THE

PRIMER

TRAINER

SELF INSTRUCTION

MANUAL

Manual Revision 2.5

EMAC, inc.

EQUIPMENT MONITOR AND CONTROL
CARBONDALE, IL 62901
618-529-4525

PRIMER TRAINER 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, incorrect assembly, 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.

PRIMER parts have been carefully packed, but should there be a shortage of parts, wrong parts, or defective parts, EMAC will supply replacement parts at no charge.

Table of Contents

Appendices

Table of Lessons

PRIMER Trainer Description

The PRIMER Trainer is a compact, low cost, 8085 based microprocessor system, designed primarily for educational purposes. In spite of its low cost and small size, it contains many important and educational features. The PRIMER has digital I/O, analog I/O, and a display plus keypad for human interface. There are also several options that can be added to the basic unit to enhance flexibility and functionality.

STANDARD FEATURES OF ALL PRIMER KITS

- 8085 microprocessor operating at 3.072 MHZ.
- 256 bytes of RAM
- 8 bit digital inputs via 8 station DIP switch.
- 8 digital outputs with LED's for status indication.
- 6 digit numeric LED display.
- 20 key keypad (discrete pushbuttons) for data entry.
- Digital to analog converter, 6 bit resolution.
- Analog to digital converter 6 bit resolution.
- A sound port, which consists of a piezoelectric beeper driven by a programmable counter, can produce variable frequencies.
- Monitor Operating System (MOS) EPROM.
- Operation from a 7 to 10 VDC supply. NOTE: A 9 volt 1/2 amp. wall mount power supply may be ordered separately.

Additionally, the PRIMER may be assembled with the following on-board options, or they may be added later :

- **Extra RAM:** A 32K byte static RAM or RAMDISK may be installed.

- **Extra RAM/Real Time Clock:** A 32K byte battery backed static RAM with Real Time Clock may be installed. The clock keeps time in hundredths of seconds, seconds, minutes, hours, day of week, day of month, month and year, even when the PRIMER is off. It automatically compensates for leap years, and keeps time in military or AM/PM mode.

- **Extra EPROM:** The student may create his own programs, or other operating systems may be added, including high level languages such as E-FORTH, MTBASIC, etc.

- **RS-232 port:** Provides the PRIMER with the ability to hook up to a "dumb terminal" or personal computer.

The following peripheral boards may be connected to the expansion bus connector CN1.

- 32 LINE PARALLEL I/O BOARD: This gives the user 32 lines of digital I/O on two Opto-22 I/O rack compatible connections. Of the 32 lines, 8 are output only and the other 24 are programmable as inputs or outputs. The 24 lines are implemented using the popular 8255 PPI chip.

- EPROM PROGRAMMER BOARD: This allows programming of popular EPROM devices such as the 27c512, 27c256, 27c128 and 27c64 with 21V and 12.5V programming voltage capability. This option is required along with the RAM and serial port upgrade in order to take full advantage of the EPROM programmer interface menu. Documentation on this interface is included when you purchase the EPROM PROGRAMMER BOARD. (To see the EPROM programmer interface menu on your PC monitor, connect your serial port to a PC running a terminal emulation package and press “func.” then “4" on the PRIMER keypad. The only interaction after this point is through the PC keyboard).

Unit Description

The standard PRIMER is a kit containing parts, an assembly manual and instruction manual. The student can assemble the PRIMER unit using the assembly manual without additional assistance. A successfully completed unit will allow the student to enter programs into the units' memory, allow the student to run them, and to view the results.

The Monitor Operating System (the standard monitor software in EPROM) included will cause the keyboard controller chip to scan the keypad buttons, interpret the entries, and allow storage of data into the 256 byte RAM included in the PRIMER. The operating system also displays related data on the six (6) digit LED display. The operating system will allow the student to examine the data stored at various memory locations, examine the contents of the microprocessor's registers and other functions, via the display and keypad. The operating system software will permit the student to run (execute) the program entered, and to execute the program in steps, instruction by instruction, to permit debugging of programs.

The instruction manual contains several lessons in which the student may write programs to access and use the digital I/O, analog I/O, and experiment with the sound port, interrupts, and timers. All that is needed to perform this is included in the basic PRIMER kit.

Optional Hardware and Software

POWER SUPPLY

Any filtered DC power source from seven (7) to ten (10) volts DC may be used to power the PRIMER. Power is fed to the unit via power jack J1. Make sure that the power supply's output plug has a positive tip and a negative sleeve. Current consumption will be less than 500 mA. (350mA to 420mA typically). If desired, a wall plug/DC power supply with built in power plugs may be ordered separately. **Note: Be careful to observe correct type of voltage and polarity, or else the PRIMER may be seriously damaged!**

MEMORY

The PRIMER's built-in RAM memory is 256 bytes. The Monitor Operating System uses a few bytes at the top of this memory. This leaves more than two-hundred bytes for code that may be entered, which is quite a lot considering it is entered by hand. However, if larger memory is desired, as is the case when higher level operating systems such as E-FORTH and MT-BASIC are used, an additional 32 K byte static RAM may be added. Also an Extended Monitor Operating System (EMOS) may be installed in place of the standard Monitor Operating System EPROM.

DATA/FUNCTION KEYS

The standard PRIMER comes with twenty key buttons to select functions, and enter data. A heavy duty keypad may be ordered in place of the buttons.

COMMUNICATION PORT

Without all the options, the PRIMER is a powerful, stand-alone unit that has the potential to do more than just be a training device. It has a easy to use interface that has the potential, with a little more hardware added to become a full blown computer/controller. One of the additions to the PRIMER that help accomplish this end is the Serial RS-232 Communication Port. With this hardware option installed, the PRIMER can communicate with a terminal or PC, permitting greatly increased flexibility. This option is essential when the high level language EPROMS are installed on the PRIMER.

EXPANSION PORT

The expansion port is a 50 pin header connector that provides access to the 8085 microprocessor's multiplexed Address/Data Bus, control signals, and PRIMER DC power. Connection of other circuit cards to this bus allows the PRIMER to perform an almost unlimited number of functions. The connector footprint allows the PRIMER to interface with most of EMAC's standard peripherals, the most notable being EMAC's EPROM Programmer card and digital parallel I/O board. The connector also has extra pins not used by EMAC peripherals, but which are useful to the students who may design peripheral boards of their own.

EXTERNAL I/O PORT

The External digital I/O port connector can be added at any time to provide direct access to the TTL level digital output and input lines, via a ribbon cable. These ports are 8 bits each, and are in parallel with the DIP switch and output LEDs. The I/O port connector permits easy access to the I/O lines so the student may connect switches and relay contacts to the digital input lines and relay drivers, solid state relays (SSRs) to the output lines. These inputs and outputs are used to allow the PRIMER to control various devices for very interesting projects.

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

s 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.

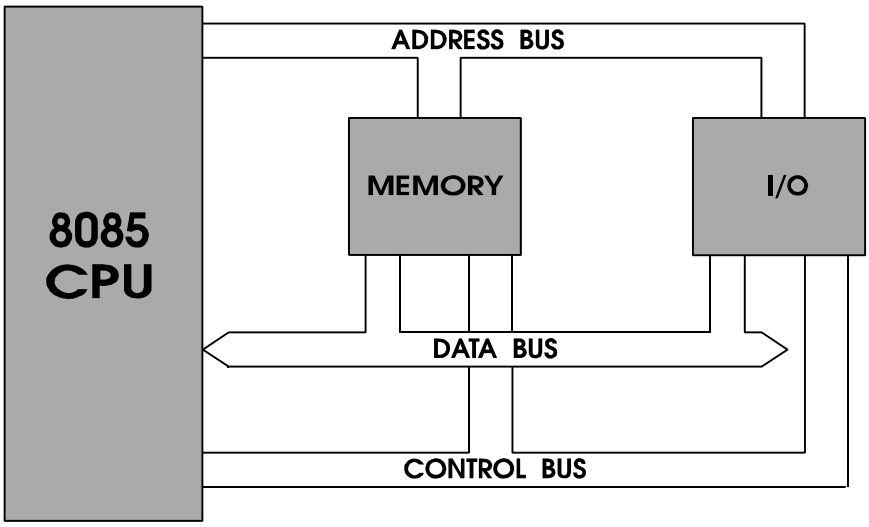
The 8085 performs these operations 8 bits at a time, by performing the logic operation on each bit position, For example:

```
        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 show the way logical operations work with bytes:

```
        00010010          01100111          11101101
AND     01000010    OR    10101110    XOR   01111001    NOT   11001010
        --------          --------          --------          --------
        00000010          11101111          10010100          00110101
```



Primer Block Diagram

MEMORY

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 PRIMER 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.

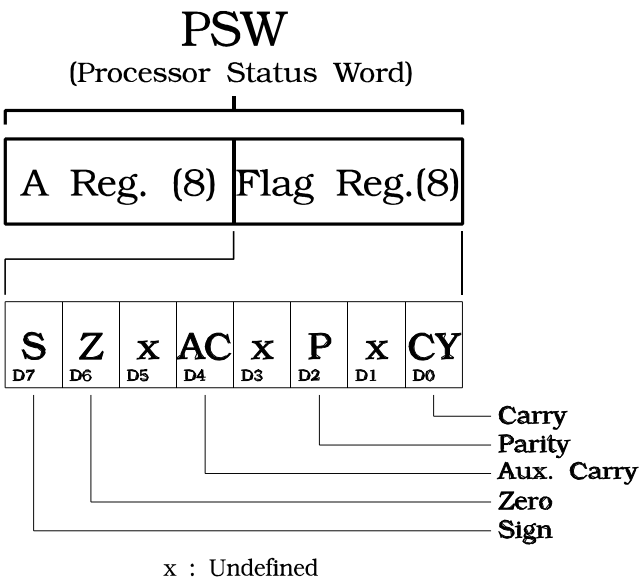
The control bus is a group of connections which provide control over reading or writing of memory or I/O devices. Below is a block diagram showing the way the CPU (microprocessor) connects to the memory and I/O devices through the address bus, data bus and control bus.

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). The A register is the most significant 8 bits and the flag register is the least significant 8 bits of register. The PSW is shown with a diagram of the individual bits of the flag register, below.

Program Counter (16)	
Stack Pointer (16)	
H Reg. (8)	L Reg. (8)
D Reg. (8)	E Reg. (8)
B Reg. (8)	C Reg. (8)
A Reg. (8)	Flag Reg.(8)



x : Undefined

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.

Technical Circuitry Description

Following is a technical description of the actual hardware circuitry of the PRIMER. It describes each of the six schematic pages (in appendix D) in detail and is intended to be understood by a person with a background in digital electronics, though a person without this background may be able to glean some useful information from it. It is not necessary to understand this section in order to use the PRIMER.

Schematic Page 1
Microprocessor and Expansion Bus

The Microprocessor

The 8085 microprocessor chip is the brain of the PRIMER system. It coordinates almost all activity in the PRIMER. The 8085 microprocessor is a collection of counters, gates, registers, decoders, (etc.) that sequentially fetches instructions from memory, finds the purposes of each instruction, and executes the purpose of each instruction. The instructions are placed in memory in the order they are expected to be used. The types of instructions are widely variant, but very concise. They operations consist of moving data, logical operations, branching and conditional branching, some mathematical operations, and input/output. When correctly assembled into a logical order, these primitive instructions form what is called a program. Programs can simply move data from an input port to an output port, using a few instructions, or they can perform complex control operations using many different input and output ports and the number of instructions would stretch out to the thousands.

The 8085 is built to be able to access 65,536 possible memory locations, and 256 I/O locations. The architecture of the microprocessor expects both the program and the data to reside in sequentially arranged memory. The programmer (a person who creates programs) must assemble the program correctly so the microprocessor will know what memory contents are intended to be instructions and which ones are data.

The 8085 and most other microprocessors must start executing the program at its beginning, which in the case of the 8085 is at memory address 0. This is accomplished by a circuit that applies a signal to the pin named RST-IN on the microprocessor when the computer is turned on. On the PRIMER, this signal comes from the Power-on/Pushbutton Reset circuit shown on the schematic. When power is applied to the PRIMER, capacitor C1, initially discharged, holds the microprocessor's RST-IN* pin, low (the "*" in "RST-IN*" means signal is active low). The low signal on the RST-IN pin causes the 8085 to reset some of its internal devices and make an internal register called the program counter point to the beginning of the program. Another event that occurs is that RST-OUT signal is asserted, which resets the display/keypad controller and PPI chip used in the PRIMER. If EMAC peripherals are connected to the expansion port they are also affected by the RST-OUT signal. This reset condition can also be brought about by pressing the reset button (PB1) which merely shorts out C1 and brings the RST-IN pin low.

As the capacitor charges through R1, the RST-IN* line voltage rises until a logic high de-asserts this signal. Diode D1 serves to quickly discharge capacitor C1 through the power supply only when power is off. With RST-IN* de-asserted, the 8085 begins operation.

The system clock oscillator, whose frequency is set to 6.144 MHZ by crystal Y1, drives all timing functions within the 8085 and it runs as long as power is applied to the PRIMER. It is possible to run the 8085 under a wide range of frequencies, but in the PRIMER we use 6.144 MHZ to provide compatibility with circuits used elsewhere in the system. The oscillator signal is divided by two within the 8085, and this signal is now the main system clock, referred to as SYSCLK. All timing of the microprocessor's operation is driven by this SYSCLK, which in our case is 3.072 MHZ. Even asynchronous input signals like RST-IN*, and interrupts are internally synchronized to this oscillator. Each pulse of this clock signal, is called a "T state", when referring to operations internal to the 8085 and its busses.

The first thing the microprocessor will do after RST-IN* de-asserts, is to fetch an instruction at memory address 0. Before the instruction can be fetched, the memory address of the instruction must be output from the microprocessor. To reduce the number of pins on the 8085's package, a scheme called multiplexing is employed. Multiplexing allows the AD0-AD7 pins to be used as the data bus and also to output the lower 8 bits of the 16 bit address needed to select the first instruction. These pins must be demultiplexed before they can be used properly. The 8085 outputs the low byte of the address on signal lines AD0-AD7, and the high byte directly on signal lines A8-A15. A signal called Address Latch Enable (ALE) strobes high, then low, causing the address latch (chip U8) to trap the address byte that is on pins AD0-AD7. The output of this latch then joins up with address lines A8-A15, to provide a 16 bit, filtered, stable, linear address to the memory chips and decoders (described later).

The 16 bit linear address is applied to the memory block (memories and decoders) to select the address that the 8085 wants to read from or write to. In the case of an instruction fetch, the 8085 signal RD* will be asserted, to read the byte from the memory selected. This byte will then go into the instruction decoder, which will determine what instruction it is, and then the 8085 will execute the instruction. Depending on what type of instruction it is, the 8085 will either fetch additional data or read/write data to other memory locations or perform I/O, (etc.). Each time a bus transaction occurs these operations will occur: the low byte of the address will be sent to AD0-AD7, which will then be held by the address latch. A complete cycle of instruction fetch, decode, and execution, forms a "bus cycle", using anywhere from 3 to 14 T-States, (SYSCLK pulses) depending on the type of instruction.

The 8085 has five pins dedicated to "interrupting" the microprocessor. These pins are named TRAP, RST 7.5, RST 6.5, RST 5.5 and INTR. When

a signal is applied to one of these pins an "interrupt" is generated. An interrupt is a special function of the 8085 to suspend the program it is running then execute a program related to the interrupt that requested it. Once the interrupt is completed, the 8085 must RESTORE the original program's data and status, and RETURN to it where it left off from. Sometimes the interrupt is deliberately ignored by MASKING the interrupt, so interrupts can be turned on or off as needed, should a section of code be so important to require that it not be interrupted until it is finished.

An interrupt may be likened to answering the telephone while reading a book. When the telephone rings, you put the book down on it's face to mark your place, then you answer the phone. When you are done, you pick up the book and begin reading where you left off. Should you be too engrossed in the story, you can ignore the ringing telephone (you can mask it). But if you ignore it too long, you'll miss the call. Sometimes interrupts can be missed as well. Some interrupts can be made to latch, usually by hardware. These latched interrupt requests will persist, just as a person on the phone might continue ringing (interrupting) until you answer it.

Bus Expansion Port

The PRIMER's Bus Expansion Port is also shown on schematic page 1. It is actually a simple connector with leads running to power sources and microprocessor pins. The 8085's AD0-AD7 lines run out to the connector, as well as address lines A0-A15, control signal IO/M* (which the 8085 asserts to tell whether it is doing a memory or I/O transaction), RD*, WR*, RST-OUT, SYSCLK, GROUND, and PWR (this is directly from the power jack, 5V is not provided on the bus connector). Also inclued are INTRQ, and INTA*, which allow the use of a multi-level interrupt controller; S0, and S1, which are part of the 8085 machine status output; RDY/WT* which is used to put the 8085 into a wait state; HOLDRQ and HLDACK which are used to allow another bus master to assume control of the bus; and finally, ID* and EXTIOCS*, which are used by external I/O devices of the EPAC 2000 family. The Bus Expansion Port footprint permits direct connection of the PRIMER to most EPAC peripheral cards, via a 50 pin ribbon cable.

The schematic shows " Module Ports " , which are connections to other parts of the board not shown on this page of the schematic. Often one page of a schematic is not big enough to show an entire circuit but, of course, the actual wiring on the printed circuit board is one complete unit. Module ports denote that these connections actually do exist, only they are shown elsewhere. The name or label of the module port must match exactly on another sheet of the schematic to denote that the connections go there. The type of module port (shape of the module) sometimes gives helpful information about the kind of connections made. For example, module port D[0..7] is a bi-directional port, and all occurrences of this port on other pages denote this is a bi-directional 8 line bus interconnection. Individual lines to module ports also connect these lines to corresponding points on other sheets of the schematic, such as lines RD*, and WR*.

Schematic Page 2
I/O Decoding and Memory

The decoder section of a microprocessor system sorts out the address bus signals from the 8085, and picks out (selects) the correct chip(s) in the memory or I/O sections that the 8085 wants to access. The decoder must evaluate the address applied to it, and determine whether the address is for a memory access, or an I/O map access. Most decoders are custom hardware collections of gates, using various levels of integration. As every system has a different configuration of memory and I/O, every decoder is unique to that design.

In the PRIMER, two types of memory maps may be decoded. In the standard map, a 16K (K=1024 bytes) byte space is reserved for program EPROM, starting at address 0000 hex ending at 3FFF hex. (remember the 8085 begins execution at 0000 hex after reset) This socket may hold 8K, 16K or 32K EPROM chips. The monitor operating system program is actually very small, requiring less than 4K, but 8K EPROM chips are the smallest memory chip of compatible footprint that will fit there.

Almost all programs (the PRIMER's monitor is no exception) will require some memory that can be *written to* as well as read. The PRIMER provides this type of memory, without additional discrete RAM chips. The 8155 PPI/MEMORY/TIMER I.C. is a multi-function chip, with digital I/O ports, a timer/counter, and 256 bytes of RAM in it. This memory is used for storage of variables, pointers, executable code and various other functions. Because the 8155 has both I/O and memory mapped elements, the decoder section provides an additional specially encoded line called 8155CS for this device. The 8155 will be described in more detail later.

The optional 32 K byte memory socket allows a 32 K static RAM chip to be installed onto the PRIMER. Option selection jumpers, OJ2 and OJ3 resolve the hardware differences between the 8 K, 16 K and 32 K EPROM chips, and allow the addition of the 32 K byte memory and by rearranging the memory map to accommodate the different chips. This is necessary when software options such as E-FORTH and MT-BASIC are installed. The memory can be altered to use the two memory maps described. In the case of the EPROM chip, an 8 K device (2764) or a 16 K device (27128) may be used interchangeably, bearing in mind the smaller sized devices do not fill the entire memory map. When the alternate memory map is used, a 32 K device, (27256) may be installed in the EPROM socket. The difference of the pinout in the 32 K device is the reason for the option jumpers. Jumper OJ2 switches pin 27 between A14 or Vcc, as required by the 32 K device.

The 8155 PPI's memory space begins at C000 hex and ends at the top of the 8085's address range at FFFF hex. Note that this range is 16 K, but the actual memory is only 256 bytes. What happens is the same 256 bytes repeat themselves over and over again, in this case 64 times over. In other words, hexadecimal addresses C000, C100, F300 and FD00 all refer to the first memory address within the 8155 and C0FF, C1FF, F3FF and

FDFF all refer to the last memory address within the 8155. Programs that are written must use only the amount of memory that is available.

The alternate memory map reserves 32 K of EPROM at addresses 0000 hex to 7FFF hex, and 32 K static Ram from 8000 hex to FFFF hex. The 256 bytes of RAM in the 8155 PPI are ignored, so in this map, the entire 64 K of memory that the 8085 is capable of accessing does not contain memory cells with duplicate addresses.

The PRIMER uses a very simple decoder. Two, "2 to 4 line" decoders decode the appropriate 8085 address lines to select the proper chips. The memory decoder observes the code on address lines A14 and A15. The state of these lines represent 16 K intervals of address. To change the decoder interval to 32 K bytes boundaries, the decoder pin that is connected to A14 can instead be connected to ground. The I/O decoder observes the state of address lines A6 and A7. The I/O map is only 256 addresses wide. The I/O decoder is maximally used at select rates of 64 addresses per output.

The 8085 signal line IO/M* tells the decoders that the address is an I/O MAP transaction, if it is high, or that it is a memory map transaction, if it is low. A simple invertor changes the polarity of IO/M* to IO*/M. Since each decoder has an active low enable input, connection of the appropriate polarity signal of IO/M* to the appropriate decoder, will enable the right decoder for the job at hand. Also all decoder outputs are active low signals, and will be de-asserted unless the right address in the right map selects it.

The EPROM chip select will be valid for memory addresses 0000 hex to 3FFF hex, (or to 7FFF hex if non-standard EPROM) and simply goes directly to the EPROM's chip enable. The optional static RAM chip select must be valid for addresses 4000 hex to BFFF hex, (or 8000 to FFFF hex if non-standard) which is an interval of 32 K. As the memory decoder outputs at intervals of 16 K, (standard map) two chip selects are logically negative OR'ed by AND gate U10A. This produces a single 32 K wide chip select for this memory.

The 8155 chip has elements of both the memory map and the I/O map, so a similar AND gate logically negative OR's the memory and I/O map selects into a single dual mapped chip select for the 8155. The memory portion of this chip select is valid throughout C000 hex to FFFF hex. The I/O portion is I/O addresses 00 hex to 3F hex.

The 8279 keypad and display controller, which will be described in more detail later, is I/O mapped from 40 hex to 7F hex, and the optional 8251A communication controller from I/O addresses 80 hex to BF hex. A special I/O chip select, used primarily for EPAC expansion peripherals, via the bus expansion connector, uses the remaining I/O space of C0 hex to FF hex. As in the memory chip select description, many of the addresses repeat in the I/O map, since these chips only use a few of the available addresses.

The rest of the address bus lines, not used by the chip select decoder described, go directly to the memory chip(s) other I/O chips, and bus expansion connector. Inside these chips are decoders which select the precise internal memory cells or device registers denoted by the address pins. In more complex systems, the chip select decoders can be more precise, but this increases the complexity and cost of the system. The simple device select decoder used here in the PRIMER will serve quite well, as long as programs are written carefully.

Schematic Page 3
Display and Keypad Circuitry

The PRIMER has a key pad matrix of pushbuttons, for entering commands, loading data or programs, and testing programs. It also has a 6 digit LED matrix display, to view data. The 8279 is a specialized controller chip, which removes the chores of keyboard scan, switch de-bounce, and display refresh, from the microprocessor. The 8279 contains its own memory which stores key press data and data to be displayed. Through the I/O map, the 8085 sends commands to the 8279 that configure the operational characteristics of the display/keypad system, then writes the display characters themselves to the 8279's internal display RAM. The 8279 scans this display RAM, and drives the outputs to the display LED's continuously thereafter. No further microprocessor action need be taken until a new display pattern is needed.

The 8279 has 8 data bus lines that connect directly to the 8085's AD0 to AD7 lines. These bi-directional bus leads transfer data and control information between the 8085 and the 8279 Display/Keypad controller. Signals RD*, and WR* determine whether data is read from or written to the 8279. Address line A0 is used by the 8279 to determine whether the data byte is a command byte, or a data byte for the internal RAM and the 8279's chip select pin determines whether it will perform bus transactions or not. Because address line A0 is used to select the 8279 function, two I/O map addresses exist for the Display/Keypad controller. A program that operates the controller merely writes to or reads from two consecutive addresses to control all 8279 functions. With A0 being low, the 8279 memory can receive data from, or send data to the data bus. If A0 is high, the bus transaction is a command or status byte.

The base address of the 8279 in the PRIMER is 40 hex. The I/O decoder's chip select will be valid for all I/O map transactions up to 7F hex. Therefore, the two addresses for the 8279 repeat themselves throughout this range, just as the memory addresses repeat in the 8155. This repetition is common to all I/O devices on the PRIMER, due to the broad nature of the chip select decoder used.

The signals RST-OUT, SYSCLK, and KEYINT also terminate at the 8279. RST-IN will hardware clear the 8279's internal control registers to their

default state, which at a later time can be changed to other states by the PRIMER's program in the EPROM. However, the data in the internal RAM is not cleared by hardware. The SYSCLK signal (the microprocessor's 3.072 MHz master clock signal), is divided internally by the 8279 to provide the keypad scan and LED display refresh clocks. Finally, the KEYINT signal is a special control line that is set active high by the 8279 when a valid key entry is made. This line goes through a jumper that (in its default connection) connects to the RST 5.5 interrupt pin on the 8085 microprocessor.

The 8279 scans the keypad buttons and debounces the key inputs to avoid erroneous entries due to contact bounce or key rollover. After this, it stores the key inputs in the internal keypad RAM. When a valid key entry is made, the 8279 generates an interrupt request to the 8085 via the KEYINT line, and holds the key data in the keypad RAM until the 8085 actually reads the key memory. Should the microprocessor not respond immediately, the 8279 will store in the internal key RAM up to 8 key entries.

When the PRIMER is turned on, the 8279 is configured by the PRIMER software to produce what is referred to as an "encoded scan" on its SL0-SL3 lines. This encoded scan is decoded by a "3 to 8" decoder, to produce 1 of 8 active low row scan outputs. Six of these outputs go to upside down biased PNP transistor followers, to provide high current sinks for the common cathode LED's, since the 74HC138 (3 to 8 decoder) can only sink a few milliamps. The scan outputs from the scan decoder also go to the keypad buttons.

When a key is pressed, a low signal will be presented to one of these corresponding return lines: RL0-RL3. These return lines are already connected to pullup resistors that are conveniently built into the 8279. The key must be closed and be scanned several times in order to be recognized by the 8279. It is then entered into key RAM and an interrupt request is generated which tells the microprocessor to read the 8279's keypad RAM and interpret the key code to perform the function that is dedicated to the key. The key must be released and stay that way for several more scans before that key or another key will be accepted. This is the key pad debounce feature. The 8279 will also ignore additional key closures while if another key is pressed.

Schematic Page 4
Digital I/O, Timer and Speaker

The 8155 combination Digital I/O, Timer, and Memory Chip provides the PRIMER with capability to experiment with Digital input and output, programmable timing, and provides the read/write memory function for the system. Its bus interface is very similar to the interface for the 8279 Display/Keypad controller. The 8155 also has 8 data bus lines, RD*, and WR* control signals, and RESET input. There are additional bus control lines, but no individual address inputs are required since the lower byte of the address and the data are demultiplexed within the 8155 from the AD0-AD7 inputs. The 8155 is missing the address line(s) the 8279 has, instead, it uses two new signals, IO/M*, and ALE. These signal lines are the same as described in the previous microprocessor description. The 8155 chip is a special function chip made specifically to work along with the 8085. It has an internal address latch, and an internal address decoder, which operates in parallel with the PRIMER's address latch. The internal address latch and decoder uses the 8085 ALE signal, to trap the low order 8 bits of multiplexed address. Since the internal RAM is only 256 bytes and there are only 4 I/O addresses, only the low 8 multiplexed address/data bits are needed anyway. The chip select signal for the 8155 resolves whether bus transactions will occur to the address internally latched and selected during microprocessor read or write cycles, to either the I/O or memory sections of the 8155.

The Digital I/O ports are two "byte wide" (8 bits) ports, and one six bit port. The ports can be independently set to be OUTPUT or INPUT ports. On the PRIMER unit, Port A is set by monitor software to function as an output port and eight LEDs serve as status indicators of the bit state for each output line. Each LED lights when the bit for it is set LOW (0), and it is off when the bit is set HIGH (1). Therefore, negative logic levels are assumed.

Port B of the 8155 is set by software to input. An eight station DIP switch is used to set each bit LOW (0) when each station is turned on. Negative logic is also assumed here. When a switch station is switched OFF, a resistor pullup brings the line HIGH. Port C of the 8155 goes to the digital to analog converter which is described later. The 8155 chip is totally flexible as to what kind and data and direction is present on each port, however, for hardware purposes on the PRIMER, the default configuration should be maintained.

The 8155 has a programmable timer/counter in it that is software programmable to count from 1 to 16,383. The counter becomes a timer if it is fed a regular pulse train, as it is in the PRIMER. The SYSCLK signal is first divided by ten, and fed to the 8155 timer input. The timer's output goes to the 8085's RST 7.5 interrupt input, and to an AND gate that drives a piezo-electric speaker element. The piezo-electric speaker has a limited frequency range, but many useful sounds can still be produced by appropriate setting of the timer's divide ratio, and turning the output on and off through the 8085's SOD signal. The SYSCLK frequency of 3.072 MHZ is divided down by U12 to 307.2 KHZ, then fed to the timer input of the 8155. By setting the timer's divide ratio from 2 to 16383 (in software), the range of frequencies can be from 153.6 KHZ down to 18.75 Hz.

Also shown on this schematic, are the spare gates that are not used by the PRIMER circuit, but present on the board. This often occurs since gate packages often come with multiple gates on one chip. As is customary, unused gate inputs should be defined to an idle logic state to prevent random oscillations, latchup, etc. This is done primarily for CMOS and NMOS devices, because TTL gates usually have built-in pullups on their inputs.

Schematic Page 5
Optional Serial Communication Port

If MT-BASIC, E-FORTH or EMOS are installed on the PRIMER, this option will be required. The 8251A chip is called a Universal Asynchronous Receiver Transmitter or UART. The UART interfaces serial data transmissions or receptions to parallel data. The PRIMER's UART uses a very common communication protocol known as the RS-232 standard which uses special voltage levels to connect to computer terminals, desktop PCs, etc.

The 8251A connects to the 8085 bus, and through software the 8085 can communicate to the terminal or PC via the 8251A. The UART operates on standard TTL-MOS voltage levels, and because the RS-232 standard uses Bi-polar voltage levels, a special interface chip, the MAX-232 (or Intersil ICL-232) converts the TTL levels to RS-232 voltages. The MAX-232 chip contains special power supply circuits, called a "charge pump" that boosts the 5 volts used in the PRIMER to +8 volts and -8 volts for the RS-232 link. Voltage translating drivers are also built into the chip. The optional serial communications port interfaces to a terminal or PC through a DB-9 shell connector.

The serial communication rate (or baud rate) must be set to the rate used by the terminal or PC. Placement of jumper JP1 can set the baud rate from 300 baud up to 19,200 baud. The 614.4 KHZ COMCLK, which is generated by a circuit on schematic page 4, is divided down by U16 providing seven different frequencies.

The PRIMER allows interrupt driven serial communications in the MT-BASIC operating system. To enable the hardware to do this, jumper OJ1 can be moved to connect the 8085's RST 6.5 interrupt input to the RXRDY (receive ready) pin of the 8251A UART, instead of its default connection to Vcc.

Schematic Page 6
Analog I/O and Power Supply

The last schematic page of the PRIMER shows the Analog I/O system and power supply for the unit. Many Microprocessor systems read analog signals, such as temperatures, pressures, and amperage. Because microprocessors are digital devices, some sort of interface is required to change an analog voltage into a digital code the microprocessor can deal with. In many cases, the microprocessor system will also output a digital signal, after processing the analog and digital inputs it has reviewed.

The PRIMER has a 1 channel analog output, and a 1 channel analog input with the range of both at approximately 0 to 5 volts. The digital to analog (D/A) output comes from a R-2R ladder network of resistors, which is then bufferred by op-amp U14A. The D/A ladder is driven by port C of the 8155. The D/A converter has 6 bit resolution, so programs may be written which write a value in the range of 0-63 decimal to the D/A port, resulting in an analog voltage proportional to the number written to the port.

The analog to digital (A/D) input is a voltage comparator, made up of U14B, and some digital level shifting circuitry. The comparator has an input pin on connector CN3, where the voltage to be measured is applied. The comparator then compares this input voltage to the D/A ladder output voltage, and sends a digital signal to the SID input of the 8085. This digital signal is high (or 1) when the D/A voltage is higher than the input voltage and the digital signal is low (or 0) when the D/A voltage is lower than the input voltage. The PRIMER's monitor operating system contains a built in program which converts an input voltage to a decimal value in the range of 0-63. Using this built in program within other programs and attaching the appropriate hardware will allow the PRIMER to function as a voltmeter, temperature gauge, pressure gauge or any other kind of analog measurement device. The PRIMER can then be used as what is often referred to as an "embedded controller".

The final area of the PRIMER schematics is the power supply section. In the PRIMER only a very simple power supply is required. The D.C. input is fed from power jack J1, and is pre-filtered by C8, then applied to an LM7805 regulator. The LM7805 regulator is an industry standard device found in almost all small microprocessor systems, because its application is very simple. The LM7805 regulator steps the 7 to 10 VDC power supply input down to the +5 volts required by the microprocessor and all the other digital logic chips. Some of the input power bypasses the LM7805, and goes directly to the LM358 dual op-amp chip, U14. The LM358 needs " voltage overhead " to work correctly for the analog voltages it operates with.

Capacitor C9, and seven other capacitors spread about the board provide D.C. power supply bypass. These bypass capacitors stabilize the power supply voltage fed to the logic devices on the PRIMER.

HARDWARE RESET: The PRIMER board can be reset through the reset button provided on the board. There is a reset output pin on the expansion connector which allows the resetting of the PRIMER to reset any devices that may be connected to the expansion connector.

SERIAL COMMUNICATION PORT: The PRIMER uses an RS232 standard serial communication port. The port interfaces to a PC or terminal through a DB-9 shell connector (communication cables are available as an accessory from EMAC). The serial communication rate (or baud rate) must be set to the rate used by the terminal or PC. Placement of jumper JP1 can set the baud rate from 300 baud up to 19,200 baud. MOS services are available which access the 8251 serial communication port.

DIP SWITCH: The DIP switch has 8 switches which may be used for applications such as selection of program options. The DIP switch is connected to the system data bus and is accessed through I/O, address 012H. A MOS service is available which accesses the DIP switch.

DIGITAL OUTPUTS: The PRIMER has 8 outputs and each output can be independently programmed to an ON (+5v or binary 1) or OFF (0v or binary 0) state. The outputs are connected directly to the digital output LEDs and an LED can be turned on by the output of a binary 0, and turned off by the output of a binary 1. The outputs are also connected to the digital I/O connector CN3.

The output is driven by the 8155 I/O I.C. which, in the standard configuration, uses PORT A (11H) as the output port, PORT B (12H) as the input port and PORT C (13H) as an analog output port.

NOTE: The standard configuration of the 8155 which is set up by MOS should not be changed. Also, since PORT A outputs have limited drive capabilities, buffering should be considered if these outputs are to be used.

DIGITAL INPUTS: The PRIMER allows 8 inputs through 8155 port B. These inputs are connected directly to the 8 station DIP switch. The inputs are also connected to digital I/O connector CN3, so if the DIP switches are all turned off, you may connect external TTL level input devices through this connector.

D/A: An analog output voltage in the range of approximately 0 to +5v can be output from the PRIMER. This digital to analog convertor is implemented through an R-2R ladder which is connected to bits 0-5 of output PORT C. The output from the R-2R ladder is available on pin 19 of the digital I/O connector CN3. MOS provides a service which uses this D/A convertor.

A/D: The PRIMER provides an analog input (digital I/O Connector CN3 pin 20) which can convert a voltage in the range of 0 to +5 volts to a 6 bit value. This conversion is done by a MOS service using the D/A convertor and a comparator. The service starts by outputting 0 volts from the D/A convertor and then increasing the output voltage until the comparator senses that the output voltage exceeds the input voltage. When the input voltage has been exceeded, the last number that was output to the D/A convertor is the digital representation of the analog input voltage. The schematic of the circuitry for D/A and A/D is on schematic page 6.

> **Note:** Since the service that performs the A/D conversion uses the D/A convertor, the D/A convertor cannot be used at the same time an analog signal is being converted to digital.

TIMER/COUNTER: The PRIMER comes equipped with a 14 bit timer/event counter. This timer/counter is resident in the 8155 I/O I.C. By loading a user programmable termination count, time intervals from 3.25 microseconds to 53.3 milliseconds. When the termination count is reached, a RST 7.5 interrupt can then be issued to the CPU. If interrupts are not desirable the timer can be read directly or the interrupt line can be polled. The timer can also be set up to reload itself or to stop counting upon reaching the termination count. In either case, an interrupt can be issued.

The timer/counter is a 14 bit down counter that counts the 'timer input' pulses and provides a pulse or square wave to the 8085 RST 7.5 interrupt when the terminal pulse is reached. The user can reprogram the length of the count before the termination pulse is reached if so desired. The user can also determine the timer interval by programming the counter register from values 2H to 3FFFH. The timer/counter has four operating modes which are:

Mode 0 No operation mode (NOP) does not affect the timer.

Mode 1 Stop mode stops the timer/counter if it is running otherwise NOP.

Mode 2 Stops the timer if running immediately after the terminal count has been reached otherwise NOP.

Mode 3 The start mode loads the output mode and count length and starts the timer/counter immediately if timer is not running, otherwise it waits for the terminal count then starts the timer/counter.

The timer/counter has four output modes which are as follows:

Mode 0 Outputs a low during the second half of the count, which is equivalent to a single square wave.

Mode 1 Outputs a continuous square wave when the terminal count is reached.

Mode 2 Outputs a single pulse when the terminal count is reached.

Mode 3 Outputs a single pulse and reloads automatically.

The timer/counter operating modes are programmed through the 8155 control register (I/O address 10H). The lower 8 bits of the count length is written to I/O address 14H. The upper 6 bits of the count length along with the 2 bit output mode is written to I/O address 15H. The timer/counter can be used as an external program interval timer. If you wish to perform a software operation at a specific time interval, then the timer/counter can be programmed to that interval. Upon the resulting timer interrupt your program can execute the desired software.

The timer/counter is also used to drive the PRIMER's speaker. Different frequencies can be output from the speaker using a MOS service.

ADDITIONAL DETAILS ON THE 8155 TIMER/COUNTER MAY BE OBTAINED FROM INTEL CORPORATION'S LITERATURE DEPARTMENT.

EXPANSION CONNECTOR

The PRIMER has a 40 pin expansion connector (CN1) on board which provides additional expansion capabilities. Primarily, this port gives access to the Data and Low Address Busses and Control Lines. See Appendix D "EXPANSION CONNECTOR CN1" drawing for a detailed description.

NOTE: EMAC HAS MODULAR EXPANSION BOARDS AVAILABLE FOR THIS CONNECTOR.

Languages Used By the PRIMER

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      equ          12h           ; port for the DIP switches
2  num       equ          4             ; number of reads of the DIP switches
3            org          0ff01h        ; 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 FF01.

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.

Before attempting to perform the lessons that follow make sure that in option jumper 1 (OJ1), pins 1 and 2 are connected together, as well as pins 3 and 4. You should have a MOS or EMOS EPROM in the U2 socket and both OJ2 and OJ3 should have connectors in position "A" if you are using MOS or position "B" if you are using EMOS

You need to provide the PRIMER with an appropriate power source. The PRIMER requires a power supply in the range of 7 to 10 volts DC that can supply more than 480 milliamps of current. This power may be taken from a bench power supply, a wall mounted power supply or any other suitable power source. The power supply's output plug tip must be positive and the sleeve must be negative. A wall mounted power supply that meets all of the previous stated requirements may be obtained from EMAC Inc.

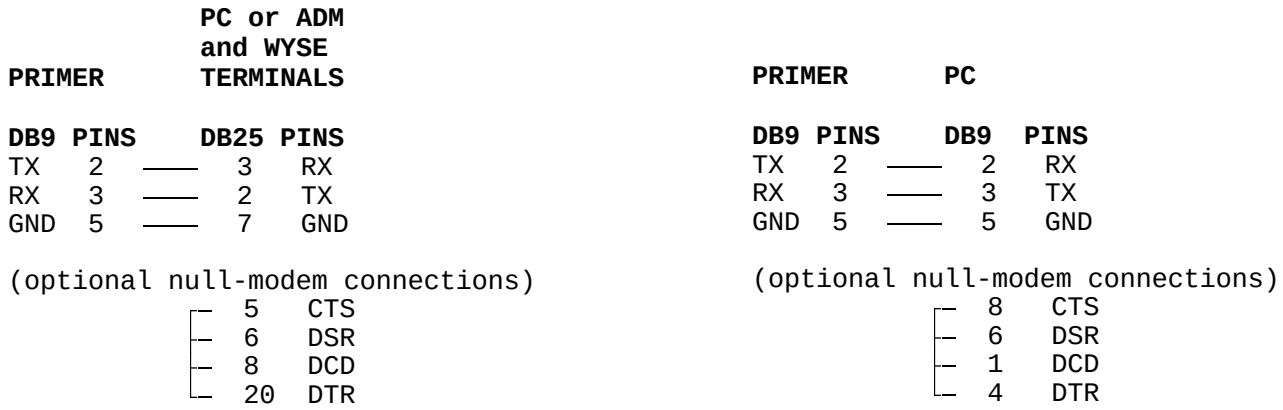
Once power has been correctly applied to the PRIMER's power jack, press the reset button to assure that the system is properly initialized. The PRIMER should give a tone and then show hex numbers on the digital displays. If this doesn't happen after about 2 seconds remove the power and make sure that the power supply meets the above stated requirements. If the power supply meets the requirements, but nothing happens you may need to do some hardware troubleshooting or send the board back to EMAC for repair.

If the PRIMER is now running you may want to run the PRIMER diagnosis function. This function allows you to check the DIP switches, digital output LEDs, A/D convertor, 8155 timer, speaker, numeric displays, keypad and the optional serial RS-232 port. If you do not have the serial communications option or if you have it but do not want to test it, skip to **To begin diagnosis:**.

Before the PRIMER can communicate through the serial port, its baud rate must be the same as the PC or terminal the PRIMER is communicating with. The PRIMER's baud rate can be set by placing a jumper in JP1 in the position corresponding to the desired baud rate. The baud rates are labeled 300, 600, 1200, 2400, 4800, 9600 and 19,200 next to JP1. The PC or terminal must use serial protocol with 1 stop bit 8 data bits and no parity.

The following information regarding the cable assembly should be followed carefully to assure correct operation.

"Handshaking" lines are not required by the PRIMER but may be necessary for the IBM PC and compatibles used as terminal emulators. If the terminal emulator requires handshaking lines, wire a null modem cable by connecting together the PC's RS-232 handshake lines CTS, DSR, DCD and DTR. The diagrams below show the required connections for PC's and terminals with DB25 connectors and PC's with DB9 connectors.



Note: When a data terminal has 2 connectors, use the one marked "MODEM".

To begin diagnosis:

Press the "Func." key then "1". When the diagnosis begins a checksum is performed on the ROM and if an error is detected you will hear a beep and "b.E." will be shown on the right 2 displays indicating "Bad EPROM". If there was an error, pressing a key will resume the diagnostics. If no error occurred "....r.d." will be shown on the numeric displays indicating that "RAM Diagnostics" are occurring. If a faulty RAM location is detected, its memory address will be shown on the left 4 numeric displays and "b.r." (indicating Bad RAM) will be shown on the right 2 displays. Note that if you don't have a 32K RAM and you get a bad RAM error at 8000 or 4000, this is not a valid error. Pressing a key following this error message will cause the diagnosis to continue. The RAM diagnostics should be completed in under 30 seconds, if the RAM is okay. If an optional 32k RAM

is in memory slot 1 and option jumper OJ3 is in position "A" you should turn off the PRIMER and move the jumper to position "B" then turn it on again and run the diagnosis function. If the RAM is okay, and you want the position "A" memory map again, turn off the PRIMER and move the jumper to its original position. Note that no memory check is done on the RAM within the 8155 chip if a 32k RAM is in slot 1.

After the RAM diagnostics, the PRIMER will try to determine whether there is a serial port. If there isn't or if it is not working, "n.u." will be displayed (indicating No UART). If a serial port is detected, it is examined to see if it is configured for a local loopback test. It is configured for local loopback when the transmit and receive lines of the serial port are connected to each other (pins 2 and 3 respectively of the DB9 connector CN2). This connection allows data transmitted by the UART to be looped back to the UART, testing both the transmitter and the receiver.

If it is configured for local loopback and the serial port is working, the bytes 00 to FF hex will be transmitted and displayed on the left 2 displays (this will happen quite rapidly at higher baud rates, so if you want to watch it, set JP1 to around 2400 baud). If there is a problem with the serial communications, you will hear a beep and "b.S." will be shown on the right 2 displays indicating a Bad Serial port. Pressing a key after this will resume the diagnostics and the UART will be disabled. If the local loopback test runs without errors "L.L." will be shown on the left 2 displays, the UART will be disabled and the diagnostics will continue. If "L.L." is not displayed and it has been verified that the transmit and receive lines were connected together, there is a problem with the serial port circuitry.

If the PRIMER serial communications option is not configured for local loopback but instead is connected to a terminal, the following will be shown on the terminal display:

```
 UART test
>
```

If you type a key at the terminal, its hexadecimal ASCII value will be shown on the left two displays on the PRIMER, and the character will be echoed back to the terminal display. For example if the letters "A" and "B" are typed at the terminal, the following will be shown on the terminal display:

```
 UART test
>A
>B
>
```

Note: If your terminal has the ability to "auto echo" you will see 2 characters displayed for each key pressed.

Also, after the RAM check is done, the hexadecimal representation of the A/D input will be shown on the right 2 displays. If you want to test the A/D convertor you need to connect a variable voltage source, ranging from 0 to +5 volts, to the analog input of the external digital I/O connector CN3 (this is above and to the left of the ADDRESS/REGISTER PAIR displays). This can be simply done with a 10K potentiometer by connecting the wiper to the analog input and one of the other two connections to +5V and the other to ground. Connector CN3 provides +5 volts on pin 22, analog input on pin 20 and ground on pin 18. When the analog input voltage is ground the display should show "00". As you slowly increase the voltage to 5 volts, the display will show a value from 02 to 3F hex. Also when the display no longer shows "00" the speaker will begin to make a high pitched tone and which will gradually become lower pitched as the voltage approaches 5 volts. Turn the voltage back to 0 and the tone will stop.

Each DIP switch is programmed to control an individual digital output LED. The best way to test the DIP switches and the LEDs is to turn each of the switches on, allowing only one switch on at a time. Then turn all the switches on at once, and finally turn all of them off.

Pressing one of the keys on the keypad will cause the hexadecimal value of that key to be shown on the middle two displays. The hexadecimal values of the keys, starting at the top row and reading from left to right are 00 to 0F for the first 4 rows and 14 to 17 for the last row.

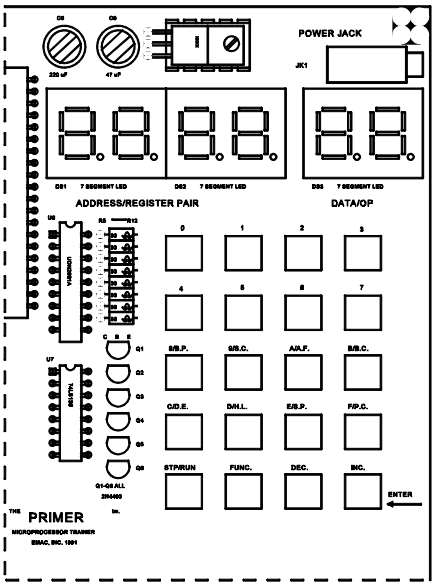
When you want to return to the Monitor Operating System, just press the reset button.

LESSON 1: Using the Monitor Operating System

The Monitor Operating System allows the user to:

- view and change memory contents
- view and change register contents
- view and change stack contents
- execute one instruction at a time
- run a program
- select a breakpoint

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



KEYPAD AND DISPLAYS

VIEWING AND CHANGING MEMORY CONTENTS

When you first turn on the PRIMER trainer, the displays marked "ADDRESS/REGISTER PAIR" will show "FF01" and the display marked "DATA/OP" will show some random byte in hex. The number on the "ADDRESS/REGISTER PAIR" display is the value of the program counter register (PC) and the number on the "DATA/OP" display is the data at the memory address ***pointed to*** by the program counter. Since PC = FF01 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 on the "DATA/OP" display. Below is an illustration the program counter pointing to the data at memory address FF01. The values shown in the memory addresses in the diagram most likely are not the actual values that are in the PRIMER's memory. This is because the memory contains mostly random values when the PRIMER is turned on; The values were chosen just for examples.

```
ADDRESS     DATA
FF01        DB     <-PC
FF02        12
FF03        D3
:
: (memory addresses FF04-FFFF)
```

When an address and the data at that address is displayed, the PRIMER is in "data entry mode". Press the "enter" key and the PC register will be

incremented to FF02 and shown on the "ADDRESS/REGISTER PAIR" display and the data pointed to by the new value of the PC will be shown on the "DATA/OP" display. If the data at memory address FF02 happened to be 12 hex, then "12" would be shown on the "DATA/OP" display. The program counter pointing to the data at memory address FF02 is illustrated below.

```
ADDRESS     DATA
FF01        DB
FF02        12    <-PC
FF03        D3
:
: (memory addresses FF04-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 on the "DATA/OP" display is the byte of data pointed to by the new PC memory address. Press the "Dec." key until PC is FF01 again or press the reset button.

Type "F" and "C", and you will see the hex number "FC" on the "DATA/OP" display. 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 FF02 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 FF01 is now 07. Press the "enter" key and you will see that the data at address FF02 is not 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 following format:

```
ADDRESS     DATA        INSTRUCTION
FF01        DB          IN   12
FF02        12
FF03        D3          OUT  11
FF04        11
FF05        C3          JMP FF01
FF06        01
FF07        FF
```

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 FF01. 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 FF01, type DB, which is the data from the table that belongs in that address, and press "enter". The PC is now FF02 so type 12 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 FF01 again. Once you become comfortable with entering programs into memory you may want to skip the verification process. Don't be alarmed if you type a program incorrectly and it doesn't work, because it is impossible for a program to damage the PRIMER. The worst thing that can happen is that you would have to press the reset button, or if the program was corrupted you may have to enter the program into memory again.

VIEWING AND CHANGING REGISTER CONTENTS

It is a good learning aid to be able to examine the values of the 8085's registers. To view the contents of a register you must first press the "func." key (which will cause the display to show "Func.") and then the key corresponding to the register you want to examine. When the "func." key is pressed, the next key pressed will invoke the alternate function of the key. The keys that have alternate functions have the standard function followed by a slash "/" and the alternate function. For example the "8/B.P." key's standard function is digit 8 and after the "func." key is pressed the alternate function "B.P." will be invoked.

Each line in the table below shows the key that should be pressed after pressing the "func." key and the data that will be shown in the "ADDRESS/REGISTER PAIR DISPLAYS" as a result.

ADDRESS/REGISTER PAIR DISPLAYS

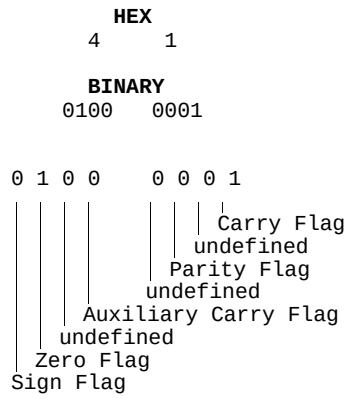
KEY	DATA DISPLAYED ON DS1 (LEFT DISPLAY)	DATA DISPLAYED ON DS2 (RIGHT DISPLAY)
A.F.	= A register	flag register
B.C.	= B register	C register
D.E.	= D register	E register
H.L.	= H register	L register
S.P.	= High order byte of SP	Low order byte of SP
P.C.	= High order byte of PC	Low order byte of PC

Pressing one of the above keys after the "func." key is pressed will cause the value in the registers to be shown in the displays marked "ADDRESS/REGISTER PAIR" and the register name to be shown in the display marked "DATA/OP". After this, pressing any key other than the numeric keys (0-F) will cause the PRIMER to return to the data entry mode.

If you want to change the value of the A register, press the "func." key and then 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","enter". After pressing the "enter" 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 "func." key, then 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", "enter".

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 "enter", 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 (see diagram to the right).



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. If the flag register is D5, all the defined flags are set. You may need to refer back to this diagram in later lessons when they ask you to examine the flag register. A shortcut to determining the status of the carry flag is to examine the rightmost digit of the flag register. If the digit is even (0,2,4,6,8,A,C,E) then the carry flag is 0, otherwise, if the digit is odd (1,3,5,7,9,B,D,F) the carry flag is 1.

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 "enter" 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 "enter" since you started doing this, you can press the "step","func." or "dec." keys to return to the data entry mode and the register will not be changed.

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 output LEDs and then repeats these instructions indefinitely. The program may be stopped by pressing the reset button.

ASSEMBLER PROGRAM

```
leds        equ   11h
dips        equ   12h

            org   0ff01h
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
FF01	DB	IN 12
FF02	12	
FF03	D3	OUT 11
FF04	11	
FF05	C3	JMP FF01
FF06	01	
FF07	FF	

To run this machine language program, turn on the PRIMER Trainer and the program counter address FF01 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 FF01 by pressing the reset button or loading the PC register with FF01, then run the program by pressing "Func." and the "Step/Run" key. You should see that as you move the DIP switches the digital output LEDs will turn off or on accordingly. Now press the reset button and the program will stop, the digital output LEDs will turn off and the display will show "FF01 db".

The PRIMER 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 PRIMER will be returned to data entry mode. To single step the first instruction, press the "stp/run" key and the displays will show "FF03 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 "FF05 C3". the OUT 11 instruction has been executed and the PC register was returned with the address of the JMP FF01 instruction.

2) Single step again and the display will show "FF01 db". This display shows FF01 because that is the value that was loaded into the PC register by the JMP FF01 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 IN 12 instruction again.

3) Single step again and the display will show "FF03 d3". The IN instruction has been executed and the PC register was returned with the address of the OUT 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 "Func." then the "A/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 output 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 output LEDs.

*** NOTE:** The PRIMER was designed so the digital output LEDs will turn on when a logic 0 is output to a particular bit of port A. So, for example, if 01h is output to the port all LEDs will be turned on except for LED 0, and if F0h is output to the port, all LEDs will be turned off except LEDs 3 to 0.

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    0ff01h
            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
FF01	3E	MVI A,1
FF02	01	
FF03	06	MVI B,2
FF04	02	
FF05	0E	MVI C,3
FF06	03	
FF07	16	MVI D,4
FF08	04	
FF09	1E	MVI E,5
FF0A	05	
FF0B	26	MVI H,6
FF0C	06	
FF0D	2E	MVI L,7
FF0E	07	
FF0F	EB	XCHG
FF10	EB	XCHG
FF11	78	MOV A,B
FF12	41	MOV B,C
FF13	FF	RST 7

Enter the program into memory and then press the reset button which will change the program counter to FF01. 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 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 "8/B.P." key and the display will show the breakpoint address in the "ADDRESS/REGISTER PAIR" displays and "B.P." in the "DATA/OP" display. 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 "enter". Remember that B.P. (breakpoint) is not a register within the 8085 microprocessor, it is just a function supported by the MOS. Follow these steps:

CURRENT PC

FF01 Set a breakpoint at address FF0F and run the program. After the breakpoint occurs the PRIMER will be in data entry mode displaying the address FF0F. 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.".

FF0F Single step and the first XCHG 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.

FF10 Single step and the second XCHG 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 XCHG instruction.

FF11 Single step and the instruction MOV A,B will be executed which copies the B register to the A register. Examine these registers to verify this.

FF12 Finally, single step the MOV B,C instruction and the C register will be copied to the B register. Examine the BC register pair.

FF13 This is the end of the program.

The last instruction in the machine language program is FF which is the RST 7 instruction. Read the instruction's definition at the beginning of this lesson, again (stack will be explained later). The PRIMER's ROM has instructions at address 0038 which return control of the microprocessor to the MOS. Try to single step the RST 7 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 RST 7 instruction and returns to the data entry mode showing "FF13 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 with the result being stored in 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   0ff01
loop:       mov   a,b    ; A is a copy of B
            add   c      ; Add C to A
            rst   7      ; return control to MOS
            end
```

ADDRESS	DATA	INSTRUCTION
FF01	78	MOV A,B
FF02	81	ADD C
FF03	FF	RST 7

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 run the program. The program will stop at FF03 because of the RST 7 instruction. (Because the microprocessor is so fast the program stops almost immediately, so you may think that it didn't actually run the program, but by examining the registers you will see that it did.) 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) Press the reset button and load the BC register pair with FFFF then run the program. When the program stops at FF03 examine the A register and flag register and you will see that the A register is now FE and that bit 0 of the flag register (the carry flag) 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 with the result being stored in 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 FF02 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   0ff01
loop:       mov   a,b    ; A is a copy of B
            sub   c      ; subtract C from A
            rst   7      ; return control to MOS
            end
```

ADDRESS	DATA	INSTRUCTION
FF01	78	MOV A,B
FF02	91	SUB C
FF03	FF	RST 7

1) To test the SUB C instruction, load the program into memory and move the program counter to FF01. Load the BC register pair with 0906, which is the same as loading B with 9 and C with 6, then run the program. When the program stops at FF03, 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.

2) Press the reset button, load B with 1F and C with 3D then run the program. 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 with the result being stored in the A register. The Z,S,P,CY and AC flags are affected.

SBB D op code = 9A
Subtracts the D register and the carry flag from the A register with the result being stored in 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   0ff01
                ; 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
FF01	79	MOV A,C
FF02	93	SUB E
FF03	4F	MOV C,A
FF04	78	MOV A,B
FF05	9A	SBB D
FF06	47	MOV B,A
FF07	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 FF01 you will see the following after each step:

CURRENT PC

FF01 A equals C which is 00.

FF02 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.

FF03 C equals A which is 01.

FF04 A equals B which is 21 hex.

FF05 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).

FF06 B equals A which is 10 hex

FF07 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
Adds the E register to the A register, storing the result in the A register. The Z,S,P,CY and AC flags are affected.

ADC D op code = 8A
Adds the D register and the carry flag to the A register, storing the result in 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   0ff01
                    ; 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
FF01	79	MOV A,C
FF02	83	ADD E
FF03	4F	MOV C,A
FF04	78	MOV A,B
FF05	8A	ADC D
FF06	47	MOV B,A
FF07	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 FF02 to 83 and the byte at address FF05 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 FF01 you will see the following after each step:

CURRENT PC

FF01 A equals C which is 02.

FF02 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.

FF03 C equals A which is 01.

FF04 A equals B which is 23 hex.

FF05 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).

FF06 B equals A which is 34 hex

FF07 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
FF01        09          DAD  B
FF02        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 to HL repeatedly. Note that the result in the carry flag will not indicate a borrow as it did in the SUB and SBC instructions.

```
            org   0ff01h
            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
FF01        F3          MOV  A,B
FF02        2F          CMA
FF03        47          MOV  B,A
FF04        79          MOV  A,C
FF05        2F          CMA
FF06        4F          MOV  C,A
FF07        03          INX  B
FF08        09          DAD  B
FF09        C3          JMP  FF08
FF0A        08
FF0B        FF
```

Enter the above program into memory, then load HL with FFFF and BC with 00FF. Set a breakpoint at FF08 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 DAD instruction and HL will be FF00 (FFFF - FF), single step the JMP and DAD again and HL will be FE01 (FF00 -FF) and so on. With each DAD 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                 67                 43
+29(hex add)       +75(hex add)       +72(hex add)
 32                 DC                 B5
+06(DAA)           +66(DAA)           +60(DAA)
 48                142                115
```

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   0ff01
            ; 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
FF01        79          MOV  A,C
FF02        83          ADD  E
FF03        27          DAA
FF04        4F          MOV  C,A
FF05        78          MOV  A,B
FF06        8A          ADC  D
FF07        27          DAA
FF08        47          MOV  B,A
FF09        FF          RST  7
```

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

CURRENT PC

FF01 Single step to FF03 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.

FF03 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.

FF04 Single step to FF07 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.

FF07 Single step the DAA instruction at this address and by examining the A register you will see that 6 was added to it.

FF08 Single step and the A register will be copied to the B register.

FF09 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 FF03 and FF07 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
Rotate all the bits in the accumulator one position left and put the carry flag in bit 0 and put the value that was rotated out of bit 7 into the carry flag. Only the carry flag is affected.

DAD H op code = 29
Add HL to HL and store the result in 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 and store the result in 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 A register is not changed.

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 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. The RAL instuction, because of the way it uses the carry flag will allow you to write programs which shift bits from register to register which can allow shifting of very large numbers. In the previous instructions the carry flag indicated a *carry* or *borrow*, but RAL uses it for two other purposes: RAL shifts the carry flag into bit 0 of the A register and then loads the carry flag with the value shifted out of bit 7 of the A register. The program below shows how the carry flag can be used to pass a bit from one register to another. The program shifts the bits in the DE register pair left one bit position.

```
            org   0ff01h
loop:       mov   a,e    ; load A with low byte first
            ora   a      ; clear the carry flag (cy=0) so RAL will shift 0 into bit 0
            ral          ; rotate left through carry
            mov   e,a    ; store in E
            mov   a,d    ; load A with high byte
                         ; what was shifted out of bit 7 in the last RAL is in the CY
flag..
            ral          ; ..which is shifted into bit 0 in this RAL
            mov   d,a    ; store in D
            rst   7      ; return to MOS
            end
```

ADDRESS	DATA	INSTRUCTION
FF01	7B	MOV A,E
FF02	B7	ORA A
FF03	17	RAL
FF04	5F	MOV E,A
FF05	7A	MOV A,D
FF06	17	RAL
FF07	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 0ff01h
loop:       dad   h          ; add hl to hl
            jmp   loop       ; do it again
```

ADDRESS	DATA	INSTRUCTION
FF01	29	DAD H
FF02	C3	JMP FF01
FF03	01	
FF04	FF	

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 = Y * 10" which is broken down into a form which the 8085 can easily handle:

```
X = Y * 10
X = (Y * 5) * 2
X = ((Y * 4) + Y) * 2
X = ((Y * 2 * 2) + Y) * 2
```

The last equation above is translated into the following program.

```
        org   0ff01h
        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
FF01	54	MOV D,H
FF02	5D	MOV E,L
FF03	29	DAD H
FF04	29	DAD H
FF05	19	DAD D
FF06	29	DAD H
FF07	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 FF01 (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 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   0ff01h
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
FF01	7A	MOV A,D
FF02	B7	ORA A
FF03	1F	RAR
FF04	57	MOV D,A
FF05	7B	MOV A,E
FF06	1F	RAR
FF07	5F	MOV E,A
FF08	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 = F6
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 check the status of an input port bit. The way this is done is illustrated below. The A register holds the value of the input port, and <byte> has a bit set which corresponds to the one you want to examine and all other bits are 0.

```
        00101111    (input port value)
ANI     00100000    (examine bit 5)
        00100000    (result)
```

All bits in the value of the input port 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 input port bit 5 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 copied to the result if the corresponding bit value in <byte> is 0, and a bit is set to 1 in the result 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 from input port B (connected to 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 PRIMER and to read data from them.

```
leds        equ   11h
dips        equ   12h
            org   0ff01h
loop:       in    dips        ; get the input port 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 discrete LEDs
            jmp   loop        ; jump to loop
            end
```

ADDRESS	DATA	INSTRUCTION	
FF01	DB	IN	12
FF02	12		
FF03	E6	ANI	3F
FF04	3F		
FF05	F6	ORI	30
FF06	30		
FF07	EE	XRI	06
FF08	06		
FF09	D3	OUT	11
FF0A	11		
FF0B	C3	JMP	FF01
FF0C	01		
FF0D	FF		

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

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

NOTE: If you are facing the PRIMER, the DIP switches are numbered 7-0 from left to right. Also, a DIP switch in the on position will produce a logic 0 input and in the off position will produce a logic 1 input.

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   11h
dips        equ   12h
            org   0ff01h
loop:       in    dips        ; load the A register with DIP switch 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.

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
FF01	DB	IN 12
FF02	12	
FF03	D3	OUT 11
FF04	11	
FF05	B7	ORA A
FF06	C2	JNZ FF01
FF07	01	
FF08	FF	
FF09	FF	RST 7

Press the reset button and enter the program into memory. Now set the DIP switches to the off position so the value of the DIP switches will not be zero. Run the program and it should work the same as the one in lesson 2, until the value of the DIP switches are changed to zero. This happens when the DIP switches are moved to the on position and when this is done, the program will execute the RST 7 instruction, return to the MOS and display "FF09 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

FF01 Move the DIP switches to the off position and single step to address FF06.

FF06 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 FF01 instruction will be executed because the conditional is true (since zero flag is 0).

FF01 Move the DIP switches to the on position, so that the value of the DIP switches wil be be 0 and single step to address FF06 again.

FF06 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 FF01 instruction won't be executed because the conditional is false (since zero flag is 1).

FF09 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

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

AFTER

ADDRESS	DATA
:(addresses	0000-FFC0)
:	
FFC1	32
FFC2	01
FFC3	03 **<-HL**
FFC4	14
:	
:(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 will copy a byte from one memory location to another and also store a value in memory illustrating the use of the HL register pair to indirectly access memory.

```
        org   0ff01h
        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
FF01	21	LXI H,FF0A
FF02	0A	
FF03	FF	
FF04	5E	MOV E,M
FF05	23	INX H
FF06	23	INX H
FF07	23	INX H
FF08	23	INX H
FF09	73	MOV M,E
FF0A	23	INX H
FF0B	36	MVI M,77h
FF0C	77	
FF0D	FF	RST 7
FF0E	00	(NOT AN INSTRUCTION, BUT DATA)
FF0F	00	(DATA)

Enter the program into memory then press reset which clears the general purpose registers and changes the program counter to FF01. Do the following:

CURRENT PC

FF01 Single step and examine the HL register which should be FF0A.

FF04 Single step and the data pointed to by HL (the data at memory address FF0A) will be put in the E register.

FF05 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.

FF09 Single step and the value in the E register will be stored at the new address pointed to by HL (at FF0E).

FF0A Single step and an INX H instruction will be executed.

FF0B Single step and 77 will be stored at the new address pointed to by HL (at FF0F).

FF0D This is the end of the program. Press "enter" and the data at FF0E will be displayed. You will see that the instruction at FF09 (MOV M,E) did store the value of the E register here. Press enter again and you will see that the instruction at FF0B (MVI M,77) actually stored 77 at FF0F.

Notice that this program loads the E register with a byte from a memory location within the program itself (FF0A) 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
Adds the byte in memory pointed to by the HL register pair to the A register, storing the result in the A register. The Z,S,P,CY and AC flags are affected.

ADC M op code = 8E
Adds the carry flag and the byte in memory pointed to by the HL register pair to the A register, storing the result in 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 FF10, FF11 and FF12 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 FF10 are the least significant bytes and the byte in C and the byte at FF12 are the most significant bytes.

```
        org   0ff01h
        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:    db    5Ch,0A3h,0Eh     ; 3 bytes of storage
        end
```

ADDRESS	DATA	INSTRUCTION
FF01	21	LXI H,FF10
FF02	10	
FF03	FF	
FF04	7B	MOV A,E
FF05	86	ADD M
FF06	5F	MOV E,A
FF07	23	INX H
FF08	7A	MOV A,D
FF09	8E	ADC M
FF0A	57	MOV D,A
FF0B	23	INX H

continued on next page...

```
FF0C        79          MOV  A,C
FF0D        8E          ADC  M
FF0E        4F          MOV  C,A
FF0F        FF          RST  7
FF10        5C          (least significant byte)
FF11        A3          (2nd most significant byte)
FF12        0E          (most significant byte)
```

Load the the above program and data into memory. After loading the program, the 3 byte number stored in memory addresses FF12, FF11 and FF10 will be 0EA35C. Press the reset button then load C with 01 and DE with F7BF so that C and DE will contain the 3 byte number 01F7BF, and do the following:

CURRENT PC

FF01 Single step and HL will be loaded with FF10 which is the address of least significant digit of the three byte number that is stored in memory.

FF04 Single step and the A register will be loaded with the value of the E register. Examine the A register (it should be BF)

FF05 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 FF10 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.

FF06 Single step and the A register will be copied to the E register.

FF07 Single step and HL will be FF11 which points to the second most significant byte of the three byte number stored in memory.

FF08 Single step and the D register will be copied to the A register. Examine the A register; it should be F7.

FF09 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 FF11 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.

FF0A Single step and the A register will be copied to the D register.

FF0B Single step and HL will be FF12 which points to the most significant byte of the three byte number stored in memory.

FF0C Single step and the A register will be loaded with the value of the C register. Examine the A register (it should be 01).

FF0D 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 FF12 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.

FF0E Single step and the A register will be stored in the C register.

FF0F 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 FF10, FF11 and FF12.

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. Copying blocks of data from one memory location to another is done quite often in word processor programs and programs that use graphics, such as video games. For this program 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   0ff01h
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 from which to get a byte.
        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
        db    0,0   ; make these bytes 0 so we can see they have changed
        end
```

ADDRESS	DATA	INSTRUCTION
FF01	0A	LDAX B
FF02	12	STAX D
FF03	03	INX B
FF04	13	INX D
FF05	2D	DCR L
FF06	C2	JNZ FF01
FF07	01	
FF08	FF	
FF09	FF	RST 7
FF0A	00	(data, not an instruction)
FF0B	00	(data also)

After entering the program into memory, press reset and load BC with FF03, load DE with FF0A, load HL with 0002 and do the following:

CURRENT PC

FF01 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 FF03 is 03).

FF02 Single step and the A register will be stored at the memory address pointed to by the DE register pair (FF0A).

FF03 Single step and the BC register pair will be incremented so that it points to the next memory address (FF04) from which to get data.

FF04 Single step and the DE register pair will be incremented so that it points to the next memory address (FF0B) to store data.

FF05 Single step and the L register will be decremented. Since the result is not 0, the zero flag is 0.

FF06 Single step and JNZ FF01 will load PC with FF01 since the zero flag is 0.

FF01 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 FF04 is 13).

FF02 Single step and the A register will be stored at the memory address pointed to by the DE register pair (FF0B).

FF03 Single step and the BC register pair will incremented so that it points to the next memory address (FF05) from which to get data.

FF04 Single step and the DE register pair will incremented so that it points to the next memory address (FF0C) to store data.

FF05 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.

FF06 Single step and since the zero flag is 1 the PC register will be loaded with FF09 which is the address of the instruction following the JNZ FF01 instruction.

FF09 This is the end of the program. Press "enter" and you will see that the byte at FF03 has been copied to memory address FF0A; Press enter again and you will see that the byte at FF04 has been copied to memory address FF0B.

Press reset, load BC with FF01 and load DE with FF0A 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 FF0A. Examine the memory from FF0A to FF12 and you will see that it matches the memory from FF01 to FF09. 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 FF01 to FF09.

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 FF01 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 latter will cause the L register to be changed to FF hex and the instructions will be repeated 256 times.

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   0ff01h
            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
FF01	21	LXI H,8000
FF02	00	
FF03	80	
FF04	11	LXI D,FF00
FF05	00	
FF06	FF	
FF07	3E	MVI A,1
FF08	01	
FF09	32	STA FF24
FF0A	24	
FF0B	FF	
FF0C	19	DAD D
FF0D	22	SHLD FF22
FF0E	22	
FF0F	FF	
FF10	2B	DCX H
FF11	7C	MOV A,H
FF12	B5	ORA L
FF13	C2	JNZ FF10
FF14	10	
FF15	FF	
FF16	3A	LDA FF24
FF17	24	
FF18	FF	
FF19	D3	OUT 11
FF1A	11	
FF1B	07	RLC
FF1C	2A	LHLD FF22
FF1D	22	
FF1E	FF	
FF1F	C3	JMP FF09
FF20	09	
FF21	FF	
FF22	FF	(dlaytm least significant byte)
FF23	FF	(dlaytm most significant byte)
FF24	FF	(leddta)

Enter the program into memory and do the following:

CURRENT PC

FF01 Single step once and HL will be loaded with the constant 8000h.

FF04 Single step again and DE will be loaded with -100h (FF00h) so subtraction can be performed by using DAD D.

FF07 Single step and the A register will be 1.

FF09 Single step and the A register will be copied to the variable space at FF24.

FF0C Load the PC register with FF24 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 FF0C. 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.

FF0D Single step and HL will be stored in the 16 bit variable space. Load the PC register with FF22 and you will see that the L register has been stored at FF22. Press enter and you will see that the H register has been stored at FF23. Load the PC register with FF10.

FF10 Set a breakpoint at FF16 and run the program from the current address (FF10). 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.

FF16 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 FF24 earlier.

FF19 Single step twice and the A register will be output to the digital output LEDs and then its value will be rotated left.

FF1C 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 SHLD FF22.

FF1F 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 equal to or less than 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 )
FF01        F3          DI
FF02        21          LXI  H,FF20
FF03        20
FF04        FF
FF05        22          SHLD FFE9
FF06        E9
FF07        FF
```

continued on next page...

```
      :
      : ( addresses FF08 - FFCF )
      :
      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          VALUE FROM C REGISTER
      FFD3        12          VALUE FROM 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 working program )
      :
      :
SP->  FFD0        78          VALUE FROM FLAGS
      FFD1        56          VALUE FROM A REGISTER
      FFD2        34          VALUE FROM C REGISTER
      FFD3        12          VALUE FROM 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. In other words, the data that was in the PSW when it was pushed on the stack will be copied into the HL register pair. Similarly if PUSH BC was executed followed by POP HL, then B would be copied to H and C would be copied to L.

To demonstrate the way PUSH and POP works, load the program counter register with FFCB then load the following program into memory, exactly as it is shown.

```
      org   0ffc7h      ; 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   ; there are zeros in the stack area so
      db    0,0,0,0,0   ; it will be in a known initial state
```

```
      ADDRESS     DATA        DESCRIPTION
PC->  FFC7        F5          PUSH  PSW
      FFC8        C5          PUSH  B
      FFC9        F1          POP   PSW
      FFCA        C1          POP   B
      FFCB        00
      FFCC        00
      FFCD        00
```

continued on next page...

```
        FFCE        00
        FFCF        00
        FFD0        00
        FFD1        00
        FFD2        00
        FFD3        00
SP->    FFD4        00          (stack pointer is here)
```

Load the program counter with FFC7 again and load the A register with 01, the flags register with 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 "func." key and then the "9/S.C." key and the stack contents will be displayed in the "ADDRESS/REGISTER PAIR" displays. Remember that S.C. is not an 8085 internal register, it is just a way that the MOS provides for viewing the two bytes on top of the stack.

3) Single step once at address FFC7. 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
        FFC7        F5          PUSH  PSW
PC->    FFC8        C5          PUSH  B
        FFC9        F1          POP   PSW
        FFCA        C1          POP   B
        FFCB        XX          (undefined value)
        FFCC        XX          (undefined value)
        FFCD        XX          (undefined value)
        FFCE        XX          (undefined value)       (See note below)
        FFCF        XX          (undefined value)
        FFD0        XX          (undefined value)
        FFD1        XX          (undefined value)
SP->    FFD2        02          value from flags
        FFD3        01          value from register A
        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
        FFC7        F5          PUSH  PSW
        FFC8        C5          PUSH  B
PC->    FFC9        F1          POP   PSW
        FFCA        C1          POP   B
        FFCB        XX          (undefined value)
        FFCC        XX          (undefined value)
        FFCD        XX          (undefined value)       (See note below)
        FFCE        XX          (undefined value)
        FFCF        XX          (undefined value)
SP->    FFD0        04          value from register C
        FFD1        03          value from register B
        FFD2        02          value from flags
        FFD3        01          value from register A
        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.

```
        ADDRESS     DATA        DESCRIPTION
        FFC7        F5          PUSH  PSW
        FFC8        C5          PUSH  B
        FFC9        F1          POP   PSW
        FFCA        C1          POP   B
PC->    FFCB        XX          (undefined value)
        FFCC        XX          (undefined value)
        FFCD        XX          (undefined value)
        FFCE        XX          (undefined value)
        FFCF        XX          (undefined value)       (See note below)
        FFD0        XX          (undefined value)
        FFD1        XX          (undefined value)
        FFD2        XX          (undefined value)
        FFD3        XX          (undefined value)
SP->    FFD4        00
```

Note: The "undefined values" shown in the machine language listing are values that MOS PUSHes and POPs from the stack on each breakpoint or single step. This is done without affecting the important values that your program stores on the stack, but this does affect the unused stack memory. So, though MOS returns with the stack pointer at the same place, some of the bytes below the stack pointer address will be changed because of the PUSHes. If it were possible to run the program full speed and stop it at FFCB without single stepping or breakpoints, the memory values would be defined as follows:

```
        ADDRESS     DATA        DESCRIPTION
        FFC7        F5          PUSH  PSW
        FFC8        C5          PUSH  B
        FFC9        F1          POP   PSW
        FFCA        C1          POP   B
PC->    FFCB        00
        FFCC        00
        FFCD        00
        FFCE        00
        FFCF        00
        FFD0        04          value from register C
        FFD1        03          value from register B
        FFD2        02          value from flags
        FFD3        01          value from register A
SP->    FFD4        00
```

Unfortunately there is no way to stop the program and view the memory without setting a breakpoint or using RST 7 so this cannot be demonstrated.

The stack has another purposes other than storing general purpose registers. It is also used by the RST 7 instruction which was discussed in an earlier lab and the XTHL, 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   0ff01h      ; 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
FF01	E5	PUSH H
FF02	21	LXI H,0000
FF03	00	
FF04	00	
FF05	44	MOV B,H
FF06	4D	MOV C,L
FF07	19	DAD D
FF08	D2	JNC FF0C
FF09	0C	
FF0A	FF	
FF0B	03	INX B
FF0C	E3	XTHL
FF0D	2B	DCX H
FF0E	7C	MOV A,H
FF0F	B5	ORA L
FF10	E3	XTHL
FF11	C2	JNZ FF07
FF12	07	
FF13	FF	
FF14	F1	POP PSW
FF15	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

FF01 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.

FF02 Single step and HL will be loaded with 0000 which is the beginning value for the sum.

FF05 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.

FF07 Single step and the DAD D instruction will cause DE to be added to the sum that is held in HL (examine HL).

FF08 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 FF0C instruction will be executed which will jump over the INX B instruction. Single step and PC will be FF0C, which is the address of the instruction following INX B.

FF0C 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.

FF0D Single step and 1 will be subtracted from HL which indicates that there is now 1 addition left to go.

FF0E 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.

FF0F 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.

FF10 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.

FF11 Single step and the JNZ instruction will be executed which will return the PC to FF07. 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.

FF07 Single step and the DAD D instruction will be executed which will add DE to the sum in HL again.

FF08 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 FF0C instruction won't be executed since the carry flag is 1, instead PC will be changed to the address of the instruction following JNC FF0C.

FF0B 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.

FF0C 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.

FF0D Single step and 1 will be subtracted from the number of additions left to do in the HL register.

FF0E Single step and the H register will be copied to the A register.

FF0F 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.

FF10 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).

FF11 Single step and the PC register will be loaded with FF14 which is the address of the instruction following the JNZ FF07 instruction. This is because the zero flag must be 0 in order for the instruction to load the PC register with FF07.

FF14 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.

FF15 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 output LEDs and delays for a period of time proportional to the value in HL.

```
leds        equ   11h          ; ouput port address for LEDs
            org   0ff01h
            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
```

continued on next page...

```
                        ; 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
FF01	21	LXI H,8000
FF02	00	
FF03	80	
FF04	3E	MVI A,0F
FF05	0F	
FF06	CD	CALL FF16
FF07	16	
FF08	FF	
FF09	2F	CMA
FF0A	CD	CALL FF16
FF0B	16	
FF0C	FF	
FF0D	3E	MVI A,55
FF0E	55	
FF0F	CD	CALL FF16
FF10	16	
FF11	FF	
FF12	25	DCR H
FF13	C3	JMP FF04
FF14	04	
FF15	FF	
FF16	E5	PUSH H
FF17	F5	PUSH PSW
FF18	D3	OUT 11
FF19	11	
FF1A	2B	DCX H
FF1B	7C	MOV A,H
FF1C	B5	ORA L
FF1D	C2	JNZ FF1A
FF1E	1A	
FF1F	FF	
FF20	F1	POP PSW
FF21	E1	POP H
FF22	C9	RET

Load the program into memory and run it; The program should display 3 different patterns on the discrete 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

FF01 Single step to the first CALL instruction at address FF06 and examine the stack content (it will be 0000) and the stack pointer will be FFD4.

FF06 Single step and the CALL FF16 instruction will be executed causing the program to jump to FF16, the start of the subroutine.

FF16 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 breakpoint at FF20, then run from the current address.

FF20 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.

FF21 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 (FF09) to the top of the stack. Examine the stack contents to verify this.

FF22 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.

FF09 Single step and the A register will be complemented.

FF0A Now that you understand what happens during the CALL of a subroutine, check the stack content to verify that it is 0000. Set a breakpoint at the address (FF0D) of the instruction following this CALL and run from the current address.

FF0D 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.

s 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 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 1000H. You may also do it without writing a program or affecting the PC register, by loading the appropriate registers and pressing "func." then "2". All registers that are not used as input or output to the service are not affected. Please refer to appendix F for the description of the MOS services.

An example program using MOS services:

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

```
mos         equ   1000h
ledhex      equ   12h
            org   0ff01h      ; 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
FF01	0E	; MVI C,12
FF02	12	
FF03	11	; LXI D,1234
FF04	34	
FF05	12	
FF06	CD	; CALL 1000
FF07	00	
FF08	10	
FF09	C3	; JMP 0FF09
FF0A	09	
FF0B	FF	

Load the program into memory, press the reset button then run it. You will see "1234" on the "ADDRESS/REGISTER PAIR" displays and it will continue to display until you press the reset button. Press the reset button and single step to FF06 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 PRIMER 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, 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 FF02 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   12h          ; port to read DIP switches
mos         equ   1000h        ; address of MOS services
            org   0ff01
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
FF01	DB	IN 12
FF02	12	
FF03	0F	RRC
FF04	0F	RRC
FF05	5F	MOV E,A
FF06	E6	ANI 3FH
FF07	3F	
FF08	57	MOV D,A
FF09	7B	MOV A,E
FF0A	E6	ANI 0C0H
FF0B	C0	
FF0C	5F	MOV E,A
FF0D	0E	MVI C,10H
FF0E	10	
FF0F	CD	CALL 1000H
FF10	00	
FF11	10	
FF12	C3	JMP FF01
FF13	01	
FF14	FF	

1) Switch the 4 DIP switches on the left "on" and the 4 on the right, "off". When this is done, the data that is input through the DIP switches will be 0F hex.

2) Press reset and load the program into memory and verify that it was loaded correctly.

3) Single step once and examine the A register to make sure it is 0F, if it isn't, do step 1 again.

In order for the DIP switches to give the broadest range of frequencies the 8 bit value of the DIP switches must be made the most significant 8 bits of the 14 bit value required 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	12h	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 FF0D and you will see that the 8 bit value of the DIP switch 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 MOS service 10 ignores 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 lesson is intended to be used by those with experience in basic electronics. Please proceed to the next lesson if you do not have basic electronics training.

This program will read the analog to digital convertor and then divide the number by 8 and produce a bar graph on the digital output LEDs that is proportional to the analog input voltage.

A variable voltage source, ranging from 0 to +5 volts, must be connected to the analog input of the external digital I/O connector CN3 (this is above and to the left of the ADDRESS/REGISTER PAIR displays). This can be simply done with a 10K potentiometer by connecting the wiper to the analog input and one of the other two connections to +5V and the other to ground. Connector CN3 provides +5 volts on pin 22, analog input on pin 20 and ground on pin 18.

Be sure there is no power to the PRIMER when connecting the potentiometer to the I/O connector. Also be sure that you do not short any of the connections when connecting the potentiometer to CN3. It may be best to use a mating connector, or to wire wrap the connections. Once the proper connections have been made and checked, apply power to the PRIMER and load the following program.

```
leds        equ   11h          ; output port for digital output LEDs
mos         equ   1000h        ; MOS CALL address
            org   0ff01h       ; start address of program
start:      mvi   c,9          ; service 9
            call  mos          ; get ADC
            mov   a,l          ; put in A register
            rrc                ; rotate A right = divide by 2
            rrc                ; divide by 2
            rrc                ; divide by 2
            ani   00011111b    ; remove upper 3 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
FF01	0E	MVI C,9
FF02	09	
FF03	CD	CALL 1000h
FF04	00	
FF05	10	
FF06	7D	MOV A,L
FF07	0F	RRC
FF08	0F	RRC
FF09	0F	RRC
FF0A	E6	ANI 01FH
FF0B	1F	
FF0C	CA	JZ FF18
FF0D	18	
FF0E	FF	
FF0F	4F	MOV C,A
FF10	3E	MVI A,0
FF11	00	
FF12	37	STC
FF13	1F	RAR
FF14	0D	DCR C
FF15	C2	JNZ LOOP
FF16	12	
FF17	FF	
FF18	D3	OUT 11
FF19	11	
FF1A	C3	JMP FF01
FF1B	01	
FF1C	FF	

Press reset and run the program. You will see that the bar graph displayed on the digital output 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 output LEDs depending on whether the key pressed was "3" or "0". If any other key is pressed the display is not changed.

```
leds        equ   11h          ; port for discrete LEDs
keyin       equ   0bh          ; service number for keyin
mos         equ   1000h        ; address of MOS services
            org   0ff01h       ; start program at FF01
            mvi   b,11110111b  ; 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
            rlc                ; rotate the bits in A left
            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
            rrc                ; rotate A right
            mov   b,a          ; put new bit pattern in B
            jmp   loop         ; display new bit pattern
            end
```

ADDRESS	DATA	INSTRUCTION
FF01	06	MVI B,F7
FF02	F7	
FF03	78	MOV A,B
FF04	D3	OUT 11
FF05	11	
FF06	0E	MVI C,0B
FF07	0B	
FF08	CD	CALL 1000
FF09	00	
FF0A	10	
FF0B	3E	MVI A,0
FF0C	00	
FF0D	BD	CMP L
FF0E	C2	JNZ FF14
FF0F	14	
FF10	FF	

continued on next page...

```
FF11        78          MOV  A,B
FF12        07          RLC
FF13        47          MOV  B,A
FF14        7D          MOV  A,L
FF15        FE          CPI  3
FF16        03
FF17        C2          JNZ  FF03
FF18        03
FF19        FF
FF1A        78          MOV  A,B
FF1B        0F          RRC
FF1C        47          MOV  B,A
FF1D        C3          JMP  FF03
FF1E        03
FF1F        FF
```

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

FF01 Set a breakpoint at address FF0D 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.

FF0D 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.

FF0E Single step and the JNZ FF14 instruction will be executed and you will see that because the zero flag is 0, the program counter will now point to address FF14. This skips the code that is to be done when the "0" key is pressed and goes to address FF14.

FF14 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.

FF17 This is the address of the JNZ FF03 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 FF03 rotate the value in the B register to the right when the "3" key is pressed.

FF1A Set a breakpoint at FF0D 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 FF0D.

FF0D This is the address of the CMP L 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 MVI A,0 instruction. Execute the CMP L instruction by single stepping and this time, since A equals L the Z flag will be 1 and the CY flag will be 0.

FF0E Single step and JNZ FF14 instruction won't execute this time because the zero flag is 1, instead the program counter will point to the instruction following this one.

FF11 The following instructions are executed when the "0" key is pressed. Single step to FF14 and the value in the B register will be rotated to the left.

FF14 Single step and the L register will be copied to the A register. Single step again and the CPI 3 instruction will be executed and since A is 0, the Z flag will be 0 and the CY flag will be 1.

FF17 The program counter now points to the JNZ FF03 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 FF03.

FF03 Run the program full speed from here to see the way the pressing keys affects the digital output 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 output 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   11h          ; output port for digital output LEDs
            org   0ff01h
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 program H is not less than L
                               ; and it is not equal to L, so H is greater than L.
            mvi   a,00111111b  ; pattern for H>L
            jmp   patout       ; output the pattern
lgreat:     mvi   a,11111100b  ; pattern for L>H
            jmp   patout       ; output the pattern
equal:      mvi   a,11100111b  ; pattern for H=L
patout:     out   leds         ; display bit pattern
            rst   7            ; end of program
```

ADDRESS	DATA	INSTRUCTION
FF01	7C	MOV A,H
FF02	BD	CMP L
FF03	DA	JC FF0E
FF04	0E	
FF05	FF	
FF06	CA	JZ FF13
FF07	13	
FF08	FF	
FF09	3E	MVI A,3F
FF0A	3F	
FF0B	C3	JMP FF15
FF0C	15	
FF0D	FF	
FF0E	3E	MVI A,FC
FF0F	FC	
FF10	C3	JMP FF15
FF11	15	
FF12	FF	
FF13	3E	MVI A,E7
FF14	E7	
FF15	D3	OUT 11
FF16	11	
FF17	FF	RST 7

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

CURRENT PC

FF01 Load HL with FFE0 so H will be greater than L then single step and H will be copied to the A register.

FF02 Single step and A will be compared with L and the flags will be set accordingly. According to the definition of the CMP L 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).

FF03 Single step and the PC will then point to the instruction following the JC FF0E instruction. Remember that because the carry flag is not 1 the JC FF0E instruction will not be executed.

FF06 Single step and the PC will point to the instruction following JZ FF13 because the zero flag must be 1 before the jump will occur.

FF09 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 11 instruction and it will display the bit pattern in the A register.

FF17 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:

FF01 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.

FF03 Single step and since the carry flag is 1 the JNC FF0E instruction will occur.

FF0E 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 11 instruction which will display the bit pattern in the A register.

FF17 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.

FF01 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.

FF03 Single step the JC FF0E instruction and it won't be executed because the carry flag is 0.

FF06 Single step and the JZ FF15 instruction will be executed because the zero flag is 1.

FF13 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 OUT 11 instruction which will display the bit pattern in the A register.

FF17 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 pin 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 which, in the case of this 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 be executed. The EI instruction is needed because whenever an interrupt occurs, the 8085 automatically performs a DI (disable interrupts). The RET instruction will return the program counter to the section of code that was interrupted.

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, while the 8155 timer is being set up and while the RST 7.5 interrupt mask is enabled through the SIM instruction.

The SIM instruction has two purposes: to control the SOD pin (this will not be used in this lesson), and to provide interrupt control. Using this instruction allows the RST 7.5, RST 6.5 and RST 5.5 interrupts can be disabled individually. Before using the SIM instruction to control interrupts the A register should be loaded with the following 8 bit binary value:

00011xyz

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.

The RST 7.5 interrupt pin is different than the other interrupt pins in that it has a flip flop connected to it. When a pulse appears on the RST 7.5 pin, the flip flop is set. If the flip flop is set and the interrupt is enabled, or it becomes enabled, the flip flop will be reset and the ISR will be executed. The ability of the flip flop to remember that an interrupt request has occurred while interrupts were disabled helps the microprocessor avoid missing interrupt requests. When enabling the RST 7.5 interrupt for the first time in a program you should clear this flip flop. This is done by making bit four of the A register a 1 before executing the SIM instruction. In this program we will enable only the RST 7.5 interrupt and reset its flip flop, so the binary value of 00011011 (1B 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 FF20, initialize the 8155 timer to send a 20hz square wave to the RST 7.5 pin 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 FF2E and sends the data to the discrete LEDs. The ISR will interrupt this loop 20 times per second and each time will increment the byte at address FF2E.

Type in the program and run it. You will see that the LEDs are displaying the increasing value of the data at address FF2B. Press the reset button and set a breakpoint at FF20, (the start of the ISR) then run the program. The execution will stop at FF20, and if you look at the stack contents (S.C.) 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 S.C. will be FF1B, if during the OUT instruction then S.C. will be FF1D or if during the JMP instruction S.C. will be FF18. Single step once, then set the breakpoint to FF20 again and then run full speed again. After the program stops, check the value of S.C. it could be any of the three addresses mentioned above.

```
vec7hlf     equ   0ffe9h      ; vector for 7.5 interrupt
leds        equ   11h         ; discrete LED port
                              ; 8155 timer ports
timerlo     equ   14h         ; low byte of timer
timerhi     equ   15h         ; hi byte and mode of timer
cmdreg      equ   10h         ; command register

            org   0ff01h
            di                ; disable interrupts
            lxi   h,ticsub    ; hl = address of ISR
            shld  vec7hlf     ; Store in vector
            mvi   a,0
            out   timerlo     ; lo 8 bits of counter are 0
            mvi   a,7ch       ; set 8155 square wave output mode
            out   timerhi     ; with timer at 20hz
            mvi   a,0cdh
            out   cmdreg      ; enable the timer
            mvi   a,11011b    ; a 0 bit enables the interrupt
            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
```

continued on next page...

```
            ; 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
FF01	F3	DI
FF02	21	LXI H,FF20
FF03	20	
FF04	FF	
FF05	22	SHLD FFE9
FF06	E9	
FF07	FF	
FF08	3E	MVI A,0
FF09	00	
FF0A	D3	OUT 14
FF0B	14	
FF0C	3E	MVI A,7C
FF0D	7C	
FF0E	D3	OUT 15
FF0F	15	
FF10	3E	MVI A,CD
FF11	CD	
FF12	D3	OUT 10
FF13	10	
FF14	3E	MVI A,1B
FF15	1B	
FF16	30	SIM
FF17	FB	EI
FF18	3A	LDA FF2B
FF19	2B	
FF1A	FF	
FF1B	D3	OUT 11
FF1C	11	
FF1D	C3	JMP FF18
FF1E	18	
FF1F	FF	
FF20	F5	PUSH PSW
FF21	3A	LDA FF2B
FF22	2B	
FF23	FF	
FF24	3C	INR A
FF25	32	STA FF2B
FF26	2B	
FF27	FF	
FF28	F1	POP PSW
FF29	FB	EI
FF2A	C9	RET
FF2B	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 "ADDRESS/REGISTER PAIR" displays. 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   0ff01h
mos         equ   1000h       ; start address of MOS subroutines
keyin       equ   0bh         ; 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
```

continued on next page...

```
            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 FF01). 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 FF02 and put 00 under the "DATA" heading. Do the same on the next line except put FF03 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
FF01        11          LXI   D,0
FF02        00
FF03        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 0B 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
FF01        11          LXI D,0
FF02        00
FF03        00
FF04        0E          MVI   C,0B
FF05        0B
FF06        CD          CALL  1000
FF07        00
FF08        10
FF09        3E          MVI   A,0F
FF0A        0F
FF0B        BD          CMP   L
FF0C        DA          JC
FF0D
FF0E
FF0F        26          MVI   H,0
FF10        00
FF11        19          DAD   D
FF12        EB          XCHG
FF13        C3          JMP
FF14
FF15
```

continued on next page...

```
FF16        0E          MVI   C,12
FF17        12
FF18        CD          CALL  1000
FF19        00
FF1A        10
FF1B        C3          JMP
FF1C
FF1D
```

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 FF01, the label "rdkey" points to the second instruction of the program at address FF04 and the label "dsplay" points to the instruction at address FF16. Fill in the addresses for each label and you will have the following completed listing:

```
ADDRESS     DATA        INSTRUCTION
FF01        11          LXI   D,0
FF02        00
FF03        00
FF04        0E          MVI   C,0B
FF05        0B
FF06        CD          CALL  1000
FF07        00
FF08        10
FF09        3E          MVI   A,0F
FF0A        0F
FF0B        BD          CMP   L
FF0C        DA          JC    FF16
FF0D        16
FF0E        FF
FF0F        26          MVI   H,0
FF10        00
FF11        19          DAD   D
FF12        EB          XCHG
FF13        C3          JMP   FF04
FF14        04
FF15        FF
FF16        0E          MVI   C,12
FF17        12
FF18        CD          CALL  1000
FF19        00
FF1A        10
FF1B        C3          JMP   FF01
FF1C        01
FF1D        FF
```

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.

Be aware of the memory limitations when you are writing a program. If you do not have one of the PRIMER Upgrade Options, the available memory is from FF01 to FFD3 (211 bytes). If you are using the stack in your program, remember that 2 bytes are used with each PUSH or CALL and when an interrupt occurs. This further reduces the amount of available memory.

After you become more skilled in writing programs you may be able to skip the pseudo-code stage, or even the assembly language stage without making many mistakes. When you write your own program, don't be discouraged if it doesn't work, 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. 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.

If you have access to a PC compatible, you may want to build your own 8085 programming environment which will speed up programming greatly. This environment requires the following: a simple text editor, an 8085 assembler that generates Intel hex files, a terminal emulator with file upload

capability, and finally a PRIMER with a Standard or Deluxe Upgrade option (or at least a serial port) and MOS version 2.5 or greater. In this environment you write your program with the text editor and generate the hex file by assembling the program. Pressing "Func." then "3" will then enable the PRIMER's serial port to receive a hex file. After this, the displays will show "rEC.." indicating that the PRIMER is ready to receive the data. After an ending record is received, (a record in which the fourth pair of digits following the colon is 01), the PRIMER will return to entry mode. If any errors occur while receiving the hex file "Err.." will be displayed followed by a hex number. The bits in this number, after it is converted to binary indicate the following errors:

BIT#	ERROR
0	(not used)
1	checksum error
2	non-hex character encountered
3	escape character encountered
4-7	(not used)

Pressing a key after the error message will put the PRIMER back into entry mode. Receiving a hex file may be aborted any time by resetting the PRIMER or by sending an escape character (1Bh) to the PRIMER's serial port. Sending an escape character will result in an "Err..08".

The program is loaded in the PRIMER's memory by using the terminal emulator's upload (or send) option to send the hex file to the PRIMER. In this environment the program is quickly and accurately loaded, eliminating the tedious process of hand assembling the program and typing it in at the PRIMER's keypad.

If you have the Standard or Deluxe Upgrade option, another programming aid is EMOS which is an operating system in which all interaction is through the serial port and is intended to be used with a data terminal or a terminal emulator running on a PC. It is an enhanced MOS with features such as dump, fill, or move memory, view register contents, read input ports, write output ports, hex and decimal addition and subtraction, program tracing, single stepping, and it has a built in disassembler.

APPENDIX A

Jumper Descriptions and I/O addresses

Jumper Descriptions

JUMPER **DESCRIPTION**

JP1 This allows the selection of one of the following baud rates: 300, 600, 1200, 4800, 9600 and 19,200.

OJ1 This is used to select the sources for the 8085's RST 5.5 and RST 6.5 interrupt inputs. The RST 5.5 interrupt pin is connected to the 8279 interrupt request line when there is a connector between pins 4 and 5, or, if a connector is between pins 3 and 4, it is connected to the 8251 receiver ready line. The RST 6.5 interrupt pin is connected to the 8251 receiver ready line when there is a connector between pins 2 and 3. Putting a connector between pins 1 and 2 connects RST 6.5 to +5v. This jumper can also be used to connect RST 5.5 and RST 6.5 to external interrupt sources. Pin 2 of the jumper is connected to RST 6.5 and pin 4 is connected to RST 5.5.

0J2 This jumper selects the EPROM size. Position 'A' allows an 8 or 16K EPROM to be placed in slot 0 and position 'B' allows a 32K EPROM to be placed in the slot.

0J3 This selects one of the two memory maps which are as follows:

```
POSITION 'A' MEMORY MAP
      SLOT 0            0000 TO 3FFF
      SLOT 1            4000 TO BFFF
      8155 RAM          C000 TO FFFF (only 256 bytes available)

POSITION 'B' MEMORY MAP
      SLOT 0            0000 TO 7FFF
      SLOT 1            8000 TO FFFF
      8155 RAM          (not accessible)
```

I/O Addresses

REFERENCE	I/O ADDRESS	DESCRIPTION
8251 DATA REGISTER	80 H	DATA INPUT/OUTPUT
8251 CONTROL REGISTER	81 H	CONFIGURATION
8155 CONTROL REGISTER	10 H	CONFIGURATION
PORT A	11 H	OUTPUT PORT
PORT B	12 H	INPUT PORT
PORT C	13 H	ANALOG OUTPUT PORT
TIMER LOW	14 H	LOW ORDER TIMING BYTE
TIMER HIGH	15 H	HIGH ORDER TIMING BYTE & CONTROL
EXPANSION I/O	C0 - FF H	EXPANSION CONNECTOR

APPENDIX B

The PRIMER Keypad Description

The PRIMER Trainer has twenty keys for input, with some keys having a second function which can be invoked using the "FUNC." key. Each key produces an audible tone when pressed, providing feedback for the user.

KEY **DESCRIPTION**

0-F Numeric keys. When a numeric key (0-F HEX) is pressed, the numeric value appears at the right side of a data field display and the number that was there before will be shifted left. To correct an entry error just repeat, using the correct number key and the error will be overwritten.

Step This key causes the microprocessor to execute one instruction (single step) at the current program counter address (the value of the P.C. register). Single stepping is a valuable debugging tool that allows the user to examine registers and memory after each instruction executes. The Step key executes the instruction at the program counter address and then returns to the Monitor Operating System and waits for another user command. When you single step a CALL instruction that is calling a subroutine in ROM (such as a service call), the processor will execute the subroutine at full speed and stop at the instruction immediately following the CALL instruction.

Dec. This key decrements the program counter and shows the program counter in the left four displays and the data at the program counter address in the right two displays.

Enter & Inc. When the monitor is in the data entry mode, pressing this key stores the data that is on the two far right displays into the program counter address then increments the program counter and displays the new program counter address in the left four displays and the data at that address in the two far right displays. This key is also used to store new values for the registers, the breakpoint and the stack content (see the section below about second function keys). If the register, breakpoint or stack content information displayed has been changed but you don't want to enter it, then simply press the "Func." key twice and the original information is maintained. This key can also be used to just view data in memory.

Func. This selects the second function of the keys that have two functions. When this key is pressed "Func." appears on the display, and if a key is pressed which has a second function, that function will be performed. This key may be pressed two times in a row to exit the current mode and return to the data entry mode.

(Below are the second functions of the keys with two functions)

B.P. 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 "Enter & Inc." key. When the program reaches the breakpoint the software breakpoint address is automatically reset to 0000. NOTE: Obviously, if the program execution never reaches the breakpoint address, the program will not stop at the breakpoint address. If the breakpoint address points to a byte other than the op code in a two or three byte instruction, 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 op code at that address is executed.

S.C. This displays the 16 bits that are at the top of the stack, or in other words, the data that will be removed from the stack with the next POP, RET or XTHL instruction. The two displays on the far left represent the data at SP + 1 (stack pointer address + 1) and next pair to the right represent the data at SP. The displayed value may be changed by entering the desired hex number using the numeric keys and then pressing the "Enter & Inc." key.

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 in the four displays starting at the left. The digit pair on the left of the four displays show the value of the A register and the pair on the right show the value of the condition Flags. The displayed value may be changed by entering the desired hex number using the numeric keys and then pressing the "Enter & Inc." key. The condition Flags are defined bit by bit as follows:

BIT 0	CARRY FLAG	(if this bit is 1 it indicates a carry)
BIT 1	not used	
BIT 2	PARITY FLAG	(if this bit is 1 it indicates an even number of bits, odd parity)
BIT 3	not used	
BIT 4	AUX. CARRY FLAG	(if this bit is 1 it indicates a carry from bit 3 to bit 4 in the A register)
BIT 5	not used	
BIT 6	ZERO FLAG	(if this bit is 1 it indicates a zero result)
BIT 7	SIGN FLAG	(if this bit is 1 it indicates bit 7 of the A register is a 1)

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

B.C. This key displays the contents of the BC register pair. The BC register pair is displayed as four hex digits. Starting from the left display, the first pair of displays represent the contents of the B register and the second pair 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 "Enter & Inc." key.

D.E. This key displays the contents of the DE register pair. The DE register pair is displayed as four hex digits. Starting from the left display, the first pair of displays represent the contents of the D register and the second pair 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 "Enter & Inc." key.

H.L. This key displays the contents of the HL register pair. The HL register pair is displayed as four hex digits. Starting from the left display, the first pair of displays represent the contents of the H register and the second pair represents 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 "Enter & Inc." key.

S.P. 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 "Enter & Inc." key.

P.C. 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 "Enter & Inc." key.

Run This key causes the microprocessor to execute a program at full speed starting at the current program counter (PC) address. The program will continue to execute until an optional software breakpoint is encountered or until a RST 7 (FF hex) instruction is executed.

1 This key invokes the PRIMER diagnostics. See "Getting Started" for more details.

2 This executes the MOS service selected by the value in the C register, only affecting the registers that pass data back from the service. No instructions need to be stored in memory and the PC register is not affected. For more details see "Using MOS Services".

3 This enables a PRIMER with a Deluxe or Standard Upgrade Option to receive an Intel hex file through its serial port. See the end of the lesson "Writing Your Own Programs" for more information.

4 This invokes the EPROM programmer menu driven interface. This requires the optional EPROM PROGRAMMER BOARD, the standard or deluxe upgrade option and a connection to a dumb terminal or terminal emulation package running on a PC. Documentation on this interface is included when you purchase the EPROM PROGRAMMER BOARD.

APPENDIX C

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

APPENDIX D

PRIMER Schematics

APPENDIX E

Assembly Language Listing
of
Monitor Operating System

APPENDIX F

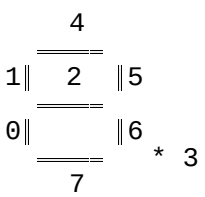
MOS Services

for
MOS version 2.7
and
EMOS version 2.0

SERVICE 0	**DEMO**	Demonstration; this service routine sends a pitch of increasing frequency to the speaker while flashing the output LEDs at an increasing rate.
	INPUT	REGISTER C: 0
	OUTPUT	NONE
SERVICE 1	**CONIN**	Console input; this service waits for a key from the terminal keyboard to be pressed. This service requires the optional serial port.
	INPUT	REGISTER C: 1
	OUTPUT	REGISTER L: ASCII character returned from keyboard.
SERVICE 2	**CONSTAT**	Console input status; this service returns a 0FFH if a key was pressed otherwise a 00H. This service is used in conjunction with a PC/Terminal device connected to the serial port. This service requires the optional serial port.
	INPUT	REGISTER C: 2
	OUTPUT	REGISTER L: Console status.
SERVICE 3	**CONOUT**	Console output; this service outputs a ASCII character to the terminal CRT. This service is used in conjunction with a PC/Terminal device connected to the serial port. This service requires the optional serial port.
	INPUT	REGISTER C: 3 REGISTER E: ASCII character.
	OUTPUT	NONE
SERVICE 4	**PSTRING**	Print string; this service prints to the terminal display the string of ASCII characters starting at the address in the DE register pair until a "$" is encountered. The "$" delimiter is not printed. This service is used in conjunction with a PC/Terminal device connected to the serial port. This service requires the optional serial port.
	INPUT	REGISTER C: 4 REGISTER PAIR DE: Starting character string address.
	OUTPUT	REGISTER PAIR DE: Address of character after the "$" character.
SERVICE 5	**UPRINT**	Unsigned print; this service prints to the terminal display a 16 bit number, in decimal without use of sign. This service is used in conjunction with a PC/Terminal device connected to the serial port. This service requires the optional serial port.
	INPUT	REGISTER C: 5 REGISTER PAIR DE: 16 bit unsigned number to print.
	OUTPUT	NONE
SERVICE 6	**SPRINT**	Signed print; this service prints a 16 bit number to the terminal display, in decimal with use of sign (2's complement). This service is used in conjunction with a PC/Terminal device connected to the serial port. This service requires the optional serial port.
	INPUT	REGISTER C: 6 REGISTER PAIR DE: 16 bit signed number to print.
	OUTPUT	NONE
SERVICE 7	**MULT**	Multiply; this service multiplies two 16 bit numbers. The HL register is returned as the most significant word and the DE register is returned as the least significant word.
	INPUT	REGISTER C: 7 REGISTER PAIR DE: 16 bit multiplicand. REGISTER PAIR HL: 16 bit multiplier.
	OUTPUT	REGISTER PAIR HL: Most significant word of product.
	OUTPUT	REGISTER PAIR DE: Least significant word of product.
SERVICE 8	**DIV**	Unsigned division; this service divides HL by DE (HL/DE). The quotient is returned in HL and the remainder in 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 returns a 6 bit value from the analog to digital convertor.
	INPUT	REGISTER C: 9
	OUTPUT	REGISTER L: 6 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: 0A
	OUTPUT	REGISTER L: DIP switch value.
SERVICE B	**KEYIN**	Read the keypad; this service waits for a key to be pressed and returns the value in the L register.
	INPUT	REGISTER C: 0B
	OUTPUT	REGISTER L: Key value. Keys "0-F" return 00-0F respectively and the "Step", "Func.", "Dec.", and "Enter" keys return 14-17 respectively.
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.
	OUTPUT	NONE
SERVICE D	**HEXPRINT**	Hex print; This service prints to the terminal display, the hex value of the DE register pair. Four hex digits are printed. This service is used in conjunction with a PC/Terminal device connected to the serial port. This service requires the optional serial port.
	INPUT	REGISTER C: 0D REGISTER PAIR DE: 16 bit number to print.
	OUTPUT	NONE
SERVICE E	**DACOUT**	Digital to Analog Converter output; This service routine outputs a 6 bit number in the E register to the Digital to Analog converter.
	INPUT	REGISTER C: 0E REGISTER E: 6 bit value to output to DAC.
	OUTPUT	NONE
SERVICE 10	**PITCH**	Pitch output; This service sends the 14 bit count (the upper two bits are ignored) in the DE register to the speaker timer. 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: 14 bit pitch value.
	OUTPUT	NONE
SERVICE 11	**LEDOUT**	LED Display output; This service routine displays the pattern of LED segments according to the binary value of the E register to the LED digit specified by the D register. The digits are numbered 5 to 0 from right to left.

If a bit is 1 in any of the 8 bits in the E register it will cause the corresponding segment to shine. Below are the segments labeled with their corresponding bit numbers.



INPUT	REGISTER C: 11 REGISTER E: Pattern of segments to display. REGISTER D: Display number (numbered 5 to 0 from left to right).
OUTPUT	NONE

SERVICE 12 **LEDHEX** LED Hexadecimal output; This service routine displays the number in the DE register pair in hex, in the four displays on the left.

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, in the four displays on the left. The maximum decimal value displayed 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: Amount of delay.

OUTPUT NONE

SERVICE 15 **PTBIN** Return the complement of input port B. This is similar to the DIPSWIN service.

INPUT REGISTER C: 15

OUTPUT REGISTER L: Complement of data input to port B

SERVICE 16 **KEYSTAT** Return the status of the keypad in the HL register pair. If a key has been pressed H will be 1 and L will contain the value of the key, otherwise HL will be 0. Keys "0-F" return 00-0F hex respectively and the "Step", "Func.", "Dec.", and "Enter" keys return 14-17 hex respectively.

INPUT REGISTER C:16

OUTPUT REGISTER PAIR HL: Keypad status

SERVICE 17 **DIGOUT** Show the hex digit in E on the display in D. The value for E must be less than 10 hex. The value in D should be a number from 0 to 5, with 0 denoting the rightmost display and 5 the leftmost.

INPUT REGISTER C:17
REGISTER E: Hex digit to show
REGISTER D: Display to put digit

OUTPUT NONE

SERVICE 18 **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 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	RANGE
DE	0.1 SEC				0.01 SEC				00-99
DE+1	10 SEC				SECONDS				00-59
DE+2	0	10 MIN			MINUTES				00-59
DE+3 (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 19 **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

SERVICE 1A **LEDSTR** LED String output; This service routine allows you to individually turn off, or on, any of the segments on one or more of the numeric displays. The HL register pair will point to a section of data for display bit patterns. The bit patterns stored here control the displays the same as the service LEDOUT (service 11). For example, the following program will send a pattern to displays 4, 3 and 2 , leaving the others unchanged.

```
LEDSTR  EQU     1AH
MOS     EQU     1000H           ; address of MOS services
        org     0ff01h
        mvi     c,LEDSTR        ; select LEDSTR service
        mvi     e,3             ; change 3 digits
        mvi     d,4             ; starting at display 4, going right
        lxi     h,bitpat                ; point to table of bit patterns
        call    MOS
loop:   jmp     loop            ; loop here so we can see displays
bitpat:         db      00010000b,00000100b,10000000b

MACHINE LANGUAGE
        ADDRESS         DATA    INSTRUCTION
        FF01            0E      MVI C,1A
        FF02            1A
        FF03            1E      MVI E,3
        FF04            03
        FF05            16      MVI D,4
        FF06            04
        FF07            21      LXI H,FF10
        FF08            10
        FF09            FF
        FF0A            CD      CALL 1000
        FF0B            00
        FF0C            10
        FF0D            C3      JMP FF03
        FF0E            03
        FF0F            FF
        FF10            10      BIT PATTERN DATA
        FF11            04
        FF12            80
```

INPUT REGISTER C: 1A
REGISTER E: Number of displays to change (1 to 6)
REGISTER D: Starting display (numbered 5-0 from left to right).
REGISTER PAIR HL: Address of string of bit pattern data to be shown on the displays
OUTPUT NONE

SERVICE 1B **DDATA** Display the hex byte in E on the DATA/OP displays
INPUT REGISTER C: 1B
OUTPUT NONE

The following five services support the optional EPROM programmer board (E020-8). This board allows you to program EPROMs of a variety of types and voltages. The smallest EPROM supported is a 2764 (8K x 8) and largest EPROM supported is 27512 (64K x 8). Each EPROM type is given an identification number. The 6 EPROM type numbers supported are as follows:

```
TYPE #1     27512 ( 64K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #2     27256 ( 32K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #3     27128 ( 16K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #4     27128 ( 16K x 8 ) EPROM WHICH PROGRAMS AT 21.0 VOLTS.
TYPE #5     2764  (  8K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #6     2764  (  8K x 8 ) EPROM WHICH PROGRAMS AT 21.0 VOLTS.
```

If an illegal type number is passed to one of the following services, the A register will return the value 4, with other registers unaffected.

There are 5 MOS services which support EMAC's EPROM programmer .

SERVICE 1C **READ** Copies a number of bytes starting at an EPROM memory address and stores them in system memory.

SERVICE 1D **VERIFY** VERIFY determines whether the data in the EPROM in the programmer matches a range of data in system memory.

SERVICE 1E **BURN** This writes a number of bytes from system memory to EPROM.

The above commands require that you load the 8085 registers as follows:

H	High order byte of system memory address
L	High order byte of EPROM address (NOTE: The low order bytes of the system memory and EPROM addresses default to 0)
DE	Number of bytes
B	EPROM type (as in the table above)
C	Service number

When VERIFY or BURN, have finished executing with no errors, the A register will be 0 and the other registers (except the flag register) will be unaffected. If A is not 0, an error has occurred. and the registers will be returned with the following information.

BC	Address that error occurred in system memory
H	Value of data at address BC in system memory
DE	Address that error occurred in the EPROM in the programmer
L	Value of data at address DE in the EPROM in the programmer

SERVICE 1F **ERASECHK**

ERASECHK tells whether an EPROM is erased. This command requires you to load the registers as follows:

DE	Starting address from which to examine EPROM. It examines from this address down to 0.
B	EPROM type (as in the table above)
C	Service number 1Fh

The A register will be returned with 0 if the EPROM is erased.

SERVICE 20 **ZAP**

ZAP allows you to put your own application program into a 32k EPROM. BURN also allows this, but ZAP will automatically examine the EPROM to see if it is erased, put in the initialization and MOS services code and verify that the data was written correctly**.** This command requires you to load the registers as follows:

H	High byte of starting address of user program (low byte defaults to 01)
DE	Number of bytes in program (this should be no bigger than 50FEH)
C	Service number 20h

If ZAP has finished executing with no errors, the A register will be 0 and the other registers (except the flag register) will be unaffected. If an error has occured, the A register will indicate the following:

A=1 Error during EPROM write (registers returned with same values as BURN error)
A=2 Error during verification (registers returned with same values as VERIFY error)
A=3 EPROM not erased (registers returned with same values as ERASECHK error)

To make an EPROM-based application:

1) Load and test your program. To make it easier to relocate your program to 2F01h, you should start at address XF01h, where X denotes a hex digit which would result in a valid RAM address. Address FF01h could be used, for example.

2) Rewrite the program to execute at address 2F01h. If you have an assembler, this will be easy. Otherwise you must look for all JMP's and CALL's that need to be changed. Remember that if your program refers to data tables, these should be relocated to EPROM, and all RAM variables should remain at the same address. Also, remember that all registers, except the stack pointer and program counter must be initialized by the user.

3) Put a blank 32k EPROM (type 2 EPROM) in the programmer, aligning pin 1 of the chip to pin 1 of the socket, then latch the socket

4) Load DE with the length of the program, H with the upper byte of the starting address of the program and C with 20h (for service 20). Press "func." then "2" to make MOS service call.

5) Remove power from the trainer and carefully replace its EPROM with the EPROM that was just programmed, making sure pin 1 of the EPROM goes to pin 1 of the socket. 0J2 must be in position B for a 32k EPROM. Place the unused EPROM in a static safe area.

6) Power up your unit and the program should begin to run. If it doesn't work, make sure you have followed these procedures carefully, especially step 2.

To insure desired results:

When inserting or removing EPROMS, make sure the EPROM burner power LED is off.
When performing an ERASECHK, VERIFY or BURN command make sure the EPROM type was correctly entered.

SERVICE 21 **DECPNT** LED Decimal Point output. This service allows you to individually turn on or off the 6 decimal points on the numeric LED display based upon bits 5 to 0 of the D register. In the D register, bit 5 corresponds to the far left digit and bit 0 to the far right digit. A 0 bit will turn off the corresponding decimal point and a 1 will turn it on.

INPUT REGISTER C: 21H
REGISTER D: bit pattern to light or unlight appropriate decimal points.
OUTPUT NONE

SERVICE 22 **BIN2BCD** Binary to Binary Coded Decimal. This service converts the 16 bit number in register pair DE to binary coded decimal with the low nibble in register E being the least significant digit

INPUT REGISTER C: 22H
REGISTER PAIR DE: The number to be converted to BCD
OUTPUT REGISTER PAIR DE: The BCD number.

SERVICE 23 **BCD2BIN** Binary Coded Decimal to Binary. This service converts a 4 digit BCD number in register pair DE to binary.

INPUT REGISTER C: 23H
REGISTER PAIR DE: 4 digit BCD number
OUTPUT REGISTER PAIR DE: converted number in binary

SERVICE 24 **KPINPUT** Keypad Input. This service allows the user to input from the keypad a hexadecimal number with up to 4 digits. The number will be returned in the DE register pair. The numbers can be displayed on the LED display by loading D with 1 before the service is called. The LED display will be turned off if D = 0. The service will not return until the "ENT" key is pressed. Once ENTER is pressed, the last 4 digits entered will be loaded in DE with the last digit being the Least significant. If 4 digits are not entered, a 0 will be assumed for the leading digits.

INPUT REGISTER C: 24H
REGISTER D: 1 for LED display on, 0 for LED display off.

OUTPUT REGISTER PAIR DE: 4 digit number from keypad