ALTAIR 8800 OPERATOR'S MANUAL

TABLE OF CONTENTS

MITS INC.
"Creative Electronics"
P.O. BOX 8636
ALBUQUERQUE, NEW MEXICO 87108

PRINTED IN U.S.A.

6328 LINN, N.E., P.O. BOX 8636, ALBUQUERQUE, N.M. 87108 U.S.A.
505/265-7553

PART 1 INTRODUCTION

Remarkable advances in semiconductor technology have made possible the development of the *ALTAIR 8800*, the most economical computer ever and the first available in both kit and assembled form. The heart of the *ALTAIR 8800* is Intel Corporation's Model 8080 Microcomputer, a complete Central Processing Unit on a single silicon chip. Fabricated with N-channel large scale integrated circuit (LSI) metal-oxide-semiconductor (MOS) technology, Intel's 8080 Microcomputer on a chip represents a major technological breakthrough.

This operating manual has been prepared to acquaint both the novice and the experienced computer user in the operation of the *ALTAIR 8800*. The computer has 78 machine language instructions and is capable of performing several important operations not normally available with conventional minicomputers. After reading this manual, even a novice will be able to load a program into the *ALTAIR 8800*.

Users of the *ALTAIR 8800* include persons with a strong electronics background and little or no computer experience and persons with considerable programming experience and little or no electronics background. Accordingly, this manual has been prepared with all types of users in mind. Part 1 of the manual prepares the user for better understanding computer terminology, technology, and operation with an introduction to conventional and electronic logic, a description of several important number systems, a discussion of basic programming, and a discourse on computer languages.

Parts 2 and 3 in the manual describe the organization and operation of the *ALTAIR 8800*. Emphasis is placed on those portions of the computer most frequently utilized by the user. Finally, Part 4 of the manual presents a detailed listing of the *ALTAIR 8800's* 78 instructions. An Appendix condenses the instructions into a quick reference listing.

Even if you have little or no experience in computer operation and organization, a careful reading of this manual will prepare you for operating the *ALTAIR 8800*. As you gain experience with the machine, you will soon come to understand its truly incredible versatility and data processing capability. Don't be discouraged if the manual seems too complicated in places. Just remember that a computer does only what its programmer instructs it to do.

A. LOGIC

George Boole, a ninteenth century British mathematician, made a detailed study of the relationship between certain fundamental logical expressions and their arithmetic counterparts. Boole did not equate mathematics with logic, but he did show how any logical statement can be analyzed with simple arithmetic relationships. In 1847, Boole published a booklet entitled Mathematical Analysis of Logic and in 1854 he published a much more detailed work on the subject. To this day, all practical digital computers and many other electronic circuits are based upon the logic concepts explained by Boole.

Boole's system of logic, which is frequently called Boolean algebra, assumes that a logic condition or statement is either true or false. It cannot be both true and false, and it cannot be partially true or partially false. Fortunately, electronic circuits are admirably suited for this type of dual-state operation. If a circuit in the ON state is said to be true and a circuit in the OFF state is said to be false, an electronic analogy of a logical statement can be readily synthesized.

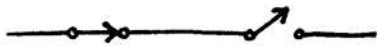
With this in mind, it is possible to devise electronic equivalents for the three basic logic statements: AND, OR and NOT. The AND statement is true if and only if either or all of its logic conditions are true. A NOT statement merely reverses the meaning of a logic statement so that a true statement is false and a false statement is true.

It's easy to generate a simple equivalent of these three logic statements by using on-off switches. A switch which is ON is said to be true while a switch which is OFF is said to be false. Since a switch which is OFF will not pass an electrical current, it can be assigned a numerical value of 0. Similarly, a switch which is ON does pass an electrical current and can be assigned a numerical value of 1.

We can now devise an electronic equivalent of the logical AND statement by examining the various permutations for a two condition AND statement:

CONDITIONS (Inputs)	CONCLUSION (Output)
1. True AND True	True
2. True AND False	False
3. False AND True	False
4. False AND False	False

The electronic ON-OFF switch equivalent of these permutations is simply:

CONDITIONS (ON-OFF)	CONCLUSION (OUTPUT)
1. 	1
2. 	0
3. 	0
4. 	0

Similarly, the numerical equivalents of these permutations is:

CONDITIONS (Inputs)	CONCLUSION (Output)
1. 1 AND 1	1
2. 1 AND 0	0
3. 0 AND 1	0
4. 0 AND 0	0

Digital design engineers refer to these table of permutations as truth tables. The truth table for the AND statement with two conditions is usually presented thusly:

A	B	OUT
1	1	1
0	1	0
1	0	0
0	0	0

FIGURE 1-1. AND Function Truth Table

It is now possible to derive the truth tables for the OR and NOT statements, and each is shown in Figures 1-2 and 1-3 respectively.

A	B	OUT
1	1	1
0	1	1
1	0	1
0	0	0

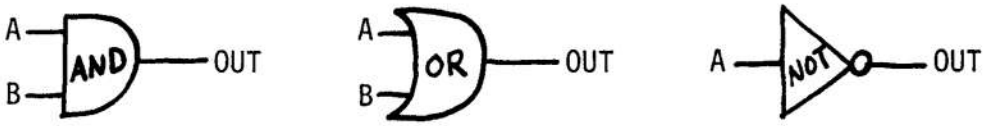
FIGURE 1-2. OR Function Truth Table

A	OUT
1	0
0	1

FIGURE 1-3. NOT Function Truth Table

B. ELECTRONIC LOGIC

All three of the basic logic functions can be implemented by relatively simple transistor circuits. By convention, each circuit has been assigned a symbol to assist in designing logic systems. The three symbols along with their respective truth tables are shown in Figure 1-4.



A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

A	OUT
0	1
1	0

FIGURE 1-4. The Three Main Logic Symbols

The three basic logic circuits can be combined with one another to produce still more logic statement analogies. Two of these circuit combinations are used so frequently that they are considered basic logic circuits and have been assigned their own logic symbols and truth tables. These circuits are the NAND (NOT-AND) and the NOR (NOT-OR). Figure 1-5 shows the logic symbols and truth tables for these circuits.

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

FIGURE 1-5. The NAND and NOR Circuits

Three or more logic circuits make a logic system. One of the most basic logic systems is the EXCLUSIVE-OR circuit shown in Figure 1-6.

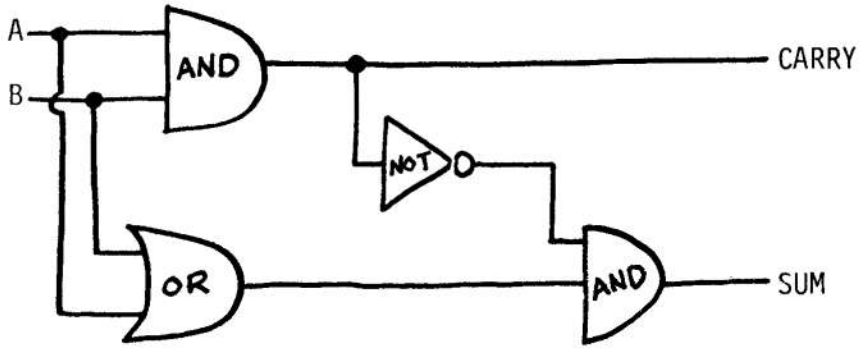


FIGURE 1-6. The EXCLUSIVE-OR Circuit

The EXCLUSIVE-OR circuit can be used to implement logical functions, but it can also be used to add two input conditions. Since electronic logic circuits utilize only two numerical units, 0 and 1, they are compatible with the binary number system, a number system which has only two digits. For this reason, the EXCLUSIVE-OR circuit is often called a binary adder.

Various combinations of logic circuits can be used to implement numerous electronic functions. For example, two NAND circuits can be connected to form a bistable circuit called a flip-flop. Since the flip-flop changes state only when an incoming signal in the form of a pulse arrives, it acts as a short term memory element. Several flip-flops can be cascaded together to form electronic counters and memory registers.

Other logic circuits can be connected together to form monostable and astable circuits. Monostable circuits occupy one of two states unless an incoming pulse is received. They then occupy an opposite state for a brief time and then resume their normal state. Astable circuits continually switch back and forth between two states.

C. NUMBER SYSTEMS

Probably because he found it convenient to count with his fingers, early man devised a number system which consisted of ten digits. Number systems, however, can be based on any number of digits. As we have already seen, dual-state electronic circuits are highly compatible with a two digit number system, and its digits are termed bits (binary digits). Systems based upon eight and sixteen are also compatible with complex electronic logic systems such as computers since they provide a convenient shorthand method for expressing lengthy binary numbers.

D. THE BINARY SYSTEM

Like virtually all digital computers, the *ALTAIR 8800* performs nearly all operations in binary. A typical binary number processed by the computer incorporates 8-bits and may appear as: 10111010. A fixed length binary number such as this is usually called a word or byte, and computers are usually designed to process and store a fixed number of words (or bytes).

A binary word like 10111010 appears totally meaningless to the novice. But since binary utilizes only two digits (bits), it is actually much simpler than the familiar and traditional decimal system. To see why, let's derive the binary equivalents for the decimal numbers from 0 to 20. We will do this by simply adding 1 to each successive number until all the numbers have been derived. Counting in any number system is governed by one basic rule: Record successive digits for each count in a column. When the total number of available digits has been used, begin a new column to the left of the first and resume counting.

Counting from 0 to 20 in binary is very easy since there are only two digits (bits). The binary equivalent of the decimal 0 is 0. Similarly, the binary equivalent of the decimal 1 is 1. Since both available bits have now been used, the binary count must incorporate a new column to form the binary equivalent for the decimal 2. The result is 10. (Incidentally, ignore any resemblance between binary and decimal numbers. Binary 10 is <u>not</u> decimal 10!) The binary equivalent of the decimal number 3 is 11. Both bits have been used again, so a third column must be started to obtain the binary equivalent for the decimal number 4 (100). You should now be able to continue counting and derive all the remaining binary equivalents for the decimal numbers 0 to 20:

DECIMAL	BINARY
0	0
1	1
2	10
3	11

DECIMAL	BINARY
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100

A simple procedure can be used to convert a binary number into its decimal equivalent. Each bit in a binary number indicates by which power of two the number is to be raised. The sum of the powers of two gives the decimal equivalent for the number. For example, consider the binary number 10011:

$$10011 = [(1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)]$$

$$= [(16) + (0) + (0) + (2) + (1)]$$

$$= 19$$

E. THE OCTAL SYSTEM

Since the binary system has only two bits, it doesn't take long to accumulate a long string of 0s and 1s. For example, a six-digit decimal number requires 19 bits.

Lengthy binary numbers can be simplified by dividing them into groups of three bits and assigning a decimal equivalent to each 3-bit group. Since the highest 3-bit binary number corresponds to the decimal 7, eight combinations of 0s and 1s are possible (0-7).

The basic *ALTAIR 8800* accepts a binary input, and any binary number loaded into the machine can be simplified into octal format. Of course the octal numbers must be changed back to binary for entry into the computer, but since only eight bit patterns are involved the procedure is both simple and fast. A typical binary instruction for the *ALTAIR 8800* is: 11101010. This instruction can be converted to octal by first dividing the number into groups of three bits beginning with the least significant bit: 11 101 010. Next, assign the decimal equivalent to each of the three bit patterns:

11	101	010
3	5	2

Therefore, 11 101 010 in binary corresponds to 352 in octal. To permit rapid binary to octal conversion throughout the remainder of this manual, most binary numbers will be presented as groups of three bits.

F. COMPUTER PROGRAMMING

As will become apparent in Part 2, the Central Processing Unit (CPU) of a computer is essentially a network of logic circuits and systems whose interconnections or organization can be changed by the user. The computer can therefore be thought of as a piece of variable hardware. Implementation of variations in a computer's hardware is achieved with a set of programmed instructions called software.

The software instructions for the *ALTAIR 8800* must be loaded into the machine in the form of sequential 8-bit words called machine language. This and other more advanced computer languages will be discussed later.

The basics of computer programming are quite simple. In fact, often the most difficult part of programming is defining the problem you wish to solve with the computer. Below are listed the three main steps in generating a program:

1. Defining the Problem
2. Establishing an Approach
3. Writing the Program

Once the problem has been defined, an approach to its solution can be developed. This step is simplified by making a diagram which shows the orderly, step-by-step solution of the problem. Such a diagram is called a flow diagram. After a flow diagram has been made, the various steps can be translated into the computer's language. This is the easiest of the three steps since all you need is a general understanding of the instructions and a list showing each instruction and its machine language equivalent.

The *ALTAIR 8800* has an extensive programming capability. For example, a program can cause data to be transferred between the computer's memory and the CPU. The program can even cause the computer to make logical decisions. For example, if a specified condition is met, the computer can jump from one place in the program to any other place and continue program execution at the new place. Frequently used special purpose programs can be stored in the computer's memory for later retrieval and use by the main program. Such a special purpose program is called a

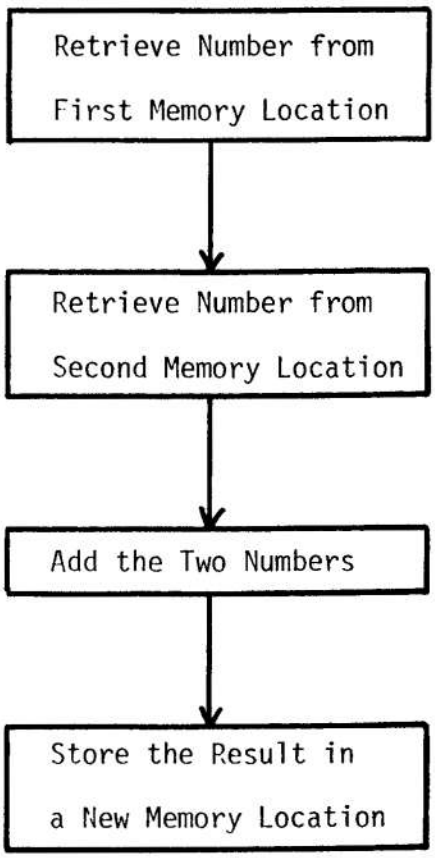
subroutine. The *ALTAIR 8800* instructions are described in detail in Part 4 of this manual.

G. A SIMPLE PROGRAM

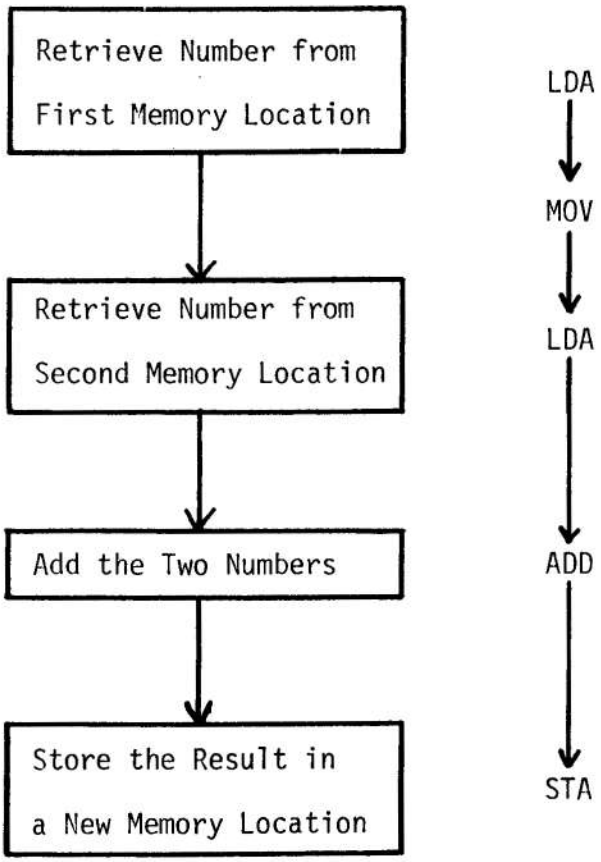
Assume you wish to use the *ALTAIR 8800* to add two numbers located at two different memory locations and store the result elsewhere in the memory. Of course this is a very simple problem, but it can be used to illustrate several basic programming techniques. Here are the steps used in generating a program to solve this problem:

1. Define the Problem--Add two numbers located in memory and store the result elsewhere in memory.

2. Establish an Approach--A flow diagram can now be generated:



3. Write the Program--Translating the flow diagram into a language or format suitable for use by the computer may seem complicated at first. However, a general knowledge of the computer's organization and operation makes the job simple. In this case, the four part flow diagram translates into five separate instructions:



These instructions may seem meaningless now, but their meaning and application will become much clearer as you proceed through this manual. For example, the need for the extra instruction (MOV) will become more obvious after you learn that the computer must temporarily store the first number retrieved from memory in a special CPU memory called a register. The first number is stored in the register until it can be added to the second number.

H. COMPUTER LANGUAGES

The software for any computer must be entered into the machine in the form of binary words called machine language. Machine language programs are generally written with the help of mnemonics which correspond to the bit patterns for various instructions. For example, 10 000 111 is an add instruction for the *ALTAIR 8800* and the corresponding mnemonic is ADD A. Obviously the mnemonic ADD A is much more convenient to remember than its corresponding machine language bit pattern.

Ultimately, however, the machine language bit pattern for each instruction must be entered into the computer one step at a time. Some instructions may require more than one binary word. For example, an *ALTAIR 8800* instruction which references a memory address such as JMP requires one word for the actual instruction and two subsequent words for the memory address.

Machine language programs are normally entered into the *ALTAIR 8800* by means of the front panel switches. A computer terminal can be used to send the mnemonics signal to the computer where it is converted into machine language by a special set of instructions (software) called an assembler.

Even more flexibility is offered by a highly complex software package called a compiler which converts higher order mnemonics into machine language. Higher order mnemonics are a type of computer language shorthand which automatically replace as many as a dozen or more machine language instructions with a single, easily recognized mnemonic. Advanced computer languages such as FORTRAN, BASIC, COBAL, and others make use of a compiler.

The higher computer languages provide a great deal of simplification when writing computer programs, particularly those that are lengthy. They are also very easy to remember. The potential versatility of machine language pro-

gramming should not be underestimated, however, and an excellent way to realize the full potential of a higher language is to learn to apply machine language.

PART 2 ORGANIZATION OF THE ALTAIR 8800

A block diagram showing the organization of the *ALTAIR 8800* is shown in Figure 2-1. It is not necessary to understand the detailed electronic operation of each part of the computer to make effective use of the machine. However, a general understanding of each of the various operating sections is important.

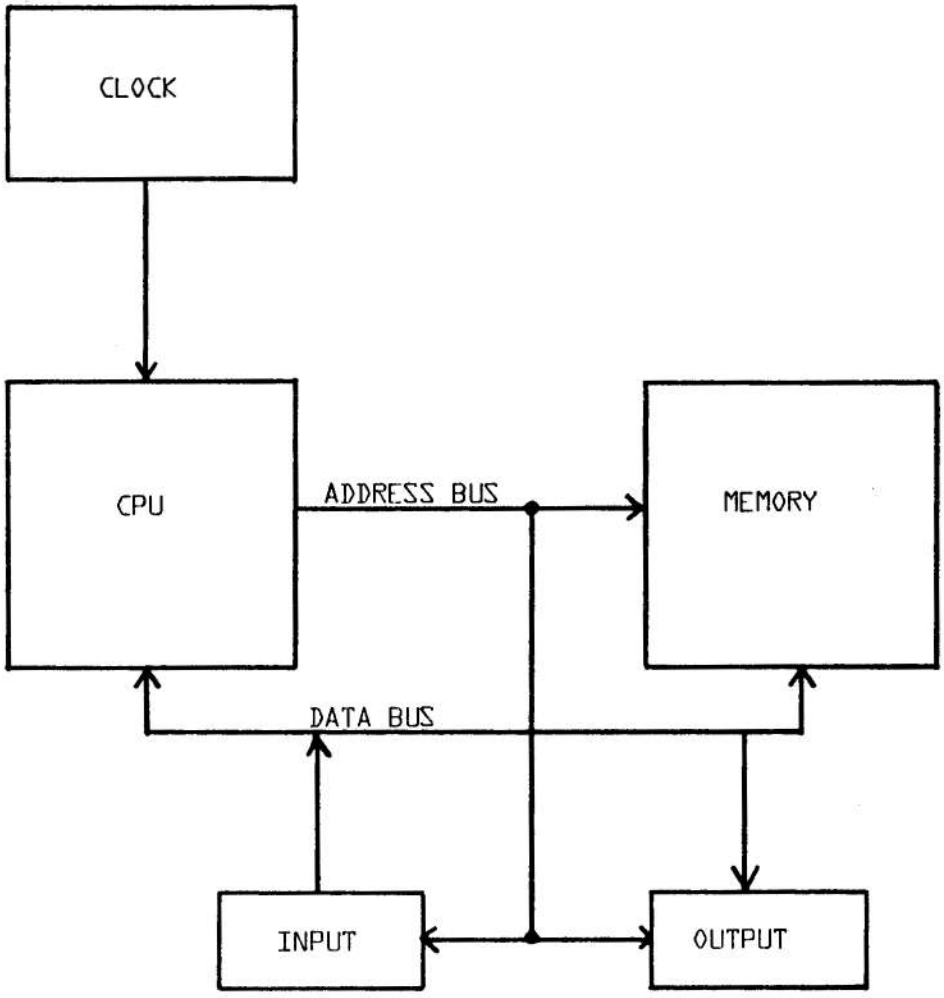


FIGURE 2-1

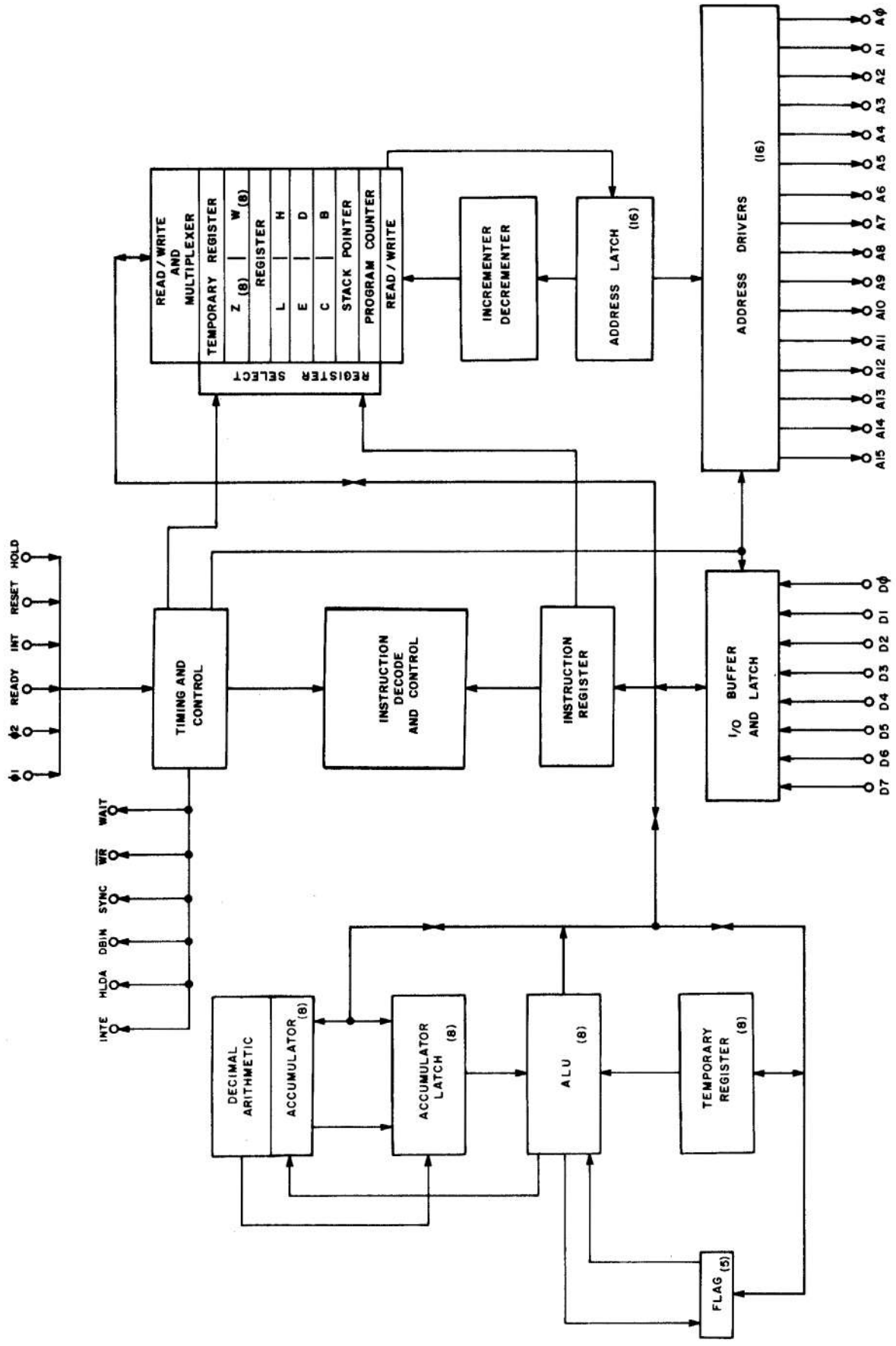


FIGURE 2-2. CPU Diagram

A. CENTRAL PROCESSING UNIT (CPU)

The Central Processing Unit (CPU) performs all arithmetic calculations, makes all logical decisions, controls access to the computer by input and output devices, stores and retrieves data from the memory, and coordinates the orderly execution of a program. The CPU is quite literally the heart of the computer.

Of course it is important to remember that the CPU is only as intelligent as the programmer, for the CPU must be instructed in precise terms just how to perform a particular operation. But since the CPU in the *ALTAIR 8800* can execute a complete instruction cycle in only 2 microseconds*, the computer can solve a highly complex problem in an incredibly brief time. In fact, the *ALTAIR 8800* can execute a six instruction addition program approximately 30,000 times in one second.

The compact size and economy of the *ALTAIR 8800* is in large part due to the CPU. Thanks to large scale integrated circuit techniques (LSI), the CPU used in the *ALTAIR 8800* is fabricated on a tiny silicon chip having a surface area of only a fraction of an inch. This chip, the Intel 8080, is installed in a protective dual-in-line mounting package having 40 pins.

The CPU is by far the most complex portion of the *ALTAIR 8800*. A complete block diagram of the CPU is shown in Figure 2-2, and while it is not necessary to possess a detailed understanding of this diagram it is important to understand the role of some of the CPU's more important systems. The interrelationship of each of these systems and their contribution to the operation of the CPU will then become more obvious.

1. TIMING AND CONTROL--The timing and Control System receives timing signals from the clock and distributes them to the appropriate portions of the CPU in order to insure coordinated instruction execution. The Timing and Control System also activates several front panel status indicators (HOLD, WAIT, INTE, STACK, OUT, IN, INP, MI MENR, HLTA, WO, INT).

*A microsecond is one millionth of a second.

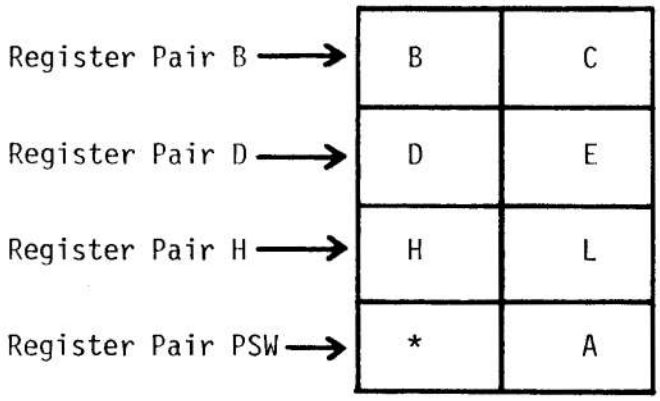
2. INSTRUCTION REGISTER--Binary machine language instructions are temporarily stored in the Instruction Register for decoding and execution by the CPU.

3. ARITHMETIC--The Arithmetic System performs both binary and decimal arithmetic. All arithmetic operations are performed by addition. Multiplication is implemented by repetitive addition. Subtraction and division are implemented by inverse addition.

4. WORKING REGISTERS--The CPU contains seven 8-bit Working Registers. The most important of these is the Accumulator, the register into which the results of many operations are eventually loaded. In addition to acting as a primary storage point for results of many program operations, numerous arithmetic and logical operations can be performed with the Accumulator and any specified register or memory address.

The six remaining registers, which are arranged in pairs to permit 16-bit operation when necessary, are "scratch-pad" registers. This simply means they are used to store temporary data or addresses on a regular basis and are available for numerous program operations.

Figure 2-3 shows the arrangement and classification of the seven Working Registers. The additional register adjacent to the Accumulator, the Status Bit Register, is a special purpose register used to store the status of certain operations.



*Status Bit Register (See Text)

FIGURE 2-3. The Working Registers

5. STATUS BIT REGISTER--The Status Bit Register is a special purpose register which stores the status of five conditions which may or may not be affected by the result of a data operation. This register contains 8-bit positions, but only 5-bits are used to store the status information. The five status bits are:

a. Carry Bit--This bit is set to 1 if a carry has occurred. The Carry Bit is usually affected by such operations as addition, subtraction, rotation, and some logical decisions. The bit is set to 0 if no carry occurs.

b. Auxiliary Carry Bit--If set to 1, this bit indicates a carry out of bit 3 of a result. 0 indicates no carry. This status bit is affected by only one instruction (DAA).

c. Sign Bit--This bit is set to show the sign of a result. If set to 1, the result is minus; if set to 0 the result is plus. The Sign Bit reflects the condition of the most significant bit in the result (bit 7). This is because an 8-bit byte can contain up to the decimal equivalent of from -128 to +127 if the most significant bit is used to indicate the polarity of the result.

d. Zero Bit--This bit is set to 1 if the result of certain instructions is zero and reset to 0 if the result is greater than zero.

e. Parity Bit--Certain operations check the parity of the result. Parity indicates the odd or even status of the 1 bits in the result. Thus if there is an even number of 1 bits, the Parity Bit is set to 1, and if there is an odd number of 1 bits, the Parity Bit is set to 0.

6. PROGRAM COUNTER--The Program Counter is a special 16-bit register which stores the address of the next program step to be executed. The Program Counter is automatically advanced to the next sequential program address upon completion of a step execution. Sometimes called the P-Counter, the Program Counter is directly accessible to the programmer via machine language instructions which implement JUMP, CALL, and RETURN instructions.

7. STACK POINTER--The Stack Pointer is another special 16-bit register. A section of memory reserved for the temporary storage of data or addresses is called the stack.

Data can be pushed onto the stack for temporary storage and popped out of the stack via several instructions.

The Stack Pointer is used to store the contents of the Program Counter during the execution of subroutines. A RETURN instruction transfers the contents of the Stack Pointer to the Program Counter and sequential execution of the main program continues. The programmer selects the location of the stack in memory by loading the Stack Pointer with the desired memory address via a special instrution (LXI).

The interrelationship of the Working Registers, Program Counter, Stack Pointer, Arithmetic System, Instruction Register, and Timing and Control System should now be more meaningful. The Working Registers incorporate six scratchpad registers and an Accumulator into which numerous operation results are temporarily stored. The Program Counter causes sequential execution of a program by keeping track of the memory address of the next instruction to be executed. The Timing and Control System supplies timing pulses which coordinate orderly program execution. The Stack Pointer is used for temporary storage of the data contained in any register pair. The Stack Pointer also saves the address in the Program Counter for retrieval after a subroutine has been executed. All these operations combine to provide an enormously flexible and versatile CPU.

B. MEMORY

Though the Working Registers, Program Counter, and Stack Pointer certainly perform memory roles, the CPU does not contain memory as it is normally defined in a computer application. The primary memory in a computer is external to the CPU.

Simple programs can be implemented with a few dozen words of memory or even less, but more complex applications such as video processing require more memory. The *ALTAIR 8800* is expandable to 65,536 8-bit words of memory.

Access to the memory is always controlled by the CPU.* 16 address lines called the Address Bus connect the CPU to the Memory. These lines permit the CPU to input or output data to or from any memory address. The addresses are specified by two 8-bit bytes. The CPU processes each address as two sequential (serial) cycles, each containing 8-parallel bits. Data stored in the Memory is exchanged between the Memory and CPU via 8 data lines called the Data Bus. This interconnection format permits parallel operation. Thus, when data is inputted or outputted in or from Memory by the CPU, it is transmitted as a complete 8-bit word.

The basic Memory in the *ALTAIR 8800* contains up to eight 256 x 4 bit random access memories (RAMs). However, any conventional memory can be used in the computer if input loading on the buss does not exceed 50 TTL loads and if the buss is driven by standard TTL loads.

*An exception to this is when the computer is connected to a Direct Memory Access Controller. DMA takes control of the address and data lines from the CPU for direct transfers of blocks of data. These transfers can take place internally (from one memory location to another) or externally (from memory to an external device).

C. CLOCK

Orderly execution of a program by the CPU is controlled by a 2-MHz crystal controlled clock. Crystal control is used to permit the clock to operate at the maximum permissible CPU speed. A clock without crystal regulation might occassionally speed up beyond the CPU's capability and program execution errors would result.

D. INPUT/OUTPUT

The *ALTAIR 8800* can be interfaced with a great many external devices. Generally, these devices provide input information to the computer and accept output information from the computer. The CPU monitors the status of program execution and Input/Output devices and provides the necessary signals for servicing external devices. The programmer can instruct the CPU to either ignore or respond to interrupt signals provided by an external device. These interrupt signals, when accepted by the CPU, cause the program execution to be temporarily halted while the external device is serviced by the computer. When the external device has been serviced, the program resumes normal execution. The *ALTAIR 8800* will service up to 256 Input and 256 Output devices.

This concludes the description of the organization of the *ALTAIR 8800*. The overall operation of the computer as a powerful and efficient data processing system will become more apparent in Part 3, a discussion of the operation of the *ALTAIR 8800*.

PART 3. OPERATION OF THE *ALTAIR 8800*

Access to the basic *ALTAIR 8800* is achieved via the front panel, and at first glance the array of 25 toggle switches and 36 indicator and status LEDs may appear confusing. Actually, operation of the *ALTAIR 8800* is very straightforward and most users learn to load a program into the machine and run it in less than an hour. If you are a typical user, you will spend far more time developing and writing programs than actually operating the machine.

This part of the *ALTAIR 8800* Operating Manual explains the purpose and application of the front panel switches and indicator and status LEDs. A sample program is then loaded into the machine and run. A detailed discussion of the role and efficient use of memory is included next. Finally, several operating hints which will help you edit and "debug" programs are included.

A. THE FRONT PANEL SWITCHES AND LEDs

Though the front panel contains 25 toggle switches and 36 indicator and status LEDs, most routine operations of the basic *ALTAIR 8800* (256 words of memory) can be performed with only 15 switches and by monitoring only 16 LEDs. The function of all the switches and LEDs is explained below:

ON-OFF Switch--The ON position applies power to the computer. The OFF position cuts off power and also erases the contents of the memory.

STOP-RUN Switch--The STOP position stops program execution. The RUN position implements program execution.

SINGLE STEP Switch--This switch implements a single machine language instruction each time it is actuated. A single machine language instruction may require as many as 5 machine cycles.

EXAMINE-EXAMINE NEXT Switch--The EXAMINE position displays the contents of any specified memory address previously loaded into the DATA/ADDRESS Switches (see below) on the 8 data LEDs. The EXAMINE NEXT position displays the contents of the next sequential memory address. Each time EXAMINE NEXT is actuated, the contents of the next sequential memory address are displayed.

DEPOSIT-DEPOSIT NEXT Switch--The DEPOSIT position causes the data byte loaded into the 8 DATA Switches to be loaded into the memory address which has been previously designated. The DEPOSIT NEXT position loads the data byte loaded into the 8 DATA Switches into the next sequential memory address. Each time DEPOSIT NEXT is actuated, the data byte loaded into the 8 DATA Switches is loaded into the next sequential memory address. The data byte loaded into the 8 DATA Switches can be changed before actuating DEPOSIT or DEPOSIT NEXT.

RESET-CLR Switch--The RESET position sets the Program Counter to the first memory address (0 000 000 000 000 000). RESET provides a rapid and efficient way to get back to the first step of a program which begins at the first memory address. CLR is a CLEAR command for external input/output equipment.

PROTECT-UNPROTECT Switch--The PROTECT position prevents memory contents from being changed. The UNPROTECT position

permits the contents of the memory to be altered.

AUX Switches--The basic *ALTAIR 8800* includes two auxiliary switches which are not yet connected to the computer. These switches will be used in conjunction with peripherals added to the basic machine.

DATA/ADDRESS Switches--The DATA Switches are those designated 7-0. The ADDRESS Switches are those designated 15-0. A switch whose toggle is in the UP position denotes a 1 bit. A switch whose toggle is in the DOWN position denotes a 0 bit. In the basic *ALTAIR 8800* (256 word memory), the ADDRESS Switches designated 8-15 are not used and should be set to 0 when an address is being entered.

2. INDICATOR LEDs

(NOTE: When machine is stopped, a glowing LED denotes a 1 bit or an active status of a specified condition; and a non-glowing LED denotes a 0 bit or inactive status. While running a program, however, LEDs may appear to give erroneous indications.)

ADDRESS--The ADDRESS LEDs are those designated A15-A0. The bit pattern shown on the ADDRESS LEDs denotes the memory address being examined or loaded with data.

DATA--The DATA LEDs are those designated D7-D0. The bit pattern shown on the DATA LEDs denotes the data in the specified memory address.

INTE--An interrupt has been enabled when this LED is glowing.

PROT--The memory is protected when this LED is glowing.

WAIT--The CPU is in a WAIT state when this LED is glowing.

HLDA--A HOLD has been acknowledged when this LED is glowing.

3. STATUS LEDs

(NOTE: A glowing LED denotes an active status for the designated condition.)

LED	DEFINITION
MEMR	The memory bus will be used for memory read data.
INP	The address bus containing the address of an input device. The input data should be placed on the data bus when the data bus is in the input mode.
M1	The CPU is processing the first machine cycle of an instruction.
OUT	The address contains the address of an output device and the data bus will contain the output data when the CPU is ready.
HLTA	A HALT instruction has been executed and acknowledged.
STACK	The address bus holds the Stack Pointer's push-down stack address.
WO	Operation in the current machine cycle will be a WRITE memory or OUTPUT function. Otherwise, a READ memory or INPUT operation will occur.
INT	An interrupt request has been acknowledged.

B. LOADING A SAMPLE PROGRAM

In Section G of Part 1, a simple addition program in machine language mnemonics is presented. The program is designed to retrieve two numbers from memory, add them together, and store the result in memory. The exact program in mnemonic form can be written thusly:

0. LDA

1. MOV (A→B)

2. LDA

3. ADD (A+B)

4. STA

5. JMP

The mnemonics for all 78 of the *ALTAIR 8800* instructions are explained in detail in Part 4 of this manual. For now, the following definitions will suffice:

1. LDA--Load the accumulator with the contents of a specified memory address.

2. MOV (A→B)--Move the contents of the accumulator into register B.

3. ADD (B+A)--Add the contents of register B to the contents of the accumulator and store the result in the accumulator.

4. STA--Store the contents of the accumulator in a specified memory address.

5. JMP--Jump to the first step in the program.*

*Once the computer has executed the program it will search its memory for something else to do. To maintain control of the CPU, we can end our sample program with a JMP instruction (followed by the memory address of the first instruction). The computer will "jump" back to the first instruction in the sample program and execute the program over and over again.

Notice how precise and specific each of these instructions is. The computer is instructed exactly how to solve the problem and where to place the result. Each of these machine language instructions requires a single byte bit pattern to implement the basic instruction. LDA and STA require two additional bytes to provide the necessary memory addresses.

To load this program into the *ALTAIR 8800*, you must first determine the memory addresses for the two numbers to be added, the result, and the program itself. In most cases, it's more convenient to store a new program by beginning at the first memory address (0). Therefore, the memory addresses for the data (the two numbers to be added and the result) should be placed at any arbitrary addresses higher in memory. Since the basic *ALTAIR 8800* has 256 words of memory, let's select a location for data addresses beginning at memory address 128. The first number to be added will be located at memory address 128 (10 000 000), the second at memory address 129 (10 000 001), and the result at memory address 130 (10 000 010). Now that the memory addresses have been specified, the program can be converted into its machine language bit patterns:

MNEMONIC	BIT PATTERN	EXPLANATION
0. LDA	00 111 010	Load Accumulator with contents
	10 000 000	of: Memory address 128 (2 bytes
	00 000 000	required for memory addresses)
1. MOV (A→B)	01 000 111	Move Accumulator to Register B
2. LDA	00 111 010	Load Accumulator with contents
	10 000 001	of: Memory address 129
	00 000 000	
3. ADD (B+A)	10 000 000	Add Register B to Accumulator

MNEMONIC	BIT PATTERN	EXPLANATION
4. STA	00 110 010	Store Accumulator contents
	10 000 010	at: Memory address 130
	00 000 000	
5. JMP	11 000 011	Jump to Memory location 0.
	00 000 000	
	00 000 000	

Usually the individual bit patterns of a machine language program are sequentially numbered to reduce the chance for error when entering them into the computer. Also, the octal equivalents of each bit pattern are frequently included since it is very easy to load octal numbers into the front panel switches. All that is necessary is to remember the binary / octal equivalents for the decimal numbers 0-7.

The resulting program may appear thusly:

STEP	MNEMONIC	BIT PATTERN	OCTAL EQUIVALENT
0.	LDA	00 111 010	0 7 2
1.	(address)	10 000 000	2 0 0
2.	(address)	00 000 000	0 0 0
3.	MOV (A→B)	01 000 111	1 0 7
4.	LDA	00 111 010	0 7 2
5.	(address)	10 000 001	2 0 1
6.	(address)	00 000 000	0 0 0
7.	ADD (B+A)	10 000 000	2 0 0
8.	STA	00 110 010	0 6 2

STEP	MNEMONIC	BIT PATTERN	OCTAL EQUIVALENT
9.	(address)	10 000 010	2 0 2
10.	(address)	00 000 000	0 0 0
11.	JMP	11 000 011	3 0 3
12.	(address)	00 000 000	0 0 0
13.	(address)	00 000 000	0 0 0

The program can now be entered into the computer by means of the front panel switches. To begin loading the program at the first memory address (0), actuate the RESET switch. The Program Counter is now loaded with the first memory address. The program is then entered into the DATA/ADDRESS switches 7-0 one step at a time. After the first step is entered, actuate the DEPOSIT switch to load the bit pattern into the memory. Then enter the second step into the DATA/ADDRESS switches and actuate the DEPOSIT NEXT switch. The bit pattern will be automatically loaded into the next sequential memory address (1). Continue loading the steps into the front panel switches and actuating DEPOSIT NEXT. The complete program loading procedure can be summarized as follows:

STEP	SWITCHES 0-7	CONTROL SWITCH
		RESET
0	00 111 010	
		DEPOSIT
1	10 000 000	
		DEPOSIT NEXT
2	00 000 000	
		DEPOSIT NEXT
3	01 000 111	

STEP	SWITCHES 0-7	CONTROL SWITCH
		DEPOSIT NEXT
4	00 111 010	
		DEPOSIT NEXT
5	10 000 001	
		DEPOSIT NEXT
6	00 000 000	
		DEPOSIT NEXT
7	10 000 000	
		DEPOSIT NEXT
8	00 110 010	
		DEPOSIT NEXT
9	10 000 010	
		DEPOSIT NEXT
10	00 000 000	
		DEPOSIT NEXT
11	11 000 011	
		DEPOSIT NEXT
12	00 000 000	
		DEPOSIT NEXT
13	00 000 000	
		DEPOSIT NEXT

The program is now ready to be run, but first it is necessary to store data at each of the two memory addresses which are to be added together. To load the first address, set the DATA/ADDRESS switches to 10 000 000 and actuate EXAMINE. You can now load any desired number into this address by loading the DATA/ADDRESS switches as appropriate. When the number has been loaded into the switches, actuate DEPOSIT to load it into the memory. To load the next address, enter the second number on the DATA/ADDRESS switches and actuate DEPOSIT NEXT. Since sequential memory addresses were selected, the number will be automatically loaded into the proper address (10 000 001). If non-sequential memory addresses had been selected, the procedure for finding the first address would have to be followed (load the address into the DATA/ADDRESS switches and actuate EXAMINE; then load the number into the DATA/ADDRESS switches and actuate DEPOSIT).

Now that the two memory addresses referenced in the program have been loaded with two numbers to be added together, the program can be run. This is accomplished by simply actuating the RESET switch and then the RUN switch. Wait a moment and then actuate the STOP switch. To see the result stored in memory, actuate the appropriate DATA/ADDRESS switches with the bit pattern for the address into which the result was stored (10 000 010) and then actuate the EXAMINE switch. The result will then be displayed on the DATA LEDs.

To test your ability to load and run this program, try changing the memory addresses for the numbers to be added and the result and then load and run the program again.

SAMPLE PROGRAM FOR BINARY MULTIPLY

MNEMONIC	ADDRESS	OCTAL CODE	EXPLANATION
MVIA	000	076	Multiplier to A Register
	001	002	
MVID	002	026	Multiplicand to D,E Registers
	003	003	
MVIE	004	036	
	005	000	
LXIH	006	041	Clear H,L Registers to initialize
	007	000	Partial Product
	010	000	
MVIB	011	006	Iteration Count to B Register
	012	010	
DADH	013	051	Shift Partial Product left into Carry
RAL	014	027	Rotate Multiplier Bit to Carry
JNC	015	322	Test Multiplier at Carry
	016	023	
	017	000	
DADD	020	031	Add Multiplicand to Partial Product if Carry = 1
ACI	021	316	
	022	000	
DCRB	023	005	Decrement Iteration Counter
JNZ	024	302	Check Iterations
	025	013	
	026	000	
SHLD	027	042	Store Answer in Locations 100,101
	030	100	
	031	000	
JMP	032	303	Restart
	033	000	
	034	000	

C. USING THE MEMORY

By now it is probably apparent that the memory plays a vital role in the efficient operation of a computer. Higher language compilers generally include a software package which automatically keeps track of the various memory addresses. Machine language operation, however, requires the programmer to keep track of the memory. Otherwise, valuable data or program instructions might be accidentally erased or replaced by other data or instructions.

You can keep track of what is stored in the *ALTAIR 8800*'s memory by means of a simple technique called memory mapping. This techmique merely assigns various types of data to certain blocks of memory reserved for a specific purpose. The technique effectively organizes the available memory into an efficient and readily accessible storage medium.

A typical memory map for the *ALTAIR 8800* with 256 words of memory might assign programs to the first 100 words, subroutines to the second 100 words, and data to the remaining 56 words. Of course the various blocks of memory can be modified at will, and the main purpose of memory mapping is to provide a cohesive organization of the available memory.

You can make a memory map each time you change the program in the *ALTAIR 8800*. After the program is written, decide how much memory space should be reserved for the program itself, the subroutines (if any), and the data. Then make a table or chart to record where various items are stored in the memory. Be sure to update the table when the memory organization is modified.

D. MEMORY ADDRESSING

The machine language instruction set for the *ALTAIR 8800* provides several methods for addressing the memory. They include direct addressing, register pair addressing, Stack Pointer addressing, immediate addressing, and stack addressing of subroutines. Each of these addressing methods will be described below.

1. Direct Addressing--The instruction supplies the specified memory address in the form of two bytes immediately following the actual instruction byte.

2. Register Pair Addressing--The contents of a register pair can contain a memory address. The H and L registers must be used for this purpose in most instructions. The H register contains the most significant 8 bits and the L register the least significant 8 bits (H is high and L is low). Two instructions (STAX and LDAX) permit the B and C or D and E register pairs to contain memory addresses.

3. Stack Pointer Addressing--There are only two stack operations: PUSH and POP. PUSHing data onto the stack causes two bytes (16 bits) of data to be stored in a special block of memory reserved by the programmer and called the stack. POPing data from the stack causes this data to be retrieved. The PUSH and POP instructions are explained in detail in Part 4 of this manual. For now it is important to know that the programmer must reserve the stack location in memory by loading a memory address into the Stack Pointer. This is accomplished by means of the LXI instruction (see Part 4). The programmer should always make note of the stack's address on his memory map.

4. Immediate Addressing--Immediate instructions contain data which is loaded into memory during program loading. Since the data is loaded along with the program in a sequential fashion, it is stored in the block of memory reserved for programming by the operator. There is no need to make any changes to the memory map when loading immediate data.

5. Stack Addressing of Subroutines--When a subroutine is CALLed by a program, the address of the next sequential instruction in the main program is automatically saved by being PUSHed onto the stack. When the subroutine has been executed, a RETURN instruction POPs the address from the stack and the main program continues execution.

E. OPERATING HINTS

As you gain experience in the operation of the *ALTAIR 8800*, you will devise methods for improving both the efficiency of your programs and the operation of the computer. Listed below are several helpful hints which you will find quite useful as you learn to operate the machine.

1. Proofreading Programs--To be safe, always proofread a program after it has been entered into the computer. This is done by returning to the first address in memory at which the program begins (actuate RESET if the program begins at memory location 0; otherwise, set the address on the ADDRESS switches and actuate EXAMINE). Check the DATA LEDs to make sure the first program step has been correctly entered. Then actuate EXAMINE NEXT and check the second step against the DATA LEDs. Continue proofreading in this fashion until the entire program has been checked. If an error is found, simply reenter the correct bit pattern on the DATA switches, actuate DEPOSIT, and continue proofreading by means of the EXAMINE NEXT switch.

2. Using NOPs--NOP is an instruction which specifies "No Operation" and is seemingly of little value. However, by scattering NOP instructions throughout a complicated program, considerable time can be saved if a program error requiring the addition of a new step or steps is found. The new instruction or data is simply entered into the program in place of the NOP instruction during the program proofreading. Always be sure to use the appropriate number of NOPs if it is felt a particular new instruction might be necessary. For example, if you think it might be necessary to add an LDA instruction to the program if it fails to execute properly, use 3 NOPs in a row at the required location. Three NOPs are required since the LDA instruction requires three separate bytes.

3. Debugging Programs--Occassionally it will be neccessary to "debug" a program. The need for debugging occurs when a program fails to execute properly because of errors (bugs). Debugging can be enhanced by use of the SINGLE STEP switch. This switch steps the computer through the program in machine cycles rather than complete program steps and permits you to observe the condition of the eight STATUS LEDs. This procedure will permit you to detect illegal entries, improper program organization, and other programming errors.

PART 4. *ALTAIR 8800* INSTRUCTION SET

The *ALTAIR 8800* has 78 basic machine language instructions. Since many of the instructions can be modified to affect different registers or register pairs, more than 200 variances of the basic instructions are possible.

A detailed description of the *ALTAIR 8800* instruction set is provided in the remainder of this operating manual. For the purpose of this description, the 78 basic machine language instructions have been grouped into seven major subdivisions:

A. Command Instructions

B. Single Register Instructions

C. Register Pair Instructions

D. Accumulator Instructions

E. Data Transfer Instructions

F. Immediate Instructions

G. Branching Instructions

Each instruction is presented as a standardized mnemonic or machine language code. Instructions may occupy from one to three sequential (serial) bytes, and the appropriate bit patterns are included. A condensed summary of the complete instruction set showing the mnemonics and instructions in both binary and octal is included as an Appendix.

A. COMMAND INSTRUCTIONS

The *ALTAIR 8800* has nine special purpose command instructions which are used to service the remaining instructions. These special purpose instructions occupy four catagories: Input/Output Instructions (IN, OUT), Interrupt Instructions (EI, DI, HLT, RST), Carry Bit Instructions (STC, CMC), and the No Operation Instruction (NOP).

1. INPUT/OUTPUT INSTRUCTIONS

There are two Input/Output Instructions and each occupies two bytes. The first byte is the instruction, and the second byte is the Input/Output device number.

IN	(INPUT)	11 011 011	(Byte 1)
		(Device No.)	(Byte 2)

Operation: An 8-bit data byte is loaded from the specified external device into the Accumulator.

Status Bits: Unaffected.

Example: Assume an input device contains the following data byte: 00 001 000. Implementation of the IN instruction (including device number) will cause the data byte to replace the contents of the Accumulator.

OUT	(OUTPUT)	11 010 011	(Byte 1)
		(Device No.)	(Byte 2)

Operation: An 8-bit data byte is loaded from the Accumulator into the specified output device.

Status Bits: Unaffected.

Example: Assume the Accumulator contains the following data byte: 00 001 000. Implementation of the OUT instruction (plus device number) will cause the data byte to be sent to the specified external device.

2. INTERRUPT INSTRUCTIONS

There are two specific Interrupt instructions (EI and DI) and two auxiliary Interrupt instructions. Interrupt instructions permit implementation of a program by a computer to be temporarily interrupted so that input/output interfacing may take place. For example, interrupts may be utilized by a computer's output device while an input device is entering data or a program.

EI (ENABLE INTERRUPTS) 11 111 011 (Byte 1)

Operation: Implementation of the EI instruction sets the interrupt flip-flop. This alerts the computer to the presence of interrupts and causes it to respond accordingly.

Status Bits: Unaffected.

DI (DISABLE INTERRUPTS) 11 110 011 (Byte 1)

Operation: Implementation of the DI instruction resets the interrupt flip-flop. This causes the computer to ignore any subsequent interrupt signals.

Status Bits: Unaffected.

HLT (HALT INSTRUCTION) 01 110 110 (Byte 1)

Operation: Implementation of the HLT instruction steps the Program Counter to the next instruction address and stops the computer until an interrupt occurs. The HLT instruction should not normally be implemented when a DI instruction has been executed. Since the DI instruction causes the computer to ignore interrupts, the computer will not operate again until the main power switch is turned off and then back on.

Status Bits: Unaffected.

RST (RESTART INSTRUCTION) 11 (esp) 111 (Byte 1)

Operation: The data byte in the Program Counter is pushed onto the stack. This provides an address for subsequent use by a RETURN instruction. Program execution then continues at memory address: 00 000 000 00 (exp) 000 where exp ranges from 000 to 111.

The RST instruction is normally used to service interrupts. The external device may cause a RST instruction to be executed during an interrupt. Implementation of RST then calls a special purpose subroutine which is stored in up to eight 8-bit bytes in the lower 64 words of memory. A RETURN instruction is included to return the computer to the original program.

Status Bits: Unaffected.

Example: Assume the following RST instruction is present: 11 001 111. Implementation of the instruction will cause the Program Counter data byte to be pushed onto the stack. The program will then continue execution at the subroutine located at memory address: 00 000 000 00 001 000. Upon completion of the subroutine, a RETURN instruction will return the computer to the next step in the main program.

3. CARRY BIT INSTRUCTIONS

There are two instructions which can be used to directly modify the status of the Carry Bit. Each instruction requires one 8-bit byte.

CMC (COMPLEMENT CARRY) 00 111 111 (Byte 1)

Operation: The Carry Bit is complemented. If it is initially 0, it is set to 1. If it is initially 1, it is reset to 0.

Status Bit Affected: Carry.

STC (SET CARRY) 00 110 111 (Byte 1)

Operation: The Carry Bit is set to 1.

Status Bit Affected: Carry.

4. NO OPERATION INSTRUCTION

There is one NO OPERATION instruction. It occupies a single 8-bit byte.

NOP (NO OPERATION) 00 000 000 (Byte 1)

Operation: No operation occurs, and the Program Counter

proceeds to the next sequential instruction. Program execution then continues.

Status Bits: Unaffected.

B. SINGLE REGISTER INSTRUCTIONS

The *ALTAIR 8800* has four single register instructions. Each instruction occupies a single byte. Two of the instructions, INR and DCR, have eight variances each. The variances are specified according to any desired register, and the following register bit patterns apply:

Register	Bit Pattern
B	000
C	001
D	010
E	011
H	100
L	101
Memory Reference M	110
A	111

If Memory Reference M (110) is specified in the instruction byte, the memory byte addressed by the contents of the H and L registers is processed. The H register contains the most significant 8 bits of the memory address and the L register contains the least significant 8 bits of the address.

INR (INCREMENT REGISTER OR MEMORY) 00 (reg) 100 (Byte 1)

Operation: The specified byte is incremented by one.

Status Bits Affected: Zero, Sign, Parity, and Auxiliary Carry.

Example: Assume the following instruction is present: 00 000 100. According to the table of register bit patterns given above, the byte in register B is to be incremented by 1. If the initial byte is 00 000 000, the incremented byte will be 00 000 001.

DCR (DECREMENT REGISTER OR MEMORY) 00 (reg) 101 (Byte 1)

Operation: The specified byte is decremented by one.

Status Bits Affected: Zero, Sign, Parity, and Auxiliary Carry.

Example: Assume the following instruction is present: 00 001 101. According to the table of register bit patterns given above, the byte in register C is to be decremented by 1. If the initial byte is 00 000 001, the decremented byte will be 00 000 000.

CMA (COMPLEMENT ACCUMULATOR) 00 101 111 (Byte 1)

Operation: Each bit in the accumulator is complemented (1s become 0s and 0s become 1s).

Status Bits: Unaffected.

Example: Assume the accumulator byte is 11 001 100. The instruction CMA will complement each bit in the accumulator byte as shown below:

11 001 100	Accumulator
----------	
00 110 011	Complemented Accumulator

DAA (DECIMAL ADJUST ACCUMULATOR) 00 100 111 (Byte 1)

Operation: The 8-bit accumulator byte is converted into two 4-bit BCD (binary-coded-decimal) numbers. The instruction affected by the Auxiliary Carry Bit.

The DAA instruction performs two operations:

1. If the least significant 4 bits in the accumulator byte (bits 0-3) represent a BCD digit greater than 9 or if the Auxiliary Carry Bit is set to 1, the four bits are automatically incremented by 6. If not, the accumulator is unaffected.

2. If the most significant 4 bits in the accumulator byte (bits 4-7) represent a BCD digit greater than 9 or if the Carry Bit is set to 1 after the previous operation, the four bits are automatically incremented by 6. If not, the accumulator is unaffected.

Status Bits Affected: Zero, Sign, Parity, Carry, and Auxiliary Carry.

Example: Assume the accumulator byte is 10 100 100. The DAA instruction will automatically consider the byte as two 4-bit bytes: 1010 0100. Since the value of the least significant 4 bits is less than 9, the accumulator is initially unaffected. The value of the most significant 4 bits is greater than 9, however, so the 4 bits are incremented by 6 to give 1 0000. The most significant bit sets the Carry Bit to 1, and the accumulator now contains: 00 000 100.

C. REGISTER PAIR INSTRUCTIONS

The *ALTAIR 8800* has eight register pair instructions. Each instruction occupies a single byte. Five of the instructions, PUSH, POP, DAD, INX, and DCX, have four variances each. The variances are specified according to any desired register pair, and the following register pair bit patterns apply:

Register Pair	Bit Pattern
B and C	00
D and E	01
H and L	10
Flags and A	11

PUSH (PUSH DATA ONTO STACK) 11 (rp)0 101 (Byte 1)

Operation: The contents of the specified register pair (rp) are stored in two bytes of memory at an address indicated by the Stack Pointer. The contents of the first register are PUSHed into the address one less than the address in the Stack Pointer. The contents of the second register are PUSHed into the address two less than the address in the Stack Pointer.

If the Status Bit Register and Accumulator (register pair PSW) pair is specified, the first byte PUSHed into memory is the Status Bit Register. This byte has the following format:

Bit Position	Contents
7	Sign Bit
6	Zero Bit
5	0
4	Auxiliary Carry Bit
3	0
2	Parity Bit
1	1

Bit Position	Contents
0	Carry Bit

For example, if the Carry Bit is set to 1 and all remaining status bits are reset to 0, the Status Bit Register will contain the following byte: 00 000 011.

After the PUSH instruction is implemented, the Stack Pointer is automatically decremented by two.

Status Bits: Unaffected.

Example: Assume PUSH BC is implemented. The instruction byte will have the following format: 11 000 101. The contents of register pair BC will be stored in memory thusly: B will be stored at the address in the Stack Pointer less one; C will be stored at the address in the Stack Pointer less two. The Stack Pointer will then be decremented by two.

POP (POP DATA OFF STACK) 11 (rp)0 001 (Byte 1)

Operation: The contents of the specified register pair (rp) are retrieved from the two bytes of memory at an address indicated by the Stack Pointer. The contents of the memory byte at the Stack Pointer address are loaded into the second register of the pair, and the contents of the byte at the Stack Pointer address plus one are loaded into the first register of the pair.

If the Status Bit Register and Accumulator (register pair PSW) pair is specified, the contents of the byte at the Stack Pointer address plus one are used to set or reset the status bits according to the format provided in the description of the PUSH instruction.

After the POP instruction is implemented, the Stack Pointer is automatically incremented by two.

Status Bits Affected: None unless register pair PSW is specified.

Example: The inverse of the example provided under the PUSH instruction will illustrate operation of the POP instruction.

DAD (DOUBLE ADD) 00 (rp)1 001 (Byte 1)

Operation: The 16-bit number formed by the two bytes in the specified register pair (rp) is added to the 16-bit number formed by the two bytes in the H and L registers. The result is stored in the H and L register pair.

Status Bits Affected: Carry.

Example: Assume the 16-bit number formed by the two bytes in register pair BC is 00 101 111 01 111 111. Assume the contents of the H and L register pair form the 16-bit number 01 100 000 00 100 101. The instruction DAD BC (00 001 001) will add the two numbers and store the result in the H and L register pair. The result of the addition is: 10 001 111 10 100 100. Since no carry occurred, the Carry Bit is reset to 0.

INX (INCREMENT REGISTER PAIR) 00 (rp)0 011 (Byte 1)

Operation: The 16-bit number formed by the two bytes in the specified register pair (rp) is incremented by one.

Status Bits: Unaffected.

Example: Assume the INX instruction 00 100 011 is present. According to the table of register pair bit patterns, the 16-bit number formed by the two bytes in the H and L register pair will be incremented by one. If the initial 16-bit number is 10 001 111 10 100 100, the new 16-bit number will be 10 001 111 10 100 101.

DCX (DECREMENT REGISTER PAIR) 00 (rp)1 011 (Byte 1)

Operation: The 16-bit number formed by the two bytes in the specified register pair is decremented by one.

Status Bits: Unaffected.

Example: Assume the DCX instruction 00 101 011 is present. According to the table of register pair bit patterns, the 16-bit number formed by the two bytes in the H and L register pair will be decremented by one. If the initial 16-bit number is 10 001 111 10 100 101, the new 16-bit number will be 10 001 111 10 100 100.

XCHG (EXCHANGE REGISTERS) 11 101 011 (Byte 1)

Operation: The 16-bit number formed by the contents of the H and L registers is exchanged with the 16-bit number formed by the contents of the D and E registers.

Status Bits: Unaffected.

Example: Assume the H register byte is 10 001 111 and the L register byte is 10 000 011. Assume the D and E register bytes are both 00 000 000. Implementation of the XCHG instruction will exchange the contents of the two register pairs so that the H and L register bytes are both 00 000 000 and the D and E register bytes are, respectively, 10 001 111 and 10 000 011.

XTHL (EXCHANGE STACK) 11 100 011 (Byte 1)

Operation: The byte stored in the L register is exchanged with the memory byte addressed by the Stack Pointer. The byte stored in the H register is exchanged with the memory byte at the address one greater than that addressed by the Stack Pointer.

Status Bits: Unaffected.

Example: The example provided under the XCHG instruction is similar to the operation which occurs when the XTHL instruction is implemented.

SPHL (LOAD SP FROM H AND L) 11 111 001 (Byte 1)

Operation: The 16-bit contents of the H and L registers replace the contents of the Stack Pointer without affecting the contents of the H and L registers.

Example: Assume the H register byte is 10 001 111 and the L register byte is 10 000 011. Assume the Stack Pointer address is 00 001 100 01 111 111. Implementation of the SPHL instruction will load the Stack Pointer with: 10 001 111 10 000 011. The contents of the H and L registers will remain unchanged.

D. ROTATE ACCUMULATOR INSTRUCTIONS

This is a special set of four instructions which apply only to the *ALTAIR 8800*'s accumulator. Only one byte of instruction is required, and no memory or register variances apply.

RLC (ROTATE ACCUMULATOR LEFT) 00 000 111 (Byte 1)

Operation: The accumulator byte is rotated one bit position to the left. The 7 bit position now occupies the 0 bit position and the Carry Bit is set with the value of the 7 bit before rotation.

Status Bits Affected: Carry.

Example: Assume the accumulator byte is 10 001 000 and the RLC instruction is present. The Carry Bit is set to equal the value of the accumulator byte's 7 bit (1), and the contents of the accumulator are rotated one bit position to the left. The 7 bit now occupies the 0 bit: 00 010 001.

RRC (ROTATE ACCUMULATOR RIGHT) 00 001 111 (Byte 1)

Operation: The accumulator byte is rotated one bit position to the right. The 0 bit position now occupies the 7 bit position and the Carry Bit is set with the value of the 0 bit before rotation.

Status Bits Affected: Carry.

Example: Assume the accumulator byte is 10 001 000 and the RRC instruction is present. The Carry Bit is set equal to the value of the accumulator byte's 0 bit (0), and the contents of the accumulator are rotated one bit position to the right. The 0 bit now occupies the 7 bit: 01 000 100.

RAL (ROTATE ACCUMULATOR LEFT THROUGH CARRY) 00 010 111

Operation: The accumulator byte is rotated one bit position to the left through the Carry Bit. The 7 bit position then occupies the Carry Bit and the Carry Bit occupies the 0 bit position.

Status Bits Affected: Carry.

Example: Assume the accumulator byte is 10 001 000, the Carry Bit is 1, and the RAL instruction is present. The contents of the accumulator are rotated one bit left through

the Carry Bit. The 7 bit now occupies the Carry Bit (1) and the Carry Bit now occupies the 0 bit: 00 010 001.

RAR (ROTATE ACCUMULATOR RIGHT THROUGH CARRY) 00 011 111

Operation: The accumulator byte is rotated one bit position to the right through the Carry Bit. The 0 bit position now occupies the Carry Bit and the Carry Bit occupies the 7 bit position.

Status Bits Affected: Carry.

Example: Assume the accumulator byte is 10 001 000, the Carry Bit is 1, and the RAR instruction is present. The contents of the accumulator are rotated one bit position right through the Carry Bit. The 0 bit now occupies the Carry Bit, and the Carry Bit now occupies the 7 bit: 11 000 100.

E. DATA TRANSFER INSTRUCTIONS

Data can be conveniently transferred between registers or between the memory and registers of the *ALTAIR 8800*. Certain of these operations are direct data transfers and no other operation is involved. For example, the MOV instruction causes a byte of data to be transferred from one register (the source register) to another register (the destination register). Other data transfers are accompanied by an arithmetic or logical operation. For example, the ADD instruction adds the contents of a specified register to the contents of the accumulator.

Still another class of data transfer instructions concerns only the accumulator and the H and L register pair. For example, the STA instruction causes the contents of the accumulator to replace the byte of data stored at a specified memory address.

This section describes fifteen separate data transfer instructions, but it is important to note that many other instructions also involve the transfer of data (e.g. PUSH, POP, DAD, XCHG, XTHL, SPHL, etc.). However, it is more appropriate to the efficient organization of this operating manual to describe these instructions elsewhere.

The data transfer instructions described in this section are grouped into three subdivisions. The first subdivision is Data Transfers (MOV, STAX, and LDAX). The second is Register/Memory to Accumulator Transfers (ADD, ADC, SUB, SBB, ANA, XRA, ORA, and CMP). And the third is Direct Addressing Transfers (STA, LDA, SHLD, and LHLD).

1. DATA TRANSFER INSTRUCTIONS

There are three data transfer instructions and each is unconditional. Each of the three instructions has at least two variances. The variances are determined by register or memory addresses which are specified by the programmer.

MOV (MOVE DATA) 01 DDD SSS (Byte 1)

Operation: The contents of SSS (the source register) are moved to DDD (the destination register). The contents of SSS remain unchanged. The following bit patterns for the source and destination registers apply:

Register	Bit Pattern
B	000
C	001
D	010
E	011
H	100
L	101
Memory Reference M	110
A	111

The source and destination registers cannot both equal 110.

Status Bits: Unaffected.

Example: Assume it is necessary to transfer the contents of register E to the accumulator. By referring to the register bit pattern table provided above, an appropriate MOV instruction can be formulated: 01 111 011.

STAX (STORE ACCUMULATOR) 00 0X0 010 (Byte 1)

Operation: The contents of the accumulator are stored in a memory address specified by registers B and C or registers D and E. Registers B and C are specified by a 0 at the 4 bit position (X). Registers D and E are specified by a 1 at the 4 bit position (X).

Status Bits: Unaffected.

Example: Assume it is necessary to store the contents of the accumulator at a memory address specified by registers D and E. The appropriate STAX instruction is: 00 010 010.

LDAX (LOAD ACCUMULATOR) 00 0X1 010 (Byte 1)

Operation: The contents of the memory address specified by registers B and C or by registers D and E replace the contents of the accumulator. Registers B and C are specified by a 0 at the 4 bit position (X). Registers D and E are specified by a 1 at the 4 bit position (X).

Status Bits: Unaffected.

Example: Assume it is necessary to load the accumulator with the contents of a memory address specified by registers B and C. The appropriate LDAX instruction is: 00 001 010.

2. REGISTER/MEMORY TO ACCUMULATOR TRANSFERS

There are eight Register/Memory to Accumulator Transfers and each is unconditional. Each of the eight instructions has eight variances determined by registers specified by the programmer. The following bit patterns for each of the registers apply:

Register	Bit Pattern
B	000
C	001
D	010
E	011
H	100
L	101
Memory Address M	110
A	111

Four of the instructions involve arithmetic (add or subtract) operations. The remaining four involve logical operations.

ADD (ADD REGISTER/ACCUMULATOR TO MEMORY) 10 000(reg) (Byte 1)

Operation: The contents of the specified register (reg) are added to the contents of the accumulator.

Status Bits Affected: Carry, Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume it is necessary to add the contents of register B to the accumulator. Referring to the register bit pattern table given above, the appropriate instruction

is: 10 000 000. If the data bytes at register B and the accumulator are 11 010 100 and 01 100 010 respectively, the following addition will be performed:

```
    11 010 100    Register B Byte

    01 100 010    Accumulator Byte
    -----------

   100 110 110    New Accumulator Byte
```

Since the new accumulator byte has nine bits, the Carry Bit will be set to 1 to indicate a carry has occurred.

ADC (ADD REGISTER/MEMORY AND CARRY TO ACCUMULATOR) 10 001 (reg)

Operation: The contents of the specified register (reg) and the content of the Carry Bit are added to the accumulator.

Status Bits Affected: Carry, Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume it is necessary to add the contents of register C and the content of the Carry Bit to the accumulator. Referring to the register bit pattern table given above, the appropriate instruction is: 10 001 001. If the data bytes at register C and the accumulator are 00 100 011 and 01 011 100 and the Carry Bit is 1, the following addition will be performed:


```
    00 100 011    Register C Byte

    01 011 100    Accumulator Byte

             1    Carry Bit
    -----------

    10 000 000    New Accumulator Byte
```

If the new accumulator byte had nine bits, the extra bit would set the Carry Bit to 1.

SUB (SUBTRACT REGISTER/MEMORY FROM ACCUMULATOR) 10 010 (reg)

Operation: The contents of the specified register are subtracted from the contents of the accumulator. The *ALTAIR 8800* achieves subtraction by means of a simple addition process called two's complement arithmetic. If there are only

eight bits in the result, no carry bit is present. This means a borrow occurred, and the Carry Bit is set to 1. Note that this operation is the inverse of what occurs in an ADD instruction.

Status Bits Affected: Carry Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume it is necessary to clear the accumulator of its contents. An efficient way to achieve this requirement is to implement a SUB A instruction (10 010 111) where A specifies the accumulator variance of the SUB instruction. Implementation of this instruction will cause the contents of the accumulator to be subtracted from itself.

SBB (SUBTRACT REGISTER/MEMORY FROM ACCUMULATOR WITH BORROW)

10 011 (reg) (Byte 1)

Operation: The content of the Carry Bit is added to the contents of the specified register and the result is then subtracted from the accumulator using two's complement arithmetic.

Status Bits Affected: Carry, Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume that the SBB instruction is implemented for the B variance (SBB B). The contents of register B will be added to the carry bit, and the result will then be subtracted from the accumulator. Status bits will be set or reset as appropriate.

ANA (LOGICAL AND REGISTER/MEMORY WITH ACCUMULATOR) 10 100 (reg)

Operation: The content of the specified register is logically ANDed with the contents of the accumulator. The Carry Bit is reset to 0.

Status Bits Affected: Carry, Zero, Sign, and Parity.

Example: Assume the content of register L is 10 001 100 and the content of the accumulator is 10 000 101. An ANA instruction will then cause the contents of the two registers to be ANDed with one another bit-by-bit. Since the logical ANDing of two bits is 1 only if both bits are 1, the following procedure occurs:

```
10 001 100    Register L
10 000 101    Accumulator
-----------
10 000 100    Register L AND Accumulator
```

XRA (LOGICAL EXCLUSIVE-OR REGISTER/MEMORY WITH ACCUMULATOR)

10 101 (reg)

Operation: The content of the specified register is logically EXCLUSIVE ORed with the contents of the accumulator. The Carry Bit is reset to 0.

Status Bits Affected: Carry, Sign, Zero, and Parity.

Example: Since the EXCLUSIVE-ORing of two bits is 1 only if the values of the bits are different, the XRA instruction can be used to clear the accumulator to 0. This function is implemented by means of the instruction XRA and the variance A. The resulting statement is 10 101 111 (see the table of register bit patterns given above).

The XRA instruction can also be used to monitor the status of individual bits in a byte which has been designated a condition byte. For example, assume a byte has been designated to record eight separate true-false conditions wherein a 1 is true and a 0 is false. In order to check whether or not any of the conditions have changed, the original data byte can be moved to the accumulator and EXCLUSIVE-ORed with the updated data byte. Conditions which have not changed will produce a 0 bit and conditions which have changed will produce a 1 bit.

ORA (LOGICAL OR REGISTER/MEMORY WITH ACCUMULATOR) 10 110 (reg)

Operation: The content of the specified register is logically ORed with the content of the accumulator. The Carry Bit is reset to zero.

Status Bits Affected: Carry, Zero, Sign, and Parity.

Example: Since the ORing of two bits is 0 only if the value of each bit is 0, the ORA instruction can be used to set a group of bits to a series of 1s.

CMP (COMPARE REGISTER/MEMORY WITH ACCUMULATOR) 10 111 (reg)

Operation: The content of the specified register is compared with the content of the accumulator by subtracting the former from the latter. The contents of the register and accumulator are unaffected by this operation, and the status bits are set or reset as appropriate.

Status Bits Affected: Carry, Sign, Zero, and Parity (Note: The sense of the Carry Bit is reversed if one byte is plus and the other is minus).

Example: The CMP instruction is useful in determining when the content of any particular register equals that of the accumulator. If the two bytes are equal, the subtraction will give a 0 result, and the Zero Status Bit will be set to 1. If the register contents are greater than the accumulator contents, the Carry Bit will be set to 1 since a subtraction has occurred. If the register contents are less than the accumulator contents, the Carry Bit will be reset to 0.

3. DIRECT ADDRESSING INSTRUCTIONS

The four instructions described in this section are used to store the contents of the accumulator and the H and L registers in the memory or to load the accumulator and H and L registers with data from the memory. All four instructions require three bytes. The first byte is the specific instruction, and the second and third bytes provide the memory address.

STA (STORE ACCUMULATOR DIRECT) 00 110 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: The contents of the accumulator are stored in the memory at the address specified in bytes 2 and 3.

Status Bits: Unaffected.

Example: Assume the accumulator byte is 00 010 110 and a STA instruction is present with the following memory address:

01 000 000 (Byte 2)

01 000 001 (Byte 3)

The accumulator byte will then be stored at this memory address.

LDA (LOAD ACCUMULATOR DIRECT) 00 111 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: The accumulator is loaded with the contents of the byte at the memory address given by bytes 2 and 3 of the instruction.

Status Bits: Unaffected.

Example: The inverse of the example given in the STA instruction will illustrate operation of the LDA instruction.

SHLD (STORE H AND L DIRECT) 00 100 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: The contents of the L register are stored in the memory at the address specified in bytes 2 and 3. The contents of the H register are stored in the memory at the next higher address.

Status Bits: Unaffected.

Example: Assume the L register byte is 00 101 100, the H register byte is 00 101 111, and an SHLD instruction is present with the following address:

01 000 101 (Byte 2)

01 110 101 (Byte 3)

The L register byte will then be stored at this memory address, and the H register byte will be stored at the next highest address.

LHLD (LOAD H AND L DIRECT) 00 101 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: The L register is loaded with the contents of the byte at the memory address given by bytes 2 and 3. The H register is loaded with the contents of the byte at the next higher memory address.

Status Bits: Unaffected.

Example: The inverse of the example given in the SHLD instruction will illustrate operation of the LHLD instruction.

F. IMMEDIATE INSTRUCTIONS

The *ALTAIR 8800* has ten immediate instructions. These instructions cause the computer to process one or two bytes of data which form a part of the instruction. Immediate instructions are available to load two bytes of data into a specified register pair, move one byte of data into a specified register or memory address, and to perform arithmetic and logical operations with the contents of the accumulator and one byte of immediate data.

A typical byte of immediate data is a mathematical constant such as pi. Immediate data can also be a number or quantity specified by the programmer such as an actual or projected inventory count. For example, a program utilizing one or more immediate instructions will permit the computer to compare the actual inventory of a particular product with the desired inventory. At any inventory count specified in the program, the computer can notify the programmer or operator of the need to reorder.

LXI	(LOAD REGISTER PAIR IMMEDIATE)	00 (rp)0 001	(Byte 1)
		(Data)	(Byte 2)
		(Data)	(Byte 3)

Operation: Two bytes of immediate data are loaded into the register pair specified rp in byte 1 of the instruction. The first byte of data (the least significant 8 bits) is loaded into the second register of the specified pair, and the second byte of data (the most significant 8 bits) is loaded into the first register of the specified pair. This procedure is reversed if the Stack Pointer is the specified register pair. The bit patterns for the register pairs are as follows:

00	Registers B and C
01	Registers D and E
10	Registers H and L
11	Stack Pointer

Status Bits: Unaffected.

Example: The following LXI instruction is inputed to the computer:

00 010 001 (Byte 1)

01 111 111 (Byte 2)

01 111 110 (Byte 3)

Bit positions 4 and 5 of byte 1 specify that the data in bytes 2 and 3 is to be loaded into registers D and E. Byte 2 is loaded into D and byte 3 is loaded into E.

MVI (MOVE IMMEDIATE DATA) 00 (reg)110 (Byte 1)

(Data) (Byte 2)

Operation: One byte of immediate data is moved into the specified register or memory byte. The following register bit patterns apply:

Register	Bit Pattern
B	000
C	001
D	010
E	011
H	100
L	101
Memory Address M	110
A	111

Status Bits: Unaffected.

Example: The following MVI instruction is inputed to the computer:

00 011 110 (Byte 1)

11 111 111 (Byte 2)

The immediate data in byte 2 is moved into register E.

ADI (ADD IMMEDIATE TO ACCUMULATOR) 11 000 110 (Byte 1)

(Data) (Byte 2)

Operation: The immediate data in byte 2 is added to the contents of the accumulator.

Status Bits Affected: Carry, Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume the accumulator byte is 11 110 000 and the ADI instruction is present. The immediate data in the ADI instruction is 10 000 000. Implementation of the ADI instruction will leave 01 110 000 in the accumulator and the Carry Bit will be set to 1. All other status bits will be reset.

ACI (ADD IMMEDIATE AND CARRY TO ACCUMULATOR) 11 001 110 (Byte 1)

(Data) (Byte 2)

Operation: The data in byte 2 and the content of the Carry Bit are added to the contents of the accumulator.

Status Bits Affected: Carry, Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume the accumulator byte is 11 110 000, the Carry Bit is set to 1, and the ACI instruction is present. The immediate data in the ACI instruction is 00 101 100. Implementation of the ACI instruction will leave the sum 00 011 101 in the accumulator and both the Carry and Parity Bits will be set to 1. The remaining status bits will be reset to 0.

SUI (SUBTRACT IMMEDIATE FROM ACCUMULATOR) 11 010 110 (Byte 1)

(Data) (Byte 2)

Operation: The data in byte 2 is subtracted from the contents of the accumulator using two's complement arithmetic. Since the *ALTAIR 8800* implements subtraction by means of addition, the Carry Bit is set to 1 if no carry occurred since this means a borrow occurred. If a borrow did not occur, a carry did occur, and the Carry Bit is reset to 0. Note that this operation is the reverse of what occurs in an ADI or ACI instruction.

Status Bits Affected: Carry, Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume it is necessary to subtract 00 000 100 from the accumulator. The resulting instruction would be as follows:

11 010 110 (Byte 1)

00 000 100 (Byte 2)

If the accumulator byte is 00 001 010, implementation of the SUI instruction will leave 00 000 110 in the accumulator. Since this is a subtraction operation and no carry is present, a borrow occurred and the Carry Bit is set to 1. The Parity Bit is also set to 1, and the remaining status bits are reset to 0.

SBI (SUBTRACT IMMEDIATE PLUS CARRY FROM ACCUMULATOR) 11 011 110

(Data)

Operation: The data in byte 2 is added to the content of the Carry Bit and the result is subtracted from the accumulator using two's complement arithmetic.

Status Bits Affected: Carry, Sign, Zero, Parity, and Auxiliary Carry.

Example: Assume it is necessary to implement the SBI instruction. The contents of the data byte will then be added to the Carry Bit and the result subtracted from the accumulator. Since this is a subtraction operation, the Carry Bit will be set to 1 if no carry occurred (meaning a borrow occurred) and reset to 0 if a carry occurred (meaning a borrow did not occur).

ANI (AND IMMEDIATE WITH ACCUMULATOR) 11 100 110 (Byte 1)

(Data) (Byte 2)

Operation: The contents of the data byte are logically ANDed with the contents of the accumulator. The Carry Bit is reset to 0.

Status Bits Affected: Carry, Sign, Zero, and Parity.

Example: Assume the content of the data byte is 00 111 011 and the content of the accumulator is 11 101 110. An ANI instruction will then cause the contents of both bytes to be ANDed together bit-by-bit. Since the logical ANDing of two bits is 1 only if both bits are 1, the following procedure occurs:

```
00 111 011      (Data Byte)
11 101 110      (Accumulator)
-----------
00 101 010      Data Byte AND Accumulator
```

XRI (EXCLUSIVE-OR IMMEDIATE WITH ACCUMULATOR) 11 101 110 (Byte 1)

(Data) (Byte 2)

Operation: The data in byte 2 of the instruction is EXCLUSIVE-ORed with the accumulator byte. The Carry Bit is reset to 0.

Status Bits Affected: Carry, Sign, Zero, and Parity.

Example: A bit is unchanged when EXCLUSIVE-ORed with a 0 and complemented when EXCLUSIVE-ORed with a 1. Therefore the EXCLUSIVE-ORed function can be used to complement any or all of the bits in the accumulator. For example, to complement all but the 7 position bit in the accumulator would require the following data byte: 01 111 111. If the accumulator byte is 10 110 001, the following operation will occur upon implementation of the XRI instruction:


```
01 111 111      (Data Byte)
10 110 001      (Accumulator)
-----------
11 001 110      Data Byte EXCLUSIVE-OR
                Accumulator
```

ORI (LOGICAL OR IMMEDIATE WITH ACCUMULATOR) 11 110 110 (Byte 1)

(Data) (Byte 2)

Operation: The data in byte 2 of the instruction is logically ORed with the accumulator byte. The Carry Bit is reset to 0.

Status Bits Affected: Carry, Sign, Zero, and Parity.

Example: The ORI instruction can be used to add 1 to the accumulator. Assume the accumulator byte is 10 000 100 and an ORI instruction is present. Since the ORing of two bits produces a 0 only if the value of the two bits is 0, the data byte 00 000 001 will add 1 to the accumulator if the 0 position bit is 0. Otherwise the accumulator byte will be unchanged.

CPI (COMPARE IMMEDIATE WITH ACCUMULATOR) 11 111 110 (Byte 1)

(Data) (Byte 2)

Operation: The data in byte 2 of the instruction is compared with the content of the accumulator by subtracting the former from the latter. The contents of the accumulator and data byte are unaffected by this operation, and the Status Bits are set or reset as appropriate.

Status Bits Affected: Carry, Zero, Sign, Parity, and Auxiliary Carry.

Example: The CPI instruction is useful in determining when the content of the accumulator equals that of the data byte. If the two bytes are equal, the subtraction process will give a 0 result, and the Zero Status Bit will be set to 1. If the data byte contents are greater than the accumulator contents, the Carry Bit will be set to 1 since a subtraction has occurred. If the Data byte contents are less than the accumulator contents, the Carry Bit will be reset to 0.

G. BRANCHING INSTRUCTIONS

The *ALTAIR 8800* has an extensive branching capability. Branching permits the computer to jump from one step in the program to another. Branching also permits the computer to call a specified set of instructions from memory and insert it into the program. A return feature permits the computer to resume normal operation after the specified instruction set is executed.

Branching is one of the most important capabilities of a computer. Jumping from one point in the program to another, for example, saves time, and calling a special set of instructions from memory means a frequently used instruction sequence need be stored at only one place in memory. The result is an important increase in computer processing speed and efficiency. Branching also adds to the economy of a computer since less memory is required to accomplish complex programs. And the ability to call frequently used instruction sets from memory can save considerable programming time.

The term subroutine is used to describe a special set of instructions stored in memory. Typical subroutines might include instruction sets for calculating trigonometric functions and square roots or making complex logical comparisons. Each of these subroutines can be quite lengthy. If a program requires a dozen or more trigonometric operations and several square root extractions, it is obvious that the use of subroutines can save considerable programming time and memory space.

Branching instructions can be either conditional or unconditional. A conditional branch means a particular branching operation is accomplished only if a specified condition is met. For example, a typical conditional branch instruction is CZ (CALL IF ZERO). If the zero bit is indeed zero when the CZ instruction is processed, the Program Counter will automatically move to the address in memory specified in the two address bytes which follow the CZ instruction in the program.

Unconditional branching causes a branch to occur without the necessity for meeting certain specified conditions.

Branching instructions require either one or three bytes per instruction. The first byte is the actual instruction while the second and third bytes are, respectively, the low and high memory addresses. The address bytes tell the Program

Counter where to move. The instructions which require only one byte need no memory addresses since some of the bits in the byte refer the Program Counter to certain registers or the Stack Pointer, either of which contains the necessary addressing information.

1. JUMP INSTRUCTIONS

JUMP instructions permit the normal execution sequence of a program to be either conditionally or unconditionally altered. For example, a program might include a set of instructions to be executed if the result of a previous operation is greater than zero. If, however, the result is zero, the set of instructions becomes superfluous and unnecessary. The program, therefore, includes a JUMP statement which instructs the computer to advance to any specified address past the instruction set. Since the jump would be implemented only if the result of the preceeding operation were zero, this would be a conditional branching operation. The actual machine language mnemonic for this particular instruction is JZ (JUMP IF ZERO).

All but one of the ten JUMP instructions require three bytes. The first byte is the specific machine language instruction, while the second and third bytes are, respectively, the low and high memory addresses for the portion of the program to be selected by the Program Counter if a jump is implemented. The PCHL instruction requires only the initial machine language instruction byte since the memory locations to which the program jumps are known by the computer. The memory locations in this case happen to be the H and L Registers, the contents of which are placed into the Program Counter.

With the exception of the PCHL and JMP instructions, all JUMP instructions are conditional. If a specified condition is true, the Program Counter automatically advances to the address specified in the instruction. If the specified condition is not true, the program continues its sequential execution and a jump does not occur.

PCHL (LOAD PROGRAM COUNTER) 11 101 001 (Byte 1)

Operation: The Program Counter jumps to the Memory address specified by the contents of the H and L Registers. The most significant 8 bits of the Program Counter are loaded with the contents of the H Register and the least significant 8

bits of the Program Counter are loaded with the contents of the L Register.

Status Bits: Unaffected.

Example: Assume the contents of the H and L Registers are as follows:

H: 10 111 000

L: 11 010 110

Instruction PCHL will automatically transfer this Memory address to the Program Counter as shown below:

	Most Significant	Least Significant
Program Counter:	10 111 000	11 010 110

The program will now continue to execute after having jumped to the new address specified in the Program Counter.

JMP (JUMP) 11 000 011 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: The Program Counter jumps unconditionally to the Memory address specified in bytes 2 and 3 and the program continues to execute from the new location.

Status Bits: Unaffected.

Example: Assume the JMP instruction and address bit pattern is as follows:

11 000 011 (Byte 1)

10 111 000 (Byte 2)

11 010 110 (Byte 3)

The Program Counter will jump to the address in Memory specified by bytes 2 and 3 and program execution will continue from the new address.

JC (JUMP IF CARRY) 11 011 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Carry Bit is 1, a carry has occurred and the Program Counter jumps to the address specified in bytes 2 and 3. Program execution then continues from the new address. If the Carry Bit is 0, no carry has occurred and the program continues sequential execution.

Status Bits: Unaffected.

Example: Assume the Carry Bit is 1 and a JC instruction is present. The Program Counter will then jump to the address specified in bytes 2 and 3 and the program will continue at the new address.

JNC (JUMP IF NO CARRY) 11 010 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Carry Bit is 0, no carry has occurred, and the Program Counter jumps to the address specified in bytes 2 and 3. Program execution then continues from the new address. If the Carry Bit is 1, a carry has occurred and the program continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the JC instruction will illustrate operation of the JNC instruction.

JZ (JUMP IF ZERO) 11 001 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Zero Bit is 1, a zero is present and the Program Counter jumps to the address specified in bytes 2 and 3.

Program execution then continues from the new address. If the Zero Bit is 0, a zero is not present and the program continues sequential operation.

Status Bits: Unaffected.

Example: Assume the Zero Bit is 1 (zero present) and a JZ instruction is present. The Program Counter will then jump to the address specified in bytes 2 and 3 and the program will continue at the new address.

JNZ (JUMP IF NOT ZERO) 11 000 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Zero Bit is 0 (zero not present) and a JNZ instruction is present, the Program Counter jumps to the address specified in bytes 2 and 3. Program execution then continues from the new address. If the Zero Bit is 1, a zero is present, and the program continues sequential operation.

Status Bits: Unaffected.

Example: The inverse of the example provided under the JZ instruction will illustrate operation of the JNZ instruction.

JM (JUMP IF MINUS) 11 111 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Sign Bit is 1 (a negative result), the Program Counter jumps to the address specified in bytes 2 and 3. Program execution then continues from the new address. If the Sign Bit is 0, the result is positive and the program continues sequential operation.

Status Bits: Unaffected.

Example: Assume the Sign Bit is 1 indicating a negative result and the JM instruction is present. The Program Counter will then jump to the address specified in bytes 2 and 3 of

the instruction and the program will continue at the new address.

JP (JUMP IF POSITIVE) 11 110 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Sign Bit is 0 (a positive result), the Program Counter jumps to the address specified in bytes 2 and 3. Program execution then continues from the new address. If the Sign Bit is 1, the result is negative and the program continues sequential operation.

Status Bits: Unaffected.

Example: The inverse of the example provided under the JM instruction will illustrate operation of the JP instruction.

JPE (JUMP IF PARITY IS EVEN) 11 101 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Parity Bit is 1 (a result with even parity), the Program Counter jumps to the address specified in bytes 2 and 3. Program execution then continues from the new address. If the Parity Bit is 0, the parity is odd and the program continues sequential operation.

Status Bits: Unaffected.

Example: Assume the Parity Bit is 1 indicating the result has even parity and the JPE instruction is present. The Program Counter will jump to the address specified in bytes 2 and 3 and the program will continue at the new address.

JPO (JUMP IF PARITY ODD) 11 100 010 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Parity Bit is 0 (a result with odd parity), the Program Counter jumps to the address specified by bytes 2 and 3. Program execution then continues from the new address. If the Parity Bit is 1, the parity is even and the program continues sequential operation.

Status Bits: Unaffected.

Example: The inverse of the example provided under the JPE instruction will illustrate operation of the JPO instruction.

2. CALL INSTRUCTIONS

CALL instructions cause a program to execute a subroutine stored at a specified location in memory. The CALL instruction may be either conditional or unconditional. Many subroutines are called unconditionally. For example, the calculation sequence for extracting a square root is relatively lengthy. In a program which requires frequent square root extractions, considerable programming time and memory space can be saved by writing a single square root extraction subroutine. This subroutine can then be stored in memory and called by the program each time it is needed.

Conditional CALL instructions are available also. They permit a great deal of flexibility since the programmer can instruct the computer to make logical decisions about the status of the program at any specified point. A subroutine can then be called if a specified condition is met.

When a subroutine has been executed, the Program Counter returns to the next step in the main program by means of a special RETURN instruction. This instruction is described in the next section.

All the CALL instructions require three bytes. The first byte is the specific machine language instruction while the second and third bytes are, respectively, the low and high Memory addresses for the first instruction of the subroutine.

CALL	(CALL)	11 001 101 (Byte 1)
		(Low Address) (Byte 2)
		(High Address) (Byte 3)

Operation: The Program Counter unconditionally moves to the Memory address specified in bytes 2 and 3. The subroutine at the new location is then executed.

Status Bits: Unaffected.

Example: Assume the CALL instruction and address bit pattern is as follows:

11 001 101 (Byte 1)

10 101 111 (Byte 2)

11 111 010 (Byte 3)

The Program Counter will move to the address in Memory specified by bytes 2 and 3 and the subroutine at that location will then be executed.

CC (CALL IF CARRY) 11 011 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This a conditional instruction. If the status of the Carry Bit is 1, a carry has occurred and the Program Counter moves to the address specified in bytes 2 and 3. The subroutine at this location is then executed. If the Carry Bit is 0, no carry has occurred, and the program continues sequential execution.

Status Bits: Unaffected.

Example: Assume the Carry Bit is 1 and the CC instruction is present. The Program Counter will then jump to the address specified in bytes 2 and 3 and the subroutine at that location will be executed.

CNC (CALL IF NO CARRY) 11 010 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Carry Bit is 0, a carry has not occurred, and the

Program Counter moves to the address specified in bytes 2 and 3. The subroutine at that location is then executed. If the Carry Bit is 1, a carry has occurred, and the Program Counter continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the CC instruction will illustrate operation of the CNC instruction.

CZ (CALL IF ZERO) 11 001 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Zero Status Bit is 1, a zero is present, and the Program Counter moves to the address specified in bytes 2 and 3. The subroutine at this location is then executed. If the Zero Status Bit is 0, no zero is present, and the program continues sequential execution.

Status Bits: Unaffected.

Example: Assume the Zero Status Bit is 1 and the CZ instruction is present. The Program Counter will then move to the address specified in bytes 2 and 3, and the subroutine at that location will be executed.

CNZ (CALL IF NOT ZERO) 11 000 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Zero Status Bit is 0, a zero is not present, and the Program Counter moves to the address specified in bytes 2 and 3. The subroutine at this location is then executed. If the Zero Status Bit is 1, a zero is present, and the program continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the CZ instruction will illustrate operation of the CNZ instruction.

CM (CALL IF MINUS) 11 111 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Sign Bit is 1 (a negative result), the Program Counter moves to the address specified in bytes 2 and 3. The subroutine at this location is then executed. If the Sign Bit is 0, the result is positive, and the program continues sequential execution.

Status Bits: Unaffected.

Example: Assume the Sign Bit is 1 and the CM instruction is present. The Program Counter will then move to the address specified in bytes 2 and 3, and the subroutine at that location will be executed.

CP (CALL IF PLUS) 11 110 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Sign Bit is 0 (a positive result), the Program Counter moves to the address specified in bytes 2 and 3. The subroutine at this location is then executed. If the Sign Bit is 1, the result is negative, and the program continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the CM instruction will illustrate operation of the CP instruction.

CPE (CALL IF PARITY EVEN) 11 101 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Parity Bit is 1 (a result with even parity), the Program Counter moves to the address specified in bytes 2

and 3. The subroutine at this location is then executed. If the Parity Bit is 0, the parity is odd, and the program continues sequential execution.

Status Bits: Unaffected.

Example: Assume the status of the Parity Bit is 1 and a CPE instruction is present. The Program Counter will then move to the address specified in bytes 2 and 3, and the subroutine at that location will be executed.

CPO (CALL IF PARITY ODD) 11 100 100 (Byte 1)

(Low Address) (Byte 2)

(High Address) (Byte 3)

Operation: This is a conditional instruction. If the status of the Parity Bit is 0 (a result with odd parity), the Program Counter moves to the address specified in bytes 2 and 3. The subroutine at this location is then executed. If the Parity Bit is 1, the parity is even, and the program continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the CPE instruction will illustrate operation of the CPO instruction.

3. RETURN INSTRUCTIONS

When a CALL subroutine instruction is executed, the address of the next sequential instruction in the program is automatically pushed onto the stack. The subroutine may have one or more RETURN statements. An unconditional RETURN instruction is included at the end of most subroutines. This instruction pops the last address stored in the stack by the CALL instruction from the stack and onto the Program Counter. When the subroutine has been executed, the program resumes sequential execution at the address following the initial CALL subroutine instruction.

Conditional RETURN instructions may be scattered throughout a subroutine. If the required condition is met, the program resumes sequential execution in the manner just described.

Since the program address to which the Program Counter returns upon receiving a RETURN instruction is already stored on the stack, RETURN instructions require only one byte. The last bit in the byte is 1 for an unconditional RETURN and 0 for conditional RETURNS.

RET (RETURN) 11 001 001 (Byte 1)

Operation: The subroutine is completed, and the Program Counter automatically and unconditionally returns to the next address following the initial CALL subroutine instruction.

Status Bits: Unaffected.

Example: Assume two of the instruction statements in an *ALTAIR 8800* program are as follows:

CALL	11 001 101	(Byte 1)
	(Low Address)	(Byte 2)
	(High Address)	(Byte 3)
CMA	00 101 111	(Byte 1)

Upon receiving the CALL instruction, the Program Counter moves to the address in Memory specified by bytes 2 and 3. Simultaneously, the address of the next sequential instruction (CMA) is pushed onto the stack.

The final instruction in the subroutine must be an unconditional RETURN (only if you wish to return). When execution of the subroutine is complete and the RET instruction is reached, the Program Counter automatically receives the address of the next instruction in the main program from the stack (CMA), and sequential execution resumes.

RC (RETURN IF CARRY) 11 011 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Carry Bit is 1, a carry has occurred and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction.

Status Bits: Unaffected.

Example: Assume three of the instructions in a subroutine are as follows:

RAL	00 10 111	(Byte 1)
RC	11 011 000	(Byte 1)
STAX	00 000 010	(Byte 1)

If the status of the Carry Bit is 1 when the RC instruction is reached, a carry has occurred and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Carry Bit is 0, the subroutine continues sequential execution by implementing the STAX instruction.

RNC (RETURN IF NO CARRY) 11 010 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Carry Bit is 0, a carry has not occurred and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Carry Bit is 1, a carry has occurred, and the subroutine continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the RC instruction will illustrate operation of the RNC instruction.

RZ (RETURN IF ZERO) 11 001 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Zero Status Bit is 1, a 0 is present and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Zero Status Bit is 0, a zero is not present and the subroutine continues sequential execution.

Status Bits: Unaffected.

Example: Assume three of the instructions in a subroutine are as follows:

ADD	10 000 101	(Byte 1)
RZ	11 001 000	(Byte 1)
LDAX	00 011 010	(Byte 1)

If the status of the Zero Status Bit is 1 when the RZ instruction is reached, a zero result is present and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL instruction. If the status of the Zero Status Bit is 0, the subroutine continues execution by implementing the LDAX instruction.

RNZ (RETURN IF NOT ZERO) 11 000 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Zero Status Bit is 0, a zero result is not present and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Zero Status Bit is 1, a zero result is present, and the subroutine continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the RZ instruction will illustrate operation of the RNZ instruction.

RM (RETURN IF MINUS) 11 111 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Sign Bit is 1 (a negative result), the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Sign Bit is 0 (a positive result), the subroutine continues sequential execution.

Status Bits: Unaffected.

Example: Assume three of the instructions in a subroutine are as follows:

SUB	10 010 001	(Byte 1)
RM	11 111 000	(Byte 1)
LDAX	00 011 010	(Byte 1)

If the status of the Sign Bit is 1 when the RM instruction is reached, a negative result is present, and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Sign Bit is 0, the subroutine continues sequential execution by implementing the LDAX instruction.

RP (RETURN IF PLUS) 11 110 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Sign Bit is 0 (a positive result), the Program Counter automatically returns to the next sequential address in the program following the initial CALL subroutine instruction. If the status of the Sign Bit is 1 (a negative result), the subroutine continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the RM instruction will illustrate operation of the RP instruction.

RPE (RETURN IF PARITY EVEN) 11 101 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Parity Bit is 1 (a result with even parity), the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Parity Bit is 0 (a result with odd parity), the subroutine continues sequential execution.

Status Bits: Unaffected.

Example: Assume three of the instructions in a subroutine are as follows:

CMP	10 111 001	(Byte 1)
RPE	11 101 000	(Byte 1)

RLC 00 000 111 (Byte 1)

If the status of the Parity Bit is 1 when the RPE instruction is reached, the parity of the result is even, and the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Parity Bit is odd, the subroutine continues sequential execution by implementing the RLC instruction.

RPO (RETURN IF PARITY ODD) 11 100 000 (Byte 1)

Operation: This is a conditional instruction which may be inserted before the end of a subroutine. If the status of the Parity Bit is 0 (a result with odd parity), the Program Counter automatically returns to the next sequential address in the main program following the initial CALL subroutine instruction. If the status of the Parity Bit is 1 (a result with odd parity), the subroutine continues sequential execution.

Status Bits: Unaffected.

Example: The inverse of the example provided under the RPE instruction will illustrate operation of the RPO instruction.

APPENDIX. ALTAIR 8800 INSTRUCTION SET

Definitions:

DDD	Destination Register
SSS	Source Register
rp	Register Pair

Register Designations:

Register (SSS or DDD)	Bit Pattern
B	000
C	001
D	010
E	011
H	100
L	101
Memory	110
Accumulator	111

Register Pair	Bit Pattern
B and C	00
D and E	01
H and L	10
SP	11

A. COMMAND INSTRUCTIONS

1. Input/Output Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
In	2	3	11 011 011	333
Out	2	3	11 010 011	323

2. Interrupt Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
EI	1	1	11 111 011	373
DI	1	1	11 110 011	363
HLT	1	1	01 110 110	166
RST	1	3	11 exp 111	3(exp)7

3. Carry Bit Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
CMC	1	1	00 111 111	077
STC	1	1	00 110 111	067

4. No Operation Instruction

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
NOP	1	1	00 000 000	000

B. SINGLE REGISTER INSTRUCTIONS

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
INR	1	3	00 DDD 100	0(DDD)4
DCR	1	3	00 DDD 101	0(DDD)5
CMA	1	1	00 101 111	057
DAA	1	1	00 100 111	047

C. REGISTER PAIR INSTRUCTIONS

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
PUSH	1	3	11 (rp)0 101	3(rp)5
POP	1	3	11 (rp)0 001	3(rp)1
DAD	1	3	00 (rp)1 001	0(rp)1
INX	1	1	00 (rp)0 011	0(rp)3
DCX	1	1	00 (rp)1 011	0(rp)3
XCHG	1	1	11 101 011	353
XTHL	1	5	11 100 011	343
SPHL	1	1	11 111 001	371

D. ROTATE ACCUMULATOR INSTRUCTIONS

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
RLC	1	1	00 000 111	007
RRC	1	1	00 001 111	017
RAL	1	1	00 010 111	027
RAR	1	1	00 011 111	037

E. DATA TRANSFER INSTRUCTIONS

1. Data Transfer Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
MOV	1	1 or 2	01 DDD SSS	1(DDD)(SSS)
STAX	1	2	00 0X0 010*	0(X)2
LDAX	1	2	00 0X0 010*	0(X)2

*NOTE: Register Pair B and C -- 0 at X
Register Pair D and E -- 1 at X

2. Register/Memory to Accumulator Transfers

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
ADD	1	1	10 000 SSS	20 (SSS)
ADC	1	1	10 001 SSS	21 (SSS)
SUB	1	1	10 010 SSS	22 (SSS)
SBB	1	1	10 011 SSS	23 (SSS)
ANA	1	1	10 100 SSS	24 (SSS)
XRA	1	1	10 101 SSS	25 (SSS)
ORA	1	1	10 110 SSS	26 (SSS)
CMP	1	1	10 111 SSS	27 (SSS)

3. Direct Addressing Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
STA	3	4	00 110 010	062
LDA	3	4	00 111 010	072
SHLD	3	5	00 100 010	042
LHLD	3	5	00 101 010	052

F. IMMEDIATE INSTRUCTIONS

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
LXI	3	3	00 (rp)0 001	0(rp)1
MVI	2	2 or 3	00 SSS 110	0(SSS)6
ADI	2	2	11 000 110	306
ACI	2	2	11 001 110	316
SUI	2	2	11 010 110	326
SBI	2	2	11 011 110	336
ANI	2	2	11 100 110	346
XRI	2	2	11 101 110	356
ORI	2	2	11 110 110	366
CPI	2	2	11 111 110	376

G. BRANCHING INSTRUCTIONS

1. Jump Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
PCHL	1	1	11 101 001	351
JMP	3	3	11 000 011	303
JC	3	3	11 011 010	332
JNC	3	3	11 010 010	322
JZ	3	3	11 001 010	312
JNZ	3	3	11 000 010	302
JM	3	3	11 111 010	372
JP	3	3	11 110 010	362
JPE	3	3	11 101 010	352
JPO	3	3	11 100 010	342

2. Call Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
CALL	3	5	11 001 101	315
CC	3	3 or 5	11 011 100	334
CNC	3	3 or 5	11 010 100	324
CZ	3	3 or 5	11 001 100	314
CNZ	3	3 or 5	11 000 100	304
CM	3	3 or 5	11 111 100	374
CP	3	3 or 5	11 110 100	364
CPE	3	3 or 5	11 101 100	354
CPO	3	3 or 5	11 100 100	344

3. Return Instructions

Mnemonic	Bytes	Cycles	Binary Code	Octal Code
RET	1	3	11 001 001	311
RC	1	1 or 3	11 011 000	330
RNC	1	1 or 3	11 010 000	320
RZ	1	1 or 3	11 001 000	310
RNZ	1	1 or 3	11 000 000	300
RM	1	1 or 3	11 111 000	370
RP	1	1 or 3	11 110 000	360
RPE	1	1 or 3	11 101 000	350
RPO	1	1 or 3	11 100 000	340