TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND SOFTWARE
PURCHASED FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND
RADIO SHACK FRANCHISEES OR DEALERS AT THEIR AUTHORIZED LOCATIONS

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment. RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations**. The warranty is void if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."**
NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the TANDY Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on a multiuser or network system only if either, the Software is expressly labeled to be for use on a multiuser or network system, or one copy of this software is purchased for each node or terminal on which Software is to be used simultaneously.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

6/86

Debug/Assembler

10 9 8 7 6 5 4 3 2

Contents

To Our Customers ...

The heart of the Model 100 is an 8085 "processor." It controls all parts of the Model 100.

The processor understands only 0s and 1s, a code not at all intelligible to the human mind. This code is called "8085 machine code."

When you run a BASIC program, a system calls the "BASIC Interpreter" and translates each statement, one at a time, into 8085 machine code. This is an easy, but inefficient, way to program.

The Model 100 Debug/Assembler lets you program using an intelligible representation of 8085 machine code, called "assembly language," that talks directly to the processor. You then assemble the entire program into 8085 machine code before running it.

Programming with the Model 100 Debug/Assembler gives you these benefits:

- You have direct and complete control of the Model 100.
- You can use its many features—such as the assembler, disassembler, and breakpoint debugger—for memory location modification.
- Your program executes faster. This is because it is already translated into 8085 machine code when you run it.

Note: The page numbers referenced in the appendices refer to the Intel publication, *The MSC® -80/85 Family User's Manual.*

To Use the Model 100 Debug/Assembler You Need...

A Model 100 Portable Computer with 32K of RAM, a cassette recorder, and the Model 100 Debug/Assembler program cassette tape.

This manual uses these terms and notations:

(KEY)	A key you must press.
Italics	A value you must supply.
filename	A Model 100 file specification. CAS:*filename.ext* *filename* has 1-6 characters. *extension* has 1-3 characters. If you omit the extension, the Model 100 Debug/Assembler uses **.DO**. CAS: saves the file to cassette.
H	A hexadecimal (Base 16) number. For example, ØFH represents hexadecimal ØF, which is equal to 15 in decimal (Base 1Ø) notation. Hexadecimal numbers should always begin with a numeral so that the Model 100 Debug/Assembler can distinguish between the number and its corresponding register name. Use a leading Ø with hexadecimal numbers beginning with an A, B, C, D, E, or F.

PART I/ GETTING STARTED

This section gets you started with the Model 100 Debug/Assembler and explains some concepts you need to know.

CHAPTER 1/ LOADING THE MODEL 100 DEBUG/ASSEMBLER

This chapter shows how to load the Model 100 Debug/Assembler from cassette tape into your Model 100 memory and save it to RAM.

Load the Model 100 Debug/Assembler From Cassette

Be sure your Model 100 is turned on and the cassette recorder properly connected. Insert the Model 100 Debug/Assembler cassette tape into the recorder and press the Play button.

Next, position the cursor over BASIC and press (ENTER). At BASIC's OK prompt, type (in either uppercase or lowercase letters):

```
SOUND OFF   (ENTER)
```

Note: Always be sure to turn off the Model 100's sound whenever you use the cassette recorder.

When the OK prompt appears, type:

```
CLEAR 0,49151   (ENTER)
```

Another OK prompt appears. Now type:

```
CLOADM   (ENTER)
```

This procedure turns the sound off before loading the program, clears memory for the program, and then loads the program from the cassette.

Note: The Model 100 Debug/Assembler program is stored with the filename **ZBGASM**. The program-debugging portion of the Debug/Assembler is called ZBUG.

The computer shows that it has located the ZBGASM program by displaying:

```
Found:ZBGASM
Top:  54272
End:  62475
Exe:  54275
```

The OK prompt signals that the Model 100 Debug/Assembler is in memory.

At this point you can either execute the program or save the program to RAM for future use.

To execute the program, type:

```
CALL 54272   (ENTER)
```

The screen shows the startup message and the ZBUG prompt:

```
M100 ZBGASM/01.xx.xx
Copr. 1984  Microsoft

#
```

To save the program to RAM, type:

```
SAVEM "ZBGASM",54272,62475,54272   (ENTER)
```

Now press (F8) to return to the main menu. The file ZBGASM.CO has been added. To execute the Model 100 Debug/Assembler, position the cursor over ZBGASM.CO and press (ENTER). The screen shows the startup message and the ZBUG prompt.

Note: the Model 100 Debug/Assembler occupies approximately 8.5K bytes of RAM. If you have other programs or files in memory, you may not be able to save ZBGASM. In this case, either load the Model 100 Debug/Assembler each time you wish to use it or delete some of your other files.

CHAPTER 2/ RUNNING A SAMPLE PROGRAM

This "sample session" gets you started writing programs and shows how to use the Model 100 Debug/Assembler. The next chapters explain why the program works the way it does.

1. Start TEXT

Use the TEXT program to enter your assembly language programs. At the main menu, position the cursor over TEXT and press (ENTER).

TEXT prompts:

```
File to edit?
```

Type SAMPLE for this sample session.

TEXT then shows a blank screen with a flashing cursor at the top left corner of the screen.

2. Type the Source Program

Begin your assembly language program by typing:

```
(TAB) ORG (TAB) ØCCØØH (ENTER)
```

The (TAB) key tabs to the next column. (ENTER) inserts the line into TEXT. The H means that the address CCØØ is a hexadecimal number.

If you make a mistake, use the arrow keys to position the cursor at the mistake and type the correction. Use the (DEL) key if necessary. TEXT is described fully in your owner's manual, and a summary of Text Editor commands is in your *Model 100 Quick Reference Guide.*

Insert the entire assembly language program listed below.

```
          ORG     0CC00H
HOME:     CALL    422DH
SCREEN:   MVI     A,0FFH
          CALL    4B44H
ROW:      LDA     0F639H
          CPI     8H
          JNZ     SCREEN
COL:      LDA     0F63AH
          CPI     28H
          JNZ     SCREEN
WAIT:     CALL    12CBH
EXIT:     CALL    5797H
          END     HOME
```

If you make a mistake, use TEXT's edit commands to correct it.

The program you have inserted is an assembly language "source" program, which we explain in the next chapter.

Press (F8) to exit TEXT. Your assembly language source program is automatically saved in RAM with the extension .DO.

3. Assemble the Source Program in Memory

If you saved ZBGASM as a RAM file, position the cursor over ZBGASM.CO and press ENTER. This starts the the Model 100 Debug/Assembler program. At the # prompt, type:

```
ASM SAMPLE.DO SAMPLE /WE ENTER
```

This loads the assembler program. The assembler then assembles your source program into 8Ø85 machine code at the memory area just above ZBGASM. To let you know what it has done, it prints this listing:

```
CCØØ                        ORG     ØCCØØH
CCØØ CD  422D    HOME:      CALL    422DH
CCØ3 3E  FF      SCREEN:    MVI     A,ØFFH
CCØ5 CD  4B44               CALL    4B44H
CCØ8 3A  F639    ROW:       LDA     ØF639H
CCØB FE  Ø8                 CPI     8H
CCØD C2  CCØ3               JNZ     SCREEN
CC1Ø 3A  F63A    COL:       LDA     ØF63AH
CC13 FE  28                 CPI     28H
CC15 C2  CCØ3               JNZ     SCREEN
CC18 CD  12CB    WAIT:      CALL    12CBH
CC1B CD  5797    EXIT:      CALL    5797H
         CCØØ               END     HOME

ØØØØØ ERRORS
```

If the assembler does not print this entire listing but stops and shows an error message instead, you have an error in the source program. Repeat Steps 1 and 2.

When the assembly is complete, a new file is created called SAMPLE.CO. This file is the "object code" of your assembly language program.

The assembler commands are explained in Chapter 5, "Assembling the Program."

4. Run the Assembled Program

To run the assembled program, you need to be at the main menu. If you are at the # prompt, press (F8) to return to the menu.

Position the cursor over SAMPLE.CO and press (ENTER). The SAMPLE program executes, filling the entire screen with a checkerboard of graphics characters.

Press (SPACEBAR) to exit the program and return to the main menu.

5. Debug the Program (if necessary)

ZBUG lets you look at memory. Load the Model 100 Debug/ Assembler again.

ZBGASM loads its ZBUG program and displays ZBUG's # prompt. You can now examine any memory address. Type:

```
ØCCØØH/
```

and ZBUG shows you what is in memory at this address. Press (↓) a few times to look at more memory addresses. When you finish, press (ENTER).

In Chapter 8, we show you how to use ZBUG to examine and test your program. To exit the Model 100 Debug/Assembler, press (F8).

CHAPTER 3/ OVERVIEW

The Model 100's 8085 processor understands only instructions written in 8085 machine code: a code of 0s and 1s containing "opcodes" and data. "Opcodes" are instructions that tell the processor to manipulate data in some way.

For example, the machine-code instruction "00111110 11111111" contains:

- The opcode "00111110" (decimal 62 or hexadecimal 3E)
- The data "11111111" (decimal 255 or hexadecimal FF)

This instruction tells the processor to move the value 11111111 into Register A.

ZBGASM looks for 3 fields in your instructions: label, command, and operand. For example, in this instruction:

```
HOME:     CALL     422DH
```

HOME is the label (a colon delimiter must follow the label); CALL is the command; and 422DH is the operand.

In the label field, the Model 100 Debug/Assembler looks for:

- Symbols (symbolic names)

In the command field, it looks for:

- Mnemonics
- Pseudo Ops

In the operand field, it looks for:

- Symbols
- Operators
- Data

Symbols

A symbol is similar to a variable. It can represent a value or a location. HOME (in the sample session) is a symbol that represents the location of the instruction CALL 422DH. Operands can also be symbols. For example, in the instruction JNZ SCREEN, SCREEN is a symbol that represents the location of MVI A,0FFH.

Mnemonics

A mnemonic is a symbolic representation of an opcode. It is a command to the processor. "MVI" is a mnemonic. In the instruction MVI A,0FFH, MVI is the mnemonic for the opcode 00111110, which moves the data 0FFH (255) into Register A.

Mnemonics are specific to a particular processor. For example, if a computer uses a Z80 processor, it would understand only Z80 mnemonics, and not 8085 mnemonics.

Pseudo Ops

A pseudo op is a command to the assembler. END (in the sample session) is a pseudo op. It tells the assembler to quit assembling the program.

Data

Data is numbers or characters. Many mnemonics and pseudo ops call for data. Unless you use an operator (described next), the assembler interprets your data as a decimal (Base 10) number.

Operators

An operator tells the assembler to perform a certain operation on the data. In the value 422DH the "H" is an operator. It tells the assembler that 422D is a hexadecimal (Base 16) number, rather than a decimal (Base 10) number.

The more commonly used operators are arithmetic and relational. Addition (+) and equation (=) are examples of these operators.

Pseudo ops, symbols, operators, and addressing-mode characters vary from one assembler to another. Section III explains them in detail.

Sample Program

This is how each line in the sample program works:

```
                ORG        0CC00H
```

ORG is a pseudo op for "originate." It tells the assembler to begin loading the program at Location CC00H (Hexadecimal CC00). This means that when you load and run the program from the Model 100 main menu the program starts at Memory Address CC00H.

```
        HOME:   CALL       422DH
```

HOME is a symbol. It equals the location where the CALL 422DH instruction is stored.

CALL is a mnemonic that executes a subroutine. In this case, the routine being "called" is the Model 100 ROM routine to move the cursor to Row 1, Column 1 (home the cursor). The Model 100 ROM routines are listed in Reference Section F.

```
        SCREEN: MVI        A,0FFH
```

SCREEN is a symbol. It equals the location where MVI A,0FFH is stored.

MVI is a mnemonic for "move immediate." It loads Register A with FFH, which is the hexadecimal ASCII code for a graphics character. Notice that the value FFH is preceded by a zero. All values that begin with a letter (A, B, C, D, E, F) must be preceded by a zero. The ASCII characters are listed in Reference Section G.

```
                CALL      4B44H
```

This instruction calls the Model 100 ROM routine to display a character at the current cursor position.

```
        ROW:    LDA       ØF639H
```

LDA is a mnemonic for "load Register A." It loads Register A with the hexadecimal address F639. This address contains the current row position of the cursor.

```
                CPI       8H
```

CPI is the mnemonic for "compare immediate." It compares the value 8H to the contents of Register A. The maximum number of rows is 8.

```
                JNZ       SCREEN
```

JNZ is a mnemonic for "jump on not zero." If Register A does not equal 8H, the Z flag is not set, and the program jumps to the line labeled SCREEN. If Register A does equal 8H, the Z flag is set, and the program continues with the next line.

```
        COL:    LDA       ØF63AH
```

This time, Register A is loaded with the current column position of the cursor (stored at address F63AH). COL: is the label.

```
                CPI       28H
```

The contents of Register A are compared to 28H (40 decimal). This is the maximum number of characters per line.

```
                JNZ       SCREEN
```

If Register A does not equal 28H, the Z flag is not set, and the program jumps to the line labeled SCREEN. If Register A does equal 28H, the Z flag is set, and the program continues with the next line.

```
    WAIT:       CALL      12CBH
```

This line calls the ROM routine that "waits" until a character is input from the keyboard.

```
    EXIT:       CALL      5797H
```

After a key on the keyboard is pressed, the program exits by calling the ROM routine that displays the main menu.

```
                END       HOME
```

END is a pseudo op that tells the assembler to quit assembling the program. It also tells the assembler to store the beginning address of the program (the value of HOME).

PART II/ COMMANDS

This section shows how to use the Model 100 Debug/Assembler commands. Knowing these commands will help you edit and test your program.

CHAPTER 4/ EXAMINING MEMORY (ZBUG Commands — Part I)

To use the Model 100 Debug/Assembler, you must understand the Model 100's memory. You need to know about memory to write the program, assemble it, debug it, and execute it.

In this chapter, we explore memory and see some of the many ways to get information. To do this, we use ZBUG.

If you are not "in" ZBUG, with the ZBUG prompt (#) displayed, you need to get in it now.

Enter BASIC and load the Model 100 Debug/Assembler, then execute the program by typing :

```
CALL 54272 (ENTER)
```

The screen shows the ZBUG prompt (#) which means you are in ZBUG and can enter a command. You must enter all ZBUG commands at this level. You can return to the command level at any time by pressing (ENTER).

Examining a Memory Location

The 8085 can address 65,535 one-byte memory addresses, numbered 0-65535 (0000H-0FFFFH).

> **Note:** Although the processor may address 65,535 addresses, the bottom 32,768 (7FFFH) are Read-Only-Memory (ROM) addresses and may not be changed.

We'll examine ROM Address 0H. At the # prompt, type:

```
B (ENTER)
```

to get into the "byte mode." Then type:

```
ØH/
```

and the screen shows the contents of Address Ø. To see the contents of the next bytes, press (↓). Use (↑) to scroll to the preceding addresses.

Continue pressing (↓) or (↑). Notice that as you use the (↑) the screen continues to scroll down. The smaller addresses are on the lower part of the screen.

All the numbers you see are hexadecimal (Base 16). You see not only the 1Ø numeric digits, but also the 6 alphabetic characters needed for Base 16 (A-F). Unless you specify another base (which we do in Chapter 6), ZBUG assumes you want to see Base 16 numbers.

Notice that a zero precedes all hexadecimal numbers that begin with an alphabetic character. This is to avoid any *confusion between hexadecimal numbers and registers.*

Examination Modes

To help you interpret the contents of memory, ZBUG lets you examine it in four ways:

Examination Mode	**Command**
Byte	B (ENTER)
Word	W (ENTER)
ASCII	A (ENTER)
Mnemonic	M (ENTER)

Byte Mode

Until now, you've been using the byte mode to examine memory. Typing B ENTER at the # prompt put you into this mode.

The byte mode displays every byte of memory as a number, whether it is part of a machine-language program or data.

In byte mode, ↓ increments the address by 1. ↑ decrements the address by 1.

Word Mode

Type ENTER to get back to the # prompt. To enter the word mode, type:

    W ENTER

Look at the same memory address again. Press the ↓ key a few times. In this mode, the ↓ increments the address by 2. The numbers contained in each address are the same, but you see them 2 bytes or 1 word at a time.

Press the ↑ a few times. The ↑ always decrements the address by 1, regardless of the examination mode.

ASCII Mode

Return to the command level. To enter the ASCII mode, type:

    A ENTER

ZBUG now assumes the content of each memory address is an ASCII code. If the "code" is between 21H and 7FH, ZBUG displays the character it represents. Otherwise, it leaves the line following the address blank.

In the ASCII mode, ↓ increments the address by 1.

Mnemonic Mode

This is the default mode. Unless you specify some other mode, you are in the default mode.

Return to the # prompt. To enter the mnemonic mode from another mode, type:

```
M (ENTER)
```

Look at the lowest ROM addresses. Type:

```
ØH/
```

and, then press (↓) a few times. In the mnemonic mode, ZBUG assumes you're examining an assembly language program. The (↓) increments memory one to 5 bytes at a time (depending on the length of the instruction) by "disassembling" the numbers into the mnemonics they represent.

In our example, the first command—JMP 7D33—is a 3-byte instruction. Go into byte mode and list 3 bytes. The first byte (C3) is the opcode for the mnemonic JMP. The second byte (33) is the least significant byte of the address, and the third byte (7D) is the most significant byte of the address.

> **Note:** In any instruction, the first byte always contains the opcode. In 3-byte instructions, **the address is stored in the second and third bytes with the least significant byte of the address in the second byte and the most significant byte of the address in the third byte.**

Begin the disassembly at a different byte. Press (ENTER) and then examine Address 1. Type:

```
1/
```

Even though you now see a different disassembly, the contents of memory have not actually changed. ZBUG is merely interpreting them differently.

For example, assume 1 contains a 33. This is really the low address byte of the JMP instruction. But when you begin disassembly at this location, ZBUG reads the first byte as an opcode. So 33 is interpreted as an INX SP instruction — and not as part of the address.

To see the program correctly, you must be sure you are beginning at the correct byte.

Sometimes, several bytes contain the symbol "??". This means ZBUG can't figure out which instruction is in that byte and is possibly disassembling from the wrong point. The only way to know you're on the right byte is to know where the program starts.

Changing Memory

As you look at individual memory addresses, notice that the cursor remains to the right of the contents of each address. This lets you change the contents of that address. After typing the new contents, press (ENTER) or (↓); the change is made. To make no changes, press (ENTER) or (↓).

To show how to change memory, we'll open an address in memory above ROM and the HIMEM area. Get into the byte mode and open Address CØØØ by typing:

```
B (ENTER)
ØCØØØH/
```

Notice that the cursor is to the right of the address. To put a 1 in that address, type:

```
1 ENTER
```

If you want to change the contents of more than 1 address, type:

```
ØCØØ1H/
```

Then type:

```
ØDD ↓
```

This changes the contents to DD and lets you change the next address. (Press ↑ to verify that the change has been made.)

The size of the changes you can make depends on the examination mode. In the byte mode, you can change only 1 byte at a time and can type 1 or 2 digits.

In the word mode, you can change 1 word at a time. Any 1-, 2-, 3-, or 4-digit number you type becomes the new value of the word.

If you type a hexadecimal number that is also the name of an 8Ø85 register (A,B,C,D,E,H,L,SP,PC), ZBUG assumes it's a register and gives you an "EXPression ERROR." To avoid this confusion, include a leading zero (ØA,ØB, etc.) whenever you enter hexadecimal numeric data that begins with a letter.

To change memory in the ASCII mode, use an apostrophe before the new letter. For example, here's how to write the letter C in memory at Address CØØØ. To get into the ASCII examination mode, type:

```
A ENTER
```

To open Address CØØØ,type:

```
ØCØØØH/
```

To change its contents to a C, type:

```
'C ↓
```

Press ↑ to confirm that the address contains the letter C.

If you are in mnemonic mode, you must change 1 to 5 bytes of memory depending on the length of the opcode. Changing memory is complex in mnemonic mode because you must type the opcodes rather than the mnemonic.

For example, get into the mnemonic mode and open Address CØØØ. Type:

```
M ENTER
ØCØØØH/
```

To change the instruction at this address to an ADD IMMEDIATE instruction, type:

```
C6 ENTER
```

Now Address ØCØØØH contains the opcode for the ADI mnemonic. Open location ØCØØ1H:

```
ØCØØ1/
```

and insert Ø6, the operand:

```
Ø6 ENTER
```

Examine Address ØCØØØH again, and you see it contains an ADI 6 instruction.

CHAPTER 5/ ASSEMBLING THE PROGRAM (Assembler Commands)

To load the assembler program and assemble the source program into 8085 machine code, the Model 100 Debug/Assembler has an "assembly command." Depending on how you enter the command, the assembler:

- Shows an "assembly listing," which gives information on how the assembler is assembling the program.
- Stores the assembled program in a RAM file.
- Stores the assembled program on tape.

This chapter shows the different ways you can control the assembly listing and the in-memory assembly. Knowing this will help you debug a program.

The Assembly Command

The command to assemble your source program into 8085 machine code is:

Assembling in a RAM file:

```
ASM <source filename> <object filename> /SW1/SW2/...SWn
```

SW1, SW2, etc. are switches.

> **Note:** If you omit the object filename, the object code is written to memory and not saved as a RAM file.

Assembling on tape:

```
ASM <source filename> CAS:<object filename> /SW1/...SWn
```

The assembled program is stored on tape under the same name as *source filename* if you do not use an object *filename*. Since source files must be RAM files created by the Text Editor (TEXT), they always carry the .DO extension.

The *switch* options are as follows:

/LP	Assembly listing on the line printer
/MR	Multiple record object code file
/NL	No listing
/NO	No object code
/NS	No symbol table
/WE	Wait on assembly errors

You may use any combination of the switch options. Be sure to include a blank space before the first switch.

Examples:

```
ASM TEST.DO /WE
```

assembles the source program in memory and stops at each error (/WE).

```
ASM TEST.DO CAS:SAMPLE /LP
```

assembles the source program and saves it on tape as SAMPLE. The listing is printed on the printer (/LP). Note that there must be a space between the *filename* and the /switch.

```
ASM TEST.DO CAS:
```

assembles the source program and saves it on tape as TEST.

```
ASM TEST.DO CAS: /MR
```

assembles TEST.DO on cassette as a multiple record object code file. Normally, the Model 100 Debug/Assembler uses the same record format as BASIC's CSAVEM command to generate single record object code files on cassette.

Saving programs on cassette as single record files, however, has 2 drawbacks. First, since *all* data must be buffered before it can be written to cassette, buffer requirements may cause you to run out of memory. Second, the single record format doesn't let you generate non-contiguous object code. The /MR multiple record switch lets you overcome both of these restrictions.

> **Note:** The multiple record feature applies only to cassette object files and not to RAM object files. Setting the /MR switch with an object file directed to RAM results in a DEVice Error.

You must load multiple record object files from the Debug/Assembler command level with the L load command. Trying to load a multiple record object file with the CLOADM command results in an OM (Out of Memory) Error.

Controlling the Assembly Listing

The assembler normally displays an assembly listing similar to that on page 13. You can alter this listing with one of these switches:

/NS	No symbol table in the listing
/NL	No listing
/LP	Listing printed on the printer

For example:

```
ASM TEST.DO /NS
```

assembles TEST.DO and shows a listing without the symbol table.

```
ASM SAMPLE.DO /LP
```

assembles SAMPLE.DO and prints the listing on the line printer.

> **Note:** If the Model 100 Debug/Assembler displays an error message during a lengthy assembly listing, you may stop the listing by pressing SHIFT BREAK. This command also terminates the assembly.

Memory Allocation

ZBGASM uses approximately 8.5K of memory starting at 54272 (D400 Hex) and requires additional memory for the symbol table and object code buffer. The memory map (Figure 1) shows the Model 100 memory allocation after the Model 100 Debug/Assembler is loaded.

Available memory is defined as the memory beginning at Himem + 1 and extending up to the start of the symbol table. Memory outside this range is considered "restricted." Any operation that results in a modification of restricted memory causes an

```
MRM?
```

warning message. This message alerts you that your operation modifies restricted memory and asks for confirmation to continue. A "Y" response to the MRM? message allows the change (if the modfication is not in ROM); a CTRL Y response allows the change and cancels further MRM? warnings (if the modification is not in ROM). Any other responses terminate the operation.

MEMORY MAP

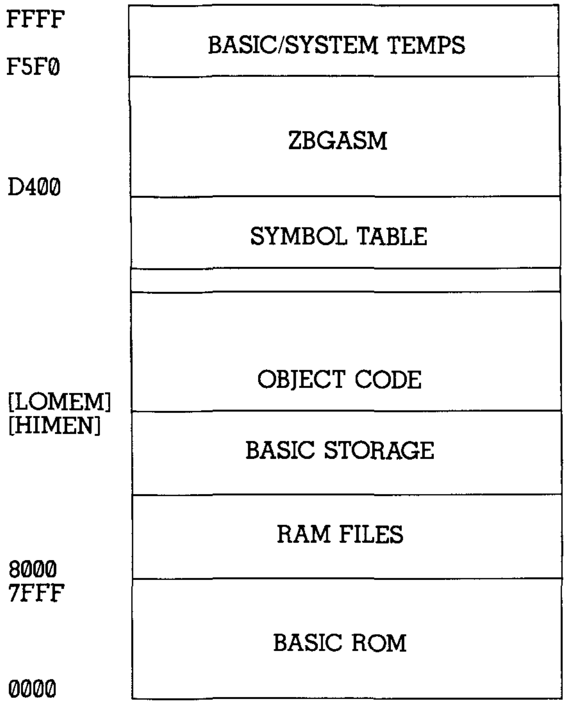


FIGURE 1

Warning: Exercise extreme care when modifying restricted memory. Know exactly what you are modifying and the consequences of the modification. Some changes in restricted memory can result in the loss of all your RAM files.

As shown in the memory map, the symbol table is built down from the start of ZBGASM. The lower portion of available memory (beginning at Himem + 1 and extending to the start of the symbol table) is used for the object code generated by in-memory assemblies and for buffering the object code to be written to a cassette file. Insufficient available memory for the above operations results in an OM (Out of Memory) Error.

You can control the amount of available memory allocated to ZBGASM by using the CLEAR statement to set the BASIC HIMEM parameter. (See your Model 100 owner's manual for details on this procedure.)

The lower the address you assign to Himem, the more available memory you allocate to ZBGASM. Lower Himem addresses, however, decrease the space available for RAM files, and may cause you to overwrite or limit RAM files.

Hints On Assembly

- Use a symbolic name to label the beginning of your program.
- The /WE switch is an excellent debugging tool. Use it to detect assembly errors before debugging the program.
- To examine the symbol table or any other location after an assembly, use ZBUG.

CHAPTER 6/ DEBUGGING THE PROGRAM (ZBUG Commands — Part II)

ZBUG has some powerful tools for a trial run of your assembled program. You can use them to look at each register, every flag, and every memory address during every step of running the program.

Before reading any further, you might want to review the ZBUG commands you learned in Chapter 4. We use these commands here.

Preparing the Program for ZBUG

In this chapter, we use the sample program from Chapter 2 to show how to test a program.

First, load and then enter ZBUG.

Next, assemble SAMPLE.DO into ZBGASM.

When the # prompt appears, you're ready to test the sample program with ZBUG.

Display Modes

In Chapter 4, we discussed 4 examination modes. ZBUG also has 3 display modes.

We'll examine each of these display modes from the mnemonic examination mode. If you're not in this mode, type M (ENTER) to get into it.

Numeric Mode

Type:

```
N (ENTER)
```

and examine the memory addresses that contain your program: CC00H-CC13H.

In the numeric mode, you do not see any of the symbols (labels) in your program (SCREEN, ROW, WAIT, EXIT, etc.). All you see are numbers. For example, Address CC0DH shows the instruction JNZ 0CC03 rather than JNZ SCREEN.

Symbolic Mode

From the command level, type:

```
S (ENTER)
```

and examine your program again. ZBUG displays your entire program in terms of its symbols (SCREEN, WAIT, ROW, CALL, WAIT, etc.). Examine the memory address containing the JNZ SCREEN instruction by typing the memory address of the previous instruction:

```
0CC0BH/
```

Now press (↓). In the symbolic mode, the screen shows your program's memory addresses and instructions as symbols (labels):

```
ROW+5/     JNZ SCREEN
```

Half-Symbolic Mode

At the # prompt, type:

```
H (ENTER)
```

and examine the program. Now all the memory addresses (on the left) are shown as symbols, but the operands (on the right) are shown as numbers.

Using Symbols to Examine Memory

Since ZBUG understands symbols, you can use them in your commands. For example, with ZBUG, both of the following commands open the same memory address regardless of the display mode:

```
HOME/
```

or

```
0CC00H/
```

Both these commands get ZBUG to display your entire program:

```
T HOME EXIT
T CC00 CC1B
```

Note: ZBUG usually directs console output to the LCD screen. Use the **K** (console change) command to redirect a listing (or any other output) to your printer or communications port. (See Reference Section B for details on using the K command.)

Executing the Program

You can run your program from ZBUG using the G (Go) command followed by the program's start address.

Type either of the following:

```
G HOME (ENTER)
G ØCCØØH (ENTER)
```

The program executes, filling all of your screen with a pattern of graphics characters. If you don't get this pattern, the program probably has a "bug." The rest of the chapter discusses program bugs.

Setting Breakpoints

If your program doesn't work properly, you might find it easier to debug it if you break it up into small units and run each unit separately. The X command sets breakpoints at which ZBUG pauses during the execution of a program. From the command level, type:

```
X
```

followed by the address where you want execution to break.

We'll set a breakpoint at the first address that contains the symbol SCREEN: CØØ3H.

Type either of the following:

```
XSCREEN (ENTER)
XCCØ3 (ENTER)
```

Now type GHOME (ENTER) to execute the program. Each time execution breaks, type:

```
C (ENTER)
```

to continue. A graphics character appears on the screen each time ZBUG executes the SCREEN loop. The line:

```
Ø BRK @ SCREEN
```

follows the graphics character to tell you that Breakpoint Ø is set at the SCREEN address. You may set up to 8 breakpoints numbered from Ø-7 in a program.

Type:

```
D (ENTER)
```

to display all the breakpoints you have set.

Type:

```
C1Ø (ENTER)
```

ZBUG continues until the tenth time it encounters that break-point, and then halts execution. Type:

```
Y (ENTER)
```

This is the command to "yank" (delete) all breakpoints. You can also delete a specific breakpoint. For example:

```
YØ (ENTER)
```

deletes the first breakpoint (Breakpoint Ø).

You may not set a breakpoint in a ROM routine. Also, if you set a breakpoint at the point where you call a ROM routine, the C command returns a CONtinuation Error and doesn't let you continue the program.

Examining Registers and Flags

Assemble SAMPLE.DO, and type:

```
R (ENTER)
```

What you see are the contents of every register before the program is executed. (See Chapter 8 for a definition of all the 8085 registers and flags.)

Your display should look something like this:

```
PC   SP   BC   DE   HL   A     F
0000 F569 0000 0000 0000 0000  00=
```

Type:

```
0CC00H;
```

The semicolon (;) executes the instruction at memory address CC00 and displays the next line.

Type ; again to execute the instruction at CC03. Now look at all the registers by typing

```
R (ENTER)
```

The Accumulator (Register A) now contains FF (the ASCII code for a graphics character) from the MOV A,0FFH instruction.

To display only the contents of Register B, type:

```
B/
```

To display the register pair HL, type:

```
HL/
```

Now it contains a 0. You can change registers in the same way you change the contents of memory. With the cursor on the same line as HL/ 0, type:

```
1234 (ENTER)
```

Check the HL Register pair to verify that it contains 1234.

Stepping Through the Program

Type:

```
SCREEN;
```

CALL 422DH is the next instruction to be executed. The first instruction, MVI A,0FFH, has just been executed. To see the next instruction, type:

```
;
```

Now, CALL 422DH has been executed, and LDA 0F639H is the next instruction. Type:

```
;
```

To see that the LDA instruction was executed, type:

```
R (ENTER)
```

Register A now contains the current row position of the cursor. (Memory address F639H is a status location that contains the current row position of the cursor.)

Use the semicolon and R command to continue to single-step through the program examining the registers at will.

You may also single-step through a program using a comma (,) instead of a semicolon. The comma, however, cannot step through a ROM subroutine. For example, type:

```
ROW,
```

Continue stepping through the program by repeatedly pressing comma. When you reach the CALL 4B44H instruction, the screen shows a CONtinuation Error because you are trying to CALL a ROM subroutine.

Transferring a Block of Memory

Type:

```
U  ØCCØØH ØCØØØH 5 ENTER
```

Now the first 5 bytes of your program have been copied to memory address beginning at CØØØ.

Hints on Debugging

- Don't expect your first program to work the first time. Have patience. Most new programs have bugs. Debugging is a fact of life for all programmers, not just beginners.

- Be sure to make a copy of what you have in the edit buffer before executing the program. The edit buffer is not protected from machine language programs.

CHAPTER 7/ USING THE ZBUG CALCULATOR (ZBUG Commands — Part III)

ZBUG has a built-in calculator that performs arithmetic, relational, and logical operations. Also, it lets you use 3 different numbering systems, ASCII characters, and symbols.

This chapter contains examples of how to use the calculator. Some use the same assembled program that we used in the last chapter.

Numbering System Modes

ZBUG recognizes numbers in 3 numbering systems: hexadecimal (Base 16), decimal (Base 1Ø), and octal (Base 8).

Output Mode

The output mode determines which numbering system ZBUG uses to output (display) numbers. From the ZBUG command level, type:

```
O1Ø (ENTER)
```

Examine memory. The T at the end of each number stands for Base 1Ø. Type:

```
O8 (ENTER)
```

Examine memory. The Q at the end of each number stands for Base 8. Type:

```
O16 (ENTER)
```

You're now back in Base 16, the default output mode.

Input Mode

You can change input modes in the same way you change output modes. For example, type:

```
I1Ø (ENTER)
```

Now, ZBUG interprets any number you input as a Base 1Ø number. For example, if you are in this mode and type:

```
T 49152 49162 (ENTER)
```

ZSBUG shows you memory addresses 49152 (Base 1Ø) through 49162 (Base 1Ø). Note that what is printed on the screen is determined by the output mode, not the input mode.

You can use these special characters to "override" your input mode:

BASE	BEFORE NUMBER	AFTER NUMBER
Base 1Ø	&	T
Base 16	#	H
Base 8	@	Q

Table 1. Special Input Mode Characters

For example, while still in the I1Ø mode, type:

```
T 49152 #ØCØ1Ø (ENTER)
```

The "#" overrides the I1Ø mode. ZBUG, therefore, interprets CØ1Ø as a hexadecimal number. As another example, get into the I16 mode and type:

```
T 49152T ØCØ1Ø (ENTER)
```

Here, the "T" overrides the I16 mode. ZBUG interprets 49152 as decimal.

Operations

ZBUG performs many kinds of operations for you. For example, type:

```
ØCØØØ+25T/
```

and ZBUG goes to memory address CØ19 (Base 16), the sum of CØØØ (Base 16) and 25 (Base 1Ø). If you simply want ZBUG to print the results of this calculation, type:

```
ØCØØØ+25T=
```

On the following pages, we use the terms *operands*, *operators*, and *operation*. An operation is any calculation you want ZBUG to solve. In this operation:

```
1 + 2 =
```

"1" and "2" are the operands. "+" is the operator.

Operands

You may use any of these as operands:

1. ASCII characters

2. Symbols

3. Numbers (in either Base 8, 1Ø, or 16).

   Note: ZBUG recognizes integers (whole numbers) only.

Examples (in the O16 output mode):

```
    'A=
```

prints 41, the ASCII code for "A".

```
    HOME=
```

prints the HOME address of the sample program. (It will print UDS Error [Undefined Symbol Error] if you don't have the sample program assembled in memory.)

```
    15Q=
```

prints the hexadecimal equivalent of octal 15.

If you want your results printed in a different numbering system, use a different output mode. For example, get into the O10 mode and try the above examples again.

Operators

You may use arithmetic, relational, or logical operators. (Get into the O16 mode for the following examples.)

Arithmetic Operators

Addition	+
Subtraction	-
Multiplication	*
Division	.DIV.
Modulus	.MOD.
Positive	+
Negative	-

Examples:

```
EXIT-HOME=
```

prints the length of the sample program (not including the END statement).

```
9.DIV.2=
```

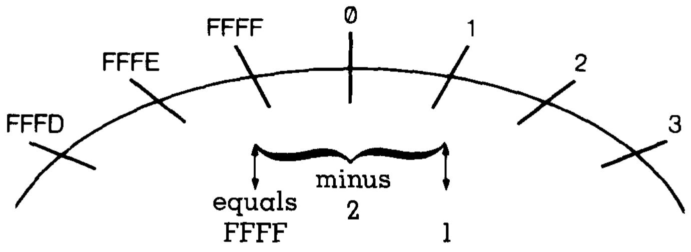
prints 4. (ZBUG can divide integers only.)

```
9.MOD.2=
```

prints 1, the remainder of 9 divided by 2.

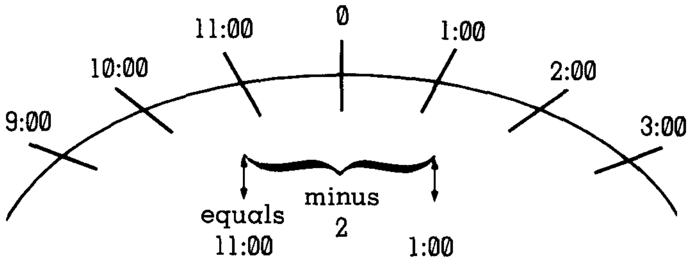
```
1-2=
```

prints OFFFF,65535T, or 177777Q, depending on the output mode. ZBUG does not use negative numbers. Instead, it uses a "number circle" that operates on modulus 10000 (hexadecimal):



Number Circle Illustration of Memory

To understand this number circle, you can use the clock as an analogy. A clock operates on modulus 12 in the same way the ZBUG operates on modulus 10000. Therefore, on a clock, 1:00 minus 2 equals 11:00:



Number Circle Illustration of Clock

Relational Operators

Equal to	.EQU.
Not Equal to	.NEQ.

These operators determine whether a relationship is true or false.

Examples:

```
    5.EQU.5=
```

prints 0FFFF, since the relationship is true. (ZBUG prints 65535T in the O10 mode or 177777Q in the O8 mode.)

```
    5.NEQ.5=
```

prints 0, since the relationship is false.

Logical Operators

Shift	<
LogicalAND	.AND.
InclusiveOR	.OR.
ExclusiveOR	.XOR.
Complement	.NOT.

Logical operators perform bit manipulation on binary numbers.

Examples:

```
10<2=
```

shifts 10 two bits to the left to equal 40. The 8085 RLC instruction also performs this operation.

```
10<-2=
```

shifts 10 two bits to the right to equal 4. The 8085 RRC instruction also performs this operation.

```
6.XOR.5=
```

prints 3, the Exclusive OR of 6 and 5. The 8085 XOR instruction also performs this operation.

Complex Operations

ZBUG calculates complex operations in this order of precedence:

```
* .DIV. .MOD. <
     .AND.
   .OR. .XOR.
      + -
   .EQU. .NEQ.
```

You may use parentheses to change this order.

Examples:

```
4+4.DIV.2=
```

The division is performed first.

```
(4+4).DIV.2=
```

The addition is performed first.

```
4*4.DIV.4=
```

The multiplication is performed first.

PART III/ ASSEMBLY LANGUAGE

This section gives details on 8085 assembly language. It does not explain the 8085 mnemonics, however, since there are many books available on the 8080 family of microprocessors.

CHAPTER 8/ WRITING THE PROGRAM

Chapter 3 gives a general description of assembly language instructions. This chapter describes them in detail.

The 8Ø85 Registers

The 8Ø85 contains 1Ø temporary storage areas (registers) that you may use in your program:

Register A manipulates data and performs arithmetic and logical calculations. It holds 1 byte of data.

Register Pair H-L are used for direct addressing. H stores the most significant byte of an address, Register L stores the least significant byte of an address. This lets the processor directly access an address with the single, least significant byte.

Register Pairs B-C and **D-E** can each hold 2 bytes of data. They are general purpose registers and may be used for temporary data storage. They may be used individually to hold 1-byte data.

Register PC (Program Counter) stores the address of the next instruction to be executed.

> **Note:** The current address of Register PC may be represented in ZBUG by using a **$** or **.** in the operand field.

Register SP (Stack Pointer) is a 16-bit register that points to the memory stack. Addresses are stored in the stack in a "last-in, first-out" manner. The last address put into the stack should be the first address to be pulled.

Status Register F is the Flag register for testing conditions and setting interrupts. It consists of 5 "flags." Many mnemonics "set" or "clear" 1 or more of these flags. Others test to see if a certain flag is set or clear.

This is the meaning of each flag, if set:

Bit 0
CY (Carry) — an 8-bit arithmetic operation caused a carry or borrow from the most significant bit.

Bit 1
Reserved

Bit 2
P (Parity) — the result of the previous operation has an even number of bits set (even parity).

Bit 3
Reserved

Bit 4
AC (Auxiliary Carry or Half Carry) — an 8-bit addition operation caused a carry.

Bit 5
Reserved

Bit 6
Z (Zero) — the result of the previous operation is zero.

Bit 7
S (Sign) — the result of the previous operation is a negative number

Assembly Language Fields

You may use 4 fields in an assembly language instruction: label, command, operand, comment. In this instruction:

```
HOME:  MOV   A,M          ;MOVES MEMORY DATA TO
                            REG A
```

HOME is the label. MOV is the command. A,M is the operand. MOVES MEMORY DATA TO REG A is the comment.

The comment is solely for your convenience. The assembler ignores comments after a semicolon.

The Label

You can use a symbol in the label field to define a memory address or data. The above instruction uses HOME to define its memory address.

Once the address is defined, you can use HOME as an operand in other instructions. For example:

```
JNZ    HOME
```

branches to the memory address defined by HOME.

The assembler stores all the symbols, with the addresses or data they define, in a "symbol table," rather than as part of the "executable program." The symbol can be a maximum of 6 characters.

The Command

The command can be either a pseudo op or a mnemonic.

Pseudo ops are commands to the assembler. The assembler does not translate them into opcodes and does not store them with the executable program. For example:

```
NAME:           EQU             43
```

defines the symbol NAME as 43. The assembler stores this in its symbol table.

```
                ORG             0CC00H
```

is a pseudo op that tells the assembler to begin the executable program at Address CC00.

```
SYMBOL:         DC              6
```

stores 6 in the current memory address and labels this address SYMBOL. The assembler stores this information in its symbol table.

Mnemonics are commands to the processor. The assembler translates them into opcodes and stores them with the executable program. For example:

```
                XCHG
```

tells the processor to exchange the data in the H-L Register Pair with the D-E Register Pair. The assembler assembles this into opcode number EBH and stores it with the executable program.

The next chapter shows how to use pseudo ops. *Reference G* lists the 8085 mnemonics.

The Operand

The operand is either a memory address or data. For example:

```
LXI BC, 3000+COUNT
```

loads Register Pair B-C with 3000H plus the value of COUNT. The operand, 3000+COUNT, specifies a data constant.

The assembler stores the operand with its opcode. Both are stored with the executable program.

Operators

The plus sign (+) in the above operand (3000+COUNT) is called an operator.

You can use any operator described in *Chapter 7,* "Using the ZBUG Calculator," as part of the operand.

Addressing Modes

8085 mnemonics have 4 addressing modes that let you address data in different ways.

1. Immediate Addressing

The operand is *data*. For example:

```
                    ADI          30H
```

adds the value 30H to the contents of Register A.

```
    DATA:           EQU          8004H
                    LXI          BC,DATA
```

loads the value 8004 into Register Pair BC.

```
                    CPI          94H
```

compares the contents of Register A with the hexadecimal value 94.

2. Direct Addressing

The operand is the exact memory *address* of the data. In this mode of addressing, you address a specific memory location.

For example:

```
                    JNZ          0CC00H
```

jumps to Address CC00.

```
    SPOT:           EQU          0CC00H
                    SHLD         SPOT
```

stores the contents of Register Pair HL in Address CC00.

If the instruction calls for data, the operand contains the address where the data is stored.

```
                    LHLD         0CC00H
```

does not load Register Pair HL with CC00. The processor loads HL with whatever data is in Address CC00. If 65H is stored in Address CC00, Register Pair HL is loaded with 65H.

```
ADD M
```

adds whatever data is stored in the Address in the HL Register Pair to the contents of Register A.

3. Register Indirect Addressing.

The operand is the HL Register Pair that contains the *address* in memory that contains the data.

In understanding this mode, think of a treasure hunt game. The first instruction is "Look in the suitcase." The suitcase contains the second instruction, "Look in the mailbox."

Examples:

```
MOV A,M
```

moves the contents of the memory location whose address is in Register pair HL. M refers to the memory address currently stored in Register Pair HL.

The following example shows the complete process:

```
LXI         HL,0CC00H
MVI         M,0FFH
INX         H
MVI         M,0FFH
.
.
LHLD        0CC00H
MOV         A,M
```

The LXI instruction loads address CC00 into the H-L Register Pair, and MVI moves the data FF into Register L. The INX instruction increments the address in the H-L Register Pair by 1, and MVI moves the data FF into memory location CC01.

Later in the program, you can access the data FFFF by loading memory location CC00 into the H-L Register Pair and then moving the data into the accumulator.

This is a good mode of addressing to use when calling ROM routines.

4. Register Addressing

This addressing mode lets you access data by specifying registers or register pairs containing the data. The operand is always a register.

For example:

```
MOV     A,B
```

transfers the data in Register B to Register A. Now both of the registers contain the same data.

Register pairs may also be addressed:

```
DAD     BC
```

adds the data in Register Pair BC to the data in Register Pair HL. The result is placed in HL.

CHAPTER 9/ USING PSEUDO OPS

As discussed earlier, pseudo ops direct the assembler. You can use them to:

- Control where the program is assembled
- Define symbols
- Insert data into the program
- Change the assembly listing
- Do a "conditional" assembly
- Include another source file in your program

Pseudo ops are unique to the assembler you are using. Other 8Ø85 assemblers may not recognize the ZBUG pseudo ops.

The ZBUG pseudo ops make it easier for you to program. This chapter shows how to use pseudo ops.

Controlling Where the Program is Assembled

ZBUG has two pseudo ops that control where the program is assembled:

- ORG, sets the first location
- END, ends the assembly

ORG
ORG *expression*

Tells the assembler to begin assembling the program at *expression*. Example:

```
        ORG                  ØCCØØH
```

tells the assembler to start assembling the program at Address CCØØ.

You can put more than one ORG command in a program. When the assembler arrives at the new ORG, it begins assembling at the new *expression*.

Note: **$** or **.** may be used as an operand to return the current location of the program counter. ORG $+50, for example, opens 50 bytes between the present location and the next assembled location.

END
END ***expression***

Tells the assembler to quit assembling the program. The *expression* option lets you store the program's start address. Use END as the last instruction in all your assembly language programs.

Example:

```
                  ORG           0CC00H
HOME:             MOV           A,M
                  .
                  .
DATA:             DC            'This is data'
                  END           HOME
```

The END pseudo op quits the assembly and stores the program's entry address (the value of HOME) in memory. When you load the program, the processor knows to start executing at HOME (the LDA instruction) rather than at DATA (the DC instruction).

DC is a pseudo op explained later in this chapter.

Defining Symbols

Symbols make it easy to write a program and also make the program easy to read and revise. The ZBUG has 2 pseudo ops for defining symbols:

- EQU, for defining a constant value
- SET, for defining a variable value

EQU
symbol **EQU** ***expression***

Equates *symbol* to *expression*. Examples:

```
CHAR:          EQU        F9H
```

makes every use of CHAR equal to F9H.

```
KYREAD:        EQU        7242H
               CALL       KYREAD
```

equates KYREAD to 7242H. The next instruction calls the ROM routine at memory address 7242H.

EQU helps set the values of constants. You can use it anywhere in your program.

SET

symbol* SET *expression

Sets *symbol* equal to *expression*. You can use SET to reset the symbol elsewhere in the program. Example:

```
SYMBOL:        SET        25
```

sets SYMBOL equal to 25. Later in the program, you can reset SYMBOL.

```
SYMBOL: SET   SYMBOL+COUNT
```

Now SYMBOL equals 25+COUNT.

Inserting Data into Your Program

ZBUG has 4 pseudo ops that make it simple for you to reserve memory and insert data in your program:

- DS, for reserving areas of memory for data
- DB, for inserting 1 byte of data in memory
- DW, for inserting 2 bytes of data in memory
- DC, for inserting a string of data in memory

Remember that the processor cannot "execute" a block of data in your program. If you use these pseudo ops:

- Use them at the end of your program (just before the END instruction), or
- Precede them with an instruction that jumps or branches to the next 'executable' instruction.

DS

symbol* DS *expression

(Define Storage) Reserves *expression* bytes of memory for data. Example:

```
    BUFFER:          DS          200
```

reserves 200 bytes for data, starting at Address BUFFER.

```
    DATA:            DS          6+SYMBOL
```

reserves 6+SYMBOL bytes for data beginning at Address DATA.

DB

symbol* DB *expression

(Define Byte) Stores a 1-byte *expression* in memory at the current address. The *symbol* is optional.

Examples:

```
    DATA:            DB          33H
```

stores 33H in Address DATA.

```
    FACTOR:          DB          NUM/2
                     LDA         FACTOR
```

stores NUM/2 in Address FACTOR and then loads NUM/2 into Register A.

DW

symbol* DW *expression

(Define Word) Stores a 2-byte *expression* in memory starting at the current address. The symbol is optional. Example:

```
    DATA:            DW          1100H
```

stores 1100H in Address DATA and DATA + 1.

DC

symbol* DC *delimiter string delimiter

Stores an ASCII string in memory, beginning at the current address. The *symbol* is optional. The *delimiter* can be any character.

Examples:

```
    TABLE:          DC          /THIS IS A
                                STRING/
```

stores the ASCII codes for THIS IS A STRING in memory locations, beginning with TABLE.

```
                    .
                    .
    INIT:           LXI         H,NAME
                    MVI         B,4
                    .
                    .
    DISP:           MOV         A,M
                    CALL        LCD
                    INX         H
                    DCR         B
                    JNZ         DISP
                    .
                    .
    NAME:           DC          'GARY'
                    DB          ØDH
```

The first instruction loads NAME into Register Pair HL. The pseudo op DC at the end of the program stores "Gary" in the four memory addresses beginning with NAME. The DISP routine processes this data.

PART IV/ ROM ROUTINES

In an assembly language program, the simplest way to use the I/O devices is with ROM routines.

This section shows how. A complete list of ROM routines is in Reference Section E.

CHAPTER 10/ USING THE KEYBOARD AND VIDEO DISPLAY (ROM ROUTINES)

The Model 100 uses its own machine-code routines to access the screen, keyboard, and tape. These routines are built into the computer's ROM. You can use the same routines in your own program.

Appendix F lists each ROM routine and the ROM address that points to it. This chapter uses 2 of these routines, CHGET and LCD, as samples in showing the steps for using ROM routines.

Steps for Calling ROM Routines

We recommend these steps for calling a ROM routine:

1. Equate the routine's address to its name. This lets you refer to the routine by its name rather than its address, making your program easier to read and revise.
2. Set up any entry conditions required by the routine. This lets you pass data to the routine.
3. Preserve the contents of the registers. Since many routines change the contents of the registers, you might want to store the registers' contents temporarily before jumping to the routine.
4. Call the ROM routine, using the indirect addressing mode.
5. Use any exit conditions that the routine passes back to your program.
6. Restore the contents of the registers (if you temporarily preserved them in Step 3).

Sample: Keyboard Input with CHGET

This short program uses 4 ROM subroutines to (1) "home" the cursor, (2) monitor the keyboard, (3) display characters, and (4) exit to the menu. Each key you press is displayed until you press (RETURN) which returns you to the main menu.

CHGET monitors the keyboard and waits for you press a key. When you do press a key, CHGET loads Register A with the key's ASCII code. LCD then displays the contents of Register A on the screen.

```
        ORG     0CC00H
HOME:   EQU     422DH       ;ROM routine homes cursor
CHGET:  EQU     12CDH       ;ROM routine waits for a key
LCD:    EQU     4B44H       ;ROM routine displays character
EXIT:   EQU     5797H       ;ROM exit routine
BEGIN:  PUSH    PSW         ;store Registers A and F data
LOOP:   CALL    CHGET
        CALL    LCD
        CPI     0DH         ;Look for a carriage return
        JNZ     LOOP        ;If no <ENTER> loop to LOOP
        POP     PSW         ;restore Registers A and F data
        CALL    EXIT
        END     BEGIN
```

This is how we applied the above steps in writing this program:

1. Equate ROM routines to their Addresses

This equates CHGET to 12CBH, the address that points to CHGET's address:

```
CHGET:          EQU             12CBH
```

2. Set Up Entry Conditions

HOME, CHGET, and EXIT have no entry conditions. LCD requires that Register A contains the character to be displayed. CHGET fulfills the entry requirement for LCD.

3. Preserve the Registers' Contents

Only Registers A and F (Carry Flag) are affected by the program. This preserves the contents of Register A and all the condition flags:

```
    BEGIN:              PUSH            PSW
```

This instruction "pushes" the value of Register A into a memory location 1 less than the SP Register, and pushes the condition flags (Register F) into a memory location 2 less than the SP Register.

4. Call ROM routines

This CALLs CHGET:

```
    LOOP:               CALL            CHGET
```

This displays the character you typed onto the LCD screen:

```
                        CALL LCD
```

5. Use Exit Conditions

After the CHGET subroutine is executed, Register A contains a character. That character is compared to a carriage return:

```
                        CPI             0DH
```

The above instruction branches back to LOOP (the JNZ LOOP instruction) if you do not press RETURN. (Pressing RETURN calls the EXIT subroutine.

6. Restore the Register's Contents

This "pops" (restores) the contents of Registers A and F:

```
                        POP             PSW
```

Now, the above registers are restored to the data they contained before executing the CHGET routine.

7. Exit the Program

This CALLs the Exit Routine that lets you properly exit the program and return to the Model 100's main menu screen:

```
                        CALL            EXIT
```

PART V/ REFERENCE

This section summarizes all the features of the Model 100 Debug/Assembler.

REFERENCE A/ ASSEMBLER COMMANDS AND SWITCHES

ASM *source filename object filename /switch/. . .*
Assembles the source program into machine code in memory. Source files must be RAM files generated by the Text Editor (TEXT) and therefore always carry the **.DO** extension. The source filename is always required. If the object code filename is omitted, the object code is written directly to memory and no object file is created.

ASM *source filename* CAS:*object filename /switch/. . .*
Assembles the source program into machine code in the cassette. When the device specifier **CAS:** precedes the object filename, the object file is created and written to the cassette recorder. If the object filename is omitted, the source filename is assigned to the object file.

The assembler switches are:

/LP	Assembly listing on the printer.
/MR	Multiple record object code file
/NL	No listing printed.
/NO	No object code generated.
/NS	No symbol table generated.
/WE	Wait on assembly errors.

Examples:

```
ASM SAMPLE.DO SAMPLE
ASM TEST.DO CAS:TEST /WE
ASM TEST.DO /NO/NS
```

REFERENCE B/ ZBUG COMMANDS

The following list defines the terms used in this section:

expression
One or more numbers, symbols, or ASCII characters. If more than one is used, you may separate them with these operators:

Multiplication	*	Addition	+
Division	.DIV.	Subtraction	-
Modulus	.MOD.	Equals	.EQU.
Shift	<	Not Equal	.NEG.
Local And	.AND.	Positive	+
Exclusive Or	.XOR.	Negative	-
Logical Or	.OR.	Complement	.NOT.

address
A location in memory. This may be specified as an expression using numbers or symbols.

filename
A BASIC cassette file specification.

Commands

C
CONTINUE FROM BREAKPOINT C *expression* (ENTER)
Continues execution of the program after interruption at a breakpoint. The expression specifies the proceed count, which is the number of times (-1) that the breakpoint will be passed before a break occurs. Numbers in the expression for the proceed count are interpreted as decimal numbers regardless of the input radix setting.

C(ENTER)
Continue by increments of 1.

D
DISPLAY BREAKPOINTS D (ENTER)
Displays all breakpoints that have been set.

G

GO EXECUTE G *expression* ENTER

Executes the program beginning at the address specified by the *expression*. The expression is required.

K

CONSOLE CHANGE K ENTER

The Debug/Assembler usually directs console output to the LCD. The K command allows you to direct the console output to any of 3 devices—screen, printer, communications—by responding to the **LCD=0 LPT=1 COM=2 CNS=** prompt with the appropriate number.

L

LOAD A MACHINE LANGUAGE FILE L *filename* ENTER

Loads a machine language file *filename* from RAM files. Using CAS: to specify the cassette recorder—L CAS:*filename* ENTER—loads a machine language file *filename* from cassette.

CL

CASSETTE LOAD A MACHINE LANGUAGE FILE

CL *filename* ENTER

Loads machine language file *filename* from cassette.

P

PUNCH *filename start address end address execution address*

Saves memory from *start address* to *end address* in a RAM file. You must also specify an *execution address*, the first address to be executed when the file is loaded. Using CAS: to specify the cassette recorder—P CAS:*filename start address end address execution address*—saves memory from *start address* to *end address* in cassette.

CP

CASSETTE PUNCH *filename start address end address execution address*

Saves memory to cassette from *start address* to *end address*. You must also specify an *execution address*, the first address to be executed when the file is loaded.

R
REGISTER DISPLAY R (ENTER)
Displays the contents of all the registers.

T
TYPE OUT T *address1 address2* (ENTER)
Displays the memory locations from *address1* to *address2*, inclusive. Uses the current Examination and Type Out Modes.

U
BLOCK TRANSFER U *source address destination address byte count*
Transfers the contents of memory beginning at *source address* and continuing for *count* bytes to another location in memory beginning with *destination address.*

X
SET BREAKPOINT X *address* (ENTER)
Sets a breakpoint at *address*. If *address* is omitted, the current location is used. Each breakpoint is assigned a number from 0 to 7. The first breakpoint set is assigned as Breakpoint 0. A maximum of eight breakpoints may be set at one time.

Y
YANK BREAKPOINT Y *n* (ENTER)
Deletes the breakpoint referenced by the *n* number (0-7). If *n* is omitted, all breakpoints are deleted.

Z
DELETE SYMBOL TABLE Z (ENTER)
Deletes the current symbol table.

Examination Mode Commands

A (ENTER)	ASCII Mode
B (ENTER)	Byte Mode
M (ENTER)	Mnemonic Mode
W (ENTER)	Word Mode

(The default is M)

Display Mode Commands

H (ENTER)	Half Symbolic
N (ENTER)	Numberic
S (ENTER)	Symbolic

(The default is S)

Numbering System Mode Commands

O*base*	Output
I*base*	Input

(Base can be 8, 1Ø, or 16. The default is 16)

Special Symbols

/
OPEN A LOCATION *address/*
Opens the specified memory location. Opens the current location if *address* is omitted.

(↓)
OPEN NEXT LOCATION (↓)
Opens next *address* and allows you to enter any change.

(↑)
OPEN PRECEDING LOCATION (↑)
Opens preceding *address* and allows you to enter any changes.

***address* (ENTER)**
register
OPEN A REGISTER AT AN ADDRESS *address/ register/*
Opens *address* of *register* and displays its contents. If *address* is omitted, the register at the last address is reopened. After the contents have been displayed, you may type:

new value	To change the contents.
(ENTER)	To close and enter any change.

,

SINGLE STEP EXECUTION *address,*

Executes the instruction at a specific memory address. A , without an address executes the instruction at the current setting of the Program Counter (PC).

;

SINGLE STEP EXECUTION—SKIP SUBROUTINES ;

Executes the instruction at the current setting of the Program Counter, but skips all subroutines except CALL. When ; encounters a CALL, it executes the entire subroutine.

:

FORCE FLAGS :

Opens the current location, forcing the flags mode. The colon does not actually have anything to do with the (status flag) Register F. It simply interprets the contents of the given address *as if* it contained flag bits.

=

EVALUATE THE EXPRESSION *expressions=*

Opens the current memory location and forces the numeric type out mode and byte examination mode to evaluate the expression.

REFERENCE C/ DEBUG/ASSEMBLER ERROR MESSAGES

These are error messages you can get while in ZBGASM:

BAS BASIC Error (General)
BASIC Errors result from ZBUG's calls to the BASIC ROM. The BASIC error code for these errors is stored at address START + 3 (where START is the start of ZBGASM).

BP BAD BREAKPOINT (ZBUG)
You are attempting to set a breakpoint (1) greater than 7, (2) in ROM, (3) at a SWI command, (4) at an address where one is already set.

BTO BYTE OVERFLOW (Assembler)
There is a field overflow in an 8-bit data quantity.

CMD COMMAND ERROR (ZBUG)
You are not using a ZBUG command.

CON CAN'T CONTINUE (ZBUG)
The system doesn't understand a command and cannot continue.

DEV BAD DEVICE (General)
The device you are trying to access does not exist.

END MISSING END STATEMENT (Assembler)
END must be the final instruction on any source assembly language program.

EXP EXPRESSION ERROR (General)
Either the syntax for the expression is incorrect or the expression is dividing by zero.

FF FILE NOT FOUND (General)
The file is not in a RAM file or on cassette tape.

FLG LOST FLAGS (ZBUG)
A breakpoint ZBUG breakpoint path has caused the flag register to be lost.

INF MISSING INFORMATION (Assembler)
(1) There is a missing delimiter in a pseudo op or (2) there is no label on a SET or EQU pseudo op.

IO	I/O ERROR (General)

Input/Output error. A checksum error was encountered while loading a file from a cassette tape. The tape may be bad, or the volume setting may be wrong. Try a higher volume.

LBL	LABEL ERROR (Assembler)

The symbol you are using is (1) not a legal symbol, (2) not terminated with either a space, a tab, or a carriage return, (3) has been used with ORG or END, which do not allow labels, or (4) longer than six characters.

LTL	LINE TOO LONG (Assembler)

Your assembler instruction line is too long.

MDS	MULTIPLE DEFINED SYMBOL (Assembler)

Your program has defined the same symbol with different values.

MEM	BAD MEMORY (General)

A bad memory error can occur during ZBUG memory modification or during an in-memory assembly in the Assembler. It indicates that the data did not store correctly in the memory location.

MRM	MODIFYING RESTRICTED MEMORY (General)

This error warns you that you are either trying to alter ROM, or you are about to overwrite a file in RAM.

NM	BAD OR MISSING FILENAME (General)

File integrity has been breached and data cannot be accessed.

OM	OUT OF MEMORY (General)

You have run out of available memory and may not continue your operation.

OPC	OP CODE ERROR (Assembler)

The op code is either not valid or is not terminated with a space, tab, or carriage return.

OPN	OPERAND ERROR (Assembler)

There is some syntax error in the operand field.

PRM BAD PARAMETERS (ZBUG)
Usually this means your command line has a syntax error, or you have specified a filename that has more than eight characters.

UDS UNDEFINED SYMBOL (General)
Your program has not defined the symbol being used. The operand field has a symbol that does have a corresponding symbol in the label field.

VFY VERIFY ERROR (General)
An error was detected in a file saved to tape.

REFERENCE D/ ASSEMBLER PSEUDO OPS

The following list defines the terms used in this section:

symbol
Any string from one to six characters long, typed in the symbol field.

expression
Any expression typed in the operand field. See *Reference B*, "ZBUG commands," for a definition of valid expressions.

END *expression*
Ends the assembly. The optional *expression* specifies the start address of the program.

symbol* EQU *expression
Equates *symbol* to an *expression.*

```
SYMBOL:     EQU         0D000H
```

symbol* DB *expression, . . .
Stores a 1-byte *expression* beginning at the current address.

```
DATA2:      DB          33H+COUNT
```

symbol* DC *delimiter string delimiter
Stores *string* in memory beginning with the current address. The *delimiter* can be any character.

```
TABLE:      DC          /THIS IS A STRING/
```

symbol* DW *expression
Stores a 2-byte expression in memory beginning at the current address.

```
DATA:       DW          0D000H
```

ORG *expression*
Originates the program at *expression* address.

```
                ORG         0CC00H
```

DS *expression*
Reserves *expression* bytes of memory for data.

```
    DATA:       DS          06H
```

symbol* SET *expression
Sets or resets *symbol* to *expression*.

```
    SYMBOL:     SET         0DD00H
```

REFERENCE E/ ROM SUBROUTINES

This reference lists the indirect addresses where the Model 100's ROM routines are stored. It also shows the entry and exit conditions for each routine.

The name of the routine is for documentation only. To jump to the routine, you must use its indirect address.

LCD Functions

LCD
Displays a character on the LCD at current cursor position.

Entry Address (Hex): 4B44
Entry conditions: A = character to be displayed
Exit conditions: None

PLOT
Turns on pixel at specified location.

Entry Address (Hex): 744C
Entry conditions: D = *x* coordinate (Ø-239)
E = *y* coordinate (Ø-63)
Exit conditions: None

UNPLOT
Turns off pixel at specified location.

Entry Address (Hex): 744D
Entry conditions: D = *x* coordinate (Ø-239)
E = *y* coordinate (Ø-63)
Exit conditions: None

POSIT
Gets current cursor position.

Entry Address (Hex): 427C
Entry condition: None
Exit conditions: H = column number (1-4Ø)
I = row number (1-8)

ESCA
Sends specified Escape Code Sequence.

Entry Address (Hex): 4270
Entry conditions: A = escape code
Exit conditions: None

LCD Functions and Escape Codes

The routines for generating common LCD functions and escape codes have no entry or exit parameters.

Routine	Function	Entry Address (Hex)	Equiv. ESC
CRLF	Generates a Carriage Return and Line Feed	4222	—
HOME	Moves cursor to Home Position (1,1)	422D	—
CLS	Clears Display	4231	—
SETSYS	Sets system line (lock line 8)	4235	T
RSTSYS	Resets system line (unlock line 8)	423A	U
LOCK	Locks display (no scrolling)	423F	Y
UNLOCK	Unlocks display (scrolling)	4244	W
CURSON	Turns on cursor	4249	P
CUROFF	Turns off cursor	424E	Q
DELLIN	Deletes line at current cursor position	4253	M
INSLIN	Inserts a blank line at cursor position	4258	L
ERAEOL	Erases from cursor to end of line	425D	K

ENTREV	Sets Reverse character mode	4269	p
EXTREV	Turns off Reverse character mode	426E	q

LCD Variable and Status Locations

Name	**Contents**	**Memory Location**
CSRY	1Cursor Position (ROW)	F639
CSRX	Cursor Position (Column)	F63A
BEGLCD	Start of LCD memory	FE00
ENDLCD	End of LCD memory	FF40

Keyboard Functions

KYREAD
Scans keyboard for a key and returns with or without one.

Entry Address (Hex): 7242
Entry conditions: None
Exit conditions: A = Character, if any

Z Flag: Set if no key found,
reset if key found
Carry: Set (character in code table below), reset (normal character set code)

When Carry is set (1), Register A will contain one of the following:

Register A	*Key Pressed*
Ø	F1
1	F2
2	F3
3	F4
4	F5
5	F6
6	F7
7	F8
8	LABEL
9	PRINT
ØA	SHIFT-PRINT
ØB	PASTE

CHGET
Waits and gets character from keyboard.

Entry Address (Hex): 12CB
Entry conditions: None
Exit conditions: A = character code
Carry: Set if special character, reset if normal character
(F1 - F8 return preprogrammed strings)

CHSNS
Checks keyboard queue for characters.

Entry Address (Hex): 13DB
Entry conditions: None
Exit conditions: Z flag: Set if queue empty, reset if keys pending

KEYX
Checks keyboard queue for characters or BREAK.

Entry Address (Hex): 7270
Entry conditions: None
Exit conditions: Z flag set if queue empty, reset if keys pending

Carry: Set when BREAK entered,
Reset with any other key

BRKCHK
Checks for BREAK characters only (CTRL C or CTRL S).

Entry Address (Hex): 7283
Entry conditions: None
Exit conditions:
Carry: Set if BREAK or PAUSE entered,
reset if no BREAK characters

INLIN
Gets line from keyboard. Terminated by (ENTER).

Entry Address (Hex): 4644
Entry conditions: None
Exit conditions: Data stored at location F685

Using Function Key Routines

The function table consists of character strings to be used by the keyboard driver when processing F1 - F8 keys. The strings have a maximum length of 16 characters and are terminated by an 80 (hex) code. If the last character of the string is ORed with 80, the character will also serve as a terminator. The entire string will be placed in the keyboard buffer when the appropriate function key is pressed. You must specify character strings for all 8 function keys. (Use the terminator byte for any string you wish to ignore.)

Example of function table:

```
FCTAB   DEFM   'Files'   ;F1
        DEFW   0D80
        DEFM   'Load'    ;F2
        DEFB   80
        DEFM   'Save'    ;F3
        DEFB   80
        DEFM   'Run'     ;F4
        DEFW   0D80
        DEFM   'List'    ;F5
        DEFW   0D80
        DEFB   80        ;Ignore F6
        DEFB   80        ;Ignore F7
        DEFM   'Menu'    ;F8
        DEFW   0D80
```

STFNK
Sets function key definitions.

Entry Address (Hex): 5A7C
Entry conditions:
HL = Address of function table (above)
Exit conditions: None

CLRFLK
Clears function key definition table (fills table with 80s).

Entry Address (Hex): 5A79
Entry conditions: None
Exit conditions: None

DSPFNK
Displays function keys.

Entry Address (Hex): 42A8
Entry conditions: None
Exit conditions: None

STDSPF
Sets and displays function keys.

Entry Address (Hex): 42A5
Entry conditions:
HL = Start address of function table
Exit conditions: None

ERAFNK
Erases function key display.

Entry Address (Hex): 428A
Entry conditions: None
Exit conditions: None

FNKSB
Displays function table (if enabled).

Entry Address (Hex): 5A9E
Entry conditions: None
Exit conditions: None

Printing Routines

PRINTR
Sends a character to the line printer.

Entry Address (Hex): 6D3F
Entry conditions: A = character to be printed
Exit conditions:
Carry: Set if cancelled by BREAK,
reset if normal return

PNOTAB
Prints character without expanding tab characters.

Entry Address (Hex): 1470
Entry conditions: A = character to be printed
Exit conditions: Unknown

PRTTAB
Prints a character expanding tabs to spaces.

Entry Address (Hex): 4B55
Entry conditions: A = character to be printed
Exit conditions: Unknown

PRTLCD
Prints contents of LCD.

Entry Address (Hex): 1E5E
Entry conditions: None
Exit conditions: None

RS232-C and Modem Routines

DISC
Disconnects phone line.

Entry Address (Hex): 52BB
Entry conditions: None
Exit conditions: None

CONN
Connects phone line.

Entry Address (Hex): 52D0
Entry conditions: None
Exit conditions: None

DIAL
Dials a specified phone number.

Entry Address (Hex): 532D
Entry conditions: HL = phone number address
Exit conditions: None

RCVX
Checks RS232 queue for characters.

Entry Address (Hex): 6D6D
Entry conditions: None
Exit conditions: A = number of characters queue
Z flag: Set if no data, reset if characters pending

RV232C
Gets a character from RS232 receive queue.

Entry Address (Hex): 6D7E
Entry conditions: None
Exit conditions: A = character received
Z flag: Set if O.K., reset if error (PE,FF, or OF)
Carry: Set if BREAK pressed, else reset

SENDCQ
Sends an XON resume character (CTRL Q).

Entry Address (Hex): 6E0B
Entry conditions: None
Exit conditions: None

SENDCS
Sends an XOFF pause character (CTRL S).

Entry Address (Hex): 6E1E
Entry conditions: None
Exit conditions: None

SC232C
Sends a character to the RS-232 or Modem (with XON/OFF).

Entry Address (Hex): 6E32
Entry conditions: A = character to be sent
Exit conditions: Unknown

CARDET
Detects carrier (for modem only).

Entry Address (Hex): 6EEF
Entry conditions: None
Exit conditions: A = 0 if carrier
Z Flag: Set if carrier, else reset

BAUDST
Sets baud rate for RS232-C.

Entry Address (Hex) 6E75
Entry conditions: H = Baud rate (1-9,M)
Exit conditions: None

INZCOM
Initializes RS232-C and Modem.

Entry Address (Hex): 6EA6
Entry conditions:
H = Baud rate (1-9,M)
L = UART configuration code
(See UART byte description below)
Carry: Set if RS232-C, reset if modem
Exit conditions: None

Bit(s)	Description	
0	Number of Stop Bits	:0=1,1=2
1	Parity setting	:0 = Odd 1 = Even
2	Parity disable	:0 = Enable 1 = Disable
3-4	Word length	:01 = 6, 10 = 7, 11 = 8

The byte is ANDed with 1FH to ignore bits 5-7. The text string containing the current STAT setting is located at F65BH (5 bytes): baud, length, parity, stop bits, and XON/XOFF switch.

SETSER
Sets serial interface parameters. Activates RS232-C/Modem.

Entry Address (Hex): 17E6
Entry conditions: HL = start address of ASCII string containing parameter terminated by zero (78E1E,0).
Syntax same as in Telcom's STAT
Carry: Set for RS232-C, reset for Modem
Exit conditions: None

CLSCOM
Deactivates RS232-C/Modem.

Entry Address (Hex): 6ECB
Entry conditions: None
Exit conditions: None

Cassette Recorder Routines

DATAR
Reads character from cassette (no checksum).

Entry Address (Hex): 702A
Entry conditions: None
Exit conditions: D = character from cassette

CTON
Turns on motor.

Entry Address (Hex): 14A8
Entry conditions: None
Exit conditions: None

CTOFF
Turns off motor.

Entry Address (Hex): 14AA
Entry conditions: None
Exit conditions: None

CASIN
Reads a character from cassette and updates checksum.

Entry Address (Hex): 14B0
Entry conditions: C = current checksum
Exit conditions: A = character
C = contains the updated checksum

CSOUT
Sends character to cassette and updates checksum.

Entry Address (Hex): 14C1
Entry conditions: A = character to be sent
C = current checksum
Exit conditions: C = updated checksum

SYNCW
Writes cassette header and sync byte only.

Entry Address (Hex): 6F46
Entry conditions: None
Exit conditions: None

SYNCR
Reads cassette header and sync byte only.

Entry Address (Hex): 6F85
Entry conditions: None
Exit conditions: None

DATAW
Writes a character to cassette (no checksum).

Entry Address (Hex): 6F5B
Entry conditions: A = character to be sent
Exit conditions: None

The directory table (F962) contains information on all file location, type, and status.

Each file is managed by an 11-byte directory entry in the format:

Byte 1	**:**	**Directory Flag (for file type and status)**
Bytes 2-3	**:**	**Address of file**
Bytes 4-11	**:**	**8-byte filename**

The Directory Flag contains the following information:

Bit 7 (MSB)	**1 if a valid entry**
Bit 6	**1 for ASCII text file (DO)**
Bit 5	**1 for machine language (CO)**
Bit 4	**1 for ROM file**
Bit 3	**1 for invisible file**
Bit 2	**reserved for future use**
Bit 1	**reserved for future use**
Bit 0	**internal use only**

MAKTXT
Creates a text file.

Entry Address (Hex): 220F
Entry conditions: Filename (max. 8 bytes) must be stored in FILNAM (FC93).'DO' extension not required.
Exit conditions: HL = TOP address of new file
DE = address of Directory entry (Flag)
Carry: Set if file already exists
Reset if new file

CHKDC
Searches for file in directory.

Entry Address (Hex): 5AA9
Entry conditions: DE = address of filename to find (ASCII filename + 0 byte terminator)
Exit conditions: HL = start address (TOP) of file
Z Flag: 0 (file found)
1 (file not found)

GTXTTB
Gets top address of file.

Entry Address (Hex): 5AE3
Entry conditions: HL = address of directory entry for file
Exit conditions: HL = TOP start address of file

KILASC
Kills a text (DO) file.

Entry Address (Hex): 1FBE
Entry conditions: DE = file TOP start address
HL = address of directory entry (flag)
Exit conditions: None

INSCHR
Inserts a character in a file.

Entry Address (Hex): 6B61
Entry conditions: A = character to insert
HL = address to insert character
Exit conditions: HL = HL + 1
Carry: Set if out of memory

MAKHOL
Inserts a specified number of spaces in a file.

Entry Address (Hex): 6B6D
Entry conditions: BC = number of spaces to insert
HL = address to insert spaces
Exit conditions: HL and BC are preserved
Carry : Set if out of memory

MASDEL
Deletes specified number of characters.

Entry Address (Hex): 6B9F
Entry conditions: BC = number of characters to delete
HL = address of deletion
Exit conditions: HL and BC are preserved

Other Routines

INITIO
Cold start reset.

Entry Address (Hex): 6CD6
Entry conditions: None
Exit conditions: None

IOINIT
Warm start reset.

Entry Address (Hex): 6CE0
Entry conditions: None
Exit conditions: None

MENU
Goes to Main Menu.

Entry Address (Hex): 5797
Entry conditions: None
Exit conditions: None

MUSIC
Makes tone.

Entry Address (Hex): 72C5
Entry conditions: DE = frequency (See Owner's manual)
B = duration (See Owner's manual)
Exit conditions: None

TIME
Read System TIME.

Entry Address (Hex): 190F
Entry conditions: HL = address of 8-byte area for TIME
Exit conditions: HL → TIME (hh:mm:ss)

DATE
Reads system DATE.

Entry Address (Hex): 192F
Entry conditions: HL = address of 8-byte area for DATE
Exit conditions: HL → DATE (mm/dd/yy)

DAY

Reads system DAY of the week.

Entry Address (Hex): 1962
Entry conditions: HL = address of 3 byte area for DAY
Exit conditions: HL → DAY (ddd)

REFERENCE F/ ASCII CODES

Decimal	Hex	Binary	Printed Character	Keyboard Character
0	00	00000000		CTRL @
1	01	00000001		CTRL A
2	02	00000010		CTRL B
3	03	00000011		CTRL C
4	04	00000100		CTRL D
5	05	00000101		CTRL E
6	06	00000110		CTRL F
7	07	00000111		CTRL G
8	08	00001000		CTRL H
9	09	00001001		CTRL I
10	0A	00001010		CTRL J
11	0B	00001011		CTRL K
12	0C	00001100		CTRL L
13	0D	00001101		CTRL M
14	0E	00001110		CTRL N
15	0F	00001111		CTRL O
16	10	00010000		CTRL P
17	11	00010001		CTRL Q
18	12	00010010		CTRL R
19	13	00010011		CTRL S
20	14	00010100		CTRL T
21	15	00010101		CTRL U
22	16	00010110		CTRL V
23	17	00010111		CTRL W
24	18	00011000		CTRL X
25	19	00011001		CTRL Y
26	1A	00011010		CTRL Z
27	1B	00011011		ESC
28	1C	00011100		→
29	1D	00011101		←
30	1E	00011110		↑
31	1F	00011111		↓
32	20	00100000		SPACEBAR
33	21	00100001	!	!

Decimal	Hex	Binary	Printed Character	Keyboard Character
172	AC	10101100	¼	CODE p
173	AD	10101101	¾	CODE ;
174	AE	10101110	½	CODE /
175	AF	10101111	¶	CODE 0
176	B0	10110000	¥	GRPH 7
177	B1	10110001	Ä	CODE A
178	B2	10110010	Ö	CODE O
179	B3	10110011	Ü	CODE U
180	B4	10110100	¢	GRPH 6
181	B5	10110101	˜	CODE [
182	B6	10110110	à	CODE a
183	B7	10110111	ö	CODE o
184	B8	10111000	u	CODE u
185	B9	10111001	ß	CODE S
186	BA	10111010	™	CODE T
187	BB	10111011	é	CODE d
188	BC	10111100	ù	CODE ,
189	BD	10111101	è	CODE v
190	BE	10111110		CODE =
191	BF	10111111	£	CODE F
192	C0	11000000	â	CODE l
193	CI	11000001	ê	CODE 3
194	C2	11000010	î	CODE 8
195	C3	11000011	ô	CODE 9
196	C4	11000100	û	CODE 7
197	C5	11000101	ˆ	CODE –
198	C6	11000110	ë	CODE e
199	C7	11000111	ï	CODE i
200	C8	11001000	á	CODE q
201	C9	11001001	í	CODE k
202	CA	11001010	ó	CODE l
203	CB	11001011	ú	CODE j
204	CC	11001100	ý	CODE y
205	CD	11001101	ñ	CODE n
206	CE	11001110	ã	CODE z

* For lowercase letters a-z, be sure CAPS LOCK is not pressed "down."

Decimal	Hex	Binary	Printed Character	Keyboard Character
207	CF	11001111	ō	CODE .
208	D0	11010000	Â	CODE !
209	D1	11010001	Ê	CODE #
210	D2	11010010	Î	CODE *
211	D3	11010011	Ô	CODE (
212	D4	11010100	Û	CODE &
213	D5	11010101	I	CODE I
214	D6	11010110	E	CODE E
215	D7	11010111	É	CODE D
216	D8	11011000	Á	CODE Q
217	D9	11011001	Í	CODE K
218	DA	11011010	Ó	CODE L
219	DB	11011011	Ú	CODE J
220	DC	11011100	Ý	CODE Y
221	DD	11011101	U	CODE <
222	DE	11011110	E	CODE V
223	DF	11011111	A	CODE X
224	ED	11100000		GRPH Z
225	E1	11100001	▘ (upper left)	GRPH !
226	E2	11100010	▝ (upper right)	GRPH @
227	E3	11100011	▖ (lower left)	GRPH #
228	E4	11100100	▗ (lower right)	GRPH $
229	E5	11100101	▚	GRPH %
230	E6	11100110	▞	GRPH ^
231	E7	11100111	▀ (upper)	GRPH Q
232	E8	11101000	▄ (lower)	GRPH W
233	E9	11101001	▌ (left)	GRPH E
234	EA	11101010	▐ (right)	GRPH R
235	EB	11101011	▛	GRPH A
236	EC	11101100	▜	GRPH S
237	ED	11101101	▙	GRPH D
238	EE	11101110	▟	GRPH F
239	EF	11101111	█	GRPH X
240	F0	11110000	┌	GRPH U
241	F1	11110001	─	GRPH P

Decimal	Hex	Binary	Printed Character	Keyboard Character
242	F2	11110010	┐	GRPH O
243	F3	11110011	┬	GRPH I
244	F4	11110100	├	GRPH J
245	F5	11110101	│	GRPH
246	F6	11110110	└	GRPH M
247	F7	11110111	┘	GRPH >
248	F8	11111000	┴	GRPH <
249	F9	11111001	┤	GRPH L
250	FA	11111010	┼	GRPH K
251	FB	11111011	◤	GRPH H
252	FC	11111100	◢	GRPH T
253	FD	11111101	◥	GRPH G
254	FE	11111110	◣	GRPH Y
255	FF	11111111	▒	GRPH C

REFERENCE G/ 8Ø85 INSTRUCTION SET

What the Instruction Set Is

A computer, no matter how sophisticated, can do only what it is instructed to do. A program is a sequence of instructions, each of which is recognized by the computer and causes it to perform an operation. Once a program is placed in memory space that is accessible to your CPU, you may run that same sequence of instructions as often as you wish to solve the same problem or to do the same function. The set of instructions to which the 8Ø85A CPU will respond is permanently fixed in the design of the chip.

Each computer instruction allows you to initiate the performance of a specific operation. The 8Ø85A implements a group of instructions that move data between registers, between a register and memory, and between a register and an I/O port. It also has arithmetic and logic instructions, conditional and unconditional branch instructions, and machine control instructions. The CPU recognizes these instructions only when they are coded in binary form.

Symbols and Abbreviations:

The following symbols and abbreviations are used in the subsequent description of the 8085A instructions:

SYMBOLS	MEANING
accumulator	Register A
addr	16-bit address quantity
data	8-bit quantity
data 16	16-bit data quantity
byte 2	The second byte of the instruction
byte 3	The third byte of the instruction
port	8-bit address of an I/O device
r,r1,r2	One of the registers A,B,C,D, E,H,L
DDD,SSS	The bit pattern designating one of the registers A,B,C,D,E,H,L (DDD = destination, SSS = source):

DDD or SSS	REGISTER NAME
111	A
000	B
001	C
010	D
011	E
100	H
101	L

rp	One of the register pairs:
	B represents the B,C pair with B as the high-order register and C as the low-order register;
	D represents the D,E pair with D as the high-order register and E as the low-order register;
	H represents the H,L pair with H as the high-order register and L as the low-order register;
	SP represents the 16-bit stack pointer register.

RP	The bit pattern designating one of the register pairs B,D,H,SP:

RP	REGISTER PAIR
00	B-C
01	D-E
10	H-L
11	SP

rh	The first (high-order) register of a designated register pair.
rl	The second (low-order) register of a designated register pair.
PC	16-bit program counter register (PCH and PCL are used to refer to the high-order and low-order 8 bits respectively).
SP	16-bit stack pointer register (SPH and SPL are used to refer to the high-order and low-order 8 bits respectively.)
rm	Bit m of the register r (bits are number 7 through 0 from left to right).
LABEL	16-bit address of subroutine.

	The condition flags:
Z	Zero
S	Size
P	Parity
CY	Carry
AC	Auxiliary Carry
( )	The contents of the memory location or registers enclosed in the parentheses.
	"Is transferred to"
	Logical AND
	Exclusive OR
	Inclusive OR
+	Addition
–	Two's complement subtraction
*	Multiplication
	"Is exchanged with"
—	The one's complement (e.g., $\overline{(A)}$)
n	The restart number 0 through 7
NNN	The binary representation 000 through 111 for restart number 0 through 7 respectively.

The instruction set encyclopedia is a detailed description of the 8085A instruction set. Each instruction is described in the following manner:

1. The MCS-85 macro assembler format, consisting of the instruction mnemonic and operand fields, is printed in **BOLDFACE** on the first line.

2. The name of the instruction is listed to the right of the mnemonic.

3. The next lines contain a symbolic description of what the instruction does.

4. This is followed by a narrative description of the operation of the instruction.

Instruction and Data Formats

Memory used in the MCS-85 system is organized in 8-bit bytes. Each byte has a unique location in physical memory. That location is described by one of a sequence of 16-bit binary addresses. The 8085A can address up to 64K (K = 1024, or 2^{10}; hence, 64K represents the decimal number 65,536) bytes of memory, which may consist of both random-access, read-write memory (RAM) and read-only memory (ROM), which is also random-access.

Data in the 8085A is stored in the form of 8-bit binary integers.

When a register or data word contains a binary number, it is necessary to establish the order in which the bits of the number are written. In the Intel 8085A, BIT 0 is referred to as the **Least Significant Bit (LSB)**, and BIT 7 (of an 8-bit number) is referred to as the **Most Significant Bit (MSB)**.

An 8085A program instruction may be one, two or three bytes in length. Multiple-byte instructions must be stored in successive memory locations; the address of the first byte is always used as the address of the instruction. The exact instruction format will depend on the particular operation to be executed.

Addressing Modes:

Often the data that is to be operated on is stored in memory. When multi-byte numeric data is used, the data, like instructions, is stored in successive memory locations, with the least significant byte first, followed by increasingly significant bytes. The 8085A has four different modes for addressing data storèd in memory or in registers:

- Direct — Bytes 2 and 3 of the instruction contain the exact memory address of the data item (the low-order bits of the address are in byte 2, the high-order bits in byte 3).

- Register — The instruction specifies the register or register pair in which the data is located.

- Register Indirect — The instruction specifies a register pair which contains the memory address where the data is located (the high-order bits of the address are in the first register of the pair the low-order bits in the second).

- Immediate — The instruction contains the data itself. This is either an 8-bit quantity or a 16-bit quantity (least significant byte first, most significant byte second).

Unless directed by an interrupt or branch institution, the execution of instructions proceeds through consecutively increasing memory locations. A branch instruction can specify the address of the next instruction to be executed in one of two ways:

- Direct — The branch instruction contains the address of the next instruction to be executed. (Except for the 'RST' instruction, byte 2 contains the low-order address and byte 3 the high-order address.)

- Register Indirect — The branch instruction indicates a register-pair which contains the address of the next instruction to be executed. (The high-order bits of the address are in the first register of the pair, the low-order bits in the second.)

The RST instruction is a special one-byte call instruction (usually used during interrupt sequences.). RST includes a three-bit field; program control is transferred to the instruction whose address is eight times the contents of this three-bit field.

Condition Flags:

There are five condition flags associated with the execution of instructions on the 8085A. They are Zero, Sign, Parity, Carry, and Auxiliary Carry. Each is represented by a 1-bit register (or flip-flop) in the CPU. A flag is set by forcing the bit to 1: it is reset by forcing the bit to 0.

Unless indicated otherwise, when an instruction affects a flag, it affects it in the following manner:

Zero:	If the result of an instruction has the value 0, this flag is set; otherwise it is reset.
Sign:	If the most significant bit of the result of the operation has the value 1, this flag is set; otherwise it is reset.
Parity:	If the modulo 2 sum of the bits of the result of the operation is 0, (i.e., if the result has even parity), this flag is set; otherwise it is reset (i.e., if the result has odd parity).
Carry:	If the instruction resulted in a carry (from addition) or a borrow (from subtraction or a comparison) out of the high-order bit, this flag is set; otherwise it is reset.

Auxiliary Carry:	If the instruction caused a carry out of bit 3 and into bit 4 of the resulting value, the auxiliary carry is set; otherwise it is reset. This flag is affected by single-precision additions, subtractions, increments, decrements, comparisons, and logical operations, but is principally used with additions and increments preceding a DAA (Decimal Adjust Accumulator) instruction.

Instruction Set Encyclopedia

In the ensuing 22 pages, the complete 8085A instruction set is described, grouped in order under five different functional headings, as follows:

1. **Data Transfer Group** — Moves data between registers or between memory locations and registers. Includes moves, loads, stores, and exchanges. (See below).

2. **Arithmetic Group** — Adds, subtracts, increments, or decrements data in registers or memory. (See pages 5-13.)

3. **Logic Group** — ANDs, ORs, XORs, compares, rotates, or complements data in registers or between memory and a register (See page 5-16.)

4. **Branch Group** — Initiates conditional or unconditional jumps, calls, returns, and restarts. (See page 5-20.)

5. **Stack, I/O, and Machine Control Group** — Includes instructions for maintaining the stack, reading from input ports, writing to output ports, setting and clearing flags. (See page 5-22.)

The formats described in the encyclopedia reflect the assembly language processed by Intel-supplied assembler, used with the Intellec® development systems.

Data Transfer Group

This group of instructions transfers data to and from registers and memory. **Condition flags are not affected by any instruction in this group.**

MOV *r1, r2* Move Register

(r1) ← (r2)

The content of register r2 is moved to register r1.

MOV *r*, M Move from memory

(r) ← ((H) (L))

The content of the memory location, whose address is in registers H and L, is moved to register r.

MOV M, *r* Move to memory

((H)) (L)) ← (r)

The content of register r is moved to the memory location whose address is in registers H and L.

MVI *r, data* Move Immediate

(r) ← (byte 2)

The content of byte 2 of the instruction is moved to register r.

MVI M, ***data*** Move to memory immediate

((H) (L)) ← (byte 2)

The content of byte 2 of the instruction is moved to the memory location whose address is in registers H and L.

LXI ***rp, data 16*** Load register pair immediate

(rh) ← (byte 3),
(rl) ← (byte 2)

Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp. Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp.

LDA ***addr*** Load Accumulator direct

(A) ← ((byte 3)(byte 2))

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register A.

STA ***addr*** Store Accumulator direct

((byte 3)(byte 2)) ← (A)

The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.

LHLD ***addr*** Load H and L direct

(L) ← ((byte 3)(byte 2))
(H) ← ((byte 3)(byte 2) + 1)

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register L. The content of the memory location at the succeeding address is moved to register H.

SHLD *addr* Store H and L direct

((byte 3)(byte 2)) $\leftarrow$ (L)
((byte 3)(byte 2)+1) $\leftarrow$ (H)

The content of register L is moved to the memory location whose address is specified in byte 2 and byte 3. The content of register H is moved to the succeeding memory location.

LDAX *rp* Load accumulator indirect

(A) ((rp))

The content of the memory location, whose address is in the register pair rp, is moved to register A. Note: only register pairs rp=B (registers B and C) or rp=D (registers D and E) may be specified.

STAX *rp* Store accumulator indirect

((rp)) $\leftarrow$ (A)

The content of register A is moved to the memory location whose address is in the register pair rp. Note: only register pairs rp=B (registers B and C) or rp=D (registers D and E) may be specified.

XCHG Exchange H and L with D and E

(H) $\longleftrightarrow$ (D)
(L) $\longleftrightarrow$ (E)

The contents of registers H and L are exchanged with the contents of registers D and E.

Arithmetic Group

This group of instructions performs arithmetic operations on data in registers and memory.

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Carry, and Auxiliary Carry flags according to the standard rules.

All subtraction operations are performed via two's complement arithmetic and set the carry flag to one to indicate a borrow and clear it to indicate no borrow.

ADD *r* Add Register

(A) ← (A) + (r)

The content of register r is added to the content of the accumulator. The result is placed in the accumulator.

ADD M Add memory

(A) ← (A) + ((H) (L))

The content of the memory location whose address is contained in the H and L registers is added to the content of the accumulator. The result is placed in the accumulator.

ADI *data* Add immediate

(A) ← (A) + (byte 2)

The content of the second byte of the instruction is added to the content of the accumulator. The result is placed in the accumulator.

ADC *r* Add Register with carry

(A) ← (A) + (r) + (CY)

The content of register r and the content of the carry bit are added to the content of the accumulator. The result is placed in the accumulator.

ADC M Add memory with carry

(A) ← (A) + ((H) (L)) + (CY)

The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are added to the accumulator. The result is placed in the accumulator.

ACI *data* Add immediate with carry

(A) ← (A) + (byte 2) + (CY)

The content of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

SUB *r* Subtract Register

(A) ← (A) − (r)

The content of register r is subtracted from the content of the accumulator. The result is placed in the accumulator.

SUB M Subtract memory

(A) ← (A) − ((H) (L))

The content of the memory location whose address is contained in the H and L registers is subtracted from the content of the accumulator. The result is placed in the accumulator.

SUI *data* Subtract immediate

(A) ← (A) − (byte 2)

The content of the second byte of the instruction is subtracted from the content of the accumulator. The result is placed in the accumulator.

SBB *r* Subtract Register with borrow

(A) ← (A) − (r) − (CY)

The content of register r and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

SBB M Subtract memory with borrow

(A) ← (A) − ((H) (L)) − (CY)

The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

SBI *data* Subtract immediate with borrow

(A) ← (A) − (byte 2) − (CY)

The contents of the second byte of the instruction and the contents of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

INR *r* Increment Register

(r) ← (r) + 1

The content of register r is incremented by one. Note: All condition flags **except CY** are affected.

INR M Increment memory

((H) (L)) ← ((H) (L)) + 1

The content of the memory location whose address is contained in the H and L registers is incremented by one. Note: All condition flags **except CY** are affected.

DCR *r* Decrement Register

(r) ← (r) − 1

The content of register r is decremented by one. Note: All condition flags **except CY** are affected.

DCR M Decrement memory

((H) (L)) ← ((H) (L)) − 1

The content of the memory location whose address is contained in the H and L registers is decremented by one. Note: All condition flags **except CY** are affected.

INX *rp* Increment register pair

(rh) (rl) ← (rh) (rl) + 1

The content of the register pair rp is incremented by one.

Note: No condition flags are affected.

DCX *rp* Decrement register pair

(rh) (rl) ← (rh) (rl) − 1

The content of the register pair rp is decremented by one.

Note: No condition flags are affected.

DAD *rp* Add register pair to H and L

(H) (L) ← (H) (L) + (rh) (rl)

The content of the register pair rp is added to the content of the register pair H and L. The result is placed in the register pair H and L. The result is placed in the register pair H and L. Note: **Only the CY flag is affected.** It is set if there is a carry out of the double precision add; otherwise it is reset.

DAA Decimal Adjust Accumulator

The eight-bit number in the accumulator is adjusted to form two four-bit Binary-Coded-Decimal digits by .the following process:

1. If the value of the least significant 4 bits of the accumulator is greater than 9 **or** if the CY flag is set, 6 is added to the most significant 4 bits of the accumulator.

Note: All flags are affected.

Logical Group

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

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Auxiliary Carry, and Carry flags according to the standard rules.

ANA *r* AND Register

(A) $\leftarrow$ (A) (r)

The content of register r is logically ANDed with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared and AC is set (8085). The CY flag is cleared and AC is set to the OR'ing of bits 3 of the operands (8080).**

ANA M AND memory

(A) ← (A) ((H) (L))

The contents of the memory location whose address is contained in the H and L registers is logically ANDed with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared and AC is set (8085). The CY flag is cleared and AC is set to the OR'ing of bits 3 of the operands (8080).**

ANI ***data*** AND immediate

(A) ← (A) (byte 2)

The content of the second byte of the instruction is logically ANDed with the contents of the accumulator. The result is placed in the accumulator. **The CY flag is cleared and AC is set (8085). The CY flag is cleared and AC is set to the OR'ing of bits 3 of the operands (8080).**

XRA ***r*** Exclusive OR Register

(A) ← (A) (r)

The content of register r is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

XRA M Exclusive OR Memory

(A) ← (A) ((H) (L))

The content of the memory location whose address is contained in the H and L registers is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

XRI ***data*** Exclusive OR immediate

(A) ← (A) (byte 2)

The content of the second byte of the instruction is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

ORA ***r*** OR Register

(A) ← (A) V (r)

The content of register r is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

ORA M OR memory

(A) ← (A) V ((H) (L))

The content of the memory location whose address is contained in the H and L registers is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

ORI ***data*** OR Immediate

(A) ← (A) V (byte 2)

The content of the second byte of the instruction is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

CMP *r* Compare Register

(A) – (r)

The content of register r is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 of (A) = (r). The CY flag is set to 1 if (A) < (r).**

CMP M Compare memory

(A) – ((H) (L))

The content of the memory location whose address is contained in the H and L registers is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A) = ((H) (L)). The CY flag is set to 1 if (A)<((H) (L)).**

CPI *data* Compare immediate
(A) – (byte 2)

The content of the second byte of the instruction is subtracted from the accumulator. The condition flags are set by the result of the subtraction. **The Z flag is set to 1 if (A) = (byte 2). The CY flag is set to 1 if (A)<(byte 2).**

RLC Rotate left

$(A_{n+1}) \leftarrow (A_n)$;$(A_0) \leftarrow (A_7)$
$(CY) \leftarrow (A_7)$

The content of the accumulator is rotated left one position. The low order bit and the CY flag are both set to the value shifted out of the high order bit position. **Only the CY flag is affected.**

RRC Rotate right

$(A_n) \leftarrow (A_{n+1})$; $(A_7) \leftarrow (A_0)$
$(CY) \leftarrow (A_0)$

The content of the accumulator is rotated right one position. The high order bit and the CY flag are both set to the value shifted out of the low order bit position. **Only the CY flag is affected.**

RAL Rotate left through carry

$(A_{n+1}) \leftarrow (A_n)$; $(CY) \leftarrow (A_7)$
$(A_0) \leftarrow (CY)$

The content of the accumulator is rotated left one position through the CY flag. The low order bit is set equal to the CY flag and the CY flag is set to the value shifted out of the high order bit. **Only the CY flag is affected.**

RAR Rotate right through carry

$(A_n) \leftarrow (A_{n+1})$;$(CY) \leftarrow (A_0)$
$(A_7) \leftarrow (CY)$

The content of the accumulator is rotated right one position through the CY flag. The high order bit is set to the CY flag and the CY flag is set to the value shifted out of the low order bit. **Only the CY flag is affected.**

CMA Complement accumulator

$(A) \leftarrow (\overline{A})$

The contents of the accumulator are complemented (zero bits become 1, one bits become 0). **No flags are affected.**

CMC Complement carry

(CY) ← ($\overline{CY}$)

The CY flag is complemented. **No other flags are affected.**

STC Set carry

(CY) ← 1

The CY flag is set to 1. **No other flags are affected.**

Branch Group

This group of instructions alter normal sequential program flow.

Conditional flags are not affected by any instruction in this group.

The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register PC (the program counter). Conditional transfers examine the status of one of the four processor flags to determine if the specified branch is to be executed. The conditions that may be specified are as follows:

CONDITION	**CCC**
NZ — not zero (Z = 0)	000
Z — zero (Z = 1)	001
NC — no carry (CY = 0)	010
C — carry (CY = 1)	011
PO — parity odd (P = 0)	00
OE — parity even (P = 1)	101
P — plus (S = 0)	110
M — minus (S = 1)	111

JMP *addr* Jump

(PC) ← (byte 3) (byte 2)

Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

J*condition addr* Conditional jump

IF (CCC),
(PC) ← (byte 3) (byte 2)

If the specified condition is true, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction; otherwise, control continues sequentially.

CALL *addr* Call

((SP) – 1) ← (PCH)
((SP) – 2) ← (PCL)
(SP) ← (SP) – 2
(PC) ← (byte 3) (byte 2)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

***C**condition addr* Condition call

IF (CCC),
((SP) – 1) ← (PCH)
((SP) – 2) ← (PCL)
(SP) ← (SP) – 2
(PC) ← (byte 3) (byte 2)

If the specified condition is true, the actions specified in the CALL instruction (see above) are performed; otherwise, control continues sequentially.

RET Return

(PCL) ← ((SP));
(PCH) ← ((SP) + 1);
(SP) ← (SP) + 2 ;

The content of the memory location whose address is specified in register SP is moved to the low-order eight bits of register PC. The content of the memory location whose address is one more than the content of register SP is moved to the high-order eight bits of register PC. The content of register SP is incremented by 2.

***R**condition* Conditional return

If (CCC),
(PCL) ← ((SP))
(PCH) ← ((SP) + 1)
(SP) ← (SP) + 2

If the specified condition is true, the actions specified in the RET instruction (see above) are performed; otherwise, control continues sequentially.

RST *n* Restart

((SP) – 1) ← (PCH)
((SP) – 2) ← (PCL)
(SP) ← (SP) – 2
(PC) ← 8 * (NNN)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two. Control is transferred to the instruction whose address is eight times the content of NNN.

PCHL Jump H and L indirect — move H and L to PC

(PCH) ← (H)
(PCL) ← (L)

The content of register H is moved to the high-order eight bits of register PC. The content of register L is moved to the low-order eight bits of register PC.

Stack, I/O, and Machine Control Group

This group of instructions performs I/O, manipulates the Stack, and alters internal control flags.

Unless otherwise specified, **condition flags are not affected by any instructions in this group.**

PUSH ***rp*** Push

$((SP) - 1) \leftarrow (rh)$
$((SP) - 2) \leftarrow (rl)$
$((SP) \leftarrow (SP) - 2$

The content of the high-order register of register pair rp is moved to the memory location whose address is one less than the content of register SP. The content of the low-order register of register pair rp is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2.

Note: Register pair rp = SP may not be specified.

PUSH PSW Push processor status word

$((SP) - 1) \leftarrow (A)$
$((SP) - 2)_0 \leftarrow (CY), ((SP) - 2)_1$ X
$((SP) - 2)_2 \leftarrow (P), ((SP) - 2)_3$ X
$((SP) - 2)_4 \leftarrow (AC), ((SP) - 2)_5$ X
$((SP) - 2)_6 \leftarrow (Z), ((SP) - 2)_7$ (S)
$(SP) \quad (SP) - 2$ X: Undefined.

The content of register A is moved to the memory location whose address is one less than register SP. The contents of the condition flags are assembled into a processor status word and the word is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two.

POP *rp* **Pop**

$(rl) \leftarrow ((SP))$
$(rh) \leftarrow ((SP) + 1)$
$(SP) \leftarrow (SP) + 2$

The content of the memory location, whose address is specified by the content of register SP, is moved to the low-order register of register pair rp. The content of the memory location, whose address is one more than the content of register SP, is moved to the high-order register of register rp. The content of register SP is incremented by 2.

Note: Register pair rp = SP may not be specified.

POP PSW Pop processor status word

$(CY) \leftarrow ((SP))_0$
$(P) \leftarrow ((SP))_2$
$(AC) \leftarrow ((SP))_4$
$(Z) \leftarrow ((SP))_6$
$(S) \leftarrow ((SP))_7$
$(A) \leftarrow ((SP) + 1)$
$(SP) \leftarrow (SP) + 2$

The content of the memory location whose address is specified by the content of register SP is used to restore the condition flags. The content of the memory location whose address is one more than the content of register SP is moved to register A. The content of register SP is incremented by 2.

XTHL Exchange stack top with H and L

$(L) \leftrightarrow ((SP))$
$(H) \leftrightarrow ((SP) + 1)$

The content of the L register is exchanged with the content of the memory location whose address is specified by the content of register SP. The content of the H register is exchanged with the content of the memory location whose address is one more than the content of register SP.

SPHL Move HL to SP

$(SP) \leftarrow (H)\ (L)$

The contents of registers H and L (16 bits) are moved to register SP.

IN ***port*** Input

$(A) \leftarrow (data)$

The data placed on the eight bit bi-directional data bus by the specified port is moved to register A.

OUT ***port*** Output

$(data) \leftarrow (A)$

The content of register A is placed on the eight bit bi-directional data bus for transmission to the specified port.

EI Enable interrupts

The interrupt system is enabled **following the execution of the next instruction. Interrupts are not recognized during the EI instruction.**

> **Note:** Placing an EI instruction on the bus in response to $\overline{\text{INTA}}$ during an INA cycle is prohibited. (8085)

DI Disable interrupts

The interrupt system is disabled **immediately following the execution of the DI instruction. Interrupts are not recognized during the DI instruction.**

> **Note:** Placing a DI instruction on the bus in response to $\overline{\text{INTA}}$ during an INA cycle is prohibited. (8085)

HLT Halt

The processor is stopped. The registers and flags are unaffected. (8080) A second ALE is generated during the execution of HLT to strobe out the Halt cycle status information. (8085)

NOP No op

No operation is performed. The registers and flags are unaffected.

RIM Read Interrupt Masks (8085 only)

The RIM instruction loads data into the accumulator relating to interrupts and the serial input. This data contains the following information:

- Current interrupt mask status for the RST 5.5, 6.5, and 7.5 hardware interrupts (1 = mask disabled)
- Current interrupt enable flag status (1 = interrupts enabled) except immediately following a TRAP interrupt. (See below.)
- Hardware interrupts pending (i.e., signal received but not yet serviced), on the RST 5.5, 6.5, and 7.5 lines.
- Serial input data.

Immediately following a TRAP interrupt, the RIM instruction must be executed as a part of the service routine if you need to retrieve current interrupt status later. Bit 3 of the accumulator is (in this special case only) loaded with the interrupt enable (IE) flag status that existed prior to the TRAP interrupt. Following an RST 5.5, 6.5, 7.5, or INTR interrupt, the interrupt flag flip-flop reflects the current interrupt enable status. Bit 6 of the accumulator (I7.5) is loaded with the status of the RST 7.5 flip-flop, which is always set (edge-triggered) by an input on the RST 7.5 input line, even when that interrupt has been previously masked. (See SIM Instruction.)

SIM Set Interrupt Masks (8085 only)

The execution of the SIM instruction uses the contents of the accumulator (which must be previously loaded) to perform the following functions:

- Program the interrupt mask for the RST 5.5, 6.5, and 7.5 hardware interrupt.
- Reset the edge-triggered RST 7.5 input latch.
- Load the SOD output latch.

To program the interrupt masks, first set accumulator bit 3 to 1 and set to 1 any bits 0, 1, and 2, which disable interrupts RST 5.5, 6.5, and 7.5, respectively. Then do a SIM instruction. If accumulator bit 3 is 0 when the SIM instruction is executed, the interrupt mask register will not change. If accumulator bit 4 is 1 when the SIM instruction is executed, the RST 7.5 latch is then reset. RST 7.5 is distinguished by the fact that its latch is always set by a rising edge on the RST 7.5 input pin, even if the jump to service routine is inhibited by masking. This latch remains high until cleared by a $\overline{\text{RESET IN}}$, by a SIM instruction with accumulator bit 4 high, or by an internal processor acknowledge to an RST 7.5 interrupt subsequent to the removal of the mask (by a SIM instruction). The $\overline{\text{RESET IN}}$ signal always sets all three RST mask bits.

If accumulator bit 6 is at the 1 level when the SIM instruction is executed, the state of accumulator bit 7 is loaded into the SOD latch and thus becomes available for interface to an external device. The SOD latch is unaffected by the SIM instruction if bit 6 is 0. SOD is always reset by the $\overline{\text{RESET IN}}$ signal.

8080A/8085A INSTRUCTION SET INDEX

Instruction		Code	Bytes	T States		Machine Cycles
				8085A	8080A	
ACI	DATA	CE data	2	7	7	F R
ADC	REG	1000 1SSS	1	4	4	F
ADC	M	8E	1	7	7	F R
ADD	REG	1000 0SSS	1	4	4	F
ADD	M	86	1	7	7	F R
ADI	DATA	C6 data	2	7	7	F R
ANA	REG	1010 0SSS	1	4	4	F
ANA	M	A6	1	7	7	F R
ANI	DATA	E6 data	2	7	7	F R
CALL	LABEL	CD addr	3	18	17	S R R W W*
CC	LABEL	DC addr	3	9 18	11/17	S R•/S R R W W*
CM	LABEL	FC addr	3	9/18	11/17	S R•/S R R W W*
CMA		2F	1	4	4	F
CMC		3F	1	4	4	F
CMP	REG	1011 1SSS	1	4	4	F
CMP	M	BE	1	7	7	F R
CNC	LABEL	D4 addr	3	9/18	11/17	S R•/S R R W W*
CNZ	LABEL	C4 addr	3	9/18	11/17	S R•/S R R W W*
CP	LABEL	F4 addr	3	9/18	11/17	S R•/S R R W W*
CPE	LABEL	EC addr	3	9/18	11/17	S R•/S R R W W*
CPI	DATA	FE data	2	7	7	F R
CPO	LABEL	E4 addr	3	9 18	11 17	S R•/S R R W W*
CZ	LABEL	CC addr	3	9 18	11 17	S R•/S R R W W*
DAA		27	1	4	4	F
DAD	RP	00RP 1001	1	10	10	F B B
DCR	REG	00SS S101	1	4	5	F
DCR	M	35	1	10	10	F R W
DCX	RP	00RP 1011	1	6	5	S*
DI		F3	1	4	4	F
EI		FB	1	4	4	F
HLT		76	1	5	7	F B
IN	PORT	DB data	2	10	10	F R
INR	REG	00SS S100	1	4	5	F*
INR	M	34	1	10	10	F R W
INX	RP	00RP 0011	1	6	5	S
JC	LABEL	DA addr	3	7/10	10	F R F R R†
JM	LABEL	FA addr	3	7/10	10	F R/F R R†
JMP	LABEL	C3 addr	3	10	10	F R R
JNC	LABEL	D2 addr	3	7/10	10	F R F R R†
JNZ	LABEL	C2 addr	3	7 10	10	F R F R R†
JP	LABEL	F2 addr	3	7 10	10	F R F R R†
JPE	LABEL	EA addr	3	7/10	10	F R F R R†
JPO	LABEL	E2 addr	3	7/10	10	F R F R R†
JZ	LABEL	CA addr	3	7/10	10	F R/F R R†
LDA	ADDR	3A addr	3	13	13	F R R R
LDAX	RP	000X 1010	1	7	7	F R
LHLD	ADDR	2A addr	3	16	16	F R R R R
LXI	RP DATA16	00RP 0001 data16	3	10	10	F R R
MOV	REG REG	01DD DSSS	1	4	5	F*
MOV	M REG	0111 0SSS	1	7	7	F W
MOV	REG M	01DD D110	1	7	7	F R
MVI	REG DATA	00DD D110 data	2	7	7	F R
MVI	M DATA	36 data	2	10	10	F R W
NOP		00	1	4	4	F
ORA	REG	1011 0SSS	1	4	4	F
ORA	M	B6	1	7	7	F R
ORI	DATA	F6 data	2	7	7	F R

Instruction	Code	Bytes	T States		Machine Cycles
			8085A	8080A	
OUT PORT	D3 data	2	10	10	F R O
PCHL	E9	1	6	5	S*
POP RP	11RP 0001	1	10	10	F R R
PUSH RP	11RP 0101	1	12	11	S W W*
RAL	17	1	4	4	F
RAR	1F	1	4	4	F
RC	D8	1	6/12	5/11	S/S R R*
RET	C9	1	10	10	F R R
RIM (8085A only)	20	1	4	–	F
RLC	07	1	4	4	F
RM	F8	1	6/12	5/11	S/S R R*
RNC	D0	1	6/12	5/11	S/S R R*
RNZ	C0	1	6/12	5/11	S/S R R*
RP	F0	1	6/12	5/11	S/S R R*
RPE	E8	1	6/12	5/11	S/S R R*
RPO	E0	1	6/12	5/11	S/S R R*
RRC	0F	1	4	4	F
RST N	11XX X111	1	12	11	S W W*
RZ	C8	1	6/12	5/11	S/S R R*
SBB REG	1001 1SSS	1	4	4	F
SBB M	9E	1	7	7	F R
SBI DATA	DE data	2	7	7	F R
SHLD ADDR	22 addr	3	16	16	F R R W W
SIM (8085A only)	30	1	4	–	F
SPHL	F9	1	6	5	S*
STA ADDR	32 addr	3	13	13	F R R W
STAX RP	000X 0010	1	7	7	F W
STC	37	1	4	4	F
SUB REG	1001 0SSS	1	4	4	F
SUB M	96	1	7	7	F R
SUI DATA	D6 data	2	7	7	F R
XCHG	EB	1	4	4	F
XRA REG	1010 1SSS	1	4	4	F
XRA M	AE	1	7	7	F R
XRI DATA	EE data	2	7	7	F R
XTHL	E3	1	16	18	F R R W W

Machine cycle types

F	Four clock period instr fetch	
S	Six clock period instr fetch	
R	Memory read	
I	I/O read	
W	Memory write	
O	I/O write	
B	Bus idle	
X	Variable or optional binary digit	
DDD	Binary digits identifying a destination register	B = 000, C = 001, D - 010 Memory = 110
SSS	Binary digits identifying a source register	E = 011, H = 100, L - 101 A = 111

RP	Register Pair	BC = 00, HL = 10
		DE = 01, SP = 11

*Five clock period instruction fetch with 8080A

†The longer machine cycle sequence applies regardless of condition evaluation with 8080A

•An extra READ cycle (R) will occur for this condition with 8080A

8085A CPU INSTRUCTIONS IN OPERATION CODE SEQUENCE

OP CODE	MNEMONIC		OP CODE	MNEMONIC		OP CODE	MNEMONIC	
00	NOP		2B	DCX	H	56	MOV	D,M
01	LXI	B,D16	2C	INR	L	57	MOV	D,A
02	STAX	B	2D	DCR	L	58	MOV	E,B
03	INX	B	2E	MVI	L,D8	59	MOV	E,C
04	INR	B	2F	CMA		5A	MOV	E,D
05	DCR	B	30	SIM		5B	MOV	E,E
06	MVI	B,D8	31	LXI	SP,D16	5C	MOV	E,H
07	RLC		32	STA	Adr	5D	MOV	E,L
08	–		33	INX	SP	5E	MOV	E,M
09	DAD	B	34	INR	M	5F	MOV	E,A
0A	LDAX	B	35	DCR	M	60	MOV	H,B
0B	DCX	B	36	MVI	M,D8	61	MOV	H,C
0C	INR	C	37	STC		62	MOV	H,D
0D	DCR	C	38	–		63	MOV	H,E
0E	MVI	C,D8	39	DAD	SP	64	MOV	H,H
0F	RRC		3A	LDA	Adr	65	MOV	H,L
10	–		3B	DCX	SP	66	MOV	H,M
11	LXI	D,D16	3C	INR	A	67	MOV	H,A
12	STAX	D	3D	DCR	A	68	MOV	L,B
13	INX	D	3E	MVI	A,D8	69	MOV	L,C
14	INR	D	3F	CMC		6A	MOV	L,D
15	DCR	D	40	MOV	B,B	6B	MOV	L,E
16	MVI	D,D8	41	MOV	B,C	6C	MOV	L,H
17	RAL		42	MOV	B,D	6D	MOV	L,L
18	–		43	MOV	B,E	6E	MOV	L,M
19	DAD	D	44	MOV	B,H	6F	MOV	L,A
1A	LDAX	D	45	MOV	B L	70	MOV	M,B
1B	DCX	D	46	MOV	B,M	71	MOV	M,C
1C	INR	E	47	MOV	B,A	72	MOV	M,D
1D	DCR	E	48	MOV	C,B	73	MOV	M,E
1E	MVI	E,D8	49	MOV	C,C	74	MOV	M,H
1F	RAR		4A	MOV	C,D	75	MOV	M,L
20	RIM		4B	MOV	C,E	76	HLT	
21	LXI	H,D16	4C	MOV	C,H	77	MOV	M,A
22	SHLD	Adr	4D	MOV	C,L	78	MOV	A,B
23	INX	H	4E	MOV	C,M	79	MOV	A,C
24	INR	H	4F	MOV	C,A	7A	MOV	A,D
25	DCR	H	50	MOV	D,B	7B	MOV	A,E
26	MVI	H,D8	51	MOV	D,C	7C	MOV	A,H
27	DAA		52	MOV	D,D	7D	MOV	A,L
28	–		53	MOV	D,E	7E	MOV	A,M
29	DAD	H	54	MOV	D,H	7F	MOV	A,A
2A	LHLD	Adr	55	MOV	D,L	80	ADD	B

OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC
81	ADD C	AC	XRA H	D7	RST 2
82	ADD D	AD	XRA L	D8	RC
83	ADD E	AE	XRA M	D9	–
84	ADD H	AF	XRA A	DA	JC Adr
85	ADD L	B0	ORA B	DB	IN D8
86	ADD M	B1	ORA C	DC	CC Adr
87	ADD A	B2	ORA D	DD	–
88	ADC B	B3	ORA E	DE	SBI D8
89	ADC C	B4	ORA H	DF	RST 3
8A	ADC D	B5	ORA L	E0	RPO
8B	ADC E	B6	ORA M	E1	POP H
8C	ADC H	B7	ORA A	E2	JPO Adr
8D	ADC L	B8	CMP B	E3	XTHL
8E	ADC M	B9	CMP C	E4	CPO Adr
8F	ADC A	BA	CMP D	E5	PUSH H
90	SUB B	BB	CMP E	E6	ANI D8
91	SUB C	BC	CMP H	E7	RST 4
92	SUB D	BD	CMP L	E8	RPE
93	SUB E	BE	CMP M	E9	PCHL
94	SUB H	BF	CMP A	EA	JPE Adr
95	SUB L	C0	RNZ	EB	XCHG
96	SUB M	C1	POP B	EC	CPE Adr
97	SUB A	C2	JNZ Adr	ED	–
98	SBB B	C3	JMP Adr	EE	XRI D8
99	SBB C	C4	CNZ Adr	EF	RST 5
9A	SBB D	C5	PUSH B	F0	RP
9B	SBB E	C6	ADI D8	F1	POP PSW
9C	SBB H	C7	RST 0	F2	JP Adr
9D	SBB L	C8	RZ	F3	DI
9E	SBB M	C9	RET Adr	F4	CP Adr
9F	SBB A	CA	JZ	F5	PUSH PSW
A0	ANA B	CB	–	F6	ORI D8
A1	ANA C	CC	CZ Adr	F7	RST 6
A2	ANA D	CD	CALL Adr	F8	RM
A3	ANA E	CE	ACI D8	F9	SPHL
A4	ANA H	CF	RST 1	FA	JM Adr
A5	ANA L	D0	RNC	FB	EI
A6	ANA M	D1	POP D	FC	CM Adr
A7	ANA A	D2	JNC Adr	FD	–
A8	XRA B	D3	OUT D8	FE	CPI D8
A9	XRA C	D4	CNC Adr	FF	RST 7
AA	XRA D	D5	PUSH D		
AB	XRA E	D6	SUI D8		

D8 = constant, or logical/arithmetic expression that evaluates
to an 8-bit data quantity

Adr = 16-bit address

D16 = *constant, or logical/arithmetic expression that evaluates*
to a 16-bit data quantity

8085A INSTRUCTION SET SUMMARY BY FUNCTIONAL GROUPING

		Instruction Code (1)								
Mnemonic	Description	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Page
MOVE, LOAD, AND STORE										
MOVr1 r2	Move register to register	0	1	D	D	D	S	S	S	5 4
MOV M r	Move register to memory	0	1	1	1	0	S	S	S	5 4
MOV r M	Move memory to register	0	1	D	D	D	1	1	0	5 4
MVI r	Move immediate register	0	0	D	D	D	1	1	0	5 4
MVI M	Move immediate memory	0	0	1	1	0	1	1	0	5 4
LXI B	Load immediate register Pair B & C	0	0	0	0	0	0	0	1	5 5
LXI D	Load immediate register Pair D & E	0	0	0	1	0	0	0	1	5 5
LXI H	Load immediate register Pair H & L	0	0	1	0	0	0	0	1	5 5
STAX B	Store A indirect	0	0	0	0	0	0	1	0	5 6
STAX D	Store A indirect	0	0	0	1	0	0	1	0	5 6
LDAX B	Load A indirect	0	0	0	0	1	0	1	0	5 5
LDAX D	Load A indirect	0	0	0	1	1	0	1	0	5 5
STA	Store A direct	0	0	1	1	0	0	1	0	5 5
LDA	Load A direct	0	0	1	1	1	0	1	0	5 5
SHLD	Store H & L direct	0	0	1	0	0	0	1	0	5 5
LHLD	Load H & L direct	0	0	1	0	1	0	1	0	5 5
XCHG	Exchange D & E H & L Registers	1	1	1	0	1	0	1	1	5 6
STACK OPS										
PUSH B	Push register Pair B & C on stack	1	1	0	0	0	1	0	1	5 15
PUSH D	Push register Pair D & E on stack	1	1	0	1	0	1	0	1	5 15
PUSH H	Push register Pair H & L on stack	1	1	1	0	0	1	0	1	5 15
PUSH PSW	Push A and Flags on stack	1	1	1	1	0	1	0	1	5 15
POP B	Pop register Pair B & C off stack	1	1	0	0	0	0	0	1	5 15
POP D	Pop register Pair D & E off stack	1	1	0	1	0	0	0	1	5 15
POP H	Pop register Pair H & L off stack	1	1	1	0	0	0	0	1	5 15
POP PSW	Pop A and Flags off stack	1	1	1	1	0	0	0	1	5 15
XTHL	Exchange top of stack H & L	1	1	1	0	0	0	1	1	5 16
SPHL	H & L to stack pointer	1	1	1	1	1	0	0	1	5 16
LXI SP	Load immediate stack pointer	0	0	1	1	0	0	0	1	5 5
INX SP	Increment stack pointer	0	0	1	1	0	0	1	1	5 9
DCX SP	Decrement stack pointer	0	0	1	1	1	0	1	1	5 9

		Instruction Code (1)								
Mnemonic	Description	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Page
JUMP										
JMP	Jump unconditional	1	1	0	0	0	0	1	1	5 13
JC	Jump on carry	1	1	0	1	1	0	1	0	5-13
JNC	Jump on no carry	1	1	0	1	0	0	1	0	5 13
JZ	Jump on zero	1	1	0	0	1	0	1	0	5-13
JNZ	Jump on no zero	1	1	0	0	0	0	1	0	5-13
JP	Jump on positive	1	1	1	1	0	0	1	0	5-13
JM	Jump on minus	1	1	1	1	1	0	1	0	5-13
JPE	Jump on parity even	1	1	1	0	1	0	1	0	5-13
JPO	Jump on parity odd	1	1	1	0	0	0	1	0	5-13
PCHL	H & L to program counter	1	1	1	0	1	0	0	1	5-15
CALL										
CALL	Call unconditional	1	1	0	0	1	1	0	1	5-13
CC	Call on carry	1	1	0	1	1	1	0	0	5-14
CNC	Call on no carry	1	1	0	1	0	1	0	0	5-14
CZ	Call on zero	1	1	0	0	1	1	0	0	5-14
CNZ	Call on no zero	1	1	0	0	0	1	0	0	5-14
CP	Call on positive	1	1	1	1	0	1	0	0	5-14
CM	Call on minus	1	1	1	1	1	1	0	0	5 14
CPE	Call on parity even	1	1	1	0	1	1	0	0	5-14
CPO	Call on parity odd	1	1	1	0	0	1	0	0	5-14
RETURN										
RET	Return	1	1	0	0	1	0	0	1	5-14
RC	Return on carry	1	1	0	1	1	0	0	0	5-14
RNC	Return on no carry	1	1	0	1	0	0	0	0	5-14
RZ	Return on zero	1	1	0	0	1	0	0	0	5-14
RNZ	Return on no zero	1	1	0	0	0	0	0	0	5-14
RP	Return on positive	1	1	1	1	0	0	0	0	5-14
RM	Return on minus	1	1	1	1	1	0	0	0	5 14
RPE	Return on parity even	1	1	1	0	1	0	0	0	5-14
RPO	Return on parity odd	1	1	1	0	0	0	0	0	5-14
RESTART										
RST	Restart	1	1	A	A	A	1	1	1	5-14
INPUT/OUTPUT										
IN	Input	1	1	0	1	1	0	1	1	5-16
OUT	Output	1	1	0	1	0	0	1	1	5-16
INCREMENT AND DECREMENT										
INR r	Increment register	0	0	D	D	D	1	0	0	5-8
DCR r	Decrement register	0	0	D	D	D	1	0	1	5-8
INR M	Increment memory	0	0	1	1	0	1	0	0	5-8
DCR M	Decrement memory	0	0	1	1	0	1	0	1	5-8
INX B	Increment B & C registers	0	0	0	0	0	0	1	1	5-9
INX D	Increment D & E registers	0	0	0	1	0	0	1	1	5-9
INX H	Increment H & L registers	0	0	1	0	0	0	1	1	5-9
DCX B	Decrement B & C	0	0	0	0	1	0	1	1	5-9
DCX D	Decrement D & E	0	0	0	1	1	0	1	1	5-9
DCX H	Decrement H & L	0	0	1	0	1	0	1	1	5-9

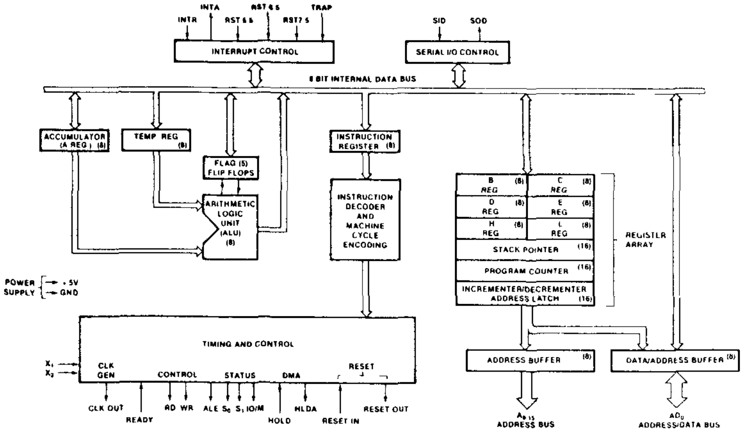
Mnemonic	Description	Instruction Code (1) D7	D6	D5	D4	D3	D2	D1	D0	Page
ADD										
ADD r	Add register to A	1	0	0	0	0	S	S	S	5 6
ADC r	Add register to A with carry	1	0	0	0	1	S	S	S	5 6
ADD M	Add memory to A	1	0	C	0	0	1	1	0	5 6
ADC M	Add memory to A with carry	1	0	0	0	1	1	1	0	5 7
ADI	Add immediate to A	1	1	0	0	0	1	1	0	5 6
ACI	Add immediate to A with carry	1	1	0	0	1	1	1	0	5 7
DAD B	Add B & C to H & L	0	0	0	0	1	0	0	1	5 9
DAD D	Add D & E to H & L	0	0	0	1	1	0	0	1	5 9
DAD H	Add H & L to H & L	0	0	1	0	1	0	0	1	5 9
DAD SP	Add stack pointer to H & L	0	0	1	1	1	0	0	1	5 9
SUBTRACT										
SUB r	Subtract register from A	1	0	0	1	0	S	S	S	5 7
SBB r	Subtract register from A with borrow	1	0	0	1	1	S	S	S	5 7
SUB M	Subtract memory from A	1	0	0	1	0	1	1	0	5 7
SBB M	Subtract memory from A with borrow	1	0	0	1	1	1	1	0	5 8
SUI	Subtract immediate from A	1	1	0	1	0	1	1	0	5 7
SBI	Subtract immediate from A with borrow	1	1	0	1	1	1	1	0	5 8
LOGICAL										
ANA r	And register with A	1	0	1	0	0	S	S	S	5 9
XRA r	Exclusive OR register with A	1	0	1	0	1	S	S	S	5 10
ORA r	OR register with A	1	0	1	1	0	S	S	S	5 10
CMP r	Compare register with A	1	0	1	1	1	S	S	S	5 11
ANA M	And memory with A	1	0	1	0	0	1	1	0	5 10
XRA M	Exclusive OR memory with A	1	0	1	0	1	1	1	0	5 10
ORA M	OR memory with A	1	0	1	1	0	1	1	0	5 11
CMP M	Compare memory with A	1	0	1	1	1	1	1	0	5 11
ANI	And immediate with A	1	1	1	0	0	1	1	0	5 10
XRI	Exclusive OR mmediate with A	1	1	1	0	1	1	1	0	5 10
ORI	OR immediate with A	1	1	1	1	0	1	1	0	5 11
CPI	Compare immediate w th A	1	1	1	1	1	1	1	0	5 11
ROTATE										
RLC	Rotate A left	0	0	0	0	0	1	1	1	5 11
RRC	Rotate A right	0	0	0	0	1	1	1	1	5 12
RAL	Rotate A left through carry	0	0	0	1	0	1	1	1	5 12
RAR	Rotate A right through carry	0	0	0	1	1	1	1	1	5 12

Mnemonic	Description	Instruction Code (1) D7	D6	D5	D4	D3	D2	D1	D0	Page
SPECIALS										
CMA	Complement A	0	0	1	0	1	1	1	1	5 12
STC	Set carry	0	0	1	1	0	1	1	1	5 12
CMC	Complement carry	0	0	1	1	1	1	1	1	5 12
DAA	Decimal adjust A	0	0	1	0	0	1	1	1	5 9
CONTROL										
EI	Enable Interrupts	1	1	1	1	1	0	1	1	5 17
DI	Disable Interrupt	1	1	1	1	0	0	1	1	5 17
NOP	No operation	0	0	0	0	0	0	0	0	5 17
HLT	Halt	0	1	1	1	0	1	1	0	5 17
NEW 8085A INSTRUCTIONS										
RIM	Read Interrupt Mask	0	0	1	0	0	0	0	0	5 17
SIM	Set Interrupt Mask	0	0	1	1	0	0	0	0	5 18

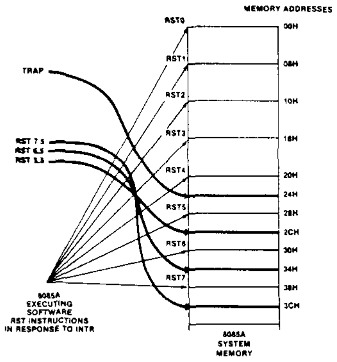
NOTES 1 DDS or SSS B 000 C 001 D 010, E011, H 100, L 101, Memory 110 A 111
2 Two possible cycle times (6/12) indicate instruction cycles dependent on condition flags

When a register or data word contains a binary number, it is necessary to establish the order in which the bits of the number are written. In the Intel 8Ø85A, BIT Ø is referred to as the **Least Significant Bit (LSB)**, and BIT 7 (of an 8-bit number) is referred to as the **Most Significant Bit (MSB)**.

An 8Ø85A program instruction may be one, two or three bytes in length. Multiple-byte instructions must be stored in successive memory locations; the address of the first byte is always used as the address of the instruction. The exact instruction format will depend on the particular operation to be executed.



8085A CPU FUNCTIONAL BLOCK DIAGRAM



8085A HARDWARE AND SOFT-
WARE RST BRANCH LOCATIONS

INDEX

Cat. No. 26-3823

[illegible]

11/86-TM 874-9893 Printed in U S A

TRS-80®

Model 100 Portable Computer

Debug/Assembler