PRIMER TRAINER

LAB MANUAL

MANUAL REVISION 1.1
September 2002

Table of Contents

LAB #1
INTRODUCTION TO COMPUTERS

INTRODUCTION

A computer is an electronic device that follows instructions in order to do useful work. There are many different kinds of computers. The computers used by the IRS for instance, require a great deal of computing power in order to process tax returns for the millions of people in the United States. These computers are quite large and cost millions of dollars. The desktop computer on the other hand is not as powerful as the IRS computer, but then again it is quit smaller and doesn't cost near as much. The desktop computers size is made possible by the use of microprocessors. Microprocessors are small electronic chips (integrated circuits) that contain at least 75% of the computing power of the digital computer. Besides reducing the size of a computer, microprocessors are also responsible for reducing their cost. The small size and low cost of microprocessors has facilitated their use in almost every electronic device from hospital equipment to military weapons, from toys to home appliances.

All computers regardless of their size and power require some way to input and output data in order to be useful. A desktop computer for example receives input commands from the keyboard and displays output on the computer monitor. A modern microwave oven which also contains a microprocessor receives input commands from the keypad and the output is seen on the calculator type display. Their are however, other inputs and outputs not so easily identified on the microwave oven. There is an input switch on the door that turns the light on and the oven off. There is an output control line that turns the oven on and off in response to the various inputs of the system.

Memory is another required component of a computer. Memory is used to store computer instructions (programs) and data. Again in the example of the microwave oven, the program is what controls the oven and allows you to enter the amount of time (the data) to cook your food.

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.

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 should 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. The microprocessor would perform these instructions 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.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

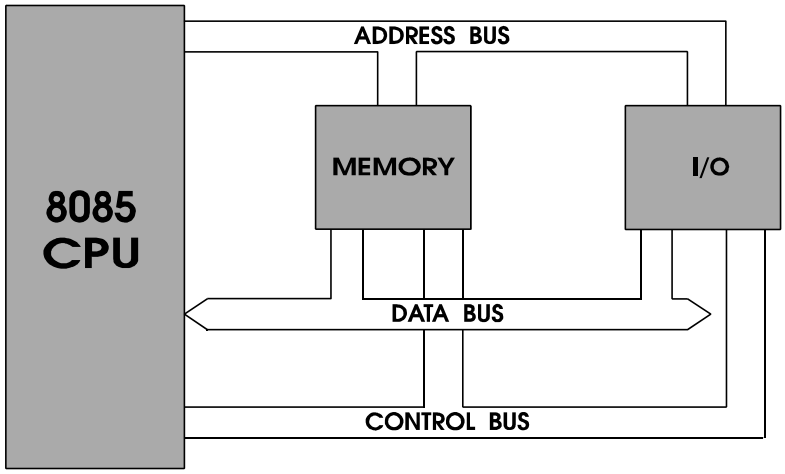
* To understand the building blocks of the computer.
* To define the various types of memory.
* To define Input/Output (I/O) as it is used on the 8085 microprocessor.
* To define the internal structure of the 8085 microprocessor.

PROCEDURE

The 8085 microprocessor with the addition of a few components is a fully functioning computer system. The microprocessor is the brains of the computer, and it executes the instructions that make up a computer program. These instructions are held in memory where the microprocessor accesses them as required. Each instruction is placed in a different location of memory with its own unique address, much the same as each house on a street has its own address. The address is placed on the address bus where it is sent to the memory. The memory then places the instruction at that address on the data bus where it is sent to the microprocessor. This is basically how a computer executes instructions. The address bus always carries the address from the microprocessor to the memory or I/O device. The data bus on the other hand is used to send information both from and to the microprocessor and therefore is called a bidirectional bus. Another bus present in the microprocessor system is the control bus. This bus basically consists of the control lines which keep the microprocessor operational. For example whether to read or write from memory, do an I/O operation, wait till the memory is ready, etc.

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).
ROM memory retains its contents when power is removed. Memory that retains its contents when power is removed is called nonvolatile memory. RAM memory on the other hand loses all of its contents when power is removed and is referred to as volatile memory. Adding a battery to a RAM chip can change it from a volatile memory to a nonvolatile memory.



MICROCOMPUTER BLOCK DIAGRAM
FIG. 1.0

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 since it is read only memory. The RAM and ROM chips in the PRIMER Trainer have eight data 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 data pins on 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

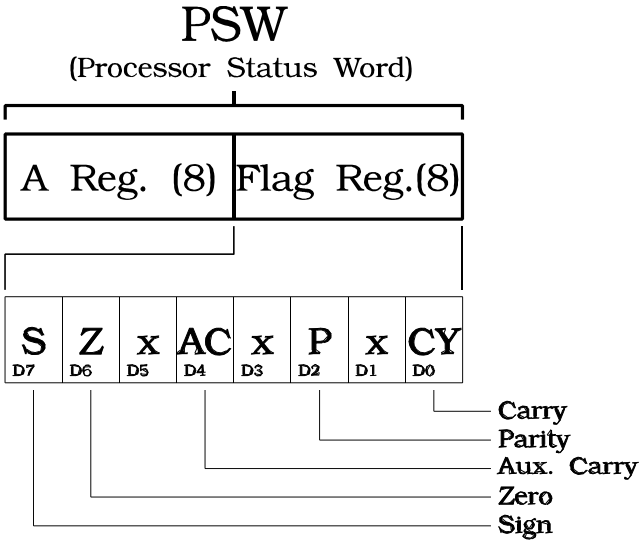
The 8085 microprocessor can also send data to and receive data from chips other than the RAM or ROM. When the microprocessor wants to perform input or output it disables the RAM and ROM chips and sends an input/output (I/O) address to the address bus. The I/O address is only 8 bits but it appears on 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. Figure 1.0 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 similar 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 A register in addition to being used as a general purpose register has some unique abilities. The A register is always used for all I/O instructions, logic instructions, and most arithmetic instructions. For example if you subtract two numbers the difference would be located in the A register.

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. Figure 1.1 shows the PSW with a diagram of the individual bits of the flag register.



x : Undefined

FIG. 1.1

There are also registers that are dedicated to special purposes. These 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 chapter 15 of The Intelligent Microcomputer by Goody.

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. The flags are covered in later labs.

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)

8085 REGISTERS
FIG. 1.2

QUESTIONS 1.0

1. Define microprocessor.

2. Why does a computer need I/O?

3. What type of memory can only be read from?

4. What type of memory loses the information contained in it when power is removed?

5. Is the address bus unidirectional or bidirectional?

6. Is the data bus unidirectional or bidirectional?

7. How many memory locations can the 8085 access?

8. How many I/O devices can the 8085 access?

9. What is another name for the A register?

10. What makes the A register different from the other general purpose registers?

11. How many bits are contained in the A register?

12. How many bits are contained in a register pair?

13. How many bits are contained in the stack pointer?

14. How many different flags are available in the 8085?

15. What does PSW stand for?

16. When referring to the B/C register pair which register holds the 8 bits of low order data?

LAB #2
COMPUTER NUMBER SYSTEMS

INTRODUCTION

In order to be able to program a computer, its necessary to be able to speak in a form that the computer can understand. Computers being digital devices (using two states) use the binary number system as their native tongue. The hexadecimal number system, like binary is a power of two number system also. Each hexadecimal digit is represented by 16 different numeric symbols verses 2 for a binary digit. Hexadecimal therefore is more efficient then using binary when listing computer codes and since hexadecimal is a power of 2 number system conversion to the binary number system is a simple procedure.
Since we humans use the decimal number system, converting from decimal to binary (the number system computers understand) is necessary in order to program computers at a low level.

OBJECTIVES

* Define a bit.
* Define a byte.
* Define a word.
* Define binary code.
* Define hexadecimal code.
* Convert a binary number to decimal.
* Convert a binary number to hexadecimal.
* Convert a hexadecimal number to binary.
* Define least significant and most significant.

PROCEDURE

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

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

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

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

As you know, in decimal numbers, each position of a digit has a different weight. For example in the decimal number 1732: (The digit numbers are read from right to left, 0 to 3)

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

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

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

If one of the bit positions had been a zero, that bit position's weight would not have been added to the sum.

To convert a decimal number to another base repeatedly divide the number by the base until the quotient equals zero. For example to find the binary (base 2) equivalent of the decimal number 58 we proceed as follows:

```
58 / 2 = 29 with a remainder of    0 (LSB)
29 / 2 = 14 with a remainder of    1
14 / 2 = 7  with a remainder of    0
07 / 2 = 3  with a remainder of    1
03 / 2 = 1  with a remainder of    1
01 / 2 = 0  with a remainder of    1 (MSB)
```

The binary equivalent of the decimal number 58 is therefore: 111010. The binary number 111010 can be varified by converting it back to decimal. Conversion from decimal to hexidecimal can be accomplished by dividing the decimal number by 16 instead instead of 2. The remainder when dividing by 16 will range from 0 to 15, or 0 to F in hex (see below). 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 more efficient then binary and it is very easy to convert numbers from binary to hexadecimal. For example, to convert the following binary number to hexadecimal:

```
1011011011010 11
```

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 (MSB) and the bit on the right end is the least significant bit (LSB). In the hex word 01FF the left two digits are the most significant byte and the two right digits are the least significant byte.

QUESTIONS 2.0

1. What two numbers represents a binary digit?
2. Give an example of non-numeric names for the two numbers of question 1.
3. How many bits are in a byte?
4. How many bits in a word?
5. Give the decimal equivalent of the binary number 10010110.
6. Give the hex equivalent of the binary number 10010110.
7. Give the binary equivalent of the decimal number 115.
8. Give the hex equivalent of the decimal number 307.
9. Give the binary equivalent of the hex number D6.
10. Explain what is meant when hex is said to be more efficient for the user than binary.

LAB #3
COMPUTER LOGIC OPERATIONS

INTRODUCTION

Computers being digital systems, are programmed using digital logic instructions. The hardware of a computer consists of logic gates which in conjunction with the software determines what the computer does. A special group of instructions called logic instructions, lets computer programs make decisions much the same as the logic gates of a computer make decisions needed for the computer to operate. As a matter of fact a computer can simulate in software, the operation of hardware logic gates.

If a computer program allows a light to come on if one of two switches is in the ON position, it is using digital logic to accomplish the operation. Other possibilities include the light only being turned on if both the switches are in the ON position. Logic instructions also allow us to isolate a particular bit position or positions. Using logic instructions a bit can be turned on, turned off, or turned to the opposite state, while not disturbing other bit positions. If each bit position is visualized as being connected to a light, a alarm, a motor, etc. with a 1 turning the particular device on and a 0 turning it off, it becomes apparent that being able to control each bit position individually is an important feature.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* Define the AND operation.
* Define the OR operation.
* Define the EXCLUSIVE OR (XOR) operation.
* Define the NOT operation.
* Define mask.

PROCEDURE

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, inverting, or toggling the bit.

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

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

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

The following shows the way logical operations work with bytes:

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

If we envision each bit position as being connected to a different light in a house, using logic operations we can control each light individually or in groups. The trick is to control a particular light without knowing the current state of each light and not upsetting the other lights in the house. For example to turn on light #2, a one must be in bit position two. To turn off light #2, a zero must be in bit position two. We would like to turn light #2 on and off without disturbing any of the other seven lights in the house. For the following example the arbitrarily chosen byte below represents the current state of eight different lights in a house.

```
01001010
```

This byte tells us that light #6 is on, light #3 is on, light #1 is on, and all the other lights are off. To turn light #2 on we use a mask value. A mask is a value used to manipulate different bit positions. This value when used with the appropriate logic operation performs the function we desire. The mask value chosen for the example is:

```
00000100
```

This mask value when ORed with the current state of the lights will force light #2 on.

```
      01001010
OR    00000100
      01001110
```

Notice that all the lights that were on are still on, and all the lights that were off are still off, with the exception of light #2. If later we decide to turn light #2 off, we use a different mask value and a different logic operation. Finding the correct mask value for turning the light off is not as simple as it was for turning the light on. First we must take the complement using the NOT operation, of the mask we used to turn the light on, then we AND this value with current state of the lights in order to turn the light off. This translates into simply placing a zero in any bit positions to be turned off and a one in all other bit positions.

```
NOT   00000100
      11111011
```

The complementing of the mask used to turn the light on, gives us the new mask value needed to turn the light off. This new mask value has a zero in each bit position to be turned off, and a one in all other bit positions.

```
      01001110
AND   11111011
      01001010
```

ANDing the new mask value with current state of the lights turns light #2 off, while not affecting any of the other lights. The current state of the lights is now the same as it was before we turned light #2 on. By adding more ones to the mask value, more then one light can be turned on, and by adding more zeros, more then one light can be turned off. Remember to use the OR operator to turn things on and the AND operator to turn things off.

To toggle or compliment a bit position the exclusive OR (XOR) logic operator is used. This allows a bit to be switched from a high to a low or from a low to a high without knowing the current state of the bit position. The mask value should contain a 1 in each bit position that requires toggling and 0's in all other bit positions. Like the other logic operations this only affects the selected bit positions. All positions that contain a 0 will retain their original values. For example if we wish to toggle bits 0 and 6 of the result of the above AND operation, the mask value would be 01000001 and yield a result of:

```
      01001010
XOR   01000001
      00001011
```

Notice that only bit positions 1 and 6 have changed, all other bit positions have retained the value they had before the XOR operation took place.

QUESTIONS 3.0

1. Complete the following logic operations:

```
      11100110          10111011          00110110
AND   00110101    OR    01101001    XOR   10011010    NOT   11101011
```

2. Define the term mask.

3. Give the 8 bit mask value and the logic operation to turn on bit #3 and bit #5 without disturbing any of the other bits.

4. Give the 8 bit mask value and the logic operation to turn off bit #1 and bit #4 without disturbing any of the other bits.

5. Give the 8 bit mask value using the XOR operation to complement all 8 bits of the following byte:

```
          10011100
XOR       ________
          01100011
```

6. Give the 8 bit mask value and the logic operation to toggle bit #7 and bit #6 without disturbing any of the other bits.

7. If a value is ANDed with itself what would be the result?

8. If a value is ORed with itself what would be the result?

9. If a value is exclusive ORed with itself what would be the result?

10. IF a value is exclusive NORed (exclusive NOT ORed) with itself what would be the result?

11. Define the term toggle.

LAB #4
COMPUTER LANGUAGES

INTRODUCTION

In order for computers to be useful they must be told what to do. When a computer is told how to perform a given task it is said to be programmed. The process of programming a computer requires a computer language. A computer language, like a spoken language, provides a means for us to tell the computer what to do. You see computers do not yet speak fluent English. As with spoken languages, there are also many computer languages. Each computer language has its advantages and disadvantages depending on the type of program that is being written. All computer languages however, end up producing instructions that the particular computer understands. These instructions which the computer understands is referred to as the computers instruction set (the set of instructions understood by the computer). Not all computers have the same instruction set, in fact most computers have different instruction sets. A compiler or assembler, is used to translate the computer language statements of a computer program into the machine code (also referred to as object code) consisting of instructions from the computers instruction set. Compilers and assemblers are actually computer programs themselves, that are used to translate other computer programs. Once translated the computer can then execute the program.

Computer languages can be broken down into two categories, high level languages and low level languages. High level languages are more English like and much easier to program in then low level languages. Languages such as BASIC, PASCAL, and C are examples of high level languages. Low level languages are more cryptic and difficult to program in, but produce programs with less instructions, that run much faster then high level languages. Examples of low level languages are machine language and assembly language. Low level languages have a one to one correspondence with the computer's instructions. Each assembly language statement translates into one computer instruction, whereas with a high level language, one statement might translate into several hundred computer instructions.

When programming in machine language, the programmer acts as the translator. This is referred to as hand assembling because you are basically doing by hand what an assembler program does. The appropriate instruction is found, then the machine code is looked up and entered into the computer. A machine code is just a number that denotes a particular computer instruction specified by the manufacturer. On older computers the machine code was actually entered and displayed in binary using switches and lights, on computers of today this is rarely the case. Today, when machine code is used, it is usually entered and displayed in hex, with either the hardware or the software doing the conversion to or from binary. Remember computers at the lowest level (the machine level) must execute binary codes.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* Define computer programming.
* Define computer language.
* Understand the difference between low level and high level languages.
* Define machine language.
* Define assembly language.
* Know the five different 8085 instruction categories.
* Define op code.
* Define mnemonics.
* Understand flowcharting concepts.

PROCEDURE

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 in appendix B, 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 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 often referred to as operands. 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 consecutive address following the instruction byte, and if the instruction requires two bytes the 8085 will get them from the next two consecutive addresses following the instruction byte. With 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.

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. Assembly Language programs are usally placed in a file known as the Source file. This file is read in by the assembler and converted to machine language. Two special files are created by the assembler the Object file and the Listing file. The Object file basically contains the machine language produced by the assembler. The Listing file is a text file that is a combination of the Source and Object files containing both the original assembly language with the corresponding machine language. Assembly language programs cannot be understood by the 8085 microprocessor 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.

Example 4.0 lists an arbitrary 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.

EXAMPLE 4.0

```
   LABEL    MNEMONIC    OPERAND         COMMENT
1  dips     equ         12h           ; port address for the dip switches
2  leds     equ         11            ; port address of leds
3
4           org         ff01h         ; starting address of the program
5  loop:    in          dips          ; load A register with the dip
6                                     ; switch values
7           cma                       ; compliment A register to convert to
8                                     ; positive logic
9           out         leds          ; output the DIP switch contents to leds
10          cpi         00000000b     ; compare DIP switch to all zeros
11          jnz         loop          ; if DIP switch equals all zeros stop
12                                    ; else loop and try again
13          rst         7             ; stop program and return to MOS
14          end                       ; end of assembly language program
```

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 can be assigned using the EQU directive. Directives are used to instruct the assembler and are not part of the 8085 instruction set, nor are they ever executed by the program being assembled. Directives do not produce any machine code. The symbol on the left of the EQU directive 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 "leds" is assigned the value 11 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 directive tells the assembler the starting address in memory of the instructions that follow it. Line 4 shows that the program is to be assembled starting at address FF01h.

When a symbol is in the label field of a line, and there is a mnemonic and not the directive EQU for that line, its value will be made the memory address of the opcode represented by the mnemonic. In line 5 of the program above, "loop" is assigned the value of the memory address containing the opcode of the "in" mnemonic. Line 11 uses the value of the label "loop" to produce the machine language for the "jnz" 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.

A directive not included in the above program is the DS (Define Storage) directive. The DS directive 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 locations allocated using the DS directive will generally change (like variables) as the program executes and accesses these locations. These locations must be initialized by the program at run time.

Another directive not included in the above program is the DB (Define Byte) directive. The assembler directive DB takes the data that follows the mnemonic and stores the 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. A DW (Define Word) directive is provided for 16 bit data. The locations allocated by the DB or DW directives generally will not change (like constants) as the program executes and accesses these locations. These locations are usually initialized at assemble time.

The directive END tells the assembler that this is the end of the program. The directives END, EQU, ORG, DB and DS are all called "pseudo-ops", which means they are not translated into machine language directly. These pseudo-ops are used by the assembler to aid in assembling the source program (the original assembly language program). Lines 5 through 13 contain mnemonics that actually correlate directly to machine language.

Below is the corresponding machine code in hex if the above program was assembled.

EXAMPLE 4.1

```
FF01    DB
FF02    12
FF03    2F
FF04    D3
FF05    11
FF06    FE
FFF7    00
FF08    C2
FF09    01
FF0A    FF
FF0B    FF
```

This code could now be entered into the trainer and executed.

Flowcharting

Before a program is actually written, a flowchart is usually constructed detailing the program algorithm (solution to the programming problem). A flowchart is a graphical representation of the operation of the program. A flowchart is composed of several different types of graphical blocks connected with lines showing program flow. The four most important of these blocks is shown in Figure 4.0.

An oval shaped block is used to indicate the start or the end of a flowchart segment. A parallelogram is used to indicate input or output, for instance reading from the DIP switches or writing to the LEDs. A diamond shape block is used when

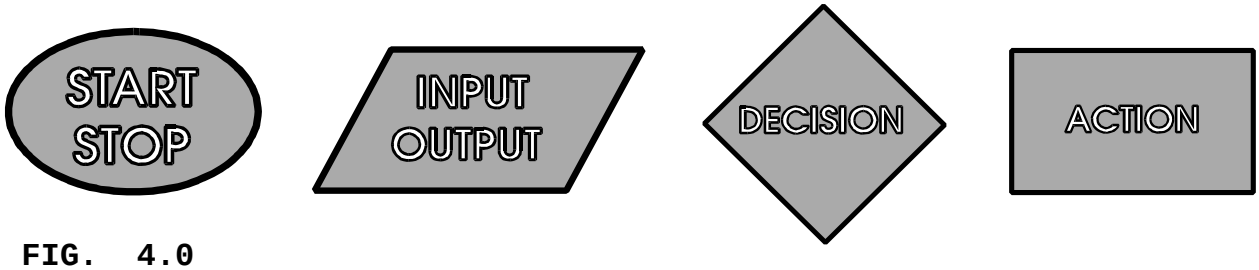


FIG. 4.0

making a decision, such as is the value greater than the setpoint yes or no. A rectangular block is used to indicate action, for example turn light on, sound alarm, etc. There are many other specialized symbols, but they are not required within this lab book.

When programming in assembly language, particularly for programs of 50 or more lines (statements/instructions), flowcharting and good documentation is essential. Assembly language programs are inherently difficult to understand, especially if someone else wrote the program. Flowcharts and good commenting practices go a long way in making a program easily readable and maintainable.

Flowcharts are not only used to aid in making a program more understandable, but also in the development of the program itself. A programmer typically will start with a high level flowchart depicting the operation of the program. With a high level flow chart, each block of the flowchart can represent hundreds of assembly language statements. The high level flowchart is an overview of the system. Each block of the high level flowchart is then broken down into a intermediate or low level flowchart. With a low level flowchart, each block may translate into one to five assembly language instructions. From the low level flowchart, the programmer writes the program. This high to low level process is known as top-down development. For simple programs a low level flowchart may be all that is necessary or for very simple programs no flowchart at all may be necessary.

Listed below is a high level BASIC program to count down from 10 and to 0 and exit, and the same program written in assembly language and machine language. Following these program examples is the corresponding low level flowchart. Note the flowchart is the same for all three examples.

BASIC PROGRAM EXAMPLE 4.2

```
LINE #.     STATEMENT                COMMENT
1           LET COUNT=10             SET COUNT TO 10
2           LET COUNT=COUNT-1        DECREMENT COUNT BY 1
3           IF COUNT<>0 THEN GOTO 2  GOTO LINE #2 IF COUNT IS NOT 0
4           STOP                     STOP PROGRAM EXECUTION
```

ASSEMBLY LANGUAGE EXAMPLE 4.3

```
LABEL       OPCODE      OPERAND      COMMENT
            ORG         FF01H        START PROGRAM AT ADDRESS FF01 HEX
START:      MVI         A,10         SET REGISTER A TO 10
LOOP:       DCR         A            DECREMENT A BY 1
            JNZ         LOOP         JUMP TO LOOP IF A IS NOT 0
            HLT                      HALT PROGRAM EXECUTION
```

MACHINE LANGUAGE EXAMPLE 4.4

ADDRESS	CONTENTS	INSTRUCTION	COMMENT
FF01	3E	MVI A,10	SET REGISTER A TO 10
FF02	0A		SECOND BYTE OF INSTRUCTION
FF03	3D	DCR A	DECREMENT A BY 1
FF04	C2	JNZ FF03	JUMP TO FF03 IF A IS NOT 0
FF05	03		LOW ORDER ADDRESS
FF06	FF		HIGH ORDER ADDRESS
FF07	76	HLT	HALT PROGRAM EXECUTION

The low level flowcharting example below for simplicity sake does not contain any input or output. As you can see the flowchart's graphical representation eases program understanding.

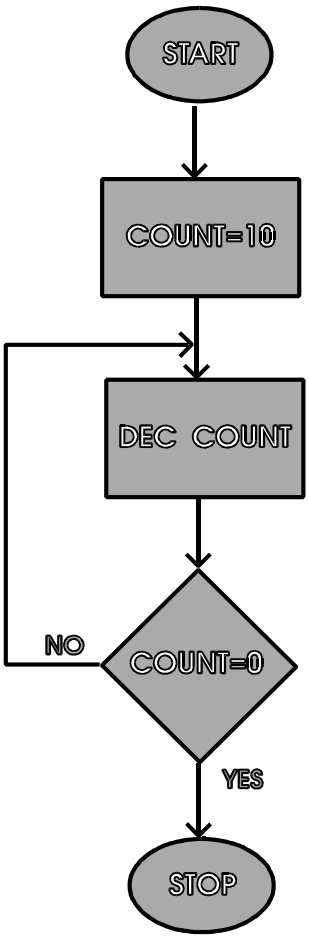


FIG. 4.1

This program example is very simple in nature, allowing an almost one-to-one correspondence between BASIC and Assembly language, in most cases this is not so. The simplicity of the example program requires only a low level flowchart to completely represent the program.

EXERCISE 4.0

1. Hand assemble the following mnemonic instructions into machine code. A table in appendix B contains each instruction with its corresponding op code. Some instructions are 1, 2, or 3 bytes in length depending on the operands of the instruction.

```
                ORG         FF01
FF01  _______   RAR

FF02  _______   MOV         A,B

FF03  _______   ADD         C

FF04  _______   MVI         D,0A3H

FF05  _______

FF06  _______   LHLD        1234H

FF07  _______

FF08  _______

FF09  _______   ANI         101B

FF0A  _______

FF0B  _______   RST         7
```

QUESTIONS 4.0

1. What is the importance of a computer language?

2. Why would a program be written in assembler instead of BASIC?

3. Why would a program be written in C instead of assembler?

4. What language is programmed using strictly numbers?

5. When entering machine code into memory why is it better if the code is in hexadecimal form?

6. Each computer (machine) instruction is referenced by a numeric code called an _____.

7. What is the minimum number of bytes of a 8085 computer instruction?

8. What is the maximum number of bytes of a 8085 computer instruction?

9. The first byte of a computer instruction is always the _____.

10. Subsequent bytes of a computer instruction are always the _____.

11. What is another name for assembler directives?

12. What is the purpose of assembler directives?

13. All comments in an assembly language program must begin with which character?

14. Symbolic addresses used in assembly language programs are referred to as _____.

15. In the machine code example 4.1, why is the first instruction at address FF01.

17. In the machine code example 4.1, the contents of addresses FF01 and FF02 were translated from what line in the corresponding assembler program of example 4.0?

18. The file that contains the original assembly language program is called the _____ file.

19. The file created by the assembler which contains only the machine code is called the _____ file.

20. The file created by the assembler which contains both the assembly code and the machine code is called the _____ file.

21. Why is it important to construct a flowchart before starting to write a program?

22. Modify the flowchart of Fig. 4.1 in order to have the program count to 10 and repeat indefinitely.

LAB #5
USING THE MONITOR OPERATING SYSTEM

INTRODUCTION

A computer operating system (OS) is a computer program that oversees the operation of the computer. It is responsible for initializing the system and controls the entry and display of data. The OS also provides services, that user written software can take advantage of, such as displaying numbers. If there were no OS services, the programmer would have to write the software necessary to perform these tasks.

A monitor assists the programmer by allowing the viewing and alteration of memory and the ability to run programs. Other monitor functions can include single stepping of programs, examining and altering the contents of CPU registers, and setting breakpoints. Combining the monitor and the operating system provides an ideal training environment, where programs can be entered, run and tested. The Monitor Operating System (MOS) provides all of the features mentioned above and more.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* Define operating system.
* Define monitor.
* View and change the contents of memory.
* View and change the register contents.
* Run a program.

PROCEDURE

The Monitor Operating System (MOS) is a powerful software program that provides the user with the tools to enter and edit programs as well as run, test, and debug machine language programs. MOS uses 6 numeric LEDs for display output. This type of display is similar to the displays used in calculators, but allows letters in addition to numbers.

The display at power-up reset, contains the default PC address (FF01H) in the left four digits of the display. This is what is referred to as the ADDRESS FIELD. In the right two digits are the contents of the displayed address. This field is known as the DATA/OP FIELD. The MOS uses 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.

GETTING STARTED

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.

To begin diagnosis:

Press the "Func." key then "1". When the diagnosis begins, "....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. Pressing a key following this error message will cause the diagnosis to continue. The RAM diagnostics should be completed in around 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.

When the RAM check is finished and the PRIMER is connected to a terminal through the serial communications option, 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.

Note the following:

M The PRIMER is rugged but it can be damaged. Please handle it with reasonable care. The underside of the board is exposed, so be careful not to place it on any object that could cause damage.

M The the voltage regulator gets hot so avoid touching it.

M There are DIP (dual inline package) switches on the PRIMER that are required in several labs. Please do not use a lead pencil or ink pen for this purpose because the lead or ink can get into the switch.

In the following sections of the lab you will be introduced to the monitor operating system. Feel free to at any time refer back to the key descriptions until they become familiar to you.

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 <u>not</u> 55 as was typed before. This is because hex numbers that are typed aren't stored until the "enter" key is pressed. This is a useful feature when you have typed a number and decide you do not want it to be stored in memory or wish not to change the original value.

LOADING A PROGRAM INTO MEMORY

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

PROGRAM EXAMPLE 5.0

```
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.

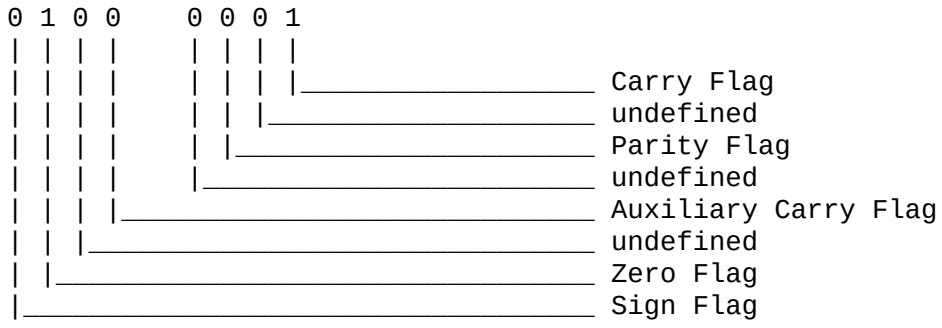
If you want to change the value of the A register, press the "func." key and then the "A.F." key and as a result the displays show "0000 A.F.". To change the value of the A register to 1F hex without changing the flag register, press the following keys in order: "1","F","0","0","enter". After pressing 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 lab #2. For example, to find out what the current flag values are, convert 41 hex to binary.

```
   HEX
  4     1

  BINARY
 0100   0001
```



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.

RUNNING A PROGRAM

Before running a program, it is very important that the PC be placed at the opcode of first instruction to be executed (in this case FF01). The PC can be placed at this address by using the DEC and ENT keys or by changing the PC register.

Once the PC is correctly positioned at the opcode of the first instruction, 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 program details are not important at this time and will be covered later in other labs.

EXERCISE 5.0

1. Enter the following hex bytes into the memory locations shown below and verify the memory contents for accuracy.

```
FF01  11
FF02  04
FF03  03
FF04  2E
FF05  06
FF06  63
FF07  24
FF08  3E
FF09  01
FF0A  47
FF0B  07
FF0C  4F
FF0D  97
FF0E  FF
```

2. The addresses FF01 to FF0E now contains the machine code of a short program. Run this program starting at address FF01 (the first op code of the program) and examine the register contents after its execution. To run the program change the contents of the PC to FF01, then press the RUN key. Fill in the register and flag contents below.

```
PC_____ B____ C____ D____ E____ H____ L____ A____ Sf_ Zf_ ACf_ Pf_ Cf_
```

3. When programming in machine language, the programmer translates each mnemonic instruction by hand, into machine code. This is referred to as hand assembling the program (see Lab #4). When programming in assembly language this translation is accomplished by the assembler. The translating back from machine code into mnemonic instructions is referred to as disassembling the program. Disassemble by hand the program of exercise question 5.0.1 starting at address FF01 and list the address and the mnemonic of each opcode below. The first address and mnemonic is given. A table in appendix B lists each instruction and its corresponding op code. There are 10 op codes (instructions) in the program. Remember that each instruction can have 1, 2, or 3 bytes depending on the operands.

```
1.    FF01  LXI   D,0304H

2.    _____ ________________

3.    _____ ________________

4.    _____ ________________

5.    _____ ________________

6.    _____ ________________

7.    _____ ________________

8.    _____ ________________

9.    _____ ________________

10.   _____ ________________
```

QUESTIONS 5.0

1. When the Trainer is powered up or reset what are the values of the registers and the flags?

   PC_____ B_____ C_____ D_____ E_____ H_____ L_____ A_____ Sf___ Zf___ ACf___ Pf___ Cf___

2. MOS does the initialization of all the registers mentioned in question #1. Which of these registers is the only register actually initialized by the 8085 microprocessor prior to the MOS initialization and to what value?

3. When displaying the Accumulator and the Flags which register is displayed on the right two digits?

4. In order to execute a program that is in memory, what first must be done?

5. Why does the ENT/INC key have two names ENT and INC?

6. Memory locations below address 8000h cannot be changed, why?

7. What type of memory must be used to enter or change programs?

8. Why is it important for the Monitor Operating System to be in nonvolatile memory?

9. List a practical use for a disassembler.

10. What software is responsible for producing the audible beep when the Trainer is reset?

11. Explain the relationship between the DIP switch ON/OFF positions and the output ON/OFF status of the LEDs, when running the PROGRAM EXAMPLE 5.0.

12. Explain the function of the reset button.

LAB #6
SOFTWARE SINGLE STEPPING AND BREAKPOINTS

INTRODUCTION

Being able to run a program at the fastest possible speed is important to maximize throughput. On an IBM PC this translates to less time the user has to wait for a response. Sometimes during program development however, it's beneficial to slow the processor down so as to see what is going on internally. This process allows the user to examine the registers and memory locations after each instruction. If this process is done an instruction at a time, where the user has control after the execution of each instruction, then it is referred to as single stepping. If this process is performed on a number of instructions, where the user has control after the execution of that number of instructions, then it is referred to as tracing. A program being traced still displays the registers and possibly certain memory locations after each instruction executes. Single stepping and program tracing are important tools in debugging and developing programs.

There are two forms of single stepping, software single stepping and hardware single stepping. Software single stepping as the name implies is performed entirely in software. Embedded within MOS is the program that performs the single stepping function. This program executes one instruction and saves the contents of the registers, then returns to MOS allowing the user to enter another command. The user can then view the contents of the registers if need be.

The software single stepper is most useful for development and debugging of software.

Another useful debug and development tool is the breakpoint. The breakpoint, like the single step, comes in two varieties, software and hardware. The software breakpoint is accomplished through software by inserting a special software interrupt code in place of the instruction, at the address where you want to program to break. When this special instruction is executed, the program jumps back to MOS, which replaces the original instruction and returns control to the user. The breakpoint and single step functions when used together provide a powerful debugging aid. For example, if the user suspects a bug in the second half of a program, the user can set a breakpoint over the first half of the program, and begin single stepping at the suspect code.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* To become familiar with the software single stepping process.
* To become familiar with the software breakpoint process.
* To become familiar with the following 8085 instructions.

IN <byte> opcode = 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. This instruction is 2 bytes in length.

CMA opcode = 2F

The contents of the accumulator are complemented (zero bits become 1, one bits become 0; 1's complement or NOT function). This instruction is 1 byte in length.

OUT <byte> opcode = D3

Send the data in the A register to the output port specified by the byte following the instruction. No flags are affected. This instruction is 2 bytes in length.

JMP <addr> opcode = C3

Load the program counter (PC) with the address contained in the two bytes following the instruction. The first byte following the opcode is the least significant byte of the address, the second byte is the most significant byte of the address. No flags are affected. This instruction is 3 bytes in length.

PROCEDURE

Below is a flow chart of a simple program that reads the DIP switches (the 8 position DIP switch labeled MANUAL DIGITAL INPUTS), complements the value, and outputs this value to the output port's LEDs. The value of the DIP switch requires complementing due to its use of negative logic (on produces a binary 0 and off, a binary 1). In fact it is very common for digital I/O ports to use negative logic, due to the fact that more current can be sinked than sourced with these type of devices. The digital output port LEDs, for example, require a low to turn on. When using the DIP switches, rather than having to think in negative logic, complement the DIP switch value immediately after inputting the value. You can then treat the DIP switch value as a positive logic value.

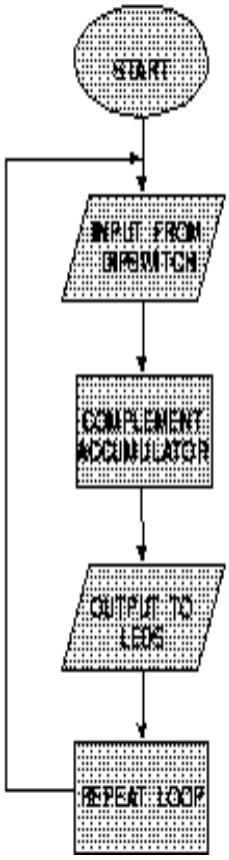


FIG. 6.0

ASSEMBLER PROGRAM EXAMPLE 6.0

```
leds        equ   11h
dips        equ   12h

            org   0ff01h
loop:       in    dips  ; load A register with DIP switch values
            cma         ; complement A register
            out   leds  ; output A register to LEDs
            jmp   loop  ; jump to loop
            end
```

MACHINE LANGUAGE PROGRAM EXAMPLE 6.1

ADDRESS	DATA	INSTRUCTION
FF01	DB	IN 12
FF02	12	I/O ADDRESS
FF03	2F	CMA
FF04	D3	OUT 11
FF05	11	I/O ADDRESS
FF06	C3	JMP FF01
FF07	01	LOW ADDRESS
FF08	FF	HIGH ADDRESS

Notice in the machine language program EXAMPLE 6.1 that the JMP instruction occupies three memory locations. The first byte of the instruction is the opcode. The next two bytes of the instruction comprise the memory address where it will jump. Since all memory addresses in the 8085 are 16 bits and each memory location holds one byte, two memory locations are required to hold a memory address. When a 16 bit address or quantity is stored in memory, the low part of the address or number is stored in the lower address and the high part of the address or number is stored in the higher address. I/O addresses on the other hand, only require an 8 bit address, allowing the IN and OUT instructions to occupy just two bytes. The IN and OUT instructions are the only two 8085 instructions that allow writing to (OUT) or reading from (IN) an I/O device.

To run this machine language program, turn on the Trainer and the program counter address FF01 will be shown on the left four displays. Load the machine language data from EXAMPLE 6.1 into memory. Once the whole program has been entered correctly return the program counter address to FF01 by pressing the reset button. 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.
Since this program uses the JMP <address> instruction it will not stop until the reset button is pressed. After pressing the reset button, the program will stop, the digital output LEDs will turn off and the display will show "FF01 db".

Single Stepping

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 2F" 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 CMA instruction. To single step the rest of the program, do the following:

1) Single step again and the display will show "FF04 D3". The CMA instruction has been executed and the PC register was returned with the address of the OUT 11 instruction.

2) Single step again and the display will show "FF06 C3". the OUT 11 instruction has been executed and the PC register was returned with the address of the JMP FF01 instruction.

3) 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.

4) Single step again and the display will show "FF03 2F". The IN instruction has been executed and the PC register was returned with the address of the CMA 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 the A register will be complemented, and after step two you will see the digital output LEDs now show the new value of the A register*. If you press "Step" four more times, the IN instruction will change the A register to the value of the DIP switches again, CMA will complement it and the OUT instruction will display the new value of the A register on the digital output LEDs.

*** REMEMBER:** 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.

Breakpoints

Sometimes it is desirable to run full speed to a certain part of a program before single stepping, to run full speed through a long loop or subroutine, or to see if a program ever gets to a certain memory location. These things can all be accomplished through the use of breakpoints.

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. When setting a breakpoint it is very important that the breakpoint memory address be that of an opcode. The program will not break if, the address of an operand is given, the opcode does not get executed, or the breakpoint address is in EPROM. Once a program breaks the breakpoint is cleared and must be set again if another breakpoint is to occur.

Set a breakpoint at FF04 which contains the opcode "D3" of the OUT instruction. Set the PC to address FF06 and run the program by pressing "Func." and the "Step/Run" key. The microprocessor will run the program at full speed breaking at address FF04. At this point the user can examine registers, single step, or set a new breakpoint.

EXERCISE 6.0

Tracing a program, as previously mentioned, is a useful debugging aid. We can use the single stepper to trace a program by interrogating any registers or memory locations of interest, and logging this information for each executed instruction. To trace the above program press reset and verify the first line of the table below. Step the microprocessor and fill in the second line. Continue in this fashion until all lines of the table are complete. Before starting, put switches 1,2,3,6, and 7, in the ON position and 4, 5, and 8, are in the OFF position.

INSTRUCTION 0 PC FF01 A 00 OPCODE DB MNEMONIC IN 12

INSTRUCTION 1 PC _____ A _____ OPCODE _____ MNEMONIC _____

INSTRUCTION 2 PC _____ A _____ OPCODE _____ MNEMONIC _____

INSTRUCTION 3 PC _____ A _____ OPCODE _____ MNEMONIC _____

INSTRUCTION 4 PC _____ A _____ OPCODE _____ MNEMONIC _____

Before continuing, move DIP switches 5, 6, and 7, to the OFF position with all other switches ON.

INSTRUCTION 5 PC _____ A _____ OPCODE _____ MNEMONIC _____

INSTRUCTION 6 PC _____ A _____ OPCODE _____ MNEMONIC _____

INSTRUCTION 7 PC _____ A _____ OPCODE _____ MNEMONIC _____

INSTRUCTION 8 PC _____ A _____ OPCODE _____ MNEMONIC _____

Note that the opcode and mnemonic reflect the next instruction to execute. In this program the only registers that were utilized were the PC and the A register, requiring only these registers to be traced. Programs can be traced without the use of a microprocessor by determining how each instruction affects registers and memory and filling in an appropriate table by hand. This method of tracing a program is referred to as hand tracing. Hand tracing is often used when developing a program to verify the program execution.

QUESTIONS 6.0

1. As stated above all memory addresses of the 8085 microprocessor are 16 bits. What is the maximum amount of memory that the 8085 microprocessor can directly access?

2. As stated above, all I/O addresses of the 8085 microprocessor consist of 8 bits. How many different Read/Write I/O devices can the 8085 microprocessor directly access?

3. If a software breakpoint was set at address FF04 and the program was run starting at address FF06, would the processor break?

4. If a software breakpoint was set at address FF05 and the program was run starting at address FF01, would the processor break?

5. Starting with the PC set at address FF07, step 2 instructions and fill in the table below.

   INSTR. 1 PC _____ A _____ OPCODE _____ MNEMONIC _____

   INSTR. 2 PC _____ A _____ OPCODE _____ MNEMONIC _____

6. Explain the results obtained in question 6.0.5.

7. If the A register contained the value 5A Hex, what would be the value of the A register after executing the CMA instruction? What would be the value of the A register after executing two successive CMA instructions?

8. For each of the following instructions state YES or NO to whether the instruction affects any of the microprocessor flags (refer to Appendix B).

   IN <BYTE> _____

   OUT <BYTE> _____

   CMA _____

   JMP <ADDRESS> _____

9. After executing the following machine instruction, what address would be in the PC?

   FF01 C3
   FF02 50
   FF03 05

10. When a software breakpoint occurs does the microprocessor stop running?

12. Why would program execution not break if a software breakpoint was set to an address in EPROM?

LAB #7
LOADING REGISTERS

INTRODUCTION

Before a microprocessor can do any useful work registers need to be loaded with appropriate values. Some of these values may be 8 bit values and require storage in a single register. Other values may be 16 bits in length and require a register pair to hold the value. Values larger than 16 bits are generally stored in memory. Instructions that load registers, or move data from one place to another are referred to as Data Transfer Instructions.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* To be able to understand the instruction coding process for the data transfer instructions presented in this lab.

* To be able to understand immediate addressing.

* To become familiar with the following 8085 instructions.

MVI r,<data8> opcode = (depends on register)

MVI r,<data8> is the move immediate instruction. This instruction is used to load binary 8 bit data (<data8>) into the various microprocessor registers. MVI r,<data8> is in the data transfer group of instructions and is a two byte instruction. The first byte of the instruction contains the opcode which tells the processor the register (r) where the data should be moved. The second byte contains the data. This instruction is 2 bytes in length.

LXI rp,<data16> opcode = (depends on register pair)

LXI rp,<data16> is the load register pair immediate instruction. This instruction is used to load binary 16 bit data (<data16>) into the various microprocessor register pairs. LXI rp,<data16> is in the data transfer group of instructions and is a three byte instruction. The first byte of the instruction contains the opcode, the next byte contains the low order data byte, and the last byte of the instruction contains the high order data byte. This instruction is 3 bytes in length.

RST 7 opcode = FF

The RST 7 is a special instruction that performs what is called a software interrupt. Whenever this instruction is executed control is transferred back to MOS, allowing the user to again enter commands. When writing a program on the PRIMER Trainer, this instruction is used to terminate (end) the program.

PROCEDURE

This lab focuses on the MVI r,<data8> and LXI r,<data16> data transfer instructions. The "I" in each of these mnemonics indicates that these instructions use immediate addressing. If an instruction uses immediate addressing, the data is part of the instruction. In the case of the MVI r,<data8> instruction, the first byte of the instruction is of course the opcode and the second byte of the instruction is the 8 bit data. The LXI rp,<data16> instruction is a three byte instruction; the opcode is followed by two bytes comprising the 16 bit data.

The immediate addressing mode is just one of four addressing modes (Immediate, Register, Direct, and Register Indirect) available on the 8085 microprocessor. The other three addressing modes will be presented in later labs. Different addressing modes allow data to be manipulated in different ways, and provides the programmer the ability to program efficiently. Some microprocessors such as the 8088 used in the IBM PC, have several addressing modes not offered by the 8085. While these additional addressing modes make the 8088 processor more flexible, it makes it comparatively more complicated than the 8085 microprocessor.

Using The MVI r,<data8> Instruction

MVI r,<data8> is a two byte instruction, where the opcode is followed by eight bits of data. To code the MVI r,<data8> instruction, the first two most significant bits, called the instruction code, are 00. These two bits indicate the class of instruction. The next three bits contain the register code that tells the microprocessor which register is the destination for the data. The registers are identified by their code in the following table:

TABLE 7.0 8 Bit Registers

Bits	Register Name
111	A
000	B
001	C
010	D
011	E
100	H
101	L

The final three bits are coded 110. These three bits are referred to as the instruction type and help further classify the instruction.

EXAMPLES 7.0

Here are two examples of how the syntax for an MVI r,<data8> instruction opcode is coded:

Example 1
To move data to register A (MVI A,<data8>), the opcode would look like this in binary form:

Class	A register code	Type
00	111	110

When converted to hexadecimal, the byte would look like this:

0011	1110
3	E

Example 2
To move data to the C register (MVI C,<data8>), the opcode would look like this in binary form:

Class	C register code	Type
00	001	110

Converted to Hexadecimal, the byte would look like this:

0000	1110
0	E

EXERCISE 7.0

To move the binary data 01011111 (5F) to the A register, you could use the following machine language program:

ADDRESS	CONTENTS	INSTRUCTION
FF01	3E	MVI A,5F
FF02	5F	Data
FF03	FF	RST 7

MVI A,5F Moves the 8 bit data (5F) immediately following the opcode in memory (next address) to the A register .

RST 7 terminates the program.

Follow the steps below regarding the above program of EXERCISE 7.0.

1. Enter the above program into the trainer.
2. Return the PC to address FF01 and run the program
3. After the program has run, Examine the A register by pressing "func." and "A/A.F.". The left two Hex digits display the data, 5F in the A register.
4. Change the above program so that the value placed into the A register is F5 rather than 5F.
5. Run the program from address FF01 and verify your change.
6. Further modify the program so that F5 is now placed into the B register instead of the A register.
7. List the modified machine language program below.

ADDRESS	CONTENTS	INSTRUCTION
_____	_____	_____
_____	_____	_____
_____	_____	_____

Using The LXI rp<data16> Instruction

LXI rp,<data16> is a two byte instruction, where the opcode is followed by two bytes comprising a 16 bit data value. The first byte following the opcode is the low order byte of the data. The second byte following the opcode (third byte of the instruction) is the high order byte of the data. Remember for 16 bit values, the low order byte is in the lower memory address and the high order byte is in the higher memory address. To code the LXI rp,<data16> instruction, the first two most significant bits called the instruction code are 00. These two bits classify the instruction. Since this instruction is similar to the MVI instruction the class is the same. The next two bits contain the register pair code that tells the microprocessor which register pair is the destination for the data. The register pairs are identified by their code in the following table:

TABLE 7.1 16 Bit Registers

Bits	Register Pair Name
00	B --> IMPLIES B/C
01	D --> IMPLIES D/E
10	H --> IMPLIES H/L
11	SP

The "X" in the LXI mnemonic indicates that the instruction deals with 16 bit data values. Since certain instructions deal with 8 bit values and other instructions deal with 16 bit values, the operands for the mnemonics can be the same. For example the instruction MVI H,01 instruction would move the value 1 into the H register. Using the instruction LXI H,01 on the other hand would move the value 1 to the L register and the value 0 to the H register. Since the LXI instruction deals with 16 bit values, the value 01 is treated as a 16 bit value (think of it as 0001 in Hex) and as such the High order is loaded into the H register and the Low order is loaded into the L register. Note that 1, 01, or 0001 have all the same value of one. The MVI H,01 instruction and the LXI H,01 instruction use the same operand of "H", although for the LXI instruction H implies the H/L register pair.

The final four bits of the LXI instruction are coded 0001, which differentiates the LXI instruction from the MVI instruction. These four bits are used to help further classify the instruction.

EXAMPLES 7.1

Here are two examples of how the syntax for an LXI rp,<data16> instruction opcode is coded:

Example 1

To move data to register pair H/L (LXI H,<data16>), the opcode would look like this in binary form:

Class	H/L register code	Type
00	10	0001

Example 2

To move data to the Stack Pointer register (LXI SP,<data16>), the opcode would look like this in binary form:

Class	SP register code	Type
00	11	0001

EXERCISE 7.1

To move the 16 bit data value "1234" to both D/E and H/L register pairs, you could use the following machine language program:

ADDRESS	CONTENTS	INSTRUCTION
FF01	16	MVI D,12
FF02	12	Data
FF03	1E	MVI E,34
FF04	34	Data
FF05	21	LXI H,1234
FF06	34	Low Data
FF07	12	High Data
FF08	FF	RST 7

Follow the steps below regarding the above program of EXERCISE 7.1.

1. Enter the above program into the trainer.

2. Return the PC to address FF01 and trace the program execution a step at a time (single step). Fill in the table below.

INSTR.	PC	D	E	H	L	OPCODE	MNEMONIC
0	FF01	00	00	00	00	16	MVI D,12
1	_____	____	____	____	____	____	_____
2	_____	____	____	____	____	____	_____
3	_____	____	____	____	____	____	_____

3. Change the above program so that the value 4321 is loaded into the B/C register pair rather than loading 1234 into the H/L register pair..

4. Run the program from address FF01 and verify your change.

5. List the memory values that are now present in the in addresses below.

   FF01 _____

   FF02 _____

   FF03 _____

EXERCISE 7.2

Programs are normally written in assembly language and then translated into machine language. This eases the burden of programming in machine language. Using instructions presented in this lab, translate (hand assemble) the assembly language program below into machine language.

```
one   equ   01
start equ   FF01h

      org   start

      lxi   h,6789h
      lxi   d,2345h
      mvi   a,one
      rst   7
      end
```

1. Verify the execution of the above program, by single stepping the resulting machine code.

2. Once verified, list the machine language program below.

ADDRESS	CONTENTS	INSTRUCTION
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

QUESTIONS 7.0

1. How many bytes comprise the MVI instruction?

2. How many bytes comprise the LXI instruction?

3. How many bytes comprise the RST 7 instruction?.

4. The "I" in the MVI and LXI instructions indicates what?

5. The "X" in the LXI instruction indicates what?

6. Give the mnemonic for the following opcodes, by decoding each of the Hex values.

   36 Hex _____ 01 Hex _____

7. Define microprocessor addressing mode.

8. Define Immediate addressing.

9. Fill in the B and C register values after execution of the following machine code sequence.

   FF01 01
   FF02 11
   FF03 00
   FF04 03
   FF05 22
   FF06 FF

   B = _____ C = _____

LAB #8
TRANSFERRING DATA BETWEEN REGISTERS

INTRODUCTION

Once data has been loaded into a register, there are often times when a copy of the data needs to be placed in another register. Situations occur within programs that require the original data to be modified, making it necessary to use the copy that was placed in another register for this purpose. More often than not, however, the need to change registers is determined by the instructions being used. Certain instructions require the use of specific registers, facilitating the need to move information from one register to another. For example the CMA (Complement Accumulator) instruction operates on data held in the A register. The instructions covered in this lab belong to the Data Transfer group.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* To be able to understand the instruction coding process of the MOV r1,r2 instruction.

* To be able to understand register addressing.

* To become familiar with the following instructions.

MOV r1,r2 opcode = (depends on registers)

MOV r1,r2 is the move register instruction. This instruction enables the 8 bit contents of register r2 (source) to be copied to register r1 (destination). This instruction is 1 byte in length.

XCHG opcode = EB

Exchanges the 16 bit value in register pair H/L with the 16 bit value in register pair D/E. Register H is exchanged with register D and register L is exchanged with register E. This instruction is 1 byte in length.

PROCEDURE

This lab looks at two data transfer instructions that allow data to be moved or exchanged between registers. The MOV r1,r2 instruction uses register addressing to determine which registers are involved in the MOV instruction. With register addressing the instruction contains (in coded form) the registers used by the instruction. The MOV r1,r2 instruction can therefore be and is, a single byte instruction. The XCHG instruction is a fixed instruction, using the H/L register pair and the D/E register pair, so no operands are required. Since there can be no variation of this instruction no addressing mode is required. The XCHG instruction is also a single byte instruction.

Using The MOV r1,r2 Instruction

MOV r1,r2 is a single byte instruction, where the opcode is the only byte of the instruction. Using register addressing, no additional bytes are required. To code the MOV r1,r2 instruction, the first two most significant bits, called the instruction code, are 01. These two bits indicate the class of the instruction. The next three bits contain the 8 bit register code that tells the microprocessor which register is the destination for the data. The last three bits of the opcode contains the 8 bit register code of the source register. The data is moved by the microprocessor from the source register to the destination register. The registers are identified by their code in the following table:

TABLE 8.0 **8 Bit Registers**

Bits	Register Name
111	A
000	B
001	C
010	D
011	E
100	H
101	L

EXAMPLES 8.0

Here are two examples of how the syntax for an MOV r1,r2 instruction opcode is coded:

Example 1
To move data to register A, from register B (MOV A,B), the opcode would look like this in binary form:

Instruction Class	Destination A register code	Source B register code
01	111	000

When converted to hexadecimal, the byte would look like this:

0111	1000
7	8

Example 2
To move data to the C register, from the D register (MOV C,D), the opcode would look like this in binary form:

Instruction Class	Destination C register code	Source D register code
01	001	010

Converted to Hexadecimal, the byte would look like this:

0100	1010
4	A

EXERCISE 8.0

1. Enter the program below into the trainer starting at address FF01.

ADDRESS	CONTENTS	INSTRUCTION
FF01	01	LXI B,1234
FF02	34	Low Data
FF03	12	High Data
FF04	69	MOV L,C
FF05	60	MOV H,B
FF06	16	MVI D,56
FF07	56	Data
FF08	1E	MVI E,78
FF09	78	Data
FF0A	EB	XCHG
FF0B	FF	RST 7

2. Return the PC to address FF01 and trace the program execution a step at a time (single step). Fill in the table below.

INSTR.	PC	B	C	D	E	H	L	OPCODE	MNEMONIC
0	FF01	00	00	00	00	00	00	01	LXI B,1234
1	_____	____	____	____	____	____	____	____	_____
2	_____	____	____	____	____	____	____	____	_____
3	_____	____	____	____	____	____	____	____	_____
4	_____	____	____	____	____	____	____	____	_____
5	_____	____	____	____	____	____	____	____	_____
6	_____	____	____	____	____	____	____	____	_____

3. In the above program of EXERCISE 8.0, if instruction MOV L,C was changed to MOV L,B and instruction MOV H,B was changed to MOV H,C, what change could be made to instruction LXI B,1234 to allow the same final register contents as listed in INSTR. 6 above for registers D, E, H, and L?

4. Write the shortest machine language program possible in terms of bytes, to initialize the A, B, C, D, E, H, and L registers to zero. (Hint the program can be written in 8 or less bytes).

QUESTIONS 8.0

1. Give the mnemonic for the following opcodes, by decoding each of the Hex values.

   47 Hex _____ 6F Hex _____

2. If the program of EXERCISE 8.0 had the following modification, what would be the register contents after the RST 7 instruction executed?

MEMORY	CONTENTS
FF0B	EB
FF0C	FF

   Fill in the register contents in the blanks below.

PC	**B**	**C**	**D**	**E**	**H**	**L**
_____	____	____	____	____	____	____

3. Explain the results of question 8.0.2.

4. What happens if you move the contents of the A register to the A register? Is the instruction MOV A,A a valid instruction?

5. When the contents of the A register are moved to another register are the contents of the A register changed?

6. Give the Hex opcode for each of the following mnemonics.

   MOV H,L _____

   MOV B,C _____

7. In the program of EXERCISE 8.0, if a breakpoint is set at address FF03 and run from address FF01, does the processor break at address FF03? Justify your answer.

8. Did any of the instructions in the program of EXERCISE 8.0 change any of the status flags? If so, list the instructions that changed the flags?

LAB #9
INCREMENTING AND DECREMENTING REGISTERS

INTRODUCTION

The incrementing and decrementing of registers is a powerful feature. Using the increment instruction, we can count the occurrences of some event, like the number of items that have come down an assembly line or the number of people that have entered a building. Using the decrement instruction, we can set time limits on a particular activity or know how much of a commodity is left. Later we will see how the increment and decrement instructions can be used to sequence through memory to access data.

The increment and decrement instructions are part of the arithmetic group of instructions. The arithmetic group differs from the data transfer group in that no instructions within the data transfer group affected the flags. Most instructions within the arithmetic group, however, do affect the flags. Having the flags affected is important in making decisions (when time gets to zero, sound an alarm). This decision making process is what gives the microprocessor its "intelligence".

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* To understand how the flags are affected by the increment and decrement instructions.

* To understand the coding process for the increment and decrement instructions.

* To become familiar with the following 8085 instructions

INR r opcode depends on the operand

The INR r instruction increments the 8 bit register r by the value of one. All flags except the carry flag are affected. This instruction is 1 byte in length.

DCR r opcode depends on the operand

The DCR r instruction decrements the 8 bit register r by the value of one. All flags except the carry flag are affected. This instruction is 1 byte in length.

INX rp opcode depends on the operand

The INX rp instruction decrements the 16 bit register pair rp by the value of one. No flags are affected. This instruction is 1 byte in length.

DCX rp opcode depends on the operand.

The DCX rp instruction decrements the 16 bit register pair rp by the value of one. No flags are affected. This instruction is 1 byte in length.

PROCEDURE

The increment and decrement instructions come in 8 and 16 bit versions. As mentioned previously the "X" in the INX and DCX instructions indicate that these instructions deal with register pairs (16 bit values). Another difference between these instructions is the way they affect the status flags. The INR and DCR instructions affect all of the flags with the exception of the carry flag. The INX and DCX instructions do not affect any flags. These instructions belong to the arithmetic group and use register addressing to determine which register is incremented.

Coding the DCR r and INR r Instructions

The DCR r and INR r instructions are both single byte instructions and are coded similarly to the MVI instruction. The instruction code for both these instructions is 00. The next three bits is the register field, which tells the microprocessor which 8 bit register to increment or decrement. See TABLE 7.0, the table of 8 bit registers. The last three bits are the instruction type and is used to further classify the instruction.

EXAMPLES 9.0

Here are two examples of how the syntax for INR r and DCR r opcodes are coded:

Example 1
To increment register A (INR A), the opcode would look like this in binary form:

Class	A register code	Type
00	111	100

When converted to hexadecimal, the byte would look like this:

0011	1100
3	C

Example 2
To Decrement register D (DCR D), the opcode would look like this in binary form:

Class	D register code	Type
00	010	101

Converted to Hexadecimal, the byte would look like this:

0001	0101
1	5

Coding the DCX rp and INX rp Instructions

The DCX rp and INX rp instructions are both single byte instructions and are coded similarly to the LXI instruction. The instruction code for both these instructions is 00. The next two bits are the register pair field, which tells the microprocessor which 16 bit register pair to increment or decrement. See TABLE 7.1, the table of 16 bit register pairs. The last four bits are the instruction type and are used to further classify the instruction.

EXAMPLES 9.1

Here are two examples of how the syntax for INX rp and DCX rp opcodes are coded:

Example 1
To increment register pair B/C (INX B), the opcode would look like this in binary form:

Class	B/C register code	Type
00	00	0011

When converted to hexadecimal, the byte would look like this:

0000	0011
0	3

Example 2
To decrement the H/L register (DCX H), the opcode would look like this in binary form:

Class	H/L register code	Type
00	10	1011

Converted to Hexadecimal, the byte would look like this:

0011	1011
2	B

EXERCISE 9.0

1. Enter the program below into the trainer starting at address FF01.

ADDRESS	CONTENTS	INSTRUCTION
FF01	01	LXI B,00FE
FF02	FE	Low Data
FF03	00	High Data
FF04	3E	MVI A,2
FF05	02	Data
FF06	16	MVI D,0FF
FF07	FF	Data
FF08	03	INX B
FF09	3D	DCR A
FF0A	14	INR D
FF0B	03	INX B
FF0C	3D	DCR A
FF0D	14	INR D
FF0E	16	MVI D,00
FF0F	00	Data
FF10	03	INX B
FF11	3D	DCR A
FF12	FF	RST 7

2. Return the PC to address FF01 and trace the program execution a step at a time (single step). Fill in the register contents and flag (Zero and Carry) contents in the spaces provided below.

INSTR.	PC	A	B	C	D	Zf	Cf	OPCODE	MNEMONIC
0	FF01	00	00	00	00	0	0	01LXI	B,00FE
1	_____	____	____	____	____	____	____	____	_____
2	_____	____	____	____	____	____	____	____	_____
3	_____	____	____	____	____	____	____	____	_____
4	_____	____	____	____	____	____	____	____	_____
5	_____	____	____	____	____	____	____	____	_____
6	_____	____	____	____	____	____	____	____	_____
7	_____	____	____	____	____	____	____	____	_____
8	_____	____	____	____	____	____	____	____	_____
9	_____	____	____	____	____	____	____	____	_____
10	_____	____	____	____	____	____	____	____	_____
11	_____	____	____	____	____	____	____	____	_____
12	_____	____	____	____	____	____	____	____	_____

3. Change the LXI B,00FE instruction to LXI B,0FEFF. Set a breakpoint at address FF0C and run the program starting at address FF01. Examine the registers and flags after the break and fill in the blanks below.

PC	A	B	C	D	Zf	Cf	OPCODE	MNEMONIC
_____	____	____	____	____	____	____	____	_____

4. Replace all the INR D instructions with INX D instructions in the above program. Run the program starting at address FF01 with no breakpoints. Examine the registers and flags after the program executes and fill in the blanks below.

PC	A	B	C	D	Zf	Cf	OPCODE	MNEMONIC
_____	____	____	____	____	____	____	____	_____

QUESTIONS 9.0

1. If the A register was initialized to zero and then incremented (INR A) 512 times, what would be the contents of the A register?

2. If the A register was initialized to zero and then decremented (DCR A) 512 times, what would be the contents of the A register?

3. If the B/C register pair was initialized to zero and then decremented (DCX B) 512 times, what would be the contents of the C register? What would be the contents of the B register?

4. Which of the instructions in the program of EXERCISE 9.0 affected the carry flag?

5. Which of the instructions in the program of EXERCISE 9.0 affected the zero flag?

6. Construct a two instruction program that uses the MVI instruction and a decrement instruction, to set the Zero flag to one.

7. In question 9.0.6, what value would be present in the A register when the Zero flag becomes one.

8. If the A register contained zero and the Zero flag was one, what increment instruction would clear the Zero flag to zero.

9. If the B register contained zero and the instruction MOV A,B was executed what flags would be affected.

10. Decode the following mnemonics to machine code.

    INX SP _____

    DCR L _____

LAB #10
PROGRAM LOOPING

INTRODUCTION

Frequently in programs it is desirable to execute a section of code a certain number of times. Rather than duplicate this section of code over and over, a program loop can be implemented. In LAB #6 we used a loop to continually check and output the DIP switch contents to the output LED's. The program of LAB #6 had no means of exiting the loop. A loop with no exit is referred to as a infinite loop, because it will execute the same instructions forever. A conditional loop, on the other hand, executes the instructions within the loop until the condition is met and then exits the loop. The branch group of instructions allow both conditional and unconditional (infinite) looping.

In order to implement a conditional jump, the flag register must be utilized. Certain instructions within the arithmetic group and logic group of instructions affect the flags. These instructions can then be utilized to make decisions on whether the program should loop or fall through (exit) the loop. This lab introduces all of the conditional jump instructions, but focuses on the jump instructions that reference the Zero flag.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* To understand the concept of an unconditional jump.
* To understand the concept of a conditional jump.
* To be able to construct a delay loop.
* To be able to construct a program from a flowchart.
* To be able to exit a loop based on a comparison.
* To become familiar with the following instructions.

NOP opcode = 00

A special instruction that does no operation. The NOP instruction does not affect any registers or flags. This instruction basically does nothing but occupy time and one byte of memory space. This instruction is 1 byte in length.

Jccc <address> opcode depends on condition ccc

Jccc is the conditional jump instruction. ccc can be one of eight conditions referencing four of the status flags. If the selected condition is true, then program execution branches to the jump address <address>. If the condition is false then execution continues at the next instruction following the jump instruction. This instruction is 3 bytes in length.

CPI <data8> opcode = FE

Compare Immediate instruction. This instruction compares the byte following the opcode with the A register (Subtracts byte2 of the insrtuction from the A register) without changing the contents of the A register. This instruction only affects the flags. The carry flag = 1 if A is less than <data8> else the carry flag = 0. The zero flag = 1 if A equals <data8> else the zero flag = 0. All flags (Z, S, P, CY, AC) are affected by this instruction. This instruction is 2 bytes in length.

PROCEDURE

In this lab we analyze programs that utilize both the conditional and unconditional jump instructions. Refer back to lab #6 if necessary in order to refresh your memory about using the JMP instruction. It is often useful in programs to delay for a period of time before continuing. For example, a delay could be used in a burglar alarm to allow a person to shut off the alarm after they have entered the house. The NOP instruction can be placed in a delay loop in order to lengthen the delay, without affecting any flags or registers.

Using The Conditional Jump Instruction Jccc <address>

The conditional jump like the unconditional jump are 3 byte instructions. The first byte is the opcode, followed by the low order address, and then the high order address. Branch type instructions like the jump instruction have their own address modes,referred to as branch addressing modes. The 8085 microprocessor supports only one branch addressing mode called absolute addressing. Absolute addressing means the branch instruction contains the complete address of the jump. With absolute addressing we can jump anywhere we want within the 64K of memory space.

To code the Jccc <address> instruction, the first two most significant bits called the instruction code are both ones. These two bits classify the class of instruction. The next three bits contain the 8 bit condition code that tells the microprocessor on what condition to jump. The conditions are identified by their code in the following table:

TABLE 10.0 Conditions

ccc	Condition
000	NZ not zero (Zf = 0)
001	Z zero (Zf = 1)
010	NC not carry (Cf = 0)
011	C carry (Cf = 1)
100	PO parity odd (Pf = 0)
101	PE parity even (Pf = 1)
110	P plus (Sf = 0)
111	M minus (Sf = 1)

The final three bits are coded 010. These bits are referred to as the instruction type and help further classify the instruction.

EXAMPLES 10.0

Here are two examples of how the syntax for Jccc opcode is coded:

Example 1
To Jump Zero (JZ <address>), the opcode of the instruction would look like this in binary form:

Class	Z condition code	Type
11	001	010

When converted to hexadecimal, the byte would look like this:

1100	1010
C	A

Example 2
To Jump Not Carry, the opcode of the instruction would look like this in binary form:

Class	NC condition code	Type
11	010	010

Converted to Hexadecimal, the byte would look like this:

1101	0010
D	2

Although the decoding of all the conditional jump instructions were presented, this lab will emphasize the JNZ and JZ instructions. When using conditional jump instructions it important to realize that the flags are the basis of whether or not the jump takes place. This provides a very useful function in that the condition flags remember the results of an operation, allowing other instructions that do not modify the flags to execute before the conditional jump instruction is encountered. If an instruction sets a particular flag and it is the basis of a decision (conditional jump), no other instruction that modifies that particular flag should execute before the conditional jump, or the value of the flag may be changed and cause an incorrect jump.

As you can see from table 10.0, JNZ will occur when the result of an operation is Not Zero (Z=0) and JZ will occur when the result of an operation is Zero (Z=1). The zero flag may seem a bit confusing because when it is 1, it indicates an operation had a 0 result and when it is 0 it indicates a non-zero result. Just remember that a 1 indicates "yes" and 0 indicates "no". So, for example, when a decrement operation results in zero this causes the zero flag to be set to 1, signaling yes, the operation result was zero. If the operation did not result in zero, the zero flag would be 0 indicating no, the result was not zero.

EXERCISE 10.0

1. Enter the following delay loop program into memory starting at address FF01.

ADDRESS	CONTENTS	INSTRUCTION
FF01	00	NOP
FF02	3E	MVI A,02
FF03	02	Data
FF04	00	NOP
FF05	00	NOP
FF06	3D	DCR A
FF07	C2	JNZ FF04
FF08	04	Address low
FF09	FF	Address high
FF0A	3C	INR A
FF0B	FF	RST 7

2. Return the PC to address FF01 and trace the program execution a step at a time (single step). Fill in the register contents and Zero flag contents in the blanks below.

INSTR.	PC	A	Zf	OPCODE	MNEMONIC
0	FF01	00	0	00	NOP
1	_____	____	____	____	_____
2	_____	____	____	____	_____
3	_____	____	____	____	_____
4	_____	____	____	____	_____
5	_____	____	____	____	_____
6	_____	____	____	____	_____
7	_____	____	____	____	_____
8	_____	____	____	____	_____
9	_____	____	____	____	_____
10	_____	____	____	____	_____
11	_____	____	____	____	_____

3. In the program of EXERCISE 10.0, how many iterations of the loop (times through the loop/number of jumps) did the program execute.

4. If in the program of EXERCISE 10.0, the MVI A,02 instruction was changed MVI A,00 how many iterations of the loop would be executed?

5. If in the program of EXERCISE 10.0, the JNZ FF04 instruction was changed to JNZ FF02 how many iterations of the loop would take place?

6. If in the program of EXERCISE 10.0, the JNZ FF04 instruction was changed to JZ FF04 how many iterations of the loop would be executed?

EXERCISE 10.1

In the program of EXERCISE 10.0, the delay is very short. The delay can be made longer by increasing the value of the A register. However even with the maximum delay set, the microprocessor can execute the program almost instantly. In order to get longer delays we can take advantage of a double loop. Enter the following double loop program into memory.

ADDRESS	CONTENTS	INSTRUCTION
FF01	01	LXI B,8000
FF02	00	Data low
FF03	80	Data high
FF04	0D	DCR C
FF05	79	MOV A,C
FF06	00	NOP
FF07	00	NOP
FF08	C2	JNZ FF04
FF09	04	Address low
FF0A	FF	Address high
FF0B	05	DCR B
FF0C	C2	JNZ FF04
FF0D	04	Address low
FF0E	FF	Address high
FF0F	00	NOP
FF10	FF	RST 7

1. Run the above program starting at address FF01 with no breakpoints set. After the program finishes running fill in the register and flag contents below.

PC	A	B	C	Zf	Cf	OPCODE	MNEMONIC
_____	____	____	____	____	____	____	_____

2. Change the two NOP's at address FF06 and FF07 to an OUT 11 (two byte) instruction. Run the program full speed and determine how many times the eight output LED's are all on at the same time. (Hint, do not try and count the number of times the LED's are all on but try to calculate the value by determining the number of iterations of the loops).

3. Construct a flowchart for the modified program of EXERCISE 10.1.2 .

4. Change the contents of memory location FF03 from 80 to 00. Does the program now take a longer time or a shorter time to execute?

5. Write an assembly language program and the corresponding machine code, that performs the actions of the flowchart below.

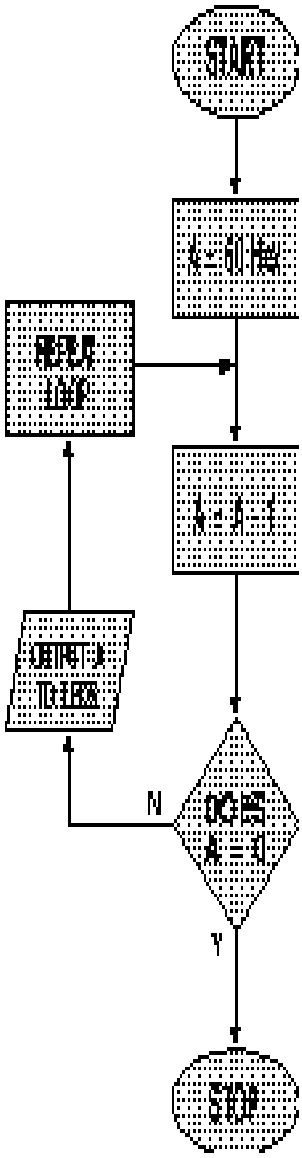


FIG. 10.0

The program of EXERCISE 10.0 utilized a down counter loop in order to achieve a short delay. The CPI <data8> instruction can be used to implement an up counter loop delay. Instead of using the DCR A instruction to set the Zero flag when the A register is empty, the CPI <data8> instruction will be used to set the Zero flag when the contents of the A register equals the value <data8> (second byte of the instruction). Figure 10.1 graphically describes how this program operates.

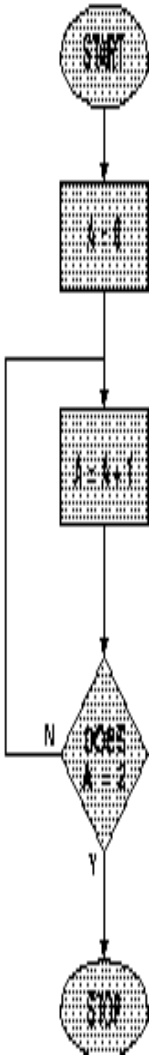


FIG. 10.1

EXERCISE 10.2

The assembly language program for the flowchart of figure 10.1 is listed below:

```
        org   FF01H
        mvi   a,0
loop:   inr   a
        cpi   2
        jnz   loop
        rst   7
```

1. For the assembly language program of EXERCISE 10.2 enter the corresponding machine language in the space provided below.

ADDRESS	CONTENTS	INSTRUCTION
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

2. Enter the machine language program of EXERCISE 10.2.1 into the trainer starting at address FF01.

3. Return the PC to address FF01 and trace the program execution a step at a time (single step). Fill in the register contents and Zero flag contents in the blanks below.

INSTR.	**PC**	**A**	**Zf**	**OPCODE**	**MNEMONIC**
0	FF01	00	0	3E	MVI A,0
1	_____	____	____	____	_____
2	_____	____	____	____	_____
3	_____	____	____	____	_____
4	_____	____	____	____	_____
5	_____	____	____	____	_____
6	_____	____	____	____	_____
7	_____	____	____	____	_____

4. Modify the assembly language program given in EXERCISE 10.2 so that the program counts from 10 to 20 (10 interations of the loop). List the modified assembly language program below.

5. Construct a flowchart and provide the resulting assembly language program to loop indefinitely until all of the DIP switches are in the ON position, whereby the loop exits and the output leds are subsequently all lit, and the program ends. (Hint: Refer to EXAMPLE 4.0).

QUESTIONS 10.0

1. A loop that uses an unconditional jump and has no exit is referred to as an infinite loop; why is this so?

2. In order for the JZ instruction to jump the Zero flag must be at what state?

3. What flags, if any, affect the JMP instruction?

4. In the program of EXERCISE 10.1, if the DCR B instruction was replaced with the DCX B instruction, how many times would the DCX B instruction get executed?

5. In the program of EXERCISE 10.1, if a breakpoint was set at address FF0B what value would be in the A register when the break occurred? Assume the program starting address is FF01.

6. In the program of EXERCISE 10.0, if a DCR A instruction is placed at address FF04 in place of the NOP instruction, how many iterations of the loop will take place?

7. In the program of EXERCISE 10.0, if a INR A instruction is placed at address FF04 in place of the NOP instruction, how many iterations of the loop will take place?

8. If we wanted to exit a program loop when all the DIP switches are in the on position, explain how this could be done using the IN, INR A, DCR A, and JNZ instructions. (Do not use the CPI <data8> instruction).

LAB #11
PUSHING, POPPING, AND THE STACK

INTRODUCTION

When writing programs for microprocessors in Assembler we frequently need to preserve the contents of a register or status flag for later use. This can be accomplished by pushing the register or flags onto the stack. A value pushed on the stack can later be removed by popping the value off into a selected register.

The stack is a First In Last Out (FILO) data structure. This is analogous to a stack of trays in a cafeteria, the last tray placed on the stack is the first tray removed. The inserting and removing of values on/off the stack therefore must be carefully managed by the programmer. Simply said, the stack occupies a predetermined chunk of memory that holds data temporarily until needed and a data structure is an organized grouping of data.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* To grasp the concept of the stack.
* To understand the concept of stack underflow and overflow.
* To understand the coding process for the push and pop instructions
* To understand the following instructions and their affect on the stack.

PUSH rp opcode depends on the operand

The PUSH rp instruction puts the 16 bit register pair rp on the stack. The Stack Pointer (SP) is decremented by two. No flags are affected. Note the SP register is not a valid rp for this instruction. This instruction is 1 byte in length.

PUSH PSW opcode = F5

The PUSH PSW instruction puts the 16 bit Program Status Word (PSW, A register and Flag register) on the stack. The stack pointer (SP) is decremented by two. No flags are affected. This instruction is 1 byte in length.

POP rp opcode depends on the operand

The POP rp instruction removes the 16 bit register pair rp off the stack. The Stack Pointer (SP) is incremented by two. No flags are affected. Note the SP register is not a valid rp for this instruction. This instruction is 1 byte in length.

POP PSW opcode = F1

The POP PSW instruction removes the 16 bit Program Status Word (PSW, A register and Flag register) off the stack. The Stack Pointer (SP) is incremented by two. No flags are affected. This instruction is 1 byte in length.

PROCEDURE

The 8085 microprocessor allows the selection of an area of memory to be deemed the stack. The key element of the stack is the Stack Pointer (SP) register located in the 8085 microprocessor. The SP always points to the Top Of Stack, that is the memory address of the last value placed on the stack. Each stack element is a 16 bit value, and as such requires two memory locations to hold the value. The low order byte is stored in the lower memory address and the high order byte is stored in the higher memory address. When a value is to be pushed on the stack, the SP is first decremented by two so as to not overwrite the last value placed on the stack. This new value is now the new Top of Stack. When a value is to be popped off the stack, the value is first removed into the selected register pair and then the SP is incremented by two to the next value on the stack making it the Top Of Stack. When all values have been removed the stack is said to be Balanced or Empty.

Since every time a value is placed on the stack the SP is decremented by two, the SP is generally initialized to a memory location high up in memory where it has room to grow. If too many values are placed on the stack without removal, the stack can overwrite program code and crash the program. This is known as Stack Overflow. Likewise removing a value from an empty stack is called Stack Underflow. When Stack Underflow occurs, the value popped off the stack is undefined (no value was ever placed on the stack in the first place) and therefore a mistake has been made that will generally cause the program to crash also. Stack Underflow can occur when there are more pops executed than pushes.

The initialization of the SP determines the starting point for the Top Of Stack. The SP is can be initialized using the LXI SP instruction. When using the PRIMER however the SP has already been initialized by MOS to memory address FFD4H, no further initialization is required.

Using the PUSH & POP Instructions

The PUSH instruction places the PSW or a register pair on the Top Of Stack pointed to by the SP plus two and the POP instruction removes the PSW or a register pair from the Top Of Stack pointed to by the SP. To determine if the PSW or a register pair will be used we must specify our choice in the operand. Lets look at how the instruction is coded to specify an operand. The first two bits as seen in previous labs is the instruction class. The instruction class for the PUSH and POP instructions is 11. The next two bits contain the register pair code that tells the microprocessor which register pair or if the PSW is the operand for the instruction. The valid operands are identified by their code in the following table:

TABLE 11.0 16 Bit Registers

Bits	Register Pair Name
00	B --> IMPLIES B/C
01	D --> IMPLIES D/E
10	H --> IMPLIES H/L
11	PSW-> PROGRAM STATUS WORD (SINCE THE SP IS AN INVALID CHOICE)

The final four bits (instruction type) determines which stack instruction (PUSH or POP) we are using and help further classify the instruction. The table below identifies the stack instructions and there types:

TABLE 11.1 STACK INSTRUCTION TYPES

Instruction	Type
PUSH	0101
POP	0001

EXAMPLES 11.0

Here are two examples of how the syntax for the PUSH instruction is coded:

Example 1

To push the H/L register pair, the opcode would look like this in binary form:

Class	H/L register code	Type
11	10	0101

When converted to hexadecimal, the byte would look like this:

1110	0101
E	5

Example 2

To push the PSW, the opcode would look like this in binary form:

Class	PSW code	Type
11	11	0101

When converted to hexadecimal, the byte would look like this:

1111	0101
F	5

EXAMPLE 11.1

Here are two examples of how the syntax for the PUSH instruction is coded:

Example 1

To pop the B/C register pair, the opcode would look like this in binary form:

Class	B/C register code	Type
11	00	0001

When converted to hexadecimal, the byte would look like this:

1100	0101
C	1

Example 2

To pop the PSW, the opcode would look like this in binary form:

Class	PSW code	Type
11	11	0001

When converted to hexadecimal, the byte would look like this:

1111	0001
F	5

EXERCISE 11.0

1. Enter the following program into memory starting at address FF01H.

ADDRESS	CONTENTS	INSTRUCTION
FF01	01	LXI B,1234H
FF02	34	Low Data
FF03	12	High Data
FF04	C5	PUSH B
FF05	D1	POP D
FF06	FF	RST 7

2. Return the PC to address FF01H and trace the program execution a step at a time (single step). Fill in the register contents and Top Of Stack (TOS) contents in the blanks below. To view the 16 bit value (TOS), use the S.C. (Stack Contents) function key on the PRIMER..

INSTR.	P.C.	B.C.	D.E.	S.P.	S.C.	OPCODE	MNEMONIC
0	FF01	0000	0000	FFD4	????	01	LXI B,1234H
1	_____	_____	_____	_____	_____	_____	_____
2	_____	_____	_____	_____	_____	_____	_____
3	_____	_____	_____	_____	_____	_____	_____

The above program simulates a 16 bit move (copy) instruction. Using the stack it PUSHes the B/C register pair on the stack and POPs it off into the D/E register pair, thus moving the contents of B/C to D/E.

3. What was the maximum stack space in bytes used by the program of Exercise 11.0?

4. Modify the program of Exercise 11.0 to move the D/E register pair to the H/L register and list the new program below.

5. Modify the program of Exercise 11.0 as follows:

ADDRESS	CONTENTS	INSTRUCTION
FF01	01	LXI B,1234H
FF02	34	Low Data
FF03	12	High Data
FF04	C5	POP B
FF05	D1	PUSH B
FF06	FF	RST 7

A. Does stack overflow occur in this program?

B. Does stack underflow occur in this program?

C. At the end of the execution of this modified program, what is the stack condition (underflow, overflow, or balanced)?

EXERCISE 11.1

1. Enter the following program into memory starting at address FF01H.

ADDRESS	CONTENTS	INSTRUCTION
FF01	11	LXI D,1234H
FF02	34	Low Data
FF03	12	High Data
FF04	21	LXI H,5678H
FF05	78	Low Data
FF06	56	High Data
FF07	D5	PUSH D
FF08	E5	PUSH H
FF09	D1	POP D
FF0A	E1	POP H
FF0B	FF	RST 7

2. Return the PC to address FF01H and trace the program execution a step at a time (single step). Fill in the register contents and Top Of Stack (S.C.) contents in the blanks below.

INSTR.	P.C.	B.C.	D.E.	S.P.	S.C.	OPCODE	MNEMONIC
0	FF01	0000	0000	FFD4	????	11	LXI D,1234H
1	_____	_____	_____	_____	_____	_____	_____
2	_____	_____	_____	_____	_____	_____	_____
3	_____	_____	_____	_____	_____	_____	_____
4	_____	_____	_____	_____	_____	_____	_____
5	_____	_____	_____	_____	_____	_____	_____
6	_____	_____	_____	_____	_____	_____	_____

3. What single 8085 instruction does the program of Exercise 11.1 perform?

4. What was the maximum stack space in bytes used by the program of Exercise 11.1?

5. What is the maximum stack space available for the program of Exercise 11.1 before stack overflow occurs?

6. Modify the program of Exercise 11.1 as follows:

ADDRESS	CONTENTS	INSTRUCTION
FF01	11	LXI D,1234H
FF02	34	Low Data
FF03	12	High Data
FF04	D5	PUSH D
FF05	11	LXI D,5678H
FF06	78	Low Data
FF07	56	High Data
FF08	D5	PUSH D
FF09	E1	POP H
FF0A	D1	POP D
FF0B	FF	RST 7

7. Run the modified program starting at address FF01H and fill in the blanks below:

P.C.	D.E.	H.L.	S.P.	S.C.	OPCODE	MNEMONIC
FF0B	_____	_____	_____	_____	_____	_____

QUESTIONS 11.0

1. A _____ is an organized grouping of data.

2. Stack _____ occurs when the stack grows to large and overwrites program code.

3. Stack _____ occurs when a value is popped off an empty stack.

4. An empty stack is also referred to as a _____ stack.

5. A stack is a _____ data structure.

6. Each stack element consists of _____ bits.

7. If the status flags and all the registers were to be saved on the stack how many bytes of stack space would be required?

8. Referring to the program of Exercise 11.1, if the instruction at address FF0BH were changed from RST 7 to JMP FF01H would stack overflow occur?

9. Fill in the blanks below to POP off the registers to their original register values prior to the PUSH instructions.

```
PUSH   PSW
PUSH   B
PUSH   H
PUSH   D

POP    _____

POP    _____

POP    _____

POP    _____
```

10. After the following instructions were executed what would be the contents of the A register.

```
LXI    H,0ABCDH
PUSH   H
POP    PSW
```

11. After the following instructions were executed what would be the condition of the stack for each of the following? Answer both A. And B. (Balanced, Overflow, Underflow)

A.
```
       PUSH PSW
LOOP   PUSH B
       PUSH D
       POP  B
       POP  H
       POP  PSW
       JMP  LOOP
```

B.
```
LOOP   PUSH PSW
       PUSH B
       POP  B
       PUSH H
       POP  PSW
       JMP  LOOP
```

LAB #12
SUBROUTINES AND SERVICES

INTRODUCTION

A subroutine is a self contained module which performs a specific function that is accessible from any part of a program. This self contained module can be thought of as a mini program in its own right, and as such, can usually be written and tested independently of the rest of the program. This also allows multiple programmers to work independently on a program at the same time, each writing subroutines that will contribute to the program as a whole.

Once these subroutines have been written and tested, they become building blocks that can be bolted together to construct the program, similar to the way the blocks of a flow chart are connected to provide a graphical representation of a program. If the subroutines are error free, then it can be a relatively simple job to connect them together. This is an efficient way of writing programs and reduces the debug time significantly. Since each subroutine is being debugged independently debugging is much easier.

The first instruction executed in a program is considered to be in the Main program. The Main program begins and ends the program as a whole. Typically between the beginning and the end of the Main program are subroutine Calls. Therefore the Main program is very easy to understand as it primarily consists of subroutines with the names of the subroutines describing what the subroutines functions are. For example COSINE might be the name of a subroutine used to find the cosine of a number.

If I wrote a subroutine called ADDNUM that added two numbers together, I would input to the subroutine the two numbers I wanted to add and the subroutine would output the sum. These values that are passed into and out of the subroutine are called parameters. Not all subroutines require parameters, but most do. Parameters are usually passed to subroutines in registers. For example with the ADDNUM subroutine, I could specify that the first number to added be placed in the B/C register pair, the other number be placed in the D/E register pair, and the sum be returned in the H/L register pair. To test this subroutine is very simple, I give it two numbers and look in the H/L register pair to see if the result is correct. Once the subroutine has been tested, it can be CALLed many times from any point in the program. As a matter of fact it can even be used in other programs that may require adding two numbers.

After a period of time writing programs, you will acquire quite a collection of subroutines that can be utilized in writing new programs. This collection of subroutines is referred to as a library. Subroutines that are to be stored in a library should be written generic in nature, like the ADDNUM subroutine. If we had written ADDNUM so that it always added ten to another number, then this subroutine becomes specific and is of little use in a library.

A subroutine can be CALLed from within another subroutine. This is referred to as nesting a subroutine and the subroutine that is CALLed is referred to as a nested subroutine. If for example we were to write a subroutine called AVERAGE to find the average of several numbers, we could CALL ADDNUM from within AVERAGE to generate a total. And then possibly CALL another subroutine called DIVIDE also from within AVERAGE to finish the computation of the average.

Operating Systems frequently provide subroutines that allow programmers to be more productive. For example when programming under MS/DOS, if a programmer needs to get a key from the keyboard or display a sentence on the video monitor, a predefined subroutine that is part of DOS is used to do this. These predefined subroutines are called services. Services are available on virtually every operating system and MOS (Monitor Operating System) is no exception. MOS has a complete set of services allowing easy access to all of the UT's I/O devices.

REFERENCE

(Reference chapters/page numbers in Kleitz & Goankar texts)

OBJECTIVES

* To understand the benefit of using subroutines and their correct usage.

* To understand subroutines and their affect on the stack.

* To understand MOS services and their usage.

* To become familiar with the following 8085 instructions.

CALL <address> opcode = CD

Put the address of the next instruction following the CALL instruction on the stack. Then load the program counter (PC) with the address contained in the two bytes following the instruction. The first byte following the opcode is the least significant byte of the address, the second byte is the most significant byte of the address. No flags affected. This instruction is 3 bytes in length.

RET opcode = C9

Remove a 16 bit value from the top of the stack and load it into the program counter (PC). No flags affected. This instruction is 1 byte in length.

PROCEDURE

A subroutine is a powerful feature available in both high and low level programming languages. It is sometimes also referred to as a procedure or a function depending on the language. A subroutine should have one entry point at the beginning of the subroutine and one exit point at the end of the subroutine. Every 8085 subroutine should end with the RETurn statement, which exits the subroutine and RETurns back to the instruction following the invoking CALL instruction. Always RETurn from a subroutine, never JMP out of a subroutine.

A subroutine can be called from many different places within a program. Each time the subroutine is invoked a CALL instruction must be used, never JMP into a subroutine. A subroutine can call another subroutine and that subroutine can call another and so forth. The limiting factor for how deep subroutines can be nested is the available stack size. Subroutines are placed sequentially after the end of the Main program.

EXAMPLE 12.0

```
; EXAMPLE PROGRAM TO DEMONSTRATE SUBROUTINES
;
START       EQU   FF01H
LEDS        EQU   11H

            ORG   START
MAIN        MVI   A,0
LOOP        CALL  NOTA        ; CALL SUB TO OUTPUT A NOT
            INR   A
            CPI   3           ; QUIT AFTER THREE ITERATIONS
            JNZ   LOOP
            RST   7           ; END MAIN
;
; THIS SUBROUTINE COMPLEMENTS AND OUTPUTS THE A REG.
;
; INPUT PARAMETERS:      A REG.
; OUTPUT PARAMETERS:     NONE
;
; ALL REGISTERS PRESERVED
;
NOTA        PUSH  PSW         ; PRESERVE A REG
            CMA
            OUT   LEDS        ; OUTPUT A NOT
            POP   PSW         ; RESTORE A REG
            RET
            END               ; END PROGRAM
```

In Example 12.0 above a description of the program comes first, followed by the EQUates, the Main, and then the subroutine. The END statement is a assembler directive to tell the assembler there is no more code to assemble. The Main program ends with the RST 7 instruction which stops executions of the program. When execution stops the stack should be empty.

Most Subroutines require parameters that are passed via the registers and our example is no exception. The A register is used to pass an input parameter to the subroutine NOTA. The subroutine PUSHes this value on the stack to preserve its original value. The subroutine then compliments the A register and outputs it to the LEDs. Once the complemented A register has been output, the original value of A is POPed off the stack and the A register is restored. The subroutine then RETurns to the instruction following the CALL (INR A). The subroutine will be called three times due to the conditional loop in the Main program.

EXERCISE 12.0

1. Enter the machine code of Example 12.0 into the trainer starting at address FF01H.

ADDRESS	CONTENTS	INSTRUCTION
FF01	3E	MVI A,0
FF02	00	Data
FF03	CD	CALL FF0DH
FF04	0D	Address Low
FF05	FF	Address High
FF06	3C	INR A
FF07	FE	CPI 3
FF08	03	Data
FF09	C2	JNZ FF03H
FF0A	03	Address Low
FF0B	FF	Address High
FF0C	FF	RST 7
FF0D	F5	PUSH PSW
FF0E	2F	CMA
FF0F	D3	OUT 11H
FF10	11	I/O Address
FF11	F1	POP PSW
FF12	C9	RET

2. Return the PC to address FF01H and trace the program execution a step at a time (single step). Fill in the register contents and Top Of Stack (S.C.) contents in the blanks below.

INSTR.	P.C.	A	S.P.	S.C.	OPCODE	MNEMONIC
0	FF01	00	FFD4	????	3E	MVI A,0
1	_____	_____	_____	_____	_____	_____
2	_____	_____	_____	_____	_____	_____
3	_____	_____	_____	_____	_____	_____
4	_____	_____	_____	_____	_____	_____
5	_____	_____	_____	_____	_____	_____
6	_____	_____	_____	_____	_____	_____
7	_____	_____	_____	_____	_____	_____
8	_____	_____	_____	_____	_____	_____
9	_____	_____	_____	_____	_____	_____
10	_____	_____	_____	_____	_____	_____
11	_____	_____	_____	_____	_____	_____
12	_____	_____	_____	_____	_____	_____
13	_____	_____	_____	_____	_____	_____
14	_____	_____	_____	_____	_____	_____
15	_____	_____	_____	_____	_____	_____
16	_____	_____	_____	_____	_____	_____
17	_____	_____	_____	_____	_____	_____
18	_____	_____	_____	_____	_____	_____
19	_____	_____	_____	_____	_____	_____

2. Run the program full speed from the address you left off at in Exercise 12.0.1. Examine the registers and S.C. and fill in the blanks below.

P.C.	A	S.P.	S.C.	OPCODE	MNEMONIC
_____	_____	_____	_____	_____	_____

3. What was the maximum stack space used by the program of Exercise 12.0.

4. Replace the PUSH PSW instruction with the NOP instruction. Step the program seven instruction starting at address FF01H.

   A. Does the subroutine return to the same address as in Exercise 12.0.1?

   B. After the seventh instruction, what is the stack condition (underflow, overflow, or balanced?

The program of Example 12.0 demonstrates how a subroutine can be utilized, however if the program is run at full speed it is impossible to see the LEDs change state. The program of Exercise 12.1 solves this problem by adding a DELAY subroutine. This subroutine is called after each output to the LEDs, assuring that each state can be seen. The DELAY subroutine is called from within the SUB subroutine, making it a nested subroutine. Note that the DELAY subroutine is also called from within the Main Program.

EXERCISE 12.1

1. Hand assemble the following assembly language program into machine language in the space provided below.

```
; EXAMPLE PROGRAM TO DEMONSTRATE NESTED SUBROUTINES
;
START       EQU   FF01H
LEDS        EQU   11H

            ORG   START
MAIN        MVI   A,0
LOOP        CALL  NOTA          ; OUTPUT A NOT
            INR   A
            CPI   3             ; QUIT AFTER THREE -
            JNZ   LOOP          ; ITERATIONS
            CALL  DELAY         ; DELAY BEFORE EXIT
            MVI   A,0           ; ZERO OUT LEDS
            OUT   LEDS
            RST   7             ; END MAIN
;
; THIS SUBROUTINE COMPLEMENTS AND OUTPUTS THE A REG.
;
; INPUT PARAMETERS:      A REG.
; OUTPUT PARAMETERS:     NONE
;
; ALL REGISTERS PRESERVED
;
NOTA        PUSH  PSW           ; PRESERVE A REG
            CMA
            OUT   LEDS          ; OUTPUT A NOT
            CALL  DELAY         ; CALL SUB TO DELAY
            POP   PSW           ; RESTORE A REG
            RET
;
; THIS SUBROUTINE EXECUTES A 16 BIT DELAY
;
; INPUT PARAMETERS:      NONE
; OUTPUT PARAMETERS:     NONE
;
; ALL REGISTERS PRESERVED
;
DELAY       PUSH  PSW           ; PRESERVE REGISTERS
            PUSH  H
            LXI   H,0C000H      ; LOAD DELAY FACTOR
LOOP1       DCR   L
            MOV   A,L
            JNZ   LOOP1         ; IS LOW ORDER 0
            DCR   H
            JNZ   LOOP1         ; IS HIGH ORDER 0
            POP   H
            POP   PSW           ; RESTORE REGISTERS
            RET

            END                 ; END PROGRAM
```

ADDRESS	CONTENTS	INSTRUCTION

_____ _____ _____

_____ _____ _____

_____ _____ _____

_____ _____ _____

2. Run the program full speed from address FF01H. Examine the registers and S.C. and fill in the blanks below.

P.C.	A	H.L.	S.P.	S.C.	OPCODE	MNEMONIC
_____	_____	_____	_____	_____	_____	_____

3. What was the maximum stack space used by the program of Exercise 12.1.

4. If the instruction CALL DELAY was placed somewhere in the DELAY subroutine, what stack condition would take place when the program was executed? (Balanced, Underflow, Overflow)

5. If the instruction CALL NOTA was replaced with JMP NOTA, what stack condition would take place when the program was executed? (Balanced, Underflow, Overflow)

USING SERVICES

As previously mentioned MOS has a set of predefined subroutines that we can make use of in our own programs. These service subroutines make it easy to add sophisticated functionality to a user program. Rather then have to remember a different address for each service subroutine, MOS uses one address and a parameter passed in the C register (service number) to indicate which subroutine should be invoked. The address used to access all MOS services is 1000H. When single stepping through a MOS Service Call, the processor will run full speed through the service and automatically break at the address following the CALL. All registers used by a service routine that do not return a value are preserved. For services that display information on the 6, 7 segment numeric LED's, the information will be lost upon termination of the program (Execution of the RST 7 instruction). A list of services and their descriptions can be found in Appendix D.

EXERCISE 12.2

1. Enter the following program into the trainer at starting at address FF01H.

ADDRESS	CONTENTS	INSTRUCTION
FF01	CD	CALL 1000H
FF02	00	Low Address
FF03	10	High Address
FF04	FF	RST 7

2. Reset the PRIMER by pressing the RESET button.

3. Run the program full speed.

4. Describe what happens when this program is running.

5. Reset the PRIMER by pressing the RESET button.

6. Change (spoon feed, see Lab #5) the contents of the C register to 0CH and the contents of the E register to 96H and run the program full speed. Be careful not to reset the PRIMER after spoon feeding the registers.

7. Record the contents of the 8 Digital Out Status LEDs in the spaces provided below (1=on/0=off).

Left ____ ____ ____ ____ ____ ____ ____ ____ Right

EXERCISE 12.3

1. Enter the following program into the trainer at starting at address FF01H.

ADDRESS	CONTENTS	INSTRUCTION
FF01	0E	MVI C,12H
FF02	12	Service Number
FF03	11	LXI D,1F95H
FF04	95	Low Data
FF05	1F	High Data
FF06	CD	CALL 1000H
FF07	00	Low Address
FF08	10	High Address
FF09	DB	IN 12H
FF0A	12	I/O Address
FF0B	FE	CPI 00
FF0C	FE	Data
FF0D	C2	JNZ FF01H
FF0E	01	Low Address
FF0F	FF	High Address
FF10	FF	RST 7

2. Position all 8 switches of dipswitch S1 to the off position.

3. Return the PC to address FF01H and run the program full speed.

4. Record the contents of the 6 numeric 7 segment LED displays in the spaces provided below.

_____ _____ _____ _____ _____ _____

5. Position dipswitch, switch #1 to the on position.

6. Record the contents of the 6 numeric 7 segment LED displays in the spaces provided below.

_____ _____ _____ _____ _____ _____

7. Change the contents of address FF02H from12H to 13H.

8. Position all 8 switches of dipswitch to the off position.

9. Return the PC to address FF01H and run the program full speed.

10. Record the contents of the 6 numeric 7 segment LED displays in the spaces provided below.

_____ _____ _____ _____ _____ _____

11. Stop execution of the program by pressing the RESET button.

12. Change the contents of address FF02H from 13H to 11H, the contents of address FF04 from 95H to 47H, and the contents of address FF05H from 1FH to 00.

13. Return the PC to address FF01H and run the program full speed.

14. Record the contents of the 6 numeric 7 segment LED displays in the spaces provided below.

_____ _____ _____ _____ _____ _____

15. Stop execution of the program by pressing the RST (Reset) button.

QUESTIONS 12.0

1. A _____ is a self contained module which performs a function accessible from different parts of a program.

2. Values passed to and from a subroutine are referred to as _____.

3 A collection of subroutines that are designed to be reused is called a _____.

4. A subroutine that is called from within a subroutine is a _____ subroutine.

5. Predefined subroutines accessible through an operating system are called _____.

6. A subroutine should have one _____ at it's beginning, and one _____ at the subroutines end.

7. A subroutine is always invoked by the _____ instruction and should never be jumped into.

8. Always use the _____ instruction to exit a subroutine, never jump out of a subroutine.

9. After the following instructions were executed what would be the condition of the stack (Balanced, Overflow, Underflow).

```
SUB     NOP
        CALL    SUB
        RET
```

10. List the Service names of the two services used in Exercise 12.2. Refer to Appendix D for Service detail descriptions.

11. List the Service names of the three services used in Exercise 12.3. Refer to Appendix D for Service detail descriptions.

12. Write an 8085 assembly language program (not machine language) using the MOS services that displays on the 7 segment numeric LEDs the current value of the dipswitch indefinitely (infinite loop) until reset. Use service A to read the dipswitch and service 12 to output the dipswitch value to the LEDs.

LAB #13
USING THE ENHANCED MONITOR OPERATING SYSTEM (EMOS)

INTRODUCTION

This lab assumes the user has the EMOS EPROM and an upgrade option installed on the board, and that the user has established communication between the board and the Terminal/PC, as outlined in the EMOS manual.

The Monitor Operating System (MOS) is a powerful software program that provides the user with the tools to enter and edit programs as well as run, test, and debug machine language programs. When the trainer is first turned on, interaction between the Monitor Operating System and the user is through the on-board keypad and display. This mode of operation is referred to as the Keypad/Display mode and was presented in LAB #5. The most productive way to use the PRIMER, however, is through EMOS in which all interaction is through a Terminal/PC. This mode obviously allows more information to be exchanged easier than with the keypad display mode. Additional functions/commands are available through the Terminal/PC mode that are not possible with the Keypad/Display mode.

OBJECTIVES

* Know how to invoke EMOS
* Know the advantages and disadvantages of using EMOS vs. MOS
* View and change the contents of memory using EMOS.
* View and change the register contents using EMOS.
* Fill and Dump memory using EMOS.
* Run a program using EMOS.

PROCEDURE

When the PRIMER is properly connected to the Terminal/PC, invoke EMOS from MOS by pressing "Func" then the "0" key. The LEDs will display **"Func 00"** and the Terminal/PC should now display the following menu:

```
                    EMOS Vx.xx HELP MENU

B --> Bring Block from RAMDISK to Memory
C --> Change register contents
D --> Dump memory contents
E --> Edit memory contents
F --> Fill memory with byte
G --> Go execute program { full speed }
H --> Hex/Decimal math {1st + 2nd, 1st - 2nd}
I --> Input from I/O port
L --> List memory contents using mnemonics
M --> Move section of memory
O --> Output to I/O port
R --> display Register contents
S --> MOS Service call
T --> Trace program execution
W --> Write memory to RAMDISK
< --> hex download from trainer to host
> --> hex upload to trainer from host
? --> display this help menu
```

If the menu is not displayed type "?" and the menu should appear. The above Help Menu can be displayed at any time from the EMOS minus "-" prompt by typing a "?".

Each of the EMOS commands are invoked by typing a single character at the EMOS prompt. EMOS will prompt the user for any necessary command input and checks the input to be sure it is of the proper form. When an "ADDRESS" is requested, a number of up to 4 HEX digits should be entered. When a "BLOCK" is requested, a number of up to 3 DEC (decimal) digits should be entered. The <ESC> key or bad input will abort the issued command and cause a "?" to be displayed. A complete description of all the EMOS commands is given in the EMOS Manual. Several of these commands and their descriptions are covered in detail in this lab.

VIEWING AND CHANGING MEMORY CONTENTS

Using The Dump Command

The "D" command shows memory across the screen in Hex and ASCII both. ASCII stands for the American Standard Code for Information Interchange. The ASCII codes 00 through 7F in hex define control and printable characters. Each character has a unique ASCII code that references that character. For instance 31 Hex is the ASCII code for the character "1". If you send the value 31 Hex to most printers, they would print a one. Below is an example of an arbitrary Dump of memory.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
8F01 30 31 0D 0A 41 42 C3 FF 61 62 23 24 2A 20 00 3F  01  AB..ab#$*  ?
```

In this example 8F01 is the starting address of the Dump line. Sixteen memory locations starting at 8F01 through 8F10 are displayed in Hexidecimal form, followed by the same sixteen addresses displayed in ASCII form. ASCII control codes such as carriage return (0Dh) and line feed (0Ah) display a space character. Memory values above 7F display a period. All other values display their ASCII character representation. This standard dump format is featured on most microprocessor development systems. It is very useful for displaying a whole block of memory that may contain data, variables, or ASCII strings.

Steps for Dumping Memory

1. Type the letter "D" at the minus prompt.
2. Enter the desired starting address in Hex. If the address is less than four digits press the <ENTER> key.
3. Enter the number of bytes from 1 to FFF in Hex. A minimum of 16 bytes is always displayed. Pressing any key will pause the display. Pressing the <ESC> key at this time will abort the Dump command or any other key will continue.
4. After completion of the Dump command the minus prompt will be displayed, indicating EMOS is ready for another command.

Using The Edit Command

The "E" command displays memory a byte at a time with its associated address and allows the contents of the location to be optionally changed. Press the <ENTER> key to view the next sequential address' memory contents without changing the contents of the preceding memory location. Typing a byte value in Hex (00 - FF) will replace the contents of the currently displayed address with the Hex value typed. Typing a minus sign followed by a single Hex digit (0 - F) will subtract the value of the digit from the current address, allowing you to back up, up to 15 memory locations. Pressing the <ESC> key at any time will return you to the EMOS minus prompt. The Edit command is particularly useful for entering and editing programs.

Steps for Editing Memory

1. Type the letter "E" at the minus prompt.
2. Enter the desired starting address in Hex. If the address is less than four digits press the <ENTER> key after typing the digits.
3. Change the contents of each memory location as required. To change the memory contents, type up to two Hex digits.
4. After you finish Editing the memory contents, press the <ESC> key to return to the EMOS minus prompt.

Using The Fill Command

The "F" command is used to Fill a section of memory with a particular byte. This command comes in handy for determining what memory locations are being altered or accessed. This is done by filling the suspect memory section with a known byte. After executing a program the suspect memory section is Dumped to see which locations have changed. The Fill command requires a starting address, the number of bytes (memory locations) to Fill, and the byte with which to Fill the memory locations. This command will obviously have no effect when trying to Fill EPROM locations.

Steps For Filling Memory

1. Type the letter "F" at the minus prompt.
2. Enter the desired starting address in Hex. If the address is less than four digits press the <ENTER> key.
3. Enter the number of bytes from 1 to FFF in Hex. If the address is less than four digits press the <ENTER> key after typing the digits.
4. Enter the Fill byte (0 - FF).
5. After completion of the FILL command the minus prompt will be diplayed, indicating EMOS is ready for another command.

Using The List Command

The "L" command disassembles the memory contents, displaying the memory contents as 8085 mnemonics with their associated addresses. This command is very useful in verifying the instructions you entered using the Edit command. The List command requires a starting address. This address must be that of an opcode. If the address of an operand or data is given, the list command will assume this address contains an opcode and will disassemble it and the contents of subsequent memory locations, giving misleading results. The List command will disassemble sixteen 8085 instructions and will pause. Pressing the <ESC> key will return you to the minus prompt. Pressing any other key will continue the disassembling process. If an invalid (not defined) opcode is encountered, three question marks are displayed. The availability of the List command is one of the benifits to using EMOS in place of MOS.

Steps For Listing Memory

1. Type the letter "L" at the minus prompt.
2. Enter the desired starting address in Hex. If the address is less than four digits press the <ENTER> key after typing the digits.
3. Press any key to continue disassembling, or press the Escape key to return to the EMOS minus prompt.

LOADING A PROGRAM INTO MEMORY

In Lab #5 we loaded a program into memory using MOS. In this lab we will load the same program using EMOS. The program is as follows:

```
ADDRESS     DATA        INSTRUCTION
8F01        DB          IN   12
8F02        12
8F03        D3          OUT  11
8F04        11
8F05        C3          JMP  8F01
8F06        01
8F07        8F
```

Use the following steps to enter the above program using EMOS.

1. Use the "F" command to fill 16 memory locations, starting at address 8F01, with 0. This step is not mandatory in loading a program, but makes later verification of the program easier.
2. Type "E" to invoke the Edit command and specify a starting address of 8FO1.
3. At address 8FO1 enter the Hex value DB from the first line of the above program.
4. Continue entering the Data Instructions in their respective addresses.
5. Press the <ESC> key to return to the EMOS minus prompt.
6. Use the "L" command to list the contents of memory starting at memory address 8F01.
7. Verify that each memory location contains the correct value (compare instructions and operands). If a mistake is found repeat from step #1.

VIEWING AND CHANGING REGISTER CONTENTS

Viewing the register contents with MOS required that each register be viewed independently. EMOS offers the luxury of viewing not only all the registers, but the top of stack and the memory contents pointed to by the PC with its associated mnemonic, all at the same time. EMOS displays each flag of the flag register individually so the user doesn't need to decode a hex flag value into binary to see which flags are set or clear. EMOS is a more efficient and robust training environment than is MOS.

When using MOS to change register contents, each change affected a register pair. With EMOS each 8 bit register is changed and displayed individually and not as a register pair.

Using The Register Command

The "R" command displays the flags, all the registers, the contents of the top of stack, and the contents of memory pointed to by the PC with its associated mnemonic. Each flag is given an individual bit, making it easy to see individual flag status. All 8 bit registers are displayed individually and not as register pairs. The data in memory pointed to by the PC is displayed and referred to as an OPCODE. The memory contents is referred to as opcode because it is assumed that any location the PC points to should contain an opcode, even though it may contain an operand or data. The format for displaying the registers in EMOS is shown below.

```
SZxAxPxC  A   B   C   D   E   H   L     TS     SP     PC    OPCODE   MNEMONIC
00000000  00  00  00  00  00  00  00    0000   0000   0000    00     NOP
```

The flags read from the left are:	S	Sign flag
	Z	Zero flag
	x	Undefined
	A	Auxiliary Carry flag
	x	Undefined
	P	Parity flag
	x	Undefined
	C	Carry flag

The 8 bit registers from the left are:	A	Accumulator register
	B	B general purpose register
	C	C general purpose register
	D	D general purpose register
	E	E general purpose register
	H	H general purpose register
	L	L general purpose register
The 16 bit memory value:	TS	Top of Stack contents
The 16 bit registers from the left:	SP	Stack Pointer register
	PC	Program Counter register
The 8 bit memory value:	OPCODE	Value pointed to by PC
	MNEMONIC	Instruction Abbreviation

To display the registers, simply type the letter "R" at the EMOS minus prompt.

Using The Change Command

The "C" command allows the modification of the flags, registers, the top of stack, and the memory location pointed to by the PC. Each register can be individually changed with EMOS unlike with MOS. Flags, although individually displayed, must be changed as a Hex 8 bit value. The Change command first prints the current register contents, and then prompts the user with the following:

```
SELECT REG.  [F, A, B, C, D, H, L, T, S, P, O]..
```

Select "F" to change the flags, "A" - "L" to change the 8 bit registers, "T" for the top of stack, "S" for the stack pointer, and "O" to change the memory contents pointed to by the PC. After entering one of the above letters, the user is then prompted for the new hex value for the selected register/memory location. Remember TS, SP, and PC all contain 16 bit values and all other selections contain 8 bit values. The Change command will display the register contents again for verification, after the change has been made and before returning to the EMOS minus prompt.

RUNNING A PROGRAM

To run a program using EMOS, the Go command is used. Using the Go command, the user can enter an optional starting address and/or optional breakpoint address. Then when the Go command prompts for the starting address and the <ENTER> key is pressed without typing a number, execution will start at the current PC value. The steps required to run a program are listed below.

1. Type the letter "G" to invoke the Go command.
2. Enter the starting address (in hex) of the instruction to be executed. If the address is less than four digits press the <ENTER> key after typing the digits.
3. At this time EMOS will prompt you for a breakpoint address. If a breakpoint address is not required, just press the <ENTER> key. (Breakpoints are not discussed in this lab).
4. The Terminal/PC will indicate that the program is running. To stop execution of the program and return to the EMOS minus prompt, press a key at the Terminal/PC.

Verify that the program entered in the LOADING A PROGRAM section, located at address 8F01 is still intact. If not, reenter the program. To execute the program use the Go command selecting the starting address of 8F01 with no breakpoint as mentioned above. The Terminal/PC will show the message "`RUNNING.. HIT ANY KEY TO STOP`". Test the program by flipping the DIP switches and you should see that each switch affects its corresponding output LED.

When you press a key, the program will stop, the registers will be displayed (just as if the "R" command had been executed) and the minus prompt will appear.

EXERCISE 13.0

1. Enter the following Hex bytes into the memory locations shown below and verify the memory contents for accuracy.

```
8F01  11
8F02  05
8F03  04
8F04  2E
8F05  07
8F06  63
8F07  24
8F08  3E
8F09  02
8F0A  47
8F0B  0F
8F0C  48
8F0D  0C
8F0E  FF
```

2. The addresses 8F01 to 8F0E now contain the machine code of a short program. Run this program starting at address 8F01 (the first opcode of the program) and examine the register contents after its execution. To run the program, use the Go command with a starting address of 8F01 and no breakpoint. After running the program, fill in the register and flag contents below.

```
PC_______ B___ C___ D___ E___ H___ L___ A___ Sf__ Zf__ ACf__ Pf__ Cf__
```

3. Using the List command, disassemble the program of exercise question #1 starting at 8F01. Fill in the mnemonics and operands below. The first mnemonic with operands is given. There are 10 opcodes (instructions) in the program and each instruction can have 1, 2, or 3 bytes depending on the operands.

```
1,    LXI   D,0405H
2.    _________________
3.    _________________
4.    _________________
5.    _________________
6.    _________________
7.    _________________
8.    _________________
9.    _________________
10.   _________________
```

4. Using the "D" command, Dump 16 memory locations starting at address 8F01. Fill in the associated blanks below.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   012345678ABCDEF
8F01 __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __   ________________
```

5. Using the "F" command Fill 16 memory locations starting at address 8F01 with the ASCII character "+" (plus). The Hex value of "+" is 2B. To verify the Fill command, Dump 16 memory locations beginning at address 8F01 and fill in the associated blanks below.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   012345678ABCDEF
8F01 __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __   ________________
```

QUESTIONS 13.0

1. List two advantages and two disadvantages of using EMOS instead of MOS.

2. List the ASCII codes in hex for the string "8085 PRIMER" . (See ASCII table in Appendix A)

3.. What EMOS command could be used to verify that the ASCII string of question #2 is correct?

4. ASCII is an acronym for _____.

5. What ASCII character is used as the EMOS prompt?

6. What ASCII character is entered to display the EMOS Help Menu?

7. What button or key is pressed to stop executing a program and return to the EMOS prompt?

8. What button or key is pressed to stop Editing and return to the EMOS prompt?

9. To back up 10 bytes when using the Edit command (without exiting), you would enter _____ .

10. If a data entry error is made using an EMOS command, what key would abort the command and return you to the EMOS prompt?

11. Are the registers initialized to the same values for EMOS as they are for MOS? If not, what register values are different?

12. If a program is entered in MOS can it be executed in EMOS? If a program is entered in EMOS can it be executed in MOS?

13. If a register is changed in MOS does it remain changed in EMOS?

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 (LEDs)
PORT B	12 H	INPUT PORT (DIPSWITCH)
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

Instruction Set Encyclopedia

8085A CPU Instructions In Operation Code Sequence

OP Code	Mnemonic	OP Code	Mnemonic	OP Code	Mnemonic	OP Code	Mnemonic
00	NOP	40	MOV B,B	80	ADD B	C0	RNZ
01	LXI B,D16	41	MOV B,C	81	ADD C	C1	POP B
02	STAX B	42	MOV B,D	82	ADD D	C2	JNZ Adr
03	INX B	43	MOV B,E	83	ADD E	C3	JMP Adr
04	INR B	44	MOV B,H	84	ADD H	C4	CNZ Adr
05	DCR B	45	MOV B,L	85	ADD L	C5	PUSH B
06	MVI B,D8	46	MOV B,M	86	ADD M	C6	ADI D8
07	RLC	47	MOV B,A	87	ADD A	C7	RST 0
08	-	48	MOV C,B	88	ADC B	C8	RZ
09	DAD B	49	MOV C,C	89	ADC C	C9	RET
0A	LDAX B	4A	MOV C,D	8A	ADC D	CA	JZ Adr
0B	DCX B	4B	MOV C,E	8B	ADC E	CB	-
0C	INR C	4C	MOV C,H	8C	ADC H	CC	CZ Adr
0D	DCR C	4D	MOV C,L	8D	ADC L	CD	CALL Adr
0E	MVI C,D8	4E	MOV C,M	8E	ADC M	CE	ACI D8
0F	RRC	4F	MOV C,A	8F	ADC A	CF	RST 1
10	-	50	MOV D,B	90	SUB B	D0	RNC
11	LXI D,D16	51	MOV D,C	91	SUB C	D1	POP D
12	STAX D	52	MOV D,D	92	SUB D	D2	JNC Adr
13	INX D	53	MOV D,E	93	SUB E	D3	OUT D8
14	INR D	54	MOV D,H	94	SUB H	D4	CNC Adr
15	DCR D	55	MOV D,L	95	SUB L	D5	PUSH D
16	MVI D,D8	56	MOV D,M	96	SUB M	D6	SUI D8
17	RAL	57	MOV D,A	97	SUB A	D7	RST 2
18	-	58	MOV E,B	98	SBB B	D8	RC
19	DAD D	59	MOV E,C	99	SBB C	D9	-
1A	LDAX D	5A	MOV E,D	9A	SBB D	DA	JC Adr
1B	DCX D	5B	MOV E,E	9B	SBB E	DB	IN D8
1C	INR E	5C	MOV E,H	9C	SBB H	DC	CC Adr
1D	DCR E	5D	MOV E,L	9D	SBB L	DD	-
1E	MVI E,D8	5E	MOV E,M	9E	SBB M	DE	SBI D8
1F	RAR	5F	MOV E,A	9F	SBB A	DF	RST 3
20	RIM	60	MOV H,B	A0	ANA B	E0	RPO
21	LXI H,D16	61	MOV H,C	A1	ANA C	E1	POP H
22	SHLD Adr	62	MOV H,D	A2	ANA D	E2	JPO Adr
23	INX H	63	MOV H,E	A3	ANA E	E3	XTHL
24	INR H	64	MOV H,H	A4	ANA H	E4	CPO Adr
25	DCR H	65	MOV H,L	A5	ANA L	E5	PUSH H
26	MVI H,D8	66	MOV H,M	A6	ANA M	E6	ANI D8
27	DAA	67	MOV H,A	A7	ANA A	E7	RST 4
28	-	68	MOV L,B	A8	XRA B	E8	RPE
29	DAD H	69	MOV L,C	A9	XRA C	E9	PCHL
2A	LHLD Adr	6A	MOV L,D	AA	XRA D	EA	JPE Adr
2B	DCX H	6B	MOV L,E	AB	XRA E	EB	XCHG
2C	INR L	6C	MOV L,H	AC	XRA H	EC	CPE Adr
2D	DCR L	6D	MOV L,L	AD	XRA L	ED	-
2E	MVI L,D8	6E	MOV L,M	AE	XRA M	EE	XRI D8
2F	CMA	6F	MOV L,A	AF	XRA A	EF	RST 5
30	SIM	70	MOV M,B	B0	ORA B	F0	RP
31	LXI SP,D16	71	MOV M,C	B1	ORA C	F1	POP PSW
32	STA Adr	72	MOV M,D	B2	ORA D	F2	JP Adr
33	INX SP	73	MOV M,E	B3	ORA E	F3	DI
34	INR M	74	MOV M,H	B4	ORA H	F4	CP Adr
35	DCR M	75	MOV M,L	B5	ORA L	F5	PUSH PSW
36	MVI M,D8	76	HLT	B6	ORA M	F6	ORI D8
37	STC	77	MOV M,A	B7	ORA A	F7	RST 6
38	-	78	MOV A,B	B8	CMP B	F8	RM
39	DAD SP	79	MOV A,C	B9	CMP C	F9	SPHL
3A	LDA Adr	7A	MOV A,D	BA	CMP D	FA	JM Adr
3B	DCX SP	7B	MOV A,E	BB	CMP E	FB	EI
3C	INR A	7C	MOV A,H	BC	CMP H	FC	CM Adr
3D	DCR A	7D	MOV A,L	BD	CMP L	FD	-
3E	MVI A,D8	7E	MOV A,M	BE	CMP M	FE	CPI D8
3F	CMC	7F	MOV A,A	BF	CMP A	FF	RST 7

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

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

Adr = 16-bit address

APPENDIX C

ASCII CHARACTER SET

ASCII CHARACTER SET FOR THE PRIMER TRAINER

ASCII is an acronym for American Standard Code for Information Interchange. ASCII provides a standard code for using numbers to represent characters. As you know a computer can only store and manipulate numbers, so ASCII provides the computer a way to store and manipulate text. Each character is represented by a single byte, allowing for up to 256 unique codes. For example if 30H was sent to a printer it would print the character 0. There is a variety of computer peripherals that use ASCII, i.e. keyboards, printers, and terminals. The table below contains a subset of the ASCII character set.

HEX CODE	CHARACTER	HEX CODE	CHARACTER
20	SPACE (BLANK)	41	A
21	!	42	B
22	“	43	C
23	#	44	D
24	$	45	E
25	%	46	F
26	&	47	G
27	‘	48	H
28	(	49	I
29	)	4A	J
2A	*	4B	K
2B	+	4C	L
2C	,	4D	M
2D	-	4E	N
2E	.	4F	O
2F	/	50	P
30	0	51	Q
31	1	52	R
32	2	53	S
33	3	54	T
34	4	55	U
35	5	56	V
36	6	57	W
37	7	58	X
38	8	59	Y
39	9	5A	Z
3A	:	5B	[
3B	;	5C	\
3C	<	5D	]
3D	=	5E	^
3E	>	5F	_
3F	?	60	`
40	@		