UNIVERSAL TRAINER

SELF INSTRUCTION MANUAL

Manual Revision 1.1

EMAC, inc.

EQUIPMENT MONITOR AND CONTROL

CARBONDALE, IL 62901
618-529-4525

WARRANTY, RETURN POLICY, AND LIABILITY DISCLAIMER

I. WARRANTY

This limited warranty is given to you by EMAC Inc.

This warranty extends only to the original customer purchase of the product.

What the warranty covers and how long:
If this product was purchased assembled and is defective in material or workmanship, return the product within one (1) year of the original date of purchase, and we will repair or replace it (with the same or an equivalent model), at our option, with no charge to you. If this product was purchased unassembed and contained defective parts, we will replace the defective part(s) for a period of 30 days from the original date of purchase. Return the product or defective parts to EMAC for replacement.

How to exercise your warranty or obtain service:
You may arrange for service or for warranty repair by obtaining a Return Authorization Number and then shipping your Product to EMAC Inc. There will be no charge for warranty service except for your PREPAID shipping cost to our site. We suggest that you retain the original packing material in case you need to ship your product. When returning your Product to our site, please be sure to include:

1. Name
2. Address
3. Phone Number
4. Dated Proof of Purchase (required)
5. A description of the operating problem
6. Serial Number (if available)

We cannot assume responsibility for loss or damage during shipping.

After we repair or replace (at our option) your Product under warranty, you will be shipped the Product at no cost to you.

What this Warranty does not cover:
This warranty does not cover damage resulting from accidents, alternations, failure to follow instructions, misuse, unauthorized service, fire, flood, acts of God, or other causes not arising out of defects in material or workmanship.

What we will not do:
WE WILL NOT PAY FOR LOSS OF TIME, INCONVENIENCE, LOSS OF USE OF THE PRODUCT, OR PROPERTY DAMAGE CAUSED BY THIS PRODUCT OR ITS FAILURE TO WORK OR ANY OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES.
THIS WARRANTY SETS FORTH ALL OUR RESPONSIBILITIES REGARDING THIS PRODUCT. REPAIR OR REPLACEMENT AT AN AUTHORIZED SERVICE LOCATION IS YOUR EXCLUSIVE REMEDY. THIS WARRANTY IS THE ONLY ONE WE GIVE ON THIS PRODUCT. THERE ARE NO OTHER EXPRESS OR IMPLIED WARRANTIES INCLUDING, BUT NOT LIMITED TO, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, FROM EMAC INC.

Other Conditions:
If we repair your product, we may use reconditioned replacement parts or materials. If we choose to replace your product, we may replace it with a reconditioned one of the same or equivalent model. Parts used in repairing or replacing the product will be warranted for one (1) year from the date that the product is returned. Product or parts deemed not defective will be replaced or repaired and shipped at your cost.

State Law Rights:
Some states do not allow limitations on how long an implied warranty lasts or the exclusion or limitation of incidental or consequential damages, so the above exclusion or limitations may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Table of Contents

Introduction to Computers

Computers basically perform three functions: inputing data, processing the data and outputing data. Some devices that are commonly used to input information to a computer are keyboards, dip switches and joysticks. Common computer output devices are light emitting diodes (LEDs), liquid crystal displays (LCDs), video monitors, and printers. Disk drives and modems are common devices that have both input and output characteristics. To process the information obtained by the input devices and to control what information will be sent to the output devices we must introduce the next lower level of a computer; the microprocessor.

A microprocessor, which is also referred to as the central processing unit (CPU), processes the information that is input to the computer and determines what data will be sent to the output devices. The microprocessor has within it an arithmetic logic unit (ALU) which performs addition, subtraction, comparisons and logical functions, which will be discussed later.

One thing you must know about computers is that they don't really think like people do. They can only do what the computer manufacturer or the computer user tells them to do. The microprocessor can do many different things, but in order for something useful to be done it must follow a group of instructions called a program. Below is a "program" or a group of instructions that you could write which a person may follow in order to quench their thirst.

1) Get a glass.
2) Get some milk from the refrigerator.
3) Pour the milk in the glass.
4) Drink the milk.
5) If you are still thirsty continue from step three.
6) Put glass in sink
7) put milk in refrigerator.

In the same way, by knowing the microprocessor's language, you can give it a group of instructions that it can perform and it would perform them exactly as you commanded it. The microprocessor can only obey one instruction of a program at a time and these instructions tell the microprocessor whether to input data, output data or perform one of the ALU functions.

Computer Math

A microprocessor performs all arithmetic in binary, although it may be translated to different forms (i.e. decimal, hexadecimal..). The binary number system consists of the numbers 0 and 1 which are called binary digits or bits for short. There are several words used to represent the binary numbers 0 and 1 and they are often used interchangeably, depending on the context in which they are used. They are as follows:

binary 1 =	**true**	**on**	**high**	**set**	**+5 volts**	**set**
binary 0 =	**false**	**off**	**low**	**reset**	**0 volts**	**clear**

In the 8085 microprocessor the binary numbers are organized in groups of 8 bits which are called bytes and groups of 16 bits which are called words. When referring to a byte, it is often necessary to describe particular bits, so the numbering of each of the 8 bits is as follows:

```
7 6 5 4 3 2 1 0
```

So in the binary number, 10001000, bit 7 and bit 3 are 1 and the rest are 0. In binary number 00010001, bit 4 and bit 0 are have a value of 1 and the rest are 0. When referring to a word, the bits are numbered:

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

As you know, in decimal numbers, each position of a digit has a different weight. For example in the decimal number 1732:

(The digit numbers are read from right to left, 0 to 3)

digit #	weight		value	= decimal result
3	10^3	*	1	= 1000
2	10^2	*	7	= 700
1	10^1	*	3	= 30
0	10^0	*	2	= 2
sum of numbers * weights				**= 1732**

In the same way the binary system has weights for each bit position. The weight for a bit is 2 to the power of the bit number ($2^{\text{(bit number)}}$). The binary number 11111111 can be converted to decimal by knowing the weights of each bit, as in the example below:

bit #	weight		value	= decimal result
7	2^7	*	1	= 128
6	2^6	*	1	= 64
5	2^5	*	1	= 32
4	2^4	*	1	= 16
3	2^3	*	1	= 8
2	2^2	*	1	= 4
1	2^1	*	1	= 2
0	2^0	*	1	= 1
sum of bits * weights				**= 255**

The three forms of numbers we will use in this manual are binary, hexadecimal (hex, for short) and decimal. Below is a table of the binary, and hex equivalents of the decimal numbers 0 through 20.

DECIMAL	HEX	BINARY
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011
20	14	10100

Hexadecimal is used to represent binary values because it is very easy to convert numbers from binary to hexadecimal. For example, to convert the following binary number to hexadecimal:

```
101101101101011
```

Start from the right digit and put the number into groups of four binary digits (bits). If there are not enough bits in the number to make a full four bits in the group on the left side, add zeros to the left of the number.

```
0101 1011 0110 1011
```

Now replace the binary groups with their hexadecimal equivalents using the table above and you will get the following result:

```
                              5   B   6   B
```

It is just as easy to convert hex to binary. Merely replace each hex digit with the corresponding 4 binary digits from the table above and you have your binary number, for example:

```
                                 HEX
                              F   C   1   8

                               BINARY
                          1111 1100 0001 1000
```

In this manual you will see the words "least significant" or "low order" and "most significant" or "high order". These refer to the mathematical weight of the part of a number that is being described. In all number systems the digit on the left end is the most significant or high order digit and the digit on the right end is the least significant or low order digit. For example in the binary number 00010010, the bit on the left end is the most significant bit and the bit on the right end is the least significant bit. In the hex word 01FF the left two digits are the most significant byte (MSB) and the two right digits are the least significant byte (LSB).

Computer Logic

The 8085 supports 4 logical operations:

1) The AND operation takes two input bits and returns a 1 bit if both input bits are 1 and a 0 bit if either bit is 0.

2) The OR operation takes two input bits and returns a 1 bit if either input bit is 1 and a 0 bit if both input bits are 0.

3) The XOR operation takes two input bits and returns a 0 bit if the input bits are the same and a 1 bit if they are different.

4) The NOT operation takes one input bit and returns a 1 if the input bit is 0 and returns a 0 if the input bit is 1. This is called complementing or inverting.

The 8085 performs these operations 8 bits at a time, by performing the logic operation on each bit position, For example:

```
        01110010 <-X
AND     10010011 <-Y
        00010010 <-Z
```

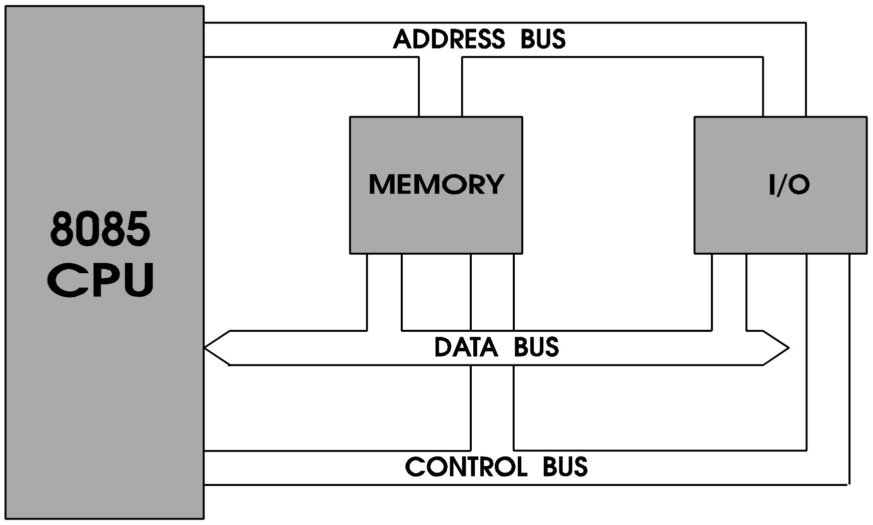
Bit 0 of X is ANDed with bit 0 of Y and gives the result in bit 0 of Z. Bit 1 of X is ANDed with bit 1 of Y and gives the result in bit 1 of Z and the pattern continues on up to bit 7.

The following shows the way logical operations work with bytes:

```
    00010010           01100111          11101101
AND 0 1 0 0 0 0 1 0 OR 10101110      XOR 01111001       NOT  11001010
                       11101111          10010100            00110101
00000010
```

A Brief Overview of the Universal Trainer Hardware

MEMORY



MICROCOMPUTER BLOCK DIAGRAM

The 8085 microprocessor can access 65536 individual memory locations in the range 0 to 65535 in decimal (0 to FFFF in hex) but only one at a time. There are two types of memory in most microprocessor based systems, memory that can be read but not written to, which is called Read Only Memory (ROM), and memory that can be read from and written to, which is called Read/Alter Memory or Random Access Memory (RAM). Both RAM and ROM chips have address pins which are connected to the microprocessor's address pins. These pins are connected through what is called an address bus and through this bus the microprocessor can select a memory location for writing or reading of data. Writing to ROM chips has no effect.

The RAM and ROM chips in the Universal Trainer have eight pins which send data to, or receive data from the microprocessor through a group of eight connections called the data bus. Since there are 8 pins in the RAM and ROM chips, this allows numbers from 0 to 255 (0 to FF in hex) to be read from or written to each memory location.

INPUT/OUTPUT

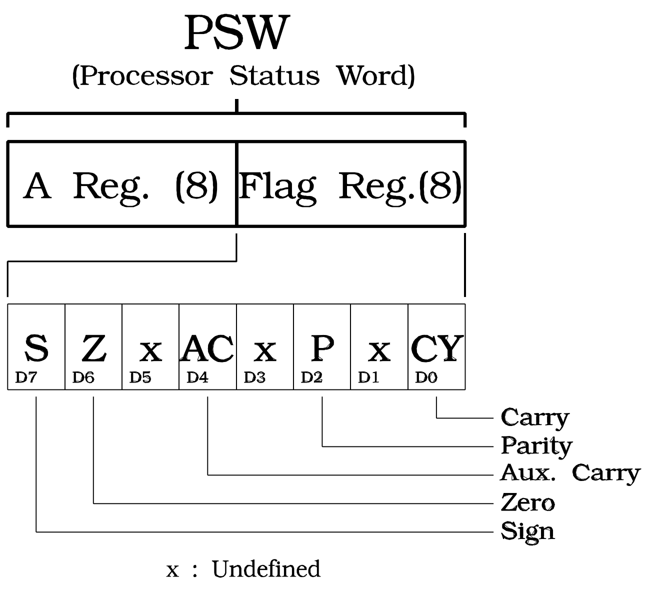
The 8085 microprocessor can also send data to and receive data from chips other than the RAM or ROM. When the microprocessor wants to perform input or output it disables the RAM and ROM chips and sends an input/output (I/O) address to the address bus. The I/O address is only 8 bits but it appears on the lower 8 bits (A0-A7) and the higher 8 bits (A8-A15) of the address bus simultaneously. Since the address generated is only 8 bits long, only I/O addresses from 0-255 (0-FF hex) can be selected. Most microprocessor-based systems have circuitry which decode the address from the address bus and select the appropriate I/O device. Usually these devices are dedicated to either input only or output only. If an input device has been selected, 8 bits of data is transmitted from the input device to the data bus and into the microprocessor. If an output device has been selected, the 8 bits of data is sent from the microprocessor to the data bus and to the output device.

The control bus is a group of connections which provide control over reading or writing of memory or I/O devices. Below is a block diagram showing the way the CPU (microprocessor) connects to the memory and I/O devices through the address bus, data bus and control bus.

REGISTERS

The 8085 microprocessor has within it temporary storage devices called registers. Registers work similarly to RAM in that they store binary values. The 8 bit general purpose registers provided by the 8085 are named A, B, C, D, E, H and L. The A register is often referred to as "the accumulator".

The 8085 has many instructions which use these individual general purpose registers. There are also instructions which view a pair of the general purpose registers as a single 16 bit register. The register pairs that are used in these instructions are BC, DE, and HL. When they are paired with other registers C,E and L represent the least significant 8 bits of the register pairs (bits 0-7) and B,D and H represent the most significant 8 bits (bits 8-15). Some instructions view the A register and flag register (described below) as a 16 bit register called the processor status word (PSW). Within this manual it is also referred to as the "AF" register. The A register is the most significant 8 bits and the flag register is the least significant 8 bits. The PSW is shown with a diagram of the individual bits of the flag register, below.

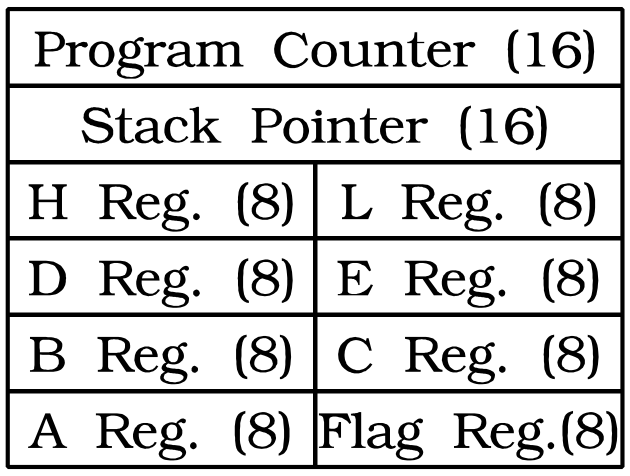


There are also registers that are dedicated to special purposes. The registers and their descriptions are as follows:

The stack pointer (SP) is a 16 bit register which points to a memory location in RAM which will hold temporary values in an area of RAM called the stack. The stack is explained in detail in a later lesson.

The program counter (PC) is a 16 bit register which points to the memory location of the next machine language instruction to be executed.

The flag register is an 8 bit register which has individual bits, called flags, that indicate the result of arithmetic or logical operations. Most of these flags will be described later in the manual.



8085 REGISTERS

Languages Used By the Universal Trainer

8085 Machine Language

The 8085 microprocessor has 246 instructions and each instruction is represented by an 8 bit binary value, which is called an op code or an instruction byte. The Instruction Set Encyclopedia ((c) Intel Corporation) included at the end of this manual, divides these instructions into five general categories which are as follows:

1) Data Transfer Group
Moves data between registers or between memory locations and registers.
2) Arithmetic Group
Adds, subtracts, increments or decrements data in registers or memory.
3) Logic Group
AND's, OR's, XOR's, compares, rotates or complements data in registers or between memory and a register.
4) Branch Group
Initiates conditional or unconditional jumps, calls, returns, and restarts.
5) Stack, I/O, and Machine Control Group
Includes instructions for maintaining the stack, reading from input ports, writing to output ports, setting and reading interrupt masks, and changing the flag register.

Some machine language instructions require a byte or even two bytes of additional information. The microprocessor reads the instruction and determines whether it requires an extra byte or two bytes. If the instruction requires another byte the 8085 will get the byte from the next consequtive address following the instruction byte, and if the instruction requires two bytes the 8085 will get them from the next two consequtive addresses following the instruction byte. In instructions that require two extra bytes, the byte following the op code is the least significant byte and the second byte after the op code is the most significant byte. To help you remember this order, remember that the high order byte is in the higher memory address than the low order byte.

Microprocessors can only run (execute) programs written in machine language. Programs written in languages such as BASIC or assembly language (discussed later) first must be translated to machine language before they can be executed.

Assembly Language

Intel (c) has designed a way to represent the 8085's machine language instructions using words called "mnemonics". Using these mnemonics, a language called "assembly language" was created which allows you to write machine language programs in a more readable form. An assembly language program cannot be understood by the 8085 until it is translated to machine language with a program called an "assembler". In the following lessons, programs will be listed in assembly language followed by the machine language translation of the program.

Below is an example assembly language program, showing the four basic fields of a line of assembly language: label, mnemonic, operand and comment. Note that assembly language programs usually don't have line numbers, but these are included to aid in explaining assembly language.

```
   LABEL    MNEMONIC    OPERAND        COMMENT
1  dips     equu        41h          ; port for the dip switches
2  num      equ         4            ; number of reads of the dip switches
3           org         08F01h       ; starting address of the program
4           lxi         h,dtaspac    ; load the HL register with start
5                                    ; of storage space
6           mvi         c,num        ; load C register with value of num
7  loop:    in          dips         ; load A register with the dip
8                                    ; switch values
9           mov         m,a          ; store the value of A register
10                                   ; at memory address pointed to by HL
11          dcr         c            ; decrement the C register
12          jnz         loop         ; if C<>0, jump to loop
13
14          rst         7            ; return to MOS
15 dtaspac: ds          num          ; set aside the number of bytes
16                                   ; specified by num
17          db        12h,4,1001b,0 ; store this data in memory
18          end                      ; end of the assembly language
```

In order to make assembly language more readable and easier to modify, the provision was included which allows the use of a string of characters called a symbol or label to represent a numerical value. In the program above, the string of characters "dips" represents the value 12 hex, so line 7 which says "in dips" means, load the A register with the data from input port 12 hex (which is the dip switch port). As you can see, this is more readable than its assembly language equivalent "in 12h". The value of a symbol is assigned using the EQU instruction. The symbol on the left of the instruction is assigned the value on the right. As you can see in line 1 "dips" is assigned the value 12 hex and in line 2 "num" is assigned the value 4 decimal which is the same as 04 hex. Most assemblers assume that a number is decimal unless it is otherwise noted. Binary numbers are indicated by ending with a "b" or "B". Hex numbers must begin with a decimal number and end with "h" or "H". If they don't begin with a decimal number, the assembler will think they are labels, as in the case of the hex numbers DEAFh or BADh (they look like words instead of numbers). Start the hex numbers with 0 to solve this problem (0DEAFh, 0BADh).

The ORG mnemonic tells the assembler the starting address in memory of the instructions that follow it. Line 3 shows that the program is to be assembled starting at address 8F01.

The DS mnemonic tells the assembler to set aside the number of bytes of memory specified by the value to the right of the mnemonic. This memory is reserved for the storage of data instead of machine language.

The mnemonic DB takes the data that follows the mnemonic and stores the hex value(s) of the data in memory. The values can be binary, hex or decimal numbers or a mixture of these but each number always uses one byte of memory. If the number is too large to be stored in one byte of memory the assembler will give an error message. In line 17 of the example program above, the first byte of data following the mnemonic is stored at the first memory location following the four bytes reserved by the DS mnemonic. The rest of the data on the line follow it sequentially in memory.

When a symbol is in the label field of a certain line, and the mnemonic for that line isn't "equ", its value will be made the memory address of the op code represented by the mnemonic. In line 7 of the program above, "loop" is assigned the value of the memory address of the op code of the "in" mnemonic. Line 12 uses the value of the label "loop" to produce the machine language for the "jnz" instruction. In line 15 the label "dtaspac" is assigned the value of the memory address of the first byte of the bytes reserved by the DS mnemonic. In line 4 the value of "dtaspac" is used to produce the machine language for the "LXI H" instruction.

Every character after a semicolon is considered a comment. Comments are used to describe the workings of a program to a person who might be reading the assembly language. They have no effect on the program because when the assembler encounters the ';' it ignores the rest of the characters on the current line.

The mnemonic END tells the assembler that this is the end of the program. The mnemonics END, EQU, ORG, DB and DS are all called "pseudo-ops", which means they are not translated into machine language directly. These are used by the assembler to aid in assembling the machine language. Lines 4 through 14 contain mnemonics that actually correlate directly to machine language.

LESSON 1: Using the Monitor Operating System

The Monitor Operating System allows the user to:

- **o** view and change memory contents
- **o** view and change register contents
- **o** view and change stack contents
- **o** execute one instruction at a time
- **o** run a program
- **o** select a software or hardware breakpoint

This lesson will introduce the first two topics listed above and the other four will be introduced in later lessons

VIEWING AND CHANGING MEMORY CONTENTS

When you first turn on the Universal Trainer, the four LED displays on the left (these will be called the word field) will show "8F01" and the display pair on the far right (these will be called the byte field) will show some random hexadecimal byte. The number shown in the word field is the value of the program counter register (PC) and the number shown on the byte field is the data at the memory address ***pointed to*** by the program counter. Since PC = 8F01 it "points" to the data at that memory address, so if the data at that memory address happened to be the hex number DB then "DB" would be shown in the byte field. Below is an illustration the program counter pointing to the data at memory address 8F01. The values shown in the memory addresses in the diagram most likely are not the actual values that are in the memory. This is because the memory contains mostly random values when the Universal Trainer is turned on; The values were chosen just for examples.

```
ADDRESS     DATA
8F01        DB     <-PC
8F02        12
8F03        D3
:
: (memory addresses 8F04-FFFF)
```

When an address and the data at that address is displayed, the Universal Trainer is in "data entry mode". Press the "enter" key and the PC register will be incremented to 8F02 and shown in the world field and the data pointed to by the new value of the PC will be shown in the byte field. If the data at memory address 8F02 happened to be 12 hex, then "12" would be shown byte field. The program counter pointing to the data at memory address 8F02 is illustrated below.

```
ADDRESS     DATA
8F01        DB
8F02        12     <-PC
8F03        D3
:
: (memory addresses 8F04-FFFF)
```

Press "enter" as many times as you want. Press the "DEC" key and the value displayed for PC will be decremented and the number in the byte field is the byte of data pointed to by the new PC memory address. Press the "DEC" key until PC is 8F01 again or press the reset button.

Type "F" and "C", and you will see the hex number "FC" in the byte field. If you type a wrong digit, or you want to change the value you typed, just type the two correct hex digits and they will overwrite the others. Now type "0" and "7" and press "enter" and the PC will be incremented and the PC value 8F02 will be displayed along with the data pointed to by the new value of PC. Type the number 55 and press the "DEC" key. You will see that the data at address 8F01 is now 07. Press the "enter" key and you will see that the data at address 8F02 is <u>not</u> 55 as was typed before. This is because hex numbers that are typed aren't stored until the "enter" key is pressed. This is a useful feature when you have typed a number and decide you do not want it to be stored in memory or wish not to change the original value.

LOADING A PROGRAM INTO MEMORY

In the lessons that follow, it is necessary to load programs into memory. The machine language for the programs are listed in the same format as the following:

```
ADDRESS     DATA        INSTRUCTION
8F01        DB          IN   41
8F02        41
8F03        D3          OUT  40
8F04        40
8F05        C3          JMP 8F01
8F06        01
8F07        8F
```

Before entering a program into memory, press the reset button. This resets the general purpose registers and flag register to zero and sets the stack pointer to FFD4 and the program counter to 8F01. Look at the program data table above. In order for a machine language program to be loaded into memory properly, the addresses under the column marked "ADDRESS" must contain the data to the right of them in the column marked "DATA" after you have completed loading the data. The column marked "instruction" just tells what instruction the data stands for, so this can be ignored. Since the PC value is 8F01, type DB, which is the data from the table that belongs in that address, and press "enter". The PC is now 8F02 so type 41 and press enter. Continue typing the data which belongs in the current PC address and pressing enter until all the addresses listed in the table have been loaded. Now press the "dec." key and verify that the data at the current PC address is the same as the data in the program data table. If the data is not the same, just type the correct number and press enter and then continue pressing the "dec." key and verifying data until the program counter is 8F01 again. Once you become comfortable with entering programs into memory you may want to skip the verification process. Don't be alarmed if you enter a program incorrectly and it doesn't work. Just press the reset button and examine the program for mistakes. Sometimes if a program is not entered correctly, the program will execute so erraticly that it will be corrupted. In this case you may have to enter the entire program into memory again.

VIEWING AND CHANGING REGISTER CONTENTS

MOS allows you to examine and change the values of the 8085's registers as well as the Stack Contents (SC), Software BreakPoint (SBP) and Hardware BreakPoint (HBP). To view the contents of the register just press the button corresponding to that register. To view and change the HBP or SBP you must first press the "FUNC" key (which will cause the display to show "FUNCTION") and then the "E" key for HBP or "F" for SBP.

Each line in the table below shows the key(s) to press and the data that will be shown in the word field as a result.

<u>**WORD FIELD**</u>

KEY		**DATA SHOWN ON FIRST PAIR OF DISPLAYS FROM THE LEFT**	**DATA SHOWN ON SECOND PAIR OF DISPLAYS ON THE LEFT**
A/F	=	A register	flag register
B/C	=	B register	C register
D/E	=	D register	E register
H/L	=	H register	L register
PC	=	High order byte of PC	Low order byte of PC
SP	=	High order byte of SP	Low order byte of SP

Pressing the above keys will cause the value in the registers to be shown in the word field and the register name to be shown to the right. After this, pressing "FUNC" twice or pressing "DEC" or "ENT" will return MOS to the data entry mode.

If you want to change the value of the A register, press the "A/F" key and as a result the displays show "0000 A/F". To change the value of the A register to 1F hex without changing the flag register, press the following keys in order: "1","F","0","0","ENT". After pressing the "ENT" key the display will no longer show the A register and flag register, instead it will return to the data entry mode. View the PSW (the A register and flag register) again by typing the A/F" key. You will see that the values that were entered before are still in the registers. Notice that in order not to change the value of the flag register, it was necessary to enter its value again (00). If the flag register had been 40 and you wanted to change the A register to 2B without changing the flag register you would have to press the following keys (don't do this, this is just an example): "2", "B", "4", "0", "ENT".

Now we will change the value of the flag register and we will assume that it doesn't matter what the value of the A register is. To make the flag register 41 hex type "4", "1". Notice that after pressing these two keys, the two zeros have been shifted to the A register display. Now press "ENT", then view the A register and flag register again and you will see the new value (0041 hex) that was just entered.

As described earlier, the flag register is an 8 bit register which has individual bits called flags, that indicate the result of arithmetic or logical operations. To see what the values of the individual bits are, you must convert the hexadecimal value of the flag register to binary. If you don't remember how to do this, refer back to the section on Computer Math in **Introduction to Computers**. For example, to find out what the current flag values are, convert 41 hex to binary.

```
        HEX
      4     1

      BINARY
 0 1 0 0    0 0 0 1
 │ │ │ │    │ │ │ └──────────── Carry Flag
 │ │ │ │    │ │ └────────────── undefined
 │ │ │ │    │ └──────────────── Parity Flag
 │ │ │ │    └────────────────── undefined
 │ │ │ └──────────────────────── Auxiliary Carry Flag
 │ │ └────────────────────────── undefined
 │ └──────────────────────────── Zero Flag
 └────────────────────────────── Sign Flag
```

According to the diagram above, the Carry and Zero flags have a value of 1 and the Sign, Auxiliary Carry, and Parity flags have a value of 0.

SOME TIPS ABOUT CHANGING REGISTERS

You can point the PC (program counter) register to any location by changing the PC register's contents. This eliminates the need of pressing the "ENT" or "DEC" keys many times in order to view or change data at memory addresses that are distant from the current address.

If you are typing a new value for a register and you haven't pressed "ENT" since you started doing this, you can press the "Func" key twice or press the "dec." key to return to the data entry mode without changing the register.

LESSON 2 : Running Your First Program

NEW INSTRUCTIONS

IN <byte> op code = DB
Load the A register with the data that is on the input port specified by the byte following the instruction. No flags are affected.

OUT <byte> op code = D3
Send the data in the A register to the output port specified by the byte following the instruction. No flags are affected.

JMP <addr> op code = C3
Load the program counter (PC) with the address contained in the two bytes following the instruction. The first byte following the op code is the least significant byte of the address, the second byte is the most significant byte of the address. No flags are affected.

Below is an example 8085 program written in assembly language. This program reads the binary value of the dip switches and outputs the value to the Digital Out Status LEDs and then repeats these instructions indefinitely. The program may be stopped by pressing the reset button.

ASSEMBLER PROGRAM

```
leds        equ   40h
dips        equ   41h

            org   08f01h
loop:       in    dips        ; load A register with
                              ; dip switch values
            out   leds        ; output A register to LEDs
            jmp   loop        ; jump to loop
            end
```

MACHINE LANGUAGE PROGRAM

ADDRESS	DATA	INSTRUCTION
8F01	DB	IN 41
8F02	41	
8F03	D3	OUT 40
8F04	40	
8F05	C3	JMP 8F01
8F06	01	
8F07	8F	

To run this machine language program, turn on the Universal Trainer and the program counter address 8F01 will be shown on the left four displays. Load the machine language data from the above table into memory. Once the whole program has been entered correctly return the program counter address to 8F01 by pressing the reset button or loading the PC register with 8F01, then run the program by pressing the "RUN" key. You should see that as you move the dip switches the Digital Out Status LEDs will turn off or on accordingly. Now press the reset button and the program will stop, the Digital Out Status LEDs will turn off and the display will show "8F01 DB"

The MOS allows you to execute a single machine language instruction starting at the address in the PC register; this procedure is called single stepping. After the instruction is completed, the PC register points to the address of the next instruction and the MOS will be returned to data entry mode. To single step the first instruction, press the "stp/run" key and the displays will show "8F03 d3" which is the new value for PC and the data pointed to by PC. The IN instruction was executed and the PC register was returned with the address of the next instruction, the OUT instruction. To single step the rest of the program, do the following:

1) Single step again and the display will show "8F05 C3". the <u>OUT 11</u> instruction has been executed and the PC register was returned with the address of the <u>JMP 8F01</u> instruction.

2) Single step again and the display will show "8F01 db". This display shows 8F01 because that is the value that was loaded into the PC register by the <u>JMP 8F01</u> instruction. You can see that the mnemonic came from the instructions ability to cause the program counter to "jump" to another location. PC now points to the <u>IN 41</u> instruction again.

3) Single step again and the display will show "8F03 d3". The <u>IN</u> instruction has been executed and the PC register was returned with 1the address of the <u>OUT</u> instruction.

Right after step three, the value of the dip switch has been loaded into the A register and it can be viewed by pressing the "A/F" key. The value of the A register will be shown in the pair of digits on the left. Change the value of the A register (in this program the flag values don't matter). Now do step one and you will see the Digital Out Status LEDs now show the new value of the A register. If you press "Step" three more times, the IN instruction will change the A register to the value of the dip switches again and the OUT instruction will display the new value of the A register on the Digital Out Status LEDs.

LESSON 3: Loading Registers and Transferring Data Between Registers.

NEW INSTRUCTIONS

MVI A,<byte> op code = 3E
MVI B,<byte> op code = 06

MVI C,<byte> op code = 0E
MVI D,<byte> op code = 16
MVI E,<byte> op code = 1E
MVI H,<byte> op code = 26
MVI L,<byte> op code = 2E
Load the register that follows the MVI mnemonic with the byte value following the op code. No flags are affected.

XCHG op code = EB
Exchange the value in register D with H and the value in register E with L. No flags are affected.

MOV A,B op code = 78
Copy the B register to the A register. No flags are affected.

MOV B,C op code = 41
Copy the C register to the B register. No flags are affected.

RST 7 op code = FF
The address following the RST 7 instruction is pushed on the stack and then the instructions starting at address 0038 are executed. No flags are affected.

The program below demonstrates the instructions used to load registers and the instructions used to move values between registers.

ASSEMBLER PROGRAM

```
            org   08F01h
            mvi   a,1   ; load A with 1
            mvi   b,2   ; load B with 2
            mvi   c,3   ; load C with 3
            mvi   d,4   ; load D with 4
            mvi   e,5   ; load E with 5
            mvi   h,6   ; load H with 6
            mvi   l,7   ; load L with 7
            xchg        ; exchange the value of DE with HL
            xchg        ; do it again
            mov   a,b   ; copy B to A
            mov   b,c   ; copy C to B
            rst   7     ; return to MOS
            end
```

MACHINE LANGUAGE PROGRAM

```
      ADDRESS     DATA        INSTRUCTION
      8F01        3E          MVI   A,1
      8F02        01
      8F03        06          MVI   B,2
      8F04        02
      8F05        0E          MVI   C,3
      8F06        03
      8F07        16          MVI   D,4
      8F08        04
      8F09        1E          MVI   E,5
      8F0A        05
      8F0B        26          MVI   H,6
      8F0C        06
      8F0D        2E          MVI   L,7
      8F0E        07
      8F0F        EB          XCHG
      8F10        EB          XCHG
      8F11        78          MOV   A,B
      8F12        41          MOV   B,C
      8F13        FF          RST   7
```

Enter the program into memory and then press the reset button which will change the program counter to 8F01. If you examine registers A,B,C,D,E,H and L you will see that pressing the reset button clears them to 0.

To examine this program we will use a software breakpoint. A breakpoint is an address in memory where you desire the program to stop (or break) and return to the data entry mode. This allows you to run part of your program at full speed and then stop before executing the instruction that the breakpoint points to so you can examine registers or single step or other things. The software break is performed by the MOS (Monitor Operating System) program using the RST 7 software interrupt. When the user specifies a break address the MOS replaces the op code at this address with the RST 7 (FF hex) instruction. The user's program can then execute full speed until encountering the RST 7 instruction. When the user's program reaches the break, the RST 7 instruction returns control to the MOS program. The MOS program then restores the op code that was replaced by the RST 7 instruction. The MOS program only allows the use of one breakpoint at a time, and this is automatically reset to 0000 after the breakpoint has been encountered. The user can however hand insert breakpoints by placing RST 7 instructions at strategic locations throughout the user's program. Remember that any hand inserted breakpoints must also removed by hand when they are no longer needed. When inserting a software breakpoint it is important to remember that if the program execution never reaches the breakpoint address it will not stop. All breakpoints must point to op codes before they will work. If the breakpoint address is pointing to a byte other than the op code in a two or three byte instruction the program will not stop at the breakpoint address. Not only will the program not stop, the program will also not work properly because one of the bytes in the two or three byte instruction will have been changed to FF hex. All breakpoint addresses must be above 8000 hex. Those below 8000 hex will not stop execution, even if the op code at that address is executed.

To select a breakpoint, press the "FUNC" key followed by the "SBP" key and the display will show the breakpoint address in the in the word field and "SBP" to the right of it. A breakpoint is selected the same way you change a register and all the rules regarding changing registers apply to setting a breakpoint; just type the address and press "ENT". Remember that SBP (software breakpoint) is not a register within the 8085 microprocessor, it is just a function supported by the MOS. Follow these steps:

CURRENT PC

8F01	Set a breakpoint at address 8F0F and run the program. After the breakpoint occurs the MOS will be in data entry mode displaying the address 8F0F. Examine the registers and you will see that A,B,C,D,E,H and L have been loaded with 1 - 7 respectively. Remember that when you examine the A register the flag register is also shown, so the displays will show "0100 A/F".
8F0F	Single step and the first <u>XCHG</u> instruction will be executed. Examine the DE and HL register pairs and you will see the previous value of HL is now in DE and the previous value of DE is now in HL.
8F10	Single step and the second <u>XCHG</u> instruction will be executed which will exchange DE with HL again. Examine the DE and HL register pairs and you will see that their values are the same as before the first <u>XCHG</u> instruction.
8F11	Single step and the instruction <u>MOV A,B</u> will be executed which copies the B register to the A register. Examine these registers to verify this.
8F12	Finally, single step the <u>MOV B,C</u> instruction and the C register will be copied to the B register. Examine the BC register pair.
8F13	This is the end of the program.

The last instruction in the machine language program is FF which is the <u>RST 7</u> instruction. Read the instruction's definition at the beginning of this lesson, again (stack will be explained later). The Universal Trainer's ROM has instructions at address 0038 which return control of the microprocessor to the MOS. Try to single step the <u>RST 7</u> instruction; As you can see nothing happens. Now press the reset button, then run the program by pressing the "STP/RUN" key (don't press "FUNC") and you will see that the program stops at the address of the <u>RST 7</u> instruction and returns to the data entry mode showing "8F13 FF" on the displays.

LESSON 4: Eight Bit Addition

NEW INSTRUCTIONS

ADD C op code = 81
The C register is added to the A register. The Z,S,P,CY and AC flags are affected.

MOV A,B op code = 78
The contents of the B register are copied to register A. No flags are affected.

This program adds the B and C registers together and stores the result in the A register. This is done by copying the value of B to A and then adding the value of C to A.

```
        org   08F01h
loop:   mov   a,b    ; A is a copy of B
        add   c      ; Add C to A
        jmp   loop   ; jump to loop
        end
```

```
    ADDRESS   DATA      INSTRUCTION
    8F01      78        MOV   A,B
    8F02      81        ADD   C
    8F03      C3        JMP   8F01
    8F04      01
    8F05      8F
```

To test the ADD C instruction, load the program into memory, press the reset button and do the following:

1) The B and C registers will be loaded with numbers whose sum will not be greater than 9 so that the result will not have to be translated to decimal to see that the instruction worked properly. Load the BC register pair with 0403, which is the same as loading B with 4 and C with 3, then single step to address 8F03. Now, if you examine the A register you will see that it has the value 7, and if you convert the flag register to binary you will see that bit 0 (the carry flag) is 0 which means that the result of the addition was small enough to fit in the A register.

2) Load the BC register pair with FFFF then single step the instruction at 8F03 (JMP 8F01) and then single step to address 8F03 again. By examining the A register and flag register you will see that the A register is now FE and that bit 0 (the carry flag) of the flag register is now 1 which indicates that the result of the addition was too big to fit in the A register.

LESSON 5: Eight Bit Subtraction

NEW INSTRUCTION

SUB C op code = 91
The C register is subtracted from the A register. The Z,S,P,CY and AC flags are affected.

If the previous program is still in memory and you change the instruction at 8F02 to 91 (SUB C) the following program is the result. This program will subtract the C register from the B register and store the result in the accumulator. This is done by copying the value of B into A and then subtracting C from A. The result will be in the A register.

```
        org   08F01h
loop:   mov   a,b    ; A is a copy of B
        sub   c      ; subtract C from A
        jmp   loop   ; jump to loop
        end
```

```
    ADDRESS   DATA      INSTRUCTION
    8F01      78        MOV   A,B
    8F02      91        SUB   C
    8F03      C3        JMP   8F01
    8F04      01
    8F05      8F
```

To test the SUB C instruction, load the program into memory and move the program counter to 8F01. Load the BC register pair with 0906, which is the same as loading B with 9 and C with 6, then single step to address 8F03. You will notice that the A register is now 3 and if you convert the flag register to binary you will see that the carry flag is 0. Single step to 8F01 then load B with 1F and C with 3D then single step to 8F03. You will see that the A register is E2 and the carry flag is 1. In every subtraction the carry flag is set if the number in the A register is smaller than the value being subtracted from it (in this case the A register was 1F and 3D was subtracted from it). The carry flag could more appropriately be referred to as a "borrow flag" when the flag value is the result of a subtraction.

LESSON 6: Sixteen Bit Subtraction

NEW INSTRUCTIONS

SUB E op code = 93
The E register is subtracted from the A register. The Z,S,P,CY and AC flags are affected.

SBB D op code = 9A
Subtract the D register and the carry flag from the A register. The Z,S,P,CY and AC flags are affected.

MOV A,C op code = 79
The contents of the C register are copied to register A. No flags are affected.

MOV C,A op code = 4F
The contents of the A register are copied to register C. No flags are affected.

MOV B,A op code = 47
The contents of the A register are copied to register B. No flags are affected.

Using the carry flag with subtraction instructions allows you to subtract 16 bit, 24 bit or even larger numbers. The following program subtracts the value of the DE register pair from the value of the BC register pair, leaving the result in the latter.

```
        org   08f01h
                    ; subtract low order bytes 1st
        mov   a,c   ; A is a copy of C
        sub   e     ; A = A - E. Set carry flag if A < E
        mov   c,a   ; C = A - E
                    ; subtract high order bytes and carry flag
        mov   a,b   ; A is a copy of B
        sbb   d     ; A = A - D - carry flag
        mov   b,a   ; B = A
        rst   7     ; stop the program
        end
```

ADDRESS	DATA	INSTRUCTION
8F01	79	MOV A,C
8F02	93	SUB E
8F03	4F	MOV C,A
8F04	78	MOV A,B
8F05	9A	SBB D
8F06	47	MOV B,A
8F07	FF	RST 7

Load the BC register pair with 2100 hex and load the DE register pair with 10FF hex. The following is an equation that the program will perform.

```
 2100 hex
-10FF hex
=1001 hex
```

If you single step the program starting at address 8F01 you will see the following after each step:

CURRENT PC

8F01	A equals C which is 00.
8F02	A = A - E. Since A (00) is less than E (FF), the carry flag is set (examine bit 0 of the flag register). The subtraction will be performed as if there was a 1 digit to the left of the A register. In this case the subtraction will be performed as if A = 100 hex. (100 hex - FF hex) = 01 so A equals 01.
8F03	C equals A which is 01.
8F04	A equals B which is 21 hex.
8F05	The D register and the Carry flag are subtracted from the A register (A = A - D - carry flag = 21 hex - 10 hex - 1 = 10 hex).
8F06	B equals A which is 10 hex
8F07	This is the end of the program.

Examine the BC register pair and you will see that it is now 1001 hex just as predicted in the equation that was shown. Try pressing the reset button and loading BC and DE with other values such that the C register is greater than the E register and single step the program again.

LESSON 7: Sixteen Bit Addition

NEW INSTRUCTIONS

ADD E op code = 83
Add the E register to the A register. The Z,S,P,CY and AC flags are affected.

ADC D op code = 8A
Add the D register and the carry flag to the A register. The Z,S,P,CY and AC flags are affected.

The same concept used in the previous lesson can be applied to make a 16 bit addition program.

```
org   08f01h
            ; add low order bytes 1st
mov   a,c   ; A is a copy of C
add   e     ; A = A + E. Set carry if result > 0FFh
mov   c,a   ; C = A + E
            ; add high order bytes and carry flag
mov   a,b   ; A is a copy of B
adc   d     ; A = A + D + carry flag
mov   b,a   ; B = A
rst   7     ; stop the program
end
```

```
ADDRESS     DATA        INSTRUCTION
8F01        79          MOV  A,C
8F02        83          ADD  E
8F03        4F          MOV  C,A
8F04        78          MOV  A,B
8F05        8A          ADC  D
8F06        47          MOV  B,A
8F07        FF          RST  7
```

If the previous program is still in memory the only changes that need to be made are that the SUB instruction must be replaced with an ADD, and the SBC with an ADC. This is done by changing the byte at address 8F02 to 83 and the byte at address 8F05 to 8A. If the program is no longer in memory, load the one listed above.

Load the BC register pair with 2302 hex and load the DE register pair with 10FF hex. The following is an equation that the program will perform.

```
 2302 hex
+10FF hex
=3401 hex
```

If you single step the program starting at address 8F01 you will see the following after each step:

CURRENT PC

8F01	A equals C which is 02.
8F02	A = A + E. Since A + E = 101 hex and that is too big to fit in the A register, the carry flag is set (examine bit 0 of the flag register) and the A register is 01.
8F03	C equals A which is 01.
8F04	A equals B which is 23 hex.
8F05	The D register and the carry flag are added to the A register (A = A + D + carry flag = 23 hex + 10 hex + 1 = 34 hex).
8F06	B equals A which is 34 hex
8F07	This is the end of the program.

Examine the BC register pair and you will see that it is now 3401 hex just as predicted in the equation that was shown. Try pressing the reset button and loading BC and DE with other values and running the program again. Try loading the C and E registers with values that will not set the carry flag when the ADD E instruction is executed.

LESSON 8: Sixteen Bit Subtraction Using Two's Complement Addition

NEW INSTRUCTIONS

DAD B op code = 09
Add the 16 bit number in BC to the 16 bit number in HL and store the result in HL. The carry flag is 1 if the result is too big to fit in HL.

CMA op code = 2F
Complement the bits in the A register (i.e. change ones to zeros and zeros to ones). No flags are affected

INX B op code = 13
Increment the value in the BC register pair. No flags are affected

The 8085 microprocessor's DAD instruction provides a quicker method of 16 bit addition. The following is a simple program that adds the BC register pair to the HL register pair. Try running the program with HL = 01FF and BC = 7001, the result in HL will be 7200 hex.

```
ADDRESS    DATA      INSTRUCTION
8F01       09        DAD  B
8F02       FF        RST  7
```

There is no instruction for 16 bit subtraction, but this can be simulated by changing the register that you want to subtract to its two's complement form and then executing a DAD instruction. A number is changed to two's complement form by complementing every bit and then incrementing the number.

Below is a program that subtracts BC from HL by changing BC to its two's complement form, then the program adds BC it to HL repeatedly. Note that the result in the carry flag will not indicate a borrow as it did in the <u>SUB</u> and <u>SBC</u> instructions.

```
            org   08f01h
            mov   a,b    ; put B in A
            cma          ; so it can be complemented
            mov   b,a    ; put the new value in B
            mov   a,c    ; Do the same for C
            cma
            mov   c,a    ; BC has been complemented (1's complement)
            inx   b      ; now increment the BC register pair which will make
                         ;   it 2's complement
loop:       dad   b      ; add BC to HL
            jmp   loop   ; add it again
            end
```

```
ADDRESS    DATA      INSTRUCTION
8F01       F3        MOV  A,B
8F02       2F        CMA
8F03       47        MOV  B,A
8F04       79        MOV  A,C
8F05       2F        CMA
8F06       4F        MOV  C,A
8F07       03        INX  B
8F08       09        DAD  B
8F09       C3        JMP  8F08
8F0A       08
8F0B       8F
```

Enter the above program into memory, then load HL with FFFF and BC with 00FF. Set a software breakpoint at 8F08 then run the program. When the program breaks, you will see that BC is now the two's complement of 00FF which is FF01. Single step the <u>DAD</u> instruction and HL will be FF00 (FFFF - FF), single step the <u>JMP</u> and <u>DAD</u> again and HL will be FE01 (FF00 -FF) and so on. With each <u>DAD</u> instruction 00FF is subtracted from HL.

LESSON 9: Binary Coded Decimal 16 Bit Addition

NEW INSTRUCTION

DAA op code = 27

Change the value in the A register to two binary coded decimal digits. This is done by: (a) Adding 6 to the A register if the value of its lower 4 bits is greater than 9, or if the auxiliary carry flag is set. (b) Adding 6 to the upper 4 bits of the A register if they are greater than 9 or if the carry flag is set. The Z,S,P,CY, and AC flags are affected.

NOP op code = 00

This instruction does nothing but take up one byte in a program. No flags are affected.

Decimal numbers can be stored in binary form as binary coded decimal (BCD) numbers. Each decimal digit takes 4 bits just as hex numbers do, but in BCD, digits A-F are not valid numbers.
For example, the hex number 9999 is 9999 in BCD. The DAA instruction was provided so it would be possible to perform binary coded decimal math using the ordinary hex addition instructions.

According to its definition, DAA will give the following results following a hex addition instruction:

```
 19                   43                   67
+29(hex add)         +72(hex add)         +75(hex add)
 32                   B5                   DC
+06(DAA)             +60(DAA)             +66(DAA)
 48                  115                  142
```

Note that DAA can only give 2 decimal digits and the carry flag as a result. In these examples the carry flag represents the hundreds digit.
The following program will add the BCD number in DE to the BCD number in BC and store the BCD result in BC.

```
        org   8f01h
                    ; add low order bytes 1st
        mov   a,c   ; A is a copy of C
        add   e     ; A = A + E. Set carry flag if A<E
        daa         ; decimal adjust accumulator
        mov   c,a   ; C = A + E
                    ; add high order bytes and carry from DAA
        mov   a,b   ; A is a copy of B
        adc   d     ; A = A + D + carry flag
        daa         ; decimal adjust accumulator
        mov   b,a   ; B = A
        rst   7     ; stop the program
        end
```

```
ADDRESS      DATA        INSTRUCTION
8F01         79          MOV  A,C
8F02         83          ADD  E
8F03         27          DAA
8F04         4F          MOV  C,A
8F05         78          MOV  A,B
8F06         8A          ADC  D
8F07         27          DAA
8F08         47          MOV  B,A
8F09         FF          RST  7
```

Enter the program into memory then press reset and load BC with 1934 and load DE with 1879.

CURRENT PC	
8F01	Single step to 8F03 and examine the A register (it will be AD hex). Since both the upper 4 and lower 4 binary digits of the A register are greater than 9 then 66 will be added to it in the DAA instruction that follows.
8F03	Single step the DAA instruction and examine the A register and you will see that 66 was added to it which set the carry flag and made the A register 13.
8F04	Single step to 8F07 and examine the A register (it will be 32) and flag register. The auxiliary carry flag will be 1 and the carry flag will be 0 so the DAA instruction that follows will add 6 to the A register.
8F07	Single step the DAA instruction at this address and by examining the A register you will see that 6 was added to it.
8F08	Single step and the A register will be copied to the B register.
8F09	This is the end of the program.

Verify that the BC register is 3813, which is the sum of the decimal numbers 1934 and 1879. Now replace the DAA instructions at 8F03 and 8F07 with 00, which is an instruction that does nothing. Load DE and BC with the same values as before then run the program again. This time the BC and DE registers will be added as if they were hex numbers and the result will be 31AD hex.

LESSON 10: Multiplication

NEW INSTRUCTIONS

RAL op code = 17
Shift all the bits in the accumulator one position left and put the carry flag in bit 0 and put the value that was shifted out of bit 7 into the carry flag. Only the carry flag is affected.

DAD H op code = 29
Add HL to HL, if the result is greater than 16 bits, the carry flag will be 1, otherwise 0. Only the carry flag is affected.

DAD D op code = 19
Add DE to HL, if the result is greater than 16 bits, the carry flag will be 1, otherwise 0. Only the carry flag is affected.

ORA A op code = B7
This instruction logically ORs the A register with itself. The CY and AC flag will be 0, and the Z,S and P flags will be affected according to the value of the A register.

The following program illustrates the use of a rotate instruction as a method of multiplication. Shifting the digits of a base 10 (decimal) number left is the same as multiplying the number by its base, which is 10. In the same way, in the base 2 (binary) number system, shifting a binary number left is the same as multiplying it by its base, which is 2. For example:

```
      DECIMAL
            1423  shifted left 1 digit    = 14230
            1423    *   10                = 14230

      BINARY
            0111  shifted left 1 digit    = 01110
            0111    *   0010              = 01110
            below is the example converted to decimal
            (7      *   2                 = 14)
```

By using a more than one left shift on a number you can effectively multiply the number by 4, 8, 16, 32, 64, 128 and so on. The number of times you shift a number left determines the amount the original number was multiplied by. For example if you shift a number right 3 times, it is the same as multiplying it by 2*2*2 or 8.

Often when doing multiplication using left shifts, the result won't fit in an 8 bit register. You can write programs which shift bits from register to register which can allow shifting of very large numbers. Below is a program which shifts the bits in the DE register pair left one bit position.

```
            org   8f01h
loop:       mov   a,e   ; load A with low byte first
            ora   a     ; clear the carry flag (cy=0)
            ral         ; rotate left through carry
            mov   e,a   ; store in E
            mov   a,d   ; load A with high byte
            ral         ; whatever was shifted out of bit 7
                        ; in the last RAL will be put in the CY flag
                        ; which is shifted into bit 0 in this RAL
            mov   d,a   ; store in D
            rst   7     ; return to MOS
            end
```

```
      ADDRESS     DATA        INSTRUCTION
      8F01        7B          MOV  A,E
      8F02        B7          ORA  A
      8F03        17          RAL
      8F04        5F          MOV  E,A
      8F05        7A          MOV  A,D
      8F06        17          RAL
      8F07        57          MOV  D,A
      FF08        FF          RST  7
```

Load the above program into memory then load the DE register pair with 0080 hex (128 decimal) and run the program. This value is chosen because it will show that after you run the program, bit 7 of E is shifted into bit 0 of D, giving a result of 0100 (128 * 2 = 256 decimal). Examine the DE register pair to verify this. Run the program using other values for DE and verify that the program really does multiply DE by 2, but make sure values are less than 8000 hex or the result won't fit in the DE register pair.

If you want to multiply the HL register pair by 2 the DAD H instruction can be used. It adds HL to HL which is the same as multiplying HL by 2. Below is a simple program which illustrates the DAD H instruction.

```
            org   8f01h
loop:       dad   h           ; add hl to hl
            jmp   loop        ; do it again

      ADDRESS     DATA        INSTRUCTION
      8F01        29          DAD  H
      8F02        C3          JMP  8F01
      8F03        01
      8F04        8F
```

Load the program into memory and press reset then perform the same tests that were done for the previous example by loading the HL register pair with the same values that DE was loaded with and single stepping. The results will be the same.

A quick "times 10" algorithm can be made using a combination of multiply by 2 instructions and an addition instruction. Below is the equation "x = a * 10" which is broken down into a form which the 8085 can easily handle:

$$x = a * 10$$
$$x = (a * 5) * 2$$
$$x = ((a*4) + a) * 2$$
$$x = ((a*2*2) + a) * 2$$

The last equation above is translated into the following program.

```
            org   8f01h
            mov   d,h   ; put original value of
            mov   e,l   ;  HL into DE
            dad   h     ; HL = HL * 2
            dad   h     ; HL = HL * 2
            dad   d     ; HL = HL + DE
            dad   h     ; HL = HL * 2
            rst   7     ; return to MOS
            end

      ADDRESS     DATA        INSTRUCTION
      8F01        54          MOV   D,H
      8F02        5D          MOV   E,L
      8F03        29          DAD   H
      8F04        29          DAD   H
      8F05        19          DAD   D
      8F06        29          DAD   H
      8F07        FF          RST   7
```

Load the program into memory then load HL with 0003. After running the program, HL will be 001E which is 30 decimal.

Change the program counter back to 8F01 (don't press reset, or the registers will be cleared) then run the program again. This time HL will be 012C which is 300 decimal.

In the lesson "Using Monitor Operating System Subroutines" is a description of a multiplication service which multiplies two 16 bit numbers.

LESSON 11: Division

NEW INSTRUCTIONS

RAR op code = 1F
Shift all the bits in the accumulator one position right and put the carry flag in bit 7 and put the value that was shifted out of bit 0 into the carry flag. Only the carry flag is affected.

MOV A,D op code = 7A
The contents of the D register are copied to register A. No flags are affected.

MOV D,A op code = 57
The contents of the A register are copied to register D. No flags are affected.

MOV A,E op code = 7B
The contents of the E register are copied to register A. No flags are affected.

MOV E,A op code = 5F
The contents of the A register are copied to register E. No flags are affected.

Just as shifting a number to the right is the same as multiplying it by its base, shifting a number to the left is the same as dividing the number by its base. This allows the 8085 to perform division by 2 by shifting a binary number to the right. When division is done this way, the remainder will be the bit shifted out of the lowest bit position. The following program divides the number in DE by two, leaving the quotient in DE and the remainder in the carry flag.

```
            org   8f01h
loop:       mov   a,d   ; load A with high byte first
            ora   a     ; clear the carry flag (cy=0)
            rar         ; rotate right through carry
            mov   d,a   ; store in D
            mov   a,e   ; load A with low byte
            rar         ; whatever was shifted out of bit 0
                        ; in the last RAR will be put in the CY flag
                        ; which is shifted into bit 7 in this RAR
            mov   e,a   ; store in E
            rst   7     ; return to MOS
            end
```

```
      ADDRESS     DATA        INSTRUCTION
      8F01        7A          MOV  A,D
      8F02        B7          ORA  A
      8F03        1F          RAR
      8F04        57          MOV  D,A
      8F05        7B          MOV  A,E
      8F06        1F          RAR
      8F07        5F          MOV  E,A
      8F08        FF          RST  7
```

Load the program into memory, load the DE register pair with 0007 hex and then run the program. You will see that DE is now 3 and if you examine the A/F register you will see that the carry flag is 1 which indicates a remainder of 1. Try other values of DE and verify that the program actually does divide the DE register by 2.

In the lesson "Using Monitor Operating System Subroutines" is a description of a division service which divides a value in the HL register pair by the value in the DE register pair.

LESSON 12: Using Logic Instructions

NEW INSTRUCTIONS

ANI **<byte>** op code = E6
Logically AND the A register with the byte following the op code and store the result in the A register. The Z,S,P,CY and AC flags are affected.

XRI **<byte>** op code = EE
Logically XOR the A register with the byte following the op code and store the result in the A register. The Z,S,P,CY and AC flags are affected.

ORI **<byte>** op code = EE
Logically OR the A register with the byte following the op code and store the result in the A register. The Z,S,P,CY and AC flags are affected.

Sometimes it is desired to change bits of a register to different values. Examine the result of the ANI instruction.

```
      10010010    (A register)
ANI   11000011    (<byte>)
      10000010    (result)
```

If there is a 0 in a bit position in <byte> then the corresponding bit in the result will be 0. If there is a 1 in a bit position in <byte> then the corresponding bit in the A register will be copied to the result. Knowing this, the AND operation would be useful in a program where you needed to know whether one of the dip switches was on. The way this is done is illustrated below. The A register holds the values of the dip switches, and <byte> has the bit corresponding to the switch you want to examine set to 1 and all other bits are 0.

```
      00101111    (dip switch value)
ANI   00100000    (examine dip switch 5)
      00100000    (result)
```

All bits in the value of the dip switch have been made 0 except that the bit that we wanted to examine is 1. According to the logic of the AND operation, all bits would have been 0 if the switch was off. This method of making certain bits 0 and passing other bits is called "masking" and the data that is ANDed to the A register is called the "mask".

In the situation where it is needed to change bits to 1, the OR operation should be used. As seen below, a bit value in A is set to 1 if the corresponding bit value in <byte> is 1.

```
      01000011 (A register)
ORI   10101110 (<byte>)
      11101111 (result)
```

To complement or toggle particular bits (change 1s to 0s and 0s to 1s), the XOR operation can be used.

```
      10101011 (A register)
XRI   11110000 (<byte>)
      01011011 (result)
```

If a bit in <byte> is 1 then the corresponding bit in A will be toggled in the result, otherwise if it is 0 the corresponding bit in A will be passed to the result unchanged.

The following program combines the instructions ANI, ORI and XRI to change the data being input through the dip switches by setting, resetting and toggling the bits. This process of changing bits is called "bit manipulation". Bit manipulation instructions are used by the MOS to allow the 8085 to control other chips in the Universal Trainer and to read data from them.

```
leds        equ   40h
dips        equ   41h
            org   8f01h
loop:       in    dips         ; get the dip switch values
            ani   00111111b    ; make bits 6 and 7, zero
            ori   00110000b    ; make bits 5 and 4, one
            xri   00000110b    ; toggle bits 2 and 1
            out   leds         ; display A on Status LEDs
            jmp   loop         ; jump to loop
            end
```

```
      ADDRESS     DATA        INSTRUCTION
      8F01        DB          IN    41
      8F02        41
      8F03        E6          ANI   3F
      8F04        3F
      8F05        F6          ORI   30
      8F06        30
      8F07        EE          XRI   06
      8F08        06
      8F09        D3          OUT   40
      8F0A        40
      8F0B        C3          JMP   8F01
      8F0C        01
      8F0D        8F
```

Load the above program into memory then press reset and run it. You will see that the Digital Out Status LEDs will light up according to the following rules:

LED NUMBER	**STATUS**
7	**off**
6	**off**
5	**on**
4	**on**
3	**on if dip switch #3 is off**
2	**off if dip switch #2 is off**
1	**off if dip switch #1 is off**
0	**on if dip switch #0 is off**

NOTE: If you are facing the Universal Trainer, the dip switches are numbered 0-7 from left to right.

LESSON 13: Using Conditionals

NEW INSTRUCTIONS

JNZ **<addr>** op code = C2

If the zero flag is 0, load the PC register with the two bytes following the op code, otherwise start executing the instruction after this one. The first byte following the op code is the low byte of the address, the second byte is the high byte of the address. No flags are affected.

The program below is the same as the one in Lesson 2 except that it provides a way to stop the program without pressing the reset button.

```
leds        equ   40h
dips        equ   41h
            org   8f01h
loop:       in    dips         ; load the A register with
                               ; dipswitch values
            out   leds        ; output the value to LEDs
            ora   a           ; Set Z flag if A = 0
            jnz   loop        ; jump to loop if A not = 0
            rst   7           ; return to MOS
            end
```

The JNZ instruction is the same as the JMP instruction only it has a conditional (NZ) attached to it. Conditionals, in general, control whether an instruction will be executed or whether it will be ignored. If the instruction is ignored, the instruction following it will be executed. Not only the can the JMP instruction be controlled by conditionals but also the CALL and RET instructions which are discussed later.

Below is a list of the different jump instructions that use conditionals and the values of the flags that are necessary for the jump to occur:

```
JNZ = Jump if result of operation was not 0          (if Z  = 0)
JZ  = Jump if result of operation was  0             (if Z  = 1)
JNC = Jump if no carry from the operation            (if CY = 0)
JC  = Jump if the operation caused a carry           (if CY = 1)
JPO = Jump if the parity of the result is odd        (if P  = 0)
JPE = Jump if the parity of the result is even       (if P  = 1)
JP  = Jump if a positive number result.              (if S  = 0)
JM  = Jump if a negative number result.              (if S  = 1)
```

```
ADDRESS     DATA        INSTRUCTION
8F01        DB          IN   41
8F02        41
8F03        D3          OUT  40
8F04        40
8F05        B7          ORA  A
8F06        C2          JNZ  8F01
8F07        01
8F08        8F
8F09        FF          RST  7
```

Press the reset button and enter the program into memory. Once the program has been entered into memory, set the dip switches so that at least one switch is moved to the opposite position of the others in order to guarantee that the dip switch value will not be zero. Run the program and it should work the same as the one in lesson two, until the dip switch value is changed to zero. Change the dip switch value to 0. When this happens, the program will execute the RST 7 instruction, return to the MOS and display "8F09 FF" which is the address and the op code of the RST 7 instruction.

To analyze the program press the reset button and perform the following steps:

CURRENT PC	
8F01	Change one of the dip switches and single step to address 8F06.
8F06	Examine the flag register and you will see that bit 6 (the zero flag) is 0 indicating that the ORA A instruction did not give a zero as a result. ORing A with A will only give a 0 result if A equals 0 and as you can see, A is not 0. Single step and the JNZ 8F01 instruction will be executed because the conditional is true (since zero flag is 0).
8F01	Restore the dip switch that was changed, to its original position, so that the dip switch value will be 0 and single step to address 8F06 again.
8F06	Examine the flag register and you will see that bit 6 (the zero flag) is 1 indicating that the ORA A instruction gave a zero as a result. As you can see, the A register is 0. Single step and the JNZ 8F01 instruction won't be executed because the conditional is false (since zero flag is 1).
8F09	This is the end of the program.

LESSON 14: Using Register Indirect Addressing

NEW INSTRUCTIONS

LXI H,<word> op code = 21
Load the HL register pair with the word (a word = 2 bytes) that follows the op code. The byte following the op code goes in the L register and the byte after that goes into the H register. No flags are affected.

MVI M,<byte> op code = 36
Copy the byte that follows the op code to the byte in memory pointed to by the HL register pair. No registers are affected.

MOV E,M op code = 7E
Copy the byte in memory pointed to by the HL register pair to the E register. No flags are affected.

INX H op code = 23
Increment the value of the HL register pair. No flags are affected

MOV M,E op code = 73
Copy the E register to the byte in memory pointed to by the HL register pair. No registers are affected.

According to the definitions of MVI **M,**<byte>, MOV **M**,E and MOV E,**M,** the **M** referred to by these instruction works the same way as a register, in that data can be loaded from it or stored to it, as bytes. But it is different than a register because **M** refers to a certain location in memory. The HL register pair *points to* this memory location. Since the HL register pair is 16 bits it can point to any address in memory from 0000 to FFFF just as the PC register can.

For example, if HL had the value of 0131 hex and the MOV E,M instruction was executed, the data at memory address 0131 will be copied to the E register. This is show below:

```
ADDRESS     DATA
:(addresses 0000-012F)
:
0130        3C
0131        01    <-HL
0132        CD
0133        D8
:
:(addresses 0134-FFFF)
```

The value loaded into E is the value of the byte *pointed to* by HL. If HL was 0133 then the value loaded into E would be D8; If HL was 0132 then the value loaded into E would be CD and so on.

The MVI M,<byte> instruction works similarly, but instead of reading the byte pointed to by HL, it writes data to the byte pointed to by HL. If HL was FFC3 and the MVI M,03 instruction was executed then 03 would be written to the data at address FFC3. The value of the data at address FFC3 before and after the MVI M,03 instruction is shown below.

```
         BEFORE                                   AFTER
ADDRESS     DATA                         ADDRESS     DATA
:(addresses 0000-FFC0)                   :(addresses 0000-FFC0)
:                                        :
FFC1        32                           FFC1        32
FFC2        01                           FFC2        01
FFC3        20 <-HL                      FFC3        03 <-HL
FFC4        14                           FFC4        14
:                                        :
:(addresses FFC5-FFFF)                   :(addresses FFC5-FFFF)
```

If the value of HL was changed to FFC1 and the MVI M,03 instruction was executed then the data at address FFC1 will be changed to 03.

This method of accessing memory described in this lesson is called register indirect addressing. The following program illustrates the use of the HL register pair to indirectly access memory.

```
            org   8f01h
            lxi   h,addr      ; load HL with the address of
                              ; the reserved byte.
            mov   e,m         ; load E with data pointed to by HL
            inx   h           ; HL=HL + 1 (point to next address)
            inx   h           ; HL=HL + 1 (point to next address)
            inx   h           ; HL=HL + 1 (point to next address)
            inx   h           ; HL=HL + 1 (point to next address)
            mov   m,e         ; store E at new address in HL
addr:       inx   h           ; HL=HL + 1 (point to next address)
            mvi   m,77h       ; store 77h at new address in HL
            rst   7           ; return to MOS
            ds    2           ; this is a 2 byte reserved location.
            end

      ADDRESS     DATA        INSTRUCTION
      8F01        21          LXI  H,8F0A
      8F02        0A
      8F03        8F
      8F04        5E          MOV  E,M
      8F05        23          INX  H
      8F06        23          INX  H
      8F07        23          INX  H
      8F08        23          INX  H
      8F09        73          MOV  M,E
      8F0A        23          INX  H
      8F0B        36          MVI  M,77h
      8F0C        77
      8F0D        FF          RST  7
      8F0E        00          (NOT AN INSTRUCTION, BUT DATA)
      8F0F        00          (DATA)
```

Enter the program into memory then press reset which clears the general purpose registers and changes the program counter to 8F01. Do the following:

CURRENT PC

8F01 Single step and examine the HL register which should be 8F0A.

8F04 Single step and the data pointed to by HL (the data at memory address 8F0A) will be put in the E register.

8F05 Single step four times, which will execute four INX H instructions, thereby pointing HL to the place to store data. Examine the HL register pair.

8F09 Single step and the value in the E register will be stored at the new address pointed to by HL (at 8F0E.

8F0A Single step and an INX H instruction will be executed.

8F0B Single step and 77 will be stored at the new address pointed to by HL (at 8F0F).

8F0D This is the end of the program. Press "enter" and you will see that the instruction at 8F09 (MOV M,E) did store the value of the E register at 8F0E. Press enter again and you will see that the instruction at 8F0B (MVI M,77) actually stored 77 at 8F0F.

Notice that this program loads the E register with a byte from a memory location within the program itself (8F0A) and that the byte is actually an instruction. Though this is totally legal in machine language it is not useful. The program was written this way so that the value loaded into the E register would be known beforehand. Remember that the value of HL can be anything from 0000 to FFFF so data can be stored or read anywhere in memory.

LESSON 15: Using Register Indirect Addressing to Perform 24 Bit Addition.

NEW INSTRUCTIONS

ADD M op code = 86
Add the byte in memory pointed to by the HL register pair to the A register. The Z,S,P,CY and AC flags are affected.

ADC M op code = 8E
Add the carry flag and the byte in memory pointed to by the HL register pair to the A register. The Z,S,P,CY and AC flags are affected.

INX H op code = 23
Increment the HL register pair. No flags are affected.

Sometimes the need arises to add two 24 bit numbers. This can be done using register indirect addressing, as in the example program below. The program adds the 3 byte number at addresses 8F10, 8F11 and 8F12 to the 3 byte number in the C,D and E registers and stores the result in C,D and E. The byte in E and the byte at 8F10 are the least significant bytes and the byte in C and the byte at 8F12 are the most significant bytes.

```
            org   8f01h
            lxi   h,num
            mov   a,e         ; A=least significant byte
            add   m           ; add data at hl address to A
            mov   e,a         ; save result in E
            inx   h           ; point to next memory address
            mov   a,d         ; A=2nd most significant byte
            adc   m           ; add carry and data at hl address to A
            mov   d,a         ; save result in D
            inx   h           ; point to next memory address
            mov   a,c         ; A=most significant byte
            adc   m           ; add carry and data at hl address to A
            mov   c,a         ; save result in C
            rst   7           ; return to MOS
num:        ds    3           ; 3 bytes of storage
            end
```

```
      ADDRESS     DATA        INSTRUCTION
      8F01        21          LXI  H,8F10
      8F02        10
      8F03        8F
      8F04        7B          MOV  A,E
      8F05        86          ADD  M
      8F06        5F          MOV  E,A
      8F07        23          INX  H
      8F08        7A          MOV  A,D
      8F09        8E          ADC  M
      8F0A        57          MOV  D,A
      8F0B        23          INX  H
      8F0C        79          MOV  A,C
      8F0D        8E          ADC  M
      8F0E        4F          MOV  C,A
      8F0F        FF          RST  7
      8F10        5C          (least significant byte)
      8F11        A3          (2nd most significant byte)
      8F12        0E          (most significant byte)
```

Load C with 01 and DE with F7BF so that C and DE will contain the 3 byte number 01F7BF, and load the program and data into memory. After loading the program, the number stored in memory addresses 8F12, 8F11 and 8F10 will be 0EA35C. Press the reset button and do the following:

CURRENT PC	
8F01	Single step and HL will be loaded with 8F10 which is the address of least significant digit of the three byte number that is stored in memory.
8F04	Single step and the A register will be loaded with the value of the E register. Examine the A register (it should be BF)
8F05	Single step and the ADD M instruction will be executed which will add to the A register the data at the memory address pointed to by HL. The memory address is 8F10 and the data at that address is 5C, so the result of this instruction will be BF + 5C = 11B. Since 11B won't fit in the A register, the carry flag is set and the A register is 1B. Examine the A register and flag register to verify this.
8F06	Single step and the A register will be copied to the E register.
8F07	Single step and HL will be 8F11 which points to the second most significant byte of the three byte number stored in memory.
8F08	Single step and the D register will be copied to the A register. Examine the A register; it should be F7.
8F09	Single step and the ADC M instruction will be executed which will add to the A register, the carry flag from the last ADD M instruction and the data at the memory address pointed to by HL. The memory address is 8F11 and the data at that address is A3, so the result of this instruction will be F7 + A3 + carry flag = 19B. Since 19B won't fit in the A register, the carry flag is set and the A register is 9B. Examine the A register and flag register to verify this.
8F0A	Single step and the A register will be copied to the D register.
8F0B	Single step and HL will be 8F12 which points to the most significant byte of the three byte number stored in memory.
8F0C	Single step and the A register will be loaded with the value of the C register. Examine the A register (it should be 01).
8F0D	Single step and the ADC M instruction will be executed which will add to the A register the carry flag from the last ADC M instruction and the data at the memory address pointed to by HL. The memory address is 8F12 and the data at that address is 0E, so the result of this instruction will be 01 + 0E + carry flag = 10. Since 10 fits in the A register, the carry flag is zero and the A register is 10. Examine the A register and flag register to verify this.
8F0E	Single step and the A register will be stored in the C register.
8F0F	This is the end of the program.

The result stored in the C,D and E registers is 109B1B which is the sum of 01F7BF and 0EA35C. Try running the program with different values in the C,D and E registers with other values stored at memory locations 8F10, 8F11 and 8F12.

LESSON 16: Using Register Indirect Addressing to Move Data

NEW INSTRUCTIONS

INX B op code = 03
Increment the BC register pair. No flags are affected.

INX D op code = 13
Increment the DE register pair. No flags are affected.

DCR L op code = 2D
Decrement the L register. Z,S,P and AC flags are affected.

LDAX B op code = 0A
Copy the byte from the memory location pointed to by the BC register pair to the A register. No flags are affected.

STAX D op code = 12
Copy the A register to the byte in memory pointed to by the DE register pair. No flags are affected.

The program below uses indirect addressing to copy a series of bytes from one memory location to another. The BC register must be loaded with the address of the start of the memory block to move, DE must be loaded with the address (a RAM address) to which you want to copy the memory block and L must be loaded with the number of bytes in the block that you want to copy.

```
            org   8f01h
loop:       ldax  b      ; load A with byte pointed to by BC
            stax  d      ; store A at address in DE
            inx   b      ; increment BC.  This points BC to the
                         ; next address to get a byte from
            inx   d      ; increment DE.  This points DE to the
                         ; next address to store a byte
            dcr   l      ; decrement L
            jnz   loop   ; if L is not 0, then loop again
            rst   7      ; return to MOS
            ds    1      ; reserve a byte to store data
            end
```

ADDRESS	**DATA**	**INSTRUCTION**
8F01	0A	LDAX B
8F02	12	STAX D
8F03	03	INX B
8F04	13	INX D
8F05	2D	DCR L
8F06	C2	JNZ 8F01
8F07	01	
8F08	8F	
8F09	FF	RST 7
8F0A	00	(data, not an instruction)
8F0B	00	(data also)

After entering the program into memory, press reset and load BC with 8F03, load DE with 8F0A, load HL with 0002 and do the following:

CURRENT PC

8F01 Single step and the A register will be loaded with the value of the byte pointed to by the BC register pair (the byte at 8F03 is 03).

8F02 Single step and the A register will be stored at the memory address pointed to by the DE register pair (8F0A).

8F03 Single step and the BC register pair will be incremented so that it points to the next memory address (8F04) from which to get data.

8F04 Single step and the DE register pair will be incremented so that it points to the next memory address (8F0B) to store data.

8F05 Single step and the L register will be decremented. Since the result is not 0, the zero flag is 0.

8F06 Single step and JNZ 8F01 will load PC with 8F01 since the zero flag is 0.

8F01 Single step and the A register will be loaded with the value of the byte pointed to by the BC register pair (the byte at 8F04 is 13).

8F02 Single step and the A register will be stored at the memory address pointed to by the DE register pair (8F0B).

8F03 Single step and the BC register pair will incremented so that it points to the next memory address (8F05) from which to get data.

8F04 Single step and the DE register pair will incremented so that it points to the next memory address (8F0C) to store data.

8F05 Single step and the L register will be decremented and since the L register has been decremented to 0 the zero flag has been set.

8F06 Single step and since the zero flag is 1 the PC register will be loaded with 8F09 which is the address of the instruction following the JNZ 8F01 instruction.

8F09 This is the end of the program. Press "enter" and you will see that the byte at 8F03 has been copied to memory address 8F0A; Press enter again and you will see that the byte at 8F04 has been copied to memory address 8F0B.

This program uses what is called "conditional looping". Conditional looping causes an instruction or group of instructions to be repeated as long as a certain condition becomes true. The instructions that are repeated in this program are LDAX B, STAX D, INX B, INX D and DCR L and the condition that must be true is that the L register must not be zero. You noticed that when the program was run, the L register was 2 and the instructions were executed 2 times. So you see that the value of L determines the number of times that the instructions will be executed. The way this works is the DCR L instruction decrements the L register and sets the zero flag if the L register is now zero and the JNZ 8F01 will jump to the first instruction as long as the L register is not 0. If the L register is 0 before the DCR L instruction is executed the later will cause the L register to be changed to FF hex and the instructions will be repeated 256 times.

Press reset, load BC with 8F01 and load DE with 8F0A as before, but this time load HL with 0009 and run the program. Running the program will make another copy of the program beginning at address 8F0A. Examine the memory from 8F0A to 8F12 and you will see that it matches the memory from 8F01 to 8F09.

You can also experiment with different values for BC, DE and L, but be careful in your choice of values for DE and L because some values will cause the program to destroy itself by writing data in addresses 8F01 to 8F09.

LESSON 17: Using Variables

NEW INSTRUCTIONS

SHLD <addr> op code = 22
Store the L register at memory address <addr> and store the H register at memory address <addr+1>. The first byte after the op code is the low order byte of <addr> and the second byte after the op code is the high order byte of the address. No flags are affected.

LHLD <addr> op code = 2A
Load the L register with the data at memory address <addr> and load the H register with the data at memory address <addr+1>. The first byte after the op code is the low order byte of <addr> and the second byte after the op code is the high order byte of the address. No flags are affected.

STA <addr> op code = 32
Store the A register at memory address <addr>. The first byte after the op code is the low order byte of <addr> and the second byte after the op code is the high order byte of the address. No flags are affected.

LDA <addr> op code = 3A
Copy the value from memory address <addr> to the A register. The first byte after the op code is the low order byte of <addr> and the second byte after the op code is the high order byte of the address. No flags are affected.

RLC op code = 07
Shift every bit left and copy the bit shifted out of bit 7 into bit 0 and the carry flag. Only the carry flag is affected.

Variables are areas of RAM that the programmer intends to be changed during the execution of a program. In contrast, constants are values that a programmer does not desire to change. For example, if you want to load the A register with an 8 bit constant, use the instruction MVI A,<byte>, but if you want to load it with an 8 bit variable, use the instruction LDA <addr>. Similarly, when you want to load the HL register pair with a 16 bit constant, use the LXI H,<addr> instruction, or if you want to load it with a 16 bit variable, use the instruction LHLD <addr>.
To store a value in an 8 bit variable, you can use the instruction STA <addr> and to store a 16 bit variable, you can use SHLD <addr>.

The program below uses a 16 bit variable and 8 bit variable to produce a LED pattern that changes at a varying rate. The program could have easily been written using registers instead of variables. It is written this way to demonstrate the new instructions.

```
            org   8f01h
            lxi   h,8000h      ; load hl with the constant 8000h
            lxi   d,-100h      ; load de with the constant -100h
            mvi   a,1          ; load A with the constant 1

dlay:       sta   leddta       ; store A in the LED data variable
            dad   d            ; add de to HL
            shld  dlaytm       ; store the new delay
dlay1:      dcx   h            ; decrement HL
            mov   a,h          ; move H to A
            ora   l            ; so it can be ORed with L
            jnz   dlay1        ; if H and L are not 0, go to dlay1
            lda   leddta       ; get the LED data
            out   leds         ; output the pattern to the LEDs
            rlc                ; rotate bits left
            lhld  dlaytm       ; load hl with delay variable again
            jmp   dlay         ; do another delay
dlaytm:     ds    2            ; length of delay
leddta:     ds    1            ; bit pattern to output to LEDs
            end
```

```
ADDRESS     DATA        INSTRUCTION
8F01        21          LXI  H,8000
8F02        00
8F03        80
8F04        11          LXI  D,FF00
8F05        00
8F06        FF
8F07        3E          MVI  A,1
8F08        01
8F09        32          STA  8F24
8F0A        24
8F0B        8F
8F0C        19          DAD  D
8F0D        22          SHLD 8F22
8F0E        22
8F0F        8F
8F10        2B          DCX  H
8F11        7C          MOV  A,H
8F12        B5          ORA  L
8F13        C2          JNZ  8F10
8F14        10
8F15        8F
8F16        3A          LDA  8F24
8F17        24
8F18        8F
8F19        D3          OUT  11
8F1A        11
8F1B        07          RLC
8F1C        2A          LHLD 8F22
8F1D        22
8F1E        8F
8F1F        C3          JMP  8F09
8F20        09
8F21        FF
8F22        FF          (dlaytm least significant byte)
8F23        FF          (dlaytm most significant byte)
8F24        FF          (leddta)
```

Enter the program into memory and do the following:

CURRENT PC

8F01 Single step once and HL will be loaded with the constant 8000h.

8F04 Single step again and DE will be loaded with -100h (FF00h) so subtraction can be performed by using DAD D.

8F07 Single step and the A register will be 1.

8F09 Single step and the A register will be copied to the variable space at 8F24.

8F0C Load the PC register with 8F24 and you will see that the data that was there before (FF) has been changed to 01. Restore the original value of the PC register by loading it with 8F0C. Single step and DE will be added to HL which will change HL to 7F00h which is the same result that would occur if 100h was subtracted from it.

8F0D Single step and HL will be stored in the 16 bit variable space. Load the PC register with 8F22 and you will see that the L register has been stored at 8F22. Press enter and you will see that the H register has been stored at 8F23. Load the PC register with 8F10.

8F10 Set a software breakpoint at 8F16 and run the program from the current address (8F10). This is done because the instruction at this address and the 3 that follow will be repeated until the HL register equals 0, which in this case will be 32,512 times.

8F16 Examine the A register then single step and examine it again. You will see that it has been loaded with the value that was stored at 8F24 earlier.

8F19 Single step twice and the A register will be output to the Digital Out Status LEDs and then its value will be rotated left.

8F1C Examine the HL register pair then single step and examine it again. HL has now been loaded with the variable that was stored earlier using the <u>SHLD 8F22</u>.

8F1F Run the program from here to see what the program does.

This program has demonstrated the usefulness of variables to remove the limitations caused by having only a few registers. As you write larger programs you may find that the use of variables is an absolute necessity. The last page of the assembly language listing of MOS shows the system variables used by MOS.

LESSON 18: The Stack and Related Instructions

NEW INSTRUCTIONS

PUSH PSW op code = F5
Put the processor status word (A register and flag register) on the stack. No flags are affected.

PUSH B op code = C5
Put the BC register pair on the stack. No flags are affected.

POP PSW op code = F1
Get the processor status word (A register and flag register) from the stack. Z,S,P,CY and AC flags are affected.

POP B op code = C1
Get the BC register pair from the stack. No flags are affected.

The 8085 microprocessor allows you to choose an area of RAM to be a section of temporary storage called a stack. The key element of the stack is the 8085's internal stack pointer register (SP). The SP can be set to start at any address in memory, but it is best to leave it at or below the address that is selected by the MOS.

Memory is most often shown with low addresses being at the top of a listing, and higher addresses being at the bottom as in the example below:

```
      ADDRESS     DATA        INSTRUCTION
      ( this is not a working program )
      8F01        F3          DI
      8F02        21          LXI  H,8F20
      8F03        20
      8F04        8F
      8F05        22          SHLD 8FE9
      8F06        E9
      8F07        8F
      :
      : ( addresses 8F08 - FFEF )
      :
      FFD0        40
      FFD1        23
      FFD2        61
      FFD3        03
SP->  FFD4        00
```

For example, if SP = FFD4 (as in the above example) and BC = 1234 and a PUSH B instruction is executed, the B register is copied to memory address SP - 1 (FFD3) and the C register is copied to memory address SP - 2 (FFD2), then 2 is subtracted from SP so that the stack now looks like this:

```
STACK ADDRESS     STACK DATA  DESCRIPTION
      ( this is not a working program )
      :
      :
      FFD0        40
      FFD1        23
SP->  FFD2        34          C REGISTER
      FFD3        12          B REGISTER
      FFD4        00
```

If the A register is 56 and the flags are 78 and a PUSH PSW is executed then the stack will look like this:

```
STACK ADDRESS     STACK DATA  DESCRIPTION
      ( this is not a program )
      :
      :
SP->  FFD0        78          FLAGS
      FFD1        56          A REGISTER
      FFD2        34          C REGISTER
      FFD3        12          B REGISTER
      FFD4        11
```

You can now see where the name "stack" comes from, data is "stacked" in memory with every PUSH instruction. Since the stack grows down in memory it is possible for it to eventually overwrite your program, so you must be careful when using the stack.

POP instructions work the opposite of PUSH instructions. For example, if a POP H instruction was executed after the PUSH PSW in the previous example, then the data at memory address SP would be copied into the L register and the data at memory address SP + 1 would be copied into the H register then 2 would be added to SP.

```
      org   0ffcbh      ; start program near top of stack
      push  psw         ; put PSW on stack (A register and flags)
      push  b           ; put BC register pair on stack
      pop   psw         ; remove data from stack and put in PSW
      pop   b           ; remove data from stack and put in BC
      db    0,0,0,0,0,0 ; there are zeros in the stack area so
                        ; changes in stack will be more visible
```

```
      ADDRESS     DATA        DESCRIPTION
PC->  FFCB        F5          PUSH  PSW
      FFCC        C5          PUSH  B
      FFCD        F1          POP   PSW
      FFCE        C1          POP   B
      FFCF        00
      FFD0        00
      FFD1        00
      FFD2        00
      FFD3        00
SP->  FFD4        00          (stack pointer is here)
```

To demonstrate the way PUSH and POP works, load the program counter register with FFCB then load the above data into memory, exactly as it is shown. Press the load the program counter with FFCB again and load the A register with 01 and the flags register 02, load BC with 0304 and do the following:

1) Examine the SP register which should be FFD4.

2) Examine the stack contents and they should be 0000. This is done the same way as examining a register. Press the SC key and the stack contents will be displayed in the word field. Remember that SC is not an 8085 internal register, it is just a way that the MOS provides for viewing the last 16 bits put on the stack.

3) Single step once at address FFCB. This pushes the PSW on the stack (the A register and flags).

4) If you examine SP then you will find that it is now FFD2 which is 2 less than before.

5) The stack contents will now be 0102 which is the value of the PSW that was pushed on the stack. Examine the stack contents to verify this.

The listing below shows what the stack looks like now and the PC and SP registers are shown pointing to their respective memory addresses.

```
       ADDRESS     DATA        DESCRIPTION
       FFCB        F5          PUSH  PSW
PC->   FFCC        C5          PUSH  B
       FFCD        F1          POP   PSW
       FFCE        C1          POP   B
       FFCF        00
       FFD0        00
       FFD1        00
SP->   FFD2        02          value of flags
       FFD3        01          A register value
       FFD4        00
```

7) Execute the PUSH B instruction by single stepping once.

8) If you examine the SP then you will find that it is now FFD0 which is 2 less than before and the stack contents will be 0304 which is the value pushed on the stack by the PUSH B instruction.

9) By examining the memory from FFD0 to FFD4 you will see that the data is as the listing below. Return PC to the value of FFCD.

```
       ADDRESS     DATA        DESCRIPTION
       FFCB        F5          PUSH  PSW
       FFCC        C5          PUSH  B
PC->   FFCD        F1          POP   PSW
       FFCE        C1          POP   B
       FFCF        00
SP->   FFD0        04          C register value
       FFD1        03          B register value
       FFD2        02          value of flags
       FFD3        01          A register value
       FFD4        00
```

10) Single step the POP PSW and observe that SP = FFD2 which is now 2 more than before, then examine the A/F register (A register and flag register) and you will see that it is now the value that was put on the stack by the PUSH B instruction.

11) Single step the POP B and you will see that SP is back to the value that it had at the beginning of the program (FFD4) and that the BC register pair is now the value which was pushed on the stack with the PUSH PSW instruction.

12) Examine addresses FFD0 to FFD4 and you will find that the memory is the same as before. POP instructions merely copy stack memory to registers and change the stack pointer without changing the memory. The memory will be changed, however, if different data is pushed on the stack. The only things that change are the stack pointer register and the register pair that was POPed from the stack.

```
      ADDRESS     DATA        DESCRIPTION
      FFCB        F5          PUSH  PSW
      FFCC        C5          PUSH  B
      FFCD        F1          POP   PSW
      FFCE        C1          POP   B
PC->  FFCF        00
      FFD0        04          C register value
      FFD1        03          B register value
      FFD2        02          value of flags
      FFD3        01          A register value
SP->  FFD4        00
```

The stack has another purpose other than storing general purpose registers. It is also used to store addresses by the RST 7 instruction which was discussed in an earlier lab and the CALL and RET instructions which will be discussed later.

LESSON 19: Using the XTHL Instruction.

NEW INSTRUCTIONS

XTHL op code = E3
Exchange L with the value at memory address SP and exchange H with the value at memory address SP + 1. No flags are affected.

PUSH H op code = C5
Put the HL register pair on the stack. No flags are affected.

POP H op code = C1
Get the HL register pair from the stack. No flags are affected.

MOV A,M op code = 7E
Copy the byte from the memory location specified by the HL register pair to the A register. No flags are affected.

MOV M,A op code = 77
Copy the byte from the A register to the memory location specified by the HL register pair. No flags are affected.

DCR D op code = 15
Decrement the D register. Z,S,P and AC flags are affected.

One of the best uses for the XTHL instruction is in programs where there is a need for more registers. You can load the HL register with a value and push it on the stack (call this value HL2) and then load HL with a another value (call this value HL1). After that, an XTHL instruction will save HL1 on the stack and then load HL with HL2 allowing HL2 to be accessed. If another XTHL instruction is executed the stack will hold HL2 again and HL will equal HL1 which will allow HL1 to be accessed.

The program below demonstrates this action by multiplying DE by HL using repeating additions and returning the result with the BC register pair being the most significant word and HL being the least significant word.

```
            org   8f01h ; starting address of program
            push  h           ; put HL on stack.  This value is
                              ; the number of adds left to go.
            lxi   h,0000      ; HL=0000
            mov   b,h         ; B=0
            mov   c,l         ; C=0 (BC=0)
loop:       dad   d           ; HL= HL + DE
            jnc   nocarry     ; skip INX B if carry flag = 0
            inx   b           ; if result too big for HL
                              ; add 1 to BC
nocarry:    xthl              ; swap sum with number of adds to go
```

```
        dcx   h            ; 1 less add left to go
        mov   a,h          ; A=H
        ora   l            ; if H and A = 0 then zero flag = 1
        xthl               ; swap new # left to go, with sum
        jnz   loop         ; if zero flag = 0, do another DAD D
                           ; At this point, adds left to go is 0.
        pop   psw          ; clear the stack and quit
        rst   7            ; end of program
```

ADDRESS	**DATA**	**INSTRUCTION**
8F01	E5	PUSH H
8F02	21	LXI H,0000
8F03	00	
8F04	00	
8F05	44	MOV B,H
8F06	4D	MOV C,L
8F07	19	DAD D
8F08	D2	JNC 8F0C
8F09	0C	
8F0A	8F	
8F0B	03	INX B
8F0C	E3	XTHL
8F0D	2B	DCX H
8F0E	7C	MOV A,H
8F0F	B5	ORA L
8F10	E3	XTHL
8F11	C2	JNZ 8F07
8F12	07	
8F13	8F	
8F14	F1	POP PSW
8F15	FF	RST 7

Load the program into memory and press the reset button. Load the HL register with 0002 and the DE register with FFFF and do the following:

CURRENT PC

8F01 Single step and the HL register will be PUSHed on the stack. Examine the stack contents and they will be the same as the HL register. This value on the stack determines how many times the addition will be performed.

8F02 Single step and HL will be loaded with 0000 which is the beginning value for the sum.

8F05 Single step twice and the MOV B,H and MOV C,L instructions will be executed which make BC=0000 which is the high order word of the result.

8F07 Single step and the DAD D instruction will cause DE to be added to the sum that is held in HL (examine HL).

8F08 Since the result of the DAD D instruction is small enough to fit in the HL register pair, the carry flag is not set. Therefore the JNC 8F0C instruction will be executed which will jump over the INX B instruction. Single step and PC will be 8F0C, which is the address of the instruction following INX B.

8F0C Examine the stack contents and the HL register then single step. Examine the stack contents and HL register again and you will see that the XTHL instruction has exchanged their values; HL is now the number of additions that are left to do, and the value on the stack holds the sum from the addition.

8F0D Single step and 1 will be subtracted from HL which indicates that there is now 1 addition left to go.

8F0E The instruction at this address and the one at the following address are used to determine whether HL is 0000. These two instructions are needed because the DCX H instruction doesn't affect any flags. Single step and the H register will be copied to the A register.

8F0F	Single step again and the A register will be ORed with the L register without changing the L register. Remember that by definition when you logically OR bits, a zero will result only if the bits being ORed are both 0. So if registers A and L are both 0 then the ORA L instruction will cause the zero flag to be 1, otherwise the zero flag will be 0. In this case the value of L is 1 and A is 0 so the zero flag will be 0.
8F10	Single step and the XTHL instruction will swap the value on the stack with the value in HL. This makes HL the sum again and the value on the stack is the number of additions left to go.
8F11	Single step and the JNZ instruction will be executed which will return the PC to 8F07. Notice that the XTHL instruction was executed after the ORA L instruction that determines whether the HL register pair is 0000. Since XTHL doesn't affect the flag register, the flags that were affected by the ORA L instruction are the same as before XTHL was executed.
8F07	Single step and the DAD D instruction will be executed which will add DE to the sum in HL again.
8F08	This time the result of the addition wouldn't fit in the HL register pair, so the carry flag is set. Single step and the JNC 8F0C instruction won't be executed since the carry flag is 1, instead PC will be changed to the address of the instruction following JNC 8F0C.
8F0B	Single step and 1 will be added to the BC register pair. This will happen in this program every time the carry flag is set by the previous DAD D instruction.
8F0C	Single step the XTHL instruction and HL is now the number of additions that are left to do, and the value on the stack holds the sum from the addition.
8F0D	Single step and 1 will be subtracted from the number of additions left to do in the HL register.
8F0E	Single step and the H register will be copied to the A register.
8F0F	Single step and the L register will be ORed with the A register and since both of the registers are 0 the result in the A register will be 0 and the zero flag will be set.
8F10	Single step and the XTHL instruction will swap the value on the stack with the value in HL. This makes HL the sum again and the value on the stack is the number of additions left to go (0000).
8F11	Single step and the PC register will be loaded with 8F14 which is the address of the instruction following the JNZ 8F07 instruction. This is because the zero flag must be 0 in order for the instruction to load the PC register with 8F07.
8F14	Examine the SP register then single step and examine it again. Before single stepping, SP was FFD2 and after single stepping it is FFD4, which is the value it had when the program started. The POP PSW instruction was used just to restore the stack to its original starting position, the value that was loaded into the A register and flag register was not needed.
8F15	This is the end of the program. Examine the BC and HL register pairs and they should be 0001 and FFFE respectively, which represents the number 0001FFFE which is the product of FFFF and 0002.

Try other values for DE and HL (HL must be greater than 0). Take note that since this program uses repeating additions to perform multiplication the larger the value of HL the longer the program will take to perform the multiplication. For example, if HL = FFFF the program will take approximately 3 seconds to perform the multiplication.

LESSON 20: Subroutines

NEW INSTRUCTIONS

CALL <addr> op code = C3
Put the address of the next instruction on the stack and load the program counter with the two bytes that follow the op code. The byte immediately following the op code is the least significant byte of the address and the one after that is the most significant byte.

RET op code = C9
Remove a 16 bit address from the stack and load it into the program counter. No flags are affected.

DCR H op code = 25
Decrement the H register. Z,S,P and AC flags are affected.

PUSH B op code = C5
Put the BC register pair on the stack. No flags are affected.

The CALL and RET instructions allow the possibility of subroutines. A subroutine is a part of a program which performs a function that is needed in many different parts of the main program. The CALL instruction jumps to a subroutine after putting on the stack the address of the instruction following the CALL. The subroutine's last instruction executed will be a RET which will cause the program to return to the address that is on the stack. Later in this manual there are some example programs which use subroutines that are built in the MOS.

The following program uses a subroutine which outputs the value in the A register to the Digital Out Status LEDs and delays for a period of time proportional to the value in HL.

```
leds        equ   40h         ; ouput port address for LEDs
            org   8f01h
            lxi   h,8000h     ; starting delay length
loop:       mvi   a,00001111b ; bit pattern for 4 left LEDs lit.
            call  delay       ; output the bit pattern and delay
            cma               ; complement A
            call  delay       ; output the bit pattern and delay
            mvi   a,01010101b ; pattern for lighting odd# LEDs
            call  delay       ; output the bit pattern and delay
            dcr   h           ; decrease the delay
            jmp   loop        ; jump to loop

delay:      push  h           ; save HL on stack
            push  psw         ; save A register & flags on stack
            out   leds        ; output pattern in A to LEDs
delay1:     dcx   h           ; decrement delay length
            mov   a,h         ; A=H so H it can be ORed with L
            ora   l           ; H  ORed with L is 0 if HL=0000
            jnz   delay1      ; if HL>0 then jump to delay1
                              ; Registers are POPed in reverse order of PUSHes
            pop   psw         ; restore A register and flags
            pop   h           ; restore HL register
            ret               ; return to instruction after the CALL
            end

      ADDRESS     DATA        INSTRUCTION
      8F01        21          LXI  H,8000
      8F02        00
      8F03        80
      8F04        3E          MVI  A,0F
      8F05        0F
      8F06        CD          CALL 8F16
      8F07        16
      8F08        8F
```

continued on next page...

```
8F09        2F          CMA
8F0A        CD          CALL 8F16
8F0B        16
8F0C        8F
8F0D        3E          MVI  A,55
8F0E        55
8F0F        CD          CALL 8F16
8F10        16
8F11        8F
8F12        25          DCR  H
8F13        C3          JMP  8F04
8F14        04
8F15        8F
8F16        E5          PUSH H
8F17        F5          PUSH PSW
8F18        D3          OUT  40
8F19        40
8F1A        2B          DCX  H
8F1B        7C          MOV  A,H
8F1C        B5          ORA  L
8F1D        C2          JNZ  8F1A
8F1E        1A
8F1F        8F
8F20        F1          POP  PSW
8F21        E1          POP  H
8F22        C9          RET
```

Load the program into memory and run it; The program should display 3 different patterns on the Digital Out Status LEDs, one after the other, and repeat the patterns again. The delay between each pattern should slowly decrease until the patterns are changing so quickly that they are not observable. Then the delay will suddenly become much longer and then slowly decrease again. If the program is working as described, press the reset button and do the following:

CURRENT PC	
8F01	Single step to the first CALL instruction at address 8F06 and examine the stack content (it will be 0000) and the stack pointer will be FFD4.
8F06	Single step and the CALL 8F16 instruction will be executed causing the program to jump to 8F16, the start of the subroutine.
8F16	The CALL instruction has put the address of the instruction following it, on the stack. Examine the stack contents to verify this. Examine and write down the value of the PSW and HL then single step twice which will first PUSH the HL register pair then PUSH the PSW on the stack. Set a software breakpoint at 8F20, then run from the current address.
8F20	This is the first instruction following the delay loop. Compare the PSW and HL with the values that you wrote down and you will see that they have been changed. Execute the POP PSW instruction by single stepping and the original value of the PSW will be restored.
8F21	Single step again and the POP H instruction will be executed which will restore the original value of HL. Notice that the order of POPing data from the stack is the opposite of the order of PUSHing. This is because the last item PUSHed on the stack is the first one that can be POPed off. POPing the PSW and HL off the stack has restored the return address (8F09) to the top of the stack. Examine the stack contents to verify this.
8F22	Execute the RET instruction by single stepping and it will remove the return address from the stack and jump to the address. The stack contents will be 0000 again with the stack pointer at FFD4 after the RET instruction.
8F09	Single step and the A register will be complemented.

8F0A	Now that you understand what happens during the CALL of a subroutine, check the stack content to verify that it is 0000. Set a software breakpoint at the address of the instruction following this CALL (address 8F0D) and run from the current address.
8F0D	The subroutine has output the bit pattern that was in the A register and paused for period of time proportional to the value of HL and returned to this address.

As you can see, a subroutine CALL acts like a user-defined op code. A subroutine can perform a task without changing the registers, as in the program above, or it can change the registers according to the rules that you define when writing the subroutine. For example, you could write a subroutine that would take the value of the B register and multiply it by the C register and return the result in the BC register. Programmers usually have a library of useful subroutines like this which can be included in their programs.

Subroutines can also make CALLs to other subroutines, and those subroutines can call others and so on. The only limit is the amount of memory the stack can use.

LESSON 21: Using Monitor Operating System Subroutines

This lesson illustrates the use of one of the 25 Monitor Operating System (MOS) subroutines. These subroutines (also called services) that are in ROM can be run by loading the C register with the service number and executing a CALL 1000. All registers that are not used as input or output to the service are not affected. Services that are underlined incorporate the COM1 and COM2 RS232 serial ports. In order to make use of these services a PC/Terminal device should be connected to the COM1 or COM2 serial connector. The services are as follows:

SERVICE 0	**DEMO**	Demonstration; this service routine sends a pitch of increasing frequency to the speaker while flashing the Digital Out Status LEDs in conjunction with the pitch.
	INPUT	REGISTER C: 0
	OUTPUT	REGISTER NONE
SERVICE 1	**<u>CONIN</u>**	Console input; this service waits for a character from the communication port selected by register B. Note that the RST 5.5 interrupt is disabled until the service terminates.
	INPUT	REGISTER C: 1
		REGISTER B: Communication port number 1 or 2.
	OUTPUT	REGISTER L: ASCII character returned from keyboard.
SERVICE 2	**<u>CONSTAT</u>**	Console input status; this examines the communication port selected by register B and returns a 0FFH if a character is ready, otherwise a 00H.
	INPUT	REGISTER C: 2
		REGISTER B: Communication port number 1 or 2.
	OUTPUT	REGISTER L: Console status.
SERVICE 3	**<u>CONOUT</u>**	Console output; this service outputs a ASCII character to the communication port selected by register B.
	INPUT	REGISTER C: 3
		REGISTER B: Communication port number 1 or 2.
		REGISTER E: ASCII character.
	OUTPUT	NONE

SERVICE 4	**PSTRING**	Print string; this service sends a string of characters to the communication port selected by register B. The string of ASCII characters starting at the address in the D/E register pair will be sent out until a "$" is encountered. The "$" is not printed and D/E is returned pointing to the character after the "$".
	INPUT	REGISTER C: 4
		REGISTER B: Communication port number 1 or 2.
		REGISTER PAIR DE: Pointer to the start of the string.
	OUTPUT	REGISTER PAIR DE: This will point to the character after the "$".
SERVICE 5	**UPRINT**	Unsigned print; this service prints a 16 bit number in decimal, without use of sign, to the communication port selected by register B.
	INPUT	REGISTER C: 5
		REGISTER B: Communication port number 1 or 2.
		REGISTER PAIR DE: 16 bit unsigned number to print.
	OUTPUT	NONE
SERVICE 6	**SPRINT**	Signed print; this service prints a 16 bit number in decimal with use of sign (2's complement), to the communication port selected by register B.
	INPUT	REGISTER C: 6
		REGISTER B: Communication port number 1 or 2.
		REGISTER PAIR DE: 16 bit signed number to print.
	OUTPUT	NONE
SERVICE 7	**MULT**	Multiply; this service multiplies HL by DE and returns the result in HL and DE.
	INPUT	REGISTER C: 7
		REGISTER PAIR DE: 16 bit multiplier
		REGISTER PAIR HL: 16 bit multiplicand
	OUTPUT	REGISTER PAIR HL: Upper 16 bits of product.
		REGISTER PAIR DE: Lower 16 bits of product
SERVICE 8	**DIV**	Divide; this service divides HL by DE.
	INPUT	REGISTER C: 8
		REGISTER PAIR HL: 16 bit dividend.
		REGISTER PAIR DE: 16 bit divisor.
	OUTPUT	REGISTER PAIR HL: 16 bit quotient.
		REGISTER PAIR DE: 16 bit remainder.
SERVICE 9	**ADCIN**	Analog to Digital input; this service reads a 8 bit value from a selected A/D channel.
	INPUT	REGISTER C: 9
		REGISTER E: Selected channel number 0, 1, 2 or 3.
	OUTPUT	REGISTER L: 8 bit analog conversion.
SERVICE A	**DIPSWIN**	DIP switch input; this service reads the current switch positions of the 8 position DIP switch.
	INPUT	REGISTER C: A
	OUTPUT	REGISTER L: DIP switch value.
SERVICE B	**PTBIN**	Port B input; this service reads the contents of the digital input port B and returns its complement in L. This is similar to service A.
	INPUT	REGISTER C: B
	OUTPUT	REGISTER L: Complement of 8 bit port B value.

SERVICE C	**PTAOUT**	Port A output; this service writes to the digital output port A.
	INPUT	REGISTER C: C
		REGISTER E: 8 bit value to write to port A (this port is equipped with 8 status LED's).
	OUTPUT	NONE
SERVICE D	**<u>HEXPRINT</u>**	Hex print; This service prints to the communication port selected by register B, the value in DE in base 16 (HEX). 4 HEX digits are printed.
	INPUT	REGISTER C: D
		REGISTER B: Communication port number 1 or 2.
		REGISTER PAIR DE: 16 bit unsigned number to print.
	OUTPUT	NONE
SERVICE E	**DACOUT**	Digital to Analog Converter output; This service routine outputs a 8 bit number in the E register to the Digital to Analog converter.
	INPUT	REGISTER C: E
		REGISTER E: 8 bit value to output to DAC.
	OUTPUT	NONE
SERVICE 10	**PITCH**	Pitch output; This service sends the 16 bit count in the DE register pair to the speaker timer (8253 timer 1). The larger the number the lower the pitch. If the DE register pair = 0, then the speaker tone is turned off.
	INPUT	REGISTER C: 10
		REGISTER PAIR DE: 16 bit pitch value.
	OUTPUT	NONE
SERVICE 11	**LEDOUT**	LED Display output; This service routine displays a ASCII character in the E register to the LED alphanumeric display at the character position specified in the D register.
	INPUT	REGISTER C: 11
		REGISTER E: ASCII character to display.
		REGISTER D: Character position (positions are numbered 0 to 7 from left to right).
	OUTPUT	NONE
SERVICE 12	**LEDHEX**	LED Hexadecimal output; This service routine displays a number in the DE register pair in HEX, on the left 4 LED alphanumeric displays.
	INPUT	REGISTER C: 12
		REGISTER PAIR DE: 16 bit number to be displayed in HEX.
	OUTPUT	NONE
SERVICE 13	**LEDDEC**	LED Decimal output; This service displays a number in the DE register pair in decimal, left justified on the LED alphanumeric display (maximum decimal value is 9999).
	INPUT	REGISTER C: 13
		REGISTER PAIR DE: number to be displayed in Decimal.
	OUTPUT	NONE
SERVICE 14	**DELAY**	Delay according to the value of the HL register pair. The larger the value, the longer the delay.
	INPUT	REGISTER C: 14
		REGISTER PAIR HL: The amount of delay.
	OUTPUT	NONE

SERVICE 15 **TUNE** Play the tune which is in the string pointed to by DE at the tempo specified by B. The larger the value of B the longer the duration of each tone. If a zero is encountered in the string, then the subroutine terminates and returns DE pointing to the byte after the 0.

INPUT REGISTER C: 15
REGISTER B: Length of each tone.
REGISTER PAIR DE: Pointer to a string of tones.

OUTPUT REGISTER PAIR DE: This is returned pointing to the byte after the 0 in the string.

SERVICE 16 **PRNOUT** Send the character in E to the parallel printer port.

INPUT REGISTER C: 16
REGISTER E: Character to be sent to the printer.

OUTPUT REGISTER L: This will be 0 if no errors occured. If an error occured then the bits in L will be returned indicating the following:

Bit 3 = Indicates a printer error in general, if it is 1.
Bit 4 = Means printer not selected, if 0.
Bit 5 = Indicates printer out of paper, if 1.
Bit 7 = Printer is busy, if 0.

SERVICE 17 **OUT422** Send the byte in E out the RS422 port.

INPUT REGISTER C: 17
REGISTER E: Character to output

OUTPUT NONE

SERVICE 18 **IN422** Get a byte from the RS422 port.

INPUT REGISTER C: 18

OUTPUT REGISTER PAIR HL: If a character has been received, H will be 1 and L will be the character. HL will be 0 if a character has not been received.

SERVICE 19 **KEYIN** Wait for a byte from the keypad and return it in L. Reading the keys starting at the top row and reading from left to right as you go down, the values returned for the first 16 keys will be 00 to 0F hex. The next 3 rows will return 14 to 1F hex.

INPUT REGISTER C: 19

OUTPUT REGISTER L: Numeric value of key pressed

SERVICE 1A **WRSCL** Write 8 bytes of memory, starting from the address in DE, to the optional real time clock. The clock provides timekeeping information in BCD including hundredths of seconds, seconds, minutes, hours, day, date, month and year information. The date at the end of the month is automatically adjusted for months with less than 31 days, including correction for leap years. The real time clock operates in either 24-hour or 12-hour format with an AM/PM indicator. The data pointed to by DE will be stored in the real time clock as follows:

```
                BIT 7                            BIT 0  RANGE
                ┌────┬────┬────┬────┬────┬────┬────┬────┐
     DE         │       0.1 SEC     │     0.01 SEC      │   00-99
                └───────────────────┴───────────────────┘
                ┌────┬────┬────┬────┬────┬────┬────┬────┐
     DE + 1     │ 0  │    10 SEC    │       SECONDS     │   00-59
                └────┴──────────────┴───────────────────┘
                ┌────┬────┬────┬────┬────┬────┬────┬────┐
     DE + 2     │ 0  │    10 MIN    │       MINUTES     │   00-59
                └────┴──────────────┴───────────────────┘
```

```
                  (AM/PM mode)
                  ┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
                  │  1  │  0  │AM/PM│10 HR│          HOURS        │   01-12
                  └─────┴─────┴─────┴─────┴───────────────────────┘
     DE + 3
                  (24 hour mode)
                  ┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
                  │  0  │  0  │  10 HOUR  │          HOURS        │   00-23
                  └─────┴─────┴───────────┴───────────────────────┘


                  ┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
     DE + 4       │  0  │  0  │STOP │  0  │           DAY         │  01-07
                  └─────┴─────┴─────┴─────┴───────────────────────┘

                  ┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
     DE + 5       │  0  │  0  │  10 DATE  │          DATE         │  01-31
                  └─────┴─────┴───────────┴───────────────────────┘

                  ┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
     DE + 6       │  0  │  0  │  0  │10MTH│          MONTH        │  01-12
                  └─────┴─────┴─────┴─────┴───────────────────────┘

                  ┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
     DE + 7       │        10 YEAR        │           YEAR        │  00-99
                  └───────────────────────┴───────────────────────┘
```

If bit 7 of address DE + 3 is 0 the clock will be in 24 hour mode after WRSCLK is executed. If it is 1 then AM/PM mode is selected and bit 5 of address DE + 3 will select AM or PM (PM is selected if bit 5 is 1). When changing from AM/PM mode to 24 hour mode and vice-versa you must change the hours to match the selected mode. Once the hours are correct, the real time clock will maintain the correct hour for the selected mode.
If bit 5 of address DE + 4 is set to 1 and WRSCL is executed, the real time clock will be stopped. The clock may be restarted by resetting the bit to 0 and executing WRSCL.

INPUT — REGISTER C: 18
REGISTER PAIR DE: Address of the first of 8 bytes to be written to the real time clock.
OUTPUT — NONE

SERVICE 1B **RDSCL** — Read 8 bytes of data from the optional real time clock and store them in the 8 consecutive bytes starting at the address in the DE register pair. The 8 bytes are formatted the same as the data passed to WRSCL.
INPUT — REGISTER C: 19
REGISTER PAIR DE: Starting address to store the 8 bytes read from the real time clock.
OUTPUT — NONE

The following example program outputs the hex number 1234 to the left four numerical displays.

```
mos         equ   1000h
ledhex      equ   12h
            org   8f01h       ; program starts at this address
            mvi   c,ledhex    ; ledhex service routine
            lxi   d,1234h     ; load DE with 1234 hex
            call  mos         ; call MOS for ledhex service
wait:       jmp   wait        ; loop here till reset is pressed
            end               ; so display won't be erased.  Normally
                              ; programs terminate with a rst 7
```

```
ADDRESS     DATA        INSTRUCTION
8F01        0E          ; MVI C,12
8F02        12
8F03        11          ; LXI D,1234
8F04        34
8F05        12
8F06        CD          ; CALL 1000
8F07        00
8F08        10
8F09        C3          ; JMP 08F09
8F0A        09
8F0B        8F
```

Load the program into memory, press the reset button then run it. You will see "1234" on the left four displays and it will continue to display until you press the reset button. Press the reset button and single step to 8F06 which is the address of the CALL 1000 instruction. Single step again and you will see that instead of the program stopping at the instruction at the address of the subroutine being CALLed, as in the previous lesson, the program will stop at the instruction following the CALL 1000 instruction. The reason for this inconsistency is that the MOS checks to see if the CALL that is being single stepped is in ROM or RAM, and if it is in ROM it will run the subroutine full speed until it is finished. This is because the method of single stepping used by the MOS requires the instructions being single stepped to be in RAM. You will also notice that you couldn't see the numbers "1234" when the subroutine was run. The numbers were actually displayed, but as soon as they were displayed the MOS returned to data entry mode and they were overwritten. This occurs any time you single step a service that uses the digital displays. Store a 0 at 8F02 so after the first instruction is executed, the C register will be loaded with 0 instead of 12h. This time when you run the program service 0 will be executed instead of service 12.

For more examples of programs that use MOS services, see the next two lessons which use the ADCIN and PITCH services. The last lesson uses the KEYIN and LEDHEX services.

LESSON 22: Using PITCH (service 10)

NEW INSTRUCTIONS

RRC op code = 0F

Shift the bits in the A register to the right and put the value shifted out of bit 0 in the carry flag and bit 7. Only the carry flag is affected.

This program translates the value of the dip switch to a 14 bit value that is stored in the DE register pair. The value is passed to service 10 which produces a speaker frequency inversely proportional to the value of DE.

```
dips        equ   41h         ; port to read dip switches
mos         equ   1000h       ; address of MOS services
            org   8f01h
loop:       in    dips        ; get the dip switch value
            rrc               ; rotate the A register right
            rrc               ; 2 times since top two bits aren't used
            mov   e,a         ; store in E temporarily
            ani   00111111b   ; mask 2 top bits to 0
            mov   d,a         ; save in D
            mov   a,e         ; get E again
            ani   11000000b   ; mask the lower 6 bits to 0
            mov   e,a         ; save in E
            mvi   c,10h       ; service 10
            call  mos         ; generate a pitch according to DE
            jmp   loop        ; do it again
            end
```

```
ADDRESS     DATA        INSTRUCTION
8F01        DB          IN   41
8F02        41
8F03        0F          RRC
8F04        0F          RRC
8F05        5F          MOV  E,A
8F06        E6          ANI  3FH
8F07        3F
8F08        57          MOV  D,A
8F09        7B          MOV  A,E
8F0A        E6          ANI  0C0H
8F0B        C0
8F0C        5F          MOV  E,A
8F0D        0E          MVI  C,10H
8F0E        10
8F0F        CD          CALL 1000H
8F10        00
8F11        10
8F12        C3          JMP  8F01
8F13        01
8F14        8F
```

1) Load the program into memory and verify that it was loaded correctly.

2) Move the program counter to 8F01 by pressing the reset button and set the dip switch to input 0F (00001111 in binary). If you are facing the Universal Trainer, the dip switches are numbered 0-7 from left to right and if the part of the individual switch that is closest to you is up, the binary value is 1 and if it is down, the binary value is 0.

3) Single step once and examine the A register to make sure it is 0F, if it isn't, go to step 2.

In order for the dip switches to give a good range of frequencies the 8 bit value of the dip switches are made into the most significant 8 bits of a 14 bit value to generate the frequency. The chart below shows how this is done. This chart shows the op codes that are used to convert the dip switch value to a value stored in the DE register pair. To the right of the op codes are the binary values that the A, D and E registers would be after the op code is executed. In the chart, the "X"s tell you that the value has not been defined yet and that they are unknown values.

```
   OP CODE        VALUES OF REGISTERS AFTER EXECUTION OF OP CODE
                              A           D           E
IN    41h                 00001111b   XXXXXXXXb   XXXXXXXXb
RRC                       10000111b   XXXXXXXXb   XXXXXXXXb
RRC                       11000011b   XXXXXXXXb   XXXXXXXXb
MOV   E,A                 11000011b   XXXXXXXXb   11000011b
ANI   00111111b           00000011b   XXXXXXXXb   11000011b
MOV   D,A                 00000011b   00000011b   11000011b
MOV   A,E                 11000011b   00000011b   11000011b
ANI   11000000b           11000000b   00000011b   11000011b
MOV   E,A                 11000000b   00000011b   11000000b
:
:
```

Single step to address 8F0D and you will see that the 8 bit value of the dipswitch is now the upper 8 bits of the 14 bit number in DE. Note that the DE register pair is always 16 bits, but we refer to the number in it as a 14 bit number since we are ignoring the upper two bits of the DE register.

	VALUE
dip switches	00001111
DE register	0000001111000000

After DE is loaded with the value of the dip switches, the C register is loaded with 10h which is the service number and then a CALL 1000h is made which produces the speaker frequency. After the CALL there is a JMP to the start of the program again and it will continue to repeat indefinitely.

Press the reset button and run the program. Move the dip switches and you will hear the speaker frequency change.

LESSON 23: Using ADCIN (service 9)

NEW INSTRUCTIONS

STC op code = 37

Make the carry flag 1. Only the carry flag is affected

JZ **<addr>** op code = CA

If the Z flag is 1 start executing the instructions at the two byte address following the instruction, otherwise start executing the instruction after this one. The first byte following the op code is the least significant byte of the address, the second byte is the most significant byte of the address. No flags are affected.

This program will read the analog to digital convertor and then divide the number by 32 and produce a bar graph on the Digital Out Status LEDs that is proportional to the analog input voltage.

Switch the Analog Input Source dip switches to INTERNAL so the on-board analog voltage source can be used, then load the following program.

```
leds        equ   40h          ; output port for Status LEDs
mos         equ   1000h        ; MOS CALL address
            org   8f01h        ; start address of program
start:      mvi   c,9          ; service 9
            mvi   e,0          ; choose ADC #0
            call  mos          ; get ADC conversion
            mov   a,l          ; put in A register
            rrc                ; rotate A right = divide by 2
            rrc                ; divide by 2
            rrc                ; divide by 2
            rrc                ; divide by 2
            rrc                ; divide by 2
            ani   00000111b    ; mask off upper 5 bits
            jz    zerout       ; if A = 0 then send it out
            mov   c,a          ; c=number of LEDs to light up
            mvi   a,0          ; A=0
loop:       stc                ; set carry
            rar                ; and rotate it into A register
            dcr   c            ; decrement the counter
            jnz   loop         ; if c > 0, loop again
zerout:     out   leds         ; send bar pattern to LEDs
            jmp   start        ; do it all again
            end
```

```
ADDRESS     DATA        INSTRUCTION
8F01        0E          MVI  C,9
8F02        09
8F03        CD          CALL 1000h
8F04        00
8F05        10
8F06        7D          MOV  A,L
8F07        0F          RRC
8F08        0F          RRC
8F09        0F          RRC
8F0A        0F          RRC
8F0B        0F          RRC
8F0C        E6          ANI  07
8F0D        07
8F0E        CA          JZ   8F1A
8F0F        18
8F10        8F
8F11        4F          MOV  C,A
8F12        3E          MVI  A,0
8F13        00
```

continued on next page...

```
      8F14        37          STC
      8F15        1F          RAR
      8F16        0D          DCR  C
      8F17        C2          JNZ  8F14
      8F18        14
      8F19        8F
      8F1A        D3          OUT  40
      8F1B        40
      8F1C        C3          JMP  8F01
      8F1D        01
      8F1E        8F
```

Press reset and run the program. Turn the potentiometer marked MAN 0 and you will see that the bar graph displayed on the Digital Out Status LEDs changes proportionally with the input voltage.

LESSON 24: Using Compare Instructions

NEW INSTRUCTIONS

RLC op code = 07
Shift the bits in the A register to the left and put the value shifted out of bit 7 in the carry flag and bit 0. Only the carry flag is affected.

CMP L op code = BD
Subtract the L register from the A register and set the condition flags accordingly without changing the A register. The carry flag is 1 if A < L and the zero flag is 1 if A = L. The Z,S,P,CY and AC flags are affected.

CPI <byte> op code = FE
Subtract the byte following the op code from the A register and set the condition flags accordingly without changing the A register. The carry flag is 1 if A < byte and the zero flag is 1 if A = byte. The Z,S,P,CY and AC flags are affected.

This program demonstrates the compare instructions CPI and CMP. Both compare instructions subtract a value from the A register and set the flags accordingly, without changing the A register. If the value subtracted from the A register is equal to the A register then the zero flag is made 1, if it is not equal then the zero flag is made 0. This program reads the value of the keypad and changes the pattern displayed on the Digital Out Status LEDs depending on whether the key pressed was "3" or "0". If any other key is pressed the display is not changed.

```
leds        equ   40h         ; port for Status LEDs
keyin       equ   19H         ; service number for keyin
mos         equ   1000h       ; address of MOS services
            org   8f01h       ; start program at 8F01
            mvi   b,00001000b ; bit pattern for 1 LED lit
loop:       mov   a,b         ; A = bit pattern in B
            out   leds        ; display bit pattern
            mvi   c,keyin     ; c= keyin service number
            call  mos         ; return a key value in L
            mvi   a,0         ; value for "0" key
            cmp   l           ; compare A with L
            jnz   check3      ; jump if L<>A,(key isn't "0")
            mov   a,b         ; A is the bit pattern
            rrc               ; rotate the bits in A right
            mov   b,a         ; save pattern in B
check3:     mov   a,l         ; A = key value from keyin service
            cpi   3           ; compare A with "3" key value
            jnz   loop        ; jump if A<>3 (key isn't "3")
            mov   a,b         ; A is the bit pattern in B
            rlc               ; rotate A left
            mov   b,a         ; put new bit pattern in B
            jmp   loop        ; display new bit pattern
            end
```

```
ADDRESS     DATA        INSTRUCTION
8F01        06          MVI  B,08
8F02        08
8F03        78          MOV  A,B
8F04        D3          OUT  40
8F05        40
8F06        0E          MVI  C,19
8F07        19
8F08        CD          CALL 1000
8F09        00
8F0A        10
8F0B        3E          MVI  A,0
8F0C        00
8F0D        BD          CMP  L
8F0E        C2          JNZ  8F14
8F0F        14
8F10        8F
8F11        78          MOV  A,B
8F12        0F          RRC
8F13        47          MOV  B,A
8F14        7D          MOV  A,L
8F15        FE          CPI  3
8F16        03
8F17        C2          JNZ  8F03
8F18        03
8F19        8F
8F1A        78          MOV  A,B
8F1B        07          RLC
8F1C        47          MOV  B,A
8F1D        C3          JMP  8F03
8F1E        03
8F1F        8F
```

To see how the compare instructions work in a program, load the program into memory and verify that the data was entered correctly then press the reset button and do the following:

CURRENT PC	
8F01	Set a software breakpoint at address 8F0D and run the program. The program will pause until a key is pressed (because of the keyin service), so press the "3" key. After that, the program will stop at the breakpoint address which is the address of the CMP L instruction.
8F0D	The keyin service should return the L register with the value 3 and the A register will be 0 because of the MVI A,0 instruction that was executed before the breakpoint was encountered. Execute the CMP L instruction by single stepping then examine the flag register. The A register is less than L and of course it is not equal, so, according to the definition, the carry flag will be 1 and the zero flag will be 0.
8F0E	Single step and the JNZ 8F14 instruction will be executed and you will see that because the zero flag is 0, the program counter will now point to address 8F14. This skips the code that is to be done when the "0" key is pressed and goes to address 8F14.
8F14	The following instructions will check to see if the "3" key was pressed. Single step once to execute MOV A,L and one more time to execute CPI 3. In this comparison L register is loaded into A and 3 is compared to it. Since A is 3 the zero flag will be 1 and the carry flag will be 0. Examine the flag register to verify this.
8F17	This is the address of the JNZ 8F03 instruction. Single step once and the jump will not occur because the zero flag is 1, instead, the program counter will point to the instruction following this one. The instructions following JNZ 8F03 rotate the value in the B register to the left when the "3" key is pressed.
8F1A	Set a software breakpoint at 8F0D again then run from the current address. The keyin service will wait on a key to be pressed, so press the "0" key and the program will stop at 8F0D.

8F0D	This is the address of the <u>CMP L</u> instruction again. The keyin service will return the L register with 0 which is the value of the "0" key and the A register has been loaded with 0 by the <u>MVI A,0</u> instruction. Execute the <u>CMP L</u> instruction by single stepping and this time, since A equals L the Z flag will be 1 and the CY flag will be 0.
8F0E	Single step and <u>JNZ 8F14</u> instruction won't execute this time because the zero flag is 1, instead the program counter will point to the instruction following this one.
8F11	The following instructions are executed when the "0" key is pressed. Single step to 8F14 and the value in the B register will be rotated to the right.
8F14	Single step and the L register will be copied to the A register. Single step again and the <u>CPI 3</u> instruction will be executed and since A is 0, the Z flag will be 0 and the CY flag will be 1.
8F17	The program counter now points to the <u>JNZ 8F03</u> instruction again and since the Z flag is 0 the NZ conditional is true, so single stepping this will cause the program counter to jump to 8F03.
8F03	Run the program full speed from here to see the way the pressing keys affects the Digital Out Status LEDs.

The previous program used compare instructions to check for equality of two values. The compare instructions can also be used to tell whether a value is greater or less than another value. The following program compares the H and L register and causes the Digital Out Status LEDs to show different patterns depending on the relative sizes of H and L. If H is greater than L then the two left LEDs will shine, if L is greater than H then the right two LEDs will shine and if H = L then the middle two LEDs will shine.

```
leds        equ   40h         ; output port for digital output LEDs
            org   8f01h

loop:       mov   a,h         ; put H in A
            cmp   l           ; set flags according to value of A & L
            jc    lgreat      ; if CY=1 then L>A, so jump
            jz    equal       ; if Z = 1 then A=L, so jump
                              ; At this point in the prog. H is not less than L
                              ; and it is not equal to L, so H is greater than L.
            mvi   a,00000011b ; pattern for H>L
            jmp   patout      ; output the pattern
lgreat:     mvi   a,11000000b ; pattern for L>H
            jmp   patout      ; output the pattern
equal:      mvi   a,00011000b ; pattern for H=L
patout:     out   leds        ; display bit pattern
            rst   7           ; end of program
```

ADDRESS	DATA	INSTRUCTION
8F01	7C	MOV A,H
8F02	BD	CMP L
8F03	DA	JC 8F0E
8F04	0E	
8F05	8F	
8F06	CA	JZ 8F13
8F07	13	
8F08	8F	
8F09	3E	MVI A,3
8F0A	03	
8F0B	C3	JMP 8F15
8F0C	15	
8F0D	8F	
8F0E	3E	MVI A,18
8F0F	C0	

continued on next page...

```
        8F10        C3          JMP   8F15
        8F11        15
        8F12        8F
        8F13        3E          MVI   A,C0
        8F14        C0
        8F15        D3          OUT   40
        8F16        40
        8F17        FF          RST   7
```

Load the program into memory and press the reset button then do the following:

CURRENT PC

8F01	Load HL with FFE0 so H will be greater than L then single step and H will be copied to the A register.
8F02	Single step and A will be compared with L and the flags will be set accordingly. According to the definition of the <u>CMP L</u> instruction if A is greater than L then the zero flag and the carry flag will both be 0 (examine the flag register to verify this).
8F03	Single step and the PC will then point to the instruction following the <u>JC 8F0E</u> instruction. Remember that because the carry flag is not 1 the <u>JC 8F0E</u> instruction will not be executed.
8F06	Single step and the PC will point to the instruction following <u>JZ 8F13</u> because the zero flag must be 1 before the jump will occur.
8F09	Single step and the A register will be loaded with the value that will be displayed on the digital output LEDs. Single step twice more and the program will jump to the OUT 40 instruction and it will display the bit pattern in the A register.
8F17	This is the end of the program.

Press the reset button and load the HL register pair with 22C0 so the H register will be less than the L register and do the following:

8F01	Single step twice to copy H to A and compare A with L. Examine the flag register and you will see that the carry flag is 1 and the zero flag is 0 because A is less than L.
8F03	Single step and since the carry flag is 1 the <u>JNC 8F0E</u> instruction will occur.
8F0E	Single step three times and A will be loaded with the bit pattern that indicates that H is less than L, the program will jump to the OUT 40 instruction which will display the bit pattern in the A register.
8F17	This is the end of the program

Press the reset button and Load the HL register pair with 1010 so H will be equal to L.

8F01	Single step twice to copy H to A and compare A with L again. Examine the flag register and you will see that the carry flag is 0 because A isn't less than L and the zero flag is 1 because A equals L.
8F03	Single step the <u>JC 8F0E</u> instruction and it won't be executed because the carry flag is 0.
8F06	Single step and the <u>JZ 8F15</u> instruction will be executed because the zero flag is 1.
8F13	Single step three times and A will be loaded with the bit pattern that indicates that H equals L, the program will jump to the <u>OUT 40</u> instruction which will display the bit pattern in the A register.
8F17	This is the end of the program.

LESSON 25: Using Interrupts

NEW INSTRUCTIONS

EI op code = FB
Enable interrupts after the next instruction is executed. No flags are affected.

DI op code = F3
Disable interrupts after the DI instruction has been executed. No flags are affected.

RIM op code = 20
(See Instruction Set Encyclopedia)

SIM op code = 30
(See Instruction Set Encyclopedia)

The 8085 has 5 pins, namely TRAP, RST 5.5, RST 6.5, RST 7.5 and INTR, dedicated to be sources of interrupts. The program that follows illustrates using the RST 7.5 interrupt. If a 1 appears on the RST 7.5 and the interrupt has not been disabled (disabling interrupts is explained later) then the program that is currently running will be "interrupted" and the address of the instruction that would have been executed next is pushed on the stack. The microprocessor then jumps to address 003C where there are MOS instructions which get an address from a reserved RAM location and then jump to that address. This location which holds the address to which the microprocessor will jump is called a vector. The vector allows the occurrence of an interrupt to run a specific program that can be located anywhere in ram. Programs of this type are called interrupt service routines (ISRs).

The ISR should always return the registers that it uses with the same values that they had when the interrupt occurred. This can be done by PUSHing the registers and then POPing them before returning to the interrupted program. After PUSHing the registers the real purpose of the ISR can be accomplished. The purpose of the ISR in the following program is to increment a value that is in memory. Before exiting the ISR, all registers that were PUSHed, must be POPed, then an EI (enable interrupt) instruction followed by a RET instruction must occur. The EI instruction is needed because whenever an interrupt occurs, the 8085 automatically performs a DI (disable interrupt).

You can cause the 8085 to ignore (disable) all interrupts, except TRAP, through the DI instruction. This is used in the following program to keep interrupts from occurring while a vector is made for the RST 7.5 interrupt and while the RST 7.5 interrupt mask is made 0 through the SIM instruction.

The SIM instruction can be used to disable the RST 7.5, RST 6.5 and RST 5.5 interrupts individually. Before using the SIM instruction to disable interrupts the A register should be loaded with the following 8 bit binary value:

```
                   00001xyz
          x is the mask bit for RST 7.5
          y is the mask bit for RST 6.5
          z is the mask bit for RST 5.5
```

A mask bit of 1 disables the corresponding interrupt. If the mask bit is 0, then the interrupt will be enabled if an EI has already occurred, or as soon as an EI is executed. To enable only the RST 7.5 interrupt the binary value of 00001011 (0B hex) would be loaded into the accumulator before executing SIM.

The program below will set the RST 7.5 interrupt vector to jump to the ISR at 8F14 and then enable the RST 7.5 interrupt. After that it will enter a loop which loads the A register with the data at address 8F1F and sends the data to the Digital Out Status LEDs. The ISR will interrupt this loop each time the RST 7.5 interrupt occurs and will increment the byte at address 8F1F.

Type in the program and run it. When you press the INT 7.5 key you will generate a RST 7.5 interrupt and you will see the Digital Out Status LEDs change. (Only about 1 interrupt per second can be generated by pressing this key because of the key's debounce circuitry). Press the reset button and set a software breakpoint at 8F14, (the start of the ISR) then run the program. The execution will stop at 8F14 and if you look at the stack contents (SC) the value will be the address of the next instruction which the main loop will execute. If the interrupt occurred during the LDA instruction then SC will be 8F0F, if during the OUT instruction then S.C. will be 8F11 or if during the JMP instruction SC will be 8F0C. Single step once, then set the software breakpoint to 8F14 again and then run full speed again. After the program stops, check the value of SC it could be any of the three addresses mentioned above.

```
vec7hlf     equ   0FFE9h      ; vector for 7.5 interrupt
leds        equ   40h         ; Status LED port

            org   8f01h
            di                ; disable interrupts
            lxi   h,ticsub    ; hl = address of ISR
            shld  vec7hlf     ; Store in vector
            mvi   a,11011b    ; reset 7.5 flip-flop and enable 7.5 int
            sim               ; clear 7.5 interrupt mask
            ei                ; enable interrupts
            ; this is the main loop that will be interrupted
            ; by the ISR
loop:       lda   cntout      ; counter to output to LEDs
            out   leds        ; display value of A register
            jmp   loop        ; jump to loop

            ; This ISR increments CNTOUT
ticsub:     push  psw         ; save A and flags
            lda   cntout      ; get counter value
            inr   a           ; increment it
            sta   cntout      ; save it back
            pop   psw         ; restore A and Flags
            ei                ; Re-enable interrupts
            ret               ; continue from point of interruption

cntout      ds    1           ; reserve 1 byte for the counter
            end
```

```
      ADDRESS     DATA        INSTRUCTION
      8F01        F3          DI
      8F02        21          LXI   H,8F14
      8F03        14
      8F04        8F
      8F05        22          SHLD  FFE9
      8F06        E9
      8F07        FF
      8F08        3E          MVI   A,1B
      8F09        1B
      8F0A        30          SIM
      8F0B        FB          EI
      8F0C        3A          LDA   8F1F
      8F0D        1F
      8F0E        8F
      8F0F        D3          OUT   40
      8F10        40
      8F11        C3          JMP   8F0C
      8F12        0C
      8F13        8F
      8F14        F5          PUSH  PSW
      8F15        3A          LDA   8F1F
      8F16        1F
      8F17        8F
      8F18        3C          INR   A
      8F19        32          STA   8F1F
      8F1A        1F
      8F1B        8F
      8F1C        F1          POP   PSW
      8F1D        FB          EI
      8F1E        C9          RET
      8F1F        00          (not an op code, but the counter value)
```

LESSON 26: Writing Your Own Programs

NEW INSTRUCTIONS

LXI D,<word> op code = 11
Load the DE register pair with the word (a word = 2 bytes) that follows the op code. The byte following the op code goes in the E register and the byte after that goes into the D register. No flags are affected.

Now that you have a basic knowledge of most of the 8085 instructions, it is possible for you to write your own programs. The first step is to decide what you want the program to do. If what you want it to do is similar to what one of the programs in the previous lessons does, you can just modify the program in the lesson to suit your needs. You can also combine programs from different lessons to make a program.

Suppose you decide to make a calculator program which adds the hex digits that you type on the keypad and then shows the total on the display when a key other than a hex digit is typed. To start write the program in your own words, in english. This is called "pseudo-code". A pseudo-code version of the program is as follows:

```
1.    make the total 0
2.    read the keypad
3.    if the key pressed wasn't a hex digit, go to step 6
4.    add the key to the total
5.    go to step 2
6.    display the total
7.    go to step 1
```

Now convert the above program to assembly language.

1. The first step requires that the total be made 0. You must decide which register or register pair will hold the total. The DE register pair would work well for this. To make the DE register 0 you can use the instruction LXI D,0.

2. The next step is to read the keypad. This was done in lesson 23 and involved two assembly language instructions: MVI C,KEYIN and CALL MOS.

3. Step three checks to see if the number that was typed isn't a hex digit. An easy way to do this is to see if the number is greater than F hex and this can be done with a compare instruction. Remember that in a compare instruction, if the number being compared to the A register is bigger, the carry flag is set. Since the keyin service returns the number of the key in the L register it is necessary to load the A register with 0F hex which is the value that you want to compare it with. To load the A register with 0F and compare it to the L register you must use the two following assembly language instructions: MVI A,0Fh and CMP L. In order to jump to step 6 when L is greater than 0F hex (in which case the carry flag is set) a JC DSPLAY instruction can be used. The DSPLAY label is used because we don't know the address of the display code yet.

4. There are several different ways to add the key value that is in L to the total that is in DE. One of the most efficient ways is to load the H register with 0 and add DE to the HL register then exchange the DE register with the HL register so the total will be in DE again. This can be done with the following assembly language instructions: MVI H,0 , DAD D, and XCHG.

5. Step 5 can be translated easily to a jump instruction. We will give the jump address the label "RDKEY" since it jumps to the instructions that read the keys. Therefore the assembly language for this instruction will be JMP RDKEY.

6. To display the total that is in the DE register you can use the LEDHEX service (service 12), which displays the hex value in the DE register pair to the word field. So the assembly language instructions for this will be: MVI C,LEDHEX and CALL MOS.

7. The last step can be easily translated to the assembly language instruction JMP START, where START is a label pointing to the first instruction.

Make a listing of the assembly language instructions that were given above. Start the listing with an ORG instruction followed by the EQU instructions necessary to define the values of "mos", "keyin" and "ledhex" and put the labels "start" and "rdkey" in the label field before their appropriate instructions. Also add comments to each line to enhance readability and put an END instruction at the end. It should look like the following:

```
            org   8f01h
mos         equ   1000h        ; start address of MOS subroutines
keyin       equ   19h          ; service # of keyin
ledhex      equ   12h          ; service # of ledhex
start:      lxi   d,0          ; clear the total in DE
rdkey:      mvi   c,keyin      ; C = keyin service number
            call  mos          ; load L with key from keypad
            mvi   a,0fh        ; max value of keys
            cmp   l            ; compare to key from keypad
            jc    dsplay       ; display the value if L>F
            mvi   h,0          ; clear the H register
            dad   d            ; add total to HL
            xchg               ; put HL in DE
            jmp   rdkey        ; get another key
dsplay:     mvi   c,ledhex     ; c = keyin service number
            call  mos          ; display total in DE
            jmp   start        ; go to start again
            end
```

To change the assembly language program to machine language, get some ruled paper and on the top line put the headings:

```
      ADDRESS     DATA        INSTRUCTION
```

On the first line under the address heading, put the hex address of the start of the program (in this case 8F01). The instruction LXI D,0 will go under the heading "INSTRUCTION" and the op code for the instruction (11) will go under the heading
"data". According to the instruction's definition, the data that will be loaded in the DE register pair will follow the op code in the next two memory addresses. Since the data that is to be loaded into DE is 00, put on the next line the address 8F02 and put 00 under the "DATA" heading. Do the same on the next line except put 8F03 under the "ADDRESS" heading. You see that each line represents a memory address just as in the machine language listings in the previous lessons. Your list should look like this so far.

```
      ADDRESS     DATA        INSTRUCTION
      8F01        11          LXI   D,0
      8F02        00
      8F03        00
```

To convert the rest of the program to machine language it will probably be easier to get the op codes from table 5-2 at the end of appendix C than to search for them in the previous lessons. The op code for the instruction MVI C,KEYIN can be found in the first mnemonic column and its op code (0E) can be found in the column to the left of it. You will notice that the mnemonic in the table is MVI C,D8. The D8 is not hex number D8, it means that the 8085 expects 8 bits (a byte) of data to follow the op code. In the case of the instruction we are now translating, the data following the op code is the value assigned to "keyin" by instruction EQU, which is 19 hex.

In the same mnemonic column, below MVI C,D8 is the instruction LXI D,D16 which we already translated to machine language. The D16 means that the 8085 will expect 16 bits (two bytes) to follow the op code of the instruction. Also within the table you will see instructions that have the abbreviation "Adr" in them (like JMP Adr). These instructions also expect two bytes of data to follow the op code, but in this case the data represents an address. The byte following the op code is the least significant byte of the address and the one after that is the most significant byte. If you encounter "Adr" as part of an instruction, most of the time it is necessary to wait until the program is almost completely translated before you can know the values to use for "Adr". Go ahead and put addresses in the address field, but leave the data field blank until later. Below is a listing of the machine language version of the program with the unknown "Adr"s left blank.

```
ADDRESS     DATA        INSTRUCTION
8F01        11          LXI D,0
8F02        00
8F03        00
8F04        0E          MVI   C,19
8F05        19
8F06        CD          CALL  1000
8F07        00
8F08        10
8F09        3E          MVI   A,0F
8F0A        0F
8F0B        BD          CMP   L
8F0C        DA          JC
8F0D
8F0E
8F0F        26          MVI   H,0
8F10        00
8F11        19          DAD   D
8F12        EB          XCHG
8F13        C3          JMP
8F14
8F15
8F16        0E          MVI   C,12
8F17        12
8F18        CD          CALL  1000
8F19        00
8F1A        10
8F1B        C3          JMP
8F1C
8F1D
```

To find the values to fill in for "Adr" in the jump instructions listed above, you must determine the values of the labels used by the instructions. The label "start" points to the first instruction in the program at address 8F01, the label "rdkey" points to the second instruction of the program at address 8F04 and the label "dsplay" points to the instruction at address 8F16. Fill in the addresses for each label and you will have the following completed listing:

```
ADDRESS     DATA        INSTRUCTION
8F01        11          LXI   D,0
8F02        00
8F03        00
8F04        0E          MVI   C,19
8F05        19
8F06        CD          CALL  1000
8F07        00
8F08        10
8F09        3E          MVI   A,0F
8F0A        0F
8F0B        BD          CMP   L
8F0C        DA          JC    8F16
8F0D        16
8F0E        8F
8F0F        26          MVI   H,0
8F10        00
8F11        19          DAD   D
8F12        EB          XCHG
8F13        C3          JMP   8F04
8F14        04
8F15        8F
8F16        0E          MVI   C,12
8F17        12
8F18        CD          CALL  1000
8F19        00
8F1A        10
8F1B        C3          JMP   8F01
8F1C        01
8F1D        8F
```

Load the program into memory and press the reset button and run it. Press one or a combination of hex keys and they will be added together (the keys are not displayed as you type them). The current data on the display will continue to show until you press a non-digit key and then the total will be displayed. When you first run the program "Func." will be displayed until you press a non-digit key.

If the first program you write doesn't work, don't be discouraged because the first version of a program rarely works. You may need to use single stepping to find out why the program isn't working the way you expected. Another useful debugging technique is to press the TRAP button. This will allow you to interrupt a program that is running at full speed and examine the registers or set breakpoints.

You will find as you are starting out, most of your programming problems will be caused by an incomplete knowledge of the way an instruction works. For this reason it is good for you to study the Intel Instruction Set Encyclopedia included in appendix C. For example, a common mistake is to expect the execution of a DCX instruction (decrement register pair) to set the condition flags. The definition of this instruction says that no flags are affected. So if this instruction was used in a conditional loop to control how many times the loop occurred, the loop may go on infinitely.

APPENDIX A
Memory Map and I/O addresses

Memory Map:

EPROM Space:	address 0000 hex to 7FFF hex, jumper JP4 selects usage of 16k or 32k EPROMS.
RAM Space:	address 8000 hex to FFFF hex, jumper JP5 selects usage of 8K or 32K RAMS.

(addresses 6000 to 7FFF hex are sacrificed when Memory Mapped I/O is selected)

I/O Device Map:

Device	I/O address(s)	Memory address(s)
Hardware Breakpoint	**00 to 02 hex**	**6000 to 6002 hex**
Low Address	00	6000
High Address	01	6001
Arming Circuit	02	6002
8259 Interrupt controller	**10 to 11 hex**	**6010 to 6011 hex**
8253 Timer	**20 to 23 hex**	**6020 to 6023 hex**
Timer #0	20	6020
Timer #1	21	6021
Timer #2	22	6022
Timer Control	23	6023
Parallel Printer	**30 to 31 hex**	**6030 to 6031 hex**
Printer Data	30	6030
Printer Control Lines	31	6031
8255 PPI Port	**40 to 43 hex**	**6040 to 6043 hex**
Port A	40	6040
Port B	41	6041
Port C	42	6042
PPI Control	43	6043
8279 LED Display/Keypad	**50 to 51 hex**	**6050 to 6051 hex**
Display/Keypad Data	50	6050
Display/Keypad Control	51	6051
RAMDISK	**60 to 80 hex**	**6060 to 6080 hex**
RAMDISK Low Address	60	6060
RAMDISK High Address	70	6070
RAMDISK Data	80	6080
Analog I/O	**90 to B0 hex**	**6090 to 60B0 hex**
A/D Sample & Hold	90	6090
A/D Select	A0	60A0
D/A select	B0	60B0
8251 Comport 1	**C0 to C1 hex**	**60C0 to 60C1 hex**
COM1 Serial Data	C0	60C0
COM1 Serial Control	C1	60C1
8251 Comport 2	**D0 to D1 hex**	**60D0 to 60D1 hex**
COM2 Serial Data	D0	60D0
COM2 Serial Control	D1	60D1
External Free I/O	**E0 to FF hex**	**60E0 to 60FF hex**

APPENDIX B

Jumper Descriptions

The trainer has an assortment of jumper options, which are used to configure various options or in some cases to cause programmable " failures ". Some jumpers have an inactive position, or " parking spot " to help keep the jumpers from getting lost when not in use at that jumper position. Most jumpers' functions are plainly marked, those that are only labeled " A " or " B ", or " 1 ", " 2 ", etc. are explained here. A list of the jumpers and their descriptions as found on the Revision 1 board is as follows:

JP1 This is a failure configuration jumper. For normal Trainer operation, this jumper should be placed in position " A ". Removal of this jumper, or placing it in its " parking spot ", position " B ", causes the -8 volt inverting switcher to shut down, creating -8 volt supply " failure ". This will only affect the analog and communication sections.

JP2 This jumper configures the Bus Expansion Connector to operate with either EPAC 1000 series expansion peripherals, or 3000 type peripherals. The jumper is clearly marked on the board as to which position is which. This jumpers ties connector CN1 pin 25 to either CPU signal ALE, or to ground. EPAC expansion cards monitor this pin, and behave accordingly. This jumper is always in one mode or the other, so it does not require a " parking spot ".

JP3 This jumper ties the RST6.5 input of the 8085 CPU either to +5v, or to the expansion connector for external use. Note that there is an R-C network to ground, the R-C network filters out noise spikes from adjacent leads when a long ribbon cable is used on the BUS EXPANSION PORT. This jumper also has no particular parking spot.

NOTE: When the trainer is using the Monitor Operating System (MOS) EPROM, the normal configuration of JP3 is placed in the +5v position, to allow software single stepping.

JP4 This jumper selects the type of EPROM chip used by the Trainer. When in the 16 K position, EPROM type 27C128's are useable, and in the 32 K position, EPROM type 27C256 is selected. The trainer monitor version 1 uses 16 K (27C128), future revisions may use the 32 K option.

JP5 This jumper selects between 8 K or 32 K static RAMs, types 6264LP or 62256LP.

JP6 This jumper selects MEMORY MAPPED I/O (in addition to regular I/O), or I/O mapped I/O only.

JP7 This jumper should be placed across pins 1 and 2 for normal trainer operation. Removing it or placing it at parking spot pins 2 and 3, places a logic HIGH on the HOLD input to the 8085 CPU. The CPU will suspend normal operation, and TRI-STATE it's outputs. It will then activate it's HOLD ACKNOWLEDGE pin, and the trainer bus buffers will also go TRI-STATE. This will leave the CPU DATA, ADDRESS, and CONTROL busses floating. The purpose of this jumper is to allow connection of other bus masters to the trainer, such as a DMA controller. JP7 is used as an input to the system in this mode (response to a DMA request), or this jumper can be used as one of the programmable " failures ".

JP8-JP13 These jumpers select the type of Memory chips used in the RAMDISK memory slots. They choose power, address, and strobe functions as per the type of memory required. The RAMDISK sockets U31 and U32 are 28 pin sockets, which will support the use of 8 K or 32 K STATIC RAMS, RAMDISKS, or 32 K EPROMS. All positions are marked " A " or " B ". Following is a table relating type of memory chip to jumper settings for the trainer's RAMDISK.

JP8, JP9, and JP10 are for RAMDISK socket 0, and JP11, JP12, and JP13 are for RAMDISK socket 1.

USAGE	TYPE	JP8/11	JP9/12	JP10/13
8 K SRAM	6264LP	A	A	A
32 K SRAM	62256LP	B	A	B
32 K EPROM	27C256	A	B	B

JP14	This jumper is normally in position " A " in Trainer operation, but can be moved to " B " to " WRITE PROTECT " the RAM ICs in the RAMDISK sockets. Positioning this jumper to " B " inhibits the CPU write strobe from reaching sockets U31 and U32. When in position " A ", WRITE is enabled, and LED D39 is lit indicating this condition.
JP15	These jumpers select whether an external CTS (Clear To Send) is recognized at the COM 1 communication port connector. The 8251 UART will not transmit data (internal hardware lockout) unless CTS is active. As a result, these jumpers double as programmable fault jumpers. As the UART must be transmit enabled, jumper JP15 provides a pullup resistor to +8 volts, (labeled +V) which through the RS-232 interface chip, will assert CTS active low at the 8251, permitting transmission of data. With systems or experiments requiring the use of CTS, the jumper is placed in the CTS position, which connects RS-232 CTS line to the connector. In normal operation, JP15 is left in the +V (TIE-UP) position. JP15 is for Com. channel COM1, at connector CN3.
JP16	This jumper permit selection of the Communication Baud rates desired at COM1. Placement of a jumper at one of the seven (7) positions for each jumper header select baud rates from 300 to 19,200.
JP17	This jumper permit selection of the Communication Baud rates desired at COM2. Placement of a jumper at one of the seven (7) positions for each jumper header select baud rates from 300 to 19,200. COM2 is optional on the Military Version of the Universal Trainer.
JP18	These jumpers select whether an external CTS (Clear To Send) is recognized at the COM 2 communication port connector. The 8251 UART will not transmit data (internal hardware lockout) unless CTS is active. As a result, these jumpers double as programmable fault jumpers. As the UART must be transmit enabled, jumper JP18 provides a pullup resistor to +8 volts, (labeled +V) which through the RS-232 interface chip, will assert CTS active low at the 8251, permitting transmission of data. With systems or experiments requiring the use of CTS, the jumper is placed in the CTS position, which connects RS-232 CTS line to the connector. In normal operation, JP18 is left in the +V (TIE-UP) position. JP18 is for COM2 at connector CN4.
JP19	This is a programmable failure jumper, it is normally in position " A ". Placement in " B " of this jumper causes the feedback loop of the analog sample/hold circuit to open, creating an analog to digital converter failure.
JP20	This is a programmable failure jumper, it is normally placed in position " B ". Placement in " A " shorts the CPU RESET input pin low, and places the CPU in eternal reset.
JP21	This is a fiendish programmable failure jumper. Normally placed in position " C ". It has two failure positions, A and B. Failure " A " shorts latched CPU address line A5 to CPU data bus line D2. Position " B " shorts CPU data line D2 to ground only. The jumper is not located physically near the CPU itself, but to the left of the Memory sockets. Therefore, a lot of signal track tracing will be required to find it. The normal placement of this jumper is at position " C ", which is the jumpers' parking spot, where no shorts are created.
JP22	This programmable failure jumper ties the chip select for the 82C55 PPI chip to +Vcc when in the " B " position, disabling the Digital I/O section. With the jumper placed in the " A " position, the trainer will operate correctly. It is also placed far away from the 82C55 PPI chip, so it will require extensive signal track tracing to locate.
JP23	This programmable failure jumper is normally in position " B ". When it is jumpered to " A ", it shorts the enable pins of LED KEYPAD display driver U54 to ground. This causes the LED display to scramble, as this driver is normally toggled by display control lead SL0 NOT. The bit patterns to the display matrix will therefore be routed incorrectly to the display, and cause the display to be unreadable in most cases.

JP24 This programmable failure jumper is normally placed in position "A". If removed or placed into position "B", it will inhibit CPU signal WR NOT from getting to system RAM. Resistor R89 will pull the WR NOT pin of system RAM chip U19 high (inactive). The system cannot function as a result.

JP25 This option jumper can double as a failure jumper. Normally placed in position " B ", it grounds the Bus Expansion Header's INT0 pin, enabling the Manual INT0 pushbutton. The Expansion INT0 pin and the Pushbutton are logically OR'ed. A pullup resistor on the Expansion connector pulls up the pin input, so a pushbutton input would be inhibited if JP25 was not in the " B " position. When an external INT0 is desired, the jumper should be is position " A ".

JP26 This programmable failure jumper is normally parked at position " A ". Placement at " B " will short Address line A1 to ground, creating a critical system failure.

JP27 On trainers with the D/A output option ordered, this jumper may be used as a failure jumper or as an output connector. Placement of the jumper across position " A " shorts out the D/A system output going to the bargraph display, creating a " failure ". Placement at " B ", is the normal operation parking spot. Also the jumper may be used as an output connector by taking the output at pin " A ", and ground at pin " B ".

JP28 This relatively simple failure jumper, when placed at " A ", shorts out the +2.5 volt reference for the A/D and optional D/A, this creates incorrect analog values. Placement at " B " is the normal condition parking spot.

JP29 This jumper, in conjunction with JP30, provides external access to timer chip U47 (82C53), Timer channel 2. Timer 2 can be used to count or time external events, presented to the trainer for experimentation purposes. Specifically, JP29 selects whether CLK2 input to the 82C55 is fed from timer 0's output, (prescaler), or is externally fed. JP29, pin 1 is grounded, and may be used as the ground side of this external input, with pin 2 as the signal in. Pin 3 is Timer 0's output. The normal configuration for this jumper is placement across pins 2 and 3. If providing an external input signal for Timer 2 be sure its TTL compatible.

JP30 This jumper is similar to JP29, and it options Timer 2's GATE input. In most cases, the GATE input of the timer must be logic high for the counter to work. The gate enable signal here may be set to +5V with jumper JP30 at positions 2-3, or an external gate signal may be fed in at pin 2 of JP30. As in JP29, pin 1 is ground. The normal position for JP30 is placement of a jumper across 2-3. If providing an external gate signal for Timer 2 be sure its TTL compatible.

JP31 This jumper is a programmable failure jumper. When placed in the failure mode, it disconnects Vcc to the 8085 CPU chip. When parked in the correct position, power is restored to the CPU. Failure position is at " B ", and normal placement is at " A ".

JP32 This jumper is a programmable failure jumper. When placed in the failure mode, it shorts the crystal input Y1, X2 of the CPU to ground, killing the oscillator. The normal position of this jumper is at " B ", and the failure position is " A ". Note: The trainer should be powered down and then re-powered after correcting this fault.

Summarization of Default Jumper Settings

Should suspicion arise that the trainer may be malfunctioning, or operational difficulty be encountered with the trainer, it is suggested that the jumpers be placed into their default positions, to allow the user to see if in fact the trainer is working correctly. The following is a listing of the "default" jumper positions for the trainer. Certain jumpers will depend on the memory configurations present marked by " * ", refer to the previous tables for these jumper settings. The Trainer unit should be powered down prior to setting the jumpers, and re-powered when jumpering is complete.

JP1	**Position**	**A**
JP2	**Position**	**1000**
JP3	**Position**	**+5V**
JP4	**Position**	**16K ***
JP5	**Position**	**8K ***
JP6	**Position**	**I/O ONLY**
JP7	**Position**	**1-2**
JP8	**Position**	**Not Critical ***
JP9	**Position**	**Not Critical ***
JP10	**Position**	**Not Critical ***
JP11	**Position**	**Not Critical ***
JP12	**Position**	**Not Critical ***
JP13	**Position**	**Not Critical ***
JP14	**Position**	**Not Critical ***
JP15	**Position**	**+V**
JP16	**Position**	**9600 (or to your terminal's rate)**
JP17	**Position**	**300**
JP18	**Position**	**+V**
JP19	**Position**	**A**
JP20	**Position**	**B**
JP21	**Position**	**C**
JP22	**Position**	**A**
JP23	**Position**	**B**
JP24	**Position**	**A**
JP25	**Position**	**B**
JP26	**Position**	**A**
JP27	**Position**	**B**
JP29	**Position**	**2-3**
JP30	**Position**	**2-3**
JP31	**Position**	**A**
JP32	**Position**	**B**

NOTE: If problems arise in addition to checking the jumpers remove any devices connected to the Bus Expansion port, Digital I/O port, signal conditioning card slot, any wires on Screw terminal ST1 and ST2, Parallel Printer Port CN2, and Communication ports CN3 and CN4. The trainer monitor program EPROM should be in place, at socket U18, and an 32 K STATIC RAM (62256LP) in socket U19. This procedure will help isolate faults created externally to the Universal Trainer board from interfering with this test.

Provide power to the trainer via power jack J1, preferably from the 12 volt D.C. wall mount power supply, (Be sure, of course, that it is plugged into a live outlet!) and switch on the unit. If trainer does not display the proper data to the LED display, something may have gone bad or been blown on the trainer. Contact EMAC for repair details.

APPENDIX C

The Universal Trainer Keypad Description

The MOS uses a twenty-eight key keypad for input. Each key has tactile feel (clicks when pressed) and produces an audible tone when pressed providing feedback for the user. The keys are in two groups of sixteen and twelve. The group of sixteen are Hexadecimal numeric keys (0 - F). The group of twelve keys are command keys. These keys allow the user to examine registers, change their contents, single step, and a variety of other functions. The UT has an additional 8 keys that are not included in the two groups mentioned above. The layout of the twenty-eight keys of the UT are as follows:

0 / EMOS	1	2	3
4	5	6	7
8	9	A	B
C	D	E / HBP	F / SBP

A/F	B/C	D/E	H/L
PC	SP	SC	DEC
STP	RUN	FUNC	ENT

KEYPAD DESCRIPTION

KEY(S)	DESCRIPTION
0-F	Numeric keys. When a numeric key (0-F HEX) is pressed, the numeric value appears at the right side of the data field display. To correct an entry error just repeat, using the correct number key and the error will be overwritten.
A/F	This key displays the contents of the A register (Accumulator) and the condition Flags. The A register and the flags are displayed as four HEX digits. The two digits on the right of the display represent the contents of the A register and the two on the left represent the condition Flags. The displayed value may be changed by entering the desired HEX number using the numeric keys and then pressing the ENT key. The condition Flags are defined bit by bit as follows:

BIT 0	CARRY FLAG	(a set condition indicates a carry)
BIT 1	NOT USED	
BIT 2	PARITY FLAG	(a set condition indicates an even number of bits, odd parity)
BIT 3	NOT USED	
BIT 4	AUX. CARRY FLAG	(a set condition indicates a carry from bit 3 to bit 4)
BIT 5	NOT USED	
BIT 6	ZERO FLAG	(a set condition indicates a zero result)
BIT 7	SIGN FLAG	(a set condition indicates bit 7 of the A register is a 1)

For example, if the display reads: 0044 A/F
This indicates the A register contains 0 and the ZERO and PARITY Flags are both set.

B/C This key displays the contents of the B/C register pair. The B/C register pair is displayed as four HEX digits. The two digits on the right of the display represent the contents of the B register and the two on the left represent the contents of the C register. The displayed value may be changed by entering the desired HEX number using the numeric keys and then pressing the ENT key.

D/E This key displays the contents of the D/E register pair. The D/E register pair is displayed as four HEX digits. The two digits on the right of the display represent the contents of the D register and the two on the left represent the contents of the E register. The displayed value may be changed by entering the desired HEX number using the numeric keys and then pressing the ENT key.

H/L This key displays the contents of the H/L register pair. The H/L register pair is displayed as four HEX digits. The two digits on the right of the display represent the contents of the H register and the two on the left represent the contents of the L register. The displayed value may be changed by entering the desired HEX number using the numeric keys and then pressing the ENT key.

PC This key displays the contents of the PROGRAM COUNTER. The PROGRAM COUNTER is a 16 bit register, displayed as four HEX digits. The displayed value may be changed by entering the desired HEX number using the numeric keys and then pressing the ENT key.

SP This key displays the contents of the STACK POINTER. The STACK POINTER is a 16 bit register, displayed as four HEX digits. The displayed value may be changed by entering the desired HEX number using the numeric keys and then pressing the ENT key.

SC This displays the two bytes that are at the top of the stack, or in other words, the data that would be removed from the stack if a POP instruction were executed. The left two displays show the byte at SP + 1 (stack pointer address + 1) and the two displays to the right of this represent the byte at SP. The displayed value may be changed by entering the desired hex number using the numeric keys and then pressing the ENT key.

DEC This key allows the user to decrement through memory, thus subtracting 1 from the PC every time the DEC key is pressed. The current PC address (4 HEX digits) and the contents of that address (2 HEX digits) are displayed. The address contents may be changed by entering the desired HEX number using the numeric keys and then pressing the ENT key.

STP This key causes the microprocessor to execute one instruction (single step) at the current PC address. Single Stepping is a valuable debugging tool that allows the user to examine registers and memory after each instruction executes. The STP key invokes a software single stepper that executes one user instruction and then returns to Monitor Operating System ready to except another user command. Unlike the Hardware Single Stepper, the Software Single

Stepper requires the processor be active throughout the single stepping process. If single stepping a Service Call, the processor will execute the Service Call at full speed and stop at the instruction immediately following the Service Call.

RUN This key cause the microprocessor to execute a program at full speed starting at the current PC address. The program will continue to execute until an optional hardware or software breakpoint is encountered, a key is pressed, or until a RST 7 (0FFH) instruction is executed.

FUNC This key selects the second function of the keys that have two functions. When this key is pressed "FUNCTION" will appear on the displays and if a key is pressed which has a second function, that function will be performed.

ENT When the UT is turned on or reset the MOS is in data entry mode. When in this mode, pressing ENT will cause the data which is shown on the right two displays to be stored into the address pointed to by the PC register (which is shown on the left four displays) and the PC register will be incremented. When registers or hard or soft breakpoints are being displayed, pressing the ENT key causes the data that is being displayed to be stored in that register or for that breakpoint to be set. If in any mode you have typed some numbers and you want to restore the original value, you can restore them if you haven't press ENT yet. Press the "Func." key twice and you will be back in data entry mode and the data, registers or breakpoints will retain their original values.

(Below are the second functions of the dual function keys)

EMOS This starts the EMOS (Extended Monitor Operating System). This requires that a PC or dumb terminal be connected to COM1.

HBP This key displays the current Hardware Breakpoint address. The displayed value may be changed by entering the desired breakpoint address in HEX using the numeric keys and then pressing the ENT key. The act of pressing the ENT key is what arms the Hardware Breakpoint arming circuit. A hardware breakpoint can only occur when the circuit is armed. Upon the occurrence of a Breakpoint, the Hardware Breakpoint address is still maintained, however the Hardware Breakpoint arming circuit is disarmed. To rearm the hardware breakpoint for the same address simply press the HRD BRK key followed by the ENT key. NOTE: If the program execution never accesses the breakpoint address, the program can not stop at the breakpoint address. The hardware breakpoint, breaks at any address including those in EPROM and operand addresses (see Section on breakpoints for additional information). This function is not provided on Revision 0 boards.

SBP This key displays the current Software Breakpoint address. If 0000 is displayed, then no breakpoint has been established. The displayed value may be changed by entering the desired breakpoint address in HEX using the numeric keys and then pressing the ENT key. Upon the occurrence of a Breakpoint the Software Breakpoint address is automatically reset to 0000. NOTE: If the program execution never reaches the breakpoint address or the breakpoint address is not that of an opcode, the program will not stop at the breakpoint address. Also any address specified that references addresses in EPROM (addresses below 8000 HEX) will not stop execution, even if the opcode at that address is executed (see section on breakpoints for additional information).

APPENDIX D

The Instruction Set Encyclopedia

The information in this appendix may be found in "The MCS `80/85 Family User's Manual" which contains more details about interfacing and programing the 8080 and 8085 microprocessors. This and other Intel literature may be obtained from:

Intel Corporation
Literature Department
3065 Bowers Avenue
Santa Clara, CA 95051