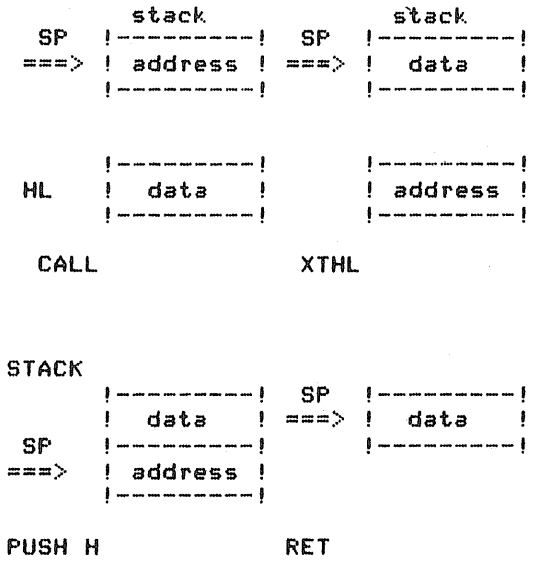


Figure 3.10. Using the stack to pass data back from a subroutine.

An extension of the XTHL technique allows additional data to be passed on the stack.

```
        CALL    FETCH
        POP     B       ;DATA 3
        POP     D       ;DATA 2
        POP     H       ;DATA 1
        . . .
        . . .
FETCH:  . . .
        LHLD    DATA1   ;DATA1 TO H,L
        XTHL            ;SWITCH STACK
        XCHG            ;STACK TO DE
        LHLD    DATA2   ;GET DATA2
        PUSH    H       ;PUT ON STACK
        LHLD    DATA3   ;GET DATA3
        PUSH    H       ;PUT ON STACK
        PUSH    D       ;RET ADDR TO STACK
        RET
```

In this example, the return address is first moved to the HL pair with the XTHL command. Then it is moved to the DE register pair with the XCHG instruction. Three sets of 16-bit data are obtained from the memory addresses pointed to by the arguments of the LHLD instructions DATA1, DATA2, and DATA3. The first set is placed on the stack with the XTHL command. Then the other two are pushed onto the stack. Next, the return address, previously saved in the DE register pair, is pushed onto the stack. A final RET instruction pops the return address from the stack into the program counter.

SETTING UP A NEW STACK

Sometimes it is desirable to save the current stack pointer and set up a new one. When this happens, the original stack pointer is restored at the conclusion of the task. The technique is particularly useful when one independent program is executed by another. The original stack pointer is saved in a memory location, then retrieved at the end of the program.

If the current program was reached through a subroutine call, the return address for the calling program should be the current address on the original stack. It is this address that must be saved.

There is a Z-80 instruction that allows the old stack pointer to be stored directly in main memory. The instruction looks like this.

```
; Z-80 VERSION
;
START:  LD      (OLDSTK),SP ;save stack
        LD      SP,STACK    ;new stack
        . . .
        . . .
        LD      SP,(OLDSTK) ;set  old stack
        RET                 ;done
```

At the conclusion of the task, the old stack pointer is restored. With an 8080 CPU, the job is more complicated since the stack pointer cannot be directly saved. In this case, the stack pointer is moved to the HL register pair which is in turn saved in memory. This is done by first zeroing HL, then adding in the stack pointer. At the end of the routine, the old stack pointer is loaded into the HL register pair then copied into the stack pointer register. Finally, a RET instruction is given.

```
START:  LXI     H,0         ;zero HL
        DAD     SP          ;SP to HL
        SHLD    OLDSTK      ;save stack
        LXI     SP,STACK    ;new stack
        . . .
        . . .
        LHLD    OLDSTK      ;get old stack
        SPHL                ;restore stack
        RET
```

CALLING A SUBROUTINE IN ANOTHER PROGRAM

A program may need to call a subroutine that resides in another program. But if the second program is revised, the subroutine address in the second program will change. This means that the argument of the CALL statement in the first program will also have to be changed.

There are two ways to solve this problem. One method is to provide a jump instruction near the beginning of the second program. The address of the jump instruction will always be the same. However, its argument, the internal subroutine address, can change from one version to the next. The first program simply calls CHEK2, and CHEK2 causes a jump to CHEK, the desired subroutine. The RET instruction at the end of CHEK will effect a proper return to program 1.

```
                    . . .                   ; Program 1
!--------           CALL        CHEK2       ; call Program 2
!                   . . .                       <------------!
!                   . . .           .                        !
!     START:        JMP         CONTIN      ;start of Program 2   !
!->   CHEK2:        JMP         CHEK          >----------------!   !
                    . . .                                     !   !
                    . . .                                     !   !
      CHEK:         . . .                           <---------!   !
                    RET                     ;to Program 1 >-------!
```

Of course, the second program may need to save the incoming stack, then restore it before returning to program 1.

A second solution is to place just the two-byte address of the subroutine near the beginning of the second program.

```
START:  JMP     CONTIN  ; Program 2
CHEK2:  DW      CHEK
```

Now the calling program must put its own return address on the stack and get the address of CHEK into the program counter. The following example is a way to do this. Notice that program 1 does not enter program 2 with a CALL instruction. It uses instead the PCHL instruction which copies the contents of HL into the program counter.

```
        PUSH    H         ;SAVE H,L
        LXI     H,NEXT    ;RET ADDR
        PUSH    H         ;ONTO STACK
        LHLD    CHEK2
        PCHL              ;INTO PC
NEXT:   POP     H         ;ORIG H,L
```

CALLING ONE SUBROUTINE FROM ANOTHER

A subroutine called by a main program may in turn call another subroutine. When the first subroutine, SUB1, is called, the return address to the main program, MAINA, is pushed onto the stack. When the second subroutine, SUB2, is called, the return address SUB1A is next pushed onto the stack. After the second subroutine has been called, there will be two return addresses on the stack: one to get back to SUB1 from SUB2, and the other to get back to the main program from SUB1.

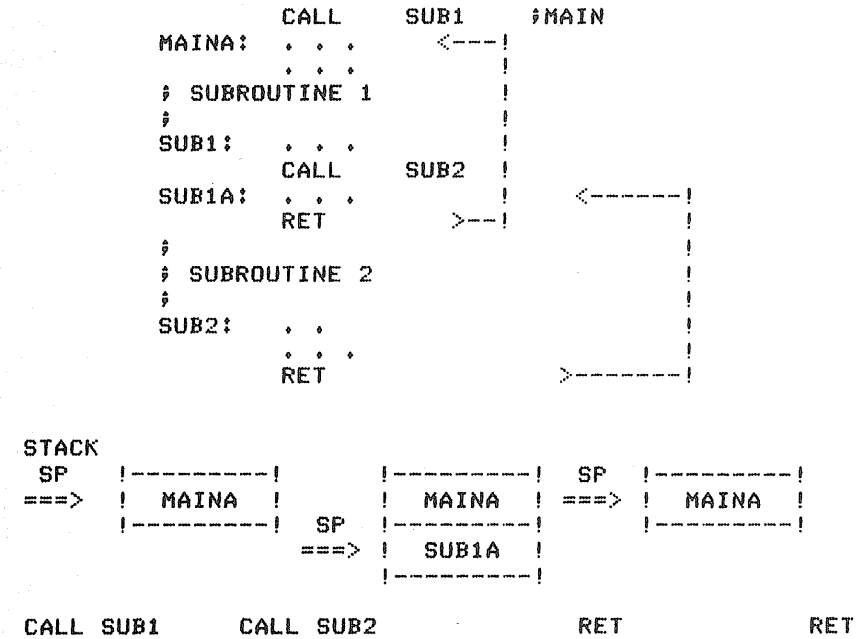


Figure 3.11. One subroutine calls another.

BYPASSING A SUBROUTINE ON RETURN

It may be that an operation in the sectond subroutine SUB2 makes it desirable to return directly to the main program from SUB2, bypassing SUB1. This is easily accomplished if the stack pointer is raised by two bytes before executing the return instruction. Of course, care should be taken to see if data has been pushed onto the stack after one or both return addresses were placed on the stack. The one-byte instruction to increment the stack pointer (INX SP) can be executed twice, to raise the stack pointer two bytes. Alternately, a one-byte POP command can be used if there is a free register pair available.

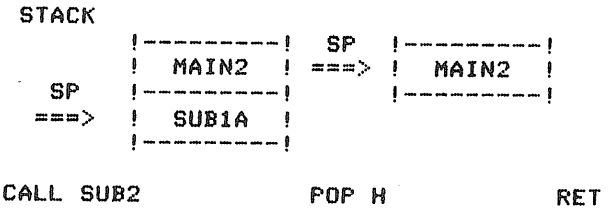


Figure 3.12. Skipping one level of subroutine during the return.

Suppose that an ordinary return from subroutine SUB2 back to subroutine SUB1 is desired if the zero flag is set to 1. On the other hand, an unusual return directly back to the main program is desired if the zero flag is reset to a value of zero. Here is a way to do this.

```
        CALL    SUB1    ;MAIN
MAIN1:  . . .                   <----------!
        . . .                              !
; SUBROUTINE 1                             !
;                                          !
SUB1:   . . .           ;SUBROUTINE 1      !
        CALL    SUB2                <---!  !
SUB1A:  . . .                           !  !
        RET                             !  !
                                        !  !
; SUBROUTINE 2                          !  !
;                                       !  !
SUB2:   . . .                           !  !
        . . .                           !  !
        RZ              ;NORMAL RETURN >-  !
        POP     PSW     ;RAISE STACK       !
        RET             ;SKIP TO MAIN >----!
```

The POP PSW instruction raises the stack two bytes so that the final RET instruction delivers the return address of MAIN1 to the program counter. This effectively bypasses the intermediate subroutine.

A PUSH WITHOUT A POP

Near the beginning of the system monitor, explained in Chapter 6, there is a restart address called WARM. The program normally branches back to this point at the conclusion of each task. Thus the final instruction of each task could be:

```
JMP     WARM
```

A more efficient method, however, is to push this restart address WARM onto the stack at the beginning of the task. Then if the task does not terminate within a subroutine, a simple return instruction, rather than a jump, can be given at the end of the task. This causes a branch back to WARM.

```
WARM:   LXI     H,WARM  ;H,L = HERE  <----!
        PUSH    H       ;ONTO STACK        !
        . . .                              !
        . . .                              !
        JZ      OPORT                      !
        . . .                              !
OPORT:  . . .                              !
        RET         >----------------------!
```

This example is an exception to the rule that we should have a POP instruction for every PUSH. Here, there is a PUSH but no POP. Of course there is also a RET with no CALL. So everything is all right—or is it? What happens if termination occurs from a subroutine?

GETTING BACK FROM A SUBROUTINE

If a particular task terminates in a subroutine, then this subroutine's return address must be popped off the stack (or an INX SP instruction must be executed twice) before the return is issued.

```
WARM:   . . .                      <---------!
        JZ      DUMP                         !
        . . .                                !
DUMP:   . . .                                !
        CALL    TSTOP                        !
        . . .           <--------------!     !
        . . .                         !     !
TSTOP:  . . .                         !     !
        RNC             ;NORMAL RETURN >-!   !
        POP     H       ;RAISE STACK         !
        RET             ;TO WARM     >-------!
```

The stack pointer grows downward through memory during use. It is therefore common practice to place the stack as high as possible in available memory. But the system monitor may be located at the actual top of memory. In this case the stack can initially be placed lower in memory at the beginning of the monitor.

```
START:
        LXI     SP,START
```

On the other hand, the monitor may be placed in read-only memory (ROM). In this case, the stack can be located at the actual top of read/write memory. (While both read/write memory and ROM are random-access memory—RAM, it is customary to refer to read/write memory as RAM and read-only memory as ROM. This convention will be followed here.)

AUTOMATIC STACK PLACEMENT

The placement of the stack at the top of RAM can be done automatically, so that the total amount of RAM can be changed without having to reprogram the PROM monitor. A short routine can test each block of memory starting at zero until it finds a location that can't be changed. The stack is then put at the beginning of this block. Remember, the stack pointer is always decremented before use; therefore, it can be initially defined as one location above usable memory.

The first part of the program is a memory search routine that starts at address zero. It moves the byte from that location into the accumulator, complements it, then moves the complemented byte back to the original location. A comparison is made to see if the memory location does indeed contain the complemented byte. If it does, the accumulator is complemented back to the original byte and returned to memory. Such an algorithm is often called a nondestructive memory test.

The first byte of each subsequent block of memory is checked in this way until a failure is found. This will usually reflect the top of usable memory, but of course, it could indicate defective memory. The following program will work properly if placed in read-only memory.

```
;
; ROUTINE TO AUTOMATICALLY PLACE THE
; STACK AT THE TOP OF MEMORY
; 8080 CODE
;
        LXI     H,0     ;FIRST ADDR
NEXTP:  MOV     A,M     ;GET BYTE
        CMA             ;COMPLEMENT
        MOV     M,A     ;PUT IT BACK
        CMP     M       ;COMPARE?
        JNZ     TOP     ;NO, DONE
        CMA             ;BACK TO ORIG
        MOV     M,A     ;PUT IT BACK
        INR     H       ;NEXT BLOCK
        JMP     NEXTP   ;KEEP GOING
;
TOP:    SPHL            ;SET STACK
        CALL    OUTHL   ;PRINT IT
        . . .
```

This program might not work, however, if it is placed in read/write memory. The problem occurs because the routine is changing various locations in memory. If it happens to change its own instructions, then the results will be unpredictable.

The shortcomings of the previous program are solved with the following version. The improved version will operate properly no matter where it is placed. The stack will be placed at the top of contiguous RAM unless the routine itself is in that part of memory. In that case, the stack will be placed at the beginning of the program. The Z-80 version is shown, but the program can be run on an 8080 if two minor changes are made. The relative jump instruction must be changed to an absolute jump and the DJNZ instruction must be changed to the equivalent DCR B and JNZ combination.

```
;
; ROUTINE TO AUTOMATICALLY PLACE THE
; STACK AT THE TOP OF MEMORY
; FAILSAFE VERSION  (Z-80 CODE)
;
START:  LD      HL,0      ;START CHECK AT 0
        LD      B,START SHR 8
NEXTP:  LD      A,(HL)    ;GET BYTE
        CPL               ;COMPLEMENT IT
        LD      (HL),A    ;PUT IT BACK
        CP      (HL)      ;DID IT GO?
        JR      Z,TOP     ;NO, DONE
        CPL               ;BACK TO ORIG
        LD      (HL),A    ;RESTORE
        INC     H         ;NEXT BLOCK
        DJNZ    NEXTP     ;ARE WE HERE?
;
TOP:    LD      SP,HL     ;SET STACK
        . . .
```

The new version works in the following way. The B register initially contains the block number of the routine itself. The value in B is decremented as each successive block is checked. If the routine is in ROM, then the end of usable memory will be found, as in the previous version. The program will loop between the label NEXTP and the DJNZ NEXTP instruction. At some point, the CP (HL) instruction will reset the zero flag and the computer will jump to the address of TOP. The stack will then be placed at the top of RAM.

Alternately, if this routine is placed in the lower memory area, then the DJNZ instruction will decrement the B register all the way to zero. The zero flag will be set and the program will move on to TOP. Now the stack will be set to the beginning of the memory block that contains the program itself.

The START SHR 8 expression at the beginning of the routine instructs the assembler to calculate the high byte of the address of START and make it the second operand of the LD B instruction. It does this by shifting the

address of START by eight bits to the right, then taking the low-order eight bits of the result. Some assemblers allow an equivalent operand of

```
HIGH    START
```

which is easier to comprehend. This automatic stack routine is incorporated into the system monitor explained in Chapter 6.

CHAPTER FOUR

Input and Output

Computers would not be very useful if they could not interact with the outside world. Commands and data are sent to the computer from the keyboard, magnetic tape, disk, and other peripherals. Results of computations are sent back from the computer to the printer, video terminal, tape unit, disk, and so on. Such input and output (I/O) transfers on a microcomputer are typically accomplished through special memory locations called I/O *ports*. One type of port is distinctly different from main memory. The other type of arrangement utilizes one of the regular main memory locations. The peripheral in this latter case is then said to use *memory-mapped* I/O. Each method has advantages and disadvantages. In either case, the I/O port will transfer eight bits, the natural word size for the 8080 and Z-80 CPUs.

MEMORY-MAPPED I/O

The I/O instructions on the 8080 microprocessor are rather limited compared to memory operations. There is a single IN and a single OUT instruction for transferring eight bits of data. In contrast, there is a much larger collection of memory operations available.

```
(8080 Mnemonics)          (Z-80 mnemonics)
   STA     80              LD      (80),A
   LDA     81              LD      A,(81)
   MOV     M,C             LD      (HL),C
   STAX    D               LD      (DE),A
   SHLD    84              LD      (84),HL
```

These additional instructions can be utilized with memory-mapped I/O, greatly increasing the versatility of the Z-80 and 8080 I/O operations.

The STA instruction stores the 8-bit accumulator value at the memory address specified by the operand. If this address corresponds to a memory-mapped port, then the byte is sent to the peripheral. The LDA command reverses the operation. It can be used to input a byte from a port. The MOV M,C instruction can be used to transfer a byte from the C register to the memory location designated by the HL register pair. The STAX D command moves a byte from the accumulator to the memory location designated by the DE register pair. The SHLD instruction opens a new dimension. Since this operation transfers 16 bits of data from the HL register pair directly into two consecutive memory locations, two adjacent ports can be simultaneously serviced.

The typical video console is a serial device that uses distinct ports. However, memory-mapped controller boards are commerically available. In this case, an ordinary TV set is then used for the video screen. There are also disk-controller boards that use memory-mapped operations to communicate with the disk drives. It is interesting to note that the Motorola 6600 CPU performs all of its I/O by memory mapping. There are no separate input or output instructions for this CPU.

DISTINCT DATA PORTS

Data ports may be designed to operate either in parallel or in serial fashion. Both the parallel and the serial I/O ports are connected to the computer through the system bus by a set of eight data lines. In addition, the parallel port is connected to the peripheral by another set of eight data lines. The serial port, by contrast, has only two data lines connecting it to the peripheral.

For some peripherals, such as a printer, data is transferred in only one direction. For others, such as the console and magnetic tape units, the peripheral is able to both send and receive data. In this latter case, there will be 16 data lines between the computer and the peripheral if a parallel port is used. Eight lines are used for sending data and eight are used for receiving data. The serial port, in contrast, will have three signal lines to the peripheral if there is two-way communication. One is for transmitting, one is for receiving, and the third is a common line for the other two.

There may be additional lines between the computer and the peripheral. One of these might indicate to the computer whether the terminal is operational. Another can be used to inform the terminal that the computer is ready. These extra lines are sometimes referred to as *handshake* lines.

The computer usually operates at a much higher speed than the peripherals. Consequently, there must be a mechanism for effectively slowing down the computer during I/O operations. For serial or parallel ports, this is typically accomplished by using two separate I/O ports for each peripheral device. One port is used for the data port and the other is used for the status port. Each of these two ports will have distinct addresses, one of the 256 values available to the 8080 or Z-80 CPU for this purpose. There are three

general methods of performing I/O through data ports: looping, polling, and interrupting.

LOOPING

Looping is the simplest method of performing I/O through separate ports, and it is the one that is most commonly employed in 8080 and Z-80 programs. The CPU performs output by sending a byte to the data port using the OUT instruction. The corresponding status port is then read with an IN instruction. One bit of the 8-bit status port reflects the condition of the corresponding peripheral.

When the CPU places a byte in the data register, using the OUT command, the output status bit of the status register is set. This may actually result in a logical 1 or a logical zero, depending on the port design. When the peripheral utilizes the byte that was placed into the data register, the output status bit of the status register is reset. These changes in the status bit are automatically handled by the I/O interface hardware. However, the programmer must include in the software the appropriate routines for monitoring the status bits.

As an example of the looping method, consider the following subroutine:

```
COUT:   IN      10H     ;CHECK STATUS
        ANI     2       ;SELECT BIT
        JZ      COUT    ;NOT READY
        MOV     A,C     ;GET BYTE
        OUT     11H     ;SEND
        RET             ;DONE
```

This routine could be used to send a byte of data to the system console. The first instruction of the listing causes the CPU to read the 8-bit status port which has the address of 10 hex. The second instruction performs a masking AND operation to select the write-ready bit, bit 1. Remember that a logical AND with zero and anything else gives a result of zero. However, a logical AND with unity and a second logical value, gives the result of that second value.

Suppose that the output status is indicated by a logical 1 of bit 1, where bit 0 is the least-significant bit of the register. Then, a logical AND with the value in the status register and with the number 2 will result in a logical 1 if the peripheral is ready. If the device is not ready, however, the result is a logical 0.

```
        0101 0111       status      0101 0101
AND     0000 0010       = 2         0000 0010
        ---------                   ---------
        0000 0000                   0000 0010
          ready                     not ready
```

Thus, the logical AND with the value of 2 in the status register gives a result of zero if bit 1 (the second bit) is 0. Otherwise, a nonzero result is obtained.

The third instruction in the looping example is a conditional jump. If the peripheral is not ready, the JZ instruction will cause the computer to loop repeatedly through the first three lines until the peripheral is ready for another byte. At this point, the write-ready bit, bit 1, will be a logical 1. Then the logical AND operation, the second instruction of the subroutine, produces the nonzero value of 2. The MOV instruction following the conditional jump will then be executed. The byte to be outputted is moved to the accumulator, and then sent to data port 11 hex by use of the OUT command.

When the byte to be output is actually sent to the data port, the write-ready flag is reset to a logical zero. The output routine may be immediately reentered for outputting another byte, but now the peripheral is not ready. Looping will occur again through the first three instructions of the output routine since the write-ready flag has been reset to zero.

The CPU clock may be operating at 2 or 4 MHz. This rate is thousands of times faster than the speed of a typical printer. Consequently, if the looping method is used, the CPU will be spending over 99 percent of its time simply looping through the first three lines of the output subroutine. The computer will be spinning its wheels, so to speak, waiting on the peripheral.

Because the CPU is operating so much faster than the peripherals, it can, in principle, service many peripherals simultaneously. A very simple but useful implementation of this idea is found on the CP/M* operating system. In the CP/M system,* console output is normally sent only to the console. This terminal is typically a high-speed video device. But if the user types a Control-P, then the list device is also turned on. Console output will now appear simultaneously at both the console and the line printer.

This technique can be easily observed if the console video accepts data much faster than the line printer. Normally, as data is sent only to the console, it appears rapidly on the video screen. But when the list device is turned on, the output appears much more slowly. The reason for the slowdown is that both peripherals are operating at the speed of the slower one, in this case the printer.

A subroutine for accomplishing such a dual output might look like this.

```
LOUT:   IN      LSTAT   ;LIST STATUS
        ANI     2       ;OUTPUT MASK
        JZ      LOUT    ;LOOP UNTIL READY
        MOV     A,C     ;GET THE BYTE
        OUT     LDATA   ;SEND TO LIST
        OUT     CDATA   ;AND CONSOLE
        RET             ;DONE
```

*CP/M is a registered trademark of Digital Research, Inc., Pacific Grove, California.

This routine is not the one that is actually used in the CP/M system since, with our routine, the console will always display everything that is sent to the printer. This feature does not increase printing time as long as the console operates faster than the printer. Notice that there is no need to check the console status register. The output rate is set at the speed of the printer, and so the console, which operates so much faster, will always be ready if the printer is ready.

POLLING

One way to improve the performance, or throughput, of a CPU is with a technique known as polling. In this method, the CPU sends a byte to each of several different peripherals. Each peripheral operates at its full speed. Polling is more efficient than the looping method, and has been incorporated into several commercial 8080 software products. One product is a multiuser BASIC which can service up to four separate consoles. Each user can independently perform calculations using the same BASIC interpreter.

Another product that uses the polling technique is known as a *spooler.* The looping method is typically utilized for all output. In this case, all other activities must be halted while the printer is working. With a spooler program, however, things are different. When this program is incorporated into the system, the user can perform other tasks using the system console while a disk file is being printed.

In the polling method, the I/O routines are somewhat different from the corresponding routines of the looping method. The output-ready flag of the status register is checked periodically as with the looping method. But if the status flag indicates that the device is not ready, the CPU returns to perform some other task. Thus, the CPU does not waste time looping around the first three instructions of the input or output routine. A typical output routine using the polling method might look like tihis.

```
LOUT:   IN      LSTAT   ;CHECK STATUS
        ANI     LMASK   ;MASK FOR OUTPUT
        RZ              ;NOT READY
        MOV     A,C     ;GET THE BYTE
        OUT     LDATA   ;SEND IT
        RET
```

While the polling method is a great improvement over the looping method, there are still problems. For example, a decision must be made as to how often each status register will be polled. An even better method is to use hardware interrupts.

HARDWARE INTERRUPTS

The 8080 and Z-80 microprocessors incorporate a hardware interrupt system. This feature allows an external device, such as the system console or printer,

to interrupt the current task of the processor. When the CPU is interrupted, it suspends its current task, and calls on one of several memory locations set aside for this purpose. The CPU services the request of the interrupting peripheral, then it returns to its previous task.

In this method, the CPU does not have to be programmed to check the peripherals on a regular basis as with the method of polling; nor does it have to waste time in a loop. Instead, the peripheral interrupts the processor when it needs service. If several peripherals are able to interrupt the CPU, then there must be a method for prioritizing the requests. This ordering is accomplished through a vectored interrupt system. For example, if a lower-priority device has interrupted the CPU for service, this phase can also be interrupted by a peripheral with a higher priority. On the other hand, a device with a lower priority cannot interrupt a higher-priority service, but must wait its turn.

Usually, the highest-priority interrupt will be assigned to updating the system clock. If the computer misses a beat, then the time will be incorrect. The next lower priority could be assigned to disk transfer. The printer could have a low priority since it is a relatively slow device, and it won't matter if it must slow down every so often.

Suppose that the printer is operated by interrupts rather than by looping or polling. The computer sends a byte to the printer, then continues with another task. When the current byte has actually been printed, the printer interrupts the CPU for another byte. In the time between the printing of two bytes, the CPU can perform many other tasks.

The console keyboard is another peripheral that can be readily serviced by an interrupt system. In this case, each time the user presses a key, the CPU is interrupted from its current task. Of course, if the CPU is currently servicing a higher-priority interrupt, then the console keyboard request will have to wait.

Both the 8080 and the Z-80 allocate eight addresses that can be used for the interrupt service routines. These addresses can be called by the eight, one-byte RST instructions.

Z-80	8080	Instruction code		Call
mnemonic	mnemonic	hex	binary	address
RST 00H	RST 0	C7	1100 0111	00H
RST 08H	RST 1	CF	1100 1111	08H
RST 10H	RST 2	D7	1101 0111	10H
RST 18H	RST 3	DF	1101 1111	18H
RST 20H	RST 4	E7	1110 0111	20H
RST 28H	RST 5	EF	1110 1111	28H
RST 30H	RST 6	F7	1111 0111	30H
RST 38H	RST 7	FF	1111 1111	38H

These instructions can be used as one-byte subroutine calls. As an example, suppose that the CPU executes an RST 5 instruction which corresponds to the instruction code EF hex. A subroutine call is then made to the corresponding address of 28 hex. The return address is pushed onto the stack,

just as for a regular subroutine call. Subsequent execution of a return instruction will cause the program flow to return to the instruction immediately following the RST 5 instruction.

Hardware interrupts operate by emulating the software RST call. When an interrupt occurs, the CPU automatically disables the interrupt flip-flop, thus further interrupts are prevented. Then a subroutine call is made to the corresponding call address. This is done by jamming the desired RST code onto the data bus. The simplest implementation is to use a single interrupting device and the RST 7 instruction. (A normal interrupt always performs an RST 7.) The interrupting peripheral momentarily changes the state of the interrupt-request bus line. For the S-100 bus, this would require that bus line 73 be pulled to a zero-voltage state from the usual 5-volt level. The CPU responds by automatically calling memory address 38 hex. The programmer will have previously placed the service routine at this location. The service routine will conclude with a command to re-enable the interrupt flip-flop. Then a return instruction will be executed.

The trouble with this simple approach is that the RST 7 call to location 38 hex interferes with system debuggers because they also use this address. Consequently, another interrupt level is more suitable. Unfortunately, a single interrupt system always calls the RST 7 location. One solution to this problem is to use a vectored interrupt board. A vectored interrupt board allows the user to select up to eight separate interrupt levels corresponding to the RST 0 to 7 instructions. The disadvantage of this approach is the cost, since a vectored interrupt board may sell for several hundred dollars.

However, there is a low-cost solution. If only one interrupt level is required, a single hardware interrupt can be converted from an RST 7 to some other level such as an RST 5 by using only two logic gates. The circuit shown in Figure 4.1 will make the needed translation. The output of the two-input NAND gate IC-1 goes low when both of the input lines are high. One of these inputs is SINTA, line 96 on the S-100 bus. It is a CPU status signal that indicates acknowledgment of the interrupt request. The other input is PDBIN, bus line 78. This signal indicates that the data bus is in the input mode.

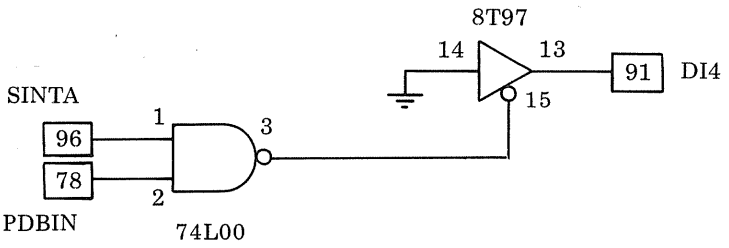


Figure 4.1. Circuit to convert an 8080 interrupt to an RST 5.

When the output of IC-1 goes low, it turns on the three-state buffer IC-2. This pulls the data-input bus line DI4 low. Since the remaining seven lines of the data-in bus are high, the CPU will see the value of

1110 1111

Notice that this is the bit pattern for the RST 5 instruction. The result is that the CPU executes an RST 5 instruction, by calling address 28 hex. The interrupt service routine, or a jump to it, is placed at this address.

AN INTERRUPT-DRIVEN KEYBOARD

We have seen that a printer operates considerably slower than a CPU. The console keyboard is even slower than the printer, especially if the operator is not an expert typist. Conversion to an interrupt-driven keyboard will considerably increase the effectiveness of a computer.

Characters entered on an interrupt-driven keyboard are temporarily stored in a memory buffer area. Each time a key is pressed on the console, the CPU is interrupted from its current task. The new byte is read and placed into the keyboard buffer. The computer then returns to its prior task. When the computer needs console input, it gets it from the input buffer, rather than from the console itself.

An interface program, utilizing a keyboard-interrupt approach, is shown in Listing 4.1. This program provides the necessary routines for interfacing the Lifeboart version of CP/M to a North Star disk system.* The portions of the program which specifically utilize the interrupt routines begin with a row of asterisks and end with a row of semicolons.

Computer pointer F400	Computer count F402	Keyboard count F403	Keyboard pointer F404	Buffer F406

Figure 4.2. The input buffer and pointers.

The layout of the memory buffer with its pointers is shown in Figure 4.2. The buffer area is arbitrarily chosen to start at the address of F400 hex. The location can be anywhere above the CP/M operating system. There is only one keyboard buffer, but there are two sets of pointers: one for the CPU and one for the keyboard. Two counters are also utilized; one shows how many characters have been entered from the keyboard and the other shows how many have been read by the computer. Since both sets of pointers grow larger, they need to be reset periodically. The two pointers are compared after each carriage return. If they are the same, then they are both reset to the beginning of the buffer.

Suppose that this interface program is incorporated into your system. CP/M might be printing something on the console video screen when a key on the console is pressed. A hardware interrupt will occur, causing the computer to stop its task and call address 28 hex (RST 5). A jump instruction at address 28 hex will transfer control to subroutine KEYBD. The keyboard

*Lifeboart Associates, 2248 Broadway, New York, N.Y. 10024.

Listing 4.1. Interrupt driven keyboard.

```
                TITLE   'Interrupt CP/M BIOS'
                ;
                ; (Put today's date here)
                ;
                ; LIFEBOAT VERSION WITH OPTION FOR
                ; EITHER SINGLE OR DOUBLE DENSITY
                ;
                ; TERMINAL DEVICES SUPPORTED:
                ;
                ;  CONSOLE       10 HEX  CON:
                ;  LIST.         12 HEX  LST:
                ;  PHONE MODEM   14 HEX  PUN:
                ;
0000 =          FALSE   EQU     0
FFFF =          TRUE    EQU     NOT FALSE
                ;
FFFF =          DOUBLE  EQU     TRUE    ;DOUBLE DENSITY
FFFF =          INTRM   EQU     TRUE    ;INTERR VERSION
                ;
                        IF      DOUBLE
0036 =          MSIZE   EQU     54      ;DECIMAL K
D600 =          BIOS    EQU     MSIZE*1024-200H
DB00 =          USER    EQU     BIOS+500H
4900 =          OFFSET  EQU     1F00H-BIOS
                        ELSE    ;SINGLE DENSITY
                MSIZE   EQU     56
                USER    EQU     MSIZE*1024-700H
                        ENDIF
0003 =          IOBYTE  EQU     3       ;I/O SETUP
000D =          CR      EQU     0DH     ;CARRIAGE RET
000A =          LF      EQU     0AH     ;LINEFEED
000C =          FFEED   EQU     12      ;FORMFEED
0003 =          CTRC    EQU     3       ;^C, KILL SCROLL
0004 =          CTRD    EQU     4       ;^D, EMPTY BUFFER
0011 =          CTRQ    EQU     17      ;^Q, SCROLL
0013 =          CTRS    EQU     19      ;^S, FREEZE SCROLL
                ;
                        IF      DOUBLE
                ; PATCH DATE
D5B1            ORG     BIOS-100H+0B1H
                        ELSE
                ORG     USER-600H+0AFH
                        ENDIF
D5B1 2E4465             DB      '.Jan 28,80' ;PATCH DATE
                ;
DB00            ORG     USER
                ;
0010 =          CSTAT   EQU     10H     ;CONSOLE STATUS
0011 =          CDATA   EQU     CSTAT+1 ;CONSOLE DATA
0001 =          CIMSK   EQU     1       ;INPUT MASK
0002 =          COMSK   EQU     2       ;OUTPUT MASK
0012 =          LSTAT   EQU     12H     ;LIST STATUS
0013 =          LDATA   EQU     LSTAT+1 ;LIST DATA
0001 =          LIMSK   EQU     1       ;INPUT MASK
0002 =          LOMSK   EQU     2       ;OUTPUT MASK
0000 =          LNULL   EQU     0       ;LIST NULLS
```

```
0014 =          MSTAT   EQU     14H     ;MODEM  STATUS
0015 =          MDATA   EQU     MSTAT+1 ;MODEM DATA
0040 =          MIMSK   EQU     40H     ;INPUT MASK
0080 =          MOMSK   EQU     80H     ;OUTPUT MASK
                ;
                ; INITIALIZE PORTS FOR COMPUTIME BOARD
                ;
00C4 =          ADATA   EQU     0C4H
00C5 =          ACONT   EQU     ADATA+1
00C6 =          BDATA   EQU     ADATA+2
00C7 =          BCONT   EQU     ADATA+3
                ;
                ;***************************************
                        IF      INTRM   ;INTERRUPTS
0095 =          STOP    EQU     95H     ;SET FOR INTERR
                ;
                ; CONSOLE INPUT-BUFFER LOCATION
                ;
F400 =          BUFFER  EQU     0F400H  ;INPUT BUFFER
F400 =          CPNTR   EQU     BUFFER  ;COMPUTER POINTER
F402 =          CCNT    EQU     CPNTR+2 ;BUFFER COUNT
F403 =          KCNT    EQU     CCNT+1  ;KEYBRD BUFF COUNT
F404 =          KPNTR   EQU     KCNT+1  ;KEYBOARD POINTER
F406 =          BUFF    EQU     KPNTR+2 ;INPUT BUFFER
0005 =          LEV     EQU     5       ;INTERR LEVEL
                        ENDIF           ;INTERRUPTS
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                START:
DB00 C315DB             JMP     INIT    ;INITIALIZATION
DB03 C3ABDB             JMP     CONST   ;CONSOLE STATUS
DB06 C3D3DB             JMP     CONIN   ;CONSOLE INPUT
DB09 C312DC             JMP     CONOUT  ;CONSOLE OUTPUT
DB0C C326DC             JMP     LOUT    ;LIST OUTPUT
DB0F C348DC             JMP     PUNCH
DB12 C3D3DB             JMP     CONIN   ;FOR READER
                ;
                ; INITIALIZATION ROUTINES
                ;
DB15 3E03       INIT:   MVI     A,3
DB17 D310               OUT     CSTAT   ;RESET
DB19 D312               OUT     LSTAT   ;INTERFACE
DB1B 3E15               MVI     A,15H
DB1D D312               OUT     LSTAT
                ;
                        IF      INTRM
DB1F 3E95               MVI     A,STOP  ;SET FOR INTERR.
                        ENDIF
                ;
DB21 D310               OUT     CSTAT   ;INTERFACE
                ;
                ; COMPUTIME BOARD INITIALIZATION
                ;
DB23 AF                 XRA     A       ;GET A ZERO
DB24 D3C5               OUT     ACONT
DB26 D3C7               OUT     BCONT
DB28 3E70               MVI     A,70H
DB2A D3C4               OUT     ADATA
```

```
DB2C 3E77               MVI     A,77H
DB2E D3C6               OUT     BDATA
DB30 3E14               MVI     A,14H
DB32 D3C5               OUT     ACONT
DB34 3E04               MVI     A,4
DB36 D3C7               OUT     BCONT
                ; ************************************
                        IF      INTRM
                ;
                ; PATCH RST LOCATION TO JUMP TO KEYBD
                ;
DB38 F3                 DI                      ;DISABLE INTERR
DB39 3EC3               MVI     A,0C3H          ;JMP INSTR
DB3B 322800             STA     8*LEV           ;PATCH RST
DB3E E5                 PUSH    H
DB3F 215BDB             LXI     H,KEYBD         ;INTERR ENTRY
DB42 222900             SHLD    8*LEV+1         ;JUMP HERE
DB45 2106F4             LXI     H,BUFF          ;BUFFER ADDR
DB48 2204F4             SHLD    KPNTR           ;RESET POINTERS
DB4B 2200F4             SHLD    CPNTR
DB4E 210000             LXI     H,0             ;2 ZEROS
DB51 2202F4             SHLD    CCNT            ;ZERO THE COUNTS
DB54 E1                 POP     H
DB55 FB                 EI                      ;RE-ENABLE INTERR
                        ENDIF                   ;INTERRUPTS
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; INITIALIZE IOBYTE
                ;
DB56 AF                 XRA     A               ;RESET IOBYTE
DB57 320300             STA     IOBYTE
DB5A C9                 RET
                ;****************************************
                        IF      INTRM
                ;
                ; INTERRUPT ENTRY FOR KEYBOARD INPUT
                ;
DB5B F5         KEYBD:  PUSH    PSW
DB5C DB10               IN      CSTAT           ;CONSOLE STATUS
DB5E E601               ANI     CIMSK
DB60 CA92DB             JZ      KEY2            ;NOT READY
DB63 DB11               IN      CDATA           ;GET DATA
DB65 E67F               ANI     7FH             ;MASK PARITY
                ;
                ; CHECK FOR ^S, ^Q SCROLL CONTROL
                ;
DB67 FE13               CPI     CTRS            ;^S
DB69 C27FDB             JNZ     KEY3            ;NO
DB6C DB10       KEY4:   IN      CSTAT           ;CHECK KEYBOARD
DB6E E601               ANI     CIMSK           ;READY?
DB70 CA6CDB             JZ      KEY4            ;LOOP UNTIL READY
DB73 DB11               IN      CDATA           ;GET BYTE
DB75 E67F               ANI     7FH             ;STRIP PARITY
DB77 FE11               CPI     CTRQ            ;^Q?
DB79 C26CDB             JNZ     KEY4            ;NO
DB7C C392DB             JMP     KEY2
                ;
```

```
DB7F E5         KEY3:   PUSH    H
DB80 FE04               CPI     CTRD    ;EMPTY BUFFER?
DB82 CA95DB             JZ      KEY6    ;YES
DB85 2A04F4             LHLD    KPNTR   ;BUFFER POINTER
DB88 77                 MOV     M,A     ;PUT IT THERE
DB89 23                 INX     H       ;INR POINTER
DB8A 2204F4             SHLD    KPNTR   ;SAVE POINTER
DB8D 2103F4             LXI     H,KCNT  ;GET COUNT
DB90 34                 INR     M       ;INCREMENT IT
DB91 E1         KEY5:   POP     H
                ;
DB92 F1         KEY2:   POP     PSW
DB93 FB                 EI
DB94 C9                 RET
                ;
DB95 CD9BDB     KEY6:   CALL    RSETP   ;RESET POINTERS
DB98 C391DB             JMP     KEY5
                ;
                ; RESET BOTH POINTERS TO START
                ;
DB9B 210000     RSETP:  LXI     H,0
DB9E 2202F4             SHLD    CCNT    ;ZERO BOTH
DBA1 2106F4             LXI     H,BUFF
DBA4 2204F4             SHLD    KPNTR   ;RESET PNTRS
DBA7 2200F4             SHLD    CPNTR
DBAA C9                 RET
                        ENDIF           ;INTERRUPTS
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; CHECK FOR CONSOLE INPUT READY
                ;
DBAB 3A0300     CONST:  LDA     IOBYTE
DBAE E602               ANI     2
DBB0 C2CBDB             JNZ     LISST   ;LIST
                ;
                ;*************************************
                        IF      INTRM
                ;
                ; CHECK INPUT BUFFER RATHER THAN KEYBOARD
                ;
DBB3 E5                 PUSH    H
DBB4 2A02F4             LHLD    CCNT    ;BOTH COUNTS
DBB7 7C                 MOV     A,H
DBB8 95                 SUB     L       ;DIFFERENCE
DBB9 E1                 POP     H
DBBA C8                 RZ              ;NO INPUT
                ;
DBBB E5                 PUSH    H
DBBC 2A00F4             LHLD    CPNTR   ;COMPUTER PNTR
DBBF 7E                 MOV     A,M     ;NEXT CHAR
DBC0 E1                 POP     H
DBC1 FE03               CPI     CTRC    ;^C?
DBC3 CAC8DB             JZ      QUIT    ;YES, QUIT
                ;
                ; MAKE CP/M THINK THERE IS NO INPUT
                ; SO SCROLLING WON'T BE ABORTED
                ;
```

```
DBC6 AF                 XRA     A       ;GET ZERO
DBC7 C9                 RET
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                        ELSE            ;NOT INTERRUPTS
                        IN      CSTAT   ;GET STATUS
                        ANI     CIMSK
                        RZ              ;NOT READY
                        ENDIF           ;INTERRUPTS
                ;
DBC8 3EFF       QUIT:   MVI     A,TRUE
DBCA C9                 RET             ;INPUT READY
                ;
                ; LIST READY FOR CONSOLE
                ;
DBCB DB12       LISST:  IN      LSTAT
DBCD E601               ANI     LIMSK
DBCF C8                 RZ              ;NOT READY
DBD0 3EFF               MVI     A,TRUE
DBD2 C9                 RET             ;READY
                ;
                ; CONSOLE INPUT
                ;
DBD3 3A0300     CONIN:  LDA     IOBYTE
DBD6 E602               ANI     2
DBD8 C206DC             JNZ     LIN     ;LIST INPUT
                ;
                ;**********************************
                        IF      INTRM   ;INTERRUPTS
                ;
                ; GET INPUT FROM KEYBOARD BUFFER
                ; INSTEAD OF FROM CONSOLE
                ;
DBDB E5                 PUSH    H
DBDC 2A02F4     CIN3:   LHLD    CCNT    ;BOTH COUNTS
DBDF 7C                 MOV     A,H
DBE0 95                 SUB     L       ;SAME?
DBE1 CADCDB             JZ      CIN3    ;KEEP TRYING
DBE4 F3                 DI              ;HOLD OFF
DBE5 2102F4             LXI     H,CCNT  ;COMPUTER COUNT
DBE8 34                 INR     M       ;INCREMENT IT
DBE9 2A00F4             LHLD    CPNTR   ;COMPUTER PNTR
DBEC 7E                 MOV     A,M     ;GET BYTE
                ;
                ; RESET BOTH POINTERS IF CARR RET FOUND
                ;
DBED 23                 INX     H       ;BUMP POINTER
DBEE 2200F4             SHLD    CPNTR   ;SAVE IT
DBF1 FE0D               CPI     CR      ;CARRIAGE RET?
DBF3 C203DC             JNZ     CIN4    ;NO
DBF6 2A02F4             LHLD    CCNT    ;GET BOTH COUNTS
DBF9 7C                 MOV     A,H
DBFA 95                 SUB     L       ;DIFFERENCE
DBFB C201DC             JNZ     CIN5    ;NOT SAME
                ;
                ; RESET BOTH POINTERS TO ZERO
                ;
```

```
DBFE CD9BDB              CALL    RSETP
DC01 3E0D      CIN5:     MVI     A,CR      ;RESTORE CR
DC03 E1        CIN4:     POP     H
DC04 FB                  EI                ;READY FOR MORE
DC05 C9                  RET
               ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                         ELSE              ;NO INTERRUPTS
               ;
               CIN2:     IN      CSTAT     ;CHECK STATUS
                         ANI     CIMSK
                         JZ      CIN2
                         IN      CDATA     ;GET DATA
                         ANI     7FH       ;MASK PARITY
                         RET
                         ENDIF             ;INTRM
               ;
               ; CONSOLE INPUT FROM LIST
               ;
DC06 DB12      LIN:      IN      LSTAT
DC08 E601                ANI     LIMSK
DC0A CA06DC              JZ      LIN
DC0D DB13                IN      LDATA
DC0F E67F                ANI     7FH
DC11 C9                  RET
               ;
               ; CONSOLE OUTPUT
               ;
DC12 3A0300    CONOUT:   LDA     IOBYTE    ;WHERE?
DC15 E603                ANI     3
DC17 B7                  ORA     A
DC18 E22EDC              JPO     LIST
               ;
DC1B DB10      CONW:     IN      CSTAT     ;CHECK STATUS
DC1D E602                ANI     COMSK
DC1F CA1BDC              JZ      CONW
DC22 79                  MOV     A,C       ;GET BYTE
DC23 D311                OUT     CDATA     ;SEND IT
DC25 C9                  RET
               ;
               ; LIST OUTPUT
               ;
DC26 3A0300    LOUT:     LDA     IOBYTE
DC29 E640                ANI     40H       ;BIT 6
DC2B C21BDC              JNZ     CONW      ;CONSOLE OUT
               ;
DC2E DB12      LIST:     IN      LSTAT     ;CHECK STATUS
DC30 E602                ANI     LOMSK
DC32 CA2EDC              JZ      LIST
DC35 79                  MOV     A,C       ;GET BYTE
DC36 D313                OUT     LDATA     ;SEND IT
               ;
                         IF      LNULL > 0
               ;
               ; NULLS FOR LIST DEVICE
               ;
```

```
                        ANI     7FH
                        CPI     CR
                        JNZ     FORM
                        MVI     C,0
                        CALL    LIST    ;1 NULL
                        CALL    LIST    ;2 NULLS
                        CALL    LIST    ;3 NULLS
                        JMP     LIST    ;4 NULLS
                        ENDIF
                ;
DC38 FE0C       FORM:   CPI     FFEED   ;FORMFEED?
DC3A C0                 RNZ             ;NO
                ;
                ; EMULATE FORMFEED WITH 9 LINES
                ;
DC3B C5                 PUSH    B
DC3C 010A09             LXI     B,900H+LF
DC3F CD2EDC     LSKIP:  CALL    LIST
DC42 05                 DCR     B
DC43 C23FDC             JNZ     LSKIP
DC46 C1                 POP     B
DC47 C9                 RET
                ;
                ; PUNCH OUTPUT SENT TO MODEM
                ;
DC48 79         PUNCH:  MOV     A,C     ;GET BYTE
DC49 E67F               ANI     7FH
DC4B B7                 ORA     A       ;NULL?
DC4C C8                 RZ              ;DON'T SEND
DC4D FE0A               CPI     LF
DC4F C8                 RZ              ;SKIP LINEFEED
DC50 CD69DC             CALL    MOUT    ;SEND
DC53 FE0D               CPI     CR
DC55 CA5EDC             JZ      MODCR   ;WAIT FOR CR
DC58 CD74DC             CALL    MIN     ;MODEM INPUT
DC5B D311               OUT     CDATA   ;SEND TO CONSOLE
DC5D C9                 RET
                ;
                ; SEND <CR> TO MODEM, WAIT FOR ONE BACK
                ;
DC5E CD74DC     MODCR:  CALL    MIN
DC61 D311               OUT     CDATA   ;TO CONSOLE
DC63 FE0D               CPI     CR
DC65 C25EDC             JNZ     MODCR   ;KEEP TRYING
DC68 C9                 RET
                ;
                ; MODEM OUTPUT
                ;
DC69 DB14       MOUT:   IN      MSTAT   ;CHECK STATUS
DC6B E680               ANI     MOMSK
DC6D CA69DC             JZ      MOUT
DC70 79                 MOV     A,C     ;GET BYTE
DC71 D315               OUT     MDATA   ;SEND IT
DC73 C9                 RET
                ;
                ; MODEM INPUT
                ;
```

```
DC74 DB14       MIN:    IN      MSTAT   ;CHECK STATUS
DC76 E640               ANI     MIMSK
DC78 CA74DC             JZ      MIN
DC7B DB15               IN      MDATA   ;GET BYTE
DC7D E67F               ANI     7FH     ;MASK PARITY
DC7F C9                 RET
                ;
DC80 31322D             DB      '1-28-80'  ;VERSION
                ;
DC88                    END


symbol table

00C5 ACONT      00C4 ADATA      00C7 BCONT      00C6 BDATA
D600 BIOS       F400 BUFFER     F406 BUFF       F402 CCNT
0011 CDATA      0001 CIMSK      DBDC CIN3       DC03 CIN4
DC01 CIN5       0002 COMSK      DBD3 CONIN      DC12 CONOUT
DBAB CONST      DC1B CONW       F400 CPNTR      000D CR
0010 CSTAT      0003 CTRC       0004 CTRD       0011 CTRQ
0013 CTRS       FFFF DOUBLE     0000 FALSE      000C FFEED
DC38 FORM       DB15 INIT       FFFF INTRM      0003 IOBYTE
F403 KCNT       DB92 KEY2       DB7F KEY3       DB6C KEY4
DB91 KEY5       DB95 KEY6       DB5B KEYBD      F404 KPNTR
0013 LDATA      0005 LEV        000A LF         0001 LIMSK
DC06 LIN        DBCB LISST      DC2E LIST       0000 LNULL
0002 LOMSK      DC26 LOUT       DC3F LSKIP      0012 LSTAT
0015 MDATA      0040 MIMSK      DC74 MIN        DC5E MODCR
0080 MOMSK      DC69 MOUT       0036 MSIZE      0014 MSTAT
4900 OFFSET     DC48 PUNCH      DBC8 QUIT       DB9B RSETP
DB00 START      0095 STOP       FFFF TRUE       DB00 USER
```

entry is read with an IN instruction. The byte is then placed into the keyboard buffer and the buffer pointer and buffer count are both incremented. The interrupt flip-flop is enabled with an EI instruction, then the computer returns to its previous task.

When the CPU needs another byte, it gets it from the keyboard buffer in memory, rather than from the keyboard itself. The instructions starting at subroutine CONIN perform this step. The separate buffer pointer and buffer count, maintained for the CPU, are both incremented.

The interrupt-driven keyboard can be utilized with most of the CP/M systems programs. For example, if a BASIC interpreter has been loaded and a source program has been entered, then the source program can first be listed, then executed by typing the following two lines.

```
LIST
RUN
```

The second command can be given immediately following the first, even though the first task has not been completed. The second command will not be displayed on the console, however, until the completion of the first task. Therefore, the operator must type carefully.

SCROLL CONTROL AND TASK ABORTION

Data can appear (scroll) too rapidly on a high-speed video screen. With the usual CP/M arrangement, the user can type a Control-S to freeze the video display. Typing any other character will cause scrolling to resume. The interrupt-driven routine given in Listing 4.1 incorporates its own scroll control. Typing a Control-S freezes the screen, just as with the usual CP/M setup. However, scrolling can only be resumed by typing a Control-Q. The two commands, Control-S and Control-Q, are treated distinctly; they are not placed into the input buffer, but are acted upon immediately.

CP/M tasks are normally aborted by typing any keyboard character. On the other hand, a Control-C is required in Microsoft BASIC, and a Control-E is used by Xitan BASIC for aborting the current task. This protocol has been altered so that characters can be entered into the keyboard buffer during a scroll operation. Nevertheless, it may be desirable to abort a task.

If no characters have been typed ahead, that is, if the computer is executing the latest command, then a Control-C command will abort the current operation. Alternatively, if there are characters waiting in the console-input buffer, then these must be flushed out by typing a Control-D. At this point, a Control-C can be typed to abort the task. This arrangement will work with most programs, including Microsoft BASIC and Tarbell BASIC. If you use Xitan BASIC, then you must change the abort command character in the interface routine from a Control-C to a Control-E.

An additional alteration is necessary for the Word-Master text editor. First of all, Word-Master buffers the keyboard buffer using software routines. Consequently, a hardware-interrupt system is unnecessary. Secondly, Word-Master uses Control-C and Control-D for system commands. Control-C is used to display the next screen and Control-D is used to move the cursor to the next word. If you want to use hardware interrupts with Word-Master, you must change the Control-C and Control-D commands in either the interface routine or in Word-Master.

DATA TRANSMISSION BY TELEPHONE

The process of transmitting information between a peripheral and the computer may be simple or it may be complex. If the system console is wired into the computer, or if the computer itself is built into the console, then the integrity of the transmitted data is not likely to be much of a problem. It may be, however, that the console is connected to the computer through a telephone line. The computer may be located across town or across the country. In any case, connection through a telephone line complicates things.

In a typical telephone arrangement, the data is sent from the console by modulating an acoustical carrier for transmission over the telephone line. The conversion is performed by an electronic device called a *modem* (the name is an abbreviation for MODulator-DEModulator). Two modems are required, one at each end of the telephone line. One converts the transmitted

signal to telephone frequencies, the other converts the signal back to the original data.

The modem may also have an *acoustical coupler*. This allows a standard telephone headset to be pressed into two rubber-lined openings in the modem, making a direct connection between the modem and the telephone line unnecessary.

There will usually be two data carrier signals at different frequencies. This allows simultaneous two-way, or *full duplex*, operation. The computer can transmit data to the console on one carrier while the console is transmitting data to the computer on the other carrier.

A microcomputer can produce a more effective link between a console and a large main-frame computer, especially if a relatively slow modem is utilized. A program can be developed using the microcomputer's editor, then the resulting file can be automatically transmitted to the larger computer. A subroutine that can be used to link a microcomputer to a large computer is given in Listing 4.2. This routine can be readily incorporated into the system monitor introduced in Chapter 6.

Listing 4.2 Connection to a large computer

```
                        ;
                        ; CONNECT TO ANOTHER COMPUTER
                        ; THROUGH PHONE MODEM
                        ; (Z-80 CODE)
                        ;
0014                    DSTAT   EQU     14H         ;STATUS
0015                    DDATA   EQU     DSTAT+1
0040                    DIMSK   EQU     40H         ;INPUT MASK
0080                    DOMSK   EQU     80H         ;OUT MASK
                        ;
0004                    CTRD    EQU     4           ;^D, COPY
57A1                    TYFLG   EQU     STACK+1     ;COPY FLAG
                        ;
5BB3 AF                 DEC:    XOR     A           ;ZERO
5BB4 32 57A1                    LD      (TYFLG),A   ;RESET COPY
5BB7 DB 14              DECIN:  IN      A,(DSTAT)   ;READY?
5BB9 E6 40                      AND     DIMSK
5BBB 28 13                      JR      Z,ALTIN     ;NO
5BBD 3A 57A1                    LD      A,(TYFLG)   ;COPY FLAG
5BC0 B7                         OR      A           ;TO MEMORY?
5BC1 28 07                      JR      Z,DIN5      ;NO
5BC3 CD 5BF1                    CALL    DINPUT      ;GET BYTE
5BC6 77                         LD      (HL),A      ;TO MEMORY
5BC7 23                         INC     HL          ;POINTER
5BC8 18 03                      JR      DEC2
                        ;
5BCA CD 5BF1            DIN5:   CALL    DINPUT      ;GET BYTE
5BCD CD 5835            DEC2:   CALL    OUTT        ;TO CONSOLE
5BD0 CD 5827            ALTIN:  CALL    INSTAT      ;CONSOLE
5BD3 28 E2                      JR      Z,DECIN     ;NOT READY
5BD5 CD 581A                    CALL    INPUT2      ;CONSOLE
5BD8 FE 04              ALT2:   CP      CTRD        ;^D
5BDA 28 1A                      JR      Z,DCOPY     ;SET FLAG
```

```
5BDC CD 5BE1    ALT5:   CALL    DECOUT  ;TO DEC
5BDF 18 D6              JR      DECIN   ;DEC INPUT
                ;
                ; OUTPUT A BYTE TO DEC
                ;
5BE1 F5         DECOUT: PUSH    AF
5BE2 CD 5BEA            CALL    DORDY
5BE5 F1                 POP     AF
5BE6 D3 15              OUT     (DDATA),A
5BE8 18 CD              JR      DECIN   ;NEXT
                ;
                ; DEC INPUT READY
                ;
5BEA DB 14      DORDY:  IN      A,(DSTAT)
5BEC E6 80              AND     DOMSK
5BEE 28 F1              JR      Z,DECOUT
5BF0 C9                 RET
                ;
                ; INPUT FROM DEC MODEM
                ;
5BF1 DB 15      DINPUT: IN      A,(DDATA)
5BF3 E6 7F              AND     DEL     ;MASK PARITY
5BF5 C9                 RET
                ;
                ; SET DEC COPY FLAG, START COPYING
                ; INTO MEMORY AT 100 HEX
                ;
5BF6 21 0100    DCOPY:  LD      HL,100H
5BF9 3E 01              LD      A,1
5BFB 32 57A1            LD      (TYFLG),A
5BFE 18 B7              JR      DECIN
                ;
                        END     START
```

PARITY CHECKING

Parity checking provides a method of monitoring the integrity of data transmission. While there are several different schemes for digitally encoding the common characters, the ASCII method is frequently used for microcomputers. The ASCII code, shown in Appendix A, requires only seven bits for each character. Since each byte of data contains eight bits, there is one bit available for use as a check bit.

Consider the 7-bit pattern for the ASCII characters 2 and 3.

ASCII 2	011 0010
ASCII 3	011 0011

The value of 2 is encoded with four logical zero bits and three logical 1 bits. The value of 3 is encoded with three logical zero bits and four logical 1 bits. A parity check can be obtained by including an additional bit on the left (high-order) end.

There are two common methods of generating the parity bit. One encoding method is called *even parity*. In this case, a leading zero bit is added if there are an even number of logical ones among the other seven bits. On the other hand, the parity bit would be a logical 1 if there are an odd number of logical ones among the other seven bits. With even parity coding, the ASCII characters 2 and 3 would look like this.

```
2    1011 0010
3    0011 0011
```

Now the 8-bit representation of both the 2 and the 3 contains an even number of logical ones (and an even number of logical zeros).

An alternate approach is called *odd parity*. In this case, the operation is simply the inverse of even parity. The logic of the parity bit is chosen so that the resulting bit pattern contains an odd number of logical ones. Either even or odd parity encoding will provide a check on the integrity of the data transmission.

Suppose that during transmission of the character 2, the rightmost bit became inverted. The console sent the even-parity bit pattern

```
1011 0010
```

but the computer received the bit pattern

```
1011 0011
```

A parity check, performed at the computer, would be able to detect the fact that there was an error.

A typical console-input routine might look like this.

```
CONIN:  IN      CMASK   ;CHECK STATUS
        ANI     CIMSK   ;MASK FOR INPUT
        JZ      CONIN   ;LOOP UNTIL READY
        IN      CDATA   ;GET THE DATA
        ANI     7FH     ;REMOVE PARITY
        RET
```

The next to the last instruction in this subroutine performs a logical AND with 7F hex. This step is used to remove the high-order bit of the byte since it is not needed for ASCII data. Instead of ignoring this eighth bit, we could use it as a parity check. An input routine to perform a check for parity looks like the following list.

```
CONIN:  IN      CMASK   ;CHECK STATUS
        ANI     CIMSK   ;MASK FOR INPUT
        JZ      CONIN   ;LOOP UNTIL READY
        IN      CDATA   ;GET THE DATA
        ORA     A       ;SET PARITY FLAG
        JPO     PERROR  ;PARITY ERROR
        ANI     7FH     ;REMOVE PARITY
        RET
;
; PARITY-ERROR MESSAGE
;
PERROR: . . .
```

This routine is essentially the same as the one given immediately before, but after the data register has been read by the computer, the parity of the byte is determined.

The ORA A instruction performs a logical OR of the accumulator with itself. A logical OR of any byte with itself will not change the byte. However, it does affect the status flags in this case. After the OR operation, the parity flag will be set according to the parity of the accumulator. If the parity is found to be odd, then an error is present. The JPO instruction causes a jump to the parity-error routine in this case. However, if the parity is found to be even, then the byte in the accumulator does not contain a parity error.

Notice that a parity check will not detect an even number of bit errors in a byte. There may be two, four, or six errors, and the parity check will not detect an error. This is not likely to be a practical problem, however, since the likelihood of two errors is much less than the likelihood of single errors.

ASCII computer terminals usually have the ability to automatically transmit an eighth parity bit with the data. Furthermore, there will typically be a user-selectable switch for choosing either even or odd parity. There may also be the additional choices of always resetting or always setting the parity bit. The input routine can check for odd parity if the JPO instruction is changed to a JPE instruction.

There are much more sophisticated methods of checking for transmission errors. One of these is the checksum approach discussed in Chapter 9. With this method, the transmitted data are added together. At regular intervals, the sum, or its complement, is transmitted along with the data. When the data are decoded, the data are added up again and compared to the checksum.

The Hamming error-correction code is even better than the checksum method. It not only detects errors, but can also correct them. In the end, however, it is wise to find out why errors occur, and to take the appropriate action to correct the problem. A dirty tape head, for example, can produce errors. Cleaning the head is better than relying on an error-correction scheme.

CHAPTER FIVE

Macros

Sophisticated assemblers incorporate a macro processor. A macro is used to define a set of instructions which are associated with the macro name. Then whenever the macro name appears in the source program, the assembler substitutes the corresponding instructions. This is called a *macro expansion.*

Suppose that we want to interchange the contents of two memory locations with the following instructions.

```
LDA     FIRST   ;GET FIRST BYTE
PUSH    PSW     ;SAVE
LDA     SECOND  ;GET SECOND
STA     FIRST   ;PUT INTO FIRST
POP     PSW     ;GET FIRST
STA     SECOND  ;PUT INTO SECOND
```

This set of instructions can be defined in a macro called SWAP.

```
SWAP    MACRO           ;SWAP FIRST AND SECOND
        LDA     FIRST   ;GET FIRST BYTE
        PUSH    PSW     ;SAVE
        LDA     SECOND  ;GET SECOND
        STA     FIRST   ;PUT INTO FIRST
        POP     PSW     ;GET FIRST
        STA     SECOND  ;PUT INTO SECOND
        ENDM
```

The macro definition is placed near the top of the assembler source program. The first line defines the macro name; the last line terminates the definition. The name SWAP can now be used like an operation code. it is placed in the source program whenever the corresponding instructions are needed. When the assembler encounters the name SWAP, it substitutes the desired

instructions. The final binary code generated by the assembler is the same as it would be if the instructions had originally been entered into the source program.

Each time the macro name SWAP appears in the source program, the same set of instructions will be generated and the same two memory locations will be interchanged. The SWAP macro becomes more versatile if the memory locations can be changed. If the names of the memory locations are placed on the first line of the macro definition, they become dummy variables.

```
SWAP    MACRO   FIRST, SECOND
        LDA     FIRST     ;GET 1ST BYTE
        PUSH    PSW       ;SAVE
        LDA     SECOND    ;GET 2ND
        STA     FIRST     ;PUT INTO 1ST
        POP     PSW       ;GET 1ST
        STA     SECOND    ;PUT INTO 2ND
        ENDM
```

The actual parameters in the macro call are substituted for the dummy parameters at assembly time. The macro call

```
SWAP    HIGH, LOW
```

generates the assembly language instructions

```
LDA     HIGH      ;GET 1ST BYTE
PUSH    PSW       ;SAVE
LDA     LOW       ;GET 2ND
STA     HIGH      ;PUT INTO 1ST
POP     PSW       ;GET 1ST
STA     LOW       ;PUT INTO 2ND
```

The statement

```
SWAP    LEFT, RIGHT
```

will produce the instructions

```
LDA     LEFT      ;GET 1ST BYTE
PUSH    PSW       ;SAVE
LDA     RIGHT     ;GET 2ND
STA     LEFT      ;PUT INTO 1ST
POP     PSW       ;GET 1ST
STA     RIGHT     ;PUT INTO 2ND
```

The structure of macros can be much more complicated than the above examples. One macro can be nested inside another.

```
OUTER   MACRO
        . . .
        IF      FAST
INNER   MACRO
        . . .
        . . .
        ENDM            ;;INNER
        . . .
        ENDIF           ;;FAST
        . . .
        ENDM            ;;OUTER
```

Conditional assembly directives can be used to create different versions. Comments in the macro definition which begin with a single semicolon are reproduced in the macro expansion along with the op codes. But if the comments are preceded by two consecutive semicolons, then they will appear only in the macro definition, not in the macro expansion.

GENERATING THREE OUTPUT ROUTINES WITH ONE MACRO

A subroutine can be used whenever a set of instructions is needed at several places in a program. But there are times when a similar but different group of instructions is needed. A subroutine cannot be used in this case. Consider the three 8080 output routines that follow. The first sends a byte to the console, the second sends a byte to the list device, and the third sends a byte to the phone modem.

```
COT:    IN      CSTAT
        ANI     COMSK
        JZ      COT
        MOV     A,C
        OUT     CDATA
        RET
;
LOT:    IN      LSTAT
        ANI     LOMSK
        JZ      LOT
        MOV     A,C
        OUT     LDATA
        RET
;
MOT:    IN      MSTAT
        ANI     MOMSK
        JNZ     MOT
        MOV     A,C
        OUT     MDATA
        RET
```

The structure of these three routines is very similar. Each begins by reading the appropriate status register. Then a logical AND is performed to select the output-ready bit. Looping occurs until the peripheral is ready. The byte is moved from the C register into the accumulator and sent to the appropriate peripheral. Finally, a return instruction is executed.

These three routines are slightly different, hence they cannot be replaced by a single subroutine. However, since they have similar structure they can be generated with a macro. The macro definition looks like this.

```
OUTPUT  MACRO   ?S,?Z      ;OUTPUT ROUTINES
?S&OT:  IN      ?S&STAT    ;CHECK STATUS
        ANI     ?S&OMSK    ;MASK FOR OUTPUT
        J&?Z    ?S&OT      ;NOT READY
        MOV     A,C        ;GET BYTE
        OUT     ?S&DATA    ;SEND IT
        RET
        ENDM
```

It would appear near the beginning of the source program. The macro name chosen is OUTPUT and the two dummy arguments are ?S and ?Z. Dummy arguments can have the same form as any other identifier. A question mark was chosen as the first character so that the dummy arguments would be easier to find in the macro definition. You must be careful not to use register names such as A, B, H, or L for dummy arguments if these register names also appear in the macro.

Each of the three output routines is generated by a one-line macro call.

```
        OUTPUT  C,Z        ;CONSOLE OUTPUT
;
        OUTPUT  L,Z        ;LIST OUTPUT
;
        OUTPUT  M,NZ       ;MODEM OUTPUT
```

Each line includes the appropriate parameters. At assembly time, the real arguments replace the dummy arguments of the macro. The ampersand character (&) is a concatenation operator. It separates a dummy argument from additional text. The macro processor substitutes the real parameter for the dummy argument, then joins it to the rest of the text. By this means the expression ?S&OT becomes LOT if the real argument is the letter L.

Macro assemblers may give the user three options for the assembly listing:

1. Show the macro call, the generated source line, and the resultant hex code.
2. Show the macro call and the hex code.
3. Show only the macro call.

If option 1 is chosen, then the above three macro calls to OUTPUT will produce the following.

```
                    OUTPUT  MACRO   ?S,?Z     ;OUTPUT ROUTINES
                    ?S&OT:  IN      ?S&STAT   ;CHECK STATUS
                            ANI     ?S&OMSK   ;MASK FOR OUTPUT
                            J&?Z    ?S&OT     ;NOT READY
                            MOV     A,C       ;GET BYTE
                            OUT     ?S&DATA   ;SEND IT
                            RET
                            ENDM
                    ;
                            OUTPUT  C,Z       ;CONSOLE OUTPUT
4000+DB10           COT:    IN      CSTAT     ;CHECK STATUS
4002+E602                   ANI     COMSK     ;MASK FOR OUTPUT
4004+CA0040                 JZ      COT       ;NOT READY
4007+79                     MOV     A,C       ;GET BYTE
4008+D311                   OUT     CDATA     ;SEND IT
400A+C9                     RET
                    ;
                            OUTPUT  L,Z       ;LIST OUTPUT
400B+DB12           LOT:    IN      LSTAT     ;CHECK STATUS
400D+E602                   ANI     LOMSK     ;MASK FOR OUTPUT
400F+CA0B40                 JZ      LOT       ;NOT READY
4012+79                     MOV     A,C       ;GET BYTE
4013+D313                   OUT     LDATA     ;SEND IT
4015+C9                     RET
                    ;
                            OUTPUT  M,NZ      ;MODEM OUTPUT
4016+DB14           MOT:    IN      MSTAT     ;CHECK STATUS
4018+E680                   ANI     MOMSK     ;MASK FOR OUTPUT
401A+C21640                 JNZ     MOT       ;NOT READY
401D+79                     MOV     A,C       ;GET BYTE
401E+D315                   OUT     MDATA     ;SEND IT
4020+C9                     RET
```

The first argument in the macro, ?S, is replaced by the actual argument. This is the letter C in the first call, the letter L in the second call, and the letter M in the third call. The second argument is used to select a JZ or JNZ instruction for the third line of the macro expansion.

Some assemblers automatically remove the ampersand symbol from the resultant assembly listing. Others leave the symbol in place. In this latter case, the first line of the first routine would look like this.

```
C&OT:   IN      C&STAT  ;CHECK STATUS
```

But this is a matter of style. The actual machine code generated is the same in either case.

GENERATING Z-80 INSTRUCTIONS WITH AN 8080 ASSEMBLER

If you have a Z-80 CPU but an 8080 macro assembler, such as the Digital Research MAC, you can run all of the 8080 programs just as they are given in this book. You can also do the Z-80 programs by using macros to generate the Z-80 instructions. For some of the instructions, the regular Zilog mnemonic can be used. For other instructions a slightly different format is

necessary. Consider, for example, the Z-80 instruction that performs a two's complement on the accumulator. The Zilog mnemonic for this operation is NEG. A Z-80 assembler converts this mnemonic into the two hex bytes ED 44. With an 8080 macro assembler you can use the same mnemonic. Define the macro

```
NEG     MACRO   ; TWO'S COMPLEMENT
        DB      0EDH, 44H
        ENDM
```

Then, the macro call

```
NEG
```

is placed in the source program when the Z-80 NEG instruction is needed. The 8080 macro assembler will insert the desired hex bytes ED 44 at this point.

As another example, consider the Z-80 relative-jump instruction. This instruction can be implemented with a macro that uses the assembler's program counter, a dollar sign. The macro definition looks like this.

```
JR      ADDR    ;RELATIVE JUMP
        DB      18H, ADDR-$-1
        ENDM
```

The dummy parameter ADDR is the destination address of the jump. The macro call

```
JR      ERROR
```

will generate the correct Z-80 code. The first byte will be 18 hex. The second byte will be the required displacement for the jump.

The Z-80 instruction, DJNZ, can be generated in a similar way. This instruction decrements the B register and jumps relative to the address of the argument if the zero flag is not set. The macro definition is

```
DJNZ    MACRO   ADDR
        DB      10H,ADDR-$-1
        ENDM
```

and the macro call looks like

```
DJNZ    LOOP
```

This approach will work with most macro assemblers. There may be a problem, however, with the interpretation of the dollar sign. This symbol usually refers to the address of the beginning of the current instruction. But for some assemblers, it is interpreted as the address of the following instruction.

If your assembler uses the latter interpretation, you will have to change the macro accordingly. If in doubt, check the user manual.

Some Z-80 mnemonics are not compatible with the macro format. For example, the Z-80 instruction

```
PUSH    IX
```

cannot be generated with a macro called

```
PUSH    MACRO   REG
```

since PUSH is a regular 8080 mnemonic. One possibility is to name the macro PUSHIX instead.

```
PUSHIX  MACRO
        DB      0DDH,0E5H
        ENDM
```

Similar problems occur with the commands POP IX, ADD IX,BC, SUB (IX+dis), and SET. A format that is different from the Z-80 mnemonic must be chosen in each of these cases.

The Digital Research macro assembler has an added bonus. Frequently-used macros can be placed into a separate macro library and given the file extension of LIB. In fact, this assembler is supplied with a macro library called Z80.LIB that will generate all of the Z-80 instructions. The statement

```
MACLIB  Z80
```

is placed near the beginning of the regular source program. The assembler will then look in the file Z80.LIB for the required macros.

EMULATING Z-80 INSTRUCTIONS WITH AN 8080 CPU

The Z-80 CPU can execute many powerful instructions that are not available to the 8080. Some of these useful instructions are difficult to implement on an 8080, while others are simply combinations of regular 8080 instructions. The NEG instruction is one of the easiest to implement. The macro definition is

```
NEG     MACRO           ;8080 TWO'S COMPLEMENT
        CMA             ;;1'S COMPLEMENT
        INR     A       ;;2'S COMPLEMENT
        ENDM
```

Now, whenever a two's complement is needed, the macro call

```
NEG
```

is placed into the program. The assembler generates the required 8080 mnemonics

```
CMA
INR     A
```

Another useful Z-80 operation is the arithmetic shift. This operation shifts all bits of a register one position to the left. The high-order bit is moved into carry, that is, the carry flag is set to a 1 if bit 7 was originally a value of 1. The carry flag is reset to zero if bit 7 was zero. A value of zero is placed into the low-order bit (bit zero).

The 8080 instruction ADD A, which adds the value in the accumulator to itself, performs the arithmetic shift left. But for the 8080, this is the only register which can perform the shift. The Z-80 has an additional instruction which allows this operation to be performed on any of the general-purpose, 8-bit registers or on the memory byte referenced by HL, IX, or IY.

The following macro will generate a set of 8080 instructions for the arithmetic shift left operation.

```
SLA     MACRO   REG     ;SHIFT LEFT ARITH
        MOV     A,REG   ;;GET BYTE
        ADD     A       ;;SHIFT LEFT
        MOV     REG,A   ;;PUT BACK
        ENDM
```

The byte is first moved to the accumulator. The next step is to add the accumulator to itself. This doubling operation performs the needed shift into carry. Then the result is returned to the original register. The value in register C can be doubled by inserting the macro call

```
SLA     C
```

This macro must be used with caution, since the accumulator will be changed during use. But the byte originally in the accumulator cannot be saved with a PUSH PSW instruction. The problem is that the subsequent POP PSW command will overlay the flag register, so that the carry result of the shift will be lost. One solution is to save the accumulator in memory.

```
SLA     MACRO   REG     ;SHIFT LEFT ARITH
        STA     SAVE    ;;SAVE A
        MOV     A,REG   ;;GET BYTE
        ADD     A       ;;SHIFT LEFT
        MOV     REG,A   ;;PUT BACK
        LDA     SAVE    ;;RESTORE A
        ENDM
```

THE REPEAT MACROS

There are times when several lines of identical or nearly identical lines of code are needed. Three repeat macros, REPT, IRP, and IRPC are provided for this purpose. The repeat macros differ from the regular macros in that they are placed directly into the source program where they are needed. The macro definition is the macro call. In the simplest form, an instruction or group of instructions can be replicated. The expression

```
REPT    4
RAR
ENDM
```

will generate the four lines

```
RAR
RAR
RAR
RAR
```

By using the SET directive, this operation can become more versatile. The SET instruction is like an EQU except that the value can be redefined. The lines

```
ADDR    SET     8000H
        REPT    4
ADDR    SET     ADDR+3
        DW      ADDR
        ENDM
```

will generate the code corresponding to

```
DW      8003H
DW      8006H
DW      8009H
DW      800CH
```

Such a series could refer to jump vectors that are spaced three bytes apart.

The repeat macro, combined with the conditional-assembly directive, can generate the required number of nulls after a carriage-return, line-feed pair. This will give the printer time to return to the left margin. Some printers need no nulls, whereas others may need as many as six or seven. The source code could be

```
;
; OUTPUT TO LIST DEVICE
;
LOUT:   IN      LSTAT
        ANI     LOMSK
        JZ      LOUT
        MOV     A,C     ;GET DATA
        OUT     LDATA   ;SEND BYTE
```

```
;
        IF      NULLS > 0
        ANI     7FH         ;REMOVE PARITY
        CPI     CR          ;CARRIAGE RETURN?
        RNZ                 ;NO
        MVI     C,0         ;GET A NULL
;
        REPT    NULLS       ;HOW MANY?
        CALL    LOUT        ;SEND NULL
        ENDM                ;REPEAT MACRO
        ENDIF               ;NULLS
;
        RET
```

The first part is a typical output subroutine. A call is made with the byte in register C. When the output device is ready, the byte is moved from the C register to the accumulator. It is then sent to the printer. If no nulls are required, then the passage from

```
IF      NULLS > 0
```

to

```
ENDIF
```

is not assembled. On the other hand, if nulls are required, then this passage is assembled. If four nulls are needed, then the assembler will generate four lines of

```
CALL  LOUT        ;SEND NULL
```

The list output routine calls itself to produce the required nulls. The identifier called NULLS must be previously set to the necessary number of nulls.

There are two other repeat macros called IRP and IRPC. A set of one-character message routines can be generated by using the indefinite repeat macro IRPC. This example will introduce something called a programming trick. Some people think that it is a horrible example of programming. Others think it is very clever. Its purpose is to save two bytes of instruction each time it is used. In addition, less branching is required.

Suppose that we need five different message routines that each produce a single character. The instructions might look like

```
CHARC:  MVI     A,'C'
        JMP     OUTT
CHARM:  MVI     A,'M'
        JMP     OUTT
CHARR:  MVI     A,'R'
        JMP     OUTT
CHAR?:  MVI     A,'?'
        JMP     OUTT
CHAR$:  MVI     A,'$'
        JMP     OUTT
        . . .
OUTT:   <output routine>
```

If the B and C registers are not in use, we can shorten the above passage by replacing each line containing the instruction JMP OUTT with a line of DB 1.

```
CHARC:  MVI     A,'C'
        DB      1
CHARM:  MVI     A,'M'
        DB      1
CHARR:  MVI     A,'R'
        DB      1
CHAR?:  MVI     A,'?'
        DB      1
CHAR$:  MVI     A,'$'
;
OUTT:   . . .
```

Let's see how this works. Suppose that a branch is made to the label CHARC. The accumulator is loaded with an ASCII letter C. The next byte, a DB 1, looks like the start of an LXI B instruction. The following two bytes, corresponding to the MVI A,'M' instruction, will be interpreted as the argument for the LXI instruction. That is, they will be considered as data. The same will hold for the other occurrences of DB 1. By this means, we have effectively shortened the code. We no longer need the JMP statements. Caution: a disassembler is not likely to interpret this passage correctly. It looks like there are labels pointing into the middle of the LXI instructions. Notice that the second version has subroutine OUTT positioned directly under the CHAR$ routine, so that no JMP instruction is needed at this point.

The second version can be easily generated with the IRPC macro. Only five lines are needed in the source program.

```
        IRPC    X,CMPR?$
        DB      1       ;FAKE LXI B
CHAR&X: MVI     A,'&X'
        ENDM
```

The five different message routines are all generated with this single macro. One replication is made for each character of the second argument to IRPC.

PRINTING STRINGS WITH MACROS

Suppose that we want to send messages to the console from various points of a program. We could write a subroutine called SENDM for this purpose.

```
SENDM:  LDAX    D       ;GET CHAR
        ORA     A       ;ZERO?
        RZ              ;YES
        INX     D       ;POINTER
        MOV     C,A
        CALL    OUTT    ;SENT
        JMP     SENDM   ;NEXT
```

The address of the message is loaded into the DE register and subroutine SENDM is called.

```
LXI     D,MESS1
CALL    SENDM
```

Subroutine SENDM prints a message by sending each character to the output subroutine OUTT. When a binary zero, used to indicate the end of the message, is found, SENDM returns to the calling program.

We can simplify the sending of messages by using a macro called PRINT. At each point we write

```
PRINT   <CHECKSUM ERROR>
  .  .  .
PRINT   <END OF FILE>
  .  .  .
PRINT   <OUTPUT TO LIST?>
```

The macro called PRINT will generate the message given in the argument. The message is enclosed in angle brackets because the blanks are part of the argument.

If subroutine SENDM were placed into the macro body, then one copy of SENDM would be inserted for each occurrence of the PRINT statement. But we don't need more than one copy of SENDM. On the other hand, if we don't include SENDM in the macro, there may not be any copies at all. What we need is a mechanism for inserting one, and only one, copy of SENDM regardless of how many times we give the PRINT command.

The solution is to write a double macro—one nested inside the other. Both macros will be given the same name. Subroutine SENDM will be part of the outer macro which will be expanded only once. The layout looks like this.

```
PRINT   MACRO   <message>  ;OUTER MACRO
        .  .  .
        [define SENDM]
        .  .  .
PRINT   MACRO   <message>  ;INNER MACRO
        .  .  .
        [send message]
        .  .  .
        ENDM               ;INNER MACRO
        .  .  .
        ENDM               ;OUTER MACRO
```

The source program in Listing 5.1 demonstrates this technique. The outer macro PRINT has the argument ?TEXT, used for the first call to the macro. Subroutine SENDM is generated at this time. Additional macro calls to PRINT utilize the inner macro which has the argument ?TEXT2. Subroutine SENDM is not generated on these subsequent calls.

Listing 5.1. Source listing for a macro demonstration program.

```
;
PRINT   MACRO   ?TEXT
        LOCAL   AROUND
;
        JMP     AROUND  ;SENDM
;
; SUBROUTINE TO SEND A STRING TO
; THE CONSOLE. BINARY ZERO AT STRING END.
; D,E IS STRING POINTER.
;
SENDM:  LDAX    D       ;GET CHAR
        ORA     A       ;ZERO?
        RZ              ;YES
        INX     D       ;POINTER
        MOV     C,A
        CALL    OUTT    ;SENT
        JMP     SENDM   ;NEXT
;
AROUND:
;
; REDEFINE THE MACRO
;
PRINT   MACRO   ?TEXT2
        LOCAL   MESG,CONT
;
        PUSH    D       ;SAVE D,E
        LXI     D,MESG  ;POINT
        CALL    SENDM
        POP     D       ;RESTORE
        JMP     CONT    ;SKIP MESSAGE
;
MESG:
        DB      CR,LF,'&?TEXT2',0
;
CONT:   ENDM            ;INNER MACRO
        PRINT   <?TEXT>
        ENDM            ;OUTER MACRO
;
CSTAT   EQU     10H     ;CONSOLE STATUS
CDATA   EQU     CSTAT+1 ;CONSOLE DATA
CR      EQU     13      ;CARRIAGE RETURN
LF      EQU     10      ;LINE FEED
;
ORG     100H
START:
        PRINT   <CHECKSUM ERROR.>
;
        PRINT   <END OF FILE.>
;
        PRINT   <OUTPUT TO LIST?>
        JMP     0       ;RETURN TO CP/M
;
; SEND CHARACTER IN C TO THE CONSOLE
;
```

```
OUTT:   IN      CSTAT
        ANI     2
        JZ      OUTT
        MOV     A,C
        OUT     CDATA
        RET
;
        END
```

Subroutine SENDM is coded into the main flow of the program, that is, it is an inline routine. It is therefore necessary to jump around SENDM. Additionally, there must be a branch around each of the messages, since they too are coded inline. Labels for the required branches are uniquely generated in the macro by declaring the corresponding labels as LOCAL. The resulting assembly listing is given in Listing 5.2. The assembler places plus symbols between the address and the generated code of the assembly listing to designate those lines that were generated by macros. Thus, lines that contain plus symbols were not present in the original source listing.

Listing 5.2. Assembly listing for a macro demonstration program.

```
                ;
                PRINT   MACRO   ?TEXT
                        LOCAL   AROUND
                ;
                        JMP     AROUND  ;SENDM
                ;
                ; SUBROUTINE TO SEND A STRING TO
                ; THE CONSOLE. BINARY ZERO AT STRING END.
                ; D,E IS STRING POINTER.
                ;
                SENDM:  LDAX    D       ;GET CHAR
                        ORA     A       ;ZERO?
                        RZ              ;YES
                        INX     D       ;POINTER
                        MOV     C,A
                        CALL    OUTT    ;SENT
                        JMP     SENDM   ;NEXT
                ;
                AROUND:
                ;
                ; REDEFINE THE MACRO
                ;
                PRINT   MACRO   ?TEXT2
                        LOCAL   MESG,CONT
                ;
                        PUSH    D       ;SAVE D,E
                        LXI     D,MESG  ;POINT
                        CALL    SENDM
                        POP     D       ;RESTORE
                        JMP     CONT    ;SKIP MESSAGE
                ;
                MESG:
                        DB      CR,LF,'&?TEXT2',0
                ;
```

```
                    CONT:   ENDM            ;INNER MACRO
                            PRINT   <?TEXT>
                            ENDM            ;OUTER MACRO
                    ;
0010 =              CSTAT   EQU     10H     ;CONSOLE STATUS
0011 =              CDATA   EQU     CSTAT+1 ;CONSOLE DATA
000D =              CR      EQU     13      ;CARRIAGE RETURN
000A =              LF      EQU     10      ;LINE FEED
                    ;
0100                ORG     100H
                    START:
                            PRINT   <CHECKSUM ERROR.>
0100+C30E01                 JMP     ??0001  ;SENDM
0103+1A             SENDM:  LDAX    D       ;GET CHAR
0104+B7                     ORA     A       ;ZERO?
0105+C8                     RZ              ;YES
0106+13                     INX     D       ;POINTER
0107+4F                     MOV     C,A
0108+CD6501                 CALL    OUTT    ;SENT
010B+C30301                 JMP     SENDM   ;NEXT
010E+D5                     PUSH    D       ;SAVE D,E
010F+111901                 LXI     D,??0002  ;POINT
0112+CD0301                 CALL    SENDM
0115+D1                     POP     D       ;RESTORE
0116+C32B01                 JMP     ??0003  ;SKIP MESSAGE
0119+0D0A434845             DB      CR,LF,'CHECKSUM ERROR.',0
                    ;
                            PRINT   <END OF FILE.>
012B+D5                     PUSH    D       ;SAVE D,E
012C+113601                 LXI     D,??0004  ;POINT
012F+CD0301                 CALL    SENDM
0132+D1                     POP     D       ;RESTORE
0133+C34501                 JMP     ??0005  ;SKIP MESSAGE
0136+0D0A454E44             DB      CR,LF,'END OF FILE.',0
                    ;
                            PRINT   <OUTPUT TO LIST?>
0145+D5                     PUSH    D       ;SAVE D,E
0146+115001                 LXI     D,??0006  ;POINT
0149+CD0301                 CALL    SENDM
014C+D1                     POP     D       ;RESTORE
014D+C36201                 JMP     ??0007  ;SKIP MESSAGE
0150+0D0A4F5554             DB      CR,LF,'OUTPUT TO LIST?',0
0162 C30000                 JMP     0       ;RETURN TO CP/M
                    ;
                    ; SEND CHARACTER IN C TO THE CONSOLE
                    ;
0165 DB10           OUTT:   IN      CSTAT
0167 E602                   ANI     2
0169 CA6501                 JZ      OUTT
016C 79                     MOV     A,C
016D D311                   OUT     CDATA
016F C9                     RET
                    ;
0170                        END
```

If you are familiar with the operation of your assembler, type up the demonstration program and try it out. Branch to the beginning and three messages will appear at the console.

```
Checksum error
End of file
Output to list?
```

Assembler operation will be considered in the next chapter.

By constructing increasingly complicated macros, it is possible to develop some of the structure that is characteristic of higher-level languages such as Pascal. The common loop constructions

```
REPEAT
.  .  .
.  .  .
UNTIL   <condition true>
```

and

```
LOOP
.  .  .
EXITIF  <condition true>
.  .  .
ENDLOOP
```

can be realized with macros called REPEAT, UNTIL, and so on. The arguments to UNTIL and EXITIF will consist of three terms. The first and third will be numeric values. The middle term will represent a logical operation such as EQUALS or LESS THAN. The spelling of the logical operators in this case will have to be unusual, since the normal spellings

```
EQ
LT
GE
```

are already utilized by the macro assembler. Macros for all of the common structures are available commercially. Also, source programs for structured macros of this type may be given in the instruction manual for your macro assembler.

CHAPTER SIX

Development of a System Monitor

The best way to learn assembly language programming is to actually do it. Consequently, in this chapter you will develop a small but very powerful utility program called a *monitor*. There are many useful things that can be done with the monitor. There is a command to examine memory and another to change it. Other commands deal with memory blocks. These allow you to move a block from one location to another. Some of the features will duplicate those found in other programs, but other features, such as a search routine and a memory test routine, will be unique.

You will not program the entire monitor at one time. Instead, you will start with just the bare essentials. You will check the monitor after each major change to ensure that the new features have been added correctly. With this so-called top-down method, any error that develops is likely to be found in the most recently added instructions. As new features are incorporated, the monitor will increase in size until it reaches 1K bytes. This is a size that can be easily programmed into a single ROM. The monitor will then be immediately available as soon as the computer is turned on.

An editor and an assembler are required for the development of the monitor. In addition, a debugger will be helpful if you have problems along the way. Each phase of the development will require the same sequence of steps.

1. Generate an assembly language source file with the editor.
2. Assemble the source program to produce an object file.
3. Compare the hex code from your assembly listing to the listing given in this chapter.
4. Load the object program into memory.
5. Branch to the monitor and try it out.

The assembly listings given in this chapter are written with 8080 mnemonics. You can use an 8080 assembler for these programs whether you have an 8080 or a Z-80 CPU. The resulting code will run on both an 8080 and a Z-80 CPU. If you have only a Z-80 assembler, you will have to change the mnemonics. The cross-reference between the 8080 and Z-80 mnemonics, given in Appendix G, can be used to find the corresponding instructions. Alternately, you can define the 8080 mnemonics as macros.

PROGRAM DEVELOPMENT DETAILS

This section describes the details of program development. Skip to the next section if you are familiar with the operation of your editor and assembler. An editor is needed to create and alter the assembly language source file. If you have CP/M, you will have an editor called ED. Other editors, such as ED-80, EDIT80, and Word-Master, are separately available.

The session begins by giving the name of the editor and the name of the source program. The following discussion assumes that you have CP/M. If you have some other operating system, the approach will be similar, but the details may differ. Put the CP/M system diskette in drive A and a working diskette in drive B if you have more than one drive. Go to drive B with the command

```
A>B:
```

The response will be

```
B>
```

Type the name of the editor followed by the name of the monitor source program. The command line might look like this.

```
B>A:ED  MON1.ASM
```

for the first version. The digit 1 in the filename refers to the version number. The file type is ASM for the Digital Research assemblers ASM and MAC. The file type should be chosen as MAC, however, if the Microsoft assembler is used.

As you type the source program, be careful to include only the instructions and the comments shown in Listing 6.1A. Do not type the resulting hex code that is also given at the beginning of each line. For example, the line that defines the parameter TOP, on the first page of the listing, should be typed as

```
TOP     EQU     24      ;MEMORY TOP, K BYTES
```

rather than as:

```
0018 =  TOP     EQU     24      ;MEMORY TOP, K BYTES
```

Type a Control-I or tab to automatically generate the blank spaces between symbols.

Most of the assembly language symbols have five or fewer characters. This is acceptable to many assemblers. However, if your assembler only allows names to have a maximum of five characters, then several symbols will have to be shortened. The TITLE directive, on the first line, is another potential problem. The CP/M version is shown. The apostrophes should be removed if the Microsoft assembler is utilized. If the TITLE directive is not available on your assembler, place a semicolon at the beginning of this first line to convert it to a comment.

VERSION 1: THE INPUT AND OUTPUT ROUTINES

Refer to Listing 6.1A. This version will contain only the input and output routines. Generate an assembler source file with the system editor. The following variables will have to be tailored to your particular system.

TOP	(top of usable memory, decimal K)
HOME	(where to return when done)
CSTAT	(console input status address)
CDATA	(console input data address)
CSTATO	(console output status address)
CDATAO	(console output data address)
INMSK	(input-ready mask)
OMSK	(output-ready mask)
BACKUP	(console backspace character)

Normally, CSTATO will be the same as CSTAT, and CDATAO will be the same as CDATA. But if your console input address is different from your console output address, then each can be separately defined. Furthermore, the address of CDATA will typically have a value one larger or smaller than that of CSTAT.

Listing 6.1A. The beginning of a system monitor.

```
                TITLE   '8080 system monitor, ver 1'
                ;
                ; (put today's date here)
                ;
0018 =          TOP     EQU     24      ;MEMORY TOP, K BYTES
5800 =          ORGIN   EQU     (TOP-2)*1024  ;PROGRAM START
                ;
5800            ORG     ORGIN
                ;
                ;
0000 =          HOME    EQU     0       ;ABORT (VER 1-2)
                ;HOME   EQU     ORGIN   ;ABORT ADDRESS
0031 =          VERS    EQU     '1'     ;VERSION NUMBER
57A0 =          STACK   EQU     ORGIN-60H
0010 =          CSTAT   EQU     10H     ;CONSOLE STATUS
0011 =          CDATA   EQU     CSTAT+1 ;CONSOLE DATA
0010 =          CSTATO  EQU     CSTAT   ;CON OUT STATUS
0011 =          CDATAO  EQU     CSTATO+1 ;OUT DATA
0001 =          INMSK   EQU     1       ;INPUT MASK
0002 =          OMSK    EQU     2       ;OUTPUT MASK
                ;
57A0 =          PORTN   EQU     STACK   ;3 BYTES I/O
57A3 =          IBUFP   EQU     STACK+3 ;BUFFER POINTER
57A5 =          IBUFC   EQU     IBUFP+2 ;BUFFER COUNT
57A6 =          IBUFF   EQU     IBUFP+3 ;INPUT BUFFER
                ;
0008 =          CTRH    EQU     8       ;^H BACKSPACE
0009 =          TAB     EQU     9       ;^I
0011 =          CTRQ    EQU     17      ;^Q
0013 =          CTRS    EQU     19      ;^S
0018 =          CTRX    EQU     24      ;^X, ABORT
0008 =          BACKUP  EQU     CTRH    ;BACKUP CHAR
007F =          DEL     EQU     127     ;RUBOUT
001B =          ESC     EQU     27      ;ESCAPE
00F7 =          APOS    EQU     (39-'0') AND 0FFH
000D =          CR      EQU     13      ;CARRIAGE RET
000A =          LF      EQU     10      ;LINE FEED
00DB =          INC     EQU     0DBH    ;IN OP CODE
00D3 =          OUTC    EQU     0D3H    ;OUT OP CODE
00C9 =          RETC    EQU     0C9H    ;RET OP CODE
                ;
                START:
5800 C34A58             JMP     COLD    ;COLD START
5803 C35358     RESTRT: JMP     WARM    ;WARM START
                ;
                ; CONSOLE INPUT ROUTINE
                ;
5806 CD1658     INPUTT: CALL    INSTAT  ;CHECK STATUS
5809 CA0658             JZ      INPUTT  ;NOT READY ***
580C DB11       INPUT2: IN      CDATA   ;GET BYTE
580E E67F               ANI     DEL
5810 FE18               CPI     CTRX    ;ABORT?
5812 CA0000             JZ      HOME    ;YES
5815 C9                 RET
                ;
                ; GET CONSOLE-INPUT STATUS
                ;
```

```
5816 DB10       INSTAT: IN      CSTAT
5818 E601               ANI     INMSK
581A C9                 RET
                ;
                ; CONSOLE OUTPUT ROUTINE
                ;
581B F5         OUTT:   PUSH    PSW
581C CD1658     OUT2:   CALL    INSTAT  ;INPUT?
581F CA3558             JZ      OUT4    ;NO ***
5822 CD0C58             CALL    INPUT2  ;GET INPUT
5825 FE13               CPI     CTRS    ;FREEZE?
5827 C21C58             JNZ     OUT2    ;NO
                ;
                ; FREEZE OUTPUT UNTIL ^Q OR ^X
                ;
582A CD0658     OUT3:   CALL    INPUTT  ;INPUT?
582D FE11               CPI     CTRQ    ;RESUME?
582F C22A58             JNZ     OUT3    ;NO
5832 C31C58             JMP     OUT2
                ;
5835 DB10       OUT4:   IN      CSTATO  ;CHECK STATUS
5837 E602               ANI     OMSK
5839 CA1C58             JZ      OUT2    ;NOT READY ***
583C F1                 POP     PSW
583D D311               OUT     CDATAO  ;SEND DATA
583F C9                 RET
                ;
5840 0D0A       SIGNON: DB      CR,LF
5842 205665             DB      ' Ver '
5847 3100               DW      VERS
5849 00                 DB      0
                ;
                ; CONTINUATION OF COLD START
                ;
584A 31A057     COLD:   LXI     SP,STACK
584D 114058             LXI     D,SIGNON ;MESSAGE
5850 CDE258             CALL    SENDM   ;SEND IT
                ;
                ; WARM-START ENTRY
                ;
5853 215358     WARM:   LXI     H,WARM  ;RETURN HERE
5856 E5                 PUSH    H
5857 CDB658             CALL    CRLF    ;NEW LINE
585A CD7758             CALL    INPLN   ;CONSOLE LINE
585D CDCC58             CALL    GETCH   ;GET CHAR
5860 FE44               CPI     'D'     ;DUMP
5862 CA5358             JZ      WARM    ;(VER 1)
                ;       JZ      DUMP    ;HEX/ASCII (2)
5865 FE43               CPI     'C'     ;CALL
5867 CA5358             JZ      WARM    ;(VER 1-2)
                ;       JZ      CALLS   ;SUBROUTINE (3)
586A FE47               CPI     'G'     ;GO
586C CA5358             JZ      WARM    ;(VER 1-2)
                ;       JZ      GO      ;SOMEWHERE (3)
586F FE4C               CPI     'L'     ;LOAD
5871 CA5358             JZ      WARM    ;(VER 1-3)
                ;       JZ      LOAD    ;INTO MEMORY (4)
5874 C35358             JMP     WARM    ;TRY AGAIN
                ;
```

```
                        ; INPUT A LINE FROM CONSOLE AND PUT IT
                        ; INTO THE BUFFER. CARRIAGE RETURN ENDS
                        ; THE LINE. RUBOUT OR ^H CORRECTS LAST
                        ; LAST ENTRY. CONTROL-X RESTARTS LINE.
                        ; OTHER CONTROL CHARACTERS ARE IGNORED
                        ;
5877 3E3E       INPLN:  MVI     A,'>'   ;PROMPT
5879 CD1B58             CALL    OUTT
587C 21A657     INPL2:  LXI     H,IBUFF ;BUFFER ADDR
587F 22A357             SHLD    IBUFP   ;SAVE POINTER
5882 0E00               MVI     C,0     ;COUNT
5884 CD0658     INPLI:  CALL    INPUTT  ;CONSOLE CHAR
5887 FE20               CPI     ' '     ;CONTROL?
5889 DAA858             JC      INPLC   ;YES
588C FE7F               CPI     DEL     ;DELETE
588E CAC058             JZ      INPLB   ;YES
5891 FE5B               CPI     'Z'+1   ;UPPER CASE?
5893 DA9858             JC      INPL3   ;YES
5896 E65F               ANI     5FH     ;MAKE UPPER
5898 77         INPL3:  MOV     M,A     ;INTO BUFFER
5899 3E20               MVI     A,32    ;BUFFER SIZE
589B B9                 CMP     C       ;FULL?
589C CA8458             JZ      INPLI   ;YES, LOOP
589F 7E                 MOV     A,M     ;GET CHAR
58A0 23                 INX     H       ;INCR POINTER
58A1 0C                 INR     C       ;AND COUNT
58A2 CD1B58     INPLE:  CALL    OUTT    ;SHOW CHAR
58A5 C38458             JMP     INPLI   ;NEXT CHAR
                        ;
                        ; PROCESS CONTROL CHARACTER
                        ;
58A8 FE08       INPLC:  CPI     CTRH    ;^H?
58AA CAC058             JZ      INPLB   ;YES
58AD FE0D               CPI     CR      ;RETURN?
58AF C28458             JNZ     INPLI   ;NO, IGNORE
                        ;
                        ; END OF INPUT LINE
                        ;
58B2 79                 MOV     A,C     ;COUNT
58B3 32A557             STA     IBUFC   ;SAVE
                        ;
                        ; CARRIAGE-RETURN, LINE-FEED ROUTINE
                        ;
58B6 3E0D       CRLF:   MVI     A,CR
58B8 CD1B58             CALL    OUTT    ;SEND CR
58BB 3E0A               MVI     A,LF
58BD C31B58             JMP     OUTT    ;SEND LF
                        ;
                        ; DELETE PRIOR CHARACTER IF ANY
                        ;
58C0 79         INPLB:  MOV     A,C     ;CHAR COUNT
58C1 B7                 ORA     A       ;ZERO?
58C2 CA8458             JZ      INPLI   ;YES
58C5 2B                 DCX     H       ;BACK POINTER
58C6 0D                 DCR     C       ;AND COUNT
58C7 3E08               MVI     A,BACKUP ;CHARACTER
58C9 C3A258             JMP     INPLE   ;SEND
                        ;
                        ; GET A CHARACTER FROM CONSOLE BUFFER
                        ; SET CARRY IF EMPTY
```

```
                        ;
58CC E5         GETCH:  PUSH    H       ;SAVE REGS
58CD 2AA357             LHLD    IBUFP   ;GET POINTER
58D0 3AA557             LDA     IBUFC   ;AND COUNT
58D3 D601               SUI     1       ;DECR WITH CARRY
58D5 DAE058             JC      GETC4   ;NO MORE CHAR
58D8 32A557             STA     IBUFC   ;SAVE NEW COUNT
58DB 7E                 MOV     A,M     ;GET CHARACTER
58DC 23                 INX     H       ;INCR POINTER
58DD 22A357             SHLD    IBUFP   ;AND SAVE
58E0 E1         GETC4:  POP     H       ;RESTORE REGS
58E1 C9                 RET
                        ;
                        ; SEND ASCII MESSAGE UNTIL BINARY ZERO
                        ; IS FOUND. POINTER IS D,E
                        ;
58E2 1A         SENDM:  LDAX    D       ;GET BYTE
58E3 B7                 ORA     A       ;ZERO?
58E4 C8                 RZ              ;YES, DONE
58E5 CD1B58             CALL    OUTT    ;SEND IT
58E8 13                 INX     D       ;POINTER
58E9 C3E258             JMP     SENDM   ;NEXT
                        ;
58EC                    END
```

If you don't know the addresses of the console status and data registers and you are using the CP/M operating system, there is another approach you can take. You can use the I/O routines in the CP/M BIOS. The disadvantage of this approach is that CP/M must always be in place whenever the monitor is used. The BIOS entry address is given at memory address 1. The console status, input and output addresses are obtained by adding, respectively, 3, 6, and 9 to this address. The following I/O routines in Listing 6.1B can be substituted for the subroutines in Listing 6.1A starting with the label INPUTT and ending with the label OUT2. If this version is utilized, the addresses in the following sections will not agree with your assembly listings.

Listing 6.1B. Alternate I/O routines using CP/M BIOS.

```
                        ; CONSOLE INPUT ROUTINE USING CP/M BIOS
                        ;
                        INPUTT:
5806 E5         INPUT2: PUSH    H       ;SAVE REGISTERS
5807 D5                 PUSH    D
5808 C5                 PUSH    B
5809 211558             LXI     H,IN5   ;RETURN ADDRESS
580C E5                 PUSH    H       ;PUT ON STACK
580D 2A0100             LHLD    1       ;BIOS WARM START
5810 110600             LXI     D,6     ;OFFSET TO INPUT
5813 19                 DAD     D       ;ADD IN
5814 E9                 PCHL            ;CALL BIOS
5815 C1         IN5:    POP     B       ;RESTORE REGISTERS
5816 D1                 POP     D
5817 E1                 POP     H
5818 FE18               CPI     CTRX    ;ABORT?
581A CA0058             JZ      START   ;YES
581D C9                 RET
```

```
                ; GET CONSOLE-INPUT STATUS USING CP/M
                ;
581E E5         INSTAT: PUSH    H       ;SAVE REGISTERS
581F D5                 PUSH    D
5820 C5                 PUSH    B
5821 212D58             LXI     H,ST5   ;RETURN ADDRESS
5824 E5                 PUSH    H       ;PUT ON STACK
5825 2A0100             LHLD    1       ;BIOS ENTRY
5828 110300             LXI     D,3     ;OFFSET TO STATUS
582B 19                 DAD     D       ;ADD TO ADDR
582C E9                 PCHL            ;CALL BIOS
582D C1         ST5:    POP     B       ;RESTORE REGISTERS
582E D1                 POP     D
582F E1                 POP     H
5830 B7                 ORA     A
5831 C9                 RET
                ;
                ; CONSOLE OUTPUT ROUTINE USING CP/M BIOS
                ;
5832 F5         OUTT:   PUSH    PSW     ;SAVE BYTE
5833 CD1E58     OUT2:   CALL    INSTAT  ;INPUT?
5836 CA4C58             JZ      OUT4    ;NO
5839 CD0658             CALL    INPUT2  ;GET INPUT
583C FE13               CPI     CTRS    ;FREEZE?
583E C23358             JNZ     OUT2    ;NO
5841 CD0658     OUT3:   CALL    INPUTT  ;INPUT?
5844 FE11               CPI     CTRQ    ;RESUME?
5846 C24158             JNZ     OUT3    ;NO
5849 C33358             JMP     OUT2
                ;
584C F1         OUT4:   POP     PSW     ;GET BYTE
584D E5                 PUSH    H       ;SAVE REGISTERS
584E D5                 PUSH    D
584F C5                 PUSH    B
5850 4F                 MOV     C,A     ;MOVE BYTE
5851 F5                 PUSH    PSW
5852 215E58             LXI     H,OUT5  ;RETURN ADDRESS
5855 E5                 PUSH    H       ;PUT ON STACK
5856 2A0100             LHLD    1       ;BIOS ENTRY
5859 110900             LXI     D,9     ;OFFSET TO OUTPUT
585C 19                 DAD     D       ;ADD TOGETHER
585D E9                 PCHL            ;CALL BIOS
585E F1         OUT5:   POP     PSW     ;RESTORE REGISTERS
585F C1                 POP     B
5860 D1                 POP     D
5861 E1                 POP     H
5862 C9                 RET
```

Some of the constants such as PORTN will not be used at this time. However, their inclusion now will simplify things later. There are four occurrences of the dummy instruction

```
JZ      WARM
```

following the label WARM. Each is followed by an instruction that will be needed later. These latter instructions are preceded by a semicolon so that they will be treated as comments by the assembler.

There are some other matters that may need to be considered. One has to do with the sense of the input and output ready flags. There are three conditional jump instructions based on console-ready flags that display a logical 1 (active high) when ready. If your flags are inverted, that is, they present a logic zero when ready, then the three JZ commands must be changed to JNZ commands. These lines, indicated by three stars in the listing below, should be changed to

```
INPUTT: CALL    INSTAT  ;CHECK STATUS
        JNZ     INPUTT  ;NOT READY ***
        . . .
OUT2:   CALL    INSTAT  ;INPUT?
        JNZ     OUT4    ;NO  ***
        . . .
OUT4:   IN      CSTAT   ;CHECK STATUS
        ANI     OMSK
        JNZ     OUT2    ;NOT READY ***
```

The routine that corrects keyboard errors is programmed for a video console. If you have a console printer instead, change the backspace character to a slash.

```
BACKUP  EQU     '/'     ;CORRECTION
```

This will print a slash when an error is corrected. Otherwise the printer will back up during error correction, overstriking the old character with the new. You may also need to add some nulls after each carriage return. The problem here will be evidenced by missing characters at the beginning of each line. The solution is to place additional instructions in the subroutine called CRLF. Replace the last statement in this routine with the following.

```
CALL    OUTT    ;SEND LINE FEED
XRA     A       ;GET A ZERO
CALL    OUTT    ;SEND NULL
CALL    OUTT    ;AND ANOTHER
. . .
. . .   (one line for each null)
JMP     OUTT    ;LAST NULL
```

The rest of the program can be copied directly as it is. The abort command is a control-X. Initially, the abort address of HOME will be needed to leave the new monitor and return to your regular system. We will change this in version 3 when we will add a routine for branching to any memory address.

If you have a TRS-80 Model I, you won't have a control key. Therefore, you will have to change several of the commands shown in the listing. The original commands follow.

```
CTRH    (Control-H)
TAB     (Control-I)
CTRQ    (Control-Q)
CTRS    (Control-S)
CTRX    (Control-X)
DEL     (DEL/RUB)
ESC     (Escape)
```

After you have finished typing the program, exit from the editor and assemble the source program with the assembler. The command line might be

```
B>A:ASM  MON1
```

or

```
B>A:MAC  MON1
```

for the Digital Research assemblers. These two assemblers will produce two files.

```
MON1.ASM        (assembly listing)
MON1.HEX        (hex code)
```

In addition, MAC will produce a symbol table

```
MON1.SYM        (symbol table)
```

Inspect the hex code given in the assembly listing to see that it matches the corresponding instructions given in this chapter. These 8080 listings have all been generated with the Digital Research assembler MAC. This assembler displays the hex code for 16-bit operands in the usual reverse order. The low-order byte appears first followed by the high-order byte. Thus:

```
CD1B58   means  CALL  581B  and
C38458   means  JMP   5884
```

By contrast, the assembly listing produced by the Microsoft assembler reverses the usual order of the two bytes. The high-order byte is given first; this is followed by the low-order byte. In this case, the listing

```
CD 581B   means  CALL  581B  and
C3 5884   means  JMP   5884
```

The next step is to load the hex program into memory using the debugger. The CP/M command would be

```
B>A:DDT MON1.HEX    or
B>A:SID MON1.HEX
```

Now branch to the beginning of the monitor using the debugger G command.

```
G5800
```

The first thing that the monitor will do is display the version number on the first line and a prompt symbol of > underneath it.

Try out this first version by typing a series of letters and numbers. Each character that is typed should appear (echo) on the console. Try the correction keys. Typing either a control-H (backspace) or the RUB/DEL key should back up the cursor on a video terminal. Type a carriage return. The prompt symbol should appear at the beginning of the next line. If all of the features are working properly, type a control-X to return to your regular system. If something appears to be wrong, carefully compare your assembly listing with the one given in Listing 6.1A. *Don't proceed to the next version until the current one is working.*

VERSION 2: A MEMORY DISPLAY

A provision for examining the contents of memory will now be added. This routine is called a *memory dump*, or dump for short; it displays the contents of memory in both hex and ASCII notation. The dump feature is initiated with a command of D followed by the address limits in hexadecimal. For example, the statement

```
>D100 18F
```

will dump memory from address 100 to 18F hex. The first address (100 in this case) must immediately follow the letter D. A space is typed and then the second address (18F in this case) is entered. Leading zeros are unnecessary.

Each line will display 16 memory locations. The hexadecimal address of the first location will appear at the beginning of the line. Then the hexadecimal representation of the contents will follow, two characters per byte. These are arranged in four groups of four bytes. The ASCII representations of the data will be given at the end of the line if printable. Otherwise, a period is given. A dump of the first line of the monitor might look like this.

```
   >D5800 580F    (your command)
5800 C35C58C3 6558CD16 58CA0658 DB11E67F .\X.eX..X..X....
```

Use your system editor to make the necessary alterations and additions to version 1. First, change the version number at the beginning of the program.

```
VERS    EQU     '2'     ;VERSION NUMBER
```

Next, locate the instruction

```
JZ      DUMP
```

that follows the label WARM. Remove the semicolon at the beginning of this line. Also delete the line just before it that jumps to WARM. The region should now look like this.

```
CALL    GETCH
CPI     'D'     ;DUMP?
JZ      DUMP
. . .   . . .
```

The remaining instructions, shown in Listing 6.2, will be placed at the end of version 1, just preceding the END statement. It might be easier to delete the END statement, type in the new code, and then add a new END statement. The END statement is usually optional, anyway. One of the subroutines (READHL) will translate the dump limits from ASCII-encoded hexadecimal into binary. One routine gets both the start and the stop address (using READHL) then checks to see that the second address is larger than the first. If the second address is smaller than the first, then the task will be aborted. Subroutine OUTHEX will convert the binary data already in memory into ASCII-coded hex for output to the console. Subroutine TSTOP is used to determine when to terminate the dump process. Finally, an error routine (ERROR) will be needed in case an invalid character is entered by the user.

Listing 6.2. Memory display

```
                ; DUMP MEMORY IN HEXADECIMAL AND ASCII
                ;
58EC CD2D59     DUMP:   CALL    RDHLDE  ;RANGE
58EF CD8359     DUMP2:  CALL    CRHL    ;NEW LINE
58F2 4E         DUMP3:  MOV     C,M     ;GET BYTE
58F3 CD9359             CALL    OUTHX   ;PRINT
58F6 23                 INX     H       ;POINTER
58F7 7D                 MOV     A,L
58F8 E60F               ANI     0FH     ;LINE END?
58FA CA0559             JZ      DUMP4   ;YES, ASCII
58FD E603               ANI     3       ;SPACE
58FF CC8E59             CZ      OUTSP   ; 4 BYTES
5902 C3F258             JMP     DUMP3   ;NEXT HEX
5905 CD8E59     DUMP4:  CALL    OUTSP
5908 D5                 PUSH    D
5909 11F0FF             LXI     D,-10H  ;RESET LINE
590C 19                 DAD     D
590D D1                 POP     D
590E CD1D59     DUMP5:  CALL    PASCI   ;ASCII DUMP
5911 CDA759             CALL    TSTOP   ;DONE?
5914 7D                 MOV     A,L     ;NO
5915 E60F               ANI     0FH     ;LINE END?
5917 C20E59             JNZ     DUMP5   ;NO
591A C3EF58             JMP     DUMP2
```

```
                        ;
                        ; DISPLAY MEMORY BYTE IN ASCII IF
                        ; POSSIBLE, OTHERWISE GIVE DECIMAL PNT
                        ;
591D 7E         PASCI:  MOV     A,M     ;GET BYTE
591E FE7F               CPI     DEL     ;HIGH BIT ON?
5920 D22859             JNC     PASC2   ;YES
5923 FE20               CPI     ' '     ;CONTROL CHAR?
5925 D22A59             JNC     PASC3   ;NO
5928 3E2E       PASC2:  MVI     A,'.'   ;CHANGE TO DOT
592A C31B58     PASC3:  JMP     OUTT    ;SEND
                        ;
                        ; GET H,L AND D,E FROM CONSOLE
                        ; CHECK THAT D,E IS LARGER
                        ;
592D CD3859     RDHLDE: CALL    HHLDE
5930 7B         RDHLD2: MOV     A,E
5931 95                 SUB     L       ;E - L
5932 7A                 MOV     A,D
5933 9C                 SBB     H       ;D - H
5934 DA7B59             JC      ERROR   ;H,L BIGGER
5937 C9                 RET
                        ;
                        ; INPUT H,L AND D,E. SEE THAT
                        ; 2 ADDRESSES ARE ENTERED
                        ;
5938 CD4459     HHLDE:  CALL    READHL  ;H,L
593B DA7B59             JC      ERROR   ;ONLY 1 ADDR
593E EB                 XCHG            ;SAVE IN D,E
593F CD4459             CALL    READHL  ;D,E
5942 EB                 XCHG            ;PUT BACK
5943 C9                 RET
                        ;
                        ; INPUT H,L FROM CONSOLE
                        ;
5944 D5         READHL: PUSH    D
5945 C5                 PUSH    B       ;SAVE REGS
5946 210000             LXI     H,0     ;CLEAR
5949 CDCC58     RDHL2:  CALL    GETCH   ;GET CHAR
594C DA6859             JC      RDHL5   ;LINE END
594F CD6B59             CALL    NIB     ;TO BINARY
5952 DA5E59             JC      RDHL4   ;NOT HEX
5955 29                 DAD     H       ;TIMES 2
5956 29                 DAD     H       ;TIMES 4
5957 29                 DAD     H       ;TIMES 8
5958 29                 DAD     H       ;TIMES 16
5959 B5                 ORA     L       ;ADD NEW CHAR
595A 6F                 MOV     L,A
595B C34959             JMP     RDHL2   ;NEXT
                        ;
                        ; CHECK FOR BLANK AT END
                        ;
595E FEF7       RDHL4:  CPI     APOS    ;APOSTROPHE
5960 CA6859             JZ      RDHL5   ;ASCII INPUT
5963 FEF0               CPI     (' '-'0') AND 0FFH
5965 C27B59             JNZ     ERROR   ;NO
5968 C1         RDHL5:  POP     B
5969 D1                 POP     D       ;RESTORE
596A C9                 RET
```

```
                        ;
                        ; CONVERT ASCII CHARACTERS TO BINARY
                        ;
596B D630       NIB:    SUI     '0'         ;ASCII BIAS
596D D8                 RC                  ; < 0
596E FE17               CPI     'F'-'0'+1
5970 3F                 CMC                 ;INVERT
5971 D8                 RC                  ;ERROR, > F
5972 FE0A               CPI     10
5974 3F                 CMC                 ;INVERT
5975 D0                 RNC                 ;NUMBER 0-9
5976 D607               SUI     'A'-'9'-1
5978 FE0A               CPI     10          ;SKIP : TO
597A C9                 RET                 ;LETTER A-F
                        ;
                        ; PRINT ? ON IMPROPER INPUT
                        ;
597B 3E3F       ERROR:  MVI     A,'?'
597D CD1B58             CALL    OUTT
5980 C30058             JMP     START       ;TRY AGAIN
                        ;
                        ; START NEW LINE, GIVE ADDRESS
                        ;
5983 CDB658     CRHL:   CALL    CRLF        ;NEW LINE
                        ;
                        ; PRINT H,L IN HEX
                        ;
5986 4C         OUTHL:  MOV     C,H
5987 CD9359             CALL    OUTHX       ;H
598A 4D         OUTLL:  MOV     C,L
                        ;
                        ; OUTPUT HEX BYTE FROM C AND A SPACE
                        ;
598B CD9359     OUTHEX: CALL    OUTHX
                        ;
                        ; OUTPUT A SPACE
                        ;
598E 3E20       OUTSP:  MVI     A,' '
5990 C31B58             JMP     OUTT
                        ;
                        ; OUTPUT A HEX BYTE FROM C
                        ; BINARY TO ASCII HEX CONVERSION
                        ;
5993 79         OUTHX:  MOV     A,C
5994 1F                 RAR                 ;ROTATE
5995 1F                 RAR                 ; FOUR
5996 1F                 RAR                 ; BITS TO
5997 1F                 RAR                 ; RIGHT
5998 CD9C59             CALL    HEX1        ;UPPER CHAR
599B 79                 MOV     A,C         ;LOWER CHAR
599C E60F       HEX1:   ANI     0FH         ;TAKE 4 BITS
599E C690               ADI     90H
59A0 27                 DAA                 ;DAA TRICK
59A1 CE40               ACI     40H
59A3 27                 DAA
59A4 C31B58             JMP     OUTT
                        ;
                        ; CHECK FOR END, H,L MINUS D,E
                        ; INCREMENT H,L
```

```
                ;
59A7 23         TSTOP:  INX     H
59A8 7B                 MOV     A,E
59A9 95                 SUB     L       ; E - L
59AA 7A                 MOV     A,D
59AB 9C                 SBB     H       ; D - H
59AC D0                 RNC             ;NOT DONE
59AD E1                 POP     H       ;RAISE STACK
59AE C9                 RET
                ;
59B1                    END
```

Type up the new instructions, then, after you leave the editor, rename the new file. The CP/M command will be

```
REN  MON2.ASM=MON1.ASM
```

Rename the backup file to its original name.

```
REN  MON1.ASM=MON1.BAK
```

Assemble version 2 and load it into memory. Start it up by branching to the address of START. Again, the version number should be printed, and the prompt symbol should appear. Test the new feature by dumping a portion of the monitor.

```
>D5800 585F
```

Be sure to type a carriage return at the end of the line. Input errors can be corrected by typing a backspace or DEL. Check to see that the hex code displayed on the screen matches the assembly listing code. Most of the ASCII representation will be meaningless. But the section from 5842 to 585A hex will read

```
Ver 2
```

Now test the scroll-freeze commands. Dump a large section of memory.

```
>D0 1000
```

Type a control-S as the data are being displayed on the console. The console screen should freeze. Now type a control-Q. The screen should again resume displaying the data. The commands of Control-S and control-Q will alternately freeze and resume the scrolling.

Try the routine that checks for proper dump limits by typing a larger address first, then a smaller address.

```
D300 200
```

As a result of this improper input, a question mark should be printed. Then the prompt will appear on a new line. If everything is all right, return to your regular system by entering a control-X.

If version 2 does not perform satisfactorily, compare the hex code in your assembly listing with the values given in Listing 6.2 for the new code. Correct any errors, reassemble the program, and try it again.

VERSION 3: A CALL AND GO ROUTINE

Now that both hex-to-binary and binary-to-hex routines are available, we can easily include new features. A CALL routine and a GO routine will be added in version 3. These routines will allow you to branch to any address in memory. The GO command will be useful for testing subroutines. For this latter command, the monitor warm-start address (WARM) is on the stack when the call is made. A subroutine can be called with the C command. The execution of an RET instruction at the end of the subroutine will cause a return to the monitor.

First, change the version number to 3. Then find the instructions corresponding to the C and G commands after the label WARM. Remove the semicolons from the beginning of the lines that branch to CALLS and GO. Delete the prior lines that jump to WARM. The program should now look like

```
CPI     'C'     ;CALL?
JZ      CALLS
CPI     'G'     ;GO?
JZ      GO
```

The remaining lines of code (and some comments) are placed at the end of the source program just prior to the END statement. They are given in Listing 6.3.

```
Listing 6.3. A CALL and a GO routine.

                ; ROUTINE TO GO ANYWHERE IN MEMORY
                ; ADDRESS OF WARM IS ON STACK, SO A
                ; SIMPLE RET WILL RETURN TO THIS MONITOR
                ;
59AF E1         GO:     POP     H       ;RAISE STACK
59B0 CD4459     CALLS:  CALL    READHL  ;GET ADDRESS
59B3 E9                 PCHL            ;GO THERE
                ;
59B4                    END
```

Another important change should be made at this time. Since we can now branch to any place in memory with the GO command, we can change the abort command, control-X. Redefine HOME near the beginning of the source program so that an abort command of control-X will restart the monitor.

```
HOME    EQU     ORGIN   ;ABORT ADDRESS
```

This line was originally entered as a comment. Remove the semicolon at the beginning of the line and delete the previous line.

Assemble the new version and load it into memory. Branch to the monitor and try the dump routine as before. Try the CALL feature by calling the monitor itself.

```
>C5800
```

The cold-start message should appear. Now use the GO routine to return to your main system. If the GO address is zero, then no argument need follow the G command.

```
>G
```

VERSION 4: A MEMORY-LOAD ROUTINE

In version 2 we added a routine that could be used to inspect any memory location. A routine which can be used to change memory will now be added. Change the version number to 4. Locate the instruction

```
;       JZ      LOAD
```

following WARM. Remove the semicolon at the beginning of the line. Delete the original JZ WARM on the prior line. The program should now look like

```
CPI     'L'
JZ      LOAD
```

Add the load routines shown in Listing 6.4 to the end of the source program.

Listing 6.4. A memory-load routine.

```
                ; LOAD HEX OR ASCII CHAR INTO MEMORY
                ; FROM CONSOLE. CHECK TO SEE IF
                ; THE DATA ACTUALLY GOT THERE
                ; APOSTROPHE PRECEEDS ASCII CHAR
                ; CARRIAGE RETURN PASSES OVER LOCATION
                ;
59B4 CD4459     LOAD:   CALL    READHL  ;ADDRESS
59B7 CD8659     LOAD2:  CALL    OUTHL   ;PRINT IT
59BA CD1D59             CALL    PASCI   ;ASCII
59BD CD8E59             CALL    OUTSP
59C0 4E                 MOV     C,M     ;ORIG BYTE
59C1 CD8B59             CALL    OUTHEX  ;HEX
59C4 E5                 PUSH    H       ;SAVE PNTR
59C5 CD7C58             CALL    INPL2   ;INPUT
59C8 CD4459             CALL    READHL  ; BYTE
59CB 45                 MOV     B,L     ; TO B
```

```
59CC E1                 POP     H
59CD FEF7               CPI     APOS
59CF CADE59             JZ      LOAD6   ;ASCII INPUT
59D2 79                 MOV     A,C     ;HOW MANY?
59D3 B7                 ORA     A       ;NONE?
59D4 CADA59             JZ      LOAD3   ;YES
59D7 CDE559     LOAD4:  CALL    CHEKM   ;INTO MEMORY
59DA 23         LOAD3:  INX     H       ;POINTER
59DB C3B759             JMP     LOAD2
                ;
                ; LOAD ASCII CHARACTER
                ;
59DE CDCC58     LOAD6:  CALL    GETCH
59E1 47                 MOV     B,A
59E2 C3D759             JMP     LOAD4
                ;
                ; COPY BYTE FROM B TO MEMORY
                ; AND SEE THAT IT GOT THERE
                ;
59E5 70         CHEKM:  MOV     M,B     ;PUT IN MEM
59E6 7E                 MOV     A,M     ;GET BACK
59E7 B8                 CMP     B       ;SAME?
59E8 C8                 RZ              ;YES
59E9 C37B59             JMP     ERROR   ;BAD
                ;
59EC                    END
```

Assemble version 4 and compare the assembly listing of the new part to Listing 6.4. Load the new program and branch to the beginning. Recheck the dump command by examining the new code for the load routine

```
>D59B4 59EB
```

Now try the load command. Great care must be taken when typing the load address. This command will actually change the contents of memory, including the monitor itself.

Type the letter L, the hexadecimal address, and a carriage return. The response will be the address that was typed and the current contents of that memory location. The data are represented two ways: in ASCII and in hex. If the ASCII value is not a printable character, it is rendered as a period.

The displayed location can now be changed by typing the new value and a carriage return. The data can be entered in several ways. It can be in the form of one or two hex characters. If more than two characters are entered, only the last two are actually used. This allows you to correct an error by continuing to type. Errors can also be corrected with the backspace or the DEL/RUB key. A single ASCII character can be entered into memory by preceding it with an apostrophe.

As each new value and a carriage return is typed, the next address and the present data value will appear. In this way, a machine-language routine can be entered from the console. Of course, using an assembler is a more efficient way to generate a long program. But our load routine will be useful for making simple changes or for writing short routines.

The load command is terminated by typing a control-X (if you redefined HOME as ORGIN back in version 3). It is also terminated if you enter a nonhex character. Control then returns to the monitor. If the load command is used to revise existing code, another feature is useful. A carriage return is given without entering any data. The memory pointer then skips over the current location and the corresponding value is not changed.

After each revised byte is entered into memory, the monitor checks to see that the new value is correct. If an attempt is made to write into protected, nonexistent, or defective memory, the load process is terminated and a question mark is printed.

Try the load routine by entering the following five bytes into a convenient location such as 4000 hex.

```
3E  7 D3 XX C9
```

This sequence corresponds to the assembly language program

```
3E07    MVI     A,7
D3XX    OUT     XX
C9      RET
```

The value of XX is the console-data address (CDATA in the source program). Check the code with the dump command.

```
D4000 4004
```

Now use the CALL command to execute the routine

```
C4000
```

The console bell should sound and control will return to the command level of our monitor.

VERSION 5: USEFUL ENTRY POINTS

Changes to the first four versions were made for the most part by adding new instructions to the end of the existing program. For versions 5, 6, and 7, we are going to start the process over to some extent by inserting some new instructions in the middle of the existing program.

At the beginning of the monitor there are two jump instructions.

```
JMP     COLD
JMP     WARM
```

Entry points such as these are sometimes called *vectors*. The first jump to COLD is the initial, cold-start entry point into the monitor. Stack initialization and printing of the sign-on message occur at this time. But other

housekeeping chores, such as interface initialization, could be performed in this section. The second vector causes a jump to WARM, a restart entry point that does not alter the stack pointer.

We will now insert some additional vectors after these first two. The additional jumps will provide fixed entry points to useful subroutines in the monitor. These routines can then be easily called by other programs outside the monitor. Since these jump instructions are all at the beginning of the monitor, their addresses won't change when the monitor is altered. Furthermore, new vectors can be added to the end of the group without affecting those already present.

Place the five jump instructions shown in Listing 6.5 at the beginning of the monitor just after the first two (START and RESTRT).

Listing 6.5. Some useful entry points.

```
                ; VECTORS TO USEFUL ROUTINES
                ;
5806 C32A58     COUT:   JMP     OUTT    ;OUTPUT CHAR
5809 C31558     CIN:    JMP     INPUTT  ;INPUT CHAR
580C C3D058     INLN:   JMP     INPLN   ;INPUT LINE
580F C32559     GCHAR:  JMP     GETCH   ;GET CHAR
5812 C3EC59     OUTH:   JMP     OUTHX   ;BIN TO HEX
                ;
```

Reassemble the monitor, load it into memory, and try the DUMP, LOAD, and GO routines again to be sure that they still work. Now, when separate, external routines are written, they need not contain subroutines for console input, output, conversion of binary to hex, and so on.

A character can be displayed on the console by calling COUT with the character in the accumulator. A single console character is obtained by calling CIN. The byte is returned in the accumulator.

An entire line of characters can be easily obtained by calling the line-input entry INLN. As each character is typed, it is automatically printed on the console. The error-correction commands are available at this time. The backspace and DEL/RUB keys can be used to delete the previously typed character. A line is normally terminated with a carriage return. After the console-input buffer has been filled by a call to INLN, the GCHAR address can be called.

A character is returned in the accumulator for each call to GCHAR. When the input buffer has been exhausted, the carry flag is set. Typing a control-X will abort a routine and return control to the monitor. Therefore, it is not necessary to include an abort routine in separate, external programs.

The fifth new entry point will perform a conversion from binary to ASCII-coded hexadecimal. This will allow display of individual memory locations or any of the CPU registers. The byte to be converted is placed in the C register and the address of OUTH is called. The accumulator is also used by the conversion routine in this case, so it may be necessary to save the accumulator's original contents on the stack by using a PUSH instruction.