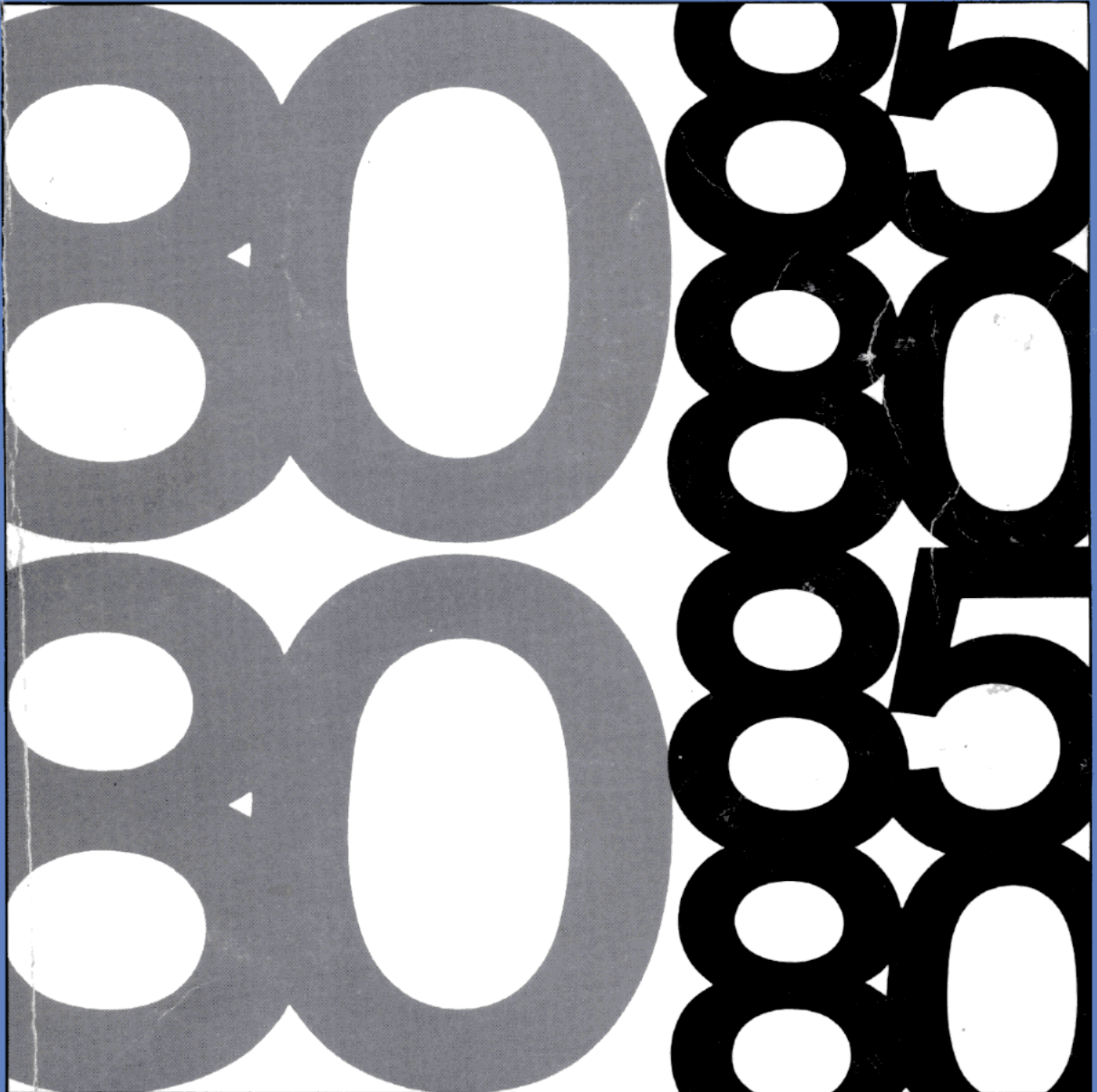
Radio Shack®
A DIVISION OF TANDY CORPORATION

Three Dollars and Ninety-Five Cents

intel®

8080/8085 Assembly Language Programming



62-1377

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Intel Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Intel.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

ICE–30	LIBRARY MANAGER
ICE–48	MCS
ICE–80	MEGACHASSIS
ICE–85	MICROMAP
INSITE	MULTIBUS
INTEL	PROMPT
INTELLEC	UPI

8080/8085 ASSEMBLY LANGUAGE PROGRAMMING MANUAL

Copyright © 1977, 1978 Intel Corporation

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

PREFACE

This manual describes programming with Intel's assembly language. It will not teach you how to program a computer.

Although this manual is designed primarily for reference, it also contains some instructional material to help the beginning programmer. The manual is organized as follows:

Chapter 1. ASSEMBLY LANGUAGE AND PROCESSORS

Description of the assembler
Overview of 8080 hardware and instruction set
Description of 8080/8085 differences

Chapter 2. ASSEMBLY LANGUAGE CONCEPTS

General assembly language coding rules

Chapter 3. INSTRUCTION SET

Descriptions of each instruction (these are listed alphabetically for quick reference)

Chapter 4. ASSEMBLER DIRECTIVES

Data definition
Conditional assembly
Relocation

Chapter 5. MACROS

Macro directives
Macro examples

Chapter 6. PROGRAMMING TECHNIQUES

Programming examples

Chapter 7. INTERRUPTS

Description of the interrupt system.

Chapters 3 and 4 will fill most of the experienced programmer's reference requirements. Use the table of contents or the index to locate information quickly.

The beginning programmer should read Chapters 1 and 2 and then skip to the examples in Chapter 6. As these examples raise questions, refer to the appropriate information in Chapter 3 or 4. Before writing a program, you will need to read Chapter 4. The 'Programming Tips' in Chapter 4 are intended especially for the beginning programmer.

RELATED PUBLICATIONS

To use your Intellec development system effectively, you should be familiar with the following Intel publications:

ISIS-II 8080/8085 MACRO ASSEMBLER OPERATOR'S MANUAL, 9800292

When you activate the assembler, you have the option of specifying a number of controls. The operator's manual describes the activation sequence for the assembler. The manual also describes the debugging tools and the error messages supplied by the assembler.

ISIS-II SYSTEM USER'S GUIDE, 9800306

User programs are commonly stored on diskette files. The ISIS-II User's Guide describes the use of the text editor for entering and maintaining programs. The manual also describes the procedures for linking and locating relocatable program modules.

Hardware References

For additional information about processors and their related components, refer to the appropriate User's Manual:

8080 MICROCOMPUTER SYSTEMS USER'S MANUAL, 9800153

8085 MICROCOMPUTER SYSTEMS USER'S MANUAL, 9800366

TABLE OF CONTENTS

Chapter 1. ASSEMBLY LANGUAGE AND PROCESSORS 1-1

Introduction 1-1
What Is An Assembler? 1-1
What the Assembler Does 1-1
Object Code 1-2
Program Listing 1-2
Symbol-Cross-Reference Listing 1-3
Do You Need the Assembler? 1-3
Overview of 8080/8085 Hardware 1-5
Memory 1-5
ROM 1-5
RAM 1-5
Program Counter 1-6
Work Registers 1-7
Internal Work Registers 1-9
Condition Flags 1-9
Carry Flag 1-10
Sign Flag 1-10
Zero Flag 1-11
Parity Flag 1-11
Auxiliary Carry Flag 1-11
Stack and Stack Pointer 1-12
Stack Operations 1-13
Saving Program Status 1-13
Input/Output Ports 1-14
Instruction Set 1-15
Addressing Modes 1-15
Implied Addressing 1-15
Register Addressing 1-15
Immediate Addressing 1-15
Direct Addressing 1-15
Register Indirect Addressing 1-16
Combined Addressing Modes 1-16
Timing Effects of Addressing Modes 1-16
Instruction Naming Conventions 1-16
Data Transfer Group 1-16
Arithmetic Group 1-17
Logical Group 1-17
Branch Group 1-18
Stack, I/O, and Machine Control Instructions 1-19
Hardware/Instruction Summary 1-19
Accumulator Instructions 1-19
Register Pair (Word) Instructions 1-21
Branching Instructions 1-22
Instruction Set Guide 1-23

8085 Processor Differences 1-24
Programming for the 8085 1-24
Conditional Instructions 1-25

Chapter 2. ASSEMBLY LANGUAGE CONCEPTS 2-1

Introduction 2-1
Source Line Format 2-1
Character Set 2-1
Delimiters 2-2
Label/Name Field 2-3
Opcode Field 2-4
Operand Field 2-4
Comment Field 2-4
Coding Operand Field Information 2-4
Hexadecimal Data 2-5
Decimal Data 2-5
Octal Data 2-5
Binary Data 2-5
Location Counter 2-6
ASCII Constant 2-6
Labels Assigned Values 2-6
Labels of Instruction or Data 2-6
Expressions 2-6
Instructions as Operands 2-7
Register-Type Operands 2-7
Two's Complement Representation of Data 2-7
Symbols and Symbol Tables 2-9
Symbolic Addressing 2-9
Symbolic Characteristics 2-9
Reserved, User-Defined, and Assembler-Generated Symbols 2-9
Global and Limited Symbols 2-10
Permanent and Redefinable Symbols 2-11
Absolute and Relocatable Symbols 2-11
Assembly-Time Expression Evaluation 2-11
Operators 2-11
Arithmetic Operators 2-12
Shift Operators 2-12
Logical Operators 2-13
Compare Operators 2-13
Byte Isolation Operators 2-14
Permissible Range of Values 2-15
Precedence of Operators 2-15
Relocatable Expressions 2-16
Chaining of Symbol Definitions 2-18

Chapter 3. INSTRUCTION SET 3-1

How to Use this Chapter 3-1
Timing Information 3-1
Instructions are listed in alphabetical order

Chapter 4. ASSEMBLER DIRECTIVES 4-1

Symbol Definition 4-2
EQU Directive 4-2
SET Directive 4-3
Data Definition 4-3
DB Directive 4-3
DW Directive 4-4
Memory Reservation 4-5
DS Directive 4-5
Programming Tips: Data Description and Access 4-6
Random Access Versus Read Only Memory 4-6
Data Description 4-6
Data Access 4-6
Add Symbols for Data Access 4-7
Conditional Assembly 4-8
IF, ELSE, ENDIF Directives 4-8
Assembler Termination 4-10
END Directive 4-10
Location Counter Control and Relocation 4-11
Location Counter Control (Non-Relocatable Mode) 4-11
ORG Directive 4-11
Introduction to Relocatability 4-12
Memory Management 4-12
Modular Program Development 4-12
Directives Used for Relocation 4-14
Location Counter Control (Relocatable Programs) 4-14
ASEG Directive 4-14
CSEG Directive 4-15
DSEG Directive 4-15
ORG Directive (Relocatable Mode) 4-16
Program Linkage Directives 4-16
PUBLIC Directive 4-17
EXTRN Directive 4-17
NAME Directive 4-18
STKLN Directive 4-18
STACK and MEMORY Reserved Words 4-19
Programming Tips: Testing Relocatable Modules 4-19
Initialization Routines 4-19
Input/Output 4-20
Remove Coding Used for Testing 4-20

Chapter 5. MACROS 5-1

Introduction to Macros 5-1
Why Use Macros? 5-1
What Is A Macro? 5-1
Macros Vs. Subroutines 5-3

Using Macros 5-3
Macro Definition 5-3
Macro Definition Directives 5-4
MACRO Directive 5-4
ENDM Directive 5-5
LOCAL Directive 5-5
REPT Directive 5-6
IRP Directive 5-8
IRPC Directive 5-8
EXITM Directive 5-9
Special Macro Operators 5-10
Nested Macro Definitions 5-12
Macros Calls 5-12
Macro Call Format 5-12
Nested Macro Calls 5-14
Macro Expansion 5-15
Null Macros 5-16
Sample Macros 5-16

Chapter 6. PROGRAMMING TECHNIQUES 6-1

Branch Tables Pseudo-Subroutine 6-1
Transferring Data to Subroutine 6-3
Software Multiply and Divide 6-7
Multibyte Addition and Subtraction 6-11
Decimal Addition 6-12
Decimal Subtraction 6-14

Chapter 7. INTERRUPTS 7-1

Interrupt Concepts 7-1
Writing Interrupt Subroutines 7-4

Appendix A INSTRUCTION SUMMARY A-1
Appendix B ASSEMBLER DIRECTIVE SUMMARY B-1
Appendix C ASCII CHARACTER SET C-1
Appendix D BINARY-DECIMAL-HEXADECIMAL CONVERSION TABLES D-1

LIST OF ILLUSTRATIONS

Figure

1-1	ASSEMBLER OUTPUTS	1-2
1-2	COMPARISON OF ASSEMBLY LANGUAGE WITH PL/M	1-4
1-3	8080/8085 INTERNAL REGISTERS	1-6
1-4	INSTRUCTION FETCH	1-8
1-5	EXECUTION OF MOV M,C INSTRUCTION	1-9

1. ASSEMBLY LANGUAGE AND PROCESSORS

INTRODUCTION

Almost every line of source coding in an assembly language source program translates directly into a machine instruction for a particular processor. Therefore, the assembly language programmer must be familiar with both the assembly language and the processor for which he is programming.

The first part of this chapter describes the assembler. The second part describes the features of the 8080 microprocessor from a programmer's point of view. Programming differences between the 8080 and the 8085 microprocessors are relatively minor. These differences are described in a short section at the end of this chapter.

WHAT IS AN ASSEMBLER?

An assembler is a software tool – a program – designed to simplify the task of writing computer programs. If you have ever written a computer program directly in a machine-recognizable form such as binary or hexadecimal code, you will appreciate the advantages of programming in a symbolic assembly language.

Assembly language operation codes (opcodes) are easily remembered (MOV for move instructions, JMP for jump). You can also symbolically express addresses and values referenced in the operand field of instructions. Since you assign these names, you can make them as meaningful as the mnemonics for the instructions. For example, if your program must manipulate a date as data, you can assign it the symbolic name DATE. If your program contains a set of instructions used as a timing loop (a set of instructions executed repeatedly until a specific amount of time has passed), you can name the instruction group TIMER.

What the Assembler Does

To use the assembler, you first need a source program. The source program consists of programmer-written assembly language instructions. These instructions are written using mnemonic opcodes and labels as described previously.

Assembly language source programs must be in a machine-readable form when passed to the assembler. The Intellec development system includes a text editor that will help you maintain source programs as paper tape files or diskette files. You can then pass the resulting *source program file* to the assembler. (The text editor is described in the ISIS-II System User's Guide.)

The assembler program performs the clerical task of translating symbolic code into *object code* which can be executed by the 8080 and 8085 microprocessors. Assembler output consists of three possible files: the *object file* containing your program translated into object code; the *list file* printout of your source code, the assembler-generated object code, and the symbol table; and the *symbol-cross-reference file*, a listing of the symbol-cross-reference records.

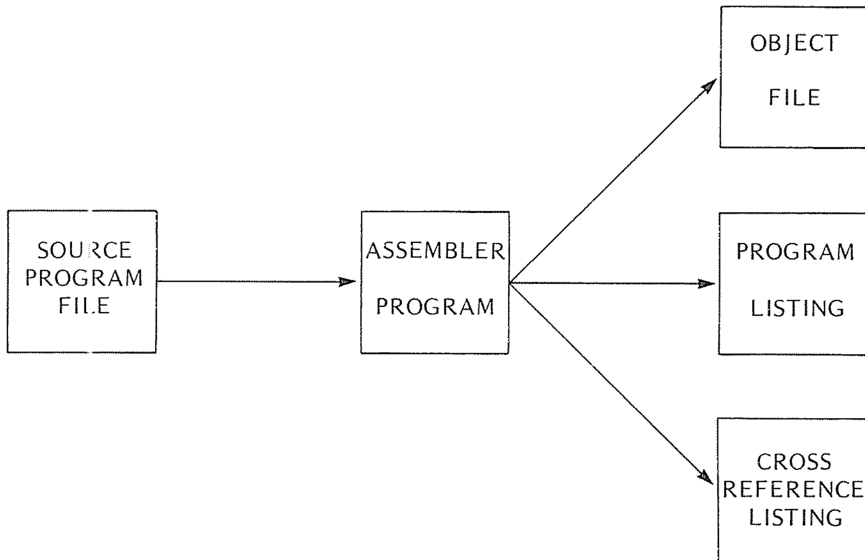


Figure 1-1. Assembler Outputs

Object Code

For most microcomputer applications, you probably will eventually load the object program into some form of read only memory. However, do not forget that the Intellec development system is an 8080 microcomputer system with random access memory. In most cases you can load and execute your object program on the development system for testing and debugging. This allows you to test your program before your prototype application system is fully developed.

A special feature of this assembler is that it allows you to request object code in a relocatable format. This frees the programmer from worrying about the eventual mix of read only and random access memory in the application system; individual portions of the program can be relocated as needed when the application design is final. Also, a large program can be broken into a number of separately assembled modules. Such modules are both easier to code and to test. See Chapter 4 of this manual for a more thorough description of the advantages of the relocation feature.

Program Listing

The program listing provides a permanent record of both the source program and the object code. The assembler also provides diagnostic messages for common programming errors in the program listing. For example, if you specify a 16-bit value for an instruction that can use only an 8-bit value, the assembler tells you that the value exceeds the permissible range.

Symbol-Cross-Reference Listing

The symbol-cross-reference listing is another of the diagnostic tools provided by the assembler. Assume, for example, that your program manipulates a data field named DATE, and that testing reveals a program logic error in the handling of this data. The symbol-cross-reference listing simplifies debugging this error because it points you to each instruction that references the symbol DATE.

Do You Need the Assembler?

The assembler is but one of several tools available for developing microprocessor programs. Typically, choosing the most suitable tool is based on cost restraints versus the required level of performance. You or your company must determine cost restraints; the required level of performance depends on a number of variables:

- The number of programs to be written: The greater the number of programs to be written, the more you need development support. Also, it must be pointed out that there can be penalties for *not* writing programs. When your application has access to the power of a microprocessor, you may be able to provide customers with custom features through program changes. Also, you may be able to add features through programming.

- The time allowed for programming: As the time allowed for programming decreases, the need for programming support increases.

- The level of support for existing programs: Sometimes programming errors are not discovered until the program has been in use for quite a while. Your need for programming support increases if you agree to correct such errors for your customers. The number of supported programs in use can multiply this requirement. Also, program support is frequently subject to stringent time constraints.

If none of the factors described above apply to your situation, you may be able to get along without the assembler. Intel's PROMPT-80, for example, allows you to enter programs directly into programmable read only memory. You enter the program manually as a string of hexadecimal digits. Such manual programming is relatively slow and more prone to human error than computer-assisted programming. However, manual systems are one of the least expensive tools available for microprocessor programming. Manual systems may be suitable for limited applications, hobbyists, and those who want to explore possible applications for microprocessors.

If most of the factors listed previously apply to you, you should explore the advantages of PL/M. PL/M is Intel's high-level language for program development. A high-level language is directed more to problem solving than to a particular microprocessor. This allows you to write programs much more quickly than a hardware-oriented language such as assembly language. As an example, assume that a program must move five characters from one location in memory to another. The following example illustrates the coding differences between assembly language and PL/M. Since instructions have not yet been described, the assembly language instructions are represented by a flowchart.

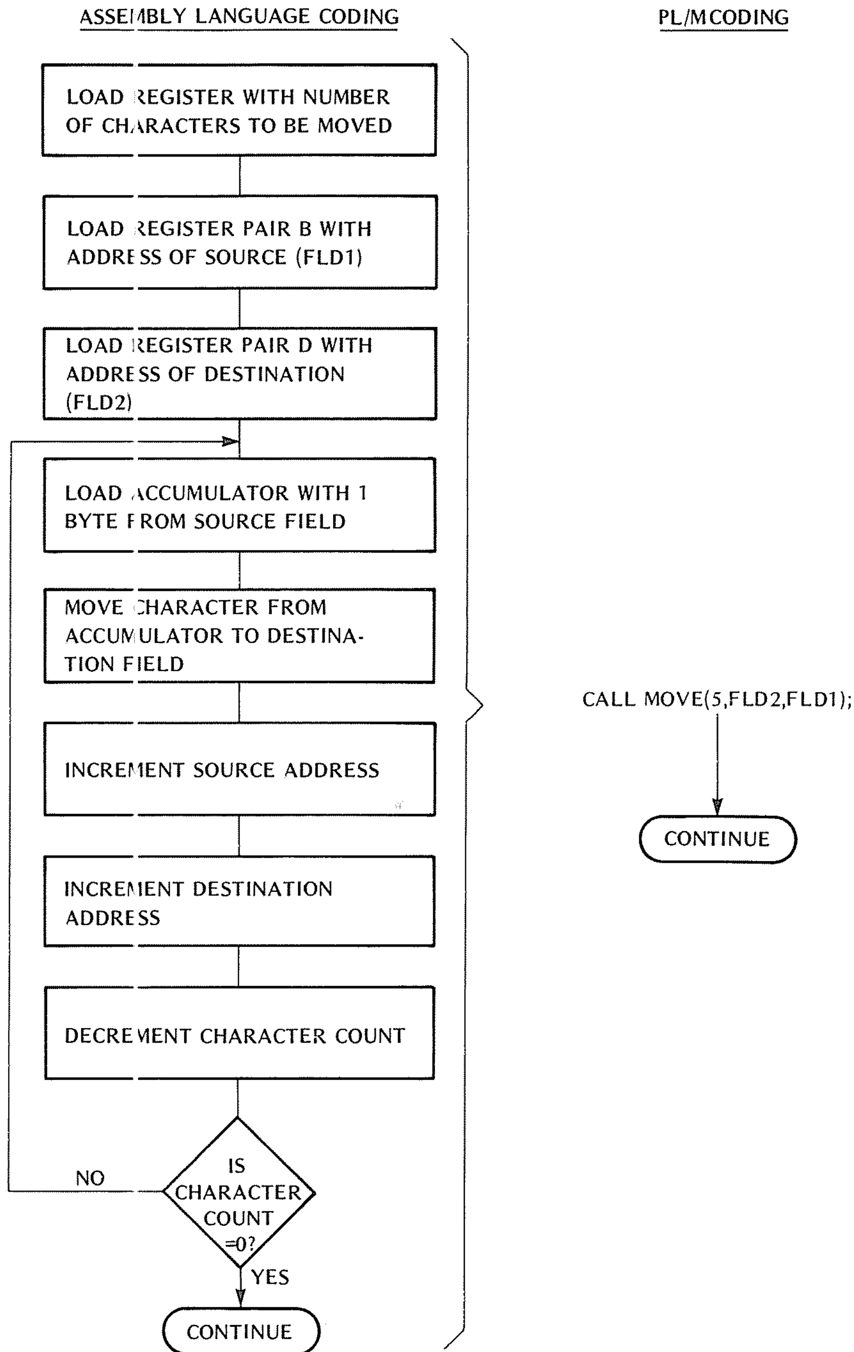


Figure 1-2. Comparison of Assembly Language with PL/M

OVERVIEW OF 8080/8085 HARDWARE

To the programmer, the computer comprises the following parts:

- Memory
- The program counter
- Work registers
- Condition flags
- The stack and stack pointer
- Input/output ports
- The instruction set

Of the components listed above, memory is not part of the processor, but is of interest to the programmer.

Memory

Since the program required to drive a microprocessor resides in memory, all microprocessor applications require some memory. There are two general types of memory: read only memory (ROM) and random access memory (RAM).

ROM

As the name implies, the processor can only read instructions and data from ROM; it cannot alter the contents of ROM. By contrast, the processor can both read from and write to RAM. Instructions and unchanging data are permanently fixed into ROM and remain intact whether or not power is applied to the system. For this reason, ROM is typically used for program storage in single-purpose microprocessor applications. With ROM you can be certain that the program is ready for execution when power is applied to the system. With RAM a program must be loaded into memory each time power is applied to the processor. Notice, however, that storing programs in RAM allows a multi-purpose system since different programs can be loaded to serve different needs.

Two special types of ROM — PROM (Programmable Read Only Memory) and EPROM (Eraseable Programmable Read Only Memory) — are frequently used during program development. These memories are useful during program development since they can be altered by a special PROM programmer. In high-volume commercial applications, these special memories are usually replaced by less expensive ROM's.

RAM

Even if your program resides entirely in ROM, your application is likely to require some random access memory. Any time your program attempts to write any data to memory, that memory must be RAM. Also, if your program uses the stack, you need RAM. If your program modifies any of its own instructions (this procedure is discouraged), those instructions must reside in RAM.

The mix of ROM and RAM in an application is important to both the system designer and the programmer. Normally, the programmer must know the physical addresses of the RAM in the system so that data variables

can be assigned within those addresses. However, the relocation feature of this assembler allows you to code a program without concern for the ultimate placement of data and instructions; these program elements can be repositioned after the program has been tested and after the system's memory layout is final. The relocation feature is fully explained in Chapter 4 of this manual.

Program Counter

With the program counter, we reach the first of the 8080's internal registers illustrated in Figure 1-3.

NOTE

> Except for the differences listed at the end of this chapter, the information in this chapter applies equally to the 8080 and the 8085.

The program counter keeps track of the next instruction byte to be fetched from memory (which may be either ROM or RAM). Each time it fetches an instruction byte from memory, the processor increments the program counter by one. Therefore, the program counter always indicates the next byte to be fetched. This process continues as long as program instructions are executed sequentially. To alter the flow of program execution as with a jump instruction or a call to a subroutine, the processor overwrites the current contents of the program counter with the address of the new instruction. The next instruction fetch occurs from the new address.

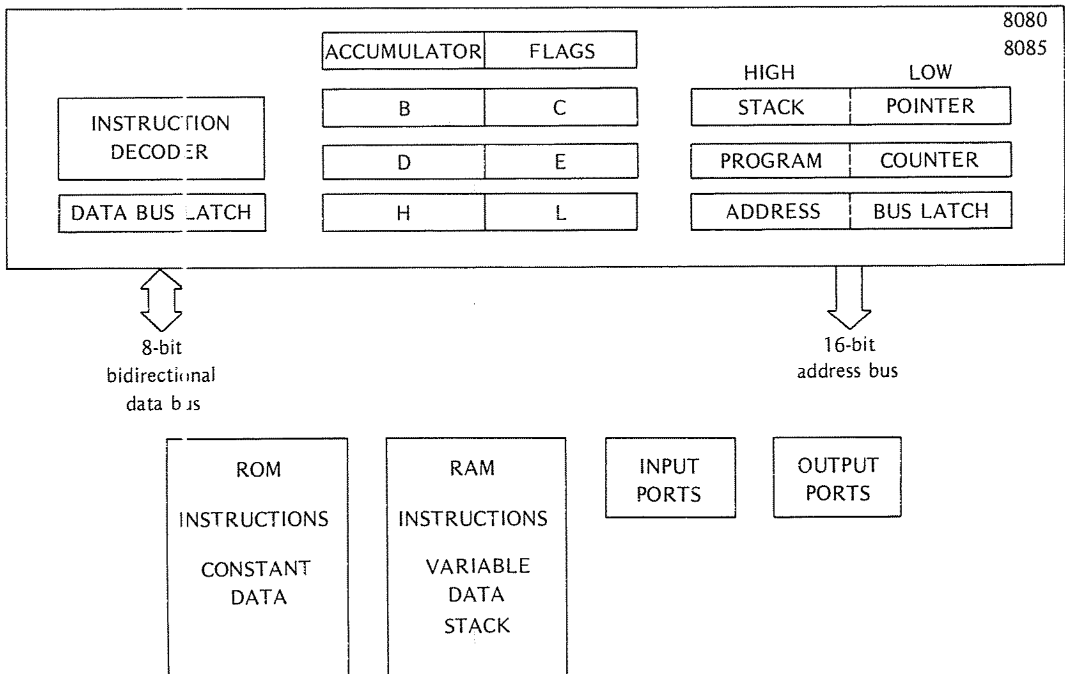


Figure 1-3. 8080/8085 Internal Registers

Work Registers

The 8080 provides an 8-bit accumulator and six other general purpose work registers, as shown in Figure 1-3.

Programs reference these registers by the letters A (for the accumulator), B, C, D, E, H, and L. Thus, the instruction ADD B may be interpreted as 'add the contents of the B register to the contents of the accumulator.

Some instructions reference a pair of registers as shown in the following:

Symbolic Reference	*Registers Referenced*
B	B and C
D	D and E
H	H and L
M	H and L (as a memory reference)
PSW	A and condition flags (explained later in this section)

The symbolic reference for a single register is often the same as for a register pair. The instruction to be executed determines how the processor interprets the reference. For example, ADD B is an 8-bit operation. By contrast PUSH B (which pushes the contents of the B and C registers onto the stack) is a 16-bit operation.

Notice that the letters H and M both refer to the H and L register pair. The choice of which to use depends on the instruction. Use H when an instruction acts upon the H and L register pair as in INX H (increment the contents of H and L by one). Use M when an instruction addresses memory via the H and L registers as in ADD M (add the contents of the memory location specified by the H and L registers to the contents of the accumulator).

The general purpose registers B, C, D, E, H, and L can provide a wide variety of functions such as storing 8-bit data values, storing intermediate results in arithmetic operations, and storing 16-bit address pointers. Because of the 8080's extensive instruction set, it is usually possible to achieve a common result with any of several different instructions. A simple add to the accumulator, for example, can be accomplished by more than half a dozen different instructions. When possible, it is generally desirable to select a register-to-register instruction such as ADD B. These instructions typically require only one byte of program storage. Also, using data already present in a register eliminates a memory access and thus reduces the time required for the operation.

The accumulator also acts as a general-purpose register, but it has some special capabilities not shared with the other registers. For example, the input/output instructions IN and OUT transfer data only between the accumulator and external I/O devices. Also, many operations involving the accumulator affect the condition flags as explained in the next section.

Example:

The following figures illustrate the execution of a move instruction. The MOV M,C moves a copy of the contents of register C to the memory location specified by the H and L registers. Notice that this location must be in RAM since data is to be written to memory.

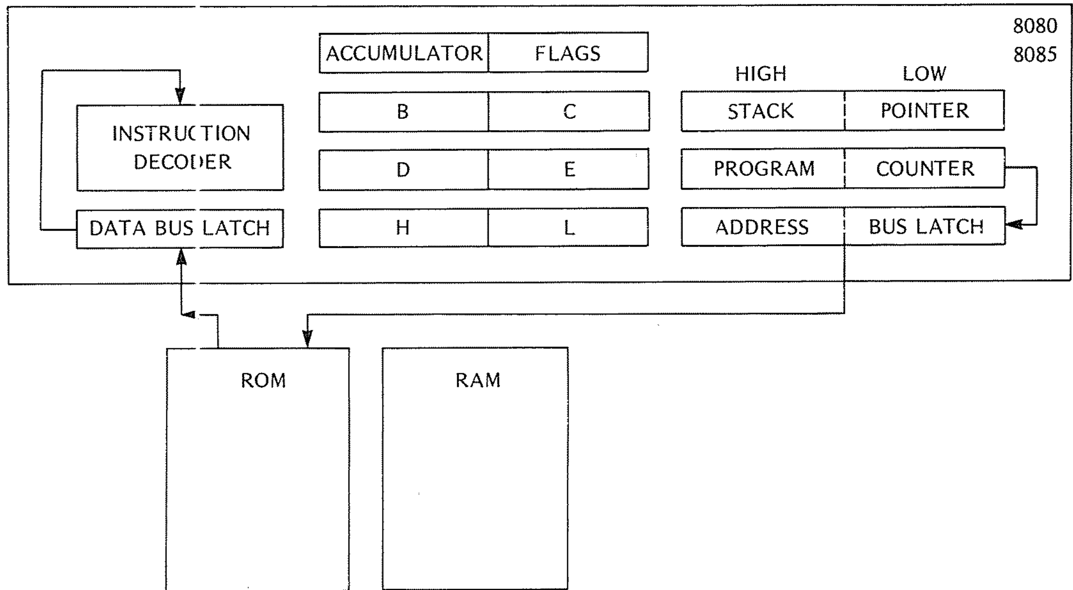


Figure 1-4. Instruction Fetch

The processor initiates the instruction fetch by latching the contents of the program counter on the address bus, and then increments the program counter by one to indicate the address of the next instruction byte. When the memory responds, the instruction is decoded into the series of actions shown in Figure 1-5.

NOTE

The following description of the execution of the MOV M,C instruction is conceptually correct, but does not account for normal bus control. For details concerning memory interface, refer to the User's Manual for your processor.

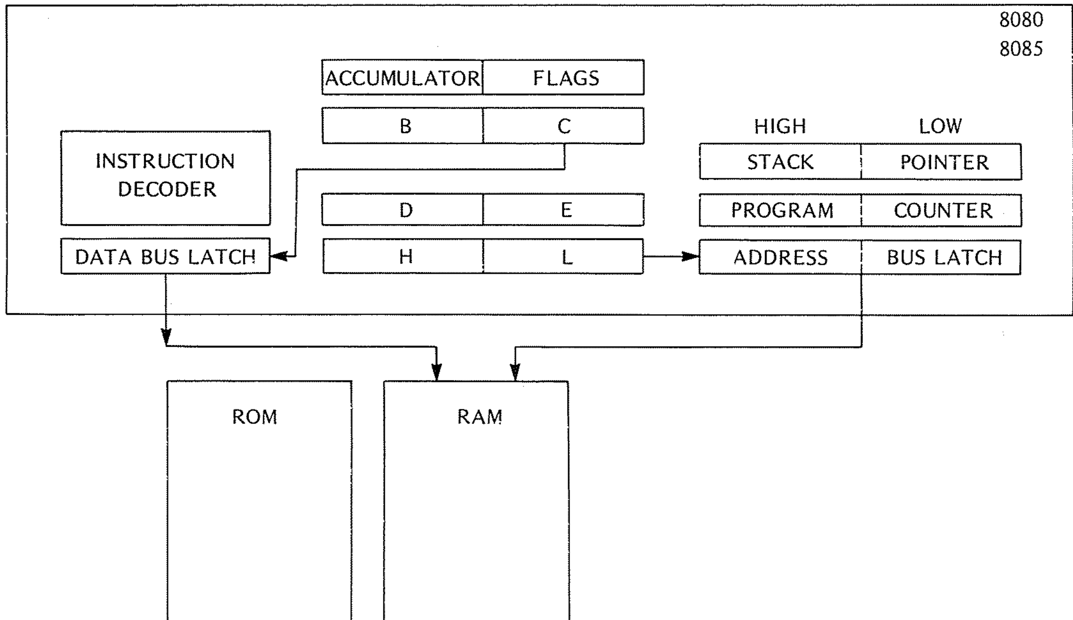


Figure 1-5. Execution of MOV M,C Instruction

To execute the MOV M,C instruction, the processor latches the contents of the C register on the data bus and the contents of the H and L registers on the address bus. When the memory accepts the data, the processor terminates execution of this instruction and initiates the next instruction fetch.

Internal Work Registers

Certain operations are destructive. For example, a compare is actually a subtract operation; a zero result indicates that the opreands are equal. Since it is unacceptable to destroy either of the operands, the processor includes several work registers reserved for its own use. The programmer cannot access these registers. These registers are used for internal data transfers and for preserving operands in destructive operations.

Condition Flags

The 8080 provides five flip flops used as condition flags. Certain arithmetic and logical instructions alter one or more of these flags to indicate the result of an operation. Your program can test the setting of four of these flags (carry, sign, zero, and parity) using one of the conditional jump, call, or return instructions. This allows you to alter the flow of program execution based on the outcome of a previous operation. The fifth flag, auxiliary carry, is reserved for the use of the DAA instruction, as will be explained later in this section.

It is important for the programmer to know which flags are set by a particular instruction. Assume, for example, that your program is to test the parity of an input byte and then execute one instruction sequence if parity is even, a different instruction set if parity is odd. Coding a JPE (jump if parity is even) or JPO (jump if parity is

odd) instruction immediately following the IN (input) instruction produces false results since the IN instruction does not affect the condition flags. The jump executed by your program reflects the outcome of some previous operation unrelated to the IN instruction. For the operation to work correctly, you must include some instruction that alters the parity flag after the IN instruction, but before the jump instruction. For example, you can add zero to the accumulator. This sets the parity flag without altering the data in the accumulator.

In other cases you will want to set a flag with one instruction, but then have a number of intervening instructions before you use it. In these cases, you must be certain that the intervening instructions do not affect the desired flag.

The flags set by each instruction are detailed in the individual instruction descriptions in Chapter 3 of this manual.

NOTE

When a flag is 'set' it is set ON (has the value one); when a flag is 'reset' it is reset OFF (has the value zero).

Carry Flag

As its name implies, the carry flag is commonly used to indicate whether an addition causes a 'carry' into the next higher order digit. The carry flag is also used as a 'borrow' flag in subtractions, as explained under 'Two's Complement Representation of Data' in Chapter 2 of this manual. The carry flag is also affected by the logical AND, OR, and exclusive OR instructions. These instructions set ON or OFF particular bits of the accumulator. See the descriptions of the ANA, ANI, ORA, ORI, XRA, and XRI instructions in Chapter 3.

The rotate instructions, which move the contents of the accumulator one position to the left or right, treat the carry bit as though it were a ninth bit of the accumulator. See the descriptions of the RAL, RAR, RLC, and RRC instructions in Chapter 3 of this manual.

Example:

Addition of two one-byte numbers can produce a carry out of the high-order bit:

```
Bit Number:   7654 3210
       AE=    1010 1110
      +74=    0111 0100
              ---------
              0010 0010 = 22 carry flag = 1
```

An addition that causes a carry out of the high order bit sets the carry flag to 1; an addition that does not cause a carry resets the flag to zero.

Sign Flag

As explained under 'Two's Complement Representation of Data' in Chapter 2, bit 7 of a result in the accumulator can be interpreted as a sign. Instructions that affect the sign flag set the flag equal to bit 7. A zero in bit 7

indicates a positive value; a one indicates a negative value. This value is duplicated in the sign flag so that conditional jump, call, and return instructions can test for positive and negative values.

Zero Flag

Certain instructions set the zero flag to one to indicate that the result in the accumulator contains all zeros. These instructions reset the flag to zero if the result in the accumulator is other than zero. A result that has a carry and a zero result also sets the zero bit as shown below:

```
 1010 0111
+0101 1001
----------
 0000 0000    Carry Flag = 1
              Zero Flag  = 1
```

Parity Flag

Parity is determined by counting the number of one bits set in the result in the accumulator. Instructions that affect the parity flag set the flag to one for even parity and reset the flag to zero to indicate odd parity.

Auxiliary Carry Flag

The auxiliary carry flag indicates a carry out of bit 3 of the accumulator. You cannot test this flag directly in your program; it is present to enable the DAA (Decimal Adjust Accumulator) to perform its function.

The auxiliary carry flag and the DAA instruction allow you to treat the value in the accumulator as two 4-bit binary coded decimal numbers. Thus, the value 0001 1001 is equivalent to 19. (If this value is interpreted as a binary number, it has the value 25.) Notice, however, that adding one to this value produces a non-decimal result:

```
 0001 1001
+0000 0001
----------
 0001 1010 = 1A
```

The DAA instruction converts hexadecimal values such as the A in the preceding example back into binary coded decimal (BCD) format. The DAA instruction requires the auxiliary carry flag since the BCD format makes it possible for arithmetic operations to generate a carry from the low-order 4-bit digit into the high-order 4-bit digit. The DAA performs the following addition to correct the preceding example:

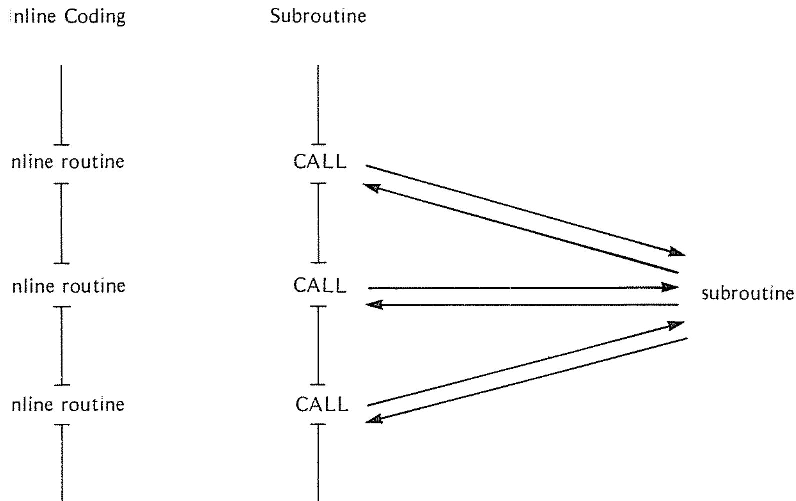
```
 0001 1010
+0000 0110
----------
 0001 0000
+0001 0000  (auxiliary carry)
----------
 0010 0000 = 20
```

The auxiliary carry flag is affected by all add, subtract, increment, decrement, compare, and all logical AND, OR, and exclusive OR instructions. (See the descriptions of these instructions in Chapter 3.) There is some difference in the handling of the auxiliary carry flag by the logical AND instructions in the 8080 processor and the 8085 processor. The 8085 logical AND instructions always set the auxiliary flag ON. The 8080 logical AND instructions set the flag to reflect the logical OR of bit 3 of the values involved in the AND operation.

Stack and Stack Pointer

To understand the purpose and effectiveness of the stack, it is useful to understand the concept of a subroutine.

Assume that your program requires a multiplication routine. (Since the 8080 has no multiply instructions, this can be performed through repetitive addition. For example, 3x4 is equivalent to 3+3+3+3.) Assume further that your program needs this multiply routine several times. You can recode this routine inline each time it is needed, but this can use a great deal of memory. Or, you can code a subroutine:



The 8080 provides instructions that call and return from a subroutine. When the call instruction is executed, the address of the next instruction (the contents of the program counter) is pushed onto the stack. The contents of the program counter are replaced by the address of the desired subroutine. At the end of the subroutine, a return instruction pops that previously-stored address off the stack and puts it back into the program counter. Program execution then continues as though the subroutine had been coded inline.

The mechanism that makes this possible is, of course, the stack. The stack is simply an area of random access memory addressed by the stack pointer. The stack pointer is a hardware register maintained by the processor. However, your program must initialize the stack pointer. This means that your program must load the base address of the stack into the stack pointer. The base address of the stack is commonly assigned to the *highest* available address in RAM. This is because the stack expands by *decrementing* the stack pointer. As items are

added to the stack, it expands into memory locations with *lower* addresses. As items are removed from the stack, the stack pointer is incremented back toward its base address. Nonetheless, the most recent item on the stack is known as the 'top of the stack.' Stack is still a most descriptive term because you can always put something else on top of the stack. In terms of programming, a subroutine can call a subroutine, and so on. The only limitation to the number of items that can be added to the stack is the amount of RAM available for the stack.

The amount of RAM allocated to the stack is important to the programmer. As you write your program, you must be certain that the stack will not expand into areas reserved for other data. For most applications, this means that you must assign data that requires RAM to the lowest RAM addresses available. To be more precise, you must count up all instructions that add data to the stack. Ultimately, your program should remove from the stack any data it places on the stack. Therefore, for any instruction that adds to the stack, you can subtract any *intervening* instruction that removes an item from the stack. The most critical factor is the maximum size of the stack. Notice that you must be sure to remove data your program adds to the stack. Otherwise, any left-over items on the stack may cause the stack to grow into portions of RAM you intend for other data.

Stack Operations

Stack operations transfer sixteen bits of data between memory and a pair of processor registers. The two basic operations are PUSH, which adds data to the stack, and POP, which removes data from the stack.

A call instruction pushes the contents of the program counter (which contains the address of the next instruction) onto the stack and then transfers control to the desired subroutine by placing its address in the program counter. A return instruction pops sixteen bits off the stack and places them in the program counter. This requires the programmer to keep track of what is in the stack. For example, if you call a subroutine and the subroutine pushes data onto the stack, the subroutine must remove that data before executing a return instruction. Otherwise, the return instruction pops data from the stack and places it in the program counter. The results are unpredictable, of course, but probably not what you want.

Saving Program Status

It is likely that a subroutine requires the use of one or more of the working registers. However, it is equally likely that the main program has data stored in the registers that it needs when control returns to the main program. As general rule, a subroutine should save the contents of a register before using it and then restore the contents of that register before returning control to the main program. The subroutine can do this by pushing the contents of the registers onto the stack and then popping the data back into the registers before executing a return. The following instruction sequence saves and restores all the working registers. Notice that the POP instructions must be in the opposite order of the PUSH instructions if the data is to be restored to its original location.

```
SUBRTN:     PUSH   PSW
            PUSH   B
            PUSH   D
            PUSH   H
              .
            subroutine coding
              .
            POP    H
            POP    D
            POP    B
            POP    PSW
            RETURN
```

The letters B, D, and H refer to the B and C, D and E, and H and L register pairs, respectively. PSW refers to the program status word. The program status word is a 16-bit word comprising the contents of the accumulator and the five condition flags. (PUSH PSW adds three bits of filler to expand the condition flags into a full byte; POP PSW strips out these filler bits.)

Input/Output Ports

The 256 input/output ports provide communication with the outside world of peripheral devices. The IN and OUT instructions initiate data transfers.

The IN instruction latches the number of the desired port onto the address bus. As soon as a byte of data is returned to the data bus latch, it is transferred into the accumulator.

The OUT instruction latches the number of the desired port onto the address bus and latches the data in the accumulator onto the data bus.

The specified port number is duplicated on the address bus. Thus, the instruction IN 5 latches the bit configuration 0000 0101 0000 0101 onto the address bus.

Notice that the IN and OUT instructions simply initiate a data transfer. It is the responsibility of the peripheral device to detect that it has been addressed. Notice also that it is possible to dedicate any number of ports to the same peripheral device. You might use a number of ports as control signals, for example.

Because input and output are almost totally application dependent, a discussion of design techniques is beyond the scope of this manual.

For additional hardware information, refer to the 8080 or 8085 Microcomputer Systems User's Manual.

For related programming information, see the descriptions of the IN, OUT, DI, EI, RST, and RIM and SIM instructions in Chapter 3 of this manual. (The RIM and SIM instructions apply only to the 8085.)

Instruction Set

The 8080 incorporates a powerful array of instructions. This section provides a general overview of the instruction set. The detailed operation of each instruction is described in Chapter 3 of this manual.

Addressing Modes

Instructions can be categorized according to their method of addressing the hardware registers and/or memory.

Implied Addressing. The addressing mode of certain instructions is implied by the instruction's function. For example, the STC (set carry flag) instruction deals only with the carry flag; the DAA (decimal adjust accumulator) instruction deals with the accumulator.

Register Addressing. Quite a large set of instructions call for register addressing. With these instructions, you must specify one of the registers A through E, H or L as well as the operation code. With these instructions, the accumulator is implied as a second operand. For example, the instruction CMP E may be interpreted as 'compare the contents of the E register with the contents of the accumulator.'

Most of the instructions that use register addressing deal with 8-bit values. However, a few of these instructions deal with 16-bit register pairs. For example, the PCHL instruction exchanges the contents of the program counter with the contents of the H and L registers.

Immediate Addressing. Instructions that use immediate addressing have data assembled as a part of the instruction itself. For example, the instruction CPI 'C' may be interpreted as 'compare the contents of the accumulator with the letter C.' When assembled, this instruction has the hexadecimal value FE43. Hexadecimal 43 is the internal representation for the letter C. When this instruction is executed, the processor fetches the first instruction byte and determines that it must fetch one more byte. The processor fetches the next byte into one of its internal registers and then performs the compare operation.

Notice that the names of the immediate instructions indicate that they use immediate data. Thus, the name of an add instruction is ADD; the name of an add immediate instruction is ADI.

All but two of the immediate instructions use the accumulator as an implied operand, as in the CPI instruction shown previously. The MVI (move immediate) instruction can move its immediate data to any of the working registers, including the accumulator, or to memory. Thus, the instruction MVI D,0FFH moves the hexadecimal value FF to the D register.

The LXI instruction (load register pair immediate) is even more unusual in that its immediate data is a 16-bit value. This instruction is commonly used to load addresses into a register pair. As mentioned previously, your program must initialize the stack pointer; LXI is the instruction most commonly used for this purpose. For example, the instruction LXI SP,30FFH loads the stack pointer with the hexadecimal value 30FF.

Direct Addressing. Jump instructions include a 16-bit address as part of the instruction. For example, the instruction JMP 1000H causes a jump to the hexadecimal address 1000 by replacing the current contents of the program counter with the new value 1000.

Instructions that include a direct address require three bytes of storage: one for the instruction code, and two for the 16-bit address.

Register Indirect Addressing. Register indirect instructions reference memory via a register pair. Thus, the instruction MOV M,C moves the contents of the C register into the memory address stored in the H and L register pair. The instruction LDAX B loads the accumulator with the byte of data specified by the address in the B and C register pair.

Combined Addressing Modes. Some instructions use a combination of addressing modes. A CALL instruction, for example, combines direct addressing and register indirect addressing. The direct address in a CALL instruction specifies the address of the desired subroutine; the register indirect address is the stack pointer. The CALL instruction pushes the current contents of the program counter into the memory location specified by the stack pointer.

Timing Effects of Addressing Modes. Addressing modes affect both the amount of time required for executing an instruction and the amount of memory required for its storage. For example, instructions that use implied or register addressing execute very quickly since they deal directly with the processor hardware or with data already present in hardware registers. More important, however, is that the entire instruction can be fetched with a single memory access. The number of memory accesses required is the single greatest factor in determining execution timing. More memory accesses require more execution time. A CALL instruction, for example, requires five memory accesses: three to access the entire instruction, and two more to push the contents of the program counter onto the stack.

The processor can access memory once during each processor cycle. Each cycle comprises a variable number of states. (The individual instruction descriptions in Chapter 3 specify the number of cycles and states required for each instruction.) The length of a state depends on the clock frequency specified for your system, and may range from 480 nanoseconds to 2 microseconds. Thus, the timing of a four state instruction may range from 1.920 microseconds through 8 microseconds. (The 8085 has a maximum clock frequency of 320 nanoseconds and therefore can execute instructions about 50% faster than the 8080.)

Instruction Naming Conventions

The mnemonics assigned to the instructions are designed to indicate the function of the instruction. The instructions fall into the following functional categories:

Data Transfer Group. The data transfer instructions move data between registers or between memory and registers.

MOV	Move
MVI	Move Immediate
LDA	Load Accumulator Directly from Memory
STA	Store Accumulator Directly in Memory
LHLD	Load H and L Registers Directly from Memory
SHLD	Store H and L Registers Directly in Memory

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

An 'X' in the name of a data transfer instruction implies that it deals with a register pair:

LXI	Load Register Pair with Immediate data
LDAX	Load Accumulator from Address in Register Pair
STAX	Store Accumulator in Address in Register Pair
XCHG	Exchange H and L with D and E
XTHL	Exchange Top of Stack with H and L

Arithmetic Group. The arithmetic instructions add, subtract, increment, or decrement data in registers or memory.

ADD	Add to Accumulator
ADI	Add Immediate Data to Accumulator
ADC	Add to Accumulator Using Carry Flag
ACI	Add Immediate Data to Accumulator Using Carry Flag
SUB	Subtract from Accumulator
SUI	Subtract Immediate Data from Accumulator
SBB	Subtract from Accumulator Using Borrow (Carry) Flag
SBI	Subtract Immediate from Accumulator Using Borrow
INR	Increment Specified Byte by One
DCR	Decrement Specified Byte by One
INX	Increment Register Pair by One
DCX	Decrement Register Pair by One
DAD	Double Register Add: Add Contents of Register Pair to H and L Register Pair

Logical Group. This group performs logical (Boolean) operations on data in registers and memory and on condition flags.

The logical AND, OR, and Exclusive OR instructions enable you to set specific bits in the accumulator ON or OFF.

ANA	Logical AND with Accumulator
ANI	Logical AND with Accumulator Using Immediate Data
ORA	Logical OR with Accumulator
ORI	Logical OR with Accumulator Using Immediate Data
XRA	Exclusive Logical OR with Accumulator
XRI	Exclusive OR Using Immediate Data

The compare instructions compare the contents of an 8-bit value with the contents of the accumulator:

CMP	Compare
CPI	Compare Using Immediate Data

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

The rotate instructions shift the contents of the accumulator one bit position to the left or right:

RLC	Rotate Accumulator Left
RRC	Rotate Accumulator Right
RAL	Rotate Left Through Carry
RAR	Rotate Right Through Carry

Complement and carry flag instructions:

CMA	Complement Accumulator
CMC	Complement Carry Flag
STC	Set Carry Flag

Branch Group. The branching instructions alter normal sequential program flow, either unconditionally or conditionally. The unconditional branching instructions are as follows:

JMP	Jump
CALL	Call
RET	Return

Conditional branching instructions examine the status of one of four condition flags to determine whether the specified branch is to be executed. The conditions that may be specified are as follows:

NZ	Not Zero (Z = 0)
Z	Zero (Z = 1)
NC	No Carry (C = 0)
C	Carry (C = 1)
PO	Parity Odd (P = 0)
PE	Parity Even (P = 1)
P	Plus (S = 0)
M	Minus (S = 1)

Thus, the conditional branching instructions are specified as follows:

Jumps	*Calls*	*Returns*	
JC	CC	RC	(Carry)
JNC	CNC	RNC	(No Carry)
JZ	CZ	RZ	(Zero)
JNZ	CNZ	RNZ	(Not Zero)
JP	CP	RP	(Plus)
JM	CM	RM	(Minus)
JPE	CPE	RPE	(Parity Even)
JPO	CPO	RPO	(Parity Odd)

Two other instructions can effect a branch by replacing the contents of the program counter:

PCHL	Move H and L to Program Counter
RST	Special Restart Instruction Used with Interrupts

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

Stack, I/O, and Machine Control Instructions. The following instructions affect the stack and/or stack pointer:

PUSH	Push Two Bytes of Data onto the Stack
POP	Pop Two Bytes of Data off the Stack
XTHL	Exchange Top of Stack with H and L
SPHL	Move contents of H and L to Stack Pointer

The I/O instructions are as follows:

IN	Initiate Input Operation
OUT	Initiate Output Operation

The machine control instructions are as follows:

EI	Enable Interrupt System
DI	Disable Interrupt System
HLT	Halt
NOP	No Operation

HARDWARE/INSTRUCTION SUMMARY

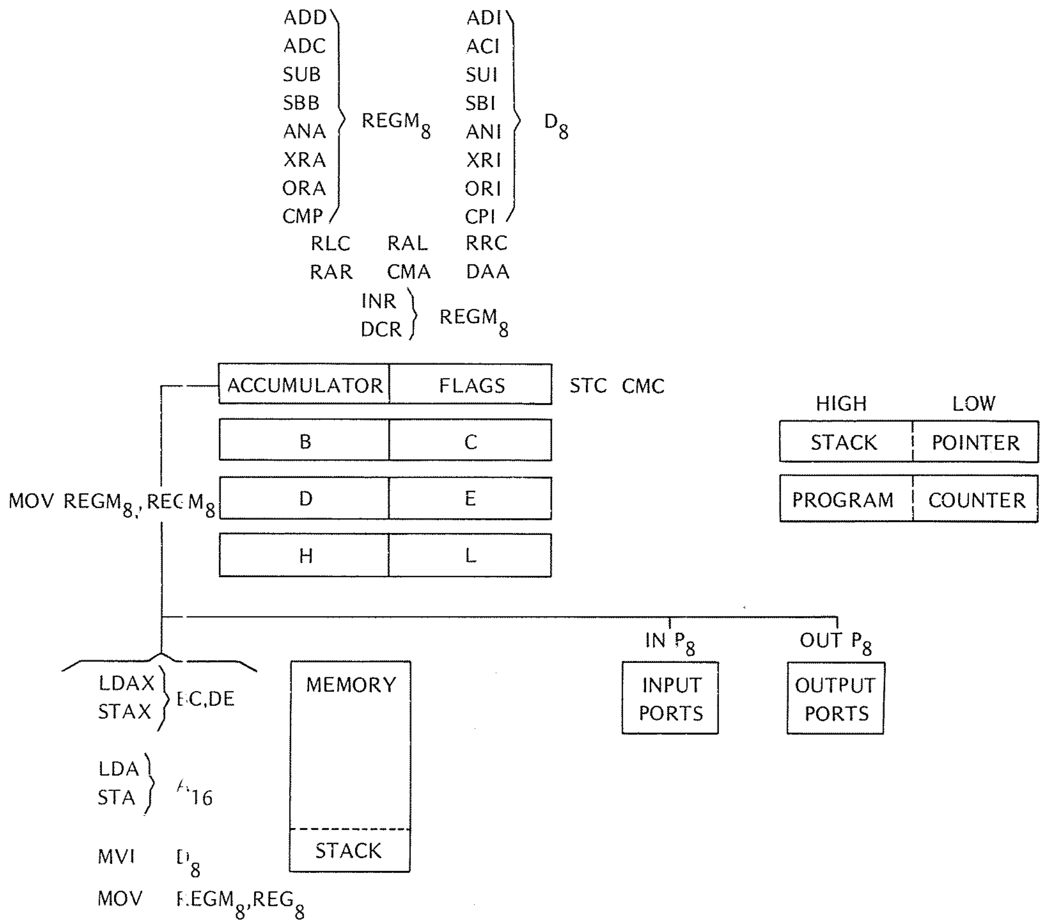
The following illustrations graphically summarize the instruction set by showing the hardware acted upon by specific instructions. The type of operand allowed for each instruction is indicated through the use of a code. When no code is given, the instruction does not allow operands.

Code	*Meaning*
$REGM_8$	The operand may specify one of the 8-bit registers A,B,C,D,E,H, or L or M (a memory reference via the 16-bit address in the H and L registers). The MOV instruction, which calls for two operands, can specify M for only one of its operands.
D_8	Designates 8-bit immediate operand.
A_{16}	Designates a 16-bit address.
P_8	Designates an 8-bit port number.
REG_{16}	Designates a 16-bit register pair (B&C,D&E,H&L, or SP).
D_{16}	Designates a 16-bit immediate operand.

Accumulator Instructions

The following illustration shows the instructions that can affect the accumulator. The instructions listed above the accumulator all act on the data in the accumulator, and all except CMA (complement accumulator) affect one or more of the condition flags. The instructions listed below the accumulator move data into or out of the accumulator, but do not affect condition flags. The STC (set carry) and CMC (complement carry) instructions are also shown here.

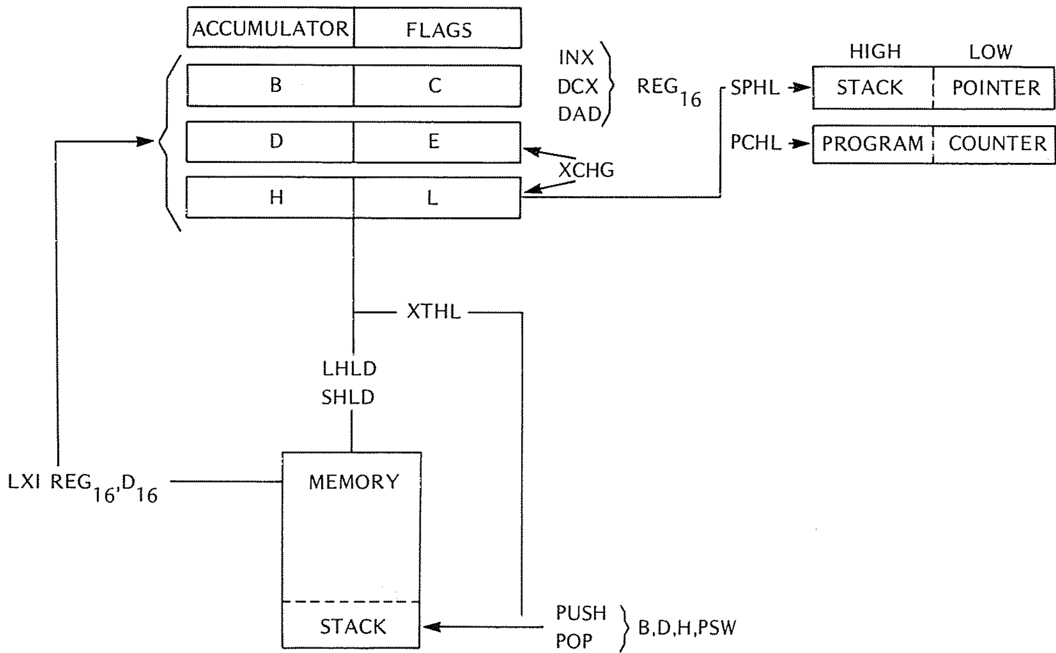
ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION



ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

Register Pair (Word) Instructions

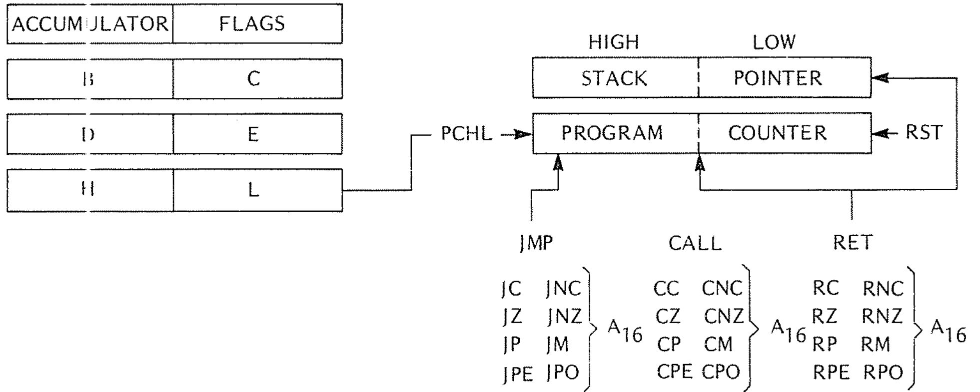
The following instructions all deal with 16-bit words. Except for DAD (which adds the contents of the B&C or D&E register pair to H&L), none of these instructions affect the condition flags. DAD affects only the carry flag.

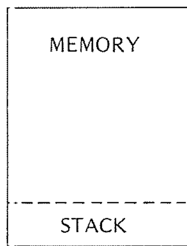


ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

Branching Instructions

The following instructions can alter the contents of the program counter, thereby altering the normal sequential execution flow. Jump instructions affect only the program counter. Call and Return instructions affect the program counter, stack pointer, and stack.





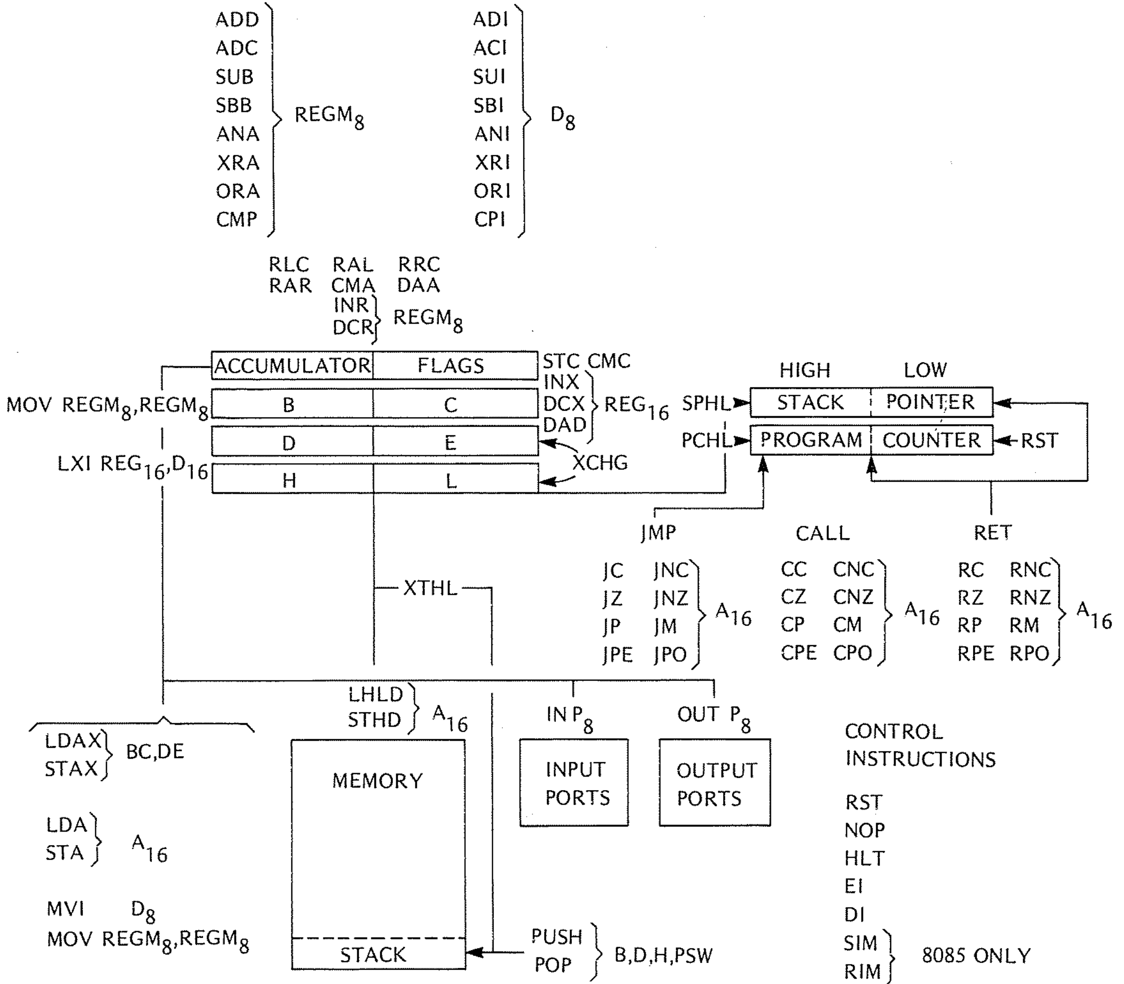
CONTROL INSTRUCTIONS

RST
NOP
HLT
EI
DI
SIM } 8085 only
RIM }

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

Instruction Set Guide

The following is a summary of the instruction set:



CODE	*MEANING*
$REGM_8$	The operand may specify one of the 8-bit registers A,B,C,D,E,H, or L or M (a memory reference via the 16-bit address in the H and L registers). The MOV instruction, which calls for two operands, can specify M for only one of its operands.
D_8	Designates 8-bit immediate operand.
A_{16}	Designates a 16-bit address.
P_8	Designates an 8-bit port number.
REG_{16}	Designates a 16-bit register pair (B&C,D&E,H&L,or SP).
D_{16}	Designates a 16-bit immediate operand.

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

8085 PROCESSOR DIFFERENCES

The differences between the 8080 processor and the 8085 processor will be more obvious to the system designer than to the programmer. Except for two additional instructions, the 8085 instruction set is identical to and fully compatible with the 8080 instruction set. Most programs written for the 8080 should operate on the 8085 without modification. The only programs that may require changes are those with critical timing routines; the higher system speed of the 8085 may alter the time values of such routines.

A partial listing of 8085 design features includes the following:

- A single 5 volt power supply.
- Execution speeds approximately 50% faster than the 8080.
- Incorporation in the processor of the features of the 8224 Clock Generator and Driver and the 8228 System Controller and Bus Driver.
- A non-maskable TRAP interrupt for handling serious problems such as power failures.
- Three separately maskable interrupts that generate internal RST instructions.
- Input/output lines for serial data transfer.

Programming for the 8085

For the programmer, the new features of the 8085 are summarized in the two new instructions SIM and RIM. These instructions differ from the 8080 instructions in that each has multiple functions. The SIM instruction sets the interrupt mask and/or writes out a bit of serial data. The programmer must place the desired interrupt mask and/or serial output in the accumulator prior to execution of the SIM instruction. The RIM instruction reads a bit of serial data if one is present and the interrupt mask into the accumulator. Details of these instructions are covered in Chapter 3.

Despite the new interrupt features of the 8085, programming for interrupts is little changed. Notice, however, that 8085 hardware interrupt RESTART addresses fall between the existing 8080 RESTART addresses. Therefore, only four bytes are available for certain RST instructions. Also, the TRAP interrupt input is non-maskable and cannot be disabled. If your application uses this input, be certain to provide an interrupt routine for it.

The interrupts have the following priority:

TRAP	highest
RST7.5	
RST6.5	
RST5.5	
INTR	lowest

When more than one interrupt is pending, the processor always recognizes the higher priority interrupt first. These priorities apply only to the sequence in which interrupts are recognized. Program routines that service interrupts have no special priority. Thus, an RST5.5 interrupt can interrupt the service routine for an RST7.5 interrupt. If you want to protect a service routine from interruption, either disable the interrupt system (DI instruction), or mask out other potential interrupts (SIM instruction).

Conditional Instructions

Execution of conditional instructions on the 8085 differs from the 8080. The 8080 fetches all three instruction bytes whether or not the condition is satisfied. The 8085 evaluates the condition while it fetches the second instruction byte. If the specified condition is not satisfied, the 8085 skips over the third instruction byte and immediately fetches the next instruction. Skipping the unnecessary byte allows for faster execution.

2. ASSEMBLY LANGUAGE CONCEPTS

INTRODUCTION

Just as the English language has its rules of grammar, assembly language has certain coding rules. The source line is the assembly language equivalent of a sentence.

This assembler recognizes three types of source lines: instructions, directives, and controls. This manual describes instructions and directives. Controls are described in the operator's manual for your version of the assembler.

This chapter describes the general rules for coding source lines. Specific instructions (see Chapter 3) and directives (see Chapters 4 and 5) may have specific coding rules. Even so, the coding of such instructions and directives must conform to the general rules in this chapter.

SOURCE LINE FORMAT

Assembly language instructions and assembler directives may consist of up to four fields, as follows:

```
{Label:}     Opcode     Operand     ;Comment
{Name  }
```

The fields may be separated by any number of blanks, but must be separated by at least one delimiter. Each instruction and directive must be entered on a single line terminated by a carriage return and a line feed. No continuation lines are possible, but you may have lines consisting entirely of comments.

Character Set

The following characters are legal in assembly language source statements:

- The letters of the alphabet, A through Z. Both upper- and lower-case letters are allowed. Internally, the assembler treats all letters as though they were upper-case, but the characters are printed exactly as they were input in the assembly listing.

- The digits 0 through 9.

- The following special characters:

Character	*Meaning*
+	Plus sign
-	Minus sign
*	Asterisk
/	Slash
,	Comma
(	Left parenthesis
)	Right parenthesis
'	Single quote
&	Ampersand
:	Colon
$	Dollar sign
@	Commercial 'at' sign
?	Question mark
=	Equal sign
<	Less than sign
>	Greater than sign
%	Percent sign
!	Exclamation point
blank	Blank or space
;	Semicolon
.	Period
CR	Carriage return
FF	Form feed
HT	Horizontal tab

- In addition, any ASCII character may appear in a string enclosed in single quotes or in a comment.

Delimiters

Certain characters have special meaning to the assembler in that they function as delimiters. Delimiters define the end of a source statement, a field, or a component of a field. The following list defines the delimiters recognized by the assembler. Notice that many delimiters are related to the macro feature explained in Chapter 5. Delimiters used for macros are shown here so that you will not accidentally use a delimiter improperly. Refer to Chapter 5 for a description of macros.

Character(s)	*Meaning*	*Use*
blank	one or more blanks	field separator or symbol terminator
,	comma	separate operands in the operands field, including macro parameters
'...'	pair of single quote characters	delimit a character string
(...)	pair of parentheses	delimit an expression
CR	carriage return	statement terminator
HT	horizontal tab	field separator or symbol terminator
;	semicolon	comment field delimiter
:	colon	delimiter for symbols used as labels
&	ampersand	delimit macro prototype text or formal parameters for concatenation
<...>	pair of angle brackets	delimit macro parameter text which contains commas or embedded blanks; also used to delimit a parameter list
%	percent sign	delimit a macro parameter that is to be evaluated prior to substitution
!	exclamation point	an escape character used to pass the following character as part of a macro parameter when the character might otherwise be interpreted as a delimiter
;;	double semicolon	delimiter for comments in macro definitions when the comment is to be suppressed when the macro is expanded

Label/Name Field

Labels are always optional. An instruction label is a symbol name whose value is the location where the instruction is assembled. A label may contain from one to six alphanumeric characters, but the first character must be alphabetic or the special characters '?' or '@'. The label name must be terminated with a colon. A symbol used as a label can be defined only once in your program. (See 'Symbols and Symbol Tables' later in this chapter.)

Alphanumeric characters include the letters of the alphabet, the question mark character, and the decimal digits 0 through 9.

A name is required for the SET, EQU, and MACRO directives. Names follow the same coding rules as labels, except that they must be terminated with a blank rather than a colon. The label/name field must be empty for the LOCAL and ENDM directives.

Opcode Field

This required field contains the mnemonic operation code for the 8080/8085 instruction or assembler directive to be performed.

Operand Field

The operand field identifies the data to be operated on by the specified opcode. Some instructions require no operands. Others require one or two operands. As a general rule, when two operands are required (as in data transfer and arithmetic operations), the first operand identifies the destination (or target) of the operation's result, and the second operand specifies the source data.

Examples:

```
MOV     A,C              ;MOVE CONTENTS OF REG C TO ACCUMULATOR
MVI     A,'B'            ;MOVE B TO ACCUMULATOR
```

Comment Field

The optional comment field may contain any information you deem useful for annotating your program. The only coding requirement for this field is that it be preceded by a semicolon. Because the semicolon is a delimiter, there is no need to separate the comment from the previous field with one or more spaces. However, spaces are commonly used to improve the readability of the comment. Although comments are always optional, you should use them liberally since it is easier to debug and maintain a well documented program.

CODING OPERAND FIELD INFORMATION

There are four types of information (a through d in the following list) that may be requested as items in the operand field; the information may be specified in nine ways, each of which is described below.

OPERAND FIELD INFORMATION

Information required	*Ways of specifying*
(a) Register	(1) Hexadecimal Data
(b) Register Pair	(2) Decimal Data
(c) Immediate Data	(3) Octal Data
(d) 16-bit Address	(4) Binary Data
	(5) Location Counter ($)
	(6) ASCII Constant
	(7) Labels assigned values
	(8) Labels of instructions or data
	(9) Expressions

Hexadecimal Data. Each hexadecimal number must begin with a numeric digit (0 through 9) and must be followed by the letter H.

Label	*Opcode*	*Operand*	*Comment*
HERE:	MVI	C,0BAH	;LOAD REG C WITH HEX BA

Decimal Data. Each decimal number may be identified by the letter D immediately after its last digit or may stand alone. Any number not specifically identified as hexadecimal, octal, or binary is assumed to be decimal. Thus, the following statements are equivalent:

Label	*Opcode*	*Operand*	*Comment*
ABC:	MVI	E,15	;LOAD E WITH 15 DECIMAL
	MVI	E,15D	

Octal Data. Each octal number must be followed by the letter O or the letter Q.

Label	*Opcode*	*Operand*	*Comment*
LABEL:	MVI	A,72Q	;LOAD OCTAL 72 INTO ACCUM

Binary Data. Each binary number must be followed by the letter B.

Label	*Opcode*	*Operand*	*Comment*
NOW:	MVI	D,11110110B	;LOAD REGISTER D
			;WITH 0F6H

Location Counter. The $ character refers to the current location counter. The location counter contains the address where the current instruction or data statement will be assembled.

Label	*Opcode*	*Operand*	*Comment*
GO:	JMP	$+6	;JUMP TO ADDRESS 6 BYTES BEYOND ;THE FIRST BYTE OF THIS ;INSTRUCTION

ASCII Constant. One or more ASCII characters enclosed in single quotes define an ASCII constant. Two successive single quotes must be used to represent one single quote within an ASCII constant.

Label	*Opcode*	*Operand*	*Comment*
	MVI	E,'*'	;LOAD E REG WITH 8-BIT ASCII ;REPRESENTATION OF *
DATE:	DB	'TODAY''S DATE'	

Labels Assigned Values. The SET and EQU directives can assign values to labels. In the following example, assume that VALUE has been assigned the value 9FH; the two statements are equivalent:

Label	*Opcode*	*Operand*	*Comment*
A1:	MVI	D,9FH	
A2:	MVI	D,VALUE	

Labels of Instruction or Data. The label assigned to an instruction or a data definition has as its value the address of the first byte of the instruction or data. Instructions elsewhere in the program can refer to this address by its symbolic label name.

Label	*Opcode*	*Operand*	*Comments*
HERE:	JMP	THERE	;JUMP TO INSTRUCTION AT THERE
	.		
	.		
THERE:	MVI	D,9FH	

Expressions. All of the operand types discussed previously can be combined by operators to form an expression. In fact, the example given for the location counter ($+6) is an expression that combines the location counter with the decimal number 6.

Because the rules for coding expressions are rather extensive, further discussion of expressions is deferred until later in this chapter.

Instructions as Operands. One operand type was intentionally omitted from the list of operand field information: Instructions enclosed in parentheses may appear in the operands field. The operand has the value of the left-most byte of the assembled instruction.

Label	*Opcode*	*Operand*
INS:	DB	(ADD C)

The statement above defines a byte with the value 81H (the object code for an ADD C instruction). Such coding is typically used where the object program modifies itself during execution, a technique that is strongly discouraged.

Register-Type Operands. Only instructions that allow registers as operands may have register-type operands. Expressions containing register-type operands are flagged as errors. Thus, an instruction like

```
JMP  A
```

is flagged as an illegal use of a register.

The only assembler directives that may contain register-type operands are EQU, SET, and actual parameters in macro calls. Registers can be assigned alternate names only by EQU or SET.

TWO'S COMPLEMENT REPRESENTATION OF DATA

Any 8-bit byte contains one of the 256 possible combinations of zeros and ones. Any particular combination may be interpreted in a number of ways. For example, the code 1FH may be interpreted as an instruction (Rotate Accumulator Right Through Carry), as the hexadecimal value 1F, the decimal value 31, or simply the bit pattern 00011111.

Arithmetic instructions assume that the data bytes upon which they operate are in the 'two's complement' format. To understand why, let us first examine two examples of decimal arithmetic:

```
  35      35
 -12     +88
 ---     ---
  23     123
```

Notice that the results of the two examples are equal if we disregard the carry out of the high order position in the second example. The second example illustrates subtraction performed by adding the ten's complement of the subtrahend (the bottom number) to the minuend (the top number). To form the ten's complement of a decimal number, first subtract each digit of the subtrahend from 9 to form the nine's complement; then add one to the result to form the ten's complement. Thus, 99–12=87; 87+1=88, the ten's complement of 12.

The ability to perform subtraction with a form of addition is a great advantage in a computer since fewer circuits are required. Also, arithmetic operations within the computer are binary, which simplifies matters even more.

The processor forms the two's complement of a binary value simply by reversing the value of each bit and then adding one to the result. Any carry out of the high order bit is ignored when the complement is formed. Thus, the subtraction shown previously is performed as follows:

```
  35 = 0010 0011                    0010 0011
 -12 = 0000 1100 = 1111  0011      +1111 0100
 ---                               ----------
  23               +         1    1 0001 0111 = 23
                   -----------
                    1111 0100
```

Again, by disregarding the carry out of the high order position, the subtraction is performed through a form of addition. However, if this operation were performed by the 8080 or the 8085, the carry flag would be set OFF at the end of the subtraction. This is because the processors complement the carry flag at the end of a subtract operation so that it can be used as a 'borrow' flag in multibyte subtractions. In the example shown, no borrow is required, so the carry flag is set OFF. By contrast, the carry flag is set ON if we subtract 35 from 12:

```
  12 = 0000 1100                    0000 1100
 -35 = 0010 0011 = 1101  1100      +1101 1101
 ---                               ----------
                   +         1      1110 1001 = 233 or -105
                   -----------
                    1101  1101
```

In this case, the absence of a carry indicates that a borrow is required from the next higher order byte, if any. Therefore, the processor sets the carry flag ON. Notice also that the result is stored in a complemented form. If you want to interpret this result as a decimal value, you must again form its two's complement:

```
   1110 1001 = 0001  0110
               +        1
               ----------
               0001  0111 = 23
```

Two's complement numbers may also be signed. When a byte is interpreted as a signed two's complement number, the high order bit indicates the sign. A zero in this bit indicates a positive number, a one a negative number. The seven low order bits provide the magnitude of the number. Thus, 0111 1111 equals +127.

At the beginning of this description of two's complement arithmetic, it was stated that any 8-bit byte may contain one of the 256 possible combinations of zeros and ones. It must also be stated that the proper interpretation of data is a programming responsibility.

As an example, consider the compare instruction. The compare logic considers only the raw bit values of the items being compared. Therefore, a negative two's complement number always compares higher than a positive number, because the negative number's high order bit is always ON. As a result, the meanings of the flags set by the compare instruction are reversed. Your program must account for this condition.

SYMBOLS AND SYMBOL TABLES

Symbolic Addressing

If you have never done symbolic programming before, the following analogy may help clarify the distinction between a symbolic and an absolute address.

The locations in program memory can be compared to a cluster of post office boxes. Suppose Richard Roe rents box 500 for two months. He can then ask for his letters by saying 'Give me the mail in box 500,' or 'Give me the mail for Roe.' If Donald Smith later rents box 500, he too can ask for his mail by either box number 500 or by his name. The content of the post office box can be accessed by a fixed, *absolute* address (500) or by a *symbolic, variable* name. The postal clerk correlates the symbolic names and their absolute values in his log book. The assembler performs the same function, keeping track of symbols and their values in a *symbol table.* Note that you do not have to assign values to symbolic addresses. The assembler references its location counter during the assembly process to calculate these addresses for you. (The location counter does for the assembler what the program counter does for the microcomputer. It tells the assembler where the next instruction or operand is to be placed in memory.)

Symbol Characteristics

A symbol can contain one to six alphabetic (A-Z) or numeric (0-9) characters (with the first character alphabetic) or the special character '?' or '@'. A dollar sign can be used as a symbol to denote the value currently in the location counter. For example, the command

```
JMP     $+6
```

forces a jump to the instruction residing six memory locations higher than the JMP instruction. Symbols of the form '??nnn' are generated by the assembler to uniquely name symbols local to macros.

The assembler regards symbols as having the following attributes: reserved or user-defined; global or limited; permanent or redefinable; and absolute or relocatable.

Reserved, User-Defined, and Assembler-Generated Symbols

Reserved symbols are those that already have special meaning to the assembler and therefore cannot appear as user-defined symbols. The mnemonic names for machine instructions and the assembler directives are all reserved symbols.

The following instruction operand symbols are also reserved:

Symbol	*Meaning*
$	Location counter reference
A	Accumulator register
B	Register B or register pair B and C
C	Register C
D	Register D or register pair D and E
E	Register E
H	Register H or register pair H and L
L	Register L
SP	Stack pointer register
PSW	Program status word (Contents of A and status flags)
M	Memory reference code using address in H and L
STACK	Special relocatability feature
MEMORY	Special relocatability feature

NOTE

The STACK and MEMORY symbols are fully discussed in Chapter 4.

User-defined symbols are symbols you create to reference instruction and data addresses. These symbols are defined when they appear in the label field of an instruction or in the name field of EQU, SET, or MACRO directives (see Chapters 4 and 5).

Assembler-generated symbols are created by the assembler to replace user-defined symbols whose scope is limited to a macro definition.

Global and Limited Symbols

Most symbols are global. This means that they have meaning throughout your program. Assume, for example, that you assign the symbolic name RTN to a routine. You may then code a jump or a call to RTN from any point in your program. If you assign the symbolic name RTN to a second routine, an error results since you have given multiple definitions to the same name.

Certain symbols have meaning only within a macro definition or within a call to that macro; these symbols are 'local' to the macro. Macros require local symbols because the same macro may be used many times in the program. If the symbolic names within macros were global, each use of the macro (except the first) would cause multiple definitions for those symbolic names.

See Chapter 5 for additional information about macros.

Permanent and Redefinable Symbols

Most symbols are permanent since their value cannot change during the assembly operation. Only symbols defined with the SET and MACRO assembler directives are redefinable.

Absolute and Relocatable Symbols

An important attribute of symbols with this assembler is that of relocatability. Relocatable programs are assembled relative to memory location zero. These programs are later relocated to some other set of memory locations. Symbols with addresses that change during relocation are relocatable symbols. Symbols with addresses that do not change during relocation are absolute symbols. This distinction becomes important when the symbols are used within expressions, as will be explained later.

External and public symbols are special types of relocatable symbols. These symbols are required to establish program linkage when several relocatable program modules are bound together to form a single application program. External symbols are those used in the current program module, but defined in another module. Such symbols must appear in an EXTRN statement, or the assembler will flag them as undefined.

Conversely, PUBLIC symbols are defined in the current program module, but may be accessed by other modules. The addresses for these symbols are resolved when the modules are bound together.

Absolute and relocatable symbols may both appear in a relocatable module. References to any of the assembler-defined registers A through E, H and L, PSW, SP, and M are absolute since they refer to hardware locations. But these references are valid in any module.

ASSEMBLY-TIME EXPRESSION EVALUATION

An expression is a combination of numbers, symbols, and operators. Each element of an expression is a term.

Expressions, like symbols, may be absolute or relocatable. For the sake of readers who do not require the relocation feature, absolute expressions are described first. However, users of relocation should read all the following.

Operators

The assembler includes five groups of operators which permit the following assembly-time operations: arithmetic operations, shift operations, logical operations, compare operations, and byte isolation operations. It is important to keep in mind that these are all assembly-time operations. Once the assembler has evaluated an expression, it becomes a permanent part of your program. Assume, for example, that your program defines a list of ten constants starting at the label LIST; the following instruction loads the address of the seventh item in the list into the H and L registers:

```
LXI  H,LIST+6
```

Notice that LIST addresses the first item, LIST+1 the second, and so on.

Arithmetic Operators

The arithmetic operators are as follows:

Operator	*Meaning*
+	Unary or binary addition
–	Unary or binary subtraction
*	Multiplication
/	Division. Any remainder is discarded (7/2=3). Division by zero causes an error.
MOD	Modulo. Result is the remainder caused by a division operation. (7 MOD 3=1)

Examples:

The following expressions generate the bit pattern for the ASCII character A:

```
5+30*2
(25/5)+30*2
5+(–30*–2)
```

Notice that the MOD operator must be separated from its operands by spaces:

```
NUMBR   MOD   8
```

Assuming that NUMBR has the value 25, the previous expression evaluates to the value 1.

Shift Operators

The shift operators are as follows:

Operator	*Meaning*
y SHR x	Shift operand 'y' to the right 'x' bit positions.
y SHL x	Shift operand 'y' to the left 'x' bit positions.

The shift operators do not wraparound any bits shifted out of the byte. Bit positions vacated by the shift operation are zero-filled. Notice that the shift operator must be separated from its operands by spaces.

Example:

Assume that NUMBR has the value 0101 0101. The effects of the shift operators is as follows:

```
NUMBR   SHR   2        0001 0101

NUMBR   SHL   1        1010 1010
```

Notice that a shift one bit position to the left has the effect of multiplying a value by two; a shift one bit position to the right has the effect of dividing a value by two.

Logical Operators

The logical operators are as follows:

Operator	*Meaning*
NOT	Logical one's complement
AND	Logical AND (=1 if both ANDed bits are 1)
OR	Logical OR (=1 if either ORed bit is 1)
XOR	Logical EXCLUSIVE OR (=1 if bits are different)

The logical operators act only upon the least significant bit of values involved in the operation. Also, these operators are commonly used in conditional IF directives. These directives are fully explained in Chapter 4.

Example:

The following IF directive tests the least significant bit of three items. The assembly language code that follows the IF is assembled only if the condition is TRUE. This means that all three fields must have a one bit in the least significant bit position.

```
IF FLD1 AND FLD2 AND FLD3
        .
        .
        .
```

Compare Operators

The compare operators are as follows:

Operator	*Meaning*
EQ	Equal
NE	Not equal
LT	Less than
LE	Less than or equal
GT	Greater than
GE	Greater than or equal
NUL	Special operator used to test for null (missing) macro parameters

The compare operators yield a yes-no result. Thus, if the evaluation of the relation is TRUE, the value of the result is all ones. If false, the value of the result is all zeros. Relational operations are based strictly on magnitude comparisons of bit values. Thus, a two's complement negative number (which always has a one in its high order bit) is greater than a two's complement positive number (which always has a zero in its high order bit).

Since the NUL operator applies only to the macro feature, NUL is described in Chapter 5.

The compare operators are commonly used in conditional IF directives. These directives are fully explained in Chapter 4.

Notice that the compare operator must be separated from its operands by spaces.

Example:

The following IF directive tests the values of FLD1 and FLD2 for equality. If the result of the comparison is TRUE, the assembly language coding following the IF directive is assembled. Otherwise, the code is skipped over.

```
IF FLD1 EQ FLD2
     .
     .
     .
```

Byte Isolation Operators

The byte isolation operators are as follows:

Operator	*Meaning*
HIGH	Isolate high-order 8 bits of 16-bit value
LOW	Isolate low-order 8 bits of 16-bit value.

The assembler treats expressions as 16-bit addresses. In certain cases, you need to deal only with a part of an address, or you need to generate an 8-bit value. This is the function of the HIGH and LOW operators.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

NOTE

Any program segment containing a symbol used as the argument of a HIGH operator should be located only on a page boundary. This is done using the PAGE option with the CSEG or DSEG directives described in Chapter 4. Carries are not propagated from the low-order byte when the assembler object code is relocated and the carry flag will be lost. Using PAGE ensures that this flag is 0.

Examples:

Assume that ADRS is an address manipulated at assembly-time for building tables or lists of items that must all be below address 255 in memory. The following IF directive determines whether the high-order byte of ADRS is zero, thus indicating that the address is still less than 256:

```
IF HIGH ADRS EQ 0
        .
        .
        .
```

Permissible Range of Values

Internally, the assembler treats each term of an expression as a two-byte, 16-bit value. Thus, the maximum range of values is 0H through 0FFFFH. All arithmetic operations are performed using unsigned two's complement arithmetic. The assembler performs no overflow detection for two-byte values, so these values are evaluated modulo 64K.

Certain instructions require that their operands be an eight-bit value. Expressions for these instructions must yield values in the range −256 through +255. The assembler generates an error message if an expression for one of these instructions yields an out-of-range value.

NOTE

Only instructions that allow registers as operands may have register-type operands. Expressions containing register-type operands are flagged as errors. The only assembler directives that may contain register-type operands are EQU, SET, and actual parameters in macro calls. Registers can be assigned alternate names only by EQU or SET.

Precedence of Operators

Expressions are evaluated left to right. Operators with higher precedence are evaluated before other operators that immediately precede or follow them. When two operators have equal precedence, the left-most is evaluated first.

Parentheses can be used to override normal rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost are evaluated first.

```
15/3 + 18/9      = 5 + 2 = 7
15/(3 + 18/9)    = 15/(3 + 2) = 15/5 = 3
```

The following list describes the classes of operators in order of precedence:

- Parenthesized expressions
- NUL
- HIGH, LOW
- Multiplication/Division: *, /, MOD, SHL, SHR
- Addition/Subtraction: +, – (unary and binary)
- Relational Operators: EQ, LT, LE, GT, GE, NE
- Logical NOT
- Logical AND
- Logical OR, XOR

The relational, logical, and HIGH/LOW operators must be separated from their operands by at least one blank.

Relocatable Expressions

Determining the relocatability of an expression requires that you understand the relocatability of each term used in the expression. This is easier than it sounds since the number of allowable operators is substantially reduced. But first it is necessary to know what determines whether a symbol is absolute or relocatable.

Absolute symbols can be defined two ways:

- A symbol that appears in a label field when the ASEG directive is in effect is an absolute symbol.
- A symbol defined as equivalent to an absolute expression using the SET or EQU directive is an absolute symbol.

Relocatable symbols can be defined a number of ways:

- A symbol that appears in a label field when the DSEG or CSEG directive is in effect is a relocatable symbol.
- A symbol defined as equivalent to a relocatable expression using the SET or EQU directive is relocatable.
- The special assembler symbols STACK and MEMORY are relocatable.
- External symbols are considered relocatable.
- A reference to the location counter (specified by the $ character) is relocatable when the CSEG or DSEG directive is in effect.

The expressions shown in the following list are the only expressions that yield a relocatable result. Assume that ABS is an absolute symbol and RELOC is a relocatable symbol:

ABS + RELOC
RELOC + ABS
RELOC – ABS

$\begin{Bmatrix}\text{HIGH}\\ \text{LOW}\end{Bmatrix}$ RELOC + ABS

$\begin{Bmatrix}\text{HIGH}\\ \text{LOW}\end{Bmatrix}$ RELOC – ABS

RELOC + $\begin{Bmatrix}\text{HIGH}\\ \text{LOW}\end{Bmatrix}$ ABS

RELOC – $\begin{Bmatrix}\text{HIGH}\\ \text{LOW}\end{Bmatrix}$ ABS

Remember that numbers are absolute terms. Thus the expression RELOC – 100 is legal, but 100 – RELOC is not.

When two relocatable symbols have both been defined with the same type of relocatability, they may appear in certain expressions that yield an absolute result. Symbols have the same type of relocatability when both are relative to the CSEG location counter, both are relative to the DSEG location counter, both are relative to MEMORY, or both are relative to STACK. The following expressions are valid and produce absolute results:

RELOC1 – RELOC2

RELOC1 { EQ / LT / LE / GT / GE / NE } RELOC2

Relocatable symbols may not appear in expressions with any other operators.

The following list shows all possible combinations of operators with absolute and relocatable terms. An A in the table indicates that the resulting address is absolute; an R indicates a relocatable address; an I indicates an illegal combination. Notice that only one term may appear with the last five operators in the list.

Operator	X absolute Y absolute	X absolute Y relocatable	X relocatable Y absolute	X relocatable Y relocatable
X + Y	A	R	R	I
X – Y	A	I	R	A
X * Y	A	I	I	I
X / Y	A	I	I	I
X MOD Y	A	I	I	I
X SHL Y	A	I	I	I
X SHR Y	A	I	I	I
X EQ Y	A	I	I	A
X LT Y	A	I	I	A
X LE Y	A	I	I	A
X GT Y	A	I	I	A
X GE Y	A	I	I	A
X NE Y	A	I	I	A
X AND Y	A	I	I	I
X OR Y	A	I	I	I
X XOR Y	A	I	I	I
NOT X	A	—	I	—
HIGH X	A	—	R	—
LOW X	A	—	R	—
unary+ X	A	—	R	—
unary– X	A	—	I	—

Chaining of Symbol Definitions

The ISIS-II 8080/8085 Macro Assembler is essentially a 2-pass assembler. All symbol table entries must be resolvable in two passes. Therefore,

```
X   EQU   Y
Y   EQU   1
```

is legal, but in the series

```
X   EQU   Y
Y   EQU   Z
Z   EQU   1
```

the first line is illegal as X cannot be resolved in two passes and remains undefined.

3. INSTRUCTION SET

HOW TO USE THIS CHAPTER

This chapter is a dictionary of 8080 and 8085 instructions. The instruction descriptions are listed alphabetically for quick reference. Each description is complete so that you are seldom required to look elsewhere for additional information.

This reference format necessarily requires repetitive information. If you are reading this manual to learn about the 8080 or the 8085, do not try to read this chapter from ACI (add immediate with Carry) to XTHL (exchange top of stack with H and L registers). Instead, read the description of the processor and instruction set in Chapter 1 and the programming examples in Chapter 6. When you begin to have questions about particular instructions, look them up in this chapter.

TIMING INFORMATION

The instruction descriptions in this manual do not explicitly state execution timings. This is because the basic operating speed of your processor depends on the clock frequency used in your system.

The 'state' is the basic unit of time measurement for the processor. A state may range from 480 nanoseconds (320 nanoseconds on the 8085) to 2 microseconds, depending on the clock frequency. When you know the length of a state in your system, you can determine an instruction's basic execution time by multiplying that figure by the number of states required for the instruction.

Notice that two sets of cycle/state specifications are given for 8085 conditional call and jump instructions. This is because the 8085 fetches the third instruction byte only if it is actually needed; i.e., the specified condition is satisfied.

This basic timing factor can be affected by the operating speed of the memory in your system. With a fast clock cycle and a slow memory, the processor can outrun the memory. In this case, the processor must wait for the memory to deliver the desired instruction or data. In applications with critical timing requirements, this wait can be significant. Refer to the appropriate manufacturer's literature for memory timing data.

ACI — ADD IMMEDIATE WITH CARRY

ACI adds the contents of the second instruction byte and the carry bit to the contents of the accumulator and stores the result in the accumulator.

Opcode	*Operand*
ACI	data

The operand specifies the actual data to be added to the accumulator except, of course, for the carry bit. Data may be in the form of a number, an ASCII constant, the label of a previously defined value, or an expression. The data may not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

1	1	0	0	1	1	1	0
data							

Cycles:	2
States:	7
Addressing:	immediate
Flags:	Z,S,P,CY,AC

Example:

Assume that the accumulator contains the value 14H and that the carry bit is set to one. The instruction ACI 66 has the following effect:

```
   Accumulator  =  14H  =  00010100
Immediate data  =  42H  =  01000010
                 Carry  =         1
                           --------
                           01010111  =  57H
```

ADC — ADD WITH CARRY

The ADC instruction adds one byte of data plus the setting of the carry flag to the contents of the accumulator. The result is stored in the accumulator. ADC then updates the setting of the carry flag to indicate the outcome of the operation.

The ADC instruction's use of the carry bit enables the program to add multi-byte numeric strings.

Add Register to Accumulator with Carry

Opccde	*Operand*
ADC	reg

The operand must specify one of the registers A through E, H or L. This instruction adds the contents of the specified register and the carry bit to the accumulator and stores the result in the accumulator.

1	0	0	0	1	S	S	S

Cycles:	1
States:	4
Addressings:	register
Flags:	Z,S,P,CY,AC

Add Memory to Accumulator with Carry

Opcode	*Operand*
ADC	M

This instruction adds the contents of the memory location addressed by the H and L registers and the carry bit to the accumulator and stores the result in the accumulator. M is a symbolic reference to the H and L registers.

1	0	0	0	1	1	1	0

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Example:

Assume that register C contains 3DH, the accumulator contains 42H, and the carry bit is set to zero. The instruction ADC C performs the addition as follows:

```
  3DH  = 00111101
  42H  = 01000010
CARRY  =        0
         --------
         01111111 = 7FH
```

The condition flags are set as follows:

Carry	= 0
Sign	= 0
Zero	= 0
Parity	= 0
Aux. Carry	= 0

If the carry bit is set to one, the instruction has the following results:

```
  3DH = 00111101
  42H = 01000010
CARRY =        1
        --------
        10000000 = 80H
```

Carry	= 0
Sign	= 1
Zero	= 0
Parity	= 0
Aux. Carry	= 1

ADD

The ADD instruction adds one byte of data to the contents of the accumulator. The result is stored in the accumulator. Notice that the ADD instruction excludes the carry flag from the addition but sets the flag to indicate the outcome of the operation.

Add Register to Register

Opcode	*Operand*
ADD	reg

The operand must specify one of the registers A through E, H or L. The instruction adds the contents of the specified register to the contents of the accumulator and stores the result in the accumulator.

1	0	0	0	0	S	S	S

Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

Add From Memory

Opcode	*Operand*
ADD	M

This instruction adds the contents of the memory location addressed by the H and L registers to the contents of the accumulator and stores the result in the accumulator. M is a symbolic reference to the H and L registers.

1	0	0	0	0	1	1	0

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Examples:

Assume that the accumulator contains 6CH and register D contains 2EH. The instruction ADD D performs the addition as follows:

```
2EH = 00101110
6CH = 01101100
--------------
9AH = 10011010
```

The accumulator contains the value 9AH following execution of the ADD D instruction. The contents of the D register remain unchanged. The condition flags are set as follows:

Carry	=	0
Sign	=	1
Zero	=	0
Parity	=	1
Aux. Carry	=	1

The following instruction doubles the contents of the accumulator:

ADD A

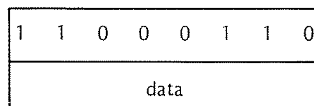
ADI — ADD IMMEDIATE

ADI adds the contents of the second instruction byte of the contents of the accumulator and stores the result in the accumulator.

Opcode	*Operand*
ADI	data

The operand specifies the actual data to be added to the accumulator. This data may be in the form of a number, an ASCII constant, the label of a previously defined value, or an expression. The data may not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.



Cycles:	2
States:	7
Addressing:	immediate
Flags:	Z,S,P,CY,AC

Example:

Assume that the accumulator contains the value 14H. The instruction ADI 66 has the following effect:

```
     Accumulator  =  14H  =  00010100
  Immediate data  =  42H  =  01000010
                             01010110  =  56H
```

Notice that the assembler converts the decimal value 66 into the hexadecimal value 42.

ANA — LOGICAL AND WITH ACCUMULATOR

ANA performs a logical AND operation using the contents of the specified byte and the accumulator. The result is placed in the accumulator.

Summary of Logical Operations

AND produces a one bit in the result only when the corresponding bits in the test data and the mask data are ones.

OR produces a one bit in the result when the corresponding bits in either the test data or the mask data are ones.

Exclusive OR produces a one bit only when the corresponding bits in the test data and the mask data are different; i.e. a one bit in either the test data or the mask data — but not both — produces a one bit in the result.

```
   AND          OR          EXCLUSIVE OR

1010 1010    1010 1010     1010    1010
0000 1111    0000 1111     0000    1111
0000 1010    1010 1111     1010    0101
```

AND Register with Accumulator

Opcode	*Operand*
ANA	reg

The operand must specify one of the registers A through E, H or L. This instruction ANDs the contents of the specified register with the accumulator and stores the result in the accumulator. The carry flag is reset to zero.

1	0	1	0	0	S	S	S

Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

AND Memory with Accumulator

Opcode	*Operand*
ANA	M

This instruction ANDs the contents of the specified memory location with the accumulator and stores the result in the accumulator. The carry flag is reset to zero.

1	0	1	0	0	1	1	0

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Example:

Since any bit ANDed with a zero produces a zero and any bit ANDed with a one remains unchanged, AND is frequently used to zero particular groups of bits. The following example ensures that the high-order four bits of the accumulator are zero, and the low-order four bits are unchanged. Assume that the C register contains 0FH:

Accumulator	=	1 1 1 1	1 1 0 0	=	0FCH
C Register	=	0 0 0 0	1 1 1 1	=	0FH
		0 0 0 0	1 1 0 0	=	0CH

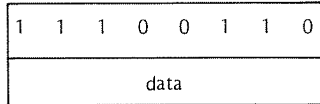
ANI — AND IMMEDIATE WITH ACCUMULATOR

ANI performs a logical AND operation using the contents of the second byte of the instruction and the accumulator. The result is placed in the accumulator. ANI also resets the carry flag to zero.

Opcode	*Operand*
ANI	data

The operand must specify the data to be used in the AND operation. This data may be in the form of a number, an ASCII constant, the label of some previously defined value, or an expression. The data may not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.



Cycles:	2
States:	7
Addressing:	immediate
Flags:	Z,S,P,CY,AC

Summary of Logical Operations

AND produces a one bit in the result only when the corresponding bits in the test data and the mask data are ones.

OR produces a one bit in the result when the corresponding bits in either the test data or the mask data are ones.

Exclusive OR produces a one bit only when the corresponding bits in the test data and the mask data are different; i.e. a one bit in either the test data or the mask data — but not both — produces a one bit in the result.

AND	OR	EXCLUSIVE OR
1010 1010	1010 1010	1010 1010
0000 1111	0000 1111	0000 1111
0000 1010	1010 1111	1010 0101

Example:

The following instruction is used to reset OFF bit six of the byte in the accumulator:

```
ANI        10111111B
```

Since any bit ANDed with a one remains unchanged and a bit ANDed with a zero is rest to zero, the ANI instruction shown above sets bit six OFF and leaves the others unchanged. This technique is useful when a program uses individual bits as status flags.

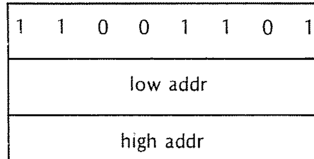
CALL

The CALL instruction combines functions of the PUSH and JMP instructions. CALL pushes the contents of the program counter (the address of the next sequential instruction) onto the stack and then jumps to the address specified in the CALL instruction.

Each CALL instruction or one of its variants implies the use of a subsequent RET (return) instruction. When a call has no corresponding return, excess addresses are built up in the stack.

Opcode	*Operand*
CALL	address

The address may be specified as a number, a label, or an expression. (The label is most common.) The assembler inverts the high and low address bytes when it assembles the instruction.



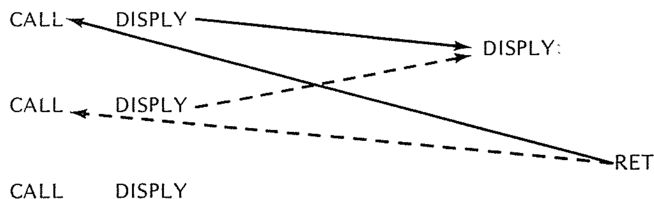
Cycles:	5
States:	17 (18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

When a given coding sequence is required several times in a program, you can usually conserve memory by coding the sequence as a subroutine invoked by the CALL instruction or one of its variants. For example, assume that an application drives a six-digit LED display; the display is updated as a result of an operator input or because of two different calculations that occur in the program. The coding required to drive the display can be included in-line at each of the three points where it is needed, or it can be coded as a subroutine. If the label DISPLY is assigned to the first instruction of the display driver, the following CALL instruction is used to invoke the display subroutine:

```
CALL        DISPLY
```

This CALL instruction pushes the address of the next program instruction onto the stack and then transfers control to the DISPLY subroutine. The DISPLY subroutine must execute a return instruction or one of its variants to resume normal program flow. The following is a graphic illustration of the effect of CALL and return instructions:



Consideration for Using Subroutines

The larger the code segment to be repeated and the greater the number of repetitions, the greater the potential memory savings of using a subroutine. Thus, if the display driver in the previous example requires one hundred

bytes, coding it in-line would require three hundred bytes. Coded as a subroutine, it requires one hundred bytes plus nine bytes for the three CALL instructions.

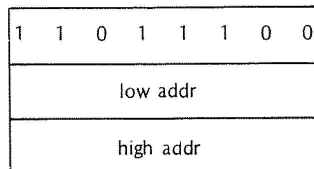
Notice that subroutines require the use of the stack. This requires the application to include random access memory for the stack. When an application has no other need for random access memory, the system designer might elect to avoid the use of subroutines.

CC — CALL IF CARRY

The CC instruction combines functions of the JC and PUSH instructions. CC tests the setting of the carry flag. If the flag is set to one, CC pushes the contents of the program counter onto the stack and then jumps to the address specified in bytes two and three of the CC instruction. If the flag is reset to zero, program execution continues with the next sequential instruction.

Opcode	*Operand*
CC	address

Although the use of a label is most common, the address may also be specified as a number or expression.



Cycles:	3 or 5 (2 or 5 on 8085)
States:	11 or 17 (9 or 18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

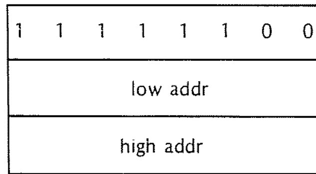
For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

CM — CALL IF MINUS

The CM instruction combines functions of the JM and PUSH instructions. CM tests the setting of the sign flag. If the flag is set to one (indicating that the contents of the accumulator are minus), CM pushes the contents of the program counter onto the stack and then jumps to the address specified by the CM instruction. If the flag is set to zero, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
CM	address

Although the use of a label is most common, the address may also be specified as a number or an expression.



Cycles:	3 or 5 (2 or 5 on 8085)
States:	11 or 17 (9 or 18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

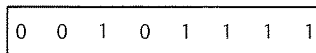
For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

CMA — COMPLEMENT ACCUMULATOR

CMA complements each bit of the accumulator to produce the one's complement. All condition flags remain unchanged.

Opcode	*Operand*
CMA	

Operands are not permitted with the CMA instruction.



Cycles:	1
States:	4
Flags:	none

To produce the two's complement, add one to the contents of the accumulator after the CMA instructions has been executed.

Example:

Assume that the accumulator contains the value 51H; when complemented by CMA, it becomes 0AEH:

51H	=	01010001
0AEH	=	10101110

CMC — COMPLEMENT CARRY

If the carry flag equals zero, CMC sets it to one. If the carry flag is one, CMC resets it to zero. All other flags remain unchanged.

Opcode	*Operand*
CMC	

Operands are not permitted with the CMC instruction.

0	0	1	1	1	1	1	1

Cycles: 1
States: 4
Flags: CY only

Example:

Assume that a program uses bit 7 of a byte to control whether a subroutine is called. To test the bit, the program loads the byte into the accumulator, rotates bit 7 into the carry flag, and executes a CC (Call if Carry) instruction. Before returning to the calling program, the subroutine reinitializes the flag byte using the following code:

```
CMC    ;SET BIT 7 OFF
RAR    ;ROTATE BIT 7 INTO ACCUMULATOR
RET    ;RETURN
```

CMP — COMPARE WITH ACCUMULATOR

CMP compares the specified byte with the contents of the accumulator and indicates the result by setting the carry and zero flags. The values being compared remain unchanged.

The zero flag indicates equality. No carry indicates that the accumulator is greater than the specified byte; a carry indicates that the accumulator is less than the byte. However, the meaning of the carry flag is reversed when the values have different signs or one of the values is complemented.

The program tests the condition flags using one of the conditional Jump, Call, or Return instructions. For example, JZ (Jump if Zero) tests for equality.

Functional Description:

Comparisons are performed by subtracting the specified byte from the contents of the accumulator, which is why the zero and carry flags indicate the result. This subtraction uses the processor's internal registers so that source data is preserved. Because subtraction uses two's complement addition, the CMP instruction recomplements the carry flag generated by the subtraction.

Compare Register with Accumulator

Opcode	*Operand*
CMP	reg

The operand must name one of the registers A through E, H or L.

1	0	1	1	1	S	S	S

Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

Compare Memory with Accumulator

Opcode	*Operand*
CMP	M

This instruction compares the contents of the memory location addressed by the H and L registers with the contents of the accumulator. M is a symbolic reference to the H and L register pair.

1	0	1	1	1	1	1	0

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Example 1:

Assume that the accumulator contains the value 0AH and register E contains the value 05H. The instruction CMP E performs the following internal subtraction (remember that subtraction is actually two's complement addition):

```
Accumulator      = 00001010
+(–E Register)   = 11111011
                   ________
                   00000101 +(–carry)
```

After the carry is complemented to account for the subtract operation, both the zero and carry bits are zero, thus indicating A greater than E.

Example 2:

Assume that the accumulator contains the value –1BH and register E contains 05H:

```
Accumulator      = 11100101
+(–E Register)   = 11111011
                   ________
                   11100000 +(–carry)
```

After the CMP instruction recomplements the carry flag, both the carry flag and zero flag are zero. Normally this indicates that the accumulator is greater than register E. However, the meaning of the carry flag is reversed since the values have different signs. The user program is responsible for proper interpretation of the carry flag.

CNC — CALL IF NO CARRY

The CNC instruction combines functions of the JNC and PUSH instructions. CNC tests the setting of the carry flag. If the flag is set to zero, CNC pushes the contents of the program counter onto the stack and then jumps to the address specified by the CNC instruction. If the flag is set to one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
CNC	address

Although the use of a label is most common, the address may also be specified as a number or an expression.

1	1	0	1	0	1	0	0
low addr							
high addr							

Cycles:	3 or 5 (2 or 5 on 8085)
States:	11 or 17 (9 or 18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

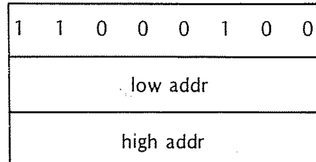
For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

CNZ — CALL IF NOT ZERO

The CNZ instruction combines functions of the JNZ and PUSH instructions. CNZ tests the setting of the zero flag. If the flag is off (indicating that the contents of the accumulator are other than zero), CNZ pushes the contents of the program counter onto the stack and then jumps to the address specified in the instruction's second and third bytes. If the flag is set to one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
CNZ	address

Although the use of a label is most common, the address may also be specified as a number or an expression.



Cycles:	3 or 5 (2 or 5 on 8085)
States:	11 or 17 (9 or 18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

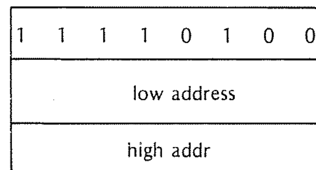
For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

CP — CALL IF POSITIVE

The CP instruction combines features of the JP and PUSH instructions. CP tests the setting of the sign flag. If the flag is set to zero (indicating that the contents of the accumulator are positive), CP pushes the contents of the program counter onto the stack and then jumps to the address specified by the CP instruction. If the flag is set to one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
CP	address

Although the use of a label is more common, the address may also be specified as a number or an expression.



Cycles:	3 or 5 (2 or 5 on 8085)
States:	11 or 17 (9 or 18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

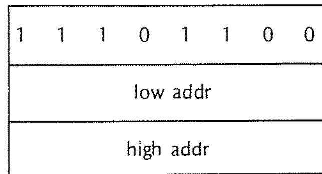
CPE — CALL IF PARITY EVEN

Parity is even if the byte in the accumulator has an even number of one bits. The parity flag is set to one to indicate this condition. The CPE and CPO instructions are useful for testing the parity of input data. However, the IN instruction does not set any of the condition flags. The flags can be set without altering the data by adding 00H to the contents of the accumulator.

The CPE instruction combines functions of the JPE and PUSH instructions. CPE tests the setting of the parity flag. If the flag is set to one, CPE pushes the contents of the program counter onto the stack and then jumps to the address specified by the CPE instruction. If the flag is set to zero, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
CPE	address

Although the use of a label is more common, the address may also be specified as a number or an expression.



Cycles:	3 or 5 (2 or 5 on 8085)
States:	11 or 17 (9 or 18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

CPI — COMPARE IMMEDIATE

CPI compares the contents of the second instruction byte with the contents of the accumulator and sets the zero and carry flags to indicate the result. The values being compared remain unchanged.

The zero flag indicates equality. No carry indicates that the contents of the accumulator are greater than the immediate data; a carry indicates that the accumulator is less than the immediate data. However, the meaning of the carry flag is reversed when the values have different signs or one of the values is complemented.

Opcode	*Operand*
CPI	data

The operand must specify the data to be compared. This data may be in the form of a number, an ASCII constant, the label of a previously defined value, or an expression. The data may not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

1	1	1	1	1	1	1	0
data							

Cycles: 2
States: 7
Addressing: register indirect
Flags: Z,S,P,CY,AC

Example:

The instruction CPI 'C' compares the contents of the accumulator to the letter C (43H).

CPO — CALL IF PARITY ODD

Parity is odd if the byte in the accumulator has an odd number of one bits. The parity flag is set to zero to indicate this condition. The CPO and CPE instructions are useful for testing the parity of input data. However, the IN instruction does not set any of the condition flags. The flags can be set without altering the data by adding 00H to the contents of the accumulator.

The CPO instruction combines functions of the JPO and PUSH instructions. CPO tests the setting of the parity flag. If the flag is set to zero, CPO pushes the contents of the program counter onto the stack and then jumps to the address specified by the CPO instruction. If the flag is set to one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
CPO	address

Although the use of a label is more common, the address may also be specified as a number or an expression.

1	1	1	0	0	1	0	0
low addr							
high addr							

Cycles: 3 or 5 (2 or 5 on 8085)
States: 11 or 17 (9 or 18 on 8085)
Addressing: immediate/register indirect
Flags: none

Example:

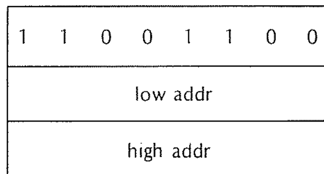
For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

CZ — CALL IF ZERO

The CZ instruction combines functions of the JZ and PUSH instructions. CZ tests the setting of the zero flag. If the flag is set to one (indicating that the contents of the accumulator are zero), CZ pushes the contents of the program counter onto the stack and then jumps to the address specified in the CZ instruction. If the flag is set to zero (indicating that the contents of the accumulator are other than zero), program execution simply continues with the next sequential instruction.

Opcode	*Operand*
CZ	address

Although the use of a label is most common, the address may also be specified as a number or an expression.



1	1	0	0	1	1	0	0
low addr							
high addr							

Cycles:	3 or 5 (2 or 5 on 8085)
States:	11 or 17 (9 or 18 on 8085)
Addressing:	immediate/register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the CALL instruction but not for each of its closely related variants.

DAA — DECIMAL ADJUST ACCUMULATOR

The DAA instruction adjusts the eight-bit value in the accumulator to form two four-bit binary coded decimal digits.

Opcode	*Operand*
DAA	

Operands are not permitted with the DAA instruction.

DAA is used when adding decimal numbers. It is the only instruction whose function requires use of the auxiliary carry flag. In multi-byte arithmetic operations, the DAA instruction typically is coded immediately after the arithmetic instruction so that the auxiliary carry flag is not altered unintentionally.

DAA operates as follows:

1. If the least significant four bits of the accumulator have a value greater than nine, or if the auxiliary carry flag is ON, DAA adds six to the accumulator.

2. If the most significant four bits of the accumulator have a value greater than nine, or if the carry flag is ON, DAA adds six to the most significant four bits of the accumulator.

0	0	1	0	0	1	1	1

Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

Example:

Assume that the accumulator contains the value 9BH as a result of adding 08 to 93:

```
CY    AC
0     0

 1001  0011
 0000  1000
 ----------
 1001  1011  = 9BH
```

Since 0BH is greater than nine, the instruction adds six to contents of the accumulator:

```
CY    AC
0     1
 1001  1011
 0000  0110
 ----------
 1010  0001  = A1H
```

Now that the most significant bits have a value greater than nine, the instruction adds six to them:

```
CY    AC
1     1
 1010  0001
 0110  0000
 ----------
 0000  0001
```

When the DAA has finished, the accumulator contains the value 01 in a BCD format; both the carry and auxiliary carry flags are set ON. Since the actual result of this addition is 101, the carry flag is probably significant to the program. The program is responsible for recovering and using this information. Notice that the carry flag setting is lost as soon as the program executes any subsequent instruction that alters the flag.

DAD — DOUBLE REGISTER ADD

DAD adds the 16-bit value in the specified register pair to the contents of the H and L register pair. The result is stored in H and L.

Opcode	*Operand*
DAD	{B, D, H, SP}

DAD may add only the contents of the B&C, D&E, H&L, or the SP (Stack Pointer) register pairs to the contents of H&L. Notice that the letter H must be used to specify that the H&L register pair is to be added to itself.

DAD sets the carry flag ON if there is a carry out of the H and L registers. DAD affects none of the condition flags other than carry.

0 0	R P	1 0 0 1

Cycles:	3
States:	10
Addressing:	register
Flags:	CY

Examples:

The DAD instruction provides a means for saving the current contents of the stack pointer.

```
LXI     H,00H    ;CLEAR H&L TO ZEROS
DAD     SP       ;GET SP INTO H&L
SHLD    SAVSP    ;STORE SP IN MEMORY
```

The instruction DAD H doubles the number in the H and L registers except when the operation causes a carry out of the H register.

DCR — DECREMENT

DCR subtracts one from the contents of the specified byte. DCR affects all the condition flags *except* the carry flag. Because DCR preserves the carry flag, it can be used within multi-byte arithmetic routines for decrementing character counts and similar purposes.

Decrement Register

Opcode	*Operand*
DCR	reg

The operand must specify one of the registers A through E, H or L. The instruction subtracts one from the contents of the specified register.

0 0	D D D	1 0 1

Cycles:	1
States:	5 (4 on 8085)
Addressing:	register
Flags:	Z,S,P,AC

Decrement Memory

Opcode	*Operand*
DCR	M

This instruction subtracts one from the contents of the memory location addressed by the H and L registers. M is a symbolic reference to the H and L registers.

0 0 1 1 0 1 0 1

Cycles:	3
States:	10
Addressing:	register indirect
Flags:	Z,S,P,AC

Example:

The DCR instruction is frequently used to control multi-byte operations such as moving a number of characters from one area of memory to another:

```
        MVI   B,5H     ;SET CONTROL COUNTER
        LXI   H,260H   ;LOAD H&L WITH SOURCE ADDR
        LXI   D,900H   ;LOAD D&E WITH DESTINATION ADDR
LOOP:   MOV   A,M      ;LOAD BYTE TO BE MOVED
        STAX  D        ;STORE BYTE
        DCX   D        ;DECREMENT DESTINATION ADDRESS
        DCX   M        ;DECREMENT SOURCE ADDRESS
        DCR   B        ;DECREMENT CONTROL COUNTER
        JNZ   LOOP     ;REPEAT LOOP UNTIL COUNTER=0
```

This example also illustrates an efficient programming technique. Notice that the control counter is decremented to zero rather than incremented until the desired count is reached. This technique avoids the need for a compare instruction and therefore conserves both memory and execution time.

DCX — DECREMENT REGISTER PAIR

DCX decrements the contents of the specified register pair by one. DCX affects none of the condition flags. Because DCX preserves all the flags, it can be used for address modification in any instruction sequence that relies on the passing of the flags.

Opcode	*Operand*
DCX	{B, D, H, SP}

DCX may decrement only the B&C, D&E, H&L, or the SP (Stack Pointer) register pairs. Notice that the letter H must be used to specify the H and L pair.

Exercise care when decrementing the stack pointer as this causes a loss of synchronization between the pointer and the actual contents of the stack.

0 0	R P	1 0 1 1

Cycles:	1
States:	5 (6 on 8085)
Addressing:	register
Flags:	none

Example:

Assume that the H and L registers contain the address 9800H when the instruction DCX H is executed. DCX considers the contents of the two registers to be a single 16-bit value and therefore performs a borrow from the H register to produce the value 97FFH.

DI — DISABLE INTERRUPTS

The interrupt system is disabled when the processor recognizes an interrupt or immediately following execution of a DI instruction.

In applications that use interrupts, the DI instruction is commonly used only when a code sequence must not be interrupted. For example, time-dependent code sequences become inaccurate when interrupted. You can disable the interrupt system by including a DI instruction at the beginning of the code sequence. Because you cannot predict the occurrence of an interrupt, include an EI instruction at the end of the time-dependent code sequence.

Opcode	*Operand*
DI	

Operands are not permitted with the DI instruction.

1	1	1	1	0	0	1	1

Cycles: 1
States: 4
Flags: none

NOTE

The 8085 TRAP interrupt cannot be disabled. This special interrupt is intended for serious problems that must be serviced regardless of the interrupt flag such as power failure or bus error. However, no interrupt including TRAP can interrupt the execution of the DI or EI instruction.

EI — ENABLE INTERRUPTS

The EI instruction enables the interrupt system following execution of the next program instruction. Enabling the interrupt system is delayed one instruction to allow interrupt subroutines to return to the main program before a subsequent interrupt is acknowledged.

In applications that use interrupts, the interrupt system is usually disabled only when the processor accepts an interrupt or when a code sequence must not be interrupted. You can disable the interrupt system by including a DI instruction at the beginning of the code sequence. Because you cannot predict the occurrence of an interrupt, include an EI instruction at the end of the code sequence.

Opcode	*Operand*
EI	

Operands are not permitted with the EI instruction.

1	1	1	1	1	0	1	1

Cycles: 1
States: 4
Flags: none

NOTE

The 8085 TRAP interrupt cannot be disabled. This special interrupt is intended for serious problems that must be serviced regardless of the interrupt flag such as power failure or bus failure. However, no interrupt including TRAP can interrupt the execution of the DI or EI instruction.

Example:

The EI instruction is frequently used as part of a start-up sequence. When power is first applied, the processor begins operating at some indeterminate address. Application of a RESET signal forces the program counter to

zero. A common instruction sequence at this point is EI, HLT. These instructions enable the interrupt system (RESET also disables the interrupt system) and halt the processor. A subsequent manual or automatic interrupt then determines the effective start-up address.

HLT HALT

The HLT instruction halts the processor. The program counter contains the address of the next sequential instruction. Otherwise, the flags and registers remain unchanged.

0	1	1	1	0	1	1	0

Cycles:	1
States:	7 (5 on 8085)
Flags:	none

Once in the halt state, the processor can be restarted only by an external event, typically an interrupt. Therefore, you should be certain that interrupts are enabled before the HLT instruction is executed. See the description of the EI (Enable Interrupt) instruction.

If an 8080 HLT instruction is executed while interrupts are disabled, the only way to restart the processor is by application of a RESET signal. This forces the program counter to zero. The same is true of the 8085, except for the TRAP interrupt, which is recognized even when the interrupt system is disabled.

The processor can temporarily leave the halt state to service a direct memory access request. However, the processor reenters the halt state once the request has been serviced.

A basic purpose for the HLT instruction is to allow the processor to pause while waiting for an interrupt from a peripheral device. However, a halt wastes processor resources and should be used only when there is no useful processing task available.

IN INPUT FROM PORT

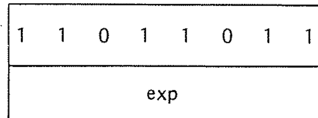
The IN instruction reads eight bits of data from the specified port and loads it into the accumulator.

NOTE

This description is restricted to the exact function of the IN instruction. Input/output structures are described in the *8080 or 8085 Microcomputer Systems User's Manual.*

Opcode	*Operand*
IN	exp

The operand expression may be a number or any expression that yields a value in the range 00H through 0FFH.



Cycles: 3
States: 10
Addressing: direct
Flags: none

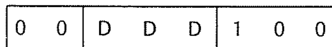
INR — INCREMENT

INR adds one to the contents of the specified byte. INR affects all of the condition flags *except* the carry flag. Because INR preserves the carry flag, it can be used within multi-byte arithmetic routines for incrementing character counts and similar purposes.

Increment Register

Opcode	*Operand*
INR	reg

The operand must specify one of the registers A through E, H or L. The instruction adds one to the contents of the specified register.

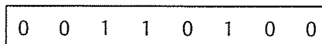


Cycles: 1
States: 5 (4 on 8085)
Addressing: register
Flags: Z,S,P,AC

Increment Memory

Opcode	*Operand*
INR	M

This instruction increments by one the contents of the memory location addressed by the H and L registers. M is a symbolic reference to the H and L registers.



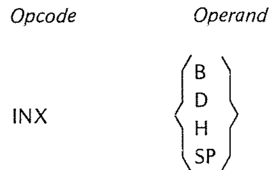
Cycles: 3
States: 10
Addressing: register indirect
Flags: Z,S,P,AC

Example:

If register C contains 99H, the instruction INR C increments the contents of the register to 9AH.

INX — INCREMENT REGISTER PAIR

INX adds one to the contents of the specified register pair. INX affects none of the condition flags. Because INX preserves all the condition flags, it can be used for address modification within multi-byte arithmetic routines.



INX may increment only the B&C, D&E, H&L, or the SP (Stack Pointer) register pairs. Notice that the letter H must be used to specify the H and L register pair.

Exercise care when incrementing the stack pointer. Assume, for example, that INX SP is executed after a number of items have been pushed onto the stack. A subsequent POP instruction accesses the high-order byte of the most recent stack entry and the low-order byte of the next older entry. Similarly, a PUSH instruction adds the two new bytes to the stack, but overlays the low-order byte of the most recent entry.

0 0	R P	0 0 1 1

Cycles:	1
States:	5 (6 on 8085)
Addressing:	register
Flags:	none

Example:

Assume that the D and E registers contain the value 01FFH. The instruction INX D increments the value to 0200H. By contrast, the INR E instruction ignores the carry out of the low-order byte and produces a result of 0100H. (This condition can be detected by testing the Zero condition flag.)

If the stack pointer register contains the value 0FFFFH, the instruction INX SP increments the contents of SP to 0000H. The INX instruction sets no flags to indicate this condition.

JC — JUMP IF CARRY

The JC instruction tests the setting of the carry flag. If the flag is set to one, program execution resumes at the address specified in the JC instruction. If the flag is reset to zero, execution continues with the next sequential instruction.

Opcode	*Operand*
JC	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the instruction.

1	1	0	1	1	0	1	0
low addr							
high addr							

Cycles: 3 (2 or 3 on 8085)
States: 10 (7 or 10 on 8085)
Addressing: immediate
Flags: none

Example:

Examples of the variations of the jump instruction appear in the description of the JPO instruction.

JM — JUMP IF MINUS

The JM instruction tests the setting of the sign flag. If the contents of the accumulator are negative (sign flag = 1), program execution resumes at the address specified in the JM instruction. If the contents of the accumulator are positive (sign flag = 0), execution continues with the next sequential instruction.

Opcode	*Operand*
JM	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the instructions.

1	1	1	1	1	0	1	0
low addr							
high addr							

Cycles: 3 (2 or 3 on 8085)
States: 10 (7 or 10 on 8085)
Addressing: immediate
Flags: none

Example:

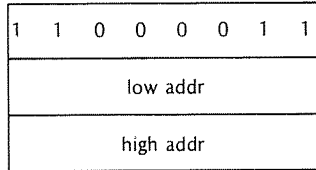
Examples of the variations of the jump instruction appear in the description of the JPO instruction.

JMP — JUMP

The JMP instruction alters the execution sequence by loading the address in its second and third bytes into the program counter.

Opcode	*Operand*
JMP	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the address.



Cycles:	3
States:	10
Addressing:	immediate
Flags:	none

Example:

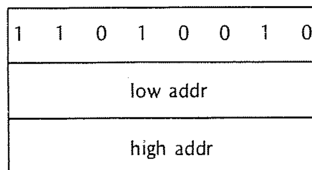
Examples of the variations of the jump instruction appear in the description of the JPO instruction.

JNC — JUMP IF NO CARRY

The JNC instruction tests the setting of the carry flag. If there is no carry (carry flag = 0), program execution resumes at the address specified in the JNC instruction. If there is a carry (carry flag = 1), execution continues with the next sequential instruction.

Opcode	*Operand*
JNC	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the instruction.



Cycles:	3 (2 or 3 on 8085)
States:	10 (7 or 10 on 8085)
Addressing:	immediate
Flags:	none

Example:

Examples of the variations of the jump instruction appear in the description of the JPO instruction.

JNZ JUMP IF NOT ZERO

The JNZ instruction tests the setting of the zero flag. If the contents of the accumulator are not zero (zero flag = 0), program execution resumes at the address specified in the JNZ instruction. If the contents of the accumulator are zero (zero flag = 1), execution continues with the next sequential instruction.

Opcode	*Operand*
JNZ	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the instruction.

1 1 0 0 0 0 1 0
low addr
high addr

Cycles:	3 (2 or 3 on 8085)
States:	10 (7 or 10 on 8085)
Addressing:	immediate
Flags:	none

Example:

Examples of the variations of the jump instruction appear in the description of the JPO instruction.

JP JUMP IF POSITIVE

The JP instruction tests the setting of the sign flag. If the contents of the accumulator are positive (sign flag = 0), program execution resumes at the address specified in the JP instruction. If the contents of the accumulator are minus (sign flag = 1), execution continues with the next sequential instruction.

Opcode	*Operand*
JP	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low order address bytes when it assembles the instruction.

1 1 1 1 0 0 1 0
low addr
high addr

Cycles:	3 (2 or 3 on 8085)
States:	10 (7 or 10 on 8085)
Addressing:	immediate
Flags:	none

Example:

Examples of the variations of the jump instruction appear in the description of the JPO instruction.

JPE — JUMP IF PARITY EVEN

Parity is even if the byte in the accumulator has an even number of one bits. The parity flag is set to one to indicate this condition.

The JPE instruction tests the setting of the parity flag. If the parity flag is set to one, program execution resumes at the address specified in the JPE instruction. If the flag is reset to zero, execution continues with the next sequential instruction.

Opcode	*Operand*
JPE	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the instruction.

The JPE and JPO (jump if parity odd) instructions are especially useful for testing the parity of input data. However, the IN instruction does not set any of the condition flags. The flags can be set by adding 00H to the contents of the accumulator.

1 1 1 0 1 0 1 0
low addr
high addr

Cycles:	3 (2 or 3 on 8085)
States:	10 (7 or 10 on 8085)
Addressing:	immediate
Flags:	none

Example:

Examples of the variations of the jump instruction appear in the description of the JPO instruction.

JPO — JUMP IF PARITY ODD

Parity is odd if the byte in the accumulator has an odd number of one bits. The parity flag is set to zero to indicate this condition.

The JPO instruction tests the setting of the parity flag. If the parity flag is reset to zero, program execution resumes at the address specified in the JPO instruction. If the flag is set to one, execution continues with the next sequential instruction.

Opcode	*Operand*
JPO	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the instruction.

The JPO and JPE (jump if parity even) instructions are especially useful for testing the parity of input data. However, the IN instruction does not set any of the condition flags. The flags can be set by adding 00H to the contents of the accumulator.

1	1	1	0	0	0	1	0
low addr							
high addr							

Cycles:	3 (2 or 3 on 8085)
States:	10 (7 or 10 on 8085)
Addressing:	immediate
Flags:	none

Example:

This example shows three different but equivalent methods for jumping to one of two points in a program based upon whether or not the Sign bit of a number is set. Assume that the byte to be tested is the C register.

Label	*Code*	*Operand*
ONE:	MOV	A,C
	ANI	80H
	JZ	PLUS
	JNZ	MINUS
TWO:	MOV	A,C
	RLC	
	JNC	PLUS
	JMP	MINUS
THREE:	MOV	A,C
	ADI	0
	JM	MINUS
PLUS:	—	;SIGN BIT RESET
MINUS:	—	;SIGN BIT SET

The AND immediate instruction in block ONE zeroes all bits of the data byte except the Sign bit, which remains unchanged. If the Sign bit was zero, the Zero condition bit will be set, and the JZ instruction will cause program control to be transferred to the instruction at PLUS. Otherwise, the JZ instruction will merely update the program counter by three, and the JNZ instruction will be executed, causing control to be transferred to the instruction at MINUS. (The Zero bit is unaffected by all jump instructions.)

The RLC instruction in block TWO causes the Carry bit to be set equal to the Sign bit of the data byte. If the Sign bit was reset, the JNC instruction causes a jump to PLUS. Otherwise the JMP instruction is executed, unconditionally transferring control to MINUS. (Note that, in this instance, a JC instruction could be substituted for the unconditional jump with identical results.)

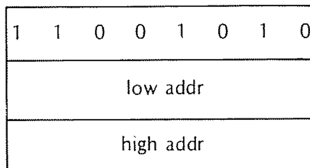
The add immediate instruction in block THREE causes the condition bits to be set. If the sign bit was set, the JM instruction causes program control to be transferred to MINUS. Otherwise, program control flows automatically into the PLUS routine.

JZ — JUMP IF ZERO

The JZ instruction tests the setting of the zero flag. If the flag is set to one, program execution resumes at the address specified in the JZ instruction. If the flag is reset to zero, execution continues with the next sequential instruction.

Opcode	*Operand*
JZ	address

The address may be specified as a number, a label, or an expression. The assembler inverts the high and low address bytes when it assembles the instruction.



Cycles:	3 (2 or 3 on 8085)
States:	10 (7 or 10 on 8085)
Addressing:	immediate
Flags:	none

Example:

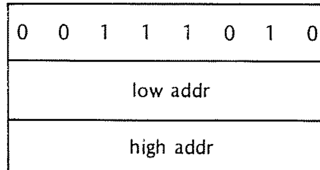
Examples of the variations of the jump instruction appear in the description of the JPO instruction.

LDA — LOAD ACCUMULATOR DIRECT

LDA loads the accumulator with a copy of the byte at the location specified in bytes two and three of the LDA instruction.

Opcode	*Operand*
LDA	address

The address may be stated as a number, a previously defined label, or an expression. The assembler inverts the high and low address bytes when it builds the instruction.



Cycles:	4
States:	13
Addressing:	direct
Flags:	none

Examples:

The following instructions are equivalent. When executed, each replaces the accumulator contents with the byte of data stored at memory location 300H.

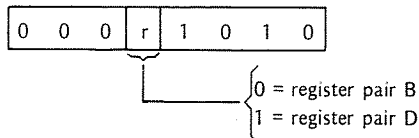
```
LOAD:   LDA   300H
        LDA   3*(16*16)
        LDA   200H+256
```

LDAX — LOAD ACCUMULATOR INDIRECT

LDAX loads the accumulator with a copy of the byte stored at the memory location addressed by register pair B or register pair D.

Opcode	*Operand*
LDAX	{B, D}

The operand B specifies the B and C register pair; D specifies the D and E register pair. This instruction may specify only the B or D register pair.



Cycles:	2
States:	7
Addressing:	register indirect
Flags:	none

Example:

Assume that register D contains 93H and register E contains 8BH. The following instruction loads the accumulator with the contents of memory location 938BH:

```
LDAX  D
```

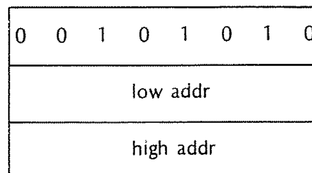
LHLD — LOAD H AND L DIRECT

LHLD loads the L register with a copy of the byte stored at the memory location specified in bytes two and three of the LHLD instruction. LHLD then loads the H register with a copy of the byte stored at the next higher memory location.

Opcode	*Operand*
LHLD	address

The address may be stated as a number, a label, or an expression.

Certain instructions use the symbolic reference M to access the memory location currently specified by the H and L registers. LHLD is one of the instructions provided for loading new addresses into the H and L registers. The user may also load the current top of the stack into the H and L registers (POP instruction). Both LHLD and POP replace the contents of the H and L registers. You can also exchange the contents of H and L with the D and E registers (XCHG instruction) or the top of the stack (XTHL instruction) if you need to save the current H and L registers for subsequent use. SHLD stores H and L in memory.



0	0	1	0	1	0	1	0
low addr							
high addr							

Cycles:	5
States:	16
Addressing:	direct
Flags:	none

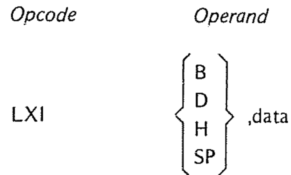
Example:

Assume that locations 3000 and 3001H contain the address 064EH stored in the format 4E06. In the following sequence, the MOV instruction moves a copy of the byte stored at address 064E into the accumulator:

```
LHLD   3000H   ;SET UP ADDRESS
MOV    A,M     ;LOAD ACCUM FROM ADDRESS
```

LXI — LOAD REGISTER PAIR IMMEDIATE

LXI is a three-byte instruction; its second and third bytes contain the source data to be loaded into a register pair. LXI loads a register pair by copying its second and third bytes into the specified destination register pair.



The first operand must specify the register pair to be loaded. LXI can load the B and C register pair, the D and E register pair, the H and L register pair, or the Stack Pointer.

The second operand specifies the two bytes of data to be loaded. This data may be coded in the form of a number, an ASCII constant, the label of some previously defined value, or an expression. The data must not exceed two bytes.

LXI is the only immediate instruction that accepts a 16-bit value. All other immediate instructions require 8-bit values.

Notice that the assembler inverts the two bytes of data to create the format of an address stored in memory. LXI loads its third byte into the first register of the pair and its second byte into the second register of the pair. This has the effect of reinverting the data into the format required for an address stored in registers. Thus, the instruction LXI B,'AZ' loads A into register B and Z into register C.

0 0	R P	0 0 0 1
low-order data		
high-order data		

Cycles:	3
States:	10
Addressing:	immediate
Flags:	none

Examples:

A common use for LXI is to establish a memory address for use in subsequent instructions. In the following sequence, the LXI instruction loads the address of STRNG into the H and L registers. The MOV instruction then loads the data stored at that address into the accumulator.

```
LXI     H,STRNG     ;SET ADDRESS
MOV     A,M         ;LOAD STRNG INTO ACCUMULATOR
```

The following LXI instruction is used to initialize the stack pointer in a relocatable module. The LOCATE program provides an address for the special reserved label STACK.

```
LXI     SP,STACK
```

MOV — MOVE

The MOV instruction moves one byte of data by copying the source field into the destination field. Source data remains unchanged. The instruction's operands specify whether the move is from register to register, from a register to memory, or from memory to a register.

Move Register to Register

Opcode	*Operand*
MOV	reg1,reg2

The instruction copies the contents of reg2 into reg1. Each operand must specify one of the registers A, B, C, D, E, H, or L.

When the same register is specified for both operands (as in MOV A,A), the MOV functions as a NOP (no operation) since it has no other noticeable effect. This form of MOV requires one more machine state than NOP, and therefore has a slightly longer execution time than NOP. Since M addresses a register pair rather than a byte of data, MOV M,M is not allowed.

0 1	D D D	S S S

Cycles: 1
States: 5 (4 on 8085)
Addressing: register
Flags: none

Move to Memory

Opcode	*Operand*
MOV	M,r

This instruction copies the contents of the specified register into the memory location addressed by the H and L registers. M is a symbolic reference to the H and L register pair. The second operand must address one of the registers.

0 1 1 1 0	S S S

Cycles: 2
States: 7
Addressing: register indirect
Flags: none

Move from Memory

Opcode	*Operand*
MOV	r,M

This instruction copies the contents of the memory location addressed by the H and L registers into the specified register. The first operand must name the destination register. The second operand must be M. M is a symbolic reference to the H and L registers.

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	none

Examples:

Label	*Opcode*	*Operands*	*Comment*
LDACC:	MOV	A,M	;LOAD ACCUM FROM MEMORY
	MOV	E,A	;COPY ACCUM INTO E REG
NULOP:	MOV	C,C	;NULL OPERATION

MVI — MOVE IMMEDIATE

MVI is a two-byte instruction; its second byte contains the source data to be moved. MVI moves one byte of data by copying its second byte into the destination field. The instruction's operands specify whether the move is to a register or to memory.

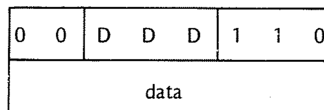
Move Immediate to Register

Opcode	*Operand*
MVI	reg,data

The first operand must name one of the registers A through E, H or L as a destination for the move.

The second operand specifies the actual data to be moved. This data may be in the form of a number, an ASCII constant, the label of some previously defined value, or an expression. The data must not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

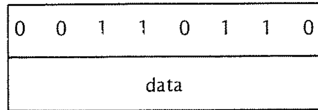


Cycles:	2
States:	7
Addressing:	immediate
Flags:	none

Move Immediate to Memory

Opcode	*Operand*
MVI	M,data

This instruction copies the data stored in its second byte into the memory location addressed by H and L. M is a symbolic reference to the H and L register pair.



Cycles:	3
States:	10
Addressing:	immediate/register indirect
Flags:	none

Examples:

The following examples show a number of methods for defining immediate data in the MVI instruction. All of the examples generate the bit pattern for the ASCII character A.

```
MVI     M,01000001B
MVI     M,'A'
MVI     M,41H
MVI     M,101Q
MVI     M,65
MVI     M,5+30*2
```

NOP — NO OPERATION

NOP performs no operation and affects none of the condition flags. NOP is useful as filler in a timing loop.

Opcode	*Operand*
NOP	

Operands are not permitted with the NOP instruction.

ORA — INCLUSIVE OR WITH ACCUMULATOR

ORA performs an inclusive OR logical operation using the contents of the specified byte and the accumulator. The result is placed in the accumulator.

Summary of Logical Operations

AND produces a one bit in the result only when the corresponding bits in the test data and the mask data are one.

OR produces a one bit in the result when the corresponding bits in either the test data or the mask data are ones.

Exclusive OR produces a one bit only when the corresponding bits in the test data and the mask data are different; i.e., a one bit in either the test data or the mask data – but not both – produces a one bit in the result.

AND	OR	EXCLUSIVE OR
1010 1010	1010 1010	1010 1010
0000 1111	0000 1111	0000 1111
0000 1010	1010 1111	1010 0101

OR Register with Accumulator

Opcode	*Operand*
ORA	reg

The operand must specify one of the registers A through E, H or L. This instruction ORs the contents of the specified register and the accumulator and stores the result in the accumulator. The carry and auxiliary carry flags are reset to zero.

1	0	1	1	0	S	S	S

Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

OR Memory with Accumulator

Opcode	*Operand*
ORA	M

The contents of the memory location specified by the H and L registers are inclusive-ORed with the contents of the accumulator. The result is stored in the accumulator. The carry and auxiliary carry flags are reset to zero.

1	0	1	1	0	1	1	0

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Example:

Since any bit inclusive-ORed with a one produces a one and any bit ORed with a zero remains unchanged, ORA is frequently used to set ON particular bits or groups of bits. The following example ensures that bit 3 of the accumulator is set ON, but the remaining bits are not disturbed. This is frequently done when individual bits are used as status flags in a program. Assume that register D contains the value 08H:

Accumulator	=	0 1 0 0 0 0 1 1
Register D	=	0 0 0 0 1 0 0 0
		0 1 0 0 1 0 1 1

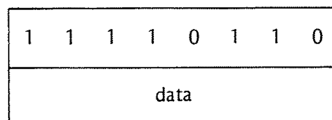
ORI — INCLUSIVE OR IMMEDIATE

ORI performs an inclusive OR logical operation using the contents of the second byte of the instruction and the contents of the accumulator. The result is placed in the accumulator. ORI also resets the carry and auxiliary carry flags to zero.

Opcode	*Operand*
ORI	data

The operand must specify the data to be used in the inclusive OR operation. This data may be in the form of a number, an ASCII constant, the label of some previously defined value, or an expression. The data may not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assume the LOW operator and issues an error message.



Cycles:	2
States:	7
Addressing:	immediate
Flags:	Z,S,P,SY,AC

Summary of Logical Operations

AND produces a one bit in the result only when the corresponding bits in both the test data and the mask data are ones.

OR produces a one bit in the result when the corresponding bits in either the test data or the mask data are ones.

Exclusive OR produces a one bit only when the corresponding bits in the test data and the mask data are different; i.e., a one bit in either the test data or the mask data — but not both — produces a one bit in the result.

AND	OR	EXCLUSIVE OR
1010 1010	1010 1010	1010 1010
0000 1111	0000 1111	0000 1111
0000 1010	1010 1111	1010 0101

Example:

See the description of the ORA instruction for an example of the use of the inclusive OR. The following examples show a number of methods for defining immediate data in the ORI instruction. All of the examples generate the bit pattern for the ASCII character A.

```
ORI     01000001B
ORI     'A'
ORI     41H
ORI     101Q
ORI     65
ORI     5+30*2
```

OUT — OUTPUT TO PORT

The OUT instruction places the contents of the accumulator on the eight-bit data bus and the number of the selected port on the sixteen-bit address bus. Since the number of ports ranges from 0 through 255, the port number is duplicated on the address bus.

It is the responsibility of external logic to decode the port number and to accept the output data.

NOTE

Because a discussion of input/output structures is beyond the scope of this manual, this description is restricted to the exact function of the OUT instruction. Input/output structures are described in the *8080 or 8085 Microcomputer Systems User's Manual.*

Opcode	*Operand*
OUT	exp

The operand must specify the number of the desired output port. This may be in the form of a number or an expression in the range 00H through 0FFH.

1	1	0	1	0	0	1	1
exp							

Cycles:	3
States:	10
Addressing:	direct
Flags:	none

PCHL — MOVE H&L TO PROGRAM COUNTER

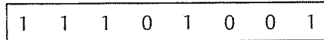
PCHL loads the contents of the H and L registers into the program counter register. Because the processor fetches the next instruction from the updated program counter address, PCHL has the effect of a jump instruction.

Opcode	*Operand*
PCHL	

Operands are not permitted with the PCHL instruction.

PCHL moves the contents of the H register to the high-order eight bits of the program counter and the contents of the L register to the low-order eight bits of the program counter.

The user program must ensure that the H and L registers contain the address of an executable instruction when the PCHL instruction is executed.



Cycles:	1
States:	5 (6 on 8085)
Addressing:	register
Flags:	none

Example:

One technique for passing data to a subroutine is to place the data immediately after the subroutine call. The return address pushed onto the stack by the CALL instruction actually addresses the data rather than the next instruction after the CALL. For this example, assume that two bytes of data follow the subroutine call. The following coding sequence performs a return to the next instruction after the call:

```
GOBACK:  POP   H   ;GET DATA ADDRESS
         INR   L   ;ADD 2 TO FORM
         INR   L   ;RETURN ADDRESS
         PCHL      ;RETURN
```

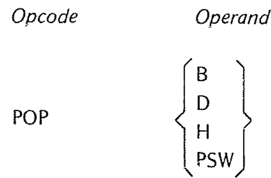
POP — POP

The POP instruction removes two bytes of data from the stack and copies them to a register pair or copies the Program Status Word into the accumulator and the condition flags.

POP Register Pair

POP copies the contents of the memory location addressed by the stack pointer into the low-order register of the register pair. POP then increments the stack pointer by one and copies the contents of the resulting address into

the high-order register of the pair. POP then increments the stack pointer again so that it addresses the next older item on the stack.

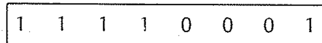


The operand may specify the B&C, D&E, or the H&L register pairs. POP PSW is explained separately.

Cycles:	3
States:	10
Addressing:	register indirect
Flags:	none

POP PSW

POP PSW uses the contents of the memory location specified by the stack pointer to restore the condition flags. POP PSW increments the stack pointer by one and restores the contents of that address to the accumulator. POP then increments the stack pointer again so that it addresses the next older item on the stack.



Cycles:	3
States:	10
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Example:

Assume that a subroutine is called because of an external interrupt. In general, such subroutines should save and restore any registers it uses so that main program can continue normally when it regains control. The following sequence of PUSH and POP instructions save and restore the Program Status Word and all the registers:

```
PUSH    PSW
PUSH    B
PUSH    D
PUSH    H
  .
  .
subroutine coding
  .
  .
POP     H
POP     D
POP     B
POP     PSW
RET
```

Notice that the sequence of the POP instructions is the opposite of the PUSH instruction sequence.

PUSH

The PUSH instruction copies two bytes of data to the stack. This data may be the contents of a register pair or the Program Status Word, as explained below:

PUSH Register Pair

PUSH decrements the stack pointer register by one and copies the contents of the high-order register of the register pair to the resulting address. PUSH then decrements the pointer again and copies the low-order register to the resulting address. The source registers remain unchanged.

Opcode	*Operand*
PUSH	{B, D, H, PSW}

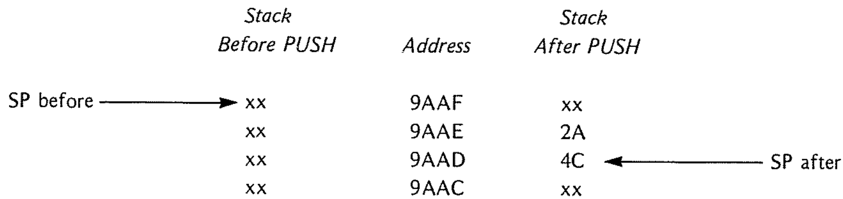
The operand may specify the B&C, D&E, or H&L register pairs. PUSH PSW is explained separately.

1 1	R P	0 1 0 1

Cycles:	3
States:	11 (13 on 8085)
Addressing:	register indirect
Flags:	none

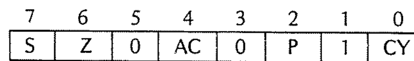
Example:

Assume that register B contains 2AH, the C register contains 4CH, and the stack pointer is set at 9AAF. The instruction PUSH B stores the B register at memory address 9AAEH and the C register at 9AADH. The stack pointer is set to 9AADH:



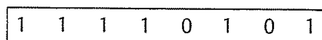
PUSH PSW

PUSH PSW copies the Program Status Word onto the stack. The Program Status Word comprises the contents of the accumulator and the current settings of the condition flags. Because there are only five condition flags, PUSH PSW formats the flags into an eight-bit byte as follows:



On the 8080, bits 3 and 5 are always zero; bit one is always set to one. These filler bits are undefined on the 8085.

PUSH PSW decrements the stack pointer by one and copies the contents of the accumulator to the resulting address. PUSH PSW again decrements the pointer and copies the formatted condition flag byte to the resulting address. The contents of the accumulator and the condition flags remain unchanged.



Cycles:	3
States:	11 (12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

When a program calls subroutines, it is frequently necessary to preserve the current program status so the calling program can continue normally when it regains control. Typically, the subroutine performs a PUSH PSW prior to execution of any instruction that might alter the contents of the accumulator or the condition flag settings. The subroutine then restores the pre-call system status by executing a POP PSW instruction just before returning control to the calling program.

RAL — ROTATE LEFT THROUGH CARRY

RAL rotates the contents of the accumulator and the carry flag one bit position to the left. The carry flag, which is treated as though it were part of the accumulator, transfers to the low-order bit of the accumulator. The high-order bit of the accumulator transfers into the carry flag.

Opcode	*Operand*
RAL	

Operands are not permitted with the RAL instruction.

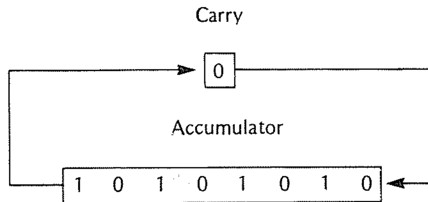
0	0	0	1	0	1	1	1

Cycles: 1
States: 4
Flags: CY only

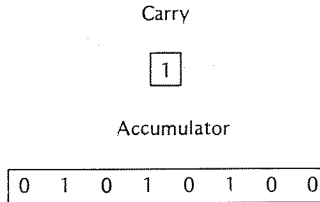
Example:

Assume that the accumulator contains the value 0AAH and the carry flag is zero. The following diagrams illustrate the effect of the RAL instruction:

Before:



After:



RAR — ROTATE RIGHT THROUGH CARRY

RAR rotates the contents of the accumulator and the carry flag one bit position to the right. The carry flag, which is treated as though it were part of the accumulator, transfers to the high-order bit of the accumulator. The low-order bit of the accumulator transfers into the carry flag.

Opcode	*Operand*
RAR	

Operands are not permitted with the RAR instruction.

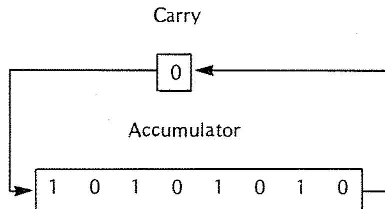
0	0	0	1	1	1	1	1

Cycles: 1
States: 4
Flags: CY only

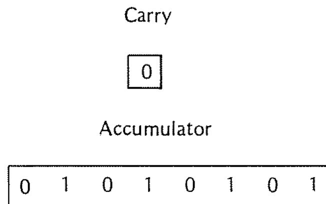
Example:

Assume that the accumulator contains the value 0AAH and the carry flag is zero. The following diagrams illustrate the effect of the RAR instruction:

Before:



After:

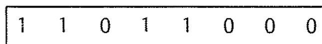


RC — RETURN IF CARRY

The RC instruction tests the carry flag. If the flag is set to one to indicate a carry, the instruction pops two bytes off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is zero, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RC	

Operands are not permitted with the RC instruction.



Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

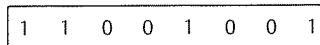
RET — RETURN FROM SUBROUTINE

The RET instruction pops two bytes of data off the stack and places them in the program counter register. Program execution resumes at the new address in the program counter.

Typically, RET instructions are used in conjunction with CALL instructions. (The same is true of the variants of these instructions.) In this case, it is assumed that the data the RET instruction pops off the stack is a return address placed there by a previous CALL. This has the effect of returning control to the next instruction after the CALL. The user must be certain that the RET instruction finds the address of executable code on the stack. If the instruction finds the address of data, the processor attempts to execute the data as though it were code.

Opcode	*Operand*
RET	

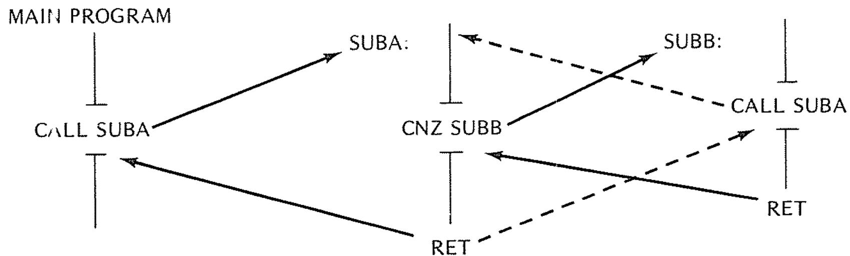
Operands are not permitted with the RET instruction.



Cycles:	3
States:	10
Addressing:	register indirect
Flags:	none

Example:

As mentioned previously, subroutines can be nested. That is, a subroutine can call a subroutine that calls another subroutine. The only practical limit on the number of nested calls is the amount of memory available for stacking return addresses. A nested subroutine can even call the subroutine that called it, as shown in the following example. (Notice that the program must contain logic that eventually returns control to the main program. Otherwise, the two subroutines will call each other indefinitely.)



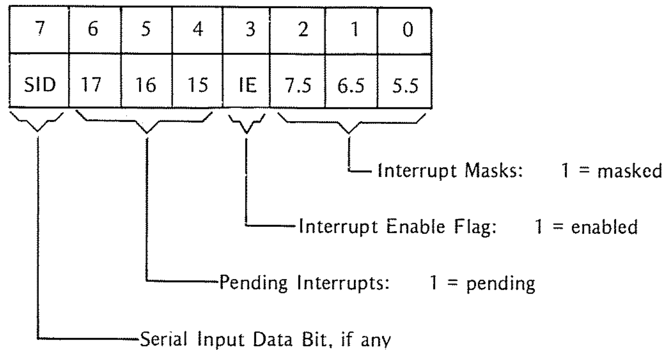
RIM (8085 PROCESSOR ONLY) — READ INTERRUPT MASK

The RIM instruction loads eight bits of data into the accumulator. The resulting bit pattern indicates the current setting of the interrupt mask, the setting of the interrupt flag, pending interrupts, and one bit of serial input data, if any.

Opcode	*Operand*
RIM	

Operands are not permitted with the RIM instruction.

The RIM instruction loads the accumulator with the following information:



The mask and pending flags refer only to the RST5.5, RST6.5, and RST7.5 hardware interrupts. The IE flag refers to the entire interrupt system. Thus, the IE flag is identical in function and level to the INTE pin on the 8080. A 1 bit in this flag indicates that the entire interrupt system is enabled.

0	0	1	0	0	0	0	0

Cycles: 1
States: 4
Flags: none

RLC — ROTATE ACCUMULATOR LEFT

RLC sets the carry flag equal to the high-order bit of the accumulator, thus overwriting its previous setting. RLC then rotates the contents of the accumulator one bit position to the left with the high-order bit transferring to the low-order position of the accumulator.

Opcode	*Operand*
RLC	

Operands are not allowed with the RLC instruction.

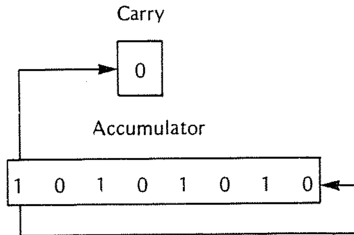
0	0	0	0	0	1	1	1

Cycles: 1
States: 4
Flags: CY only

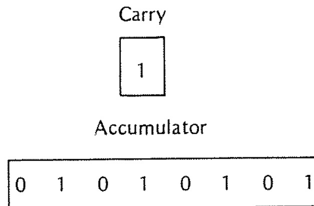
Example:

Assume that the accumulator contains the value 0AAH and the carry flag is zero. The following diagrams illustrate the effect of the RLC instruction.

Before:



After:

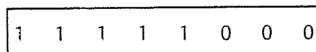


RM — RETURN IF MINUS

The RM instruction tests the sign flag. If the flag is set to one to indicate negative data in the accumulator, the instruction pops two bytes off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is set to zero, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RM	

Operands are not permitted with the RM instruction.



Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

RNC RETURN IF NO CARRY

The RNC instruction tests the carry flag. If the flag is set to zero to indicate that there has been no carry, the instruction pops two bytes off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RNC	

Operands are not permitted with the RNC instruction.

1	1	0	1	0	0	0	0

Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

RNZ RETURN IF NOT ZERO

The RNZ instruction tests the zero flag. If the flag is set to zero to indicate that the contents of the accumulator are other than zero, the instruction pops two bytes off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is set to one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RNZ	

Operands are not permitted with the RNZ instruction.

1	1	0	0	0	0	0	0

Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

RP — RETURN IF POSITIVE

The RP instruction tests the sign flag. If the flag is reset to zero to indicate positive data in the accumulator, the instruction pops two bytes off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is set to one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RP	

Operands are not permitted with the RP instruction.

1	1	1	1	0	0	0	0

Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

RPE — RETURN IF PARITY EVEN

Parity is even if the byte in the accumulator has an even number of one bits. The parity flag is set to one to indicate this condition. The RPE and RPO instructions are useful for testing the parity of input data. However, the IN instruction does not set any of the condition flags. The flags can be set without altering the data by adding 00H to the contents of the accumulator.

The RPE instruction tests the parity flag. If the flag is set to one to indicate even parity, the instruction pops two bytes off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is zero, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RPE	

Operands are not permitted with the RPE instruction.

1	1	1	0	1	0	0	0

Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

RPO — RETURN IF PARITY ODD

Parity is odd if the byte in the accumulator has an odd number of one bits. The parity flag is reset to zero to indicate this condition. The RPO and RPE instructions are useful for testing the parity of input data. However, the IN instruction does not set any of the condition flags. The flags can be set without altering the data by adding 00H to the contents of the accumulator.

The RPO instruction tests the parity flag. If the flag is reset to zero to indicate odd parity, the instruction pops two bytes off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is set to one, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RPO	

Operands are not permitted with the RPO instruction.

1	1	1	0	0	0	0	0

Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

RRC — ROTATE ACCUMULATOR RIGHT

RRC sets the carry flag equal to the low-order bit of the accumulator, thus overwriting its previous setting. RRC then rotates the contents of the accumulator one bit position to the right with the low-order bit transferring to the high order position of the accumulator.

Opcode	*Operand*
RRC	

Operands are not permitted with the RRC instruction.

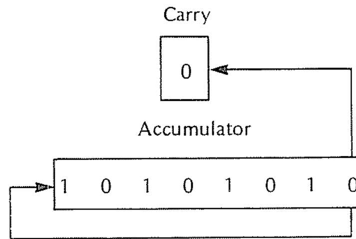
0	0	0	0	1	1	1	1

Cycles:	1
States:	4
Flags:	CY only

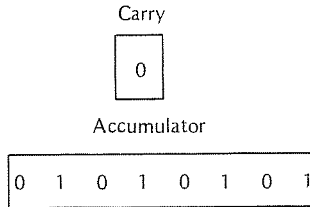
Example:

Assume that the accumulator contains the value 0AAH and the carry flag is zero. The following diagrams illustrate the effect of the RRC instruction:

Before:



After:



RST RESTART

RST is a special purpose CALL instruction designed primarily for use with interrupts. RST pushes the contents of the program counter onto the stack to provide a return address and then jumps to one of eight predetermined addresses. A three-bit code carried in the opcode of the RST instruction specifies the jump address.

The restart instruction is unique because it seldom appears as source code in an applications program. More often, the peripheral devices seeking interrupt service pass this one-byte instruction to the processor.

When a device requests interrupt service and interrupts are enabled, the processor acknowledges the request and prepares its data lines to accept any one-byte instruction from the device. RST is generally the instruction of choice because its special purpose CALL establishes a return to the main program.

The processor moves the three-bit address code from the RST instruction into bits 3, 4, and 5 of the program counter. In effect, this multiplies the code by eight. Program execution resumes at the new address where eight bytes are available for code to service the interrupt. If eight bytes are too few, the program can either jump to or call a subroutine.

8085 NOTE

The 8085 processor includes four hardware inputs that generate internal RST instructions. Rather than send a RST instruction, the interrupting device need only apply a signal to the RST5.5, RST6.5, RST7.5, or TRAP input pin. The processor then generates an internal RST instruction. The execution depends on the input:

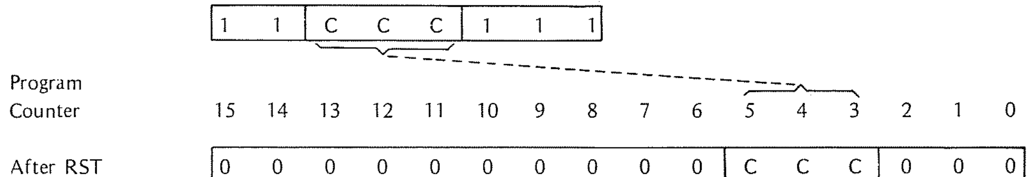
INPUT NAME	*RESTART ADDRESS*
TRAP	24H
RST5.5	2CH
RST6.5	34H
RST7.5	3CH

Notice that these addresses are within the same portion of memory used by the RST instruction, and therefore allow only four bytes — enough for a call or jump and a return — for the interrupt service routine.

If included in the program code, the RST instruction has the following format:

Opcode	*Operand*
RST	code

The address code must be a number or expression within the range 000B through 111B.



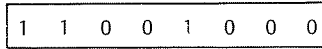
Cycles: 3
States: 11 (12 on 8085)
Addressing: register indirect
Flags: none

RZ RETURN IF ZERO

The RZ instruction tests the zero flag. If the flag is set to one to indicate that the contents of the accumulator are zero, the instruction pops two bytes of data off the stack and places them in the program counter. Program execution resumes at the new address in the program counter. If the flag is zero, program execution simply continues with the next sequential instruction.

Opcode	*Operand*
RZ	

Operands are not permitted with the RZ instruction.



Cycles:	1 or 3
States:	5 or 11 (6 or 12 on 8085)
Addressing:	register indirect
Flags:	none

Example:

For the sake of brevity, an example is given for the RET instruction but not for each of its closely related variants.

SBB — SUBTRACT WITH BORROW

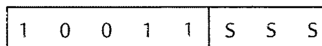
SBB subtracts one byte of data and the setting of the carry flag from the contents of the accumulator. The result is stored in the accumulator. SBB then updates the setting of the carry flag to indicate the outcome of the operation.

SBB's use of the carry flag enables the program to subtract multi-byte strings. SBB incorporates the carry flag by adding it to the byte to be subtracted from the accumulator. It then subtracts the result from the accumulator by using two's complement addition. These preliminary operations occur in the processor's internal work registers so that the source data remains unchanged.

Subtract Register from Accumulator with Borrow

Opcode	*Operand*
SBB	reg

The operand must specify one of the registers A through E, H or L. This instruction subtracts the contents of the specified register and the carry flag from the accumulator and stores the result in the accumulator.



Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

Subtract Memory from Accumulator with Borrow

Opcode	*Operand*
SBB	M

This instruction subtracts the carry flag and the contents of the memory location addressed by the H and L registers from the accumulator and stores the result in the accumulator.

1	0	0	1	1	1	1	0

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Example:

Assume that register B contains 2, the accumulator contains 4, and the carry flag is set to 1. The instruction SBB B operates as follows:

2H + carry = 3H
2's complement of 3H = 11111101

```
Accumulator = 00000100
              11111101
              --------
              00000001 = 1H
```

Notice that this two's complement addition produces a carry. When SBB complements the carry bit generated by the addition, the carry flag is reset OFF. The flag settings resulting from the SBB B instruction are as follows:

Carry	=	0
Sign	=	0
Zero	=	0
Parity	=	0
Aux. Carry	=	1

SBI — SUBTRACT IMMEDIATE WITH BORROW

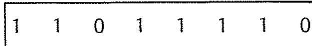
SBI subtracts the contents of the second instruction byte and the setting of the carry flag from the contents of the accumulator. The result is stored in the accumulator.

SBI's use of the carry flag enables the program to subtract multi-byte strings. SBI incorporates the carry flag by adding it to the byte to be subtracted from the accumulator. It then subtracts the result from the accumulator by using two's complement addition. These preliminary operations occur in the processor's internal work registers so that the immediate source data remains unchanged.

The assembler s relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

Opcode	*Operand*
SBI	data

The operand must specify the data to be subtracted. This data may be in the form of a number, an ASCII constant, the label of some perviously defined value, or an expression. The data may not exceed one byte.



1	1	0	1	1	1	1	0

Cycles:	2
States:	7
Addressing:	immediate
Flags:	Z,S,P,CY,AC

Example:

The following sequence of instructions enables the program to test the setting of the carry flag:

```
XRA     A
SBI     1
```

The exclusive OR with the accumulator clears the accumulator to zeros but does not affect the setting of the carry flag. (The XRA instruction is explained later in this chapter.) When the carry flag is OFF, SBI 1 yields a minus one. When the flag is set ON, SBI 1 yields a minus two.

NOTE

This example is included for illustrative purposes. In most cases, the carry flag can be tested more efficiently by using the JNC instruction (jump if no carry).

SHLD — STORE H AND L DIRECT

SHLD stores a copy of the L register in the memory location specified in bytes two and three of the SHLD instruction. SHLD then stores a copy of the H register in the next higher memory location.

Opcode	*Operand*
SHLD	address

The address may be stated as a number, a previously defined label, or an expression.

SHLD is one of the instructions provided for saving the contents of the H and L registers. Alternately, the H and L data can be placed in the D and E registers (XCHG instruction) or placed on the stack (PUSH and XTHL instructions).

0 0 1 0 0 0 1 0
low addr
high addr

Cycles: 5
States: 16
Addressing: direct
Flags: none

Example:

Assume that the H and L registers contain 0AEH and 29H, respectively. The following is an illustration of the effect of the SHLD 10AH instruction:

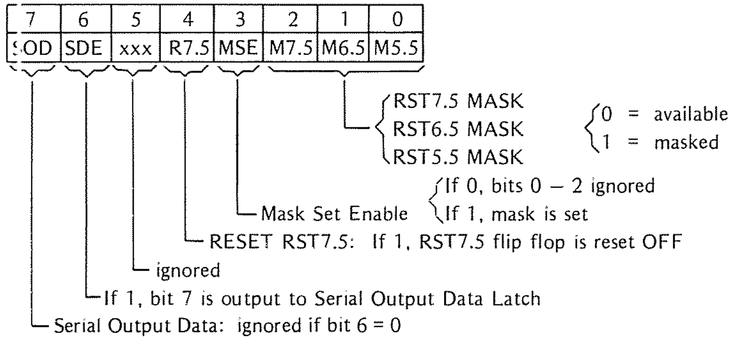
	MEMORY ADDRESS			
	109	10A	10B	10C
Memory Before SHLD	00	00	00	00
Memory After SHLD	00	29	AE	00

SIM (8085 PROCESSOR ONLY) SET INTERRUPT MASK

SIM is a multi-purpose instruction that uses the current contents of the accumulator to perform the following functions: Set the interrupt mask for the 8085's RST5.5, RST6.5, and RST7.5 hardware interrupts; reset RST7.5's edge sensitive input; and output bit 7 of the accumulator to the Serial Output Data latch.

Opcode	*Operand*
SIM	

Operands are not permitted with the SIM instruction. However, you must be certain to load the desired bit configurations into the accumulator before executing the SIM instruction. SIM interprets the bits in the accumulator as follows:



Accumulator bits 3 and 6 function as enable switches. If bit 3 is set ON (set to 1), the set mask function is enabled. Bits 0 through 2 then mask or leave available the corresponding RST interrupt. A 1 bit masks the interrupt making it unavailable; a 0 bit leaves the interrupt available. If bit 3 is set OFF (reset to 0), bits 0 through 2 have no effect. Use this option when you want to send a serial output bit without affecting the interrupt mask.

Notice that the DI (Disable Interrupts) instruction overrides the SIM instruction. Whether masked or not, RST5.5, RST6.5, and RST7.5 are disabled when the DI instruction is in effect. Use the RIM (Read Interrupt Mask) instruction to determine the current settings of the interrupt flag and the interrupt masks.

If bit 6 is set to 1, the serial output data function is enabled. The processor latches accumulator bit 7 into the SOD output where it can be accessed by a peripheral device. If bit 6 is reset to 0, bit 7 is ignored.

A 1 in accumulator bit 4 resets OFF the RST7.5 input flip flop. Unlike RST5.5 and 6.5, RST7.5 is sensed via a processor flip flop that is set when a peripheral device issues a pulse with a rising edge. This edge triggered input supports devices that cannot maintain an interrupt request until serviced. RST7.5 is also useful when a device does not require any explicit hardware service for each interrupt. For example, the program might increment and test an event counter for each interrupt rather than service the device directly.

The RST7.5 flip flop remains set until reset by 1) issuing a RESET to the 8085, 2) recognizing the interrupt, or 3) setting accumulator bit 4 and executing a SIM instruction. The Reset RST7.5 feature of the SIM instruction allows the program to override the interrupt.

The RST7.5 input flip flop is not affected by the setting of the interrupt mask or the DI instruction and therefore can be set at any time. However, the interrupt cannot be serviced when RST7.5 is masked or a DI instruction is in effect.

0	0	1	1	0	0	0	0

Cycles:	1
States:	4
Flags:	none

Example 1: Assume that the accumulator contains the bit pattern 00011100. The SIM instruction resets the RST7.5 flip flop and sets the RST7.5 interrupt mask. If an RST7.5 interrupt is pending when this SIM instruction is executed, it is overridden without being serviced. Also, any subsequent RST7.5 interrupt is masked and cannot be serviced until the interrupt mask is reset.

Example 2: Assume that the accumulator contains the bit pattern 11001111. The SIM instruction masks out the RST5.5, RST6.5, and RST7.5 level interrupts and latches a 1 bit into the SOD input. By contrast, the bit pattern 10000111 has no effect since the enable bits 3 and 6 are not set to ones.

SPHL — MOVE H&L TO SP

SPHL loads the contents of the H and L registers into the SP (Stack Pointer) register.

Opcode	*Operand*
SPHL	

Operands are not permitted with the SPHL instruction.

SP is a special purpose 16-bit register used to address the stack; the stack must be in random access memory (RAM). Because different applications use different memory configurations, the user program must load the SP register with the stack's beginning address. The stack is usually assigned to the highest available location in RAM. The hardware decrements the stack pointer as items are added to the stack and increments the pointer as items are removed.

The stack pointer must be initialized before any instruction attempts to access the stack. Typically, stack initialization occurs very early in the program. Once established, the stack pointer should be altered with caution. Arbitrary use of SPHL can cause the loss of stack data.

1	1	1	1	1	0	0	1

Cycles:	1
States:	5 (6 on 8085)
Addressing:	register
Flags:	none

Example:

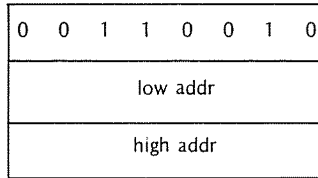
Assume that the H and L registers contain 50H and 0FFH, respectively. SPHL loads the stack pointer with the value 50FFH.

STA — STORE ACCUMULATOR DIRECT

STA stores a copy of the current accumulator contents into the memory location specified in bytes two and three of the STA instruction.

Opcode	*Operand*
STA	address

The address may be stated as a number, a previously defined label, or an expression. The assembler inverts the high and low address bytes when it builds the instruction.



Cycles:	4
States:	13
Addressing:	direct
Flags:	none

Example:

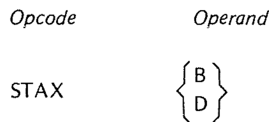
The following instruction stores a copy of the contents of the accumulator at memory location 5B3H:

```
STA     5B3H
```

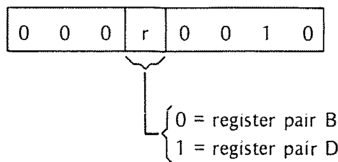
When assembled, the previous instruction has the hexadecimal value 32 B3 05. Notice that the assembler inverts the high and low order address bytes for proper storage in memory.

STAX — STORE ACCUMULATOR INDIRECT

The STAX instruction stores a copy of the contents of the accumulator into the memory location addressed by register pair B or register pair D.



The operand B specifies the B and C register pair; D specifies the D and E register pair. This instruction may specify only the B or D register pair.



Cycles:	2
States:	7
Addressing:	register indirect
Flags:	none

Example:

If register B contains 3FH and register C contains 16H, the following instruction stores a copy of the contents of the accumulator at memory location 3F16H:

```
STAX   B
```

STC — SET CARRY

STC sets the carry flag to one. No other flags are affected.

Opcode	*Operand*
STC	

Operands are not permitted with the STC instruction.

0	0	1	1	0	1	1	1

Cycles:	1
States:	4
Flags:	CY

When used in combination with the rotate accumulator through the carry flag instructions, STC allows the program to modify individual bits.

SUB — SUBTRACT

The SUB instruction subtracts one byte of data from the contents of the accumulator. The result is stored in the accumulator. SUB uses two's complement representation of data as explained in Chapter 2. Notice that the SUB instruction excludes the carry flag (actually a 'borrow' flag for the purposes of subtraction) but sets the flag to indicate the outcome of the operation.

Subtract Register from Accumulator

Opcode	*Operand*
SUB	reg

The operands must specify one of the registers A through E, H or L. The instruction subtracts the contents of the specified register from the contents of the accumulator using two's complement data representation. The result is stored in the accumulator.

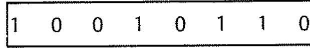
1	0	0	1	0	S	S	S

Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

Subtract Memory from Accumulator

Opcode	*Operand*
SUB	M

This instruction subtracts the contents of the memory location addressed by the H and L registers from the contents of the accumulator and stores the result in the accumulator. M is a symbolic reference to the H and L registers.



Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Example:

Assume that the accumulator contains 3EH. The instruction SUB A subtracts the contents of the accumulator from the accumulator and produces a result of zero as follows:

	3EH =	00111110	
	+(−3EH) =	11000001	one's complement
		1	add one to produce two's complement
carry out = 1		00000000	result = 0

The condition flags are set as follows:

Carry	=	0
Sign	=	0
Zero	=	1
Parity	=	1
Aux. Carry	=	1

Notice that the SUB instruction complements the carry generated by the two's complement addition to form a 'borrow' flag. The auxiliary carry flag is set because the particular value used in this example causes a carry out of bit 3.

SUI — SUBTRACT IMMEDIATE

SUI subtracts the contents of the second instruction byte from the contents of the accumulator and stores the result in the accumulator. Notice that the SUI instruction disregards the carry ('borrow') flag during the subtraction but sets the flag to indicate the outcome of the operation.

Opcode	*Operand*
SUI	data

The operand must specify the data to be subtracted. This data may be in the form of a number, an ASCII constant, the label of some previously defined value, or an expression. The data must not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the

HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

1	1	0	1	0	1	1	0

Cycles:	2
States:	7
Addressing:	immediate
Flags:	Z,S,P,CY,AC

Example:

Assume that the accumulator contains the value 9 when the instruction SUI 1 is executed:

Accumulator	=	00001001	=	9H
Immediate data (2's comp)	=	11111111	=	-1H
		00001000	=	8H

Notice that this two's complement addition results in a carry. The SUI instruction complements the carry generated by the addition to form a 'borrow' flag. The flag settings resulting from this operation are as follows:

Carry	=	0
Sign	=	0
Zero	=	0
Parity	=	0
Aux. Carry	=	1

XCHG — EXCHANGE H AND L WITH D AND E

XCHG exchanges the contents of the H and L registers with the contents of the D and E registers.

Opcode	*Operand*
XCHG	

Operands are not allowed with the XCHG instruction.

XCHG both saves the current H and L and loads a new address into the H and L registers. Since XCHG is a register-to-register instruction, it provides the quickest means of saving and/or altering the H and L registers.

1	1	1	0	1	0	1	1

Cycles:	1
States:	4
Addressing:	register
Flags:	none

Example:

Assume that the H and L registers contain 1234H, and the D and E registers contain 0ABCDH. Following execution of the XCHG instruction, H and L contain 0ABCDH, and D and E contain 1234H.

XRA — EXCLUSIVE OR WITH ACCUMULATOR

XRA performs an exclusive OR logical operation using the contents of the specified byte and the accumulator. The result is placed in the accumulator.

Summary of Logical Operations

AND produces a one bit in the result only when the corresponding bits in the test data and the mask data are ones.

OR produces a one bit in the result when the corresponding bits in either the test data or the mask data are ones.

Exclusive OR produces a one bit only when the corresponding bits in the test data and the mask data are different; i.e., a one bit in either the test data or the mask data – but not both – produces a one bit in the result.

AND	OR	EXCLUSIVE OR
1010 1010	1010 1010	1010 1010
0000 1111	0000 1111	0000 1111
0000 1010	1010 1111	1010 0101

XRA Register with Accumulator

Opcode	*Operand*
XRA	reg

The operand must specify one of the registers A through E, H or L. This instruction performs an exclusive OR using the contents of the specified register and the accumulator and stores the result in the accumulator. The carry and auxiliary carry flags are reset to zero.

1	0	1	0	1	S	S	S

Cycles:	1
States:	4
Addressing:	register
Flags:	Z,S,P,CY,AC

XRA Memory with Accumulator

Opcode	*Operand*
XRA	M

The contents of the memory location specified by the H and L registers is exclusive-ORed with the contents of the accumulator. The result is stored in the accumulator. The carry and auxiliary carry flags are reset to zero.

1	0	1	0	1	1	1	0

Cycles:	2
States:	7
Addressing:	register indirect
Flags:	Z,S,P,CY,AC

Examples:

Since any bit exclusive-ORed with itself produces zero, XRA is frequently used to zero the accumulator. The following instructions zero the accumulator and the B and C registers.

```
XRA    A
MOV    B,A
MOV    C,A
```

Any bit exclusive-ORed with a one bit is complemented. Thus, if the accumulator contains all ones (OFFH), the instruction XRA B produces the one's complement of the B register in the accumulator.

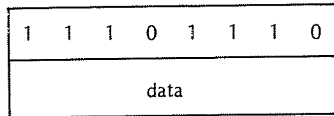
XRI — EXCLUSIVE OR IMMEDIATE WITH ACCUMULATOR

XRI performs an exclusive OR operation using the contents of the second instruction byte and the contents of the accumulator. The result is placed in the accumulator. XRI also resets the carry and auxiliary carry flags to zero.

Opcode	*Operand*
XRI	data

The operand must specify the data to be used in the OR operation. This data may be in the form of a number, an ASCII constant, the label of some previously defined value, or an expression. The data may not exceed one byte.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in the operand expression of an immediate instruction, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.



Cycles:	2
States:	7
Addressing:	immediate
Flags:	Z,S,P,CY,AC

Summary of Logical Operations

AND produces a one bit in the result only when the corresponding bits in the test data and the mask data are ones.

OR produces a one bit in the result when the corresponding bits in either the test data or the mask data are ones.

Exclusive OR produces a one bit only when the corresponding bits in the test data and the mask data are different; i.e., a one bit in either the test data or the mask data — but not both — produces a one bit in the result.

AND	OR	EXCLUSIVE OR
1010 1010	1010 1010	1010 1010
0000 1111	0000 1111	0000 1111
0000 1010	1010 1111	1010 0101

Example:

Assume that a program uses bits 7 and 6 of a byte as flags that control the calling of two subroutines. The program tests the bits by rotating the contents of the accumulator until the desired bit is in the carry flag; a CC instruction (Call if Carry) tests the flag and calls the subroutine if required.

Assume that the control flag byte is positioned normally in the accumulator, and the program must set OFF bit 6 and set bit 7 ON. The remaining bits, which are status flags used for other purposes, must not be altered. Since any bit exclusive-ORed with a one is complemented, and any bit exclusive-ORed with a zero remains unchanged, the following instruction is used:

```
XRI        11000000B
```

The instruction has the following results:

Accumulator	=	01001100
Immediate data	=	11000000
		10001100

XTHL EXCHANGE H&L WITH TOP OF STACK

XTHL exchanges two bytes from the top of the stack with the two bytes stored in the H and L registers. Thus, XTHL both saves the current contents of the H and L registers and loads new values into H and L.

Opcode	*Operand*
XTHL	

Operands are not allowed with the XTHL instruction.

XTHL exchanges the contents of the L register with the contents of the memory location specified by the SP (Stack Pointer) register. The contents of the H register are exchanged with the contents of SP+1.

1	1	1	0	0	0	1	1

Cycles: 5
States: 18 (16 on 8085)
Addressing: register indirect
Flags: none

Example:

Assume that the stack pointer register contains 10ADH; register H contains 0BH and L contains 3CH; and memory locations 10ADH and 10AEH contain F0H and 0DH, respectively. The following is an illustration of the effect of the XTHL instruction:

	MEMORY ADDRESS				H	L
	10AC	10AD	10AE	10AF		
Before XTHL	FF	F0	0D	FF	0B	3C
After XTHL	FF	3C	0B	FF	0D	F0

The stack pointer register remains unchanged following execution of the XTHL instruction.

4. ASSEMBLER DIRECTIVES

This chapter describes the assembler directives used to control the 8080/85 assembler in its generation of object code. This chapter excludes the macro directives, which are discussed as a separate topic in Chapter 5.

Generally, directives have the same format as instructions and can be interspersed throughout your program. Assembler directives discussed in this chapter are grouped as follows:

GENERAL DIRECTIVES:

- Symbol Definition

  EQU
  SET

- Data Definition

  DB
  DW

- Memory Reservation

  DS

- Conditional Assembly

  IF
  ELSE
  ENDIF

- Assembler Termination

  END

LOCATION COUNTER CONTROL AND RELOCATION:

- Location Counter Control

  ASEG
  DSEG
  CSEG
  ORG

- Program Linkage

  PUBLIC
  EXTRN
  NAME
  STKLN

Three assembler directives — EQU, SET, and MACRO — have a slightly different format from assembly language instructions. The EQU, SET, and MACRO directives require a *name* for the symbol or macro being defined to be present in the label field. Names differ from labels in that they must *not* be terminated with a colon (:) as labels are. Also, the LOCAL and ENDM directives prohibit the use of the label field.

The MACRO, ENDM, and LOCAL directives are explained in Chapter 5.

SYMBOL DEFINITION

The assembler automatically assigns values to symbols that appear as instruction labels. This value is the current setting of the location counter when the instruction is assembled. (The location counters are explained under 'Address Control and Relocation,' later in this chapter.)

You may define other symbols and assign them values by using the EQU and SET directives. Symbols defined using EQU cannot be redefined during assembly; those defined by SET can be assigned new values by subsequent SET directives.

The name required in the label field of an EQU or SET directive must *not* be terminated with a colon.

Symbols defined by EQU and SET have meaning throughout the remainder of the program. This may cause the symbol to have illegal multiple definitions when the EQU or SET directive appears in a macro definition. Use the LOCAL directive (described in Chapter 5) to avoid this problem.

EQU Directive

EQU assigns the value of 'expression' to the name specified in the label field.

Label	*Opcode*	*Operand*
name	EQU	expression

The required name in the label field may not be terminated with a colon. This name cannot be redefined by a subsequent EQU or SET directive. The EQU expression cannot contain any external symbol. (External symbols are explained under 'Location Counter Control and Relocation,' later in this chapter.)

Assembly-time evaluation of EQU expressions always generates a modulo 64K address. Thus, the expression always yields a value in the range 0-65,536.

Example:

The following EQU directive enters the name ONES into the symbol table and assigns the binary value 11111111 to it:

```
ONES        EQU        0FFH
```

The value assigned by the EQU directive can be recalled in subsequent source lines by referring to its assigned name as in the following IF directive:

```
IF TYPE EQ ONES
.
.
.
ENDIF
```

SET Directive

SET assigns the value of 'expression' to the name specified in the label field.

Label	*Opcode*	*Operand*
name	SET	expression

The assembler enters the value of 'expression' into the symbol table. Whenever 'name' is encountered subsequently in the assembly, the assembler substitutes its value from the symbol table. This value remains unchanged until altered by a subsequent SET directive.

The function of the SET directive is identical to EQU except that 'name' can appear in multiple SET directives in the same program. Therefore, you can alter the value assigned to 'name' throughout the assembly.

Assembly-time evaluation of SET expressions always generates a modulo 64K address. Thus, the expression always yields a value in the range 0-65,536.

Examples:

Label	*Opcode*	*Operand*	*Assembled Code*
IMMED	SET	5	
	ADI	IMMED	C605
IMMED	SET	10H-6	
	ADI	IMMED	C60A

DATA DEFINITION

The DB (define byte) and DW (define word) directives enable you to define data to be stored in your program. Data can be specified in the form of 8-bit or 16-bit values, or as a string of text characters.

DB Directive

The DB directive stores the specified data in consecutive memory locations starting with the current setting of the location counter.

Label	*Opcode*	*Operands*
optional:	DB	expression(s) or string(s)

The operand field of the DB directive can contain a list of expressions and/or text strings. The list can contain up to eight total items; list items must be separated by commas. Because of limited workspace, the assembler may not be able to handle a total of eight items when the list includes a number of complex expressions. If you ever have this problem, it is easily solved: simply use two or more directives to shorten the list.

Expressions must evaluate to 1-byte (8-bit) numbers in the range –256 through 255. Text strings may comprise a maximum of 128 ASCII characters enclosed in quotes.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in an operand expression of the DB directive, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

If the optional label is present, it is assigned the starting value of the location counter, and thus references the first byte stored by the DB directive. Therefore, the label STR in the following examples refers to the letter T of the string TIME.

Examples:

Label	*Opcode*	*Operands*	*Assembled Code*
STR:	DB	'TIME'	54494D45
HERE:	DB	0A3H	A3
WORD1:	DB	–03H,5*2	FD0A

DW Directive

The DW directive stores each 16-bit value from the expression list as an address. The values are stored starting at the current setting of the location counter.

Label	*Opcode*	*Operands*
optional:	DW	expression list

The least significant eight bits of the first value in the expression list are stored at the current setting of the location counter; the most significant eight bits are stored at the next higher location. This process is repeated for each item in the expression list.

Expressions evaluate to 1–word (16-bit) numbers, typically addresses. If an expression evaluates to a single byte, it is assumed to be the low order byte of a 16-bit word where the high order byte is all zeros.

List items must be separated by commas. The list can contain up to eight total items. Because of limited work-space, the assembler may not be able to handle eight complex expressions. If you ever have this problem, simply use two or more DW directives to shorten the list.

The reversed order for storing the high and low order bytes is the typical format for addresses stored in memory. Thus, the DW directive is commonly used for storing address constants.

Strings containing one or two ASCII characters enclosed in quotation marks may also appear in the expression list. When using such strings in your program, remember that the characters are stored in reversed order. Specifying a string longer than two characters causes an error.

If the optional label is present, it is assigned the starting address of the location counter, and thus references the first byte stored by the DW directive. (This is the low order byte of the first item in the expression list.)

Examples:

Assume that COMP and FILL are labels defined elsewhere in the program. COMP addresses memory location 3B1CH. FILL addresses memory location 3EB4H.

Label	*Opcode*	*Operands*	*Assembled Code*
ADDR1:	DW	COMP	1C3B
ADDR2:	DW	FILL	B43E
STRNG:	DW	'A','AB'	41004241
FOUR:	DW	4H	0400

MEMORY RESERVATION

DS Directive

The DS directive can be used to define a block of storage.

Label	*Opcode*	*Operand*
optional:	DS	expression

The value of 'expression' specifies the number of bytes to be reserved for data storage. In theory, this value may range from 00H through 0FFFFH; in practice, you will reserve no more storage than will fit in your available memory and still leave room for the program.

Any symbol appearing in the operand expression must be defined before the assembler reaches the DS directive.

Unlike the DB and DW directives, DS assembles no data into your program. The contents of the reserved storage are unpredictable when program execution is initiated.

If the optional label is present, it is assigned the current value of the location counter, and thus references the first byte of the reserved memory block.

If the value of the operand expression is zero, no memory is reserved. However, if the optional label is present, it is assigned the current value of the location counter.

The DS directive reserves memory by incrementing the location counter by the value of the operand expression.

Example:

```
TTYBUF:    DS    72    ;RESERVE 72 BYTES FOR
                       ;A TERMINAL OUTPUT BUFFER
```

Programming Tips: Data Description and Access

Random Access Versus Read Only Memory

When coding data descriptions, keep in mind the mix of ROM and RAM in your application.

Generally, the DB and DW directives define constants, items that can be assigned to ROM. You can use these items in your program, but you cannot modify them. If these items are assigned to RAM, they have an initial value that your program can modify during execution. Notice, however, that these initial values must be reloaded into memory prior to each execution of the program.

Variable data in memory must be assigned to RAM.

Data Description

Before coding your program, you must have a thorough understanding of its input and output data. But you'll probably find it more convenient to postpone coding the data descriptions until the remainder of the program is fairly well developed. This way you will have a better idea of the constants and workareas needed in your program. Also, the organization of a typical program places instructions in lower memory, followed by the data, followed by the stack.

Data Access

Accessing data from memory is typically a two-step process: First you tell the processor where to find the data, then the processor fetches the data from memory and loads it into a register, usually the accumulator. Therefore, the following code sequences have the identical effect of loading the ASCII character A into the accumulator.

```
AAA:      DB    'A'          ALPHA:    DB    'ABC'
          .                            .
          .                            .
          LXI   B,AAA                  LXI   B,ALPHA
          LDAX  B                      LDAX  B
          .                            .
          .                            .
```

In the examples, the LXI instructions load the address of the desired data into the B and C registers. The LDAX instructions then load the accumulator with one byte of data from the address specified in the B and C registers. The assembler neither knows nor cares that only one character from the three-character field ALPHA has been accessed. The program must account for the characters at ALPHA+1 and ALPHA+2, as in the following coding sequence:

```
ALPHA:    DB      'ABC'        ;DEFINE ALPHA
          .
          LXI     B,ALPHA      ;LOAD ADDRESS OF ALPHA
          LDAX    B            ;FETCH 1ST ALPHA CHAR
          .
          INX     B            ;SET B TO ALPHA+1
          LDAX    B            ;FETCH 2ND ALPHA CHAR
          .
          INX     B            ;SET B TO ALPHA+2
          LDAX    B            ;FETCH 3RD ALPHA CHAR
```

The coding above is acceptable for short data fields like ALPHA. For longer fields, you can conserve memory by setting up an instruction sequence that is executed repeatedly until the source data is exhausted.

Add Symbols for Data Access

The following example was presented earlier as an illustration of the DS directive:

Label	*Opcode*	*Operand*	*Comment*
TTYBUF:	DS	72	;RESERVE TTY BUFFER

To access data in this buffer using only expressions such as TTYBUF+1, TTYBUF+2, . . . TTYBUF+72 can be a laborious and confusing chore, especially when you want only selected fields from the buffer. You can simplify this task by subdividing the buffer with the EQU directive:

Label	*Opcode*	*Operand*	*Comment*
TTYBUF:	DS	72	;RESERVE TTY BUFFER
ID	EQU	TTYBUF	;RECORD IDENTIFIER
NAME	EQU	TTYBUF+6	;20-CHAR NAME FIELD
NUMBER	EQU	TTYBUF+26	;10-CHAR EMPLOYEE NUMBER
DEPT	EQU	TTYBUF+36	;5-CHAR DEPARTMENT NUMBER
SSNO	EQU	TTYBUF+41	;SOCIAL SEC. NUMBER
DOH	EQU	TTYBUF+50	;DATE OF HIRE
DESC	EQU	TTYBUF+56	;JOB DESCRIPTION

Subdividing data as shown in the example simplifies data access and provides useful documentation throughout your program. Notice that these EQU directives can be inserted anywhere within the program as you need them, but coding them as shown in the example provides a more useful record description.

CONDITIONAL ASSEMBLY

The IF, ELSE, and ENDIF directives enable you to assemble portions of your program conditionally, that is, only if certain conditions that you specify are satisfied.

Conditional assembly is especially useful when your application requires custom programs for a number of common options. As an example, assume that a basic control program requires customizing to accept input from one of six different sensing devices and to drive one of five different control devices. Rather than code some thirty separate programs to account for all the possibilities, you can code a single program. The code for the individual sensors and drivers must be enclosed by the conditional directives. When you need to generate a custom program, you can insert SET directives near the beginning of the source program to select the desired sensor and driver routines.

IF, ELSE, ENDIF Directives

Because these directives are used in conjunction, they are described together here.

Label	*Opcode*	*Operand*
optional:	IF	expression
optional:	ELSE	—
optional:	ENDIF	—

The assembler evaluates the expression in the operand field of the IF directive. If bit 0 of the resulting value is one (TRUE), all instructions between the IF directive and the next ELSE or ENDIF directive are assembled. When bit 0 is zero (FALSE) these instructions are ignored. (A TRUE expression evaluates to 0FFFFH and FALSE to 0H; only bit zero need be tested.)

All statements included between an IF directive and its required associated ENDIF directive are defined as an IF-ENDIF block. The ELSE directive is optional, and only one ELSE directive may appear in an IF-ENDIF block. When included, ELSE is the converse of IF. When bit 0 of the expression in the IF directive is zero, all statements between ELSE and the next ENDIF are assembled. If bit 0 is one, these statements are ignored.

Operands are not allowed with the ELSE and ENDIF directives.

An IF-ENDIF block may appear within another IF-ENDIF block. These blocks can be nested to eight levels.

Macro definitions (explained in the next chapter) may appear within an IF-ENDIF block. Conversely, IF-ENDIF blocks may appear within macro definitions. In either case, you must be certain to terminate the macro definition

or IF-ENDIF block so that it can be assembled completely. For example, when a macro definition begins in an IF block but terminates after an ELSE directive, only a portion of the macro can be assembled. Similarly, an IF-ENDIF block begun within a macro definition must terminate within that same macro definition.

NOTE

> Caution is required when symbols are defined in IF-ENDIF blocks and referenced elsewhere within the program. These symbols are undefined when the evaluation of the IF expression suppresses the assembly of the IF-ENDIF block.

Example 1. Simple IF-ENDIF Block:

```
COND1:  IF TYPE EQ 0
        .
        .                       ;ASSEMBLED IF 'TYPE = 0'
        .                       ;IS TRUE
        .
        ENDIF
```

Example 2. IF-ELSE-ENDIF Block:

```
COND2:  IF TYPE EQ 0
        .                       ;ASSEMBLED IF 'TYPE = 0'
        .                       ;IS TRUE
        .
        ELSE
        .
        .
        .                       ;ASSEMBLED IF 'TYPE = 0'
        .                       ;IS FALSE
        .
        ENDIF
```

Example 3. Nested IF's:

```
        COND3:  IF TYPE EQ 0
                .
                .                       ;ASSEMBLED IF 'TYPE = 0'
                .                       ;IS TRUE
                .
                IF MODE EQ 1
                .
        LEVEL   .                       ;ASSEMBLED IF 'TYPE = 0'
          1     .                       ;AND 'MODE = 1' ARE BOTH
                .                       ;TRUE
                ENDIF
                ELSE
                .
LEVEL           .                       ;ASSEMBLED IF 'TYPE = 0'
  2             .                       ;IS FALSE
                .
                IF MODE EQ 2
                .
                .                       ;ASSEMBLED IF 'TYPE = 0'
                .                       ;IS FALSE AND 'MODE = 2'
                .                       ;IS TRUE
        LEVEL   ELSE
          1     .
                .                       ;ASSEMBLED IF 'TYPE = 0'
                .                       ;AND 'MODE = 2' ARE BOTH
                .                       ;FALSE
                ENDIF
                ENDIF
```

ASSEMBLER TERMINATION

END Directive

The END directive identifies the end of the source program and terminates each pass of the assembler.

Label	*Opcode*	*Operand*
optional:	END	expression

Only one END statement may appear in a source program, and it must be the last source statement.

If the optional expression is present, its value is used as the starting address for program execution. If no expression is given, the assembler assumes zero as the starting address.

When a number of separate program modules are to be joined together, only one may specify a program starting address. The module with a starting address is the main module. When source files are combined using the INCLUDE control, there are no restrictions on which source file contains the END.

END-OF-TAPE INDICATION

The EOT directive allows you to specify the physical end of paper tape to simplify assembly of multiple-tape source programs.

EOT Directive

Label	*Opcode*	*Operand*
optional:	EOT	---

When EOT is recognized by the assembler, the message 'NEXT TAPE' is sent to the console and the assembler pauses. After the next tape is loaded, a 'space bar' character received at the console signals continuation of the assembly.

Data in the operand field causes an error.

LOCATION COUNTER CONTROL AND RELOCATION

All the directives discussed in the remainder of this chapter relate directly to program relocation except for the ASEG and ORG directives. These directives are described first for the convenience of readers who do not use the relocation feature.

Location Counter Control (Non-Relocatable Mode)

When you elect not to use the relocation feature, an assembler default generates an ASEG directive for you. The ASEG directive specifies that the program is to be assembled in the non-relocatable mode and establishes a location counter for the assembly.

The location counter performs the same function for the assembler as the program counter performs during execution. It tells the assembler the next memory location available for instruction or data assembly.

Initially, the location counter is set to zero. The location counter can be altered by the ORG (origin) directive.

ORG Directive

The ORG directive sets the location counter to the value specified by the operand expression.

Label	*Opcode*	*Operand*
optional:	ORG	expression

The location counter is set to the value of the operand expression. Assembly-time evaluation of ORG expressions always yields a modulo 64K address. Thus, the expression always yields an address in the range 0 through 65,535. Any symbol in the expression must be previously defined. The next machine instruction or data item is assembled at the specified address.

If no ORG directive is included before the first instruction or data byte in your program, assembly begins at location zero.

Your program can include any number of ORG directives. Multiple ORG's need not specify addresses in ascending sequence, but if you fail to do so, you may instruct the assembler to write over some previously assembled portion of the program.

If the optional label is present, it is assigned the current value of the location counter *before* it is updated by the ORG directive.

Example:

Assume that the current value of the location counter is 0FH (decimal 15) when the following ORG directive is encountered:

```
PAG1:     ORG     0FFH     ;ORG ASSEMBLER TO LOCATION
                           ;0FFH (decimal 225)
```

The symbol PAG1 is assigned the address 0FH. The next instruction or data byte is assembled at location 0FFH.

Introduction to Relocatability

A major feature of this assembler is its system for creating relocatable object code modules. Support for this new feature includes a number of new directives for the assembler and three new programs included in ISIS-II. The three new programs — LIB, LINK, and LOCATE — are described in the ISIS-II System User's Guide. The new assembler directives are described later in this chapter.

Relocatability allows the programmer to code programs or sections of programs without worrying about the final arrangement of the object code in memory. This offers developers of microcomputer systems major advantages in two areas: memory management and modular program development.

Memory Management

When developing, testing, and debugging a system on your Intellec microcomputer development system, your only concern with locating a program is that it doesn't overlap the resident routines of ISIS-II. Because the Intellec system has 32K, 48K, or 64K of random access memory, the location of your future program is not a great concern. However, the program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocatability feature allows you to develop, test, and debug your program on the Intellec development system and then simply relocate the object code to suit your application.

The relocatability feature also has a major advantage at assembly-time: often, large programs with many symbols cannot be assembled because of limited work space for the symbol table. Such a program can be divided into a number of modules that can be assembled separately and then linked together to form a single object program.

Modular Program Development

Although 'relocatability' may seem to be a formidable term, what it really means is that you can subdivide a complex program into a number of smaller, simpler programs. This concept is best illustrated through the use of an example. Assume that a microcomputer program is to control the spark advance on an automobile engine. This requires the program to sample the ambient air temperature, engine air intake temperature, coolant temperature, manifold vacuum, idle sensor, and throttle sensor.

Let us examine the approaches two different programmers might take to solve this problem. Both programmers want to calculate the degree of spark advance or retardation that provides the best fuel economy with the lowest emissions. Programmer A codes a single program that senses all inputs and calculates the correct spark advance. Programmer B uses a modular approach and codes separate programs for each input plus one program to calculate spark advance.

Although Programmer A avoids the need to learn to use the relocatability feature, the modular approach used by Programmer B has a number of advantages you should consider:

- Simplified Program Development

  It is generally easier to code, test, and debug several simple programs than one complex program.

- Sharing the Programming Task

  If Programmer B finds that he is falling behind schedule, he can assign one or more of his subprograms to another programmer. Because of his single program concept, Programmer A will probably have to complete the program himself.

- Ease of Testing

  Programmer B can test and debug most of his modules as soon as they are assembled; Programmer A must test his program as a whole. Notice that Programmer B has an extra advantage if the sensors are being developed at the same time as the program. If one of the sensors is behind schedule, Programmer B can continue developing and testing programs for the sensors that are ready. Because Programmer A cannot test his program until all the sensors are developed, his testing schedule is dependent on events beyond his control.

- Programming Changes

  Given the nature of automotive design, it is reasonable to expect some changes during system development. If a change to one of the sensors requires a programming change, Programmer A must search through his entire program to find and alter the coding for that sensor. Then he must retest the entire program to be certain that those changes do not affect any of the other sensors. By contrast, Programmer B need be concerned only with the module for that one sensor. This advantage continues throughout the life of the program.

DIRECTIVES USED FOR RELOCATION

Several directives have been added to the assembler to support the relocation feature. These fall into the general categories of location counter control and program linkage.

Location Counter Control (Relocatable Programs)

Relocatable programs or program modules may use three location counters. The ASEG, DSEG, and CSEG directives specify which location counter is to be used.

The ASEG directive specifies an absolute code segment. Even in a relocatable program module, you may want to assign certain code segments to specific addresses. For example, restart routines invoked by the RST instruction require specific addresses.

The CSEG directive specifies a relocatable code segment. In general, the CSEG location counter is used for portions of the program that are to be in some form of read-only memory, such as machine instructions and program constants.

The DSEG location counter specifies a relocatable data segment. This location counter is used for program elements that must be located in random access memory.

These directives allow you to control program segmentation at assembly time. The LOCATE program, described in the ISIS-II System User's Guide, gives you control over program segment location. Therefore, the guidelines given above are only general since they can be overridden by the LOCATE program.

Regardless of how many times the ASEG, CSEG, and DSEG directives appear in your program, the assembler produces a single, contiguous module. This module comprises four segments: code, data, stack and memory. The LINK and LOCATE programs are used to combine segments from individual modules and relocate them in memory. These programs are explained in the ISIS-II System User's Guide.

ASEG Directive

ASEG directs the assembler to use the location counter for the absolute program segment.

Label	*Opcode*	*Operand*
optional:	ASEG	——

Operands are not permitted with the ASEG directive.

All instructions and data following the ASEG directive are assembled in the absolute mode. The ASEG directive remains in effect until a CSEG or DSEG directive is encountered.

The ASEG location counter has an initial value of zero. The ORG directive can be used to assign a new value to the ASEG location counter.

When assembly begins, the assembler assumes the ASEG directive to be in effect. Therefore, a CSEG or DSEG must precede the first instruction or data definition in a relocatable module. If neither of these directives appears in the program, the entire program is assembled in absolute mode and can be executed immediately after assembly without using the LINK or LOCATE programs.

CSEG Directive

CSEG directs the assembler to assemble subsequent instructions and data in the relocatable mode using the code segment location counter.

Label	*Opcode*	*Operand*
optional:	CSEG	{ blank / PAGE / INPAGE }

When a program contains multiple CSEG directives, all CSEG directives throughout the program must specify the same operand. The operand of a CSEG directive has no effect on the current assembly, but is stored with the object code to be passed to the LINK and LOCATE programs. (These programs are described in the ISIS-II System User's Guide.) The LOCATE program uses this information to determine relocation boundaries when it joins this code segment to code segments from other programs. The meaning of the operand is as follows:

- blank — This code segment may be relocated to the next available byte boundary.
- PAGE — This code segment must begin on a page boundary when relocated. Page boundaries occur in multiples of 256 bytes beginning with zero (0, 256, 512, etc.).
- INPAGE — This code segment must fit within a single page when relocated.

The CSEG directive remains in effect until an ASEG or DSEG directive is encountered.

The code segment location counter has an initial value of zero. The ORG directive can be used to assign a new value to the CSEG location counter.

DSEG Directive

DSEG directs the assembler to assemble subsequent instructions and data in the relocatable mode using the data segment location counter.

Label	*Opcode*	*Operand*
optional:	DSEG	{ blank / PAGE / INPAGE }

When multiple DSEG directives appear in a program, they must all specify the same operand throughout the program. The operands for the DSEG directive have the same meaning as for the CSEG directive except that they apply to the data segment.

There is no interaction between the operands specified for the DSEG and CSEG directives. Thus, a code segment can be byte relocatable while the data segment is page relocatable.

The DSEG directive remains in effect until an ASEG or CSEG directive is encountered.

The data segment location counter has an initial value of zero. The ORG directive can be used to assign a new value to the DSEG location counter.

ORG Directive (Relocatable Mode)

The ORG directive can be used to alter the value of the location counter presently in use.

Label	*Opcode*	*Operand*
optional:	ORG	expression

There are three location counters, but only one location counter is in use at any given point in the program. Which one depends on whether the ASEG, CSEG, or DSEG directive is in effect.

Any symbol used in the operand expression must have been previously defined. An exception causes phase errors for all labels that follow the ORG and a label error if the undefined error is defined later.

When the ORG directive appears in a relocatable program segment, the value of its operand expression must be either absolute or relocatable within the current segment. Thus, if the ORG directive appears within a data segment, the value of its expression must be relocatable within the data segment. An error occurs if the expression evaluates to an address in the code segment.

If the optional label is present, it is assigned the current value of the location counter presently in use *before* the ORG directive is executed.

Program Linkage Directives

Modular programming and the relocation feature enable you to assemble and test a number of separate programs that are to be joined together and executed as a single program. Eventually, it becomes necessary for these separate programs to communicate information among themselves. Establishing such communication is the function of the program linkage directives.

A program may share its data addresses and instruction addresses with other programs. Only items having an entry in the symbol table can be shared with other programs; therefore, the item must be assigned a name or a label when it is defined in the program. Items to be shared with other programs must be declared in a PUBLIC directive.

Your program can directly access data or instructions defined in another program if you know the actual address of the item, but this is unlikely when both programs use relocation. Your program can also gain access to data or instructions declared as PUBLIC in other programs. Notice, however, that the assembler normally

flags as an error any reference to a name or label that has not been defined in your program. To avoid this, you must provide the assembler with a list of items used in your program but defined in some other program. These items must be declared in an EXTRN directive.

The two remaining program linkage directives, NAME and STKLN, are individually explained later in this chapter.

PUBLIC Directive

The PUBLIC directive makes each of the symbols listed in the operand field available for access by other programs.

Label	*Opcode*	*Operands*
optional:	PUBLIC	name—list

Each item in the operand name—list must be the name or label assigned to data or an instruction elsewhere in this program. When multiple names appear in the list, they must be separated by commas. Each name may be declared PUBLIC only once in a program module. Reserved words and external symbols (see the EXTRN directive below) cannot be declared to be PUBLIC symbols.

PUBLIC directives may appear anywhere within a program module.

If an item in the operand name—list has no corresponding entry in the symbol table (implying that it is undefined), it is flagged as an error.

Example:

```
        PUBLIC    SIN,COS,TAN,SQRT
```

EXTRN Directive

The EXTRN directive provides the assembler with a list of symbols referenced in this program but defined in a different program. Because of this, the assembler establishes linkage to the other program and does not flag the undefined references as errors.

Label	*Opcode*	*Operands*
optional:	EXTRN	name—list

Each item in the name—list identifies a symbol that may be referenced in this program but is defined in another program. When multiple items appear in the list, they must be separated by commas.

If a symbol in the operand name—list is also defined in this program by the user, or is a reserved symbol, the effect is the same as defining the same symbol more than once in a program. The assembler flags this error.

EXTRN directives may appear anywhere within a program module.

A symbol may be declared to be external only once in a program module. Symbols declared to be PUBLIC cannot also be declared to be EXTRN symbols.

If you omit a symbol from the name—list but reference it in the program, the symbol is undefined. The assembler flags this error. You may include symbols in the operand name—list that are not referenced in the program without causing an error.

Example:

```
EXTRN     ENTRY,ADDRTN,BEGIN
```

NAME Directive

The NAME directive assigns a name to the object module generated by this assembly.

Label	*Opcode*	*Operand*
optional:	NAME	module—name

The NAME directive requires the presence of a module—name in the operand field. This name must conform to the rules for defining symbols.

Module names are necessary so that you can refer to a module and specify the proper sequence of modules when a number of modules are to be bound together.

The NAME directive must precede the first data or instruction coding in the source program, but may follow comments and control lines.

If the NAME directive is missing from the program, the assembler supplies a default NAME directive with the module—name MODULE. This will cause an error if you attempt to bind together several object program modules and more than one has the name MODULE. Also, if you make an error coding the NAME directive, the default name MODULE is assigned.

The module—name assigned by the NAME directive appears as part of the page heading in the assembly listing.

Example:

```
NAME     MAIN
```

STKLN Directive

Regardless of the number of object program modules you may bind together, only one stack is generated. The STKLN directive allows you to specify the number of bytes to be reserved for the stack for each module.

Label	*Opcode*	*Operand*
optional:	STKLN	expression

The operand expression must evaluate to a number which will be used as the maximum size of the stack.

When the STKLN directive is omitted, the assembler provides a default STKLN of zero. This is useful when multiple programs are bound together; only one stack will be generated, so only one program module need specify the stack size. However, you should provide a STKLN if your module is to be tested separately and uses the stack.

If your program includes more than one STKLN directive, only the last value assigned is retained.

Example:

```
STKLN     100
```

STACK and MEMORY Reserved Words

The reserved words STACK and MEMORY are not directives but are of interest to programmers using the relocation feature. These reserved words are external references whose addresses are supplied by the LOCATE program.

STACK is the symbolic reference to the stack origin address. You need this address to initialize the stack pointer register. Also, you can base data structures on this address using symbolic references such as STACK+1, STACK+2, etc.

MEMORY is the symbolic reference to the first byte of unused memory past the end of your program. Again, you can base data structures on this address using symbolic references such as MEMORY, MEMORY+1, etc.

Programming Tips: Testing Relocatable Modules

The ability to test individual program modules is a major advantage of modular programming. However, many program modules are not logically self-sufficient and require some modification before they can be tested. The following is a discussion of some of the more common modifications that may be required.

Initialization Routines

In most complete programs, a number of housekeeping or initialization procedures are performed when execution first begins. If the program module you are testing relies on initialization procedures assigned to a different module, you must duplicate those procedures in the module to be tested. (Notice, however, that you can link any number of modules together for testing.)

One of the most important initialization procedures is to set the stack pointer. The LOCATE program determines the origin of the stack.

Your program should include the following instruction to initialize the stack pointer:

```
LXI     SP,STACK
```

Input/Output

When testing program modules, it is likely that some input or output procedures appear in other modules. Your program must simulate any of these procedures it needs to operate. Since your Intellec development system probably has considerably more random access memory than you need to test a program module, you may be able to simulate input and output data right in memory. The LOCATE program supplies an address for the reserved word MEMORY; this is the address of the first byte of unused memory past the end of your program. You can access this memory using the symbolic reference MEMORY, MEMORY+1, and so on. This memory can be used for storing test data or even for a program that generates test data.

Remove Coding Used for Testing

After testing your program, be certain to remove any code you inserted for testing. In particular, make certain that only one module in the complete program initializes the stack pointer.

5. MACROS

INTRODUCTION TO MACROS

Why Use Macros?

A macro is essentially a facility for replacing one set of parameters with another. In developing your program, you will frequently find that many instruction sequences are repeated several times with only certain parameters changed.

As an example, suppose that you code a routine that moves five bytes of data from one memory location to another. A little later, you find yourself coding another routine to move four bytes from a different source field to a different destination field. If the two routines use the same coding techniques, you will find that they are identical except for three parameters: the character count, the source field starting address, and the destination field starting address. Certainly it would be handy if there were some way to regenerate that original routine substituting the new parameters rather than rewrite that code yourself. The macro facility provides this capability and offers several other advantages over writing code repetitiously:

- The tedium of frequent rewrite (and the probability of error) is reduced.
- Symbols used in macros can be restricted so that they have meaning only within the macro itself. Therefore, as you code your program, you need not worry that you will accidentally duplicate a symbol used in a macro. Also, a macro can be used any number of times in the same program without duplicating any of its own symbols.
- An error detected in a macro need be corrected only once regardless of how many times the macro appears in the program. This reduces debugging time.
- Duplication of effort between programmers can be reduced. Useful functions can be collected in a library to allow macros to be copied into different programs.

In addition, macros can be used to improve program readability and to create structured programs. Using macros to segment code blocks provides clear program notation and simplifies tracing the flow of the program.

What Is A Macro?

A macro can be described as a routine *defined* in a formal sequence of prototype instructions that, when *called* within a program, results in the replacement of each such call with a code *expansion* consisting of the actual instructions represented.

The concepts of macro definition, call, and expansion can be illustrated by a typical business form letter, where the prototype instructions consist of preset text. For example, we could define a macro CNFIRM with the text

> Air Flight welcomes you as a passenger.
> Your flight number FNO leaves at DTIME and arrives in DEST at ATIME.

This macro has four dummy parameters to be replaced, when the macro is called, by the actual flight number, departure time destination, and arrival time. Thus the macro call might look like

```
CNFIRM      123, '10:45', 'Ontario', '11:52'
```

A second macro, CAR, could be called if the passenger has requested that a rental car be reserved at the destination airport. This macro might have the text

> Your automobile reservation has been confirmed with MAKE rent-a-car agency.

Finally, a macro GREET could be defined to specify the passenger name.

> Dear NAME:

The entire text of the business letter (source file) would then look like

> GREET 'Ms. Scannel'
> CNFIRM 123, '10:45', 'Ontario', '11:52'
> CAR 'Blotz'
> We trust you will enjoy your flight.
>
> Sincerely,

When this source file is passed through a macro processor, the macro calls are expanded to produce the following letter.

> Dear Ms. Scannel:
>
> Air Flight welcomes you as a passenger. Your flight number 123 leaves at 10:45 and arrives in Ontario at 11:52. Your automobile reservation has been confirmed with Blotz rent-a-car agency.
>
> We trust you will enjoy your flight.
>
> Sincerely,

While this example illustrates the substitution of parameters in a macro, it overlooks the relationship of the macro processor and the assembler. The purpose of the macro processor is to generate source code which is then assembled.

Macros Vs. Subroutines

At this point, you may be wondering how macros differ from subroutines invoked by the CALL instruction. Both aid program structuring and reduce the coding of frequently executed routines.

One distinction between the two is that subroutines necessarily branch to another part of your program while macros generate in-line code. Thus, a program contains only one version of a given subroutine, but contains as many versions of a given macro as there are calls for that macro.

Notice the emphasis on 'versions' in the previous sentence, for this is a major difference between macros and subroutines. A macro does not necessarily generate the same source code each time it is called. By changing the parameters in a macro call, you can change the source code the macro generates. In addition, macro parameters can be tested at assembly-time by the conditional assembly directives. These two tools enable a general-purpose macro definition to generate customized source code for a particular programming situation. Notice that macro expansion and any code customization occur at assembly-time and at the source code level. By contrast, a generalized subroutine resides in your program and requires execution time.

It is usually possible to obtain similar results using either a macro or a subroutine. Determining which of these facilities to use is not always an obvious decision. In some cases, using a single subroutine rather than multiple in-line macros can reduce the overall program size. In situations involving a large number of parameters, the use of macros may be more efficient. Also, notice that macros can call subroutines, and subroutines can contain macros.

USING MACROS

The assembler recognizes the following macro operations:

- MACRO directive
- ENDM directive
- LOCAL directive
- REPT directive
- IRP directive
- IRPC directive
- EXITM directive
- Macro call

All of the directives listed above are related to macro definition. The macro call initiates the parameter substitution (macro expansion) process.

Macro Definition

Macros must be defined in your program before they can be used. A macro definition is initiated by the MACRO assembler directive, which lists the *name* by which the macro can later be called, and the *dummy parameters* to be replaced during macro expansion. The macro definition is terminated by the ENDM directive. The prototype instructions bounded by the MACRO and ENDM directives are called the *macro body*.

When label symbols used in a macro body have 'global' scope, multiply-defined symbol errors result if the macro is called more than once. A label can be given limited scope using the LOCAL directive. This directive assigns a unique value to the symbol each time the macro is called and expanded. Dummy parameters also have limited scope.

Occasionally you may wish to duplicate a block of code several times, either within a macro or in line with other source code. This can be accomplished with minimal coding effort using the REPT (repeat block), IRP (indefinite repeat), and IRPC (indefinite repeat character) directives. Like the MACRO directive, these directives are terminated by ENDM.

The EXITM directive provides an alternate exit from a macro. When encountered, it terminates the current macro just as if ENDM had been encountered.

Macro Definition Directives

MACRO Directive

Label	*Opcode*	*Operand*
name	MACRO	optional dummy parameter(s)

The name in the label field specifies the name of the macro body being defined. Any valid user-defined symbol name can be used as a macro name. Note that this name must be present and must *not* be terminated by a colon.

A dummy parameter can be any valid user-defined symbol name or can be null. When multiple parameters are listed, they must be separated by commas. The scope of a dummy parameter is limited to its specific macro definition. If a reserved symbol is used as a dummy parameter, its reserved value is not recognized. For example, if you code A,B,C as a dummy parameter list, substitutions will occur properly. However, you cannot use the accumulator or the B and C registers within the macro. Because of the limited scope of dummy parameters, the use of these registers is not affected outside the macro definition.

Dummy parameters in a comment are not recognized. No substitution occurs for such parameters.

Dummy parameters may appear in a character string. However, the dummy parameter must be adjacent to an ampersand character (&) as explained later in this chapter.

Any machine instruction or applicable assembler directive can be included in the macro body. The distinguishing feature of macro prototype text is that parts of it can be made variable by placing substitutable dummy parameters in instruction fields. These dummy parameters are the same as the symbols in the operand field of the MACRO directive.

Example:

Define macro MAC1 with dummy parameters G1, G2, and G3.

NOTE

The following macro definition contains a potential error that is clarified in the description of the LOCAL directive later in this chapter.

```
MAC1    MACRO   G1,G2,G3    ;MACRO DIRECTIVE
MOVES:  LHLD    G1          ;MACRO BODY
        MOV     A,M
        LHLD    G2
        MOV     B,M
        LHLD    G3
        MOV     C,M
        ENDM                ;ENDM DIRECTIVE
```

ENDM Directive

Label	*Opcode*	*Operand*
——	ENDM	——

The ENDM directive is required to terminate a macro definition and follows the last prototype instruction. It is also required to terminate code repetition blocks defined by the REPT, IRP, and IRPC directives.

Any data appearing in the label or operand fields of an ENDM directive causes an error.

NOTE

Because nested macro calls are not expanded during macro definition, the ENDM directive to close an outer macro cannot be contained in the expansion of an inner, 'nested' macro call. (See 'Nested Macro Definitions' later in this chapter.)

LOCAL Directive

Label	*Opcode*	*Operand*
——	LOCAL	label name(s)

The specified label names are defined to have meaning only within the current macro expansion. Each time the macro is called and expanded, the assembler assigns each local symbol a unique symbol in the form ??nnnn.

The assembler assigns ??0001 to the first local symbol, ??0002 to the second, and so on. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions. The assembler never duplicates these symbols. The user should avoid coding symbols in the form ??nnnn so that there will not be a conflict with these assembler-generated symbols.

Dummy parameters included in a macro call cannot be operands of a LOCAL directive. The scope of a dummy parameter is always local to its own macro definition.

Local symbols can be defined only within a macro definition. Any number of LOCAL directives may appear in a macro definition, but they must all follow the macro call and must precede the first line of prototype code.

A LOCAL directive appearing outside a macro definition causes an error. Also, a name appearing in the label field of a LOCAL directive causes an error.

Example:

The definition of MAC1 (used as an example in the description of the MACRO directive) contains a potential error because the symbol MOVES has not been declared local. This is a potential error since no error occurs if MAC1 is called only once in the program, and the program itself does not use MOVES as a symbol. However, if MAC1 is called more than once, or if the program uses the symbol MOVES, MOVES is a multiply-defined symbol. This potential error is avoided by naming MOVES in the operand field of a LOCAL directive:

```
MAC1     MACRO    G1,G2,G3
         LOCAL    MOVES
MOVES:   LHLD     G1
         MOV      A,M
         LHLD     G2
         MOV      B,M
         LHLD     G3
         MOV      C,M
         ENDM
```

Assume that MAC1 is the only macro in the program and that it is called twice. The first time MAC1 is expanded, MOVES is replaced with the symbol ??0001; the second time, MOVES is replaced with ??0002. Because the assembler encounters only these special replacement symbols, the program may contain the symbol MOVES without causing a multiple definition.

REPT Directive

Label	*Opcode*	*Operand*
optional:	REPT	expression

The REPT directive causes a sequence of source code lines to be repeated 'expression' times. All lines appearing between the REPT directive and a subsequent ENDM directive constitute the block to be repeated.

When 'expression' contains symbolic names, the assembler must encounter the definition of the symbol prior to encountering the expression.

The insertion of repeat blocks is performed in-line when the assembler encounters the REPT directive. No explicit call is required to cause the code insertion since the definition is an implied call for expansion.

Example 1:

Rotate accumulator right six times.

```
ROTR6:    REPT    6
          RRC
          ENDM
```

Example 2:

The following REPT directive generates the source code for a routine that fills a five-byte field with the character stored in the accumulator:

```
PROGRAM CODE          GENERATED CODING

LHLD    CNTR1         LHLD    CNTR1
REPT    5             MOV     M,A
MOV     M,A           INX     H
INX     H             MOV     M,A
ENDM                  INX     H
                      MOV     M,A
                      INX     H
                      MOV     M,A
                      INX     H
                      MOV     M,A
                      INX     H
```

Example 3:

The following example illustrates the use of REPT to generate a multiplication routine. The multiplication is accomplished through a series of shifts. If this technique is unfamiliar, refer to the example of multiplication in Chapter 6. The example in Chapter 6 uses a program loop for the multiplication. This example replaces the loop with seven repetitions of the four instructions enclosed by the REPT—ENDM directives.

Notice that the expansion specified by this REPT directive causes the label SKIPAD to be generated seven times. Therefore, SKIPAD must be declared local to this macro.

```
FSTMUL:   MVI     D,0       ;FAST MULTIPLY ROUTINE
          LXI     H,0       ;MULTIPLY E*A – 16-BIT RESULT
                            ;IN H&L
          REPT    7
          LOCAL   SKIPAD
          RLC               ;;GET NEXT MULTIPLIER BIT
          JNC     SKIPAD    ;;DON'T ADD IF BIT = 0
          DAD     D         ;;ADD MULTIPLICAND INTO ANSWER
SKIPAD:   DAD     H
          ENDM
          RLC
          RNC
          DAD     D
          RET
```

This example illustrates a classic programming trade-off: speed versus memory. Although this example executes more quickly than the example in Chapter 6, it requires more memory.

IRP Directive

Label	*Opcode*	*Operand*
optional:	IRP	dummy param, <list>

The operand field for the IRP (indefinite repeat) directive must contain one macro dummy parameter followed by a list of actual parameters enclosed in angle brackets. IRP expands its associated macro prototype code substituting the first actual parameter for each occurrence of the dummy parameter. IRP then expands the prototype code again substituting the second actual parameter from the list. This process continues until the list is exhausted.

The list of actual parameters to be substituted for the dummy parameter must be enclosed in angle brackets (< >). Individual items in the list must be separated by commas. The number of actual parameters in the list controls the number of times the macro body is repeated; a list of n items causes n repetitions. An empty list (one with no parameters coded) specifies a null operand list. IRP generates one copy of the macro body substituting a null for each occurrence of the dummy parameter. Also, two commas with no intervening character create a null parameter within the list. (See 'Special Operators' later in this chapter for a description of null operands.)

Example:

The following code sequence gathers bytes of data from different areas of memory and then stores them in consecutive bytes beginning at the address of STORIT:

```
PROGRAM CODE                        GENERATED CODING

LXI     H,STORIT                    LXI     H,STORIT
IRP     X,<FLD1,3E20H,FLD3>         LDA     FLD1
LDA     X                           MOV     M,A
MOV     M,A                         INX     H
INX     H                           LDA     3E20H
ENDM                                MOV     M,A
                                    INX     H
                                    LDA     FLD3
                                    MOV     M,A
                                    INX     H
```

IRPC Directive

Label	*Opcode*	*Operand*
optional:	IRPC	dummy param,text

The IRPC (indefinite repeat character) directive causes a sequence of macro prototype instructions to be repeated for each text character of the actual parameter specified. If the text string is enclosed in optional angle brackets, any delimiters appearing in the text string are treated simply as text to be substituted into the prototype code. The assembler generates one iteration of the prototype code for each character in the text string. For each iteration, the assembler substitutes the next character from the string for each occurrence of the dummy parameter. A list of n text characters generates n repetitions of the IRPC macro body. An empty string specifies a null actual operand. IRPC generates one copy of the macro body substituting a null for each occurrence of the dummy parameter.

Example:

	PROGRAM CODE		*GENERATED CODING*	
	LHLD	DATE-1	LHLD	DATE-1
MVDATE:	IRPC	X,1977	INX	H
	INX	H	MVI	M,1
	MVI	M,X	INX	H
	ENDM		MVI	M,9
			INX	H
			MVI	M,7
			INX	H
			MVI	M,7

IRPC provides the capability to treat each character of a string individually; concatenation (described later in this chapter) provides the capability for building text strings from individual characters.

EXITM Directive

Label	*Opcode*	*Operand*
optional:	EXITM	––

EXITM provides an alternate method for terminating a macro expansion or the repetition of a REPT, IRP, or IRPC code sequence. When EXITM is encountered, the assembler ignores all macro prototype instructions located between the EXITM and ENDM directive for this macro. Notice that EXITM may be used in addition to ENDM, but not in place of ENDM.

When used in nested macros, EXITM causes an exit to the previous level of macro expansion. An EXITM within a REPT, IRP, or IRPC terminates not only the current expansion, but all subsequent iterations as well.

Any data appearing in the operand field of an EXITM directive causes an error.

Example:

EXITM is typically used to suppress unwanted macro expansion. In the following example, macro expansion is terminated when the EXITM directive is assembled because the condition X EQ 0 is true.

```
MAC3    MACRO   X,Y
          .
          .
          .
        IF X EQ 0
        EXITM
          .
          .
          .
        ENDM
```

Special Macro Operators

In certain special cases, the normal rules for dealing with macros do not work. Assume, for example, that you want to specify three actual parameters, and the second parameter happens to be the comma character. To the assembler, the list PARM1,,,PARM3 appears to be a list of four parameters where the second and third parameters are missing. The list can be passed correctly by enclosing the comma in angle brackets: PARM1,<,>,PARM3. These special operators instruct the assembler to accept the enclosed character (the comma) as an actual parameter rather than a delimiter.

The assembler recognizes a number of operators that allow special operations:

& — Ampersand. Used to concatenate (link) text and dummy parameters. See the further discussion of ampersands below.

<> — Angle brackets. Used to delimit text, such as lists, that contain other delimiters. Notice that blanks are usually treated as delimiters. Therefore, when an actual parameter contains blanks (passing the instruction MOV A,M, for example) the parameter must be enclosed in angle brackets. This is also true for any other delimiter that is to be passed as part of an actual parameter. To pass such text to nested macro calls, use one set of angle brackets for each level of nesting. (See 'Nested Macro Definitions,' below.)

;; — Double semicolon. Used before a comment in a macro definition to prevent inclusion of the comment in expansions of the macro and reduce storage requirements. The comment still appears in the listing of the definition.

! — Exclamation point (escape character). Placed before a character (usually a delimiter) to be passed as literalized text in an actual parameter. Used primarily to pass angle brackets as part of an actual parameter. To pass a literalized exclamation point, issue !!. Carriage returns cannot be passed as actual parameters.

The '!' is always preserved while building an actual parameter. It is not echoed when an actual parameter is substituted for a dummy parameter, except when the substitution is being used to build another actual parameter.

NUL In certain cases it is not necessary to pass a parameter to a macro. It is necessary, however, to indicate the omission of the parameter. The omitted (or null) parameter can be represented by two consecutive delimiters as in the list PARM1,,PARM3. A null parameter can also be represented by two consecutive single quotes: '',PARM2,PARM3. Notice that a null is quite different from a blank: a blank is an ASCII character with the hexadecimal representation 20H; a null has no character representation. In the assembly listing a null looks the same as a blank, but that is only because no substitution has taken place. The programmer must decide the meaning of a null parameter. Although the mechanism is somewhat different, the defaults taken for assembler controls provide a good example of what a null parameter can mean. For example, coding MOD85 as an assembler control specifies that the assembler is to generate object code for the 8085. The absence of this control (which in effect is a null parameter) specifies that the assembler is to generate only 8080 object code.

Assembler controls are explained in the *ISIS-II 8080/8085 Macro Assembler Operator's Manual*, 9800292.

Example:

In a macro with the dummy parameters W,X,Y,Z it is acceptable for either the X or Y parameter to be null, but not both. The following IF directive tests for the error condition:

```
IF NUL X&Y
EXITM
```

When a macro is expanded, any ampersand preceding or following a dummy parameter in a macro definition is removed and the substitution of the actual parameter occurs at that point. When it is not adjacent to a dummy parameter, the ampersand is not removed and is passed as part of the macro expansion text.

NOTE

The ampersand must be immediately adjacent to the text being concatenated; intervening blanks are not allowed.

If nested macro definitions (described below) contain ampersands, the only ampersands removed are those adjacent to dummy parameters belonging to the macro definition currently being expanded. All ampersands must be removed by the time the expansion of the encompassing macro body is performed. Exceptions force illegal character errors.

Ampersands placed inside strings are recognized as concatenation delimiters when adjacent to dummy parameters; similarly, dummy parameters within character strings are recognized only when they are adjacent to ampersands. Ampersands are not recognized as operators in comments.

Nested Macro Definitions

A macro definition can be contained completely within the body of another macro definition (that is, macro definitions can be *nested*). The body of a macro consists of all text (including nested macro definitions) bounded by matching MACRO and ENDM directives. The assembler allows any number of macro definitions to be nested.

When a higher-level macro is called for expansion, the next lower-level macro is defined and eligible to be called for expansion. A lower-level macro cannot be called unless all higher-level macro definitions have already been called and expanded.

A new macro may be defined or an existing macro redefined by a nested macro definition depending on whether the name of the nested macro is a new label or has previously been established as a dummy parameter in a higher-level macro definition. Therefore, each time a higher-level macro is called, a lower-level definition can be defined differently if the two contain common dummy parameters. Such redefinition can be costly, however, in terms of assembler execution speed.

Since IRP, IRPC, and REPT blocks constitute macro definitions, they also can be nested within another definition created by IRP, IRPC, REPT, or MACRO directives. In addition, an element in an IRP or IRPC actual parameter list (enclosed in angle brackets) may itself be a list of bracketed parameters; that is, lists of parameters can contain elements that are also lists.

Example:

```
LISTS   MACRO   PARAM1,PARAM2
          .
          .
          .
        ENDM
          .
          .
          .
        LISTS <A, <B,C>>
```

MACRO CALLS

Once a macro has been defined, it can be called any number of times in the program. The call consists of the macro name and any actual parameters that are to replace dummy parameters during macro expansion. During assembly, each macro call is replaced by the macro definition code; dummy parameters are replaced by actual parameters.

Macro Call Format

Label	*Opcode*	*Operand*
optional:	macro name	optional actual parameter(s)

The assembler must encounter the macro definition before the first call for that macro. Otherwise, the macro call is assumed to be an illegal opcode. The assembler inserts the macro body identified by the macro name each time it encounters a call to a previously defined macro in your program.

The positioning of actual parameters in a macro call is critical since the substitution of parameters is based solely on position. The first-listed actual parameter replaces each occurrence of the first-listed dummy parameter; the second actual parameter replaces the second dummy parameter, and so on. When coding a macro call, you must be certain to list actual parameters in the appropriate sequence for the macro.

Notice that blanks are usually treated as delimiters. Therefore, when an actual parameter contains blanks (passing the instruction MOV A,M, for example) the parameter must be enclosed in angle brackets. This is also true for any other delimiter that is to be passed as part of an actual parameter. Carriage returns cannot be passed as actual parameters.

If a macro call specifies more actual parameters than are listed in the macro definition, the extra parameters are ignored. If fewer parameters appear in the call than in the definition, a null replaces each missing parameter.

Example:

The following example shows two calls for the macro LOAD. LOAD is defined as follows:

```
LOAD      MACRO     G1,G2,G3
          LOCAL     MOVES
MOVES:    LHLD      G1
          MOV       A,M
          LHLD      G2
          MOV       B,M
          LHLD      G3
          MOV       C,M
          ENDM
```

LOAD simply loads the accumulator with a byte of data from the location specified by the first actual parameter, the B register with a byte from the second parameter, and the C register with a byte from the third parameter.

The first time LOAD is called, it is used as part of a routine that inverts the order of three bytes in memory. The second time LOAD is called, it is part of a routine that adds the contents of the B register to the accumulator and then compares the result with the contents of the C register.

MAIN PROGRAM

```
JNZ NEXT
LOAD   FLD,FLD+1,FLD+2
MOV    M,A     ;INVERT BYTES
DCX    H
MOV    M,B
DCX    H
MOV    M,C
LOAD   3E0H,BYTE,CHECK
ADD    B       ;CHECK DIGIT
CMP    C
CNZ    DGTBAD
```

SUBSTITUTION

```
         JNZ   NEXT
??0001:  LHLD  FLD
         MOV   A,M
         LHLD  FLD+1
         MOV   B,M
         LHLD  FLD+2
         MOV   C,M
         MOV   M,A     ;INVERT BYTES
         DCX   H
         MOV   M,B
         DCX   H
         MOV   M,C
??0002:  LHLD  3E0H
         MOV   A,M
         LHLD  BYTE
         MOV   B,M
         LHLD  CHECK
         MOV   C,M
         ADD   B       ;CHECK DIGIT
         CMP   C
         CNZ   DGTBAD
```

Nested Macro Calls

Macro calls (including any combination of nested IRP, IRPC, and REPT constructs) can be nested within macro definitions up to eight levels. The macro being called need not be defined when the enclosing macro is defined; however, it must be defined before the enclosing macro is called.

A macro definition can also contain nested calls to itself *(recursive macro calls)* up to eight levels, as long as the recursive macro expansions can be terminated eventually. This operation can be controlled using the conditional assembly directives described in Chapter 4 (IF, ELSE, ENDIF).

Example:

Have a macro call itself five times after it is called from elsewhere in the program.

```
PARAM1   SET      5
RECALL   MACRO
           .
           .
           .
         IF       PARAM1 NE 0
PARAM1   SET      PARAM1-1
         RECALL                 ;RECURSIVE CALL
         ENDIF
           .
           .
           .
         ENDM
```

Macro Expansion

When a macro is called, the actual parameters to be substituted into the prototype code can be passed in one of two modes. Normally, the substitution of actual parameters for dummy parameters is simply a *text* substitution. The parameters are not evaluated until the macro is expanded.

If a percent sign (%) precedes the actual parameter in the macro call, however, the parameter is evaluated immediately, before expansion occurs, and is passed as a decimal number representing the value of the parameter. In the case of IRPC, a '%' preceding the actual parameter causes the entire text string to be treated as a single parameter. One IRPC iteration occurs for each digit in the decimal string passed as the result of immediate evaluation of the text string.

The normal mechanism for passing actual parameters is adequate for most applications. Using the percent sign to pre-evaluate parameters is necessary only when the value of the parameter is different within the local context of the macro definition as compared to its global value outside the macro definition.

Example:

The macro shown in this example generates a number of rotate instructions. The parameters passed in the macro call determine the number of positions the accumulator is to be rotated and whether rotate right or rotate left instructions are to be generated. Some typical calls for this macro are as follows:

```
SHIFTR    'R',3
SHIFTR    L,%COUNT-1
```

The second call shows an expression used as a parameter. This expression is to be evaluated immediately rather than passed simply as text.

The definition of the SHIFTR macro is shown below. This macro uses the conditional IF directive to test the validity of the first parameter. Also, the REPT macro directive is nested within the SHIFTR macro.

```
SHIFTR  MACRO     X,Y
        IF X EQ 'R'
           REPT Y
              RAR
           ENDM
        ENDIF
        IF X NE 'L'
           EXITM
        ELSE
           REPT Y
              RAL
           ENDM
        ENDIF
        ENDM
```

The indentation shown in the definition of the SHIFTR macro graphically illustrates the relationships of the IF, ELSE, ENDIF directives and the REPT, ENDM directives. Such indentation is not required in your program, but may be desirable as documentation.

The SHIFTR macro generates nothing if the first parameter is neither R nor L. Therefore, the following calls produce no code. The result in the object program is as though the SHIFTR macro does not appear in the source program.

```
SHIFTR      5
SHIFTR      'B',2
```

The following call to the SHIFTR macro generates three RAR instructions:

```
SHIFTR      'R',3
```

Assume that a SET directive elsewhere in the source program has given COUNT the value 6. The following call generates five RAL instructions:

```
SHIFTR      'L',%COUNT-1
```

The following is a redefinition of the SHIFTR macro. In this definition, notice that concatenation is used to form the RAR or RAL operation code. If a call to the SHIFTR macro specifies a character other than R or L, illegal operation codes are generated. The assembler flags all illegal operation codes as errors.

```
SHIFTR      MACRO      X,Y
            REPT       Y
            RA&X
            ENDM
            ENDM
```

NULL MACROS

A macro may legally comprise only the MACRO and ENDM directives. Thus, the following is a legal macro definition:

```
NADA        MACRO      P1,P2,P3,P4
            ENDM
```

A call to this macro produces no source code and therefore has no effect on the program.

Although there is no reason to write such a macro, the null (or empty) macro body has a practical application. For example, all the macro prototype instructions might be enclosed with IF-ENDIF conditional directives. When none of the specified conditions is satisfied, all that remains of the macro is the MACRO directive and the ENDM directive.

SAMPLE MACROS

The following sample macros further demonstrate the use of macro directives and operators.

Example 1: Nested IRPC

The following macro definition contains a nested IRPC directive. Notice that the third operand of the outer macro becomes the character string for the IRPC:

```
MOVE    MACRO   X,Y,Z
        IRPC    PARAM,Z
        LHLD    X&&PARAM
        SHLD    Y&&PARAM
        ENDM
        ENDM
```

Assume that the program contains the call MOVE SRC,DST,123. The third parameter of this call is passed to the IRPC. This has the same effect as coding IRPC PARAM,123. When expanded, the MOVE macro generates the following source code:

```
LHLD  SRC1
SHLD  DST1
LHLD  SRC2
SHLD  DST2
LHLD  SRC3
SHLD  DST3
```

Notice the use of concatenation to form labels in this example.

Example 2: Nested Macros Used to Generate DB Directives

This example generates a number of DB 0 directives, each with its own label. Two macros are used for this purpose: INC and BLOCK. The INC macro is defined as follows:

```
    INC      MACRO     F1,F2
$ SAVE  GEN
    F1&F2:   DB        0         ;GENERATE LABELS & DB's
$ RESTORE
             ENDM
```

The BLOCK macro, which accepts the number of DB's to be generated (NUMB) and a label prefix (PREFIX), is defined as follows:

```
    BLOCK    MACRO    NUMB,PREFIX
$   SAVE  NOGEN
    COUNT    SET      0
             REPT     NUMB
    COUNT    SET      COUNT+1
             INC      PREFIX,%COUNT  ;NESTED MACRO CALL
             ENDM
$   RESTORE
             ENDM
```

The macro call BLOCK 3,LAB generates the following source code:

```
          BLOCK     3,LAB
LAB1:     DB        0
LAB2:     DB        0
LAB3:     DB        0
```

The assembler controls specified in these two macros (the lines beginning with $) are used to clean up the assembly listing for easier reading. The source code shown for the call BLOCK 3,LAB is what appears in the assembly listing when the controls are used. Without the controls, the assembly listing appears as follows:

```
          BLOCK     3,LAB
COUNT     SET       0
          REPT      3
COUNT     SET       COUNT+1
          INC       LAB,%COUNT
          ENDM
COUNT     SET       COUNT+1
          INC       LAB,%COUNT
LAB1:     DB        0
COUNT     SET       COUNT+1
          INC       LAB,%COUNT
LAB2:     DB        0
COUNT     SET       COUNT+1
          INC       LAB,%COUNT
LAB3:     DB        0
```

Example 3: A Macro that Converts Itself into a Subroutine

In some cases, the in-line coding substituted for each macro call imposes an unacceptable memory requirement. The next three examples show three different methods for converting a macro call into a subroutine call. The first time the SBMAC macro is called, it generates a full in-line substitution which defines the SUBR subroutine. Each subsequent call to the SBMAC macro generates only a CALL instruction to the SUBR subroutine.

Within the following examples, notice that the label SUBR must be global so that it can be called from outside the first expansion. This is possible only when that part of the macro definition containing the global label is called only once in the entire program.

Method #1: Nested Macro Definitions

Macros can be redefined during the course of a program. In the following example, the definition of SBMAC contains its own redefinition as a nested macro. The first time SBMAC is called, it is full expanded, and the redefinition of SBMAC replaces the original definition. The second time SBMAC is called, only its redefinition (a CALL instruction) is expanded.

```
SBMAC   MACRO
SBMAC   MACRO
        CALL    SUBR    ;;REDEFINITION OF SBMAC
        ENDM
        CALL    SUBR
LINK:   JMP     DUN
SUBR:   .
        .
        .
        RET
DUN:
        ENDM
```

Notice that both versions of SBMAC contain CALL SUBR instructions. This is necessary to provide a return address at the end of the SUBR routine. The jump instruction labelled LINK is required to prevent the SUBR subroutine from executing a return to itself. Notice that the return address for the second CALL SUBR instruction would be SUBR if the jump instruction were omitted. The JMP DUN instruction simply transfers control past the end of the subroutine.

NOTE

The assembler allows the use of a source line consisting only of a label. Such a label is assigned to the next source line for which code or data is generated. Notice that neither code nor data is generated for an ENDM directive, so the label DUN is assigned to whatever instruction follows the ENDM directive. This construct is required because the ENDM directive itself may not be given a label.

Method #2: Conditional Assembly

The second method for altering the expansion of the SBMAC macro uses conditional assembly. In this example, a switch (FIRST) is set TRUE just before the first call for SBMAC. SBMAC is defined as follows:

```
TRUE    EQU     0FFH
FALSE   EQU     0
FIRST   SET     TRUE
SBMAC   MACRO
        CALL SUBR
        IF      FIRST
FIRST   SET     FALSE
LINK:   JMP     DUN
SUBR:   .
        .
        .
        .
        RET
DUN:
        ENDIF
        ENDM
```

The first call to SBMAC expands the full definition, including the call to and definition of SUBR:

```
        SBMAC
        CALL    SUBR
        IF      FIRST
LINK:   JMP     DUN
SUBR:   .
        .
        .
        .
        RET
DUN:
        ENDIF
```

Because FIRST is TRUE when encountered during the first expansion of SBMAC, all the statements between IF and ENDIF are assembled into the program. In subsequent calls, the conditionally-assembled code is skipped so that the subroutine is not regenerated. Only the following expansion is produced:

```
        SBMAC
        CALL    SUBR
        IF      FIRST
```

Method #3: Conditional Assembly with EXITM

The third method for altering the expansion of SBMAC also uses conditional assembly, but uses the EXITM directive to suppress unwanted macro expansion after the first call. EXITM is effective when FIRST is FALSE, which it is after the first call to SBMAC.

```
TRUE    EQU     0FFH
FALSE   EQU     0
FIRST   SET     TRUE
SBMAC   MACRO
        CALL    SUBR
        IF      NOT FIRST
        EXITM
        ENDIF
FIRST   SET     FALSE
        JMP     DUN
SUBR:   .
        .
        .
        .
        RET
DUN:
        ENDM
```

Example 4: Computed GOTO Macro

This sample macro presents an implementation of a computed GOTO for the 8080 or 8085. The computed GOTO, a common feature of many high level languages, allows the program to jump to one of a number of different locations depending on the value of a variable. For example, if the variable has the value zero, the program jumps to the first item in the list; if the variable has the value 3, the program jumps to the fourth address in the list.

In this example, the variable is placed in the accumulator. The list of addresses is defined as a series of DW directives starting at the symbolic address TABLE. This macro (TJUMP) also modifies itself with a nested definition. Therefore, only the first call to the TJUMP macro generates the calculated GOTO routine. Subsequent calls produce only the jump instruction JMP TJCODE.

```
TJUMP       MACRO               ;JUMP TO A-TH ADDR IN TABLE
TJCODE:     ADD     A           ;MULTIPLY A BY 2
            MVI     D,0         ;CLEAR D REG
            MOV     E,A         ;GET TABLE OFFSET INTO D&E
            DAD     D           ;ADD OFFSET TO TABLE ADDR IN H&L
            MOV     E,M         ;GET 1ST ADDRESS BYTE
            INX     H
            MOV     D,M         ;GET 2ND ADDRESS BYTE
            XCHG
            PCHL                ;JUMP TO ADDRESS
TJUMP       MACRO               ;REDEFINE TJUMP TO SAVE CODE
            JMP     TJCODE      ;NEXT CALL JUMPS TO ABOVE CODE
            ENDM
            ENDM
```

Notice that the definition of the TJUMP macro does not account for loading the address of the address table into the H and L registers; the user must load this address just before calling the TJUMP macro. The following shows the coding for the address table (TABLE) and a typical call sequence for the TJUMP macro:

```
            MVI     A,2
            LXI     H,TABLE
            TJUMP
            .
            .
            .
            .
TABLE:      DW      LOC0
            DW      LOC1
            DW      LOC2
```

The call sequence shown above causes a jump to LOC2.

Example 5: Using IRP to Define the Jump Table

The TJUMP macro becomes even more useful when a second macro (GOTO) is used to define the jump table, load the address of the table into the H and L registers, and then call TJUMP. The GOTO macro is defined as follows:

```
GOTO        MACRO     INDEX,LIST
            LOCAL     JTABLE
            LDA       INDEX          ;LOAD ACCUM WITH INDEX
            LXI       H,JTABLE       ;LOAD H&L WITH TABLE ADDRESS
            TJUMP                    ;CALL TJUMP MACRO
JTABLE:     IRP       FORMAL,<LIST>
            DW        FORMAL         ;SET UP TABLE
            ENDM
            ENDM
```

A typical call to the GOTO macro would be as follows:

```
            GOTO      CASE,<COUNT,TIMER,DATE,PTDRVR>
```

This call to the GOTO macro builds a table of DW directives for the labels COUNT, TIMER, DATE, and PTDRVR. It then loads the base address of the table into the H and L registers and calls the TJUMP macro. If the value of the variable CASE is 2 when the GOTO macro is called, the GOTO and TJUMP macros together cause a jump to the address of the DATE routine.

Notice that any number of addresses may be specified in the list for the GOTO routine as long as they all fit on a single source line. Also, the GOTO macro may be called any number of times, but only one copy of the coding for the TJUMP is generated since the TJUMP macro redefines itself to generate only a JMP TJCODE instruction.

6. PROGRAMMING TECHNIQUES

This chapter describes some techniques that may be of help to the programmer.

BRANCH TABLES PSEUDO-SUBROUTINE

Suppose a program consists of several separate routines, any of which may be executed depending upon some initial condition (such as a number passed in a register). One way to code this would be to check each condition sequentially and branch to the routines accordingly as follows:

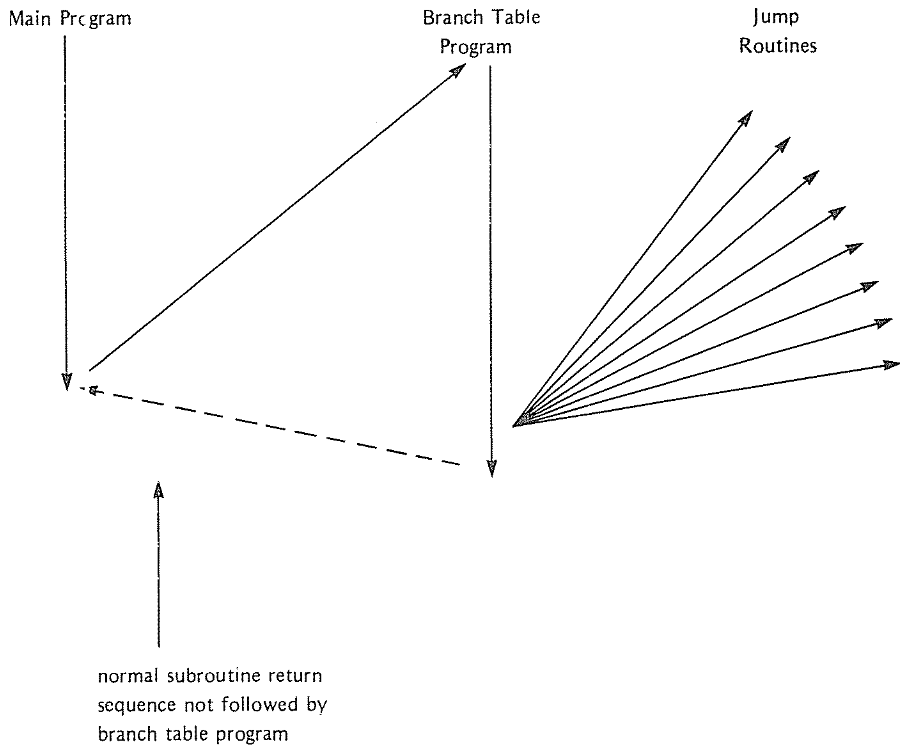
```
CONDITION = CONDITION 1?
IF YES BRANCH TO ROUTINE 1
CONDITION = CONDITION 2?
IF YES BRANCH TO ROUTINE 2
            .
            .
            .
BRANCH TO ROUTINE N
```

A sequence as above is inefficient, and can be improved by using a branch table.

The logic at the beginning of the branch table program loads the starting address of the branch table into the H and L registers. The branch table itself consists of a list of starting addresses for the routines to be branched to. Using the H and L registers as a pointer, the branch table program loads the selected routine's starting address into the program counter, thus effecting a jump to the desired routine. For example, consider a program that executes one of eight routines depending on which bit of the accumulator is set:

```
Jump to routine 1 if the accumulator holds 00000001
 "    "    "    2 "  "      "        "   00000010
 "    "    "    3 "  "      "        "   00000100
 "    "    "    4 "  "      "        "   00001000
 "    "    "    5 "  "      "        "   00010000
 "    "    "    6 "  "      "        "   00100000
 "    "    "    7 "  "      "        "   01000000
 "    "    "    8 "  "      "        "   10000000
```

A program that provides such logic follows. The program is termed a 'pseudo-subroutine' because it is treated as a subroutine by the programmer (i.e., it appears just once in memory), but is entered via a regular JUMP instruction rather than via a CALL instruction.


Main Prcgram
Branch Table
Program
Jump
Routines
normal subroutine return
sequence not followed by
branch table program

```
Label       Code    Operand

START:      LXI     H,BTBL      ;REGISTERS H AND L WILL
                                ;POINT TO BRANCH TABLE
GTBIT:      RAR
            JC      GETAD
            INX     H           ;(H,L)=(H,L)+2 TO
            INX     H           ;POINT TO NEXT ADDRESS
                                ;IN BRANCH TABLE
            JMP     GTBIT
GETAD:      MOV     E,M         ;BIT FOUND
            INX     H           ;LOAD JUMP ADDRESS
                                ;INTO D AND E REGISTERS
            MOV     D,M
            XCHG                ;EXCHANGE D AND E
                                ;WITH H AND L
            PCHL                ;JUMP TO ROUTINE
                                ;ADDRESS
              .
              .
              .
BTBL:       DW      ROUT1       ;BRANCH TABLE. EACH
            DW      ROUT2       ;ENTRY IS A TWO-BYTE
            DW      ROUT3       ;ADDRESS
            DW      ROUT4       ;HELD LEAST SIGNIFICANT
            DW      ROUT5       ;BYTE FIRST
            DW      ROUT6
            DW      ROUT7
            DW      ROUT8
```

The control routine at START uses the H and L registers as a pointer into the branch table (BTBL) corresponding to the bit of the accumulator that is set. The routine at GETAD then transfers the address held in the corresponding branch table entry to the H and L registers via the D and E registers, and then uses a PCHL instruction, thus transferring control to the selected routine.

TRANSFERRING DATA TO SUBROUTINES

A subroutine typically requires data to perform its operations. In the simplest case, this data may be transferred in one or more registers.

Sometimes it is more convenient and economical to let the subroutine load its own registers. One way to do this is to place a list of the required data (called a parameter list) in some data area of memory, and pass the address of this list to the subroutine in the H and L registers.

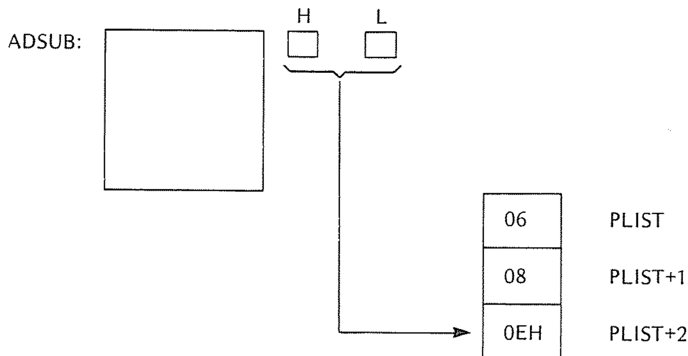
For example, the subroutine ADSUB expects the address of a three-byte parameter list in the H and L registers. It adds the first and second bytes of the list, and stores the result in the third byte of the list:

```
Label       Code     Operand     Comment

            LXI      H,PLIST     ;LOAD H AND L WITH
                                 ;ADDRESSES OF THE PARAM-
                                 ;ETER LIST
            CALL     ADSUB       ;CALL THE SUBROUTINE
RET1:         .
              .
PLIST:      DB       6           ;FIRST NUMBER TO BE ADDED
            DB       8           ;SECOND NUMBER TO BE
                                 ;ADDED
            DS       1           ;RESULT WILL BE STORED HERE
            LXI      H,LIST2     ;LOAD H AND L REGISTERS
            CALL     ADSUB       ;FOR ANOTHER CALL TO ADSUB
RET2:         .
              .
LIST2:      DB       10
            DB       35
            DS       1
              .
              .
ADSUB:      MOV      A,M         ;GET FIRST PARAMETER
            INX      H           ;INCREMENT MEMORY
                                 ;ADDRESS
            MOV      B,M         ;GET SECOND PARAMETER
            ADD      B           ;ADD FIRST TO SECOND
            INX      H           ;INCREMENT MEMORY
                                 ;ADDRESS
            MOV      M,A         ;STORE RESULT AT THIRD
                                 ;PARAMETER STORE
            RET                  ;RETURN UNCONDITIONALLY
```

The first time ADSUB is called, it loads the A and B registers from PLIST and PLIST+1 respectively, adds them, and stores the result in PLIST+2. Return is then made to the instruction at RET1.

First call to ADSUB:



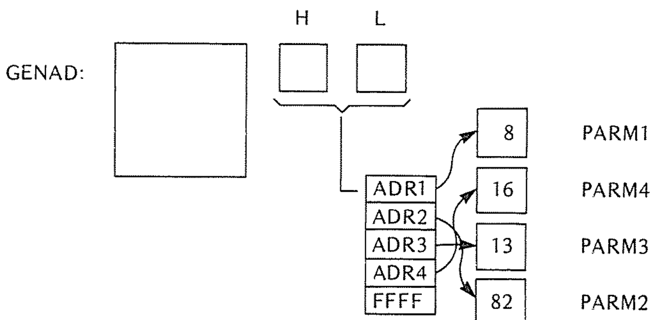
The second time ADSUB is called, the H and L registers point to the parameter list LIST2. The A and B registers are loaded with 10 and 35 respectively, and the sum is stored at LIST2+2. Return is then made to the instruction at RET2.

Note that the parameter lists PLIST and LIST2 could appear anywhere in memory without altering the results produced by ADSUB.

This approach does have its limitations, however. As coded, ADSUB must receive a list of two and only two numbers to be added, and they must be contiguous in memory. Suppose we wanted a subroutine (GENAD) which would add an arbitrary number of bytes, located anywhere in memory, and leave the sum in the accumulator.

This can be done by passing the subroutine a parameter list which is a list of *addresses* of parameters, rather than the parameters themselves, and signifying the end of the parameter list be a number whose first byte is FFH (assuming that no parameters will be stored above address FF00H).

Call to GENAD:



As implemented below, GENAD saves the current sum (beginning with zero) in the C register. It then loads the address of the first parameter into the D and E registers. If this address is greater than or equal to FF00H, it reloads the accumulator with the sum held in the C register and returns to the calling routine. Otherwise, it

loads the parameter into the accumulator and adds the sum in the C register to the accumulator. The routine then loops back to pick up the remaining parameters.

```
Label       Code    Operand     Comment

            LXI     H,PLIST     ;LOAD ADDRESS OF
            CALL    GENAD       ;PARAMETER ADDRESS LIST
              .
              .
            HALT
PLIST:      DW      PARM1       ;LIST OF PARAMETER ADDRESSES
            DW      PARM2
            DW      PARM3
            DW      PARM4
            DW      0FFFFH      ;TERMINATOR
              .
              .
PARM1:      DB      6
PARM4:      DB      16
              .
              .
PARM3:      DB      13
              .
              .
              .
PARM2:      DB      82
              .
              .
GENAD:      XRA     A           ;CLEAR ACCUMULATOR
LOOP:       MOV     C,A         ;SAVE CURRENT TOTAL IN C
            MOV     E,M         ;GET LOW ORDER ADDRESS BYTE
                                ;OF FIRST PARAMETER
            INX     H
            MOV     A,M         ;GET HIGH ORDER ADDRESS BYTE
                                ;OF FIRST PARAMETER
            CPI     0FFH        ;COMPARE TO FFH
            JZ      BACK        ;IF EQUAL, ROUTINE IS COMPLETE
            MOV     D,A         ;D AND E NOW ADDRESS PARAMETER
            LDAX    D           ;LOAD ACCUMULATOR WITH PARAMETER
            ADD     C           ;ADD PREVIOUS TOTAL
            INX     H           ;INCREMENT H AND L TO POINT
                                ;TO NEXT PARAMETER ADDRESS
            JMP     LOOP        ;GET NEXT PARAMETER
BACK:       MOV     A,C         ;ROUTINE DONE - RESTORE TOTAL
            RET                 ;RETURN TO CALLING ROUTINE
            END
```

Note that GENAD could add any combination of the parameters with no change to the parameters themselves.

The sequence:

```
               LXI        H,PLIST
               CALL       GENAD
                .
                .
                .
PLIST:         DW         PARM4
               DW         PARM1
               DW         0FFFFH
```

would cause PARM1 and PARM4 to be added, no matter where in memory they might be located (excluding addresses above FF00H).

Many variations of parameter passing are possible. For example, if it is necessary to allow parameters to be stored at any address, a calling program can pass the total number of parameters as the first parameter; the subroutine then loads this first parameter into a register and uses it as a counter to determine when all parameters had been accepted.

SOFTWARE MULTIPLY AND DIVIDE

The multiplication of two unsigned 8-bit data bytes may be accomplished by one of two techniques: repetitive addition, or use of a register shifting operation.

Repetitive addition provides the simplest, but slowest, form of multiplication. For example, 2AH*74H may be generated by adding 74H to the (initially zeroed) accumulator 2AH times.

Shift operations provide faster multiplication. Shifting a byte left one bit is equivalent to multiplying by 2, and shifting a byte right one bit is equivalent to dividing by 2. The following process will produce the correct 2-byte result of multiplying a one byte multiplicand by a one byte multiplier:

A. Test the least significant bit of multiplier. If zero, go to step b. If one, add the multiplicand to the *most* significant byte of the result.

B. Shift the entire two-byte result right one bit position.

C. Repeat steps a and b until all 8 bits of the multiplier have been tested.

For example, consider the multiplication: 2AH*3CH=9D8H

Step 1: Test multiplier 0-bit; it is 0, so shift 16-bit result right one bit.

Step 2: Test multiplier 1-bit; it is 0, so shift 16-bit result right one bit.

Step 3: Test multiplier 2-bit; it is 1, so add 2AH to high-order byte of result and shift 16-bit result right one bit.

Step 4: Test multiplier 3-bit; it is 1, so add 2AH to high-order byte of result and shift 16-bit result right one bit.

Step 5: Test multiplier 4-bit; it is 1, so add 2AH to high-order byte of result and shift 16-bit result right one bit.

Step 6: Test multiplier 5-bit; it is 1, so add 2AH to high-order byte of result and shift 16-bit result right one bit.

Step 7: Test multiplier 6-bit; it is 0, so shift 16-bit result right one bit.

Step 8: Test multiplier 7-bit; it is 0, so shift 16-bit result right one bit.

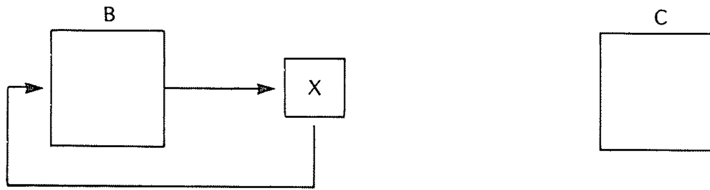
The result produced is 09D8.

	MULTIPLIER	MULTIPLICAND	HIGH-ORDER BYTE OF RESULT	LOW-ORDER BYTE OF RESULT
Start	00111100(3C)	00101010(2A)	00000000	00000000
Step 1 a				
b			00000000	00000000
Step 2 a				
b			00000000	00000000
Step 3 a			00101010	00000000
b			00010101	00000000
Step 4 a			00111111	00000000
b			00011111	10000000
Step 5 a			01001001	10000000
b			00100100	11000000
Step 6 a			01001110	11000000
b			00100111	01100000
Step 7 a				
b			00010011	10110000
Step 8 a				
b			00001001	11011000(9D8)

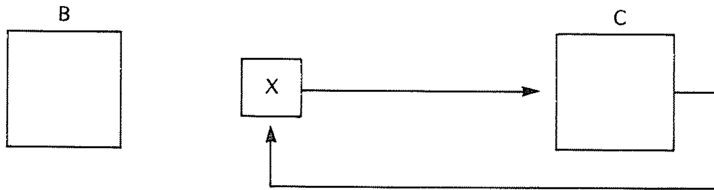
Since the multiplication routine described above uses a number of important programming techniques, a sample program is given with comments.

The program uses the B register to hold the most significant byte of the result, and the C register to hold the least significant byte of the result. The 16-bit right shift of the result is performed in the accumulator by two rotate-right-through-carry instructions.

Zero carry and then rotate B:



Then rotate C to complete the shift:



Register D holds the multiplicand, and register C originally holds the multiplier.

```
MULT:    MVI    B,0      ;INITIALIZE MOST SIGNIFICANT BYTE
                         ;OF RESULT
         MVI    E,9      ;BIT COUNTER
MULT0:   MOV    A,C      ;ROTATE LEAST SIGNIFICANT BIT OF
         RAR             ;MULTIPLIER TO CARRY AND SHIFT
         MOV    C,A      ;LOW-ORDER BYTE OF RESULT
         DCR    E
         JZ     DONE     ;EXIT IF COMPLETE
         MOV    A,B
         JNC    MULT1
         ADD    D        ;ADD MULTIPLICAND TO HIGH-
                         ;ORDER BYTE OF RESULT IF BIT
                         ;WAS A ONE
MULT1:   RAR             ;CARRY=0 HERE SHIFT HIGH-
                         ;ORDER BYTE OF RESULT
         MOV    B,A
         JMP    MULT0
DONE:
```

An analogous procedure is used to divide an unsigned 16-bit number by an unsigned 16-bit number. Here, the process involves subtraction rather than addition, and rotate-left instructions instead of rotate-right instructions.

The followin; reentrant program uses the B and C registers to hold the dividend and quotient, and the D and E register to hcld the divisor and remainder. The H and L registers are used to store data temporarily.

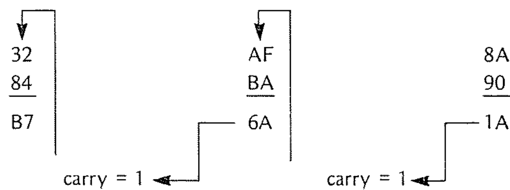
```
DIV:    MOV     A,D     ;NEGATE THE DIVISOR
        CMA
        MOV     D,A
        MOV     A,E
        CMA
        MOV     E,A
        INX     D       ;FOR TWO'S COMPLEMENT
        LXI     H,0     ;INITIAL VALUE FOR REMAINDER
        MVI     A,17    ;INITIALIZE LOOP COUNTER
DV0:    PUSH    H       ;SAVE REMAINDER
        DAD     D       ;SUBTRACT DIVISOR (ADD NEGATIVE)
        JNC     DV1     ;UNDER FLOW, RESTORE HL
        XTHL
DV1:    POP     H
        PUSH    PSW     ;SAVE LOOP COUNTER (A)
        MOV     A,C     ;4 REGISTER LEFT SHIFT
        RAL             ;WITH CARRY
        MOV     C,A     ;CY->C->B->L->H
        MOV     A,B
        RAL
        MOV     B,A
        MOV     A,L
        RAL
        MOV     L,A
        MOV     A,H
        RAL
        MOV     H,A
        POP     PSW     ;RESTORE LOOP COUNTER (A)
        DCR     A       ;DECREMENT IT
        JNZ     DV0     ;KEEP LOOPING
;
;POST-DIVIDE CLEAN UP
;SHIFT REMAINDER RIGHT AND RETURN IN DE
;
        ORA     A
        MOV     A,H
        RAR
        MOV     D,A
        MOV     A,L
        RAR
        MOV     E,A
        RET
        END
```

MULTIBYTE ADDITION AND SUBTRACTION

The carry flag and the ADC (add with carry) instructions may be used to add unsigned data quantities of arbitrary length. Consider the following addition of two three-byte unsigned hexadecimal numbers:

```
 32AF8A
+84BA90
 ------
 B76A1A
```

To perform this addition, add to the low-order byte using an ADD instruction. ADD sets the carry flag for use in subsequent instructions, but does not include the carry flag in the addition. Then use ADC to add to all higher order bytes.

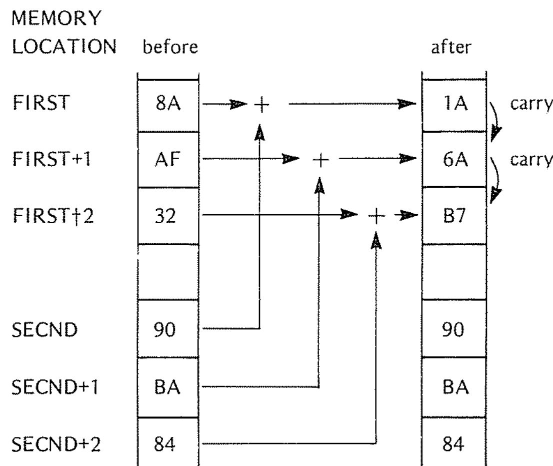


The following routine will perform this multibyte addition, making these assumptions:

The E register holds the length of each number to be added (in this case, 3).

The numbers to be added are stored from low-order byte to high-order byte beginning at memory locations FIRST and SECND, respectively.

The result will be stored from low-order byte to high-order byte beginning at memory location FIRST, replacing the original contents of these locations.



The following routine uses an ADC instruction to add the low-order bytes of the operands. This could cause the result to be high by one if the carry flag were left set by some previous instruction. This routine avoids the problem by clearing the carry flag with the XRA instruction just before LOOP.

```
Label      Code      Operand     Comment

MADD:      LXI       B,FIRST     ;B AND C ADDRESS FIRST
           LXI       H,SECND     ;H AND L ADDRESS SECND
           XRA       A           ;CLEAR CARRY FLAG
LOOP:      LDAX      B           ;LOAD BYTE OF FIRST
           ADC       M           ;ADD BYTE OF SECND
                                 ;WITH CARRY
           STAX      B           ;STORE RESULT AT FIRST
           DCR       E           ;DONE IF E = 0
           JZ        DONE
           INX       B           ;POINT TO NEXT BYTE OF
                                 ;FIRST
           INX       H           ;POINT TO NEXT BYTE OF
                                 ;SECND
           JMP       LOOP        ;ADD NEXT TWO BYTES
DONE:      .
           .
FIRST:     DB        90H
           DB        0BAH
           DB        84H
SECND:     DB        8AH
           DB        0AFH
           DB        32H
```

Since none of the instructions in the program loop affect the carry flag except ADC, the addition with carry will proceed correctly.

When location DONE is reached, bytes FIRST through FIRST+2 will contain 1A6AB7, which is the sum shown at the beginning of this section arranged from low-order to high-order byte.

In order to create a multibyte subtraction routine, it is necessary only to duplicate the multibyte addition routine of this section, changing the ADC instruction to an SBB instruction. The program will then subtract the number beginning at SECND from the number beginning at FIRST, placing the result at FIRST.

DECIMAL ADDITION

Any 4-bit data quantity may be treated as a decimal number as long as it represents one of the decimal digits from 0 through 9, and does not contain any of the bit patterns representing the hexadecimal digits A through F. In order to preserve this decimal interpretation when performing addition, the value 6 must be added to the 4-bit quantity whenever the addition produces a result between 10 and 15. This is because each 4-bit data quantity can hold 6 more combinations of bits than there are decimal digits.

Decimal addition is performed by letting each 8-bit byte represent two 4-bit decimal digits. The bytes are summed in the accumulator in standard fashion, and the DAA (decimal adjust accumulator) instruction is then used to convert the 8-bit binary result to the correct representation of 2 decimal digits. For multibyte strings, you must perform the decimal adjust before adding the next higher-order bytes. This is because you need the carry flag setting from the DAA instruction for adding the higher-order bytes.

To perform the decimal addition:

```
 2985
+4936
 ----
 7921
```

the process works as follows:

1. Clear the Carry and add the two lowest-order digits of each number (remember that each 2 decimal digits are represented by one byte).

```
                 85 = 10000101B
                 36 = 00110110B
        carry       =        0
                     ---------
                  [0]10111011B
Carry = 0 ------^      ^------ Auxiliary Carry = 0
```

   The accumulator now contains 0BBH.

2. Perform a DAA operation. Since the rightmost four bits are greater than 9, a 6 is added to the accumulator.

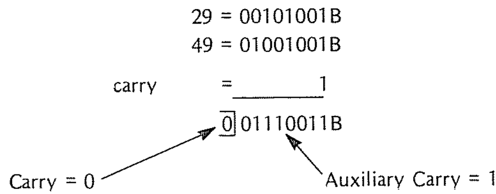
```
   Accumulator = 10111011B
             6 =     0110B
                 ---------
                 11000001B
```

   Since the leftmost bits are greater than 9, a 6 is added to these bits, thus setting the carry flag.

```
   Accumulator = 11000001B
             6 = 0110    B
                 ---------
              [1]00100001B
Carry flag = 1 --^
```

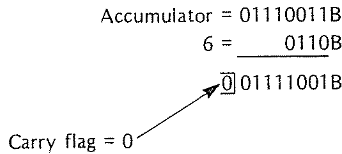
   The accumulator now contains 21H. Store these two digits.

3. Add the next group of two digits:



The accumulator now contains 73H.

4. Perform a DAA operation. Since the auxiliary carry flag is set, 6 is added to the accumulator.



Since the leftmost 4 bits are less than 10 and the carry flag is reset, no further action occurs.

Thus, the correct decimal result 7921 is generated in two bytes.

A routine which adds decimal numbers, then, is exactly analogous to the multibyte addition routine MADD of the last section, and may be produced by inserting the instruction DAA after the ADC M instruction of that example.

Each iteration of the program loop will add two decimal digits (one byte) of the numbers.

DECIMAL SUBTRACTION

Decimal subtraction is considerably more complicated than decimal addition. In general, the process consists of generating the tens complement of the subtrahend digit, and then adding the result to the minuend digit. For example, to subtract 34 from 56, form the tens complement of 34 (99-34=65+1=66). Then, 56+66=122. By truncating off the carry out of the high order digit, we get 22, the correct result.

The problem of handling borrows arises in multibyte decimal subtractions. When no borrow occurs from a subtract, you want to use the tens complement of the subtrahend for the next operation. If a borrow does occur, you want to use the nines complement of the subtrahend.

Notice that the meaning of the carry flag is inverted because you are dealing with complemented data. Thus, a one bit in the carry flag indicates no borrow; a zero bit in the carry flag indicates a borrow. This inverted carry flag setting can be used in an add operation to form either the nines or tens complement of the subtrahend.

The detailed procedure for subtracting multi-digit decimal numbers is as follows:

1. Set the carry flag = 1 to indicate no borrow.

2. Load the accumulator with 99H, representing the number 99 decimal.

3. Add zero to the accumulator with carry, producing either 99H or 9AH, and resetting the carry flag.

4. Subtract the subtrahend digits from the accumulator, producing either the nines or tens complement.

5. Add the minuend digits to the accumulator.

6. Use the DAA instruction to make sure the result in the accumulator is in decimal format, and to indicate a borrow in the carry flag if one occurred.

7. If there are more digits to subtract, go to step 2. Otherwise, stop.

Example:

Perform the decimal subtraction:

```
  4358D
 -1362D
 ------
  2996D
```

1. Set carry = 1.

2. Load accumulator with 99H.

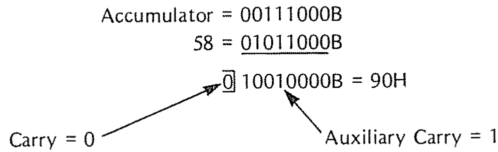
3. Add zero with carry to the accumulator, producing 9AH.

```
Accumulator = 10011001B
            = 00000000B
Carry       = _______1
              10011010B = 9AH
```

4. Subtract the subtrahend digits 62 from the accumulator.

```
Accumulator = 10011010B
       ‾62‾ = 10011110B
             -----------
            [1]00111000B
```

5. Add the minuend digits 58 to the accumulator.

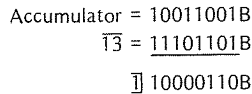


6. DAA converts accumulator to 96 (since Auxiliary Carry = 1) and leaves carry flag = 0 indicating that a borrow occurred.

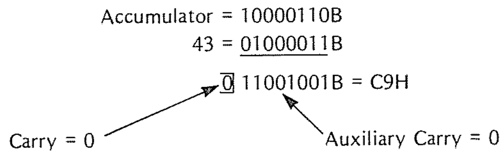
7. Load accumulator with 99H.

8. Add zero with carry to accumulator, leaving accumulator = 99H.

9. Subtract the subtrahend digits 13 from the accumulator.



10. Add the minuend digits 43 to the accumulator.



11 DAA converts accumulator to 29 and sets the carry flag = 1, indicating no borrow occurred.

Therefore, the result of subtracting 1362 from 4358 is 2996.

The following subroutine will subtract one 16-digit decimal number from another using the following assumptions:

The minuend is stored least significant (2) digits first beginning at location MINU.

The subtrahend is stored least significant (2) digits first beginning at location SBTRA.

The result will be stored least significant (2) digits first, replacing the minuend.

```
Label     Code    Operand    Comment

DSUB:     LXI     D,MINU     ;D AND E ADDRESS MINUEND
          LXI     H,SBTRA    ;H AND L ADDRESS SUBTRA-
                             ;HEND
          MVI     C,8        ;EACH LOOP SUBTRACTS 2
                             ;DIGITS (ONE BYTE),
                             ;THEREFORE PROGRAM WILL
                             ;SUBTRACT 16 DIGITS.
          STC                ;SET CARRY INDICATING
                             ;NO BORROW
LOOP:     MVI     A,99H      ;LOAD ACCUMULATOR
                             ;WITH 99H.
          ACI     0          ;ADD ZERO WITH CARRY
          SUB     M          ;PRODUCE COMPLEMENT
                             ;OF SUBTRAHEND
          XCHG               ;SWITCH D AND E WITH
                             ;H AND L
          ADD     M          ;ADD MINUEND
          DAA                ;DECIMAL ADJUST
                             ;ACCUMULATOR
          MOV     M,A        ;STORE RESULT
          XCHG               ;RESWITCH D AND E
                             ;WITH H AND L
          DCR     C          ;DONE IF C = 0
          JZ      DONE
          INX     D          ;ADDRESS NEXT BYTE
                             ;OF MINUEND
          INX     H          ;ADDRESS NEXT BYTE
                             ;OF SUBTRAHEND
          JMP     LOOP       ;GET NEXT 2 DECIMAL DIGITS
DONE:     NOP
```

7. INTERRUPTS

INTERRUPT CONCEPTS

The following is a general description of interrupt handling and applies to both the 8080 and 8085 processors. However, the 8085 processor has some additional hardware features for interrupt handling. For more information on these features, see the description of the 8085 processor in Chapter 1 and the descriptions of the RIM, SIM, and RST instructions in Chapter 3.

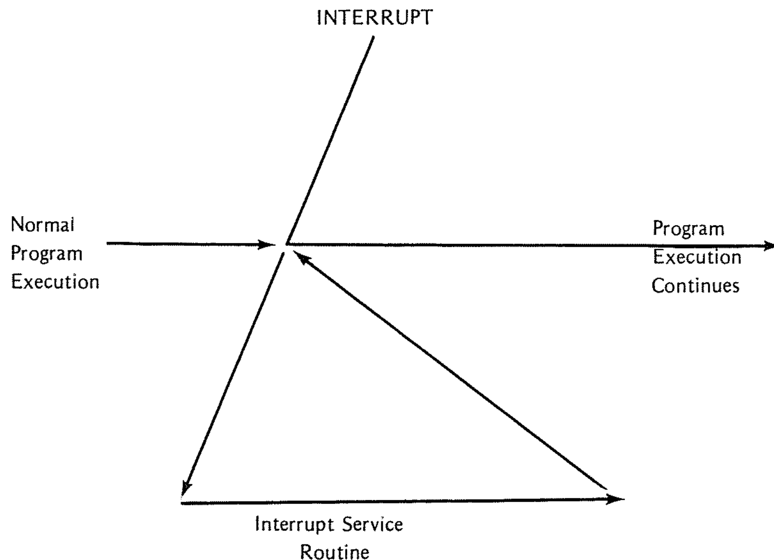
Often, events occur external to the central processing unit which require immediate action by the CPU. For example, suppose a device is sending a string of 80 characters to the CPU, one at a time, at fixed intervals. There are two ways to handle such a situation:

A. A program could be written which accepts the first character, waits until the next character is ready (e.g., executes a timeout by incrementing a sufficiently large counter), then accepts the next character, and proceeds in this fashion until the entire 80 character string has been received.

   This method is referred to as programmed Input/Output.

B. The device controller could interrupt the CPU when a character is ready to be input, forcing a branch from the executing program to a special interrupt service routine.

   The interrupt sequence may be illustrated as follows:



The 8080 contains a bit named INTE which may be set or reset by the instructions EI and DI described in Chapter 3. Whenever INTE is equal to 0, the entire interrupt handling system is disabled, and no interrupts will be accepted.

When the 8080 recognizes an interrupt request from an external device, the following actions occur:

1. The instruction currently being executed is completed.
2. The interrupt enable bit, INTE, is reset = 0.
3. The interrupting device supplies, via hardware, one instruction which the CPU executes. This instruction does not appear anywhere in memory, and the programmer has no control over it, since it is a function of the interrupting device's controller design. The program counter is not incremented before this instruction.

The instruction supplied by the interrupting device is normally an RST instruction (see Chapter 3), since this is an efficient one byte call to one of 8 eight-byte subroutines located in the first 64 words of memory. For instance, the device may supply the instruction:

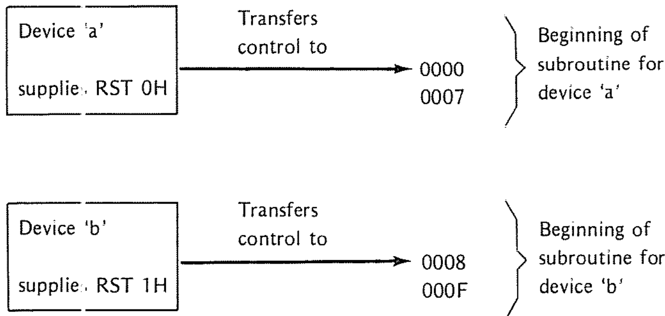
RST 0H

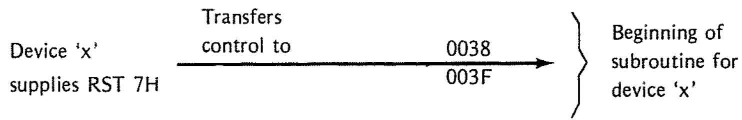
with each input interrupt. Then the subroutine which processes data transmitted from the device to the CPU will be called into execution via an eight-byte instruction sequence at memory locations 0000H to 0007H.

A digital input device may supply the instruction:

RST 1H

Then the subroutine that processes the digital input signals will be called via a sequence of instructions occupying memory locations 0008H to 000FH.

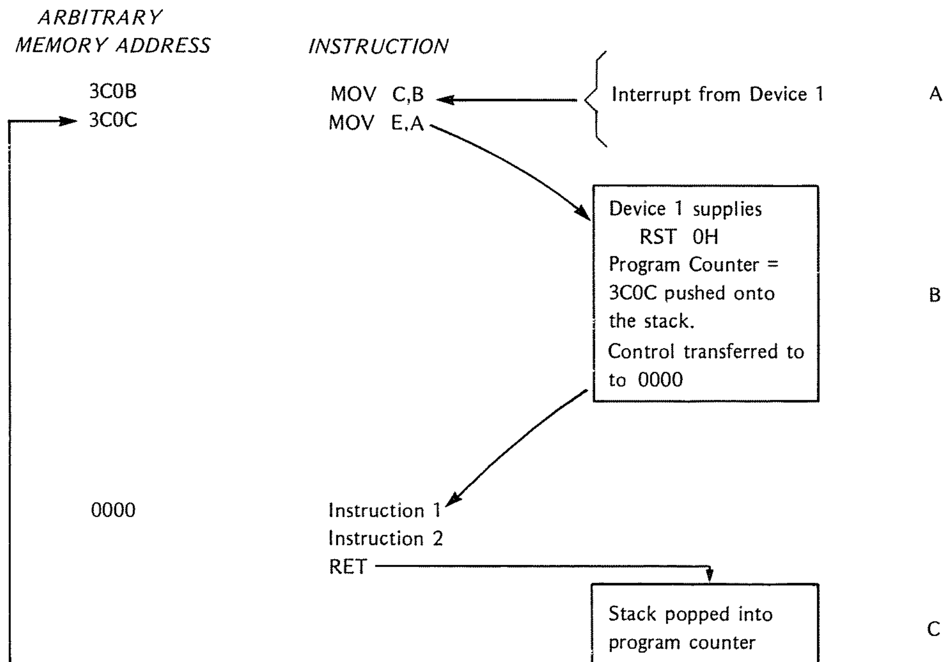




Note that any of these 8-byte subroutines may in turn call longer subroutines to process the interrupt, if necessary.

Any device may supply an RST instruction (and indeed may supply any one-byte 8080 instruction).

The following is an example of an Interrupt sequence:



Device 1 signals an interrupt as the CPU is executing the instruction at 3C0B. This instruction is completed. The program counter remains set to 3C0C, and the instruction RST 0H supplied by device 1 is executed. Since this is a call to location zero, 3C0C is pushed onto the stack and program control is transferred to location 0000H. (This subroutine may perform jumps, calls, or any other operation.) When the RETURN is executed, address 3C0C is popped off the stack and replaces the contents of the program counter, causing execution to continue at this point.

WRITING INTERRUPT SUBROUTINES

In general, any registers or condition bits changed by an interrupt subroutine must be restored before returning to the interrupted program, or errors will occur.

For example, suppose a program is interrupted just prior to the instruction:

JC LOC

and the carry bit equals 1. If the interrupt subroutine happens to reset the carry bit before returning to the interrupted program, the jump to LOC which should have occurred will not, causing the interrupted program to produce erroneous results.

Like any other subroutine then, any interrupt subroutine should save at least the condition bits and restore them before performing a RETURN operation. (The obvious and most convenient way to do this is to save the data in the stack, using PUSH and POP operations.)

Further, the interrupt enable system is automatically disabled whenever an interrupt is acknowledged. Except in special cases, therefore, an interrupt subroutine should include an EI instruction somewhere to permit detection and handling of future interrupts. One instruction after an EI is executed, the interrupt subroutine may itself be interrupted. This process may continue to any level, but as long as all pertinent data are saved and restored, correct program execution will continue automatically.

A typical interrupt subroutine, then, could appear as follows:

```
Code      Operand     Comment

PUSH      PSW         ;SAVE CONDITION BITS AND ACCUMULATOR
EI                    ;RE-ENABLE INTERRUPTS
.                     ;
.                     ;PERFORM NECESSARY ACTIONS TO SERVICE
.                     ;THE INTERRUPT
.
.
POP       PSW         ;RESTORE MACHINE STATUS
RET                   ;RETURN TO INTERRUPTED PROGRAM
```

APPENDIX A. INSTRUCTION SUMMARY

This appendix summarizes the bit patterns and number of time states associated with every 8080 CPU instruction. The instructions are listed in both mnemonic (alphabetical) and operation code (numerical) sequence.

When using this summary, note the following symbology.

DDD represents a destination register. SSS represents a source register. Both DDD and SSS are interpreted as follows:

DDD or SSS	Interpretation
000	Register B
001	Register C
010	Register D
011	Register E
100	Register H
101	Register L
110	A memory register or stack pointer or PSW (flags + accumulator)
111	The accumulator

Instruction execution time equals number of time periods multiplied by the duration of a time period.

A time period may vary from 480 nanoseconds to 2 microseconds on the 8080 or 320 nanoseconds to 2 microseconds on the 8085. Where two numbers of time periods are shown (eq.5/11), it means that the smaller number of time periods is required if a condition is not met, and the larger number of time periods is required if the condition is met.

MNEMONIC	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	NUMBER OF TIME PERIODS	
									8080	8085
CALL	1	1	0	0	1	1	0	1	17	18
CC	1	1	0	1	1	1	0	0	11/17	9/18
CNC	1	1	0	1	0	1	0	0	11/17	9/18
CZ	1	1	0	0	1	1	0	0	11/17	9/18
CNZ	1	1	0	0	0	1	0	0	11/17	9/18
CP	1	1	1	1	0	1	0	0	11/17	9/18
CM	1	1	1	1	1	1	0	0	11/17	9/18
CPE	1	1	1	0	1	1	0	0	11/17	9/17
CPO	1	1	1	0	0	1	0	0	11/17	9/18
RET	1	1	0	0	1	0	0	1	10	10
RC	1	1	0	1	1	0	0	0	5/11	6/12
RNC	1	1	0	1	0	0	0	0	5/11	6/12
RZ	1	1	0	0	1	0	0	0	5/11	6/12

ALL MNEMONICS© 1974, 1975, 1976, 1977 INTEL CORPORATION

MNEMONIC	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	NUMBER OF TIME PERIODS 8080	8085
RNZ	1	1	0	0	0	0	0	0	5/11	6/12
RP	1	1	1	1	0	0	0	0	5/11	6/12
RM	1	1	1	1	1	0	0	0	5/11	6/12
RPE	1	1	1	0	1	0	0	0	5/11	6/12
RPO	1	1	1	0	0	0	0	0	5/11	6/12
RST	1	1	A	A	A	1	1	1	11	12
IN	1	1	0	1	1	0	1	1	10	10
OUT	1	1	0	1	0	0	1	1	10	10
LXI B	0	0	0	0	0	0	0	1	10	10
LXI D	0	0	0	1	0	0	0	1	10	10
LXI H	0	0	1	0	0	0	0	1	10	10
LXI SP	0	0	1	1	0	0	0	1	10	10
PUSH B	1	1	0	0	0	1	0	1	11	12
PUSH D	1	1	0	1	0	1	0	1	11	12
PUSH H	1	1	1	0	0	1	0	1	11	12
PUSH PSW	1	1	1	1	0	1	0	1	11	12
POP B	1	1	0	0	0	0	0	1	10	10
POP D	1	1	0	1	0	0	0	1	10	10
POP H	1	1	1	0	0	0	0	1	10	10
POP PSW	1	1	1	1	0	0	0	1	10	10
STA	0	0	1	1	0	0	1	0	13	13
LDA	0	0	1	1	1	0	1	0	13	13
XCHG	1	1	1	0	1	0	1	1	4	4
XTHL	1	1	1	0	0	0	1	1	18	16
SPHL	1	1	1	1	1	0	0	1	5	6
PCHL	1	1	1	0	1	0	0	1	5	6
DAD B	0	0	0	0	1	0	0	1	10	10
DAD D	0	0	0	1	1	0	0	1	10	10
DAD H	0	0	1	0	1	0	0	1	10	10
DAD SP	0	0	1	1	1	0	0	1	10	10
STAX B	0	0	0	0	0	0	1	0	7	7
STAX D	0	0	0	1	0	0	1	0	7	7
LDAX B	0	0	0	0	1	0	1	0	7	7
LDAX D	0	0	0	1	1	0	1	0	7	7
INX B	0	0	0	0	0	0	1	1	5	6
INX D	0	0	0	1	0	0	1	1	5	6
INX H	0	0	1	0	0	0	1	1	5	6
INX SP	0	0	1	1	0	0	1	1	5	6
MOV r_1,r_2	0	1	D	D	D	S	S	S	5	4
MOV M,r	0	1	1	1	0	S	S	S	7	7
MOV r,M	0	1	D	D	D	1	1	0	7	7
HLT	0	1	1	1	0	1	1	0	7	5
MVI r	0	0	D	D	D	1	1	0	7	7
MVI M	0	0	1	1	0	1	1	0	10	10
INR	0	0	D	D	D	1	0	0	5	4
DCR	0	0	D	D	D	1	0	1	5	4

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

MNEMONIC	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	NUMBER OF TIME PERIODS	
									8080	8085
INR A	0	0	1	1	1	1	0	0	5	4
DCR A	0	0	1	1	1	1	0	1	5	4
INR M	0	0	1	1	0	1	0	0	10	10
DCR M	0	0	1	1	0	1	0	1	10	10
ADD r	1	0	0	0	0	S	S	S	4	4
ADC r	1	0	0	0	1	S	S	S	4	4
SUB r	1	0	0	1	0	S	S	S	4	4
SBB r	1	0	0	1	1	S	S	S	4	4
AND r	1	0	1	0	0	S	S	S	4	4
XRA r	1	0	1	0	1	S	S	S	4	4
ORA r	1	0	1	1	0	S	S	S	4	4
CMP r	1	0	1	1	1	S	S	S	4	4
ADD M	1	0	0	0	0	1	1	0	7	7
ADC M	1	0	0	0	1	1	1	0	7	7
SUB M	1	0	0	1	0	1	1	0	7	7
SBB M	1	0	0	1	1	1	1	0	7	7
AND M	1	0	1	0	0	1	1	0	7	7
XRA M	1	0	1	0	1	1	1	0	7	7
ORA M	1	0	1	1	0	1	1	0	7	7
CMP M	1	0	1	1	1	1	1	0	7	7
ADI	1	1	0	0	0	1	1	0	7	7
ACI	1	1	0	0	1	1	1	0	7	7
SUI	1	1	0	1	0	1	1	0	7	7
SBI	1	1	0	1	1	1	1	0	7	7
ANI	1	1	1	0	0	1	1	0	7	7
XRI	1	1	1	0	1	1	1	0	7	7
ORI	1	1	1	1	0	1	1	0	7	7
CPI	1	1	1	1	1	1	1	0	7	7
RLC	0	0	0	0	0	1	1	1	4	4
RRC	0	0	0	0	1	1	1	1	4	4
RAL	0	0	0	1	0	1	1	1	4	4
RAR	0	0	0	1	1	1	1	1	4	4
JMP	1	1	0	0	0	0	1	1	10	10
JC	1	1	0	1	1	0	1	0	10	7/10
JNC	1	1	0	1	0	0	1	0	10	7/10
JZ	1	1	0	0	1	0	1	0	10	7/10
JNZ	1	1	0	0	0	0	1	0	10	7/10
JP	1	1	1	1	0	0	1	0	10	7/10
JM	1	1	1	1	1	0	1	0	10	7/10
JPE	1	1	1	0	1	0	1	0	10	7/10
JPO	1	1	1	0	0	0	1	0	10	7/10
DCX B	0	0	0	0	1	0	1	1	5	6
DCX D	0	0	0	1	1	0	1	1	5	6
DCX H	0	0	1	0	1	0	1	1	5	6
DCX SP	0	0	1	1	1	0	1	1	5	6

ALL MNEMONICS ©1974, 1975, 1976, 1977 INTEL CORPORATION

MNEMONIC	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	NUMBER OF TIME PERIODS	
									8080	8085
CMA	0	0	1	0	1	1	1	1	4	4
STC	0	0	1	1	0	1	1	1	4	4
CMC	0	0	1	1	1	1	1	1	4	4
DAA	0	0	1	0	0	1	1	1	4	4
SHLD	0	0	1	0	0	0	1	0	16	16
LHLD	0	0	1	0	1	0	1	0	16	16
RIM	0	0	1	0	0	0	0	0	–	4
SIM	0	0	1	1	0	0	0	0	–	4
EI	1	1	1	1	1	0	1	1	4	4
DI	1	1	1	1	0	0	1	1	4	4
NOP	0	0	0	0	0	0	0	0	4	4

ALL MNEMONICS ©1974, 1975, 1976, 1977 INTEL CORPORATION

The following is a summary of the instruction set:

8080/85 CPU INSTRUCTIONS IN OPERATION CODE SEQUENCE

OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC
00	NOP	2B	DCX H	56	MOV D,M	81	ADD C	AC	XRA H	D7	RST 2
01	LXI B,D16	2C	INR L	57	MOV D,A	82	ADD D	AD	XRA L	D8	RC
02	STAX B	2D	DCR L	58	MOV E,B	83	ADD E	AE	XRA M	D9	–
03	INX B	2E	MVI L,D8	59	MOV E,C	84	ADD H	AF	XRA A	DA	JC Adr
04	INR B	2F	CMA	5A	MOV E,D	85	ADD L	B0	ORA B	DB	IN D8
05	DCR B	30	SIM	5B	MOV E,E	86	ADD M	B1	ORA C	DC	CC Adr
06	MVI B,D8	31	LXI SPD16	5C	MOV E,H	87	ADD A	B2	ORA D	DD	–
07	RLC	32	STA Adr	5D	MOV E,L	88	ADC B	B3	ORA E	DE	SBI D8
08	–	33	INX SP	5E	MOV E,M	89	ADC C	B4	ORA H	DF	RST 3
09	DAD B	34	INR M	5F	MOV E,A	8A	ADC D	B5	ORA L	E0	RPO
0A	LDAX B	35	DCR M	60	MOV H,B	8B	ADC E	B6	ORA M	E1	POP H
0B	DCX B	36	MVI M,D8	61	MOV H,C	8C	ADC H	B7	ORA A	E2	JPO Adr
0C	INR C	37	STC	62	MOV H,D	8D	ADC L	B8	CMP B	E3	XTHL
0D	DCR C	38	––	63	MOV H,E	8E	ADC M	B9	CMP C	E4	CPO Adr
0E	MVI C,D8	39	DAD SP	64	MOV H,H	8F	ADC A	BA	CMP D	E5	PUSH H
0F	RRC	3A	LDA Adr	65	MOV H,L	8G	SUB B	BB	CMP E	E6	ANI D8
10	––	3B	DCX SP	66	MOV H,M	91	SUB C	BC	CMP H	E7	RST 4
11	LXI D,D16	3C	INR A	67	MOV H,A	92	SUB D	BD	CMP L	E8	RPE
12	STAX D	3D	DCR A	68	MOV L,B	93	SUB E	BE	CMP M	E9	PCHL
13	INX D	3E	MVI A,D8	69	MOV L,C	94	SUB H	BF	CMP A	EA	JPE Adr
14	INR D	3F	CMC	6A	MOV L,D	95	SUB L	C0	RNZ	EB	XCHG
15	DCR D	40	MOV B,B	6B	MOV L,E	96	SUB M	C1	POP B	EC	CPE Adr
16	MVI D,D8	41	MOV B,C	6C	MOV L,H	97	SUB A	C2	JNZ Adr	ED	––
17	RAL	42	MOV B,D	6D	MOV L,L	98	SBB B	C3	JMP Adr	EE	XRI D8
18	––	43	MOV B,E	6E	MOV L,M	99	SBB C	C4	CNZ Adr	EF	RST 5
19	DAD D	44	MOV B,H	6F	MOV L,A	9A	SBB D	C5	PUSH B	F0	RP
1A	LDAX D	45	MOV B,L	70	MOV M,B	9B	SBB E	C6	ADI D8	F1	POP PSW
1B	DCX D	46	MOV B,M	71	MOV M,C	9C	SBB H	C7	RST 0	F2	JP Adr
1C	INR E	47	MOV B,A	72	MOV M,D	9D	SBB L	C8	RZ	F3	DI
1D	DRC E	48	MOV C,B	73	MOV M,E	9E	SBB M	C9	RET Adr	F4	CP Adr
1E	MVI E,D8	49	MOV C,C	74	MOV M,H	9F	SBB A	CA	JZ	F5	PUSH PSW
1F	RAR	4A	MOV C,D	75	MOV M,L	A0	ANA B	CB	––	F6	ORI D8
20	RIM	4B	MOV C,E	76	HLT	A1	ANA C	CC	CZ Adr	F7	RST 6
21	LXI H,D16	4C	MOV C,H	77	MOV M,A	A2	ANA D	CD	CALL Adr	F8	RM
22	SHLD Adr	4D	MOV C,L	78	MOV A,B	A3	ANA E	CE	ACI D8	F9	SPHL
23	INX H	4E	MOV C,M	79	MOV A,C	A4	ANA H	CF	RST 1	FA	JM Adr
24	INR H	4F	MOV C,A	7A	MOV A,D	A5	ANA L	D0	RNC	FB	EI
25	DCR H	50	MOV D,B	7B	MOV A,E	A6	ANA M	D1	POP D	FC	CM Adr
26	MVI H,D8	51	MOV D,C	7C	MOV A,H	A7	ANA A	D2	JNC Adr	FD	––
27	DAA	52	MOV D,D	7D	MOV A,L	A8	XRA B	D3	OUT D8	FE	CPI D8
28	––	53	MOV D,E	7E	MOV A,M	A9	XRA C	D4	CNC Adr	FF	RST 7
29	DAD H	54	MOV D,H	7F	MOV A,A	AA	XRA D	D5	PUSH D		
2A	LHLD Adr	55	MOV D,L	80	ADD B	AB	XRA E	D6	SUI D8		

D8 = constant, or logical/arithmetic expression that evaluates to an 8 bit data quantity.

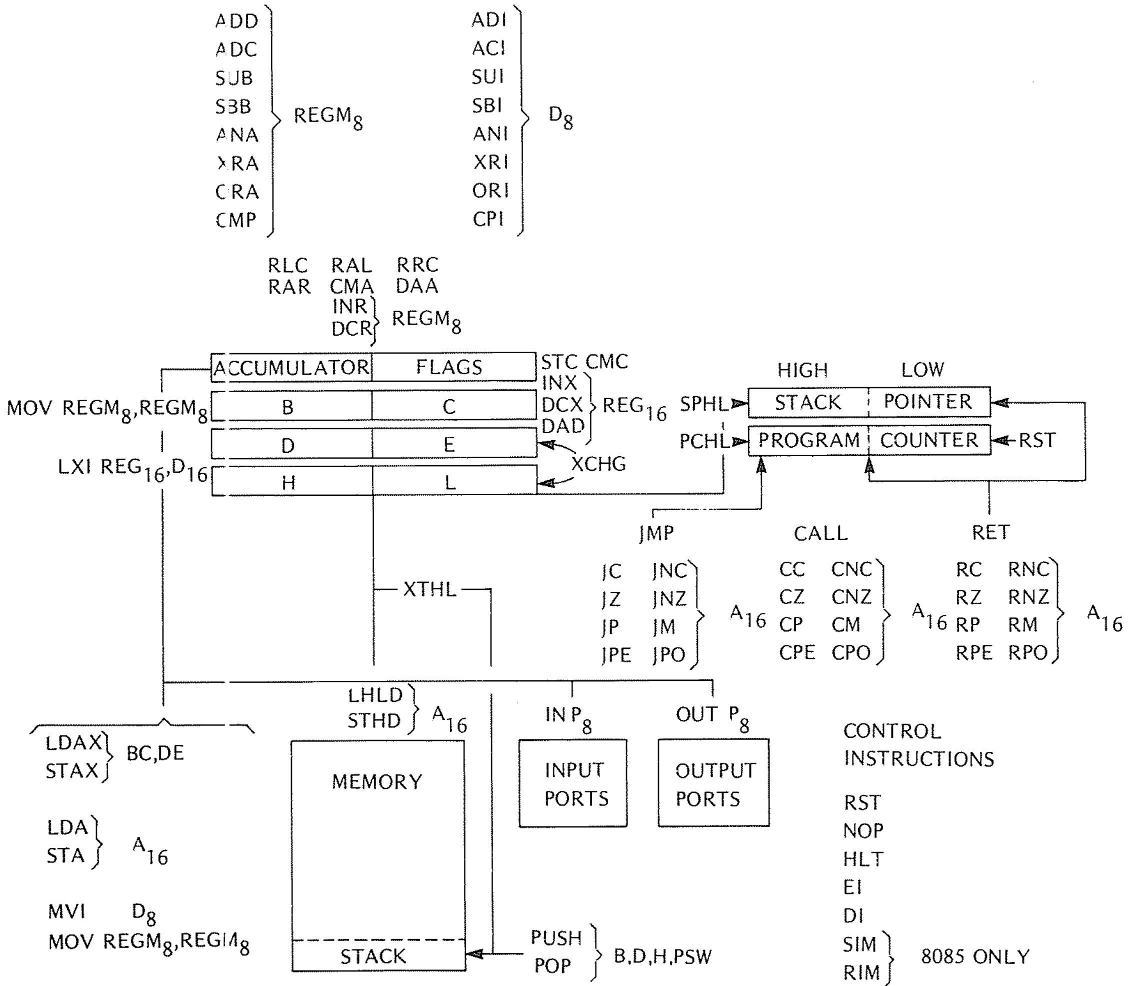
D16 = constant, or logical/arithmetic expression that evaluates to a 16 bit data quantity

Adr = 16-bit address

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

Instruction Set Guide

The following is a summary of the instruction set:



CODE	*MEANING*
$REGM_8$	The operand may specify one of the 8-bit registers A,B,C,D,E,H, or L or M (a memory reference via the 16-bit address in the H and L registers). The MOV instruction, which calls for two operands, can specify M for only one of its operands.
D_8	Designates 8-bit immediate operand.
A_{16}	Designates a 16-bit address.
P_8	Designates an 8-bit port number.
REG_{16}	Designates a 16-bit register pair (B&C,D&E,H&L,or SP).
D_{16}	Designates a 16-bit immediate operand.

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

APPENDIX B. ASSEMBLER DIRECTIVE SUMMARY

Assembler directives are summarized alphabetically in this appendix. The following terms are used to describe the contents of directive fields.

NOTATION

Term	*Interpretation*
Expression	Numerical expression evaluated during assembly; must evaluate to 8 or 16 bits depending on directive issued.
List	Series of symbolic values or expressions, separated by commas.
Name	Symbol name terminated by a space.
Null	Field must be empty or an error results.
Oplab	Optional label; must be terminated by a colon.
Parameter	Dummy parameters are symbols holding the place of actual parameters (symbolic values or expressions) specified elsewhere in the program.
String	Series of any ASCII characters, surrounded by single quote marks. Single quote within string is shown as two consecutive single quotes.
Text	Series of ASCII characters.

Macro definitions and calls allow the use of the special characters listed below.

Character	*Function*
&	Ampersand. Used to concatenate symbols.
< >	Angle brackets. Used to delimit text, such as lists, that contain other delimiters.
;;	Double semicolon. Used before a comment in a macro definition to prevent inclusion of the comment in each macro expansion.
!	Exclamation point (escape character). Placed before a delimiter to be passed as a literal in an actual parameter. To pass a literal exclamation point, issue '!!.'
%	Percent sign. Precedes actual parameters to be evaluated immediately when the macro is called.

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

SUMMARY OF DIRECTIVES

FORMAT			*FUNCTION*
Label	*Opcode*	*Operand(s)*	
oplab:	DB	exp(s) or string(s)	Define 8-bit data byte(s). Expressions must evaluate to one byte.
oplab:	DS	expression	Reserve data storage area of specified length.
oplab:	DW	exp(s) or string(s)	Define 16-bit data word(s). Strings limited to 1-2 characters.
oplab:	ELSE	null	Conditional assembly. Code between ELSE and ENDIF directives is assembled if expression in IF clause is FALSE. (See IF.)
oplab:	END	expression	Terminate assembler pass. Must be last statement of program. Program execution starts at 'exp,' if present; otherwise, at location 0.
oplab:	ENDIF	null	Terminate conditional assembly block.
name	EQU	expression	Define symbol 'name' with value 'exp.' Symbol is not redefinable.
oplab:	IF	expression	Assemble code between IF and following ELSE or ENDIF directive if 'exp' is true.
oplab:	ORG	expression	Set location counter to 'expression.'
name	SET	expression	Define symbol 'name' with value 'expression.' Symbol can be redefined.

MACRO DIRECTIVES

FORMAT			*FUNCTION*
Label	*Opcode*	*Operand(s)*	
null	ENDM	null	Terminate macro definition.
oplab:	EXITM	null	Alternate terminator of macro definition. (See ENDM.)
oplab:	IRP	dummy param,<list>	Repeat instruction sequence, substituting one character form 'list' for 'dummy param' in each iteration.

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

FORMAT			FUNCTION
Label	*Opcode*	*Operand(s)*	
oplab:	IRPC	dummy param,text	Repeat instruction sequence, substituting one character from 'text' for 'dummy param' in each iteration.
null	LOCAL	label name(s)	Specify label(s) in macro definition to have local scope.
name	MACRO	dummy param(s)	Define macro 'name' and dummy parameter(s) to be used in macro definition.
oplab:	REPT	expression	Repeat REPT block 'expression' times.

RELOCATION DIRECTIVES

FORMAT			FUNCTION
Label	*Opcode*	*Operand(s)*	
oplab:	ASEG	null	Assemble subsequent instructions and data in the absolute mode.
oplab:	CSEG	boundary specification	Assemble subsequent instructions and data in the relocatable mode using the code location counter.
oplab:	DSEG	boundary specification	Assemble subsequent instructions and data in the relocatable mode using the data location counter.
oplab:	EXTRN	name(s)	Identify symbols used in this program module but defined in a different module.
oplab:	NAME	module—name	Assigns a name to the program module.
oplab:	PUBLIC	name(s)	Identify symbols defined in this module that are to be available to other modules.
oplab:	STKLN	expression	Specify the number of bytes to be reserved for the stack for this module.

ALL MNEMONICS © 1974, 1975, 1976, 1977 INTEL CORPORATION

APPENDIX C. ASCII CHARACTER SET

ASCII CODES

The 8080 and 8085 use the seven-bit ASCII code, with the high-order eighth bit (parity bit) always reset.

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
NUL	00
SOH	01
STX	02
ETX	03
EOT	04
ENQ	05
ACK	06
BEL	07
BS	08
HT	09
LF	0A
VT	0B
FF	0C
CR	0D
SO	0E
SI	0F
DLE	10
DC1 (X-ON)	11
DC2 (TAPE)	12
DC3 (X-OFF)	13
DC4 (~~TAPE~~)	14
NAK	15
SYN	16
ETB	17
CAN	18
EM	19
SUB	1A
ESC	1B
FS	1C
GS	1D
RS	1E
US	1F
SP	20
!	21
"	22
#	23
$	24
%	25
&	26
'	27
(	28
)	29
*	2A
+	2B
,	2C
–	2D
.	2E
/	2F
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47
H	48
I	49
J	4A
K	4B
L	4C
M	4D
N	4E
O	4F
P	50
Q	51
R	52
S	53
T	54
U	55
V	56
W	57
X	58
Y	59
Z	5A
[	5B
\	5C
]	5D
∧ (↑)	5E
— (←)	5F
`	60
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78
y	79
z	7A
{	7B
|	7C
} (ALT MODE)	7D
~	7E
DEL (RUB OUT)	7F

APPENDIX D.

BINARY-DECIMAL-HEXADECIMAL CONVERSION TABLES.

POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

POWERS OF 16 (IN BASE 10)

16^n	n	16^{-n}					
1	0	0.10000	00000	00000	00000	x	10
16	1	0.62500	00000	00000	00000	x	10^{-1}
256	2	0.39062	50000	00000	00000	x	10^{-2}
4 096	3	0.24414	06250	00000	00000	x	10^{-3}
65 536	4	0.15258	78906	25000	00000	x	10^{-4}
1 048 576	5	0.95367	43164	06250	00000	x	10^{-6}
16 777 216	6	0.59604	64477	53906	25000	x	10^{-7}
268 435 456	7	0.37252	90298	46191	40625	x	10^{-8}
4 294 967 296	8	0.23283	06436	53869	62891	x	10^{-9}
68 719 476 736	9	0.14551	91522	83668	51807	x	10^{-10}
1 099 511 627 776	10	0.90949	47017	72928	23792	x	10^{-12}
17 592 186 044 416	11	0.56843	41886	08080	14870	x	10^{-13}
281 474 976 710 656	12	0.35527	13678	80050	09294	x	10^{-14}
4 503 599 627 370 496	13	0.22204	46049	25031	30808	x	10^{-15}
72 057 594 037 927 936	14	0.13877	78780	78144	56755	x	10^{-16}
1 152 921 504 606 846 976	15	0.86736	17379	88403	54721	x	10^{-18}

POWERS OF 10 (IN BASE 16)

10^n	n	10^{-n}					
1	0	1.0000	0000	0000	0000		
A	1	0.1999	9999	9999	999A		
64	2	0.28F5	C28F	5C28	F5C3	x	16^{-1}
3E8	3	0.4189	374B	C6A7	EF9E	x	16^{-2}
2710	4	0.68DB	8BAC	710C	B296	x	16^{-3}
1 86A0	5	0.A7C5	AC47	1B47	8423	x	16^{-4}
F 4240	6	0.10C6	F7A0	B5ED	8D37	x	16^{-4}
98 9680	7	0.1AD7	F29A	BCAF	4858	x	16^{-5}
5F5 E100	8	0.2AF3	1DC4	6118	73BF	x	16^{-6}
3B9A CA00	9	0.44B8	2FA0	9B5A	52CC	x	16^{-7}
2 540B E400	10	0.6DF3	7F67	5EF6	EADF	x	16^{-8}
17 4876 E800	11	0.AFEB	FF0B	CB24	AAFF	x	16^{-9}
E8 D4A5 1000	12	0.1197	9981	2DEA	1119	x	16^{-9}
918 4E72 A000	13	0.1C25	C268	4976	81C2	x	16^{-10}
5AF3 107A 4000	14	0.2D09	370D	4257	3604	x	16^{-11}
3 8D7E A4C6 8000	15	0.480E	BE7B	9D58	566D	x	16^{-12}
23 8652 6FC1 0000	16	0.734A	CA5F	6226	F0AE	x	16^{-13}
163 4578 5D8A 0000	17	0.B877	AA32	36A4	B449	x	16^{-14}
DE0 B6B3 A764 0000	18	0.1272	5DD1	D243	ABA1	x	16^{-14}
8AC7 2304 89E8 0000	19	0.1D83	C94F	B6D2	AC35	x	16^{-15}

HEXADECIMAL-DECIMAL INTEGER CONVERSION

The table below provides for direct conversions between hexadecimal integers in the range 0-FFF and decimal integers in the range 0-4095. For conversion of larger integers, the table values may be added to the following figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
020	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
030	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
040	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
050	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
060	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
070	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
080	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
090	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A0	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B0	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C0	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D0	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E0	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F0	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0331	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
200	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0301	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0212	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
400	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
510	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
590	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	4761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	3851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2866	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB0	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

INDEX

Absolute symbols 2-11, 2-16
Accumulator 1-6, 1-7
Accumulator Instructions *1-19*
ACI Instruction *3-2*
ADC Instruction 3-2
ADD Instruction 3-4
ADI Instruction *3-5*
Addressing Modes *1-15*
Addressing Registers 1-7
ANA (AND) Instruction *3-6*
AND Operator *2-13*
ANI (AND Immediate) Instruction *3-7*
Arithmetic Expression Operators 2-12
Arithmetic Instructions *1-17*
ASCII Constant 2-6
ASEG (Absolute Segment) Directive *4-14*
Assembler, Need for 1-3
Assembler Character Set 2-1
Assembler Compared with PL/M 1-3
Assembler Function 1-1
Assembler Termination *4-10*
Assembly-Time Expression Evaluation 2-11
Auxiliary Carry Flag *1-11*
Auxiliary Carry Flag Setting – 8080/8085 Differences *1-12*

Binary Data (Coding Rules) 2-6
blank (character) 2-3
Branching Instructions *1-18, 1-22*
Branch Table 6-1
Byte Isolation Operations *2-14*

CALL Instruction *3-8*
Carry Flag *1-10*
CC (Call if Carry) Instruction *3-10*
CM (Call if Minus) Instruction *3-10*
CMA (Complement Accumulator) Instruction *3-11*
CMC (Complement Carry) Instruction *3-12*
CMP (Compare) Instruction *3-12*
CNC (Call if no carry) Instruction *3-14*
CNZ (Call if not Zero) Instruction *3-14*
Combined Addressing Modes *1-16*
Comment Field 2-4
Compare Operators *2-13*
Comparing Complemented Data *2-8*
Comparisons in Expressions 2-13
Complement Used for Subtraction 2-7
Complemented Data 2-8
Concatenation 5-10, 5-11, 5-15, 5-16

Condition Flags . . . 1-9
Conditional Assembly . . . 4-8
CP (Call if Positive) Instruction . . . 3-15
CPE (Call if Parity Even) Instruction . . . 3-16
CPI (Compare Immediate) Instruction . . . 3-16
CPO (Call if Parity Odd) Instruction . . . 3-17
CSEG (Code Segment) Directive . . . 4-15
CZ (Call if Zero) Instruction . . . 3-18

DAA (Decimal Adjust Accumulator) Instruction . . . 3-18
DAD (Double Register Add) Instruction . . . 3-20
Data Access Example . . . 4-7
Data Definition . . . 4-3
Data Description Example . . . 4-6
Data for Subroutines . . . 6-3
Data Label . . . 2-5
Data Transfer Instructions . . . 1-16
DB (Define Byte) Directive . . . 4-3
DCR (Decrement) Instruction . . . 3-20
DCX (Decrement Register Pair) . . . 3-22
Decimal Addition Routine . . . 6-12
Decimal Data (Coding Rules) . . . 2-5
Decimal Subtraction Routine . . . 6-14
Delimiters . . . 2-2
DI (Disable Interrupts) Instruction . . . 3-22, 3-60
Direct Addressing . . . 1-15
Divide (Software Example) . . . 6-9
Division in Expressions . . . 2-12
DS (Define Storage) Directive . . . 4-5
DSEG (Data Segment) Directive . . . 4-15
Dummy Parameters . . . 5-4
DW (Define Word) Directive . . . 4-4

EI (Enable Interrupts) Instruction . . . 3-23
ELSE Directive . . . 4-8
END Directive . . . 4-10
ENDIF Directive . . . 4-8
ENDM (End Macro) Directive . . . 5-5, 5-6, 5-7, 5-12
EOT Directive . . . 4-11
EPROM . . . 1-5
EQ Operator . . . 2-13
EQU Directive . . . 4-2
EXITM (Exit Macro) Directive . . . 5-9
Expression Evaluation . . . 2-11
Expression Operators . . . 2-11
Expressions . . . 2-6
Expressions, Precedence of Operators . . . 2-15
Expressions, Range of Values . . . 2-15
EXTRN Directive . . . 4-17

GE Operator *2-13*
General Purpose Registers *1-7*
GT Operator *2-13*

Hardware Overview 1-5
Hexadecimal Data (Coding Rules) 2-5
HIGH Operator *2-14*, 3-2, 3-5, 3-7, 404
HLT (Halt) Instruction *3-24*

IF Directive *4-8*
Immediate Addressing *1-15*
Implied Addressing *1-15*
IN (Input) Instruction *1-14*, *3-24*
INPAGE Reserved Word *4-14*, *4-15*
Input/Output Ports *1-14*
INR (Increment) Instruction *3-25*
Instruction Addressing Modes *1-15*
Instruction Execution 1-9
Instruction Fetch 1-8
Instruction Label 2-6
Instruction Naming Conventions 1-16
Instruction Set Guide *1-23*
Instruction Summary 1-19, *1-23*
Instruction Timing *3-1*
Instructions as Operands 2-7
INTE Pin 3-49
Internal Registers 1-6
Interrupt Subroutines 7-4
Interrupts 7-1
Interrupts (8085) 1-24
INX (Increment Register Pair) Instructions *3-26*
IRP (Indefinite Repeat) Directive *5-8*, 5-12, 5-22
IRPC (Indefinite Repeat Character) *5-8*, 5-12, 5-17

JC (Jump if Carry) Instruction *3-26*
JM (Jump if Minus) Instruction *3-27*
JMP (Jump) Instruction *3-28*
JNC (Jump if no carry) Instruction *3-28*
JNZ (Jump if not zero) Instruction *3-29*
JP (Jump if Positive) Instruction *3-29*
JPE (Jump if parity Even) *3-30*
JPO (Jump if parity Odd) *3-31*
JZ (Jump if Zero) Instruction *3-32*

Label Field *2-3*
Labels 2-6
LDA (Load Accumulator Direct) Instruction *3-32*
LDAX (Load Accumulator Indirect) *3-33*

LE Operator . . . 2-13
LIB Program . . . 4-12
LHLD (Load L Direct) Instruction . . . 3-34
LINK Program . . . 4-12, 4-14, 4-15
Linkage . . . 4-16
List File . . . 1-1
LOCAL Directive . . . 5-5
LOCAL Symbols . . . 5-6
LOCATE Program . . . 4-12, 4-13, 4-14, 4-19
Location Counter (Coding Rules) . . . 2-6
Location Counter Control (Absolute Mode) . . . 4-11
Location Counter Control (Relocatable Mode) . . . 4-14
Logical Instructions . . . 1-17
Logical Instructions, Summary . . . 3-6
Logical Operators . . . 2-13
LOW Operator . . . 2-14, 3-2, 3-5, 3-7, 4-4
LT Operator . . . 2-13
LXI (Load Register Pair Immediate) . . . 3-35

Macros . . . 5-1
Macro Calls . . . 5-12
Macro Definition . . . 5-4
MACRO Directive . . . 5-4
Macro Expansion . . . 5-15
Macro Parameters . . . 5-5
Macros versus Subroutines . . . 5-3
Manual Programming . . . 1-3
Memory . . . 1-5
Memory Management with Relocation . . . 4-12
Memory Reservation . . . 4-5
MEMORY Reserved Word . . . 4-19
MOD Operator . . . 2-12
Modular Programming . . . 4-12
MODULE Default Name . . . 4-17
MOV (Move) Instruction . . . 3-36
Multibyte Addition Routines . . . 6-11
Multibyte Subtraction Routine . . . 6-11
Multiplication in Expressions . . . 2-12
Multiply (Software Example) . . . 6-7
MVI (Move Immediate) . . . 3-37

NAME Directive . . . 4-18
NE Operator . . . 2-13
Nested Macro Calls . . . 5-14
Nested Macro Definitions . . . 5-12
Nested Subroutines . . . 3-48
Nine's Complement . . . 2-7
NOP (No Operation) Instruction . . . 3-38

NOP via MOV 3-36
NOT Operator *2-13*
NUL Operator 2-13, *5-11*
Null Macros 5-16
Null Parameter 5-11

Object Code *1-2*
Object File 1-1
Octal Data (Coding Rules) 2-5
One's Complement 2-7
Opcode 1-1
Opcode Field 2-4
Operand Field 2-4
Operand Field (Coding Rules) 2-4
Operands 2-5
Operators, Expression 2-11
OR Operator *2-13*
ORG (Origin) Directive (Absolute Mode) *4-11*
ORG (Origin) Directive (Relocatable Mode) *4-16*
ORA (Inclusive OR) Instruction *3-38*
ORI (Inclusive OR Immediate) *3-40*
OUT Instruction 1-14, *3-41*

PAGE Reserved Word *4-14*, *4-15*
Parity Flag *1-11*
PCHL (Move H & L to Program Counter) Instruction *3-42*
Permanent Symbols 2-11
PL/M 1-3
PL/M Compared with Assembler 1-3
POP Instruction *3-42*
POP PSW instruction *3-43*
Precedence of Expression Operators *2-15*
Processor Registers 1-9
Program Counter 1-6
Program Linkage Directives *4-16*
Program Listing 1-2
Program Status 1-13
Program Status Word (PSW) *1-14*
Programming the 8085 1-24
PROM 1-5
PSW *1-14*, 3-45
PUBLIC Directive *4-17*
PUSH Instruction *3-44*
PUSH PSW Instruction *3-45*

RAM 1-5
RAM versus ROM *4-6*
RAL (Rotate Left through Carry) Instruction *3-45*

RAR (Rotate Right through Carry) Instruction 3-46
RC (Return if Carry) Instruction 3-47
Redefinable Symbols 2-11
Register Addressing 1-15
Register Indirect Addressing 1-16
Register Pair Instructions 1-21
Register Pairs 1-7
Relocatability Defined 4-12
Relocatable Expressions 2-16, 2-19
Relocatable Symbols 2-11
Relocation Feature 1-2
Reserved Symbols 2-9
RESET Signal 3-24
RET (Return) Instruction 3-48
REPT Directive 5-6, 5-12, 5-15, 5-16, 5-17, 5-18
RIM (Read Interrupt Mask) 8085 Instruction 3-48
RLC (Rotate Accumulator Left) Instruction 3-49
RM (Return if Minus) Instruction 3-50
RNC (Return if no Carry) Instruction 3-51
RNZ (Return if not Zero) Instruction 3-51
ROM 1-5
RP (Return if Positive) Instruction 3-52
RPE (Return if Parity Even) Instruction 3-52
RPO (Return if Parity Odd) Instruction 3-53
RRC (Rotate Accumulator Right) Instruction 3-53
RST (Restart) Instruction 3-54
RST5.5 . . . . 3-49, 3-55, 3-59, 3-60
RST6.5 . . . . 3-49, 3-55, 3-59, 3-60
RST7.5 . . . . 3-49, 3-55, 3-59, 3-60
RZ (Return if Zero) Instruction 3-55

Savings Program Status 1-13
SBB (Subtract with Borrow) Instruction 3-56
SBI (Subtract Immediate with Borrow) Instruction 3-57
Scope of Symbols 2-10
SET Directive 4-3
Shift Expression Operators 2-12
Shift Operations in Expressions 2-12
SHL Operator 2-12
SHLD (Store H & L Direct) Instruction 3-58
SHR Operator 2-12
Sign Flag 1-10
SIM (Set Interrupt Mask) 8085 Instruction 3-59
Software Divide Routine 6-7
Software Multiply Routine 6-7
Source Code Format 2-1
Source Line Fields 2-1
Source Program File 1-1
SPHL (Move H & L to Stack Pointer) Instruction 3-67

SP (Stack Pointer Register) . . . 3-35
STA (Store Accumulator Direct) Instruction . . . *3-61*
Stack . . . *1-12*
Stack and Machine Control Instructions . . . *1-19*
Stack Operations . . . 1-13
Stack Pointer . . . *1-12*
STACK Reserved Word . . . *4-19*, 3-35
Start Execution Address . . . 4-10
STAX (Store Accumulator Indirect) Instruction . . . *3-62*
STC (Set Carry) Instruction . . . *3-63*
STKLN Directive . . . *4-18*
SUB (Subtract) Instruction . . . *3-63*
Subroutine Data . . . 6-3
Subroutines . . . *1-12*, 3-9
Subroutines versus Macros . . . 5-3
Subtraction for Comparison . . . 3-12
SUI (Subtract Immediate) Instruction . . . *3-64*
Symbol-Cross-Reference File . . . 1-1, *1-3*
Symbol Definition . . . *4-2*
Symbol Table . . . 2-9
Symbolic Addressing . . . 2-9
Symbols . . . *2-9*
Symbols, Absolute . . . 2-11
Symbols (Coding Rules) . . . 2-9
Symbols, Global . . . 2-10
Symbols, Limited . . . 2-10
Symbols, Permanent . . . 2-11
Symbols, Redefinable . . . 2-11
Symbols, Relocatable . . . 2-11
Symbols, Reserved . . . 2-9

TRAP Interrupt . . . 3-54
Ten's Complement . . . 2-7
Testing Relocatable Modules . . . 4-19
Timing Effects of Addressing Modes . . . 1-16
TRAP (8085) . . . 3-23
Two's Complement Data . . . 2-7

Use of Macros . . . 5-1
Using Symbols for Data Access . . . *4-7*

Value of Expressions . . . *2-15*

What is a Macro? . . . 5-2
Word Instructions . . . *1-21*
Word Storage in Memory . . . 4-4
Work Registers . . . 1-7

XCHG (Exchange H & L with D & E) Instruction 3-65
XOR Operator 2-13
XRA (Exclusive OR) Instruction 3-66
XRI (Exclusive OR Immediate) Instruction 3-67
XTHL (Exchange H & L with Top of Stack) Instruction 3-69

Zero Flag 1-11

& (ampersand) 5-10
<> (angle brackets) 5-10
CR (carriage return character) 2-2
: (colon) 2-2
, (comma) 2-2
;; (double semicolon) 5-10
/ (division) Operator 2-12
! (exclamation point) 5-10
HT (horizontal tab character) 2-2
— (minus) Operator 2-12
* (multiplication) Operator 2-12
( ) (parentheses) 2-2
+ (plus) Operator 2-12
??nnnn Symbols 5-5
; (semicolon) 2-2
' (single quote) 2-2
space (character) 2-2

8080/8085 Differences 1-24
8085 Features 1-24
8085 Processor 1-24
8085 Programming 1-24

NOTES

NOTES