

ABACUS 1 Minicomputer

Operators Manual

Jules Carboni, 2016-2018

Edition 1, 28 October 2018

MONITOR FOR 6802 1.
C000
C000 8E 00 70 START

* FUNCTION: none
* INPUT: none
* OUTPUT: none
* CALLS: none
* DESTROYS: acc A
RESETA EQU %00010011
CTLREG EQU %00010001
INITA
LDA A #RESETA
STA A ACIA
LDA A #CTLREG
STA A ACIA
JMP SIGNON

* FUNCTION: INCH - Input character
* INPUT: char in acc A
* OUTPUT: none
* DESTROYS: acc A
* CALLS: none
* DESCRIPTION: Gets 1 character from terminal
LDA A ACIA
ASR A INCH
BCC INCH+1
LDA A ACIA+1
AND A #\$7F
JMP OUTCH

* FUNCTION: INHEX - Input hexadecimal
* INPUT: Digit in acc A
* OUTPUT: INCH
* CALLS: none
* DESTROYS: acc A
* Returns to INHEX
INHEX
C010 B6 80 04 INCH
C013 47
C014 24 FA
C016 B6 80 05
C019 84 7E
C01B 7E C0 79
C01E 8D F0
C020 81 30
C022 2B 11 C0
C024 81 39
C026 2E 0A
C028 81 41
C02A 2B 09
C02C 81 46
C02E 2E 07
C030 80
C032
C034

Contents

Part One – *Introduction* **Page 3**

Background, For Beginners, System Specifications

Part Two – *Design of the Abacus* **Page 5**

Physical Organisation, Clock, Registers, Arithmetic Logic Unit, Memory, External Storage, Instruction & Control Logic, Input & Output

Part Three – *Operation of the Abacus* **Page X**

Power Supply, Front Panel Switches and Indicators, Loading a Program, Using the Memory, Memory Addressing, Operating Hints

Part Four – *Abacus 1 Instruction Set* **Page X**

Control Pins, Instructions, Flags & Conditional Jumps

Part Five – *Programming Guide* **Page X**

Fundamentals of Programming, A Simple Program, Assembly Language, Machine Code, Debugging, Sample Programs, Concept of Operating Systems

Appendix One – *Maintenance* **Page X**

Hardware Vulnerabilities, Maintenance Advice, Bad Circuitry, Bad Microcode, Parts List

Appendix Two – *Supplementary Information* **Page X**

Boolean Algebra, Electronic Logic, Number Systems & Binary, System Schematics, Data Sheets

Part 1 — Introduction

Background Page 4

Rationale, Design

For Beginners Page 4

Required Understanding

System Specifications Page 4

Overview

1.1 Background

The *Abacus 1* was created as a research project with the purpose of exploring how computers function at the lowest level.

Abacus 1 uses a modified version of the *SAP-1* architecture described by Albert Paul Malvino in his book ‘Digital Computer Electronics’. The computer is Turing complete; it can compute anything computable within the constraints of its memory size and processing speed (explained further in [parts 2.5 & 2.2](#) respectively).

The physical construction of *Abacus 1* is highly reminiscent of home computers of the 1970s, i.e. *MITS Altair 8800*. The case opens to allow access to the internals. The “slot card” design allows components—called “modules”—to be easily removed for repair or replacement. The front panel uses LEDs to indicate the logical states of every component in the system, e.g. the registers, to assist the operator in debugging.

1.2 For Beginners

This operating manual will describe the architecture, functions and processes of the *Abacus 1*. An understanding of Boolean algebra, electronic logic, number systems (specifically the binary system) is assumed knowledge. It is highly recommended to understand these concepts fully before continuing. To assist beginners, the introductory section of MITS’ ‘Altair 8800 Operators Manual’ has been included in [appendix two “Supplementary Information”](#) of this manual. The excerpt details these concepts to a beginner audience in sufficient detail for comprehension of this manual.

1.3 System Specifications

This section of the manual outlines the key technical specifications of the *Abacus 1* system. For more detail see [part two](#).

Power	5 Volts (V_{cc}), direct current, ~1 Ampere
Architecture	Modified SAP-1, 16-bit, Von Neumann
Instruction Set	<i>Abacus 1</i> ISA, 15 instructions, 4-bit opcodes
Clock	Bistable, ~1Hz-5KHz
ALU	16-bit, addition/subtraction, xor/and/or/not
Memory	4096 16-bit words (4KB, 12-bit addresses), static
Bus	16-bits, shared, internal/external
Output	5 digit seven-segment display, signed/unsigned decimal & hexadecimal, machine state indicators

Part 2 — Design of the Abacus

Physical Organisation	Page X
Modules	
 Clock	 Page X
Astable & Monostable Modes, Speeds	
 Registers	 Page X
Importance of Accumulator, Flags Register	
 Arithmetic Logic Unit	 Page X
Arithmetic and Logic Operations	
 Memory	 Page X
Word Length, Addressing, Program Counter	
 External Storage	 Page X
Expansion Capabilities, Punch Tape Reader	
 Instruction & Control Logic	 Page X
Word Length, Control Lines, Fetch Cycle	
 Input & Output	 Page X
Front Panel Interface, Seven-Segment Display, Internal State Indicators	

2.2 Clock

The clock synchronises the actions of the whole system, keeping every component coordinated. Each component of the system activates and operates in time with the pulsing of the clock—the more frequent the clock pulses the faster the system operates.

The *Abacus 1* internal clock has an adjustable clock speed in the range of 0.721Hz to 4810Hz (~5KHz). The duty cycle ranges from 50% high at 0.7Hz to 66.67% high at 4.8KHz.

Each fetch-decode-execute cycle requires five clock pulses to complete, resulting in a maximum execution rate of 962 instructions per second.

The clock has two operational modes; a-stable “pulse mode” and monostable “step mode”. Pulse mode is considered the normal operating mode, where the clock rhythmically pulses at its set speed. Step mode overrides the internal clock with a non-latching switch labelled Step on the front panel. When Step is pressed, the clock will go high until the switch is released. Note: a logical process is used when switching between modes to ensure the wire never floats.

The Hlt control feeds back to the clock and halts it, so that neither a-stable nor monostable modes will output a clock pulse to the system, effectively freezing the computer until it is reset. This control is tied to the Hlt instruction which can be used in programs.

The LED labelled Clock on the front panel indicates the computer’s current clock speed. The LED labelled Halt indicates whether the computer is halted... If so, the computer has finished executing the program in memory. When Halt is high, the clock will be stopped.

Part 5 — Programming Guide

Fundamentals of Programming Page 8

Theory of Simulating Circuits, Methodical Approach to Programming

A Simple Program Page 9

Defining a Problem, Creating a Flow Diagram

Assembly Language Page 10

Purpose, Flow Diagram to Assembly Language, Representing Values

Machine Code Page 11

Assembly Language to Machine Code, Hexadecimal Representation

Debugging Page 12

Where Bugs Are Born, Tools to Use, Step-By-Step Process

Sample Programs Page 13

[names of sample programs]

Concept of Operating Systems Page 14

Purpose, Basic Application

5.1 Fundamentals of Programming

As has become apparent through the exploration of the *Abacus 1* and its control logic, to be meaningfully applied, the computer must be programmed. Sets of digital instructions, called programs, routines, code or software, implement variations in hardware, altering the functionality of the computer. In programming a computer, the operator is effectively wiring an invisible circuit; and so long that the circuit does not change, the process which it carries out will not change. For example, imagine the programmed “circuit” is one that adds numbers; given this “circuit” or process remains the same, the outputs will always be the same for given inputs. Computers are a piece of variable hardware, that can be programmed to imitate a constant piece of hardware reliably.

The software instructions for the *Abacus 1* must be inputted into the machine in the form of sequential 16-bit words known as machine language. Machine code will be discussed in [part 5.3](#).

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.

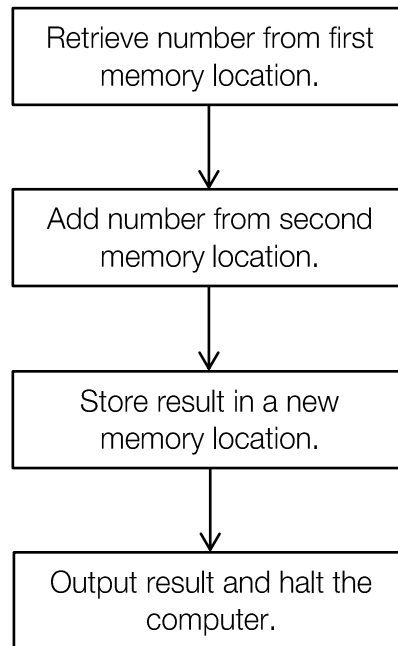
The *Abacus 1* has a small instruction set with extensive capability. For example, a program can cause data to be transferred between the computer's memory and the registers. The program can even cause the computer to make logical decisions. For example, if a specified condition is met, the computer can jump from one place in the program to any other place and continue program execution at the new place.

The *Abacus 1* instruction set was described in detail in [part four](#) of this manual.

5.2 A Simple Program

Assume you wish to use the *Abacus 1* 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, then display the number and halt execution.
2. Establish an approach—A flow diagram can now be generated:



3. Write the program—This step will be covered in the next part, [part 5.3](#), after an explanation of assembly language.

Note: Many flow diagrams can be created to compare the structure of different approaches to a problem.

5.3 Assembly Language

The software for any computer must be entered the machine in the form of binary words called machine language/code. Machine language programs are generally written with the help of mnemonics which correspond to the bit patterns for various instructions. For example, 0010 is an add instruction for the *Abacus 1* and the corresponding mnemonic is Add. Obviously, the mnemonic Add 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.

We are now ready to translate the simple program from [part 5.2](#) into assembly language. 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:

Line	Instruction	Address
0	LDA (load value into accumulator)	5
1	ADD (add value to accumulator)	6
2	STA (store accumulator to memory)	7
3	OUT (output result to display)	null
4	HLT (halt/stop the computer)	null

Make note of the line numbering, the program begins at line zero. This is because each line of code represents an address in physical memory, which is indexed from zero.

As described in [part four](#), many instructions have arguments that are required for the instruction to execute. For example, the Lda instruction on line zero looks up the value at memory address three and moves it into the accumulator. If the address space were to be empty, the instruction would attempt to load memory address zero into the accumulator, which happens to be itself; a value of 0001, not what we intended.

For this reason, we need to store our values/integers in the memory too. The following lines of this program could look like this.

Line	Instruction	Address
5	137 (the firsts number)	
6	6200 (the second number)	
7	null (the result will go here)	

The lines of a program that contain values should always be written in the final lines of the program, sequentially after the program is halted. Values stored in memory can take up the whole 16-bits of data at any address, so were these placed in the middle of a program, they could be interpreted as an instruction and cause errors in execution. Shared instruction and data memory is one limitation of the Von Neumann architecture.

5.4 Machine Code

To put it simply, “machine code” is ones and zeroes. The mnemonics of assembly language are directly interchangeable with the raw binary instructions of machine code. This makes “assembling programs” simple.

Assembled, the program from [part 5.3](#) would be written like this:

Line	Instruction	Address
... 0000	0001	0000 0000 0101
... 0001	0010	0000 0000 0110
... 0010	0100	0000 0000 0111
... 0011	1110	0000 0000 0000
... 0100	1111	0000 0000 0000
... 0101	0000	0000 1000 1001
... 0110	0001	1000 0011 1000
... 0111	0000	0000 0000 0000

Make note of how binary representation of the decimal value 6200 flows into the instruction space.

The binary view of the program is not elegant nor space efficient. To neaten things up we can use hexadecimal notation to display this program; where in the contents the first digit represents to 4-bit instruction and the following three represent the 12-bit address.

Memory Address	Memory Contents
000	1 005
001	2 006
002	4 007
003	E 000
004	F 000
005	0 089
006	1 838
007	0 000

Congratulations, you’ve just written your first program for the *Abacus 1* minicomputer! This code can be loaded into the memory of the computer and executed. Additionally, the code can be stored on paper or punch tape for storage outside of the volatile system memory for later recall/usage.

5.5 Debugging

Occasionally, the code loaded into the computer's memory will not execute as expected. This is most likely the result of an error, called a "bug", in the program. Rarely, this can be the result of a hardware failure; [appendix one](#) details the proper method of identifying and repairing faulty hardware. A bug often arises at three times in the process of programming a computer:

1. When establishing an approach.
2. When writing the program.
3. When loading the program into memory.

There are a few things that will make debugging your code easier. These include:

1. The program's flow diagram.
2. The program itself, written in assembly code.
3. The instruction set reference, including all control signals (see [part four](#)).

The process of debugging can be both time consuming and infuriating; patience is essential. Rest assured, finally executing your program without error will be a very satisfying and rewarding experience. To successfully debug a program, follow these steps:

1. Switch the clock mode from Pulse to Step; this ensures that the computer will not run before we're ready.
2. Load the program from scratch; sometimes a "buggy" program can alter its own code, further worsening the problem.
3. Reset the computer by pressing the Reset switch. Initialisation of the computer is done.
4. Look up the current instruction in your program and make note of the controls that it activates at each "time". The clock should be at "time zero", the first half of the fetch cycle. It is safe to step the clock once, moving to "time one", the second half of the fetch cycle. If there are any erroneous controls at this point, there is an error in the computer's microcode; see [appendix 1.4 "Bad Microcode"](#).
5. Step the clock by one, progressing to "time two". Check that the instruction in the Instruction Register is the same as the one in your program. If not, there is an error in translation from assembly language to machine code.
6. Compare the active controls to that of the instruction to check that there are no errors in the Microcode. If there are any erroneous controls at this point, there is an error in the computer's microcode; see [appendix 1.4 "Bad Microcode"](#).
7. Repeat steps five to six until at "time four".
8. Compare the result of the instruction to the expected results described in your flow diagram and program (check register indicators). If there is a discrepancy, there may be an error in the usage of certain instructions, the implementation of code from the flow diagram, or simply a flow diagram that does not describe the process desired for execution.
9. Repeat steps four to eight until a bug is found. If the program is not to blame for the erroneous operation of the computer, there may be an error in the wiring of the computer's circuitry; see [appendix 1.3 "Bad Circuitry"](#).

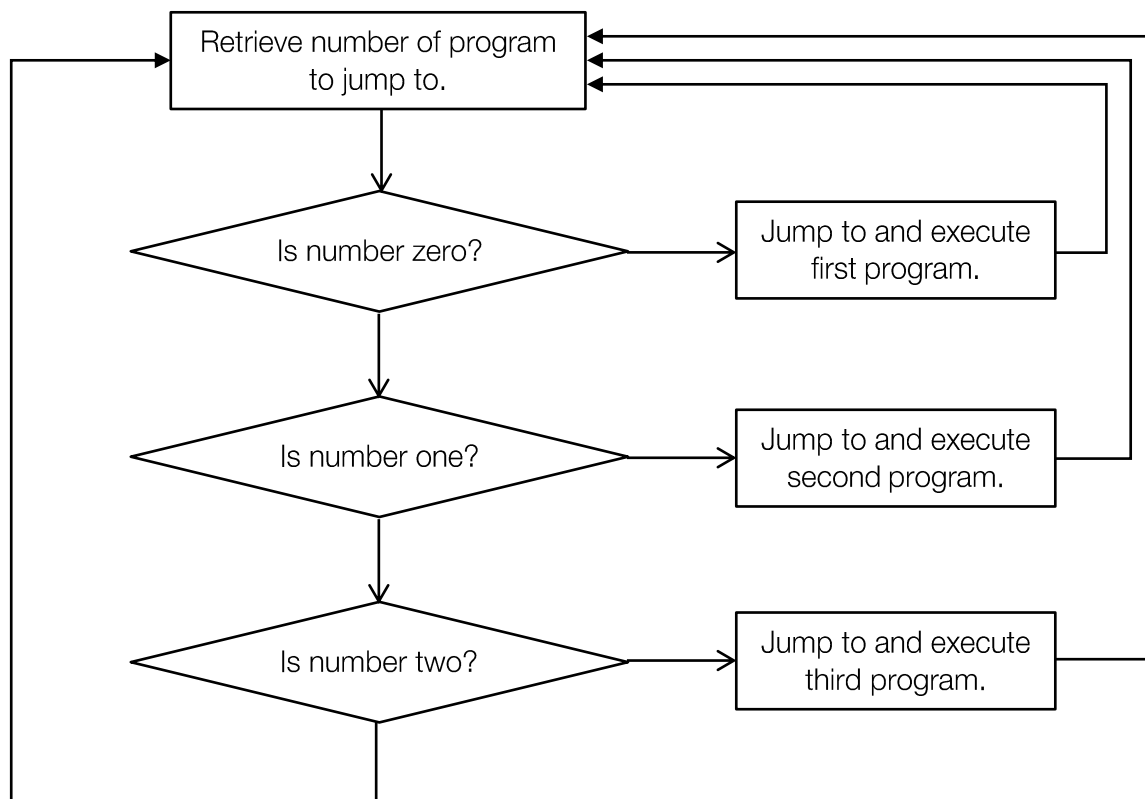
5.6 Sample Programs

[programs]

5.7 Concept of Operating Systems

Traditionally, a program is loaded into a computer's memory and is executed until completed, at which point the computer halts. Only one program can be loaded into a computer at once and must be reprogrammed before each use. Operating systems can be used in computers to elevate this limitation, allowing the user to select from hundreds of programs which have all been loaded into memory at once.

The simplest form of an operating system is a small program located at the beginning of the system's memory which, referencing a user-defined value, can jump to the location of and execute many other programs. This can be represented in a flow diagram:



Appendix 1 — Supplementary Information

Boolean Algebra (Logic) Page 16

History, Logical Operations, Binary States, Truth Tables

Electronic Logic Page 19

Logic Gates, Layers

Number Systems & Binary Page 21

Overview, The Binary System, Conversion

System Schematics Page 25

[to be done]

Data Sheets Page 26

[names of datasheets]

A. LOGIC

George Boole, a nineteenth 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.

3

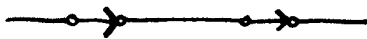
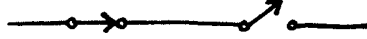
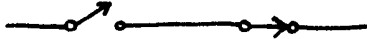

With this in mind, it is possible to devise electronic equivalents for the three basic logic statements: AND, OR and NOT. The AND statement is true if and only if either or all of its logic conditions are true. A NOT statement merely reverses the meaning of a logic statement so that a true statement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

5

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.

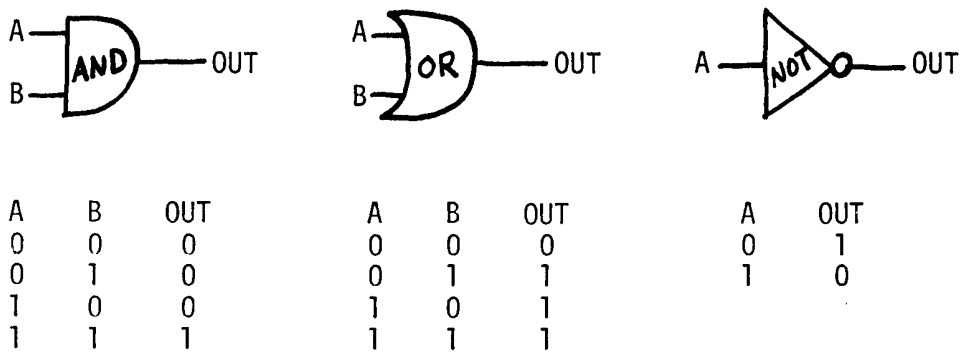


FIGURE 1-4. The Three Main Logic Symbols

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.

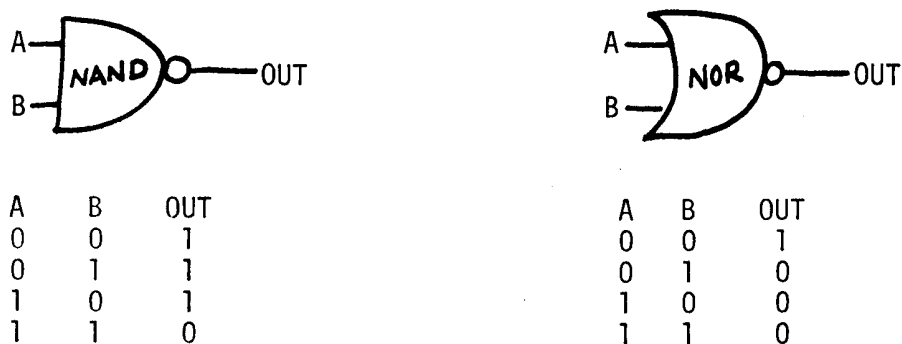


FIGURE 1-5. The NAND and NOR Circuits

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.

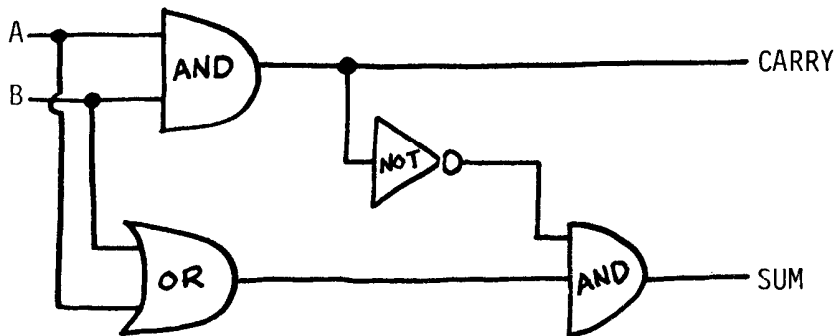


FIGURE 1-6. The EXCLUSIVE-OR Circuit

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 not 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:

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

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

$$= 19$$