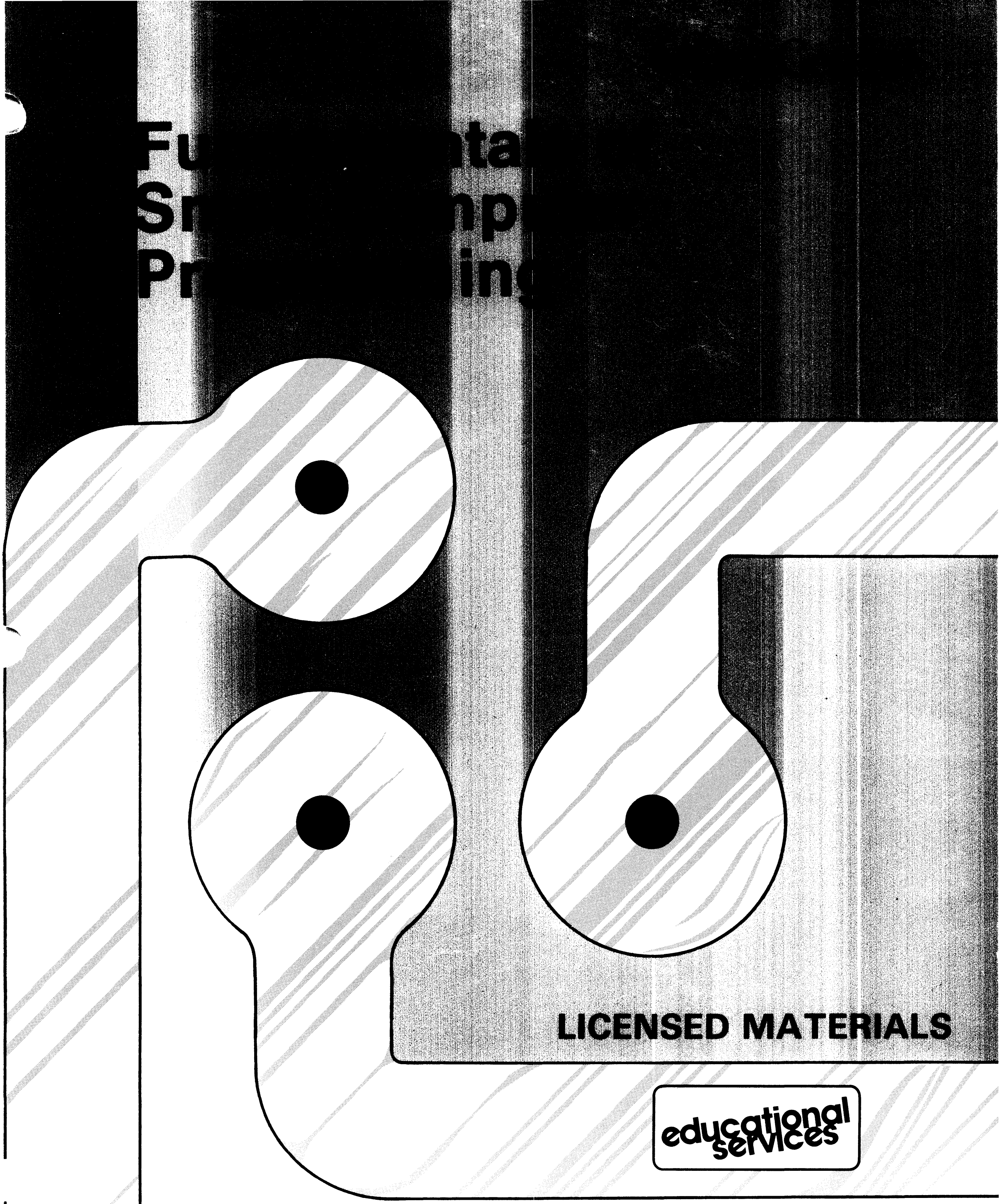


Fu[illegible]nta[illegible]
Sm[illegible]mp[illegible]
Pr[illegible]ing

LICENSED MATERIALS

educational services

SELF-STUDY COURSE

053-000039-00

NOTICE

DATA GENERAL CORPORATION (DGC) HAS PREPARED THIS DOCUMENT FOR USE BY DGC PERSONNEL AND CUSTOMERS AS A GUIDE TO THE PROPER INSTALLATION, OPERATION, AND MAINTENANCE OF DGC EQUIPMENT AND SOFTWARE. THE DRAWINGS AND SPECIFICATIONS CONTAINED HEREIN ARE THE PROPERTY OF DGC AND SHALL NEITHER BE REPRODUCED IN WHOLE OR IN PART WITHOUT DGC PRIOR WRITTEN APPROVAL NOR BE IMPLIED TO GRANT ANY LICENSE TO MAKE, USE, OR SELL EQUIPMENT MANUFACTURED IN ACCORDANCE HEREWITH.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

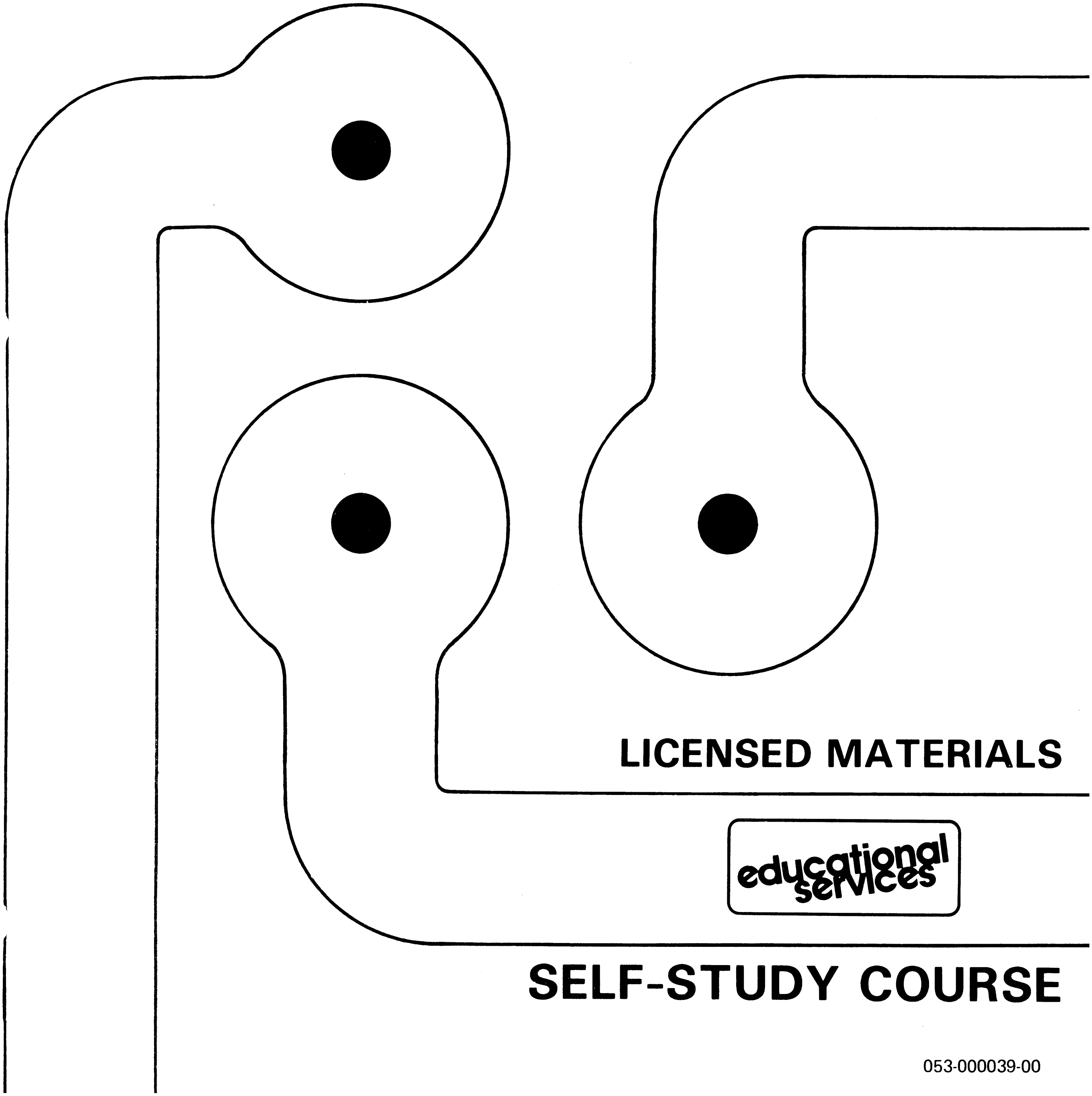
THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF ACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

CEO, DASHER, DATAPREP, ECLIPSE, ENTERPRISE, INFOS, MANAP, microNOVA, NOVA, PROXI, SUPERNOVA, ECLIPSE MV/4000, ECLIPSE MV/6000, and ECLIPSE MV/8000 are U.S. registered trademarks of Data General Corporation. AZ-TEXT, DG/L, ECLIPSE MV/10000, GW/4000, GDC/1000 GENAP, PRESENT, REV-UP, SWAT, TRENDVIEW, DEFINE, SLATE, microECLIPSE, BusiPEN, BusiGEN, BusiTEXT, and XODIAC are U.S. trademarks of Data General Corporation.

Data General

Fundamentals of Small Computer Programming



LICENSED MATERIALS

educational services

SELF-STUDY COURSE

053-000039-00

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION TO MINICOMPUTERS

CHAPTER 2 - BINARY - THE LANGUAGE OF THE COMPUTER

CHAPTER 5 - I/O DEVICE HANDLING

APPENDIX A - SAMPLE PROGRAMS

APPENDIX B - PROGRAMMING TRICKS

APPENDIX C - INSTRUCTION MNEMONICS

APPENDIX D - IN-OUT CODES

CHAPTER 1

INTRODUCTION TO MINICOMPUTERS

1.1 MACHINES

By just about any definition, a computer is a machine. According to Marvin Minsky, professor of electrical engineering at MIT, a machine may be defined as "the realization in material of an abstract concept."[1] Let's use an every day example. Consider the simple act of cutting grass. A machine called a lawn mower is what takes this abstract concept and makes it very real. So the lawn mower is the realization of the abstract concept of cutting grass.

1.1.1 Physical Process Machines

In its function of cutting grass, the lawn mower is the mechanization of a process. The lawn mower, like most other machines before the advent of the computer, performed physical processes. That is, the machine controlled the transformation and use of energy.

1.1.2 Intellectual Process Machines

With the advent of the computer came a machine that would perform an intellectual process. That is, the computer controls the transformation and use of information. As an intellectual processor, the computer must do three types of operations in order to work with information. The computer must:

1. Get information from and give information to the environment.
2. Transform information from one form to another.
3. Remember information for future recall.

---

[1] Marvin Minsky, Computation - Finite and Infinite Machines. Prentice - Hall, Englewood Cliffs, NJ, 1967.

1.1.2 Intellectual Process Machines (Continued)

Relating these three types of operations to ourselves as information processors, consider the job of getting up in the morning:

1. Ears hear an alarm.
2. Brain perceives this noise as much louder than other noises; therefore, it must be important.
3. Brain checks with memory for a record of such noises.
4. Brain gathers all the available memory data about such a noise, and attempts to match a memory pattern to the input noise.
5. After the match is found, the brain directs the body to turn off the alarm and get up.

1.2 Computers

The three types of operations that the computer must do, as an intellectual processor, in order to work with information, can be directly related to the three main sections of the computer:

1. Central Processing Unit (CPU) transforms information from one form to another.
2. Input/Output (I/O) interacts with the environment; acts as the CPU's sensors.
3. Main Memory remembers information for future recall.

Figure 1.1 shows the relationship of these three units to each other.

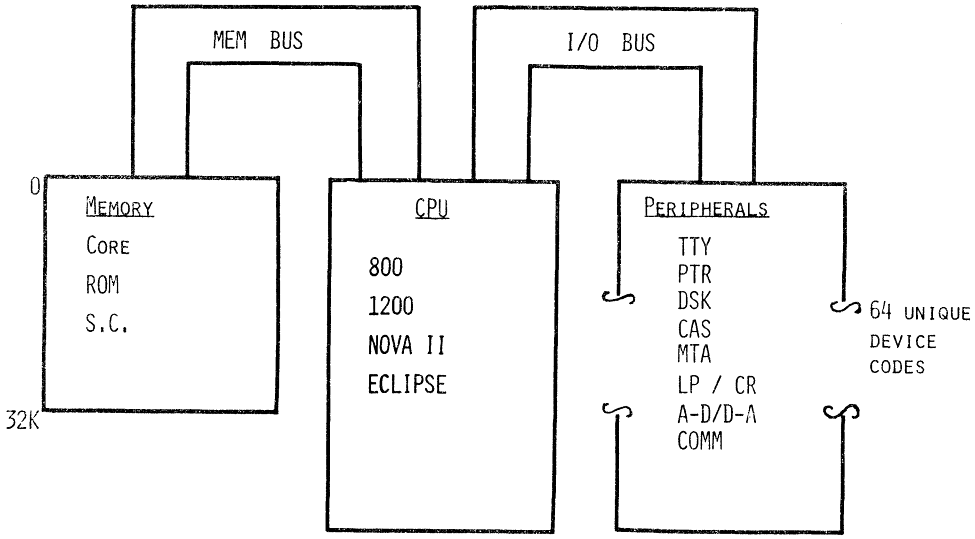


Figure 1.1 System Block Diagram

1.2 COMPUTERS (Continued)

Because the computer has no intelligence of its own, it must be told to perform every task desired of it. It is the Central Processing Unit (CPU) that is the heart of the computer. The CPU is the main director of computer operations in that it deciphers all instructions to the computer in such a manner as to accomplish the desired task.

If the computer is to solve a problem or perform a function, it must have available to it all the commands and additional information necessary to accomplish the task. This information is retained in the main memory of the computer and is available for access by the CPU.

Let us examine memory first, so that we might better understand how it does its job of remembering for future recall.

1.2.1 Memory

There are two basic types of main memory: read/write and read only.

1.2.1.1 Read Only

Read only memory (ROM) is analagous to a reference manual. The information it contains is accessible, but for practical purposes, unalterable. Read only memory might be used for storage of frequently used constants and subroutines, or for storing information permanently, where the loss of the information would be catastrophic. Because of the physical nature of a ROM, execution of a program stored in read only memory is usually significantly faster than execution of the same program stored in read/write memory.

1.2.1.2 Read/Write

Read/write memory contains physical elements that are capable of having information read out of them, and of having new information stored into them. In other words, it is possible to read from and write into this type of memory.

1.2.1.2
Read/Write
(Continued

A common read/write memory element is the magnetic core. It is a donut-shaped piece of ferromagnetic material with a wire running through it. By passing a direct current through the wire, it is possible to magnetize the core.

By reversing the energizing current in the wire, it is possible to change the magnetization of the core. Thus, a core magnetized in one direction has a value of 1, and a core magnetized in the other direction has a value of 0. We are able to read from this memory by detecting the polarity of magnetization, and we are able to write into memory by energizing the wire in the appropriate direction.

A commonly used analogy for understanding read/write memories is that of the pigeonholes in the post office. In the following statements, the underlined terms refer to read/write memories, while information within parenthesis refers to the pigeonhole analogy.

In memory every <u>location</u> (box) has its own unique <u>address</u> (1432 Franklin Park Circle).

What lives at that address (i.e., its <u>contents</u>) is called <u>data</u> (the Joneses).

Many people come to 1432 Franklin Park Circle, and visit with the Joneses (some go away with a picture of the Joneses) but when they go, (the Joneses are still there). So too, you can <u>read</u> from memory without changing its content.

If the stork comes, they may (gain a Jones) or if the preacher comes they may (lose a Jones); with such minor <u>modifications</u> they are (still basically the Joneses).

1.2.1.2
Read/Write
(Continued)

However, if they fail to make their mortgage payments, the Joneses may not live at 1432 Franklin Park Circle anymore; (the old residents may be replaced by new ones).

The content of an address has been referred to as data. Data can be one of three things; it depends upon who is calling:

a. An instruction (daddy). When the CPU needs to know what to do next, it uses the program counter to call on memory.

b. An address (husband). When the CPU needs to know where to look, it uses the instruction register or indeed the content of one memory location to call on another. It's sort of like going to your mother's house to find out where you live.

c. An operand (Tom). When the CPU has decided that it is at the final address, the content is the data to be manipulated in accordance with the instruction.

Daddy, husband, and Tom are all the same person; it depends on who is calling as to how that person will be addressed.

Table 1.A Summary of Pigeonhole Analysis

Term	Memory	Pigeonhole
location	address	1432 Franklin Park Circle
content	data instruction address operand	The Joneses daddy husband Tom
read	unaltered	still live there
modify	plus one, minus one	still basically Joneses.
write	new replaces old	evicted for nonpayment

1.2.3 Central Processing Unit

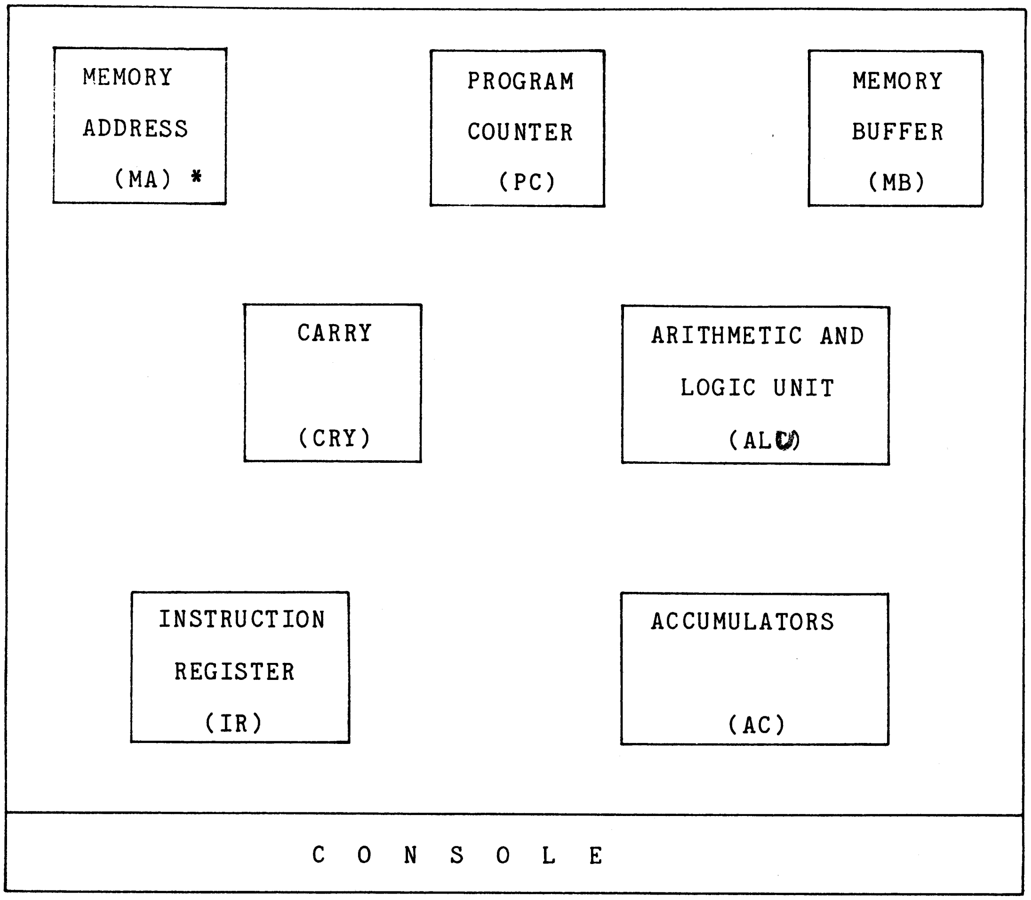
Now let us take a closer look at the CPU so that we might better understand how it does its job. Figure 1.2 might represent a typical CPU. The entries that we see in this block diagram are as follows:

1. Program Counter (PC) - Holds the address of the <u>next</u> instruction to be executed.
2. Instruction Register (IR) - Holds a copy of the current instruction for decoding and execution.
3. Arithmetic and Logic Unit (ALU) - That's where the number crunching takes place; where all data manipulation takes place.
4. Accumulators (AC) - An internal, easily-accessible, limited-storage area for the temporary storage and manipulation of operands. This type of storage is often referred to as scratch-pad memory.
5. Carry (CRY) - An arithmetic extension of the ALU used to indicate overflow; a carry out of the most significant digit.
6. Memory Address register* (MA) - Keeps track of the last address that was referenced.
7. Memory Buffer register* (MB) - Contains the content of the last address that was referenced.
8. Console - This term should not be confused with the Teletype®** keyboard.

---

* Each memory also contains its own MA and MB registers.

** Teletype is a registered trademark of Teletype Corporation, Skokie, Illinois.



* Each memory also contains its own MA and MB registers.

Figure 1.2 Typical CPU

1.2.3 Central Processing Unit (Continued)

The console contains the switches for controlling the operation of the computer. There are switches for starting, stopping, resetting, and examining the various components of the system. In addition, a number of indicator lights are provided on the console to allow the computer operator to determine visually the status of the computer at any time. As well as being the manual control panel for the computer, the console enables the programmer to follow the execution of his program to detect any flaws, or bugs, in the program. The console is actually a manual control panel connected to the input/output facilities, supplying information to the CPU, and displaying information from the CPU.

1.2.4 Input/Output Interface

The third section of the computer is the input/output interface. This is the section that connects the CPU with its environment to provide the channel for the flow of information from the outside world into the computer, and vice versa. This section connects to, and controls, such devices as keyboards, printers, paper tape punches, paper tape readers, magnetic tape recorders, magnetic discs, magnetic drums, CRT displays, analog to digital converters, digital to analog converters, card readers, card punches, etc.

Through the I/O section, the CPU can obtain and retain data and/or additional instructions from the outside world. This is known as the interactive portion of the computer.

1.3 REVIEW QUIZ

1. The computer controls the transformation and use of ____________.

2. The three main sections of a computer are:
   a. ____________________________
   b. ____________________________
   c. ____________________________

3. The pigeonhole analogy is used to illustrate that R/W memory was designed by the post office. True or False. (Circle one.)

4. The content of a memory location can be one of three things:
   a. ____________________________
   b. ____________________________
   c. ____________________________

5. "Scratchpad memory" is used to handle the overflow from main memory. True or False. (Circle one.)

6. The indicator for arithmetic overflow is called _________________.

7. The purpose of the console is twofold:
   a. manual control
   b. ________________________

8. The term that applies collectively to all I/O devices is: ___________.

Check your answers on the next page.

Chapter 1
Review Quiz
Answers

1. Information

2. a. Input/output
   b. CPU
   c. Memory

3. False

4. a. Instruction
   b. Address
   c. Operand

5. False

6. Carry or CRY

7. Display

8. Peripherals

CHAPTER 2

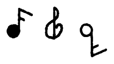
BINARY - THE LANGUAGE OF THE COMPUTER

Since all of the information that passes through a computer is in the form of numbers, and since all of the instructions that the computer executes are also in the form of numbers, it is helpful to have a basic understanding of the number systems that a computer uses.

2.1 NUMBERING SYSTEMS

A number system is just one type of information system. Information systems in general are simply abstract concepts represented by symbols and interpreted according to a set of rules. Table 2.A below lists various systems of symbols and their associated rules for interpretation.

Table 2.A Symbols and Rules

Symbols	Rules
A-Z et.al.	Grammar
.-	Morse Code
0-9 et.al.	Mathematics
	Music

To understand the symbols, you've got to learn and adhere to the rules.

The number system that the computer uses, called the binary numbering system, follows the same set of rules as the number system with which we are most familiar: the decimal numbering system. The primary difference is in the number of distinct marks or digits that exist within each system. As their names imply, the DECimal system has ten distinct marks and the BInary system has two distinct marks.

2.1 NUMBERING SYSTEMS (Continued)

Before we look at the rules for interpreting these numbering systems, why do you suppose binary, a system with only two digits, became the language of the computer? Actually, early analog computers attempted to use the decimal numbering system.

As you look around you'll notice that many physical devices have two states:

- The light bulb is on or off.
- The door is open or closed.
- A memory core is magnetized in one direction or the other.
- A switching circuit is either saturated or cutoff.
- The answer to number five is true or false.
- This is getting ridiculous, yes or no.

The purpose of the last two entries is to show that the two-state world is not restricted to physical devices. Indeed, some of the most complex problem-solving can be broken down into a series of yes-no questions.

2.2 DECIMAL NUMBERING SYSTEM

Now down to the business at hand. To more easily understand the binary numbering system, let's start by reviewing the one with which we are most familiar.

2.2.1 Digits

The decimal numbering system contains ten distinct marks, called digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

2.2.1 Digits (Continued)

Each digit from left to right is the result of increasing the value of the previous digit by "1;" e.g.,

$$\begin{array}{r} 3 \\ +1 \\ \hline 4 \end{array} \qquad \begin{array}{r} 5 \\ +1 \\ \hline 6 \end{array} \qquad \begin{array}{r} 7 \\ +1 \\ \hline 8 \end{array} \qquad \begin{array}{r} 9 \\ +1 \\ \hline ? \end{array}$$

2.2.2 Overflow and Carry-In

What happens when the largest digit, nine, is increased by one? This is the very basic concept that most of us missed in learning by rote. Nine plus one is <u>not</u> ten; when one is added to the largest digit, it results in an overflow condition. That is, a zero is recorded in this digit position and a one is carried over to the next highest digit position. There the one becomes a carry-in, or is added into the new position.

$$\begin{array}{rr} \text{carry} \longrightarrow & 1 \\ & 09 \\ & +01 \\ \hline & 10 \end{array}$$

2.2.3 Digit Position

The concepts of overflow and carry-in have introduced a new concept: positional value. The value of a digit depends upon the digit's position within the number. In the number 1234, the digit 2, although a lesser digit than 4, has a greater value because of its position within the number.

The value of a position, called its weight, indicates a power of the base* or, how many times the base* has been multiplied by itself.

$$\begin{array}{c} 10^3 \searrow \ 10^2 \downarrow \ 10^1 \swarrow \ 10^0 \swarrow \\ 1234 \end{array}$$

$$\begin{array}{rcrcrcl} 1000 & = & 1 \times 1000 & = & 1 \times 10^3 & \leftarrow & 1 \\ 200 & = & 2 \times 100 & = & 2 \times 10^2 & \leftarrow & 2 \\ 30 & = & 3 \times 10 & = & 3 \times 10^1 & \leftarrow & 3 \\ + \quad 4 & = & 4 \times 1 & = & 4 \times 10^0 & \leftarrow & 4 \\ \hline 1234 & & & & & & \end{array}$$

---

* Base refers to the number of distinct digits; in decimal, it's ten.

2.2.3 Digit Position (Continued)

Let's take the analysis of digit versus position one step further. In the previous example, the digit 1 was raised to the third base three times as follow:

1 X 10 X 10 X 10, to this quantity was added
\+ 2 X 10 X 10, to this quantity was added
\+ 3 X 10, to this quantity was added
\+ 4.

Another way of writing the same procedure is:

Notice that by the time the total has been reached, the original 1 gets multiplied by the base three-times (which corresponds to its power of the base in the final number: $1X10^{3}$), the original 2 gets multiplied by the base twice (its power of the base: $2X10^{2}$), the original 3 once ($3X10^{1}$) and the original 4 never gets multiplied by the base, just added in to the total (corresponding to $4X10^{0}$)

$$\begin{array}{r} 1 \\ \text{X } 10 \\ \hline 10 \\ +\ 2 \\ \hline 12 \\ \text{X } 10 \\ \hline 120 \\ +\ 3 \\ \hline 123 \\ \text{X } 10 \\ \hline 1230 \\ +\ 4 \\ \hline 1234 \end{array}$$

This procedure has hidden in it another basic concept that will be used shortly to convert numbers between different bases.

Before we introduce new bases, let's highlight the concepts we've discussed about decimal.

1. Ten distinct marks.
2. Largest digit plus one results in zero and a one carry to the next digit position.
3. The value of a position indicates its power of the base.
4. A digit's value depends upon its position within the number.

2.3 BINARY NUMBERING SYSTEM

Now let's apply the concepts to the Binary numbering system.

2.3.1 Digits

The distinct marks, called digits, are:

0,1

Each digit is the result of increasing the previous digit by "1;" e.g.,

$$\begin{array}{r}0\\+1\\\hline 1\end{array}\qquad\begin{array}{r}1\\+1\\\hline ?\end{array}$$

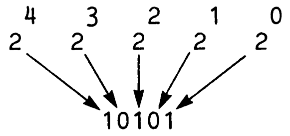
2.3.2 Overflow and Carry-In

The largest digit plus one results in a zero and a one carry to the next digit position:

$$\begin{array}{r}1\\+1\\\hline 10\end{array}$$

2.3.3 Digit Position

The value of a position indicates its power of the base.



A digit's value depends upon its position within the number.

```
                                      10101
                                      |||||
10000 = 1 X 10000 = 1 X 2◄────────────┘||||
    0 = 0 X  1000 = 0 X 2◄─────────────┘|||
  100 = 1 X   100 = 1 X 2◄──────────────┘||
    0 = 0 X    10 = 1 X 2◄───────────────┘|
    1 = 1 X     1 = 1 X 2◄────────────────┘
─────
10101
```

As in the decimal numbering system, the power of the base can be thought of as the number of zeroes to the right of the digit 1.

2.3.3 Digit Position (Continued)

For example:

decimal	binary
$10^2 = 100$	$2^2 = 100$
$10^3 = 1000$	$2^3 = 1000$
$10^4 = 10000$	$2^4 = 10000$

The type of thinking applied in the previous statement helps us over the hump of saying $2^2 = 4$ or $2^3 = 8$. That type of thinking was fine in decimal but becomes a stumbling block when we go to other bases.

2.4 OCTAL NUMBERING SYSTEM

While the computer uses binary because of its simplicity, we as humans can't handle all that simplicity all at once. In other words, it becomes very cumbersome when you have to represent even relatively small quantities with binary numbers. For instance, if you give me 11010_2 cents for a 39_{10} cent item, one of us is getting a deal. What we need is a system that will reduce all those 1s and 0s into something more manageable. There are actually two equally suitable alternatives, a base sixteen numbering system and a base eight numbering system. This book is only going to deal with the base eight numbering system, otherwise known as octal. At this time we will introduce octal using the same concepts that were established for decimal and then used to introduce binary. In the next section where we convert numbers from one base into other bases, we will see why octal is referred to as binary shorthand.

Now let's apply the concepts established for decimal to the octal numbering system.

2.4.1 Digits

The distinct marks, called digits, are:

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

Each digit is the result of increasing the previous digit by "1;" e.g.,

$$\begin{array}{r} 3 \\ +1 \\ \hline 4 \end{array} \qquad \begin{array}{r} 5 \\ +1 \\ \hline 6 \end{array} \qquad \begin{array}{r} 7 \\ +1 \\ \hline ? \end{array}$$

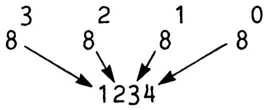
2.4.2 Overflow and Carry-In

The largest digit plus one results in a zero and a one carry to the next digit position:

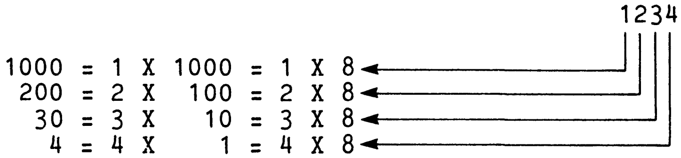
$$\begin{array}{r} 7 \\ +1 \\ \hline 10 \end{array}$$

2.4.3 Digit Position

The value of a position indicates its power of the base.



A digit's value depends upon its position within the number



Doesn't it look amazingly like decimal! Why shouldn't it? The digits are the same (as far as they go; there is no 8 or 9 in octal), and the rules are the same. If you have trouble accepting this, I think what is probably blowing your mind is the fact that:

8^2 = 100 not 64,
8^1 = 10 not 8,
8^3 = 1000 not 512.

2.4.3 Digit Position (Continued)

Just as a point in passing for those who have never seen it before, the distinct marks of the hexadecimal numbering system are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B

C, D, E, F.

2.5 CONVERTING NUMBERS BETWEEN BASES

Since most of us are used to working in decimal, yet the computer "speaks" binary, and octal is most often used as a compromise, we are going to have to know how to convert numbers of one base into other bases. So, let's establish the rules.

2.5.1 Converting from Decimal to Another Base

The procedure for converting a decimal number to some other base B is:

1. Divide the decimal number by B, and separate the answer into a quotient and a remainder.
2. Record the remainder.
3. Divide the quotient by B, and separate the answer into another quotient and a remainder.
4. Repeat Steps 2 and 3 until a quotient of 0 is obtained.
5. Record the remainders in the reverse order of their occurrence. The result is the converted number.

Examples: Convert to base 2.

$$21_{10} = 10101_2$$

2.5.1 Converting from Decimal to Another Base (Continued)

Example: Continued

$259_{10} = 100000011_2$

$$\begin{array}{r|l l l}
2 & \underline{259} & & \\
2 & \underline{129} & 1 & \uparrow \\
2 & \underline{64} & 1 & \Big| \\
2 & \underline{32} & 0 & \Big| \\
2 & \underline{16} & 0 & \Big| \\
2 & \underline{8} & 0 & \Big| \\
2 & \underline{4} & 0 & \Big| \\
2 & \underline{2} & 0 & \Big| \\
2 & \underline{1} & 0 & \Big| \\
 & 0 & 1 & \Big| \quad \text{reverse order of occurrence}
\end{array}$$

$17_{10} = 10001_2$

$$\begin{array}{r|l l l}
2 & \underline{17} & & \\
2 & \underline{8} & 1 & \uparrow \\
2 & \underline{4} & 0 & \Big| \\
2 & \underline{2} & 0 & \Big| \\
2 & \underline{1} & 0 & \Big| \\
 & 0 & 1 & \Big| \quad \text{reverse order of occurrence}
\end{array}$$

Try it yourself.

$39_{10} = \underline{\qquad}_2$

$$\begin{array}{r|l}
2 & \overline{39}
\end{array}$$

Try another one.

$123_{10} = \underline{\qquad}_2$

2.5.1 Converting from Decimal to Another Base (Continued)

Now let's take the same decimal numbers, and the same rules and convert to base 8.

$$21_{10} = 25_8$$

$$\begin{array}{r|l l} 8 & 21 & \\ 8 & 2 & 5 \uparrow \\ & 0 & 2 \end{array}$$

reverse order of occurrence

Proof:

$$25_8$$

$$\rightarrow 5 \text{ X } 8^{0}_{10} = 5 \text{ X } 1 = 5_{10}$$

$$\rightarrow 2 \text{ X } 8^{1}_{10} = 2 \text{ X } 8_{10} = \underline{16_{10}}$$

$$21_{10}$$

$$259_{10} = 403_8$$

$$\begin{array}{r|l l} 8 & 259 & \\ 8 & 32 & 3 \uparrow \\ 8 & 4 & 0 \\ & 0 & 4 \end{array}$$

reverse order of occurrence

Proof:

$$403_8$$

$$\rightarrow 3 \text{ X } 8^{0}_{10} = 3 \text{ X } 1 = 3_{10}$$

$$\rightarrow 4 \text{ X } 8^{2}_{10} = 4 \text{ X } 64 = \underline{256_{10}}$$

$$259_{10}$$

2.5.1 Converting from Decimal to Another Base (Continued)

$$17_{10} = 21_8$$

$$\begin{array}{r|ll} 8 & 17 & \\ 8 & 2 & 1 \uparrow \\ & 0 & 2 \end{array}$$

reverse order of occurrence

Proof:

$$21_8$$

$$\longrightarrow 1 \times 8^0 = 1 \times 1 = 1$$

$$\longrightarrow 2 \times 8^1 = 2 \times 8 = \underline{16}$$

$$17_{10}$$

Try it yourself.

$$39_{10} = \underline{\qquad\qquad}_8$$

$$8\overline{)39}$$

Try another one.

$$123_{10} = \underline{\qquad\qquad}_8$$

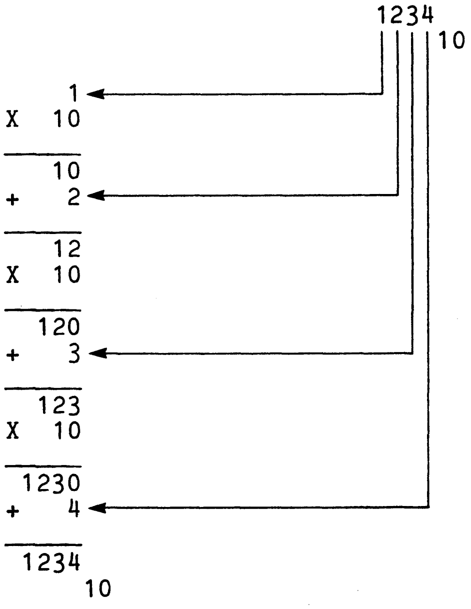
2.5.2 Converting from Other Bases to Decimal

Being able to convert decimal numbers into other bases may prove helpful if you have to enter information through the console data switches. By the same token, if you have to interpret the console data display, it may prove helpful to convert numbers from other bases B into their decimal equivalent.

The procedure for converting a base B number to decimal is:

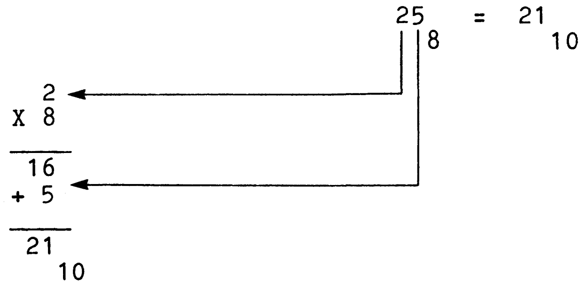
1. Start with the most significant digit.
2. Multiply by B.
3. To the result, add the next least significant digit.
4. Repeat Steps 2 and 3 until the B^0 digit gets added by Step 3. The last step in the sequence is always an addition.

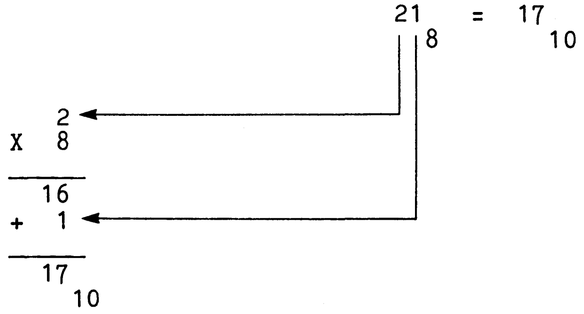
If the procedure works, the least it ought to do is convert base 10 numbers to decimal. Let's give it a try:

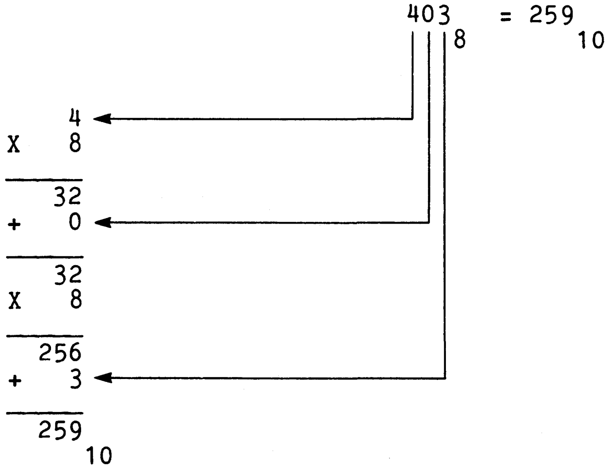


2.5.2 Converting from Other Bases to Decimal (Continued)

How about that, sports fans! If the sequence doesn't look familiar, look back on page 2-4. Now let's try it for binary and octal.







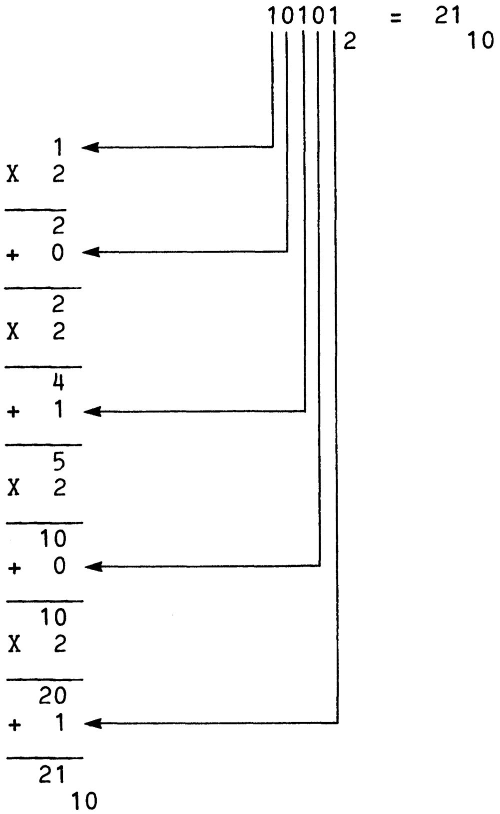
2.5.2 Converting from Other Bases to Decimal (Continued)

Now it's your turn. Use the numbers that you got as answers to the problems on page 2-11.

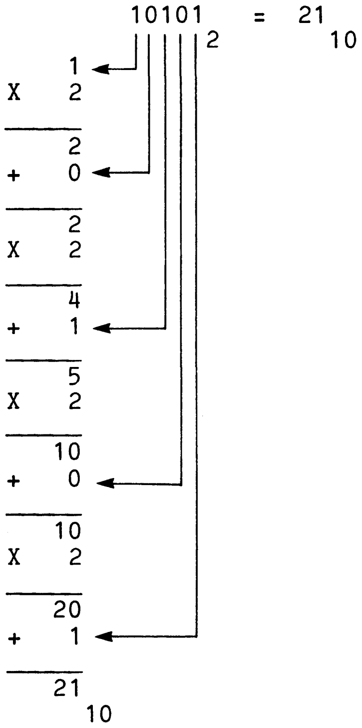
39_{10} = ________$_8$ and 123_{10} = ________$_8$

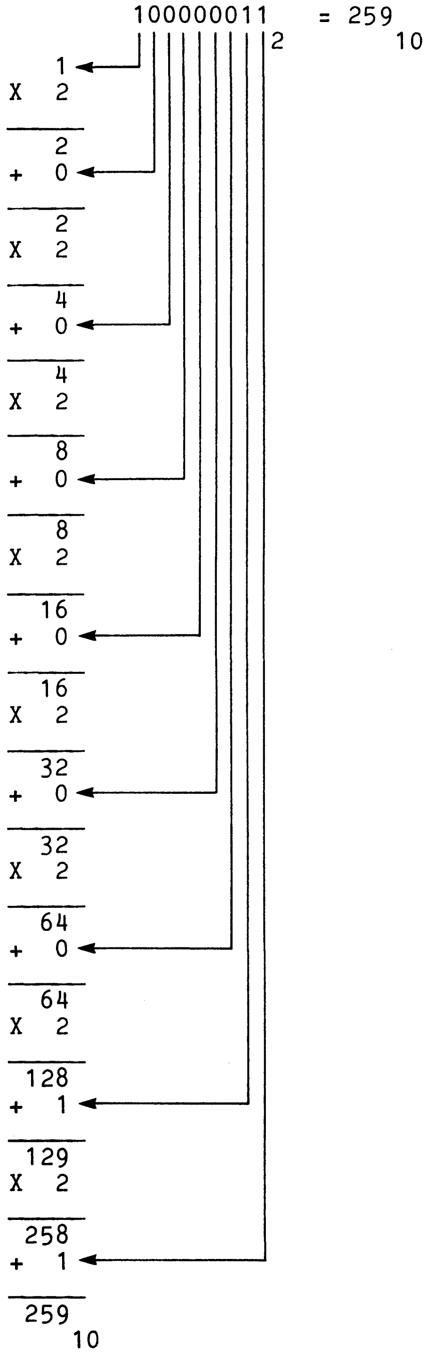
Convert the missing octal numbers back to decimal using the method shown above.

Given enough room, it works for binary also.



2.5.2 Converting from Other Bases to Decimal (Continued)





Now it's your turn. Use the numbers that you got as answers to the problems on page 2-11.

39_{10} = ___________$_2$ and 123_{10} = ________$_2$

Convert the missing octal numbers back to decimal by using the method shown above.

2.5.3
Converting Between Octal and Binary

Of the various conversions between bases, the one we use most often hasn't been discussed, yet it is the easiest conversion to do. The conversion of octal numbers to binary and vice versa is based on one simple fact: $2^3 = 8$. Let's examine the binary counting sequence alongside the octal counting sequence

Binary		Octal
0	=	0
+ 1		
1	=	1
+ 1		
10	=	2
+ 1		
11	=	3
+ 1		
100	=	4
+ 1		
101	=	5
+ 1		
110	=	6
+ 1		
111	=	7
+ 1		
1000	=	10

If we take the binary numbers that have been equated to octal numbers and append leading zeroes to make all binary numbers three-digit numbers, we observe the following:

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

$1\ 000 = 2^3 = 8 = 1\ 0$

2.5.3 Converting Between Octal and Binary (Continued)

Even more interesting, at the same time that the binary numbering system runs out of numbers that it can record using three digits, so the octal numbering system runs out of numbers that it can record with one digit. This three-to-one relationship is the whole key to binary to octal conversion. The only thing you have to "memorize" is the binary equivalent for 0 - 7_8 shown above. Most people would tell you, "It's as simple as one, two, three." But in this case, it's as simple as three-to-one.

Examples:

$$\begin{array}{ccccl} 100 & 011 & 010 & 001 & \\ \underline{\qquad} & \underline{\qquad} & \underline{\qquad} & \underline{\qquad} & {}^{2} \\ 100 & 011 & 010 & 001 & \\ 4 & 3 & 2 & 1 & \\ & & & & {}_{8} \end{array}$$

On the previous pages you converted the same decimal numbers into both binary and octal. Now, by the method shown above, see if your binary and octal answers agree with each other. Don't be afraid to add leading zeroes if necessary to maintain the three-to-one relationship.

1. $21_{10} = \underset{2}{\underline{010}}\ \underset{5_8}{\underline{101}}{}_2$

2. $259_{10} = \underline{\qquad\qquad}_8$
$= \underline{\qquad\qquad}_2$

3. $17_{10} = \underline{\qquad\qquad}_2$
$= \underline{\qquad\qquad}_8$

4. $39_{10} = \underline{\qquad\qquad}_8$

$= \underline{\qquad\qquad}_2$

2.5.3 Converting Between Octal and Binary (Continued)

5. $123_{10} = \underline{\hspace{3em}}_{2}$

   $= \underline{\hspace{3em}}_{8}$

2.6 ARITHMETIC

Now that we have become so adept at manipulating numbers from one base to another, let's try performing arithmetic operations on numbers of the same base.

2.6.1 Decimal Addition

The addition of binary numbers follows the same procedure as the more familiar addition of decimal numbers.

To add two decimal numbers, proceed as follows:

1. Add the right-most digit of each number to obtain a sum digit and a carry digit.
2. Record the sum digit.
3. Add the next right-most digit of each number, plus the carry digit left from the previous addition, and obtain another sum digit and carry digit.
4. Repeat Steps 2 and 3, proceeding from right to left, until all the digits have been added.
5. The number constructed from the individual sum digits is the final sum.

2.6.1 Decimal Addition (Continued)

Add the two decimal numbers 566 + 624.

$$\begin{array}{r}566\\624\\\hline\end{array}$$

6 + 4 = 10 where Carry = 1 Sum = 0

$$\begin{array}{r}1\\566\\624\\\hline 0\end{array}$$

1 + 6 + 2 = 9 where Carry = 0 Sum = 9

$$\begin{array}{r}01\\566\\624\\\hline 90\end{array}$$

0 + 5 + 6 = 11 where Carry = 1 Sum = 1

$$\begin{array}{r}101\\0566\\0624\\\hline 190\end{array}$$

1 + 0 + 0 = 1 where Carry = 0 Sum = 1

$$\begin{array}{r}101\\0566\\0624=\\\hline 1190\end{array}$$

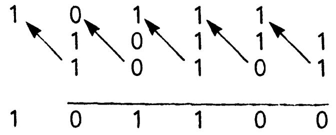
Thus 566 + 624 = 1190

2.6.2 Binary Addition

Binary addition follows exactly the same five steps used in decimal addition. But remember, the binary numbering system has only two digits (0 and 1). The following example examines the addition of all possible operands resulting from the addition of two binary numbers:

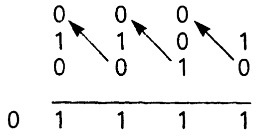
Carry	0	0	0	0	1	1	1	1
Bit A	+0	+0	+1	+1	+0	+0	+1	+1
Bit B	+0	+1	+0	+1	+0	+1	+0	+1
Carry Sum	00	01	01	10	01	10	10	11

The next example shows the normal method of keeping track of the sum and carry digits



Thus 10111 + 10101 = 101100.

Add the two binary numbers 1101 + 10.



Thus 1101 + 10 = 1111.

2.6.3 Overflow and Carry-In

If the number of digits in the answer exceeds the maximum allowable number of digits, the answer is said to overflow, and the left-most digit of the answer is called the overflow digit.

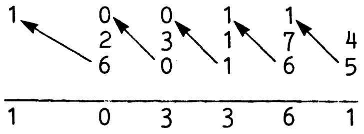
If, in the second example above, the maximum allowable number of digits is five, then there is an overflow, and the overflow digit is 1. In the third example above, if the maximum allowable number of digits is four, then there is no overflow, so the overflow digit is 0.

2.6.3 Overflow and Carry-In (Continued)

The procedure for performing octal addition is similar to that used for decimal and binary addition. Keep in mind that the octal numbering system has eight digits (0 through 7), and a carry occurs when the sum exceeds 7.

In the following examples, assume the maximum allowable number of digits is five.

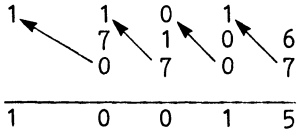
Add the two octal numbers 23174 + 60165.



Thus 23174 + 60165 = 103361.

NOTE: In this example, an overflow occurred.

Add the two octal numbers 7106 + 707.



Thus 7106 + 707 = 10015.

NOTE: In this example, no overflow occurred since the sum did not exceed the maximum allowable five digits.

2.6.4 SUBTRACTION

2.6.4.1 Complementary Arithmetic

In the previous section, the concept of "maximum allowable number of digits" was introduced. This concept is of great importance in the understanding of complementary arithmetic.

2.6.4.1 Complementary Arithmetic

If the maximum allowable number of digits is six, for example, then the decimal numbers

```
        0 9 8 6 3 2 7
and     1 9 8 6 3 2 7
```

represent the same magnitude since the left-most digit is an indication of overflow, and adds nothing to the value of the right-most, maximum six digits.

The normal counting sequence from zero is as follows:

```
   0 0 0 0 0 0
   0 0 0 0 0 1
   0 0 0 0 0 2
   0 0 0 0 0 3
        .
        .
        .
   9 9 9 9 9 7
   9 9 9 9 9 8
   9 9 9 9 9 9
 1 0 0 0 0 0 0
 1 0 0 0 0 0 1
```

Notice that if 1 is added to the largest number, 999999, zero is obtained and the normal counting sequence is recycled.

What happens if the counting sequence is reversed?

```
        .
        .
        .
   0 0 0 0 0 3
   0 0 0 0 0 2
   0 0 0 0 0 1
   0 0 0 0 0 0
   9 9 9 9 9 9
   9 9 9 9 9 8
   9 9 9 9 9 7
        .
        .
        .
```

Here, notice that when 1 is subtracted from zero, the number simply cycles back to 999999. To our way of thinking, 1 subtracted from zero is a minus 1, or a negative one. Regardless of what you call it, in a cyclic system of

2.6.4.1 Complementary Arithmetic (Continued)

counting, numbers equidistant from either side of zero are referred to as complementary numbers. Therefore, in the six digit system shown here, 1 and 999999 (-1) are complementary numbers, 2 and 999998 (or -2) are complementary numbers. As a matter of fact, any pair of numbers which added together total zero (in a cyclic system) are complementary numbers.

To obtain the complement of a number, it is not necessary to count forwards and backwards from 000000. Simply subtract the number from the largest possible number +1.

For the six-digit maximum numbers used here, the largest possible number is 999999, and the largest possible number +1 is 1000000.

Find the complement of 000004.

```
1 0 0 0 0 0 0 (largest possible number +1)
- 0 0 0 0 0 4 (minus the number)
-------------
  9 9 9 9 9 6 (complement of the number)
```

Find the complement of 923156.

```
1 0 0 0 0 0 0
- 9 2 3 1 5 6 (number)
-------------
  0 7 6 8 4 4 (complement)
```

Find the complement of 000000.

```
1 0 0 0 0 0 0
- 0 0 0 0 0 0
-------------
1 0 0 0 0 0 0
```

2.6.4.2 Ten's Complement

The complementary numbers obtained in the preceeding examples are more correctly referred to as the 10's (ten's) complement of the number (999996 is the 10's complement of 000004, etc.). This further description of the complement is used to indicate the value from which the number was subtracted to obtain the complement. The complement of a six-digit number is obtained by subtracting that number from 10^6. This value is the next power of the base (in this case, base 10).

It is interesting to note that the original number was subtracted from the largest possible number +1 in order to obtain the 10's complement. The same result could be obtained if the number is subtracted from the largest possible number, and 1 added to the answer.

Find the 10's complement of 923156.

$$\begin{array}{rcccccl} & 9 & 9 & 9 & 9 & 9 & 9 \quad \text{(largest possible number)} \\ - & 9 & 2 & 3 & 1 & 5 & 6 \quad \text{(number)} \\ \hline & 0 & 7 & 6 & 8 & 4 & 3 \\ & & & & & + & 1 \quad \text{(plus 1)} \\ \hline & 0 & 7 & 6 & 8 & 4 & 4 \quad \text{(10's complement)} \end{array}$$

NOTE: 076843 is known as the 9's complement of 923156.

Therefore, an easier method of finding the 10's complement of a number is as follows:

10's complement of X = 9's complement of X plus 1

Find the 10's complement of 000000.

2.6.4.2 Ten's Complement (Continued)

$$\begin{array}{ccccccc l}
 & 9 & 9 & 9 & 9 & 9 & 9 & \text{(largest possible number)}\\
- & 0 & 0 & 0 & 0 & 0 & 0 & \text{(X)}\\
\hline
 & 9 & 9 & 9 & 9 & 9 & 9 & \text{(9's complement of X)}\\
 & & & & & + & 1 & \text{(plus 1)}\\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & \text{(10's complement of X)}
\end{array}$$

2.6.4.3 Two's Complement

Now let's apply the general rules of complementation to the binary number system. In the binary number system, the complement desired is the <u>2's complement</u> of the number.

In the following examples, assume that the maximum allowable number of BITs (BInary digiTs) is 7.

Find the 2's complement of 0000011.

$$\begin{array}{cccccccc l}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{(largest possible}\\
- & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \text{number* +1)}\\
\hline
 & & & & & & & & \text{(2's complement}\\
 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & \text{or X**)}
\end{array}$$

* The largest possible number is 1111111.
** Direct binary subtraction follows the same rules as direct decimal subtraction: 0-0 = 0; 1-0 = 1; 1-1 = 0; 0-1 = 1 and borrow 1.

But as was shown before, it is possible to subtract the number from the largest possible number and add 1 to the result.

Find the 2's complement of 0000011.

$$\begin{array}{cccccccc l}
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \text{(largest possible number)}\\
- & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \text{(X)}\\
\hline
 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & \\
 & & & & & & + & 1 & \text{(plus 1)}\\
\hline
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \text{(2's complement)}
\end{array}$$

NOTE: 111110 is known as the 1's complement of 0000011.

2.6.4.3
Two's
Complement
(Continued)

Therefore, the 2's complement of a binary number may be obtained as follows:

2's complement of X = 1's complement of X, plus 1

Find the 2's complement of 1011101.

```
  1  1  1  1  1  1  1    (largest possible
                          number)
- 1  0  1  1  1  0  1    (X)
---------------------
  0  1  0  0  0  1  0    (1's complement of X)
                 +  1    (plus 1)
---------------------
  0  1  0  0  0  1  1    (2's complement of X)
```

Find the 2's complement of 0000000

```
  1  1  1  1  1  1  1    (largest possible
                          number)
- 0  0  0  0  0  0  0    (X)
---------------------
  1  1  1  1  1  1  1    (1's complement of X)
                 +  1    (plus 1)
---------------------
  1  0  0  0  0  0  0    (2's complement of X)
```

Looking closely at the 1's complements of the numbers in the last three examples, we see that the 1's complement of the number is the number with all the 0s changed to 1s and the 1s changed to 0s.

Find the 2's complement of 1110110.

```
  1  1  1  0  1  1  0    (X)
  0  0  0  1  0  0  1    (1's complement of X)
                 +  1    (plus 1)
---------------------
  0  0  0  1  0  1  0    (2's complement of X)
```

Find the 2's complement of 0000000.

```
  0  0  0  0  0  0  0    (X)
  1  1  1  1  1  1  1    (1's complement of X)
                 +  1    (plus 1)
---------------------

1 0  0  0  0  0  0  0    (2's complement of X)
```

2.6.4.4 Eight's Complement

Applying the rules of complementation to octal numbers, we see that the 8's complement of a number is the 7's complement of the number, plus 1.

Find the 8's complement of 77341.

```
  7 7 7 7 7      (largest possible number)
- 7 7 3 4 1      (X)
-----------
  0 0 4 3 6      (7's complement of X)
        + 1      (plus 1)
-----------
  0 0 4 3 7      (8's complement of X)
```

Find the 8's complement of 00000.

```
  7 7 7 7 7
- 0 0 0 0 0
-----------
  7 7 7 7 7      (7's complement)
        + 1
-----------
  1 0 0 0 0      (8's complement)
```

2.6.5 Binary Subtraction

By employing the techniques of complementary arithmetic, it is possible to effect a subtraction using the addition process.

To perform A - B, either of two methods may be used:

1. Direct subtraction of B from A; or

2. The addition of A to the complement of B.

When using method 2, both A and B must contain the same number of digits (use leading zeroes where necessary) and the answer is contained in the same number of digits (ignore the overflow digit if it occurs).

Perform $783_{10} - 25_{10}$

Method 1:

```
 783
 -25
 ---
 758
```

783-25 = 758

2.6.5 Binary Subtraction (Continued)

Method 2:

$999 - 25 = 974$ (9's complement of 25)

$974 + 1 = 975$ (10's complement of 25)

$$\begin{array}{rl} 7\ 8\ 3 & \text{(A)} \\ +\ 9\ 7\ 5 & \text{(10's complement of B)} \\ \hline 1\ 7\ 5\ 8 & \end{array}$$

$783 - 25 = 758$

Perform $1101101_2 - 1011_2$

NOTE: First add leading 0s to make numbers the same length.

Thus we are to perform $1101101_2 - 0001011_2$

$$\begin{array}{rl} 1\ 1\ 0\ 1\ 1\ 0\ 1 & \text{(A)} \\ +\ 1\ 1\ 1\ 0\ 1\ 0\ 1 & \text{(2's complement of B)} \\ \hline 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0 & \end{array}$$

$1101101 - 1011 = 1100010$

Perform $101011_2 - 101011_2$ (A-A)

$$\begin{array}{rl} 1\ 0\ 1\ 0\ 1\ 1 & \text{(A)} \\ +\ 0\ 1\ 0\ 1\ 0\ 1 & \text{(2's complement of A)} \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 0 & \end{array}$$

Thus, a number plus its complement always equals zero. This is an easy way to confirm that you have the correct complement of a number.

2.6.6 Octal Subtraction

Octal subtraction (A - B) may be performed by adding A to the 8's complement of B.

Perform $6275_8 - 31_8$

Add leading 0's $6275_8 - 0031_8$

```
   6 2 7 5      (A)
 + 7 7 4 7      (8's complement of B)
 ---------
 1 6 2 4 4
```

6275 - 31 = 6244

Perform $7000_8 - 76_8$

```
  7 0 0 0       (A)
+ 7 7 0 2       (8's complement of B)
---------
1 6 7 0 2
```

7000 - 76 = 6702

2.7 SIGNED NUMBER REPRESENTATION

2.7.1 Sign Bit Definition

In many applications where the use of both positive and negative numbers is required, some method to indicate the sign of the number must be employed. In written text, this is done with the + and - signs. The computer, however, works with binary numbers and would not easily recognize a + or - sign. Another method must be used to indicate the sign of the number. One possibility is to define the left-most bit of the binary number as the sign indicator or sign bit. A one (1) in this position indicates that the number represented by the bits to the right is negative; a zero (0) indicates that the number is positive. Using this technique of signed number representation, the sign bit is followed by the absolute value of the number. Another method of representing signed numbers employes the concept of

2.7.1
Sign Bit
Definition
(Continued)

complementary numbers, as described in the previous section. It is this last method that will be pursued further here.

If the maximum allowable number of bits is 4, then the following numbers are possible:

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

This set of 16 numbers is cyclic because adding 1 to 1111 brings us back to 0000.

Also, subtracting 1 from 0000, gives us 1111. If this set of numbers is said to contain only positive values, then the range of values is:

0 0 0 0 through 1 1 1 1

or

0_{10} through 15_{10}

Suppose we divide this set in half, and define one half as representing positive values, and the other half negative values (column A). Also, let's restack the set so that 0000 is at the center (column B). Column C represents the decimal equivalent of column B.

2.7.1 Signed Bit Definition (Continued)

A		B	C
0 0 0 0	Positive Numbers	0 1 1 1	7
0 0 0 1		0 1 1 0	6
0 0 1 0		0 1 0 1	5
0 0 1 1		0 1 0 0	4
0 1 0 0		0 0 1 1	3
0 1 0 1		0 0 1 0	2
0 1 1 0		0 0 0 1	1
0 1 1 1		0 0 0 0	0
1 0 0 0	Negative Numbers	1 1 1 1	-1
1 0 0 1		1 1 1 0	-2
1 0 1 0		1 1 0 1	-3
1 0 1 1		1 1 0 0	-4
1 1 0 0		1 0 1 1	-5
1 1 0 1		1 0 1 0	-6
1 1 1 0		1 0 0 1	-7
1 1 1 1		1 0 0 0	-8

Notice that all the negative numbers have a 1 in the left-most bit position and all the positive numbers have a 0 in the left-most bit position. Thus, if 0000 is defined as a positive number, there is the same quantity of positive and negative values.

NOTE: If the programmer is using the left-most bit for sign definition, care should be taken not to overflow the range of values.

Perform 5 + (-4)

$$\begin{array}{r} 5 \\ +\ (-4) \\ \hline 1 \end{array} \qquad \begin{array}{r} 0101 \\ +\ \ 1100 \\ \hline 1\ \ 0001 \end{array}$$

Perform 6 + (-6)

$$\begin{array}{r} 6 \\ +\ (-6) \\ \hline 0 \end{array} \qquad \begin{array}{r} 0110 \\ +\ \ 1010 \\ \hline 1\ \ 0000 \end{array}$$

Perform 7 + 2

$$\begin{array}{r} 7 \\ +\ \ 2 \\ \hline 9 \end{array} \qquad \begin{array}{r} 0111 \\ +\ \ 0010 \\ \hline 0\ \ 1001 \end{array}$$

2.7.1 Signed Bit Definition (Continued)

Note that in this example the desired result was not obtained because the range has been exceeded. 1001 represents -7, not +9.

2.7.2 Range of Signed Numbers

In the 4-bit number set of the previous section, the range of unsigned numbers is as follows:

0000 through 1111
or 0_{10} through 15_{10}
or 0_{8} through 17_{8}

If a number set contains 16-bit numbers, the ranges are as follows:

<u>UNSIGNED</u>

0000000000000000 through 1111111111111111

or

0_{10} through $65{,}535_{10}$

or

0_{8} through 177777_{8}

<u>SIGNED</u>

1000000000000000 through 0111111111111111

or

$-32{,}768_{10}$ through $+32{,}767_{10}$

or

-100000_{8} through $+077777_{8}$

2.7.2 Range of Signed Numbers (Continued)

If a number set contains 8-bit numbers, the ranges are as follows:

UNSIGNED

00000000 through 11111111

or

0_{10} through 255_{10}

or

0_8 through 377_8

SIGNED

10000000 through 01111111

or

-128_{10} through $+127_{10}$

or

-200_8 through $+177_8$

2.8 LOGICAL AND

In the binary number system, additional operations exist over and above addition, subtraction, multiplication, and division. These additional operations are known as logical or Boolean operations.

One such logical operation is the AND function.

Consider the drawbridge in the following figure:

The bridge consists of two spans that can be opened: A and B. Obviously, the path across this bridge is continuous only if both A and B are closed.

2.8 LOGICAL AND (Continued)

SPAN A	SPAN B	BRIDGE
OPEN	OPEN	OPEN
OPEN	CLOSED	OPEN
CLOSED	OPEN	OPEN
CLOSED	CLOSED	CLOSED

If the two states of each span are assigned the binary values OPEN = 0 and CLOSED = 1, this can be rewritten.

A	B	A AND B A . B A ^ B
0	0	0
0	1	0
1	0	0
1	1	1

Two binary numbers can be ANDed by simply ANDing respective bits from each number.

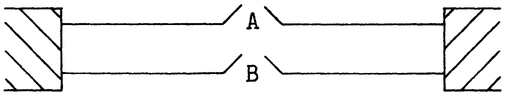
```
                          *
Perform           10111011 ^  00011011

1 0 1 1 1 0 1 1           (A)
0 0 0 1 1 0 1 1           (B)
_______________
0 0 0 1 1 0 1 1           (A . B)
```

NOTE: Both corresponding bits in A and B must be 1 for the resulting bit A . B to be a 1.

2.9 LOGICAL OR

Consider two drawbridges spanning a river as shown below:



*^ = logical AND symbol.

2.9
LOGICAL OR
(Continued)

A path from one side of the river to the other exists if A OR B or both is closed.

SPAN A	SPAN B	PATH
OPEN	OPEN	OPEN
OPEN	CLOSED	CLOSED
CLOSED	OPEN	CLOSED
CLOSED	CLOSED	CLOSED

If we assign binary values to the states of each drawbridge, this can be rewritten as follows:

A	B	A OR B A + B A V B
0	0	0
0	1	1
1	0	1
1	1	1

Notice that with the OR operation, if either of the corresponding bits in A or B is a 1, the resulting bit (A + B) is a 1.

Two binary numbers can be ORed by simply ORing respective bits from each number.

Perform 10111011 V* 00011011

1 0 1 1 1 0 1 1	(A)
0 0 0 1 1 0 1 1	(B)
1 0 1 1 1 0 1 1	

Perform 10111011 V 01000100

This is equivalent to A V 1's complement of A.

1 0 1 1 1 0 1 1	(A)
0 1 0 0 0 1 0 0	(1's complement of A)
1 1 1 1 1 1 1 1	(A + 1's complement of A)

* V = logical inclusive OR symbol.

2.10 LOGICAL EXCLUSIVE OR

The logical OR function described in the previous section is more precisely known as the logical _inclusive OR_ function.

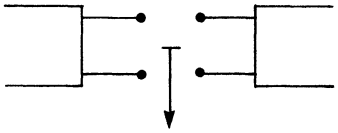
The _exclusive OR_ function can be defined as follows: The resulting bit of A + B is a 1 if bit A does not equal bit B.

Bit A	0	0	1	1
Bit B	0	1	0	1
A + B	0	1	1	0

Perform 10111011 $\oplus$* 00011011

1 0 1 1 1 0 1 1 (A)
0 0 0 1 1 0 1 1 (B)

1 0 1 0 0 0 0 0 ($A \oplus B$)



* $\oplus$ = logical exclusive OR symbol.

CHAPTER 3

PROGRAMMING FUNDAMENTALS AND BASIC CONCEPTS

Now that we know what a computer is, and the most elementary steps of talking to the computer, it's time to start building on this until we can get the computer to do what we want it to do.

3.1 COMPUTER PROGRAM

The job that we want the computer to do is called the computer program. The procedure for writing computer programs can be broken down into five parts:

1. Problem definition.
2. Formulation of an algorithm for solving the problem.
3. Structuring of a detailed flowchart solution to the problem.
4. Translation of the detailed solution into a computer programming language.
5. Testing and debugging the computer program.

The assemblage of information from these five steps constitutes the documentation of the program. From this documentation the definition of problem could become a program abstract, an entry in a library for use by others attempting to accomplish the same task.

The algorithm and flowchart are of particular use to you when you write the program. They help to ensure that all phases of the problem are covered by instructions. The algorithm and flowchart are also of particular interest to you or the maintenance programmer who - six months from now - has to remember or figure out what a particular block of instructions might be doing, as a "bug" has been discovered in the program and a fix must be effected.

Now let's look at each of the five steps in more detail.

3.1.1 Problem Definition

The definition of the problem should not be bypassed as a trivial step. In many instances, avoiding this preliminary step results in wasted time and effort on future steps. The purpose of the program must be known before proceeding.

The definition should be explicit and complete; state how many, or what to do if some phase cannot be completed, or, as encountered, does not conform.

Consider the following problem definitions in terms of the criteria just described.

1. Paint a room.
2. Sort 25 numbers.
3. Change a tire.
4. Convert binary numbers into hexadecimal equivalents.

Obviously, these statements do not qualify as definitions of problems to be solved. But, let's take one of the statements and further define it until it does qualify as the definition of a problem.

Changing a tire.

The tire is mounted on a car. The car is in your garage. There is a working bumper jack available and a spare tire in good shape - full of air. Remove the tire that is mounted on the car and replace it with the available spare.

3.1.1 Problem Definition (Continued)

Try another one.

Sort 25 numbers.

Given a table of known size containing random positive entries, sort the entries into ascending order. First, keep a copy of the original table; then perform the Sort on the original table.

Of the two problem definitions given above, the first one is beyond the scope of this book in terms of Steps 4 and 5 of writing computer programs. However, the second one (Sort 25 numbers.) could be carried through Step 4 (translation into programming language) and would still be within the scope of this book.

3.1.2 Algorithm

The algorithm is a step-by-step sequence to the solution of a problem. It should account for every possible condition, including any foreseeable what-ifs.

One thing you should bear in mind: there is seldom, if ever, just one solution to a problem. If the same problem were given to fifteen programmers, there would likely be fifteen algorithms to the solution of that problem. So what follows is just one programmer's solution to the problem. It may not be the best solution, but its purpose here is just to show you examples of algorithms.

Changing a Tire

1. Get the spare tire from the spare tire mount.
2. Get the bumper jack and assemble it at the corner of the car closest to the tire to be changed.
3. Secure the car, so that the car may be jacked up without danger of rolling.

3.1.2
Algorithm
(Continued)

4. Remove the wheel cover and check the ends of the lugs for a stamping of L or R, indicating a left- or right-hand thread.

5. Jack the car up enough so that the pressure is off the lug nuts but the tire is still on the ground.

6. Loosen the lug nuts (left-hand thread loosens clockwise, right-hand thread counter-clockwise) about 1/4 to 1/2 turn each.

7. Now jack the car up until the tire clears the ground.

8. Remove the lug nuts the rest of the way.

9. Remove the tire.

10. Place the spare tire on over the lugs.

11. Replace the lug nuts snugly, tightening them in a 1-3-2-4 pattern for four lugs, or a 1-3-5-2-4 pattern for five lugs. Do not tighten excessively at this time, since the force required might cause the car to slip off the jack.

12. Lower the car until the tire has good traction on the ground, but is not bearing the total weight that it will receive.

13. Now finish tightening the lug nuts in the pattern established in Step 11.

14. Lower the car the rest of the way and remove the bumper jack.

15. Replace the wheel cover.

16. Return the bumper jack from whence it came.

3.1.2 Algorithm (Continued)

Sort

1. Set up pointers and counters.
2. Transfer entry from Table 1 to Table 2.
3. Repeat Step 2 until Table 2 looks like Table 1.
4. Get first two entries from Table 1.
5. Put the smaller of the two entries into the first position of Table 1.
6. If the last entry has not been tested and positioned, get the next entry.
7. Test each entry against the larger of the two values from the previous test.
8. Put the smaller of the two entries into the next sequential position of Table 1.
9. When the largest value is in the last position, reduce the size of Table 1 by one and go to Step 4.
10. When the reduction of Step 9 indicates that Table 1 is only one entry, you are done.

3.1.3 Flowchart

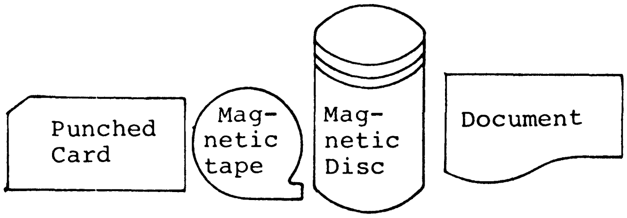
The flowchart differs from the algorithm in that the flowchart goes deeper into exactly how the problem is going to be solved. While the algorithm is general enough to apply to anybody's computer, the flowchart shows evidence of one computer's instruction set.

3.1.3
Flowchart
(Continued)

Flowcharting is a language of symbols with English and mathematical statements within joined lines. Flowcharting may be broken down into two categories: system flowcharting and program flowcharting.

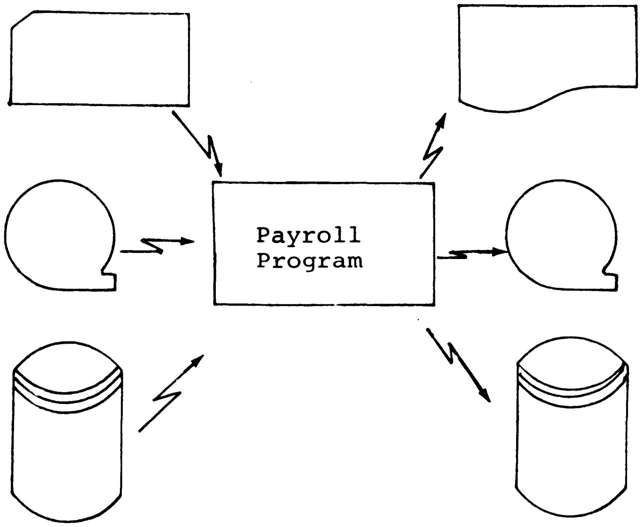
System flowcharting consists of peripheral devices represented by symbols with interconnections to show the relationship of the device to the overall program. The system flowchart is helpful to you and the maintenance programmer because it presents a big-picture overview of what the program is going to do. It also serves as a reminder to you of which devices the program uses to communicate.

Some of the symbols used in system flowcharting are shown below.



3.1.3
Flowchart
(Continued)

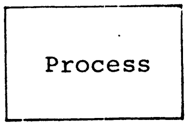
These symbols may be combined to show, for instance, a payroll program.



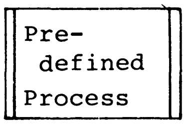
For the payroll program, the program itself "lives" on the disc, the employee's old records are on the magnetic tape, and the weekly time sheets are converted into punched cards and fed into the program. The program then prints the employee's check, updates his year-to-date information on the magnetic tape, and keeps a copy of this run of the program on the disc.

Program flowcharting consists of brief statements and questions within different shaped boxes to graphically illustrate the logical flow of the program. Some of the symbols used for this purpose are shown on page 3-8.

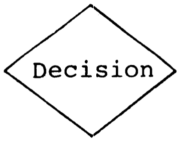
3.1.3
Flowchart
(Continued)



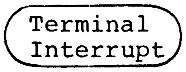
Defined operation(s) causing change in value, form, or location of information.



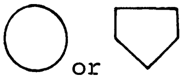
Operations or program steps specified in a subroutine or another set of flowcharts.



An operation that determines which of a number of alternative paths to be followed.



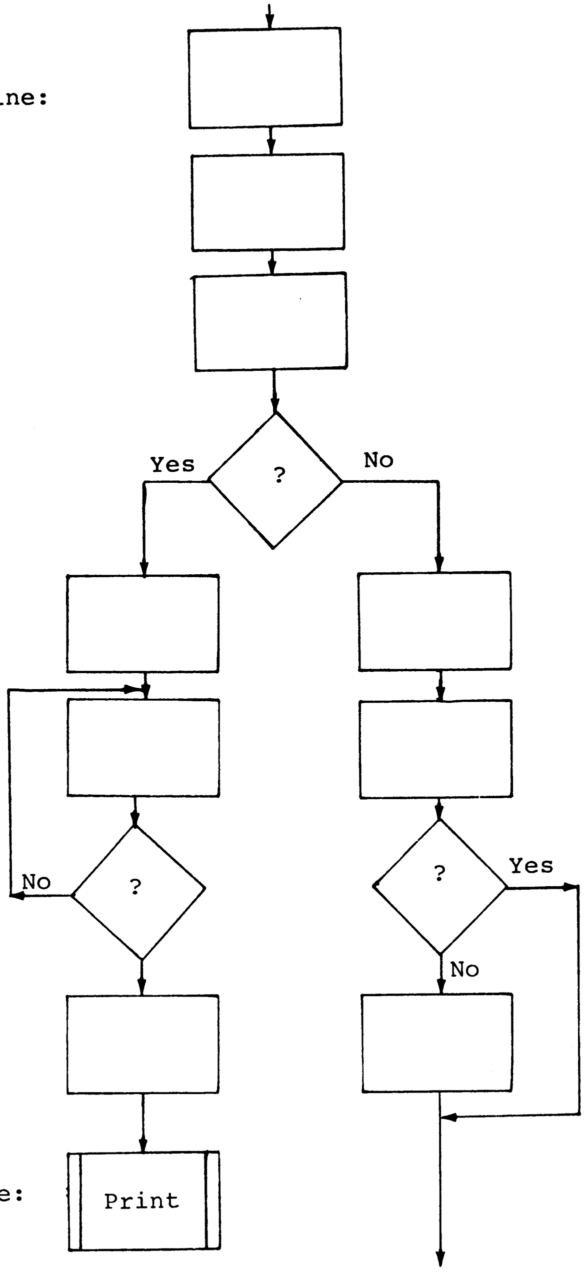
Indicates start, stop, halt, delay, or interrupt; may show exit from a closed subroutine.



Exit to, or entry from, another part of the chart or page.

Novices take note: Regardless of the complexity of the program, using the symbols just presented, it can be broken down into one or a combination of the following types of program flow.

Program Flow



1. Straight line:

2. Branching:

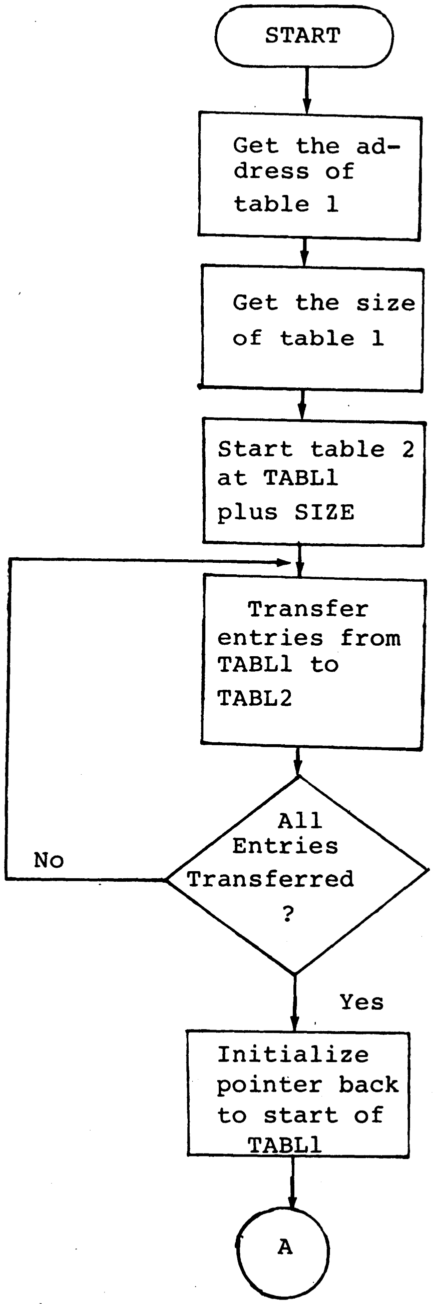
3. Looping:

4. Subroutine:

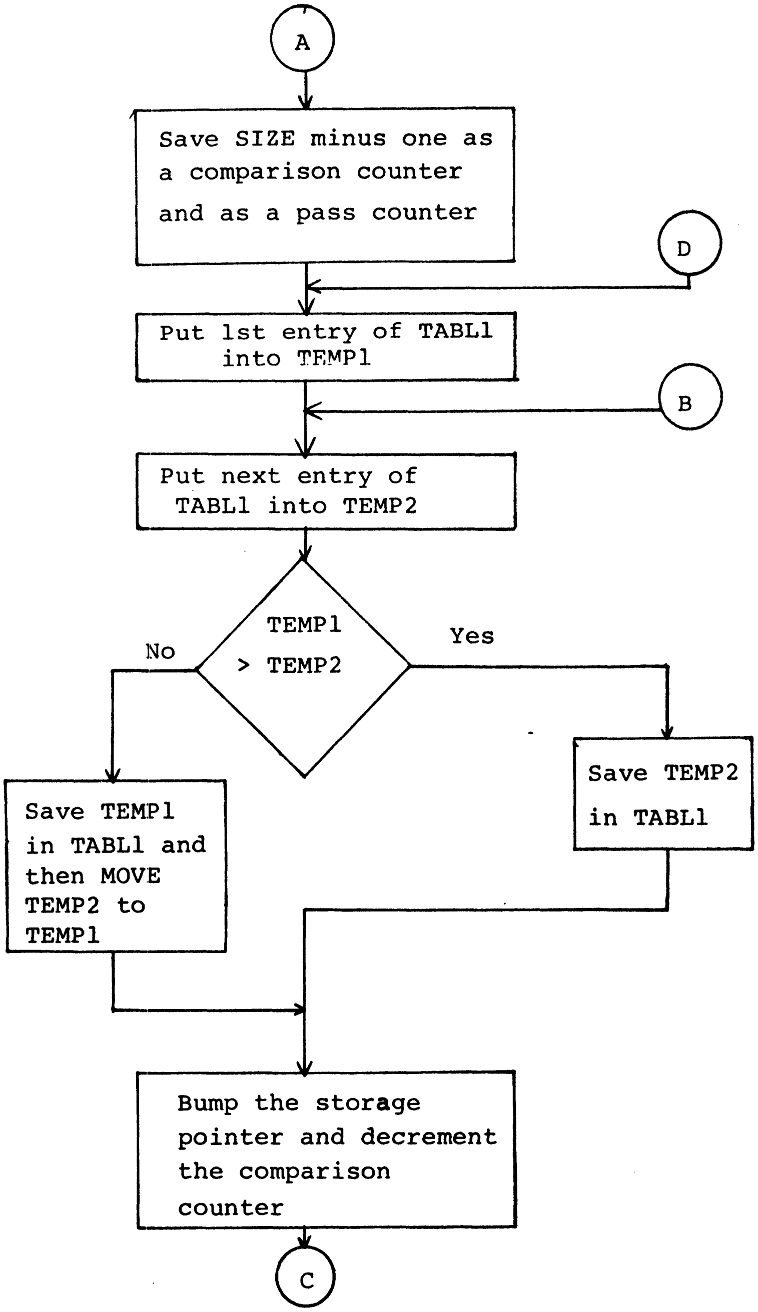
3.1.3
Flowchart
(Continued)

For our example of program flow, let's examine the flowchart for the Sort routine introduced during problem definition and algorithm discussions.

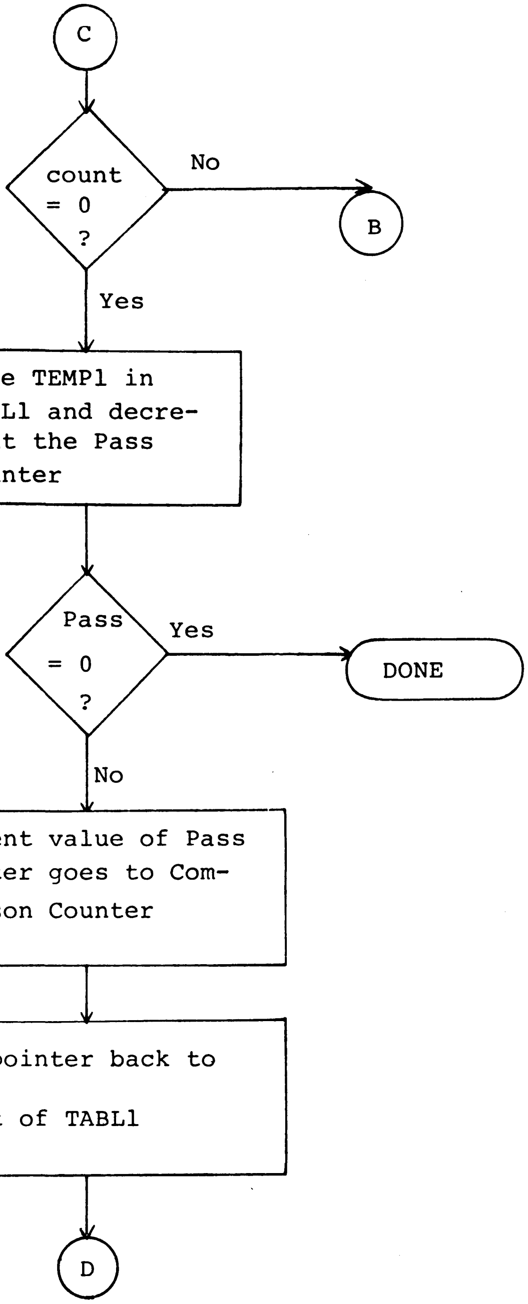
3.1.3
Flowchart
(Continued)



3.1.3 Flowchart (Continued)



3.1.3
Flowchart
(Continued)



3.1.3 Flowchart (Continued)

In the flowchart just presented, the process block tends to be on the wordy side. This was done intentionally so that there would be no misunderstanding of the intention of the block. However, it is not a bad practice even when you are writing it to yourself. You would be surprised how much of a program you can forget after six months; and then there's the maintenance programmer who has never seen the program before.

3.2 BASIC CONCEPTS

Some of the basic concepts of programming are so simple (here it comes) that most books wouldn't even mention them; these are things that we do automatically, like saying "nine plus one equals ten" when we know it is really zero with a one carry. These concepts are so basic that we often overlook them in the flowcharting stage, and then too often neglect them in the first pass of the coding stage. Three such basic concepts are: tables, pointers, and counters.

3.2.1 Tables

A _table_ is a collection of similar data generally stored in sequential locations. Examples might be a table of random positive numbers, or a table of ASCII characters, or a table of addresses to various subroutines.

3.2.2 Pointers

A _pointer_ is an indicator of where the table lives, or which was the last entry referenced. If a pointer is going to be used over and over, it generally does not want to be altered. In this case, the programmer will obtain a copy of the pointer, place it in a temporary storage cell, and work on it there, to preserve the original pointer. This would be true in the case of one program building a table from a given start address, and a second program operating on the same table of operands. If the first program destroys the pointer, then the second program will have either no data or the wrong data.

3.2.2 Pointers (Continued)

Relating this to the Sort routine, the very first step tells us to "get the address of TABL1." Here we are getting a copy of the pointer to the beginning of TABL1. Later on, when the flowchart indicates "save TEMP2 in TABL1" and "Bump the storage pointer," we are incrementing our copy of the pointer. This technique always keeps the original pointer intact.

3.2.3 Counters

A counter is an indicator of how many. For our purposes we will consider two types of counters: event counters and iteration counters. An event counter starts at zero and increments by one each time the event takes place. This was the case when the original table was built. As each entry was made, an event counter was incremented so that when the table is complete, the event counter is an indicator of the size of the table.

The other type of counter is an iteration counter. An iteration counter tells you how many times to perform an operation or process. This can be done by starting with a specific value and decrementing the iteration counter to zero, or by starting at zero and incrementing until a predetermined value is reached. In the Sort routine, three such counters were used. The state of the first counter is being tested when the flowchart asks, "All entries transferred?" The second wants to know if the comparison "Count=0?" and the third if "Pass = 0?"

Further discussion of basic programming concepts appears in later chapters where individual instructions or small routines can better demonstrate the concept.

3.2.3 Counters

At this time let's pause and take inventory of where we've been, and where we're going. First, we looked at the computer as a machine and got a bit of a feel for what the machine needed (a program counter, an instruction register, a console, memory and peripherals) to perform tasks for us. Secondly, we looked at the language (binary numbering systems) that the computer could understand. Thirdly, we looked at the elementary phases of program development. The next step is the instruction set. By placing various combinations of ones and zeros in the instruction register, we can get the computer to execute elementary operations, the sum total of which will be our program. Rather than having to enter these instructions in the form of ones and zeros, we develop software to facilitate the job.

The program development software will consist of a text editor to allow us to generate the source program and make corrections, deletions, and insertions where we need them without having to rewrite the entire program. The next phase of program development software is the assembler. One job of the assembler is to convert our instruction mnemonics (symbols that are easier for us to remember than binary ones and zeroes) into the language that the computer understands: binary. From the assembler phase we will go to the binary loaders. The binary loader is a program to read and decode the information into its correct location in memory. After our program is loaded, like all good programs, it never runs the first time! This is where we use the program development aid called debugger. The debugger allows us to run portions of our program and check results dynamically, and where necessary make corrections, deletions, and insertions dynamically. Chapter 4 deals with the next phase of program development, the instruction set.

CHAPTER 4

THE INSTRUCTION SET *

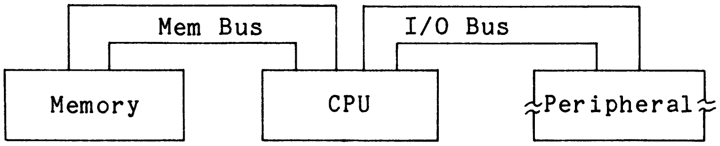
Back in Chapter 1, when we first established what a computer is, we spoke of it as having three main sections:

1. Central Processing Unit (CPU) - Where all data manipulation takes place.

2. Main Memory - Where the instructions are stored. Also where tables of addresses and operands may be stored.

3. Input/Output (I/O) - The CPU's link to its environment.

These three main sections are linked to each other as shown below.



Corresponding to these three main sections, the instruction set may be divided into three categories according to the sections with which they are primarily concerned. The three categories, and the operations of each, are outlined below:

1. Input/Output. Operations involve:

    a. Starting and stopping a peripheral device.
    b. Transfer of data from the device to an accumulator in the CPU.

---

* When reading this chapter, the reader should refer to his Programmer's Reference Card.

    c. Transfer of data from an accumulator in the CPU to the device.
    d. Testing the status of the device.

2. Memory Reference Instructions (MRI). Operations involve:

    a. Modifying the Program Counter (PC).
    b. Modifying an operand in memory.
    c. Transfer of data from memory to an accumulator.
    d. Transfer of data from an accumulator to memory.

3. Arithmetic-Logic Class (ALC). Performs data manipulations between the accumulators.

Each instruction within the instruction set consists of a string of 16 bits or binary digits, numbered 0 through 15. These sixteen bits make up a computer "word." Each of the three categories of instructions has its own unique "word" format as outlined below.

4.1 I/O INSTRUCTIONS

We will look at the I/O instructions first, for two reasons. First, the majority of information that enters the CPU (and then memory) comes from I/O devices. And secondly, by choosing the Teletype as an I/O device we can see some mechanical reaction to our instructions.

To understand some of the restrictions or limitations of the I/O instructions, let's begin by looking at an I/O instruction as it appears in the instruction register (IR). In terms of the instruction register (IR), every I/O instruction has the following format:

011	AC	TRANSFER	CONTROL	DEVICE CODE
0 2	3 4	5 7	8 9	10 15

4.1
I/O
INSTRUCTIONS
(Continued)

Any transferring of data is done between a particular device and a particular accumulator. The accumulator involved is specified by bits 3 and 4. The device involved is specified by the device code in bits 10 through 15. Bits 10 through 15 decode to 64 unique possibilities; however, only 62_{10} devices may be addressed (01_8 through 76_8). Device code 00 is not used, and 77_8 is a special function code denoting the CPU.* In a device, there may be up to three data buffers (A, B, and C). Bits 5 through 7, the transfer field, specify the buffer involved and the direction of the data transfer, whether IN or OUT. An IN transfer implies a data transfer from the device buffer to the processor. An OUT transfer implies a data transfer from the processor to the device buffer.

Transfer Field	Transfer	Mnemonic
0	No I/O transfer	NIO
1	Data IN from buffer A	DIA
2	Data OUT to buffer A	DOA
3	Data IN from buffer B	DIB
4	Data OUT to buffer B	DOB
5	Data In from buffer C	DIC
6	Data OUT to buffer C	DOC
7	(Reserved for skip tests described later.)	

The format of an I/O instruction as the assembler looks at it is:

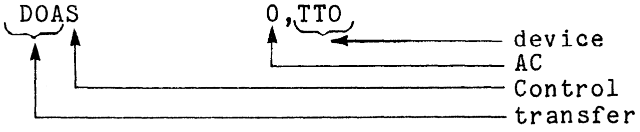
```
        Transfer Control        AC,Device Code
```

To type the character in AC0 on the Teletype:

---

* The complete cross-reference between device codes and their associated mnemonics may be found in Appendix D, In-Out Codes.

4.1
I/O
INSTRUCTIONS
(Continued)



The Teletype keyboard/reader (input) has a device code 10, and the Teletype printer/punch (output) has a device code 11. Both the Teletype input and output have an 8-bit storage capacity in the form of an 8-bit long A buffer. These eight bits correspond to the right-most eight bits of a 16-bit computer word.

Write an instruction that "transfers a unit of data from AC1 to the A buffer of the Teletype output device 11."

```
        DOA            1,TTO
```

Write an instruction that "transfers a unit of data to AC2 from the A buffer of the Teletype input device 10."

```
        DIA            2,TTI
```

It is possible to transfer data to or from any device. It should be noted that these transfers have no effect on the devices themselves; they serve only to pass information. Before a device reacts to transferred data, some control information must be issued by the program. This control information acts to Start and stop (Clear) the particular device involved.

Associated with every device are two one-bit storage elements (flip-flops) called Busy and Done. If both flip-flops are clear (reset), the device is in the idle mode. To place the device in operation, the Busy flip-flop must be set. After the device has processed the unit of data on a DATA OUT instruction, or when a device has information available in a buffer register on a DATA IN instruction, the Busy flip-flop is cleared and the Done flip-flop is set.

4.1
I/O
INSTRUCTIONS
(Continued)

Using the control field in an I/O instruction, the following control functions can be specified by appending the appropriate mnemonic to the instruction.

Mnemonic	Control Function
-	No control.
S	Set the Busy flip-flop and clear the Done flip-flop, thus starting the device.
C	Clear both the Busy and Done flip-flops, thus idling the device.
P	Special pulse output for customer application.

Write an instruction that "transfers a unit of data from AC1 to the A buffer of the Teletype output device 11," then "starts" that device, causing the transferred character to be printed.

```
        DOAS            1,TTO
```

The NIO mnemonic effects no transfer of data, but it does allow for "control only" instructions.

Write an instruction that "idles" the Teletype input (device 10).

```
        NIOC            TTI
```

It is not usually advisable to perform any I/O operations on a device that is busy. Using the special transfer code 7, it is possible to test the status of the Busy and Done flip-flops and to conditionally skip the next instruction as a result of the test.

4.3 I/O INSTRUCTIONS (Continued)

Mnemonic	Transfer Field	Control Code	Operation
SKPBN	7	0	Skip the next instruction if the Busy flip-flop is nonzero.
SKPBZ	7	1	Skip the next instruction if the Busy flip-flop is zero.
SKPDN	7	2	Skip if the Done flip-flop is nonzero.
SKPDZ	7	3	Skip if the Done flip-flop is zero.

Each skip-on-flag function must designate a specific device.

```
SKPDN   TTI     Tests the Done flag of the TTI.

SKPBZ   36      Test the Busy flag of Device 36.
```

Read a character from the TTY; wait until it is in the Done state.

```
NIOS    TTI     ;Start a read cycle.
SKPDN   TTI     ;Skip when TTI done.
                ;(Could be SKPBZ TTI.)
JMP     .-1     ;Continue sensing status.
DIAC    0,TTI   ;Fetch the character and
                ;idle TTI.
```

Write a group of instructions that outputs the character in AC2 to the Teletype printer.

```
SKPBZ   TTO     ;Is the Teletype printer
                ;(code 11) Busy?
JMP     .-1     ;Yes, test it again.
DOAS    2,TTO   ;No, output the character and
                ;start the Teletype printer.
```

4.1
I/O
INSTRUCTIONS
(Continued)

Write a group of instructions that requests a character from the Teletype keyboard or reader, waits until the character is available, then brings it into AC2.

```
NIOS     TTI      ;Start the Teletype input (code
                  ;10), thus requesting a character.
SKPDN    TTI      ;Is the Teletype input Done
                  ;(i.e., is the character in the
                  ;A buffer)?
JMP      .-1      ;No, test it again.
DIA      2,TTI    ;Yes, bring the character (con-
                  ;tents of the A buffer) into
                  ;AC2.
```

NOTE: The Teletype input and output are two unique and separate devices. Each has its own A buffer, Busy and Done flags, and device code. When typing a character on a normal typewriter, the user expects to see the character printed (this is known as "echoing" a character). If a character is typed on a Teletype keyboard, it is only set into the CPU. The character is printed (echoed) only if the program outputs the character. This is called full duplex; indeed, you can be typing one input and completely different results may be printing.

Write a program that inputs and echoes characters from the Teletype keyboard, thus making the Teletype appear as a normal typewriter.

```
NIOS     TTI      ;Start the Teletype input.
SKPDN    TTI      ;Has a character been input
                  ;yet?
JMP      .-1      ;No, keep testing.
DIA      0,TTI    ;Yes, bring character into AC0.
SKPBZ    TTO      ;Is the Teletype printer Busy?
JMP      .-1      ;Yes, keep testing.
DOAS     0,TTO    ;No, output the character in
                  ;AC0.
JMP      .-7      ;Repeat this program.
```

4.1.1 Special Mnemonic Instructions

There does exist a special class of I/O instructions for which the device code (bits 10-15) is 77, or the CPU. Since the CPU is not literally an I/O device with A, B, and C buffers, it is interesting to see what happens when these instructions are executed. Since these instructions are special, the assembler accepts special mnemonics as their equivalent. At this time we will only discuss those special mnemonic instructions that are not associated with interrupts; this is the topic of a later chapter on I/O device handling. Right now, consider the following:

```
READS    AC  = DIA AC,CPU         ;Causes the contents
                                  ;of the console data
                                  ;switches to be
                                  ;read and loaded into
                                  ;the specific AC.

IORST = DICC  0,CPU               ;Clears the control
                                  ;flip-flops (Busy,
                                  ;Done, and Interrupt
                                  ;Disable) in all
                                  ;devices connected
                                  ;to the I/O bus.

HALT  =  DOC  0,CPU               ;Terminates pro-
                                  ;gram execution.
```

4.2 MEMORY REFERENCE INSTRUCTIONS

Now that we have received the character from the device's buffer, we need a place to store it before we can accept too many more characters. For this reason our next category of instruction will be the Memory Reference Instructions (MRI). If we turn back to page 4-2, we can review the operations of this category of instructions.

4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

Since the memory into which we are going to place this data can be as large as 32,768 storage locations (requiring a 15-bit address pointer), and since the IR is only 16 bits long, some scheme had to be devised whereby both the operation and its address could be coded in the 16-bit instruction. Let's look at the IR format of a MRI to see how this is accomplished.

4.2.1 Addressing

The technique is known as <u>indexed addressing</u>.

Indexed addressing is accomplished by coding two numbers into the MRI: an Index (X) Mode and a Displacement (D).

IR	FUNCT/AC	I	X	D
BIT	0 1 2 3 4	5	6 7	8 9 10 11 12 13 14 15

POSITION

The bits contained in bit positions 6 and 7 of the MRI specify the Index (X) Mode, and those contained in bit positions 8 through 15 of the MRI specify the Displacement (D).

X can take on four possible values:

X

$00_2 = 0_8$

$01_2 = 1_8$

$10_2 = 2_8$

$11_2 = 3_8$

Each of these four values for X instructs the CPU to extract a 15-bit number (address) from somewhere in the CPU.

4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

In the CPU there are five accessible temporary storage registers. Four of these storage registers are 16-bit accumulators and the fifth is the 15-bit program counter. (The PC is 15 bits long since any address can be expressed with 15 bits.)

If X is:	The extracted 15-bit number is:
0	00000_8
1	the 15-bit contents of the Program Counter.
2	the right-most 15 bits of AC2.
3	the right-most 15 bits of AC3.

The Displacement (D) is an 8-bit number that can take on the following octal values:

UNSIGNED: 000 through 377
SIGNED: -200 through +177

To use the concept of Index Addressing, the programmer decides which location in memory the MRI is to reference. The address of this location is known as the Effective Address (E). The programmer then forms E by referencing one of the four indexes to which will be added or subtracted the displacement, such that:

$$E = (X) \pm D$$

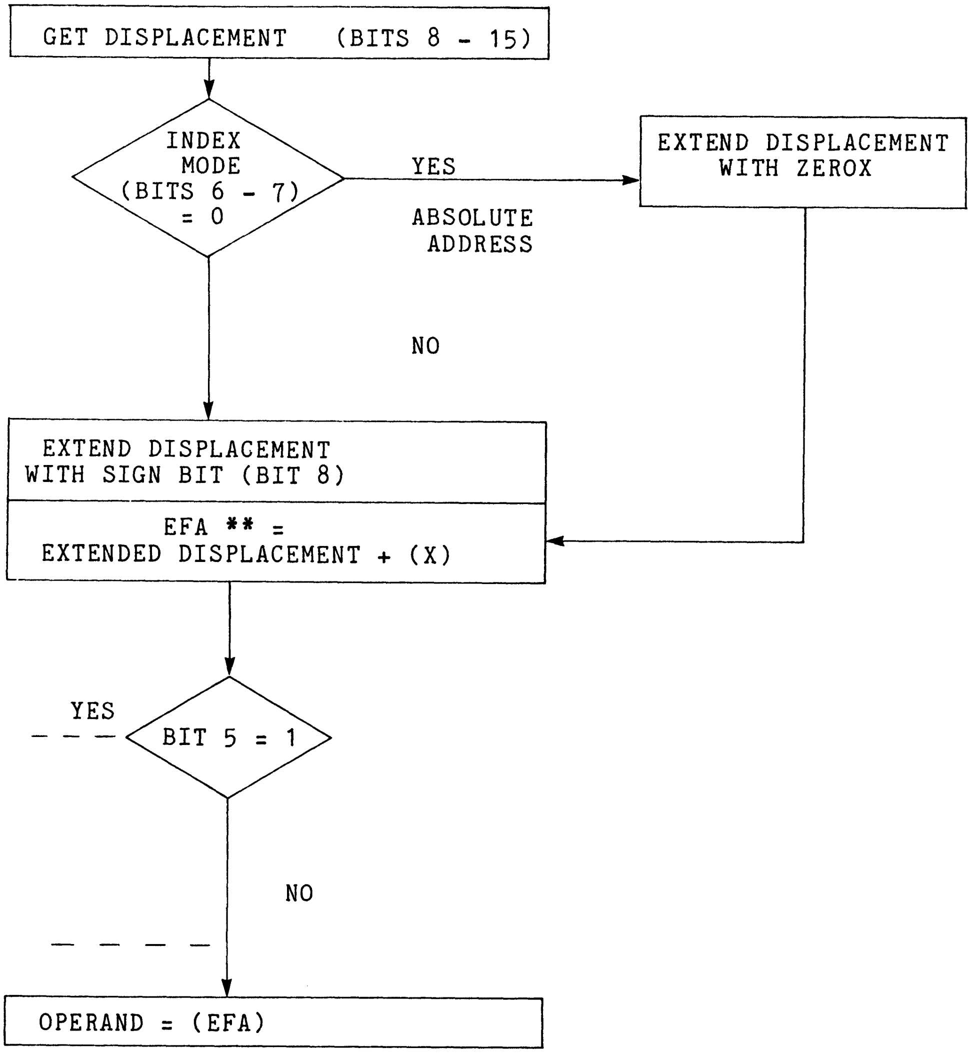
Where, in this case, the notation (X) refers to the extracted 15-bit number. It should be noted that if X=0, then $(X)=00000_8$ and E will actually be the value of D. Also, if X=1, then (X) equals the 15-bit contents of PC. At the time that this MRI is being executed by the computer, the PC contains the 15-bit address of the location in memory where this MRI was fetched. Thus the contents of the PC is sometimes referred to as the "present location in the program," "present location," or "present address."

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

If X Is:	Then (X) Is:	The Possible Values for D Are:	The Possible Effective Addresses (E):
0	00000	000 through 377	00000 through 00377
1	(PC)	-200 through +177	(Present location -200) through (Present location +177)
2	(AC2)	-200 through +177	[(AC2) -200] through [(AC2) +177]
3	(AC3)	-200 through +177	[(AC3) -200] through [(AC3) +177]

Notice that the possible effective addresses, resulting when index mode 0 is used, are always between memory locations 0 and 377. This fixed, addressable area is known as page 0 and the possible effective addresses, resulting when index mode 1, 2, or 3 is used, are dependent upon the contents of the PC, AC2, or AC3 respectively. Index mode 1 addressing is commonly referred to as relative addressing, since the E produced will be distance D relative to the present address (PC). Index modes 2 and 3 addressing are commonly referred to as base register addressing, since the E produced will be a function of the contents of the base register (accumulator) used. In base register addressing, the contents of the base register (accumulator) is commonly referred to as a memory pointer, since it contains the 15-bit address of a location in memory, i.e., points to that location.

The procedure for calculating the address can be seen in the flowchart on the next page.



** EFA MEANS EFFECTIVE ADDRESS

Figure 4.1 Flowchart of Direct Address Calculations

000000	PAGE 0, ABSOLUTE	
000100	RANGE OF ADDRESSES DIRECTLY	X=0
000377	ACCESSIBLE WHEN X=0	
	PAGE 1	
000600	RANGE SERVED BY ADDRESSES	
AC2=001000	RELATIVE TO THE CONTENTS OF	X=2
001177	ACCUMULATOR TWO.	
004600	RANGE SERVED BY ADDRESS	
PC=005000	RELATIVE TO THE LOCATION OF	X=1
005177	THE INSTRUCTION: LDA 0,D,1	
012145	RANGE SERVED BY ADDRESSES	
AC3=012345	RELATIVE TO THE CONTENT OF	X=3
012544	ACCUMULATOR THREE.	
077777	TOP OF 32K OF CORE	

Figure 4.2 Memory Addressing Map

4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

The FUNCT/AC field (bits 0-4) can code one of the following instructions:

```
LDA  AC
STA  AC
  ISZ
  DSZ
  JMP
  JSR
```

The format of MRIs as interpreted by the assembler is:

```
FUNCT <AC,>   D  <,X >
```

where < > means optional entry. In other words, if the FUNCT requires an accumulator (LDA and STA), the <AC,> field must have an entry. Also, if no <,X> is specified, the default value of zero will be assumed.

Now let us examine the individual functions specified above.

4.2.2 LDA

LoaD Accumulator

LDA - "LoaD the contents of a memory location into an Accumulator."

The LDA instruction is used to transfer the contents of a memory location to the CPU (one of the four accumulators). For the CPU to execute an LDA instruction, it must know which one of the four accumulators is to receive the word (0, 1, 2, or 3), and which memory location (E) contains the word to be transferred. The instruction is in the form:

```
LDA  AC,D,X
```

where:
AC is the accumulator number (0, 1, 2, or 3)
D is the displacement (000 through 377) or (-200 through +177)
X is the index (0, 1, 2, or 3)

The combination of D and X form E, the memory address.

This technique takes advantage of the assembler's capability to calculate displacements. More important, however, it relieves you of worrying whether the displacement should be stated in decimal or octal. And secondly, it allows you to insert instructions (as we are about to do) in any area of memory without having to change all MRI displacements that might be affected.

The other change that you should notice is that the accumulators selected for input (DIAS 0,TTI) versus output (DOAS 1,TTO) are different. This was done with forethought, so that the instructions that we will insert will be more meaningful, and less redundant.

Now for the new instructions. After we input the character from the Teletype, we want to store it in a table, thereby freeing up the input accumulator to receive the next character. Also, prior to outputting a character we will get the character from the same table. This may be done by modifying the program as follows:

```
        NIOS    TTI
IN:     SKPDN   TTI
        JMP     .-1
        DIAS    0,TTI   ;Get this char. and
                        ;start the next.
        STA     0,TABLE ;Save this char. in
                        ;TABLE.
         .
         .
         .
         .
         .
         .
         .
         .
        LDA     1,TABLE ;Get the char. for output.
   OUT: SKPBZ   TTO
        JMP     OUT
        DOAS    1,TTO   ;Output the char. for
                        ;printing
        JMP     IN      ;Go get next char.
   TABLE: 0
```

4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

STore Accumulator

STA - "STore the contents of an Accumulator into a memory location."

4.2.3 STA

The STA instruction is used in a manner similar to that of the LDA instruction. The difference is that the STA instruction causes data to be transferred from the CPU (one of the four accumulators) into a memory location, whereas the LDA instruction causes data to be transferred from a memory location to the CPU (one of the four accumulators).

To apply these two instructions in a practical situation, let's look back at the I/O program that we wrote to "echo" characters (see page 4-7). In the program as written, there is no provision to save one character before inputting the next. Let's modify this basic echo routine as follows:

```
        NIOS     TTI      ;Start the Teletype input.
IN:     SKPDN    TTI      ;Has character been input?
        JMP      .-1      ;No, test it again.
        DIAS     0,TTI    ;Yes, get this char. and
          .               ;start the next.
          .
          .
          .
          .
          .
          .
          .
OUT:    SKPBZ    TTO      ;Is the Teletype printer
                          ;busy?
        JMP      OUT      ;Yes, test it again.
        DOAS     1,TTO    ;No, output the character
                          ;and start the printer.
        JMP      IN       ;Go get next character.
```

Before we discuss the instructions that we are going to insert into this program, let's discuss the changes that exist between this version and the one on page 4-7. Here we see that the location of the SKP--instructions has been given a name (IN: and OUT:). This relieves you of worrying about where the instruction "lives" by allowing you to reference the location by a name that you have chosen; a name that has meaning to you. This name is then substituted in the displacement field of the instruction:

```
        JMP  IN
```

4.2
MEMORY REFERENCE INSTRUCTIONS
(Continued)

The program now allows us to use accumulators zero and one in the dot-dot-dot area without losing the character that was input. However, the severe restriction still exists that we can only input one character. That is, we only have one memory location (TABLE) designated to store characters. What would be nice would be the ability to store many characters into sequential locations and perhaps even keep a tally of how many characters were input. Enter, the next two instructions.

4.2.4
ISZ

Increment and Skip on a Zero result.

ISZ

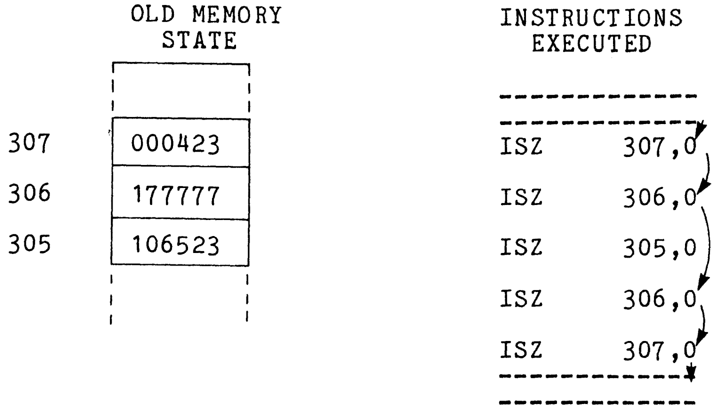
The ISZ instruction causes the contents of a desired memory location to be "Incremented" by one. The only additional information that must be supplied to the CPU is the address of the memory location whose contents are to be altered (incremented). Thus, this instruction takes the form:

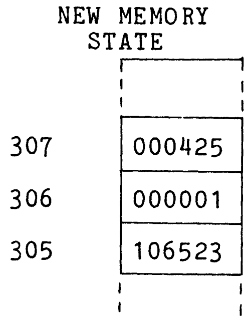
```
ISZ   D,X
```

The combination of D and X form E, the memory address.

The ISZ instruction provides an additional feature. If, after being incremented by one, the new contents of the altered memory location are 000000, then the CPU skips the next instruction in the program -- "Increment and Skip on a Zero result."

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)





Decrement and Skip on Zero result.

4.2.5
DSZ

DSZ

The DSZ -- "Decrement and Skip on a Zero result" -- instruction performs similarly to the ISZ instruction, except that contents of the desired memory location are "Decremented" by one, instead of being "Incremented" by one as in the ISZ instruction.

It is important to realize that both of these instructions are modifying the content of an address, not the actual address. In other words, if we were to insert this instruction:

```
ISZ  TABLE
```

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

into our program, it would *not* serve our purpose. The effect of this instruction would be to add one to the character stored at location TABLE. What we need is to have our address, TABLE, stored as the *content* of another address. The technique for doing so might be:

```
        ATABL: TABLE
```

which may be read as "location ATABL contains the value TABLE," or points to location TABLE. Now let's modify the program again to include the new features.

```
        LDA     0,ATABL    ;Get the address TABLE.
        STA     0,TEMP     ;Save it in a temporary
                           ;location.
        NIOS    TTI        ;Start the Teletype
                           ;input.
  IN:   SKPDN   TTI        ;Has the character been
                           ;input?
        JMP     .-1        ;No, test it again.
        DIAS    0,TTI      ;Yes, get this char-
                           ;acter and start the
                           ;input for the next.
        LDA     2,TEMP     ;Get the address of
                           ;TABLE.
        STA     0,0,2      ;Store the character
                           ;in the table.
         .
         .
         .
         .
         .
         .
         .
        LDA     2,TEMP     ;Get the address of
                           ;TABLE.
        LDA     1,0,2      ;Get the character from
                           ;the table.
   OUT: SKPBZ   TTO        ;Is the Teletype printer
                           ;busy?
        JMP     OUT        ;Yes, test it again.
        DOAS    1,TTO      ;No, output the char.
                           ;and start printer.
```

```
        ISZ     TEMP        ;Advance the table
                            ;pointer.
        ISZ     COUNT       ;Advance the tally
                            ;counter.
  TEMP: 0                   ;
  COUNT:0                   ;Keep a tally of the
                            ;number of entries.
  ATABL:TABLE               ;Pointer to the table.
  TABLE:0                   ;The table starts here.
```

Before we examine the additions that were made to the program, remember under algorithms and flowcharting we introduced the concepts of table, pointers, and counters (refer to pages 3-14 through 3-17). Now we see them implemented in instructions.

As was mentioned in the discussion on pointers, "The programmer will obtain a copy of the pointer, place it in a temporary storage cell, and work on it there to preserve the original pointer." This is the purpose of the first two instructions in our modified program:

```
        LDA   0,ATABL
        STA   0,TEMP
```

Locations ATABL and TEMP appear at the end of the program where ATABL is initialized statically to the value TABLE and TEMP is initialized statically to zero. The program then dynamically reinitializes location TEMP to the value TABLE.

When the program is ready to store or retrieve a character, it is done with the following two-instruction combinations:

```
        LDA  2,TEMP
        STA  0,0,2
             .
             .
             .
        LDA  2,TEMP
        LDA  1,0,2
```

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

The operation of the STA 0,0,2 and LDA 1,0,2 instructions can be reviewed on page 4-19. The LDA 2,TEMP instruction is repeated to allow the dot-dot-dot area of the program to use accumulator two.

After this character has been placed in the table, and sufficiently massaged and output for printing, then the pointer is advanced to the next sequential address in the table:

```
        ISZ  TEMP
```

Also the tally counter is incremented:

```
        ISZ  COUNT
```

In both instances we never expect the "skip on zero result" to take place. We are simply using the increment memory portion of the instruction. Further applications of these instructions will be seen as we continue to modify the program.

As for the remaining MRIs, we have been using one of them ever since we started applying the instructions. Now we will formally define it.

4.2.6
JMP

JuMP

JMP

The JMP -- "JuMP" -- instruction is used specifically to alter the flow of a program. The program that is stored in memory is normally executed sequentially, since the Program Counter (PC) is incremented by 1 following execution of an instruction.

It may be desirable, at some point in a program, to branch to another group of instructions that resides somewhere else in memory. To perform this branching, it is necessary to provide the memory address where the new block of instructions begins. Thus, JMP instructions are of the form:

```
        JMP  D,X
```

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

The combination of D and X form E, the memory address where the new block of instructions begins. The (PC) are replaced by E, causing program execution to proceed sequentially from this new address. This results in the branching of the program to a new block of code.

4.2.7 JSR

JSR

The JSR -- "Jump to SubRoutine" -- provides branching similar to that of the JMP instruction; the main difference between the JMP and JSR instructions is that the JSR instruction not only branches to some other group of instructions, but it also retains the memory address that it jumped from. This feature is extremely useful when writing groups of instructions that perform specific tasks (subroutines).

For example, if the square root operation is used many times in a program, it may be more advantageous to write the square root subroutine as one block of code. Whenever the square root of a number is desired, simply do a JSR to the square root subroutine and have the square root subroutine return to the calling program when finished. For this purpose the CPU puts the return address -- (PC)+1 at the time the JSR instruction occurred -- into AC3.

The formal definition for JMP talks about branching to a new block of code. However, as you can see from our application of the JMP instruction, it works equally well branching back to repeat a block of code. As for the JSR instruction, we are not ready for the square root subroutine; however, we can take our program and make the input and output portions of it into subroutines as follows:

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

```
                LDA     0,ATABL     ;Get the address
                                    ;TABLE.
                STA     0,TEMP      ;Save it in a
                                    ;temporary location.
        GET:    JSR     GCHAR       ;Get a CHARacter.
                LDA     2,TEMP      ;Get the address of
                                    ;TABLE.
                STA     0,0,2       ;Store the char. in
                                    ;the table.
                   .
                   .
                   .
                   .
                LDA     2,TEMP      ;Get the address of
                                    ;TABLE.
                LDA     1,0,2       ;Get the char. from
                                    ;the table.
                JSR     PCHAR       ;Print the CHARacter.
                ISZ     TEMP        ;Advance the table
                                    ;pointer.
                ISZ     COUNT       ;Advance the tally
                                    ;counter.
                JMP     GET         ;Go get next char-
                                    ;acter.
         TEMP: 0
         COUNT:0
         ATABL:TABLE
         TABLE:0

         GCHAR:NIOS     TTI         ;Start the Teletype
                                    ;input.
                SKPDN   TTI         ;Has the char. been
                                    ;input?
               *JMP     .-1         ;No, test it again.
                DIAS    0,TTI       ;Yes, Get char. and
                                    ;start input again.
                JMP     0,3         ;Return to the
                                    ;address saved in
                                    ;AC3 by the JSR
                                    ;instruction.

        PCHAR:SKPBZ     TTO         ;Is the Teletype
                                    ;printer busy?
                JMP     .-1         ;Yes, test it again.
                DOAS    1,TTO       ;No, output char.
                                    ;and start printer.
                JMP     0,3         ;Return to the
                                    ;address saved in
                                    ;AC3 by the JSR
                                    ;instruction.
```

* JMP .-1 means "jump to my current location minus one."

Program A

```
B:
         .
         .
         .
         LDA      0,@BA1   ;Program A's access
                           ;to Argl.
         .
         .
         .
         JSR      C        ;Program A calls C
         .
         .
         .
BA1:     @AA1              ;Indirect pointer to
                           ;Argl.
         .
         .
         .
```

Now that we have looked at all of the memory reference instructions, let's pause for this reminder: The following message is brought to you on behalf of the assembler.

> Hey, remember me! I'm the guy that has to take these mnemonics and convert them into something the CPU can understand. As long as you conform to prescribed rules, I can do my job. I'm the guy that allows you to give a name to a location (GET:) and then to reference that location by name (JMP GET). What you have to remember is that I have to code your reference into binary bits. So keep your references within range, and we will get along just great.

I think what he is trying to tell us is that it is time to go back and take another look at bits 5 through 15 of the instruction register for a MRI. Since we haven't given any numeric addresses for the locations of our instructions, the references could be to page zero (location 0-377_8) or page one (locations 400_8 to top of available memory). Figure 4.3 shows how the assembler will code bits 6-15 of the instruction.

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

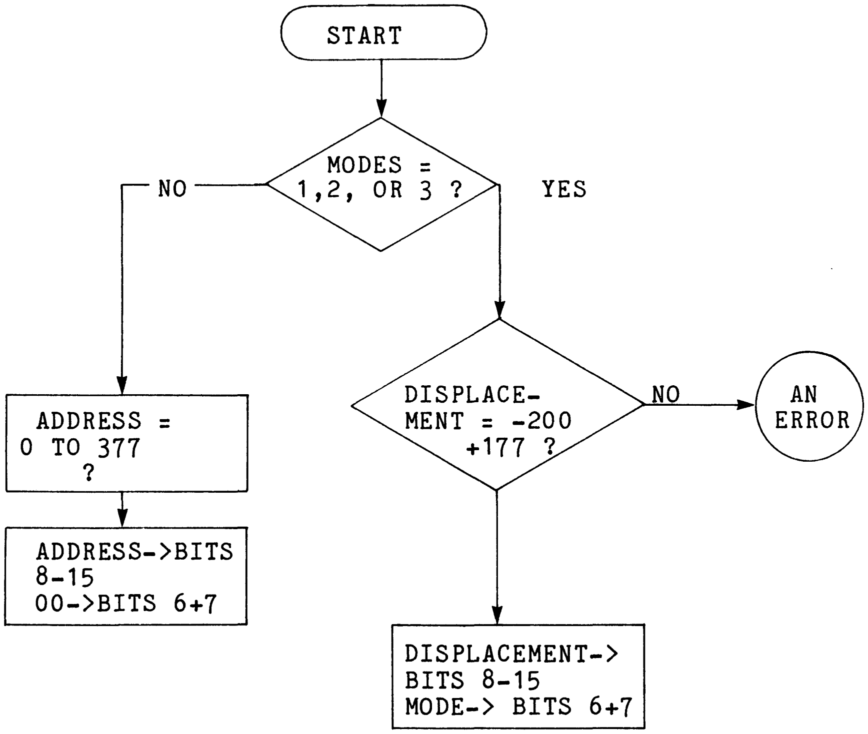


Figure 4.3 Formation of Effective Address for MR Instruction

4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

Referring once again to the Memory Addressing Map on page 4-13, it appears that of a total of 32K of possible addresses, we only have access to a maximum of 1K at any given point in time. In other words, with specific values already in the PC, AC2, and AC3, bits 8 through 15 can only displace these values by a fixed range. One out of 32; wouldn't it be nice if we could reach the other 31K without having to alter the PC, AC2, or AC3? Behold, IR bit 5! Up to this point, IR bit 5 has been a zero, which defines IR bits 6 through 15 as the address of an operand, or the final address. Now, if we could just get bit 5 set to a 1, the CPU would then interpret IR bits 6 through 15 as the address of an address, or an indirect address. We have already seen this concept in application when we, in our program, statically set the content of address ATABL equal to TABLE:

```
ATABL: TABLE,
```

and again when we dynamically set the content of address TEMP equal to TABLE:

```
STA     0,TEMP
```

Without indirect addressing, we then picked up our characters in the following sequence:

```
LDA     2,TEMP
LDA     0,0,2
```

4.2.8 INDIRECT ADDRESSING

Wouldn't it be nice if we could load accumulator zero by simply going indirect through location TEMP to arrive at the table. Hey assembler, what's the procedure?

4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

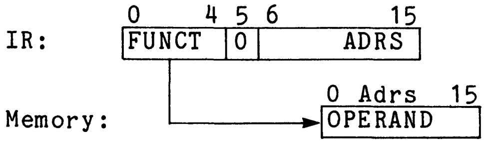
If you use the "at" symbol (@) anywhere in a MRI instruction, I will interpret this to mean that the address is indirect and will therefore set bit 5 to a 1.

Example:

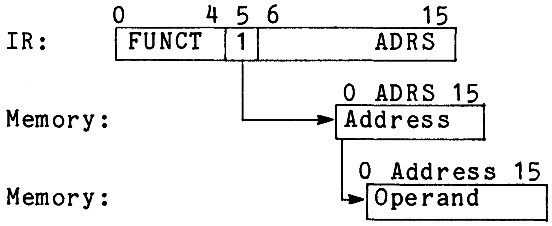
```
        LDA     0,@TEMP
or      LDA     0,TEMP@
or      LDA     @,0,TEMP
or      @LDA    0,TEMP
or      LDA@    0,TEMP
```

All of the above will be treated identically by the assembler. What indirect addressing buys us is 16 bits worth of address. Let me explain. While we were confined to the instruction register, a portion of the 16 bits had to specify FUNCT, a portion for AC, a portion for Index mode, and finally eight bits for Displacement. Once we leave the confines of the IR, and begin obtaining our addresses from memory locations, we have the full 16-bit content of the memory location with which to specify a new address. This difference can be seen in the diagram below.

<u>Direct Addressing</u> - IR bit 5 = 0



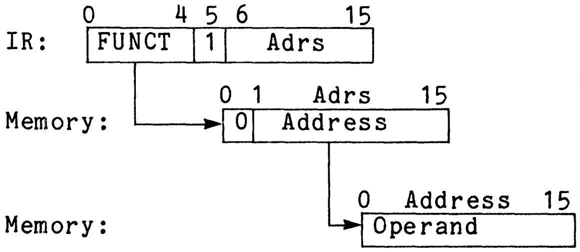
<u>Indirect Addressing</u> (@) - IR bit 5 = 1

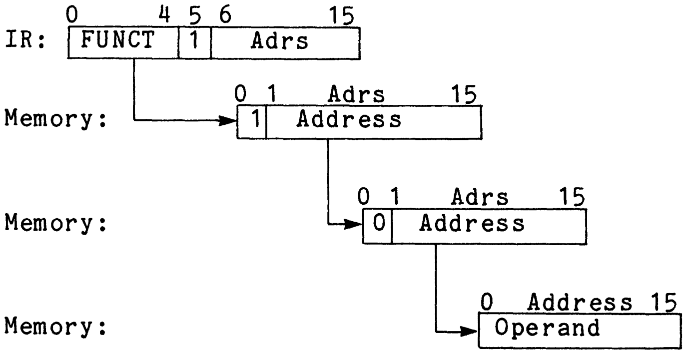


4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

The question that you are undoubtedly waiting to ask is, "I thought we only needed 15 bits to access any address in the 32K range?"
The answer: You're absolutely correct! Therefore, every time we extract a 15-bit address from memory, we have a whole bit left over. What do you think we should use it for? Sorry, the decision has been made for us. Just as IR bit 5 begins the indirect addressing chain, so memory bit 0 can be used to perpetuate the chain.

Example:





4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

The chain, then, can be as long or as short as you desire simply by setting bit 0 in those memory locations that the chain references.

The way that the CPU calculates the address is shown in the flowchart on the following page.

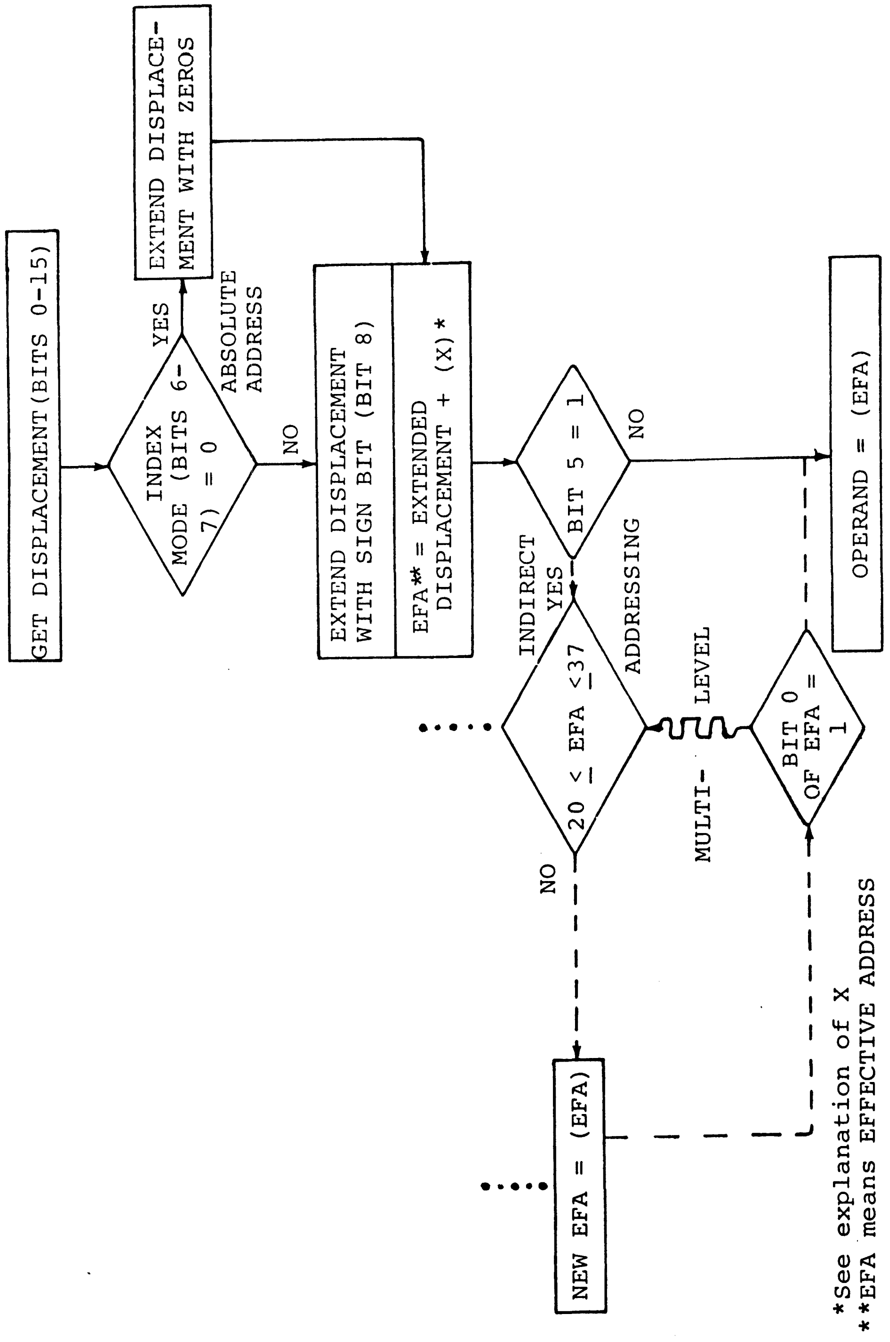
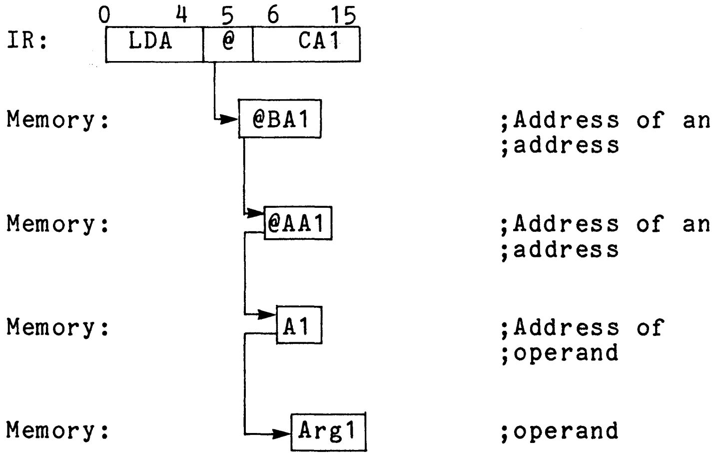


Figure 4.4 Flow Chart of Indirect Address Calculations

Program B

```
B:        .
          .
          .
          LDA     0,@CA1   ;Program B's access
          .                ;to Argl.
          .
          .
          JSR     ECT      ;Call to next level
          .
          .
          .
  CA1: @BA1                ;Indirect pointer
          .                ;to Argl
          .
          .
```

Now consider that we are executing the LDA instruction in Program B:



What is not shown in the block approach above is how the indirect pointer at each level might be done dynamically by each program before calling the next level.

Let's return now to our program as we left it. By using indirect addressing, it now appears as follows:

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

```
        LDA    0,ATABL
        STA    0,TEMP
GET:    JSR    GCHAR
        STA    0,@TEMP
        .
        .
        .
        LDA    1,@TEMP ;Get the character
                       ;from the table.
        JSR    PCHAR   ;Print the CHARacter.
        ISZ    TEMP    ;Advance the table
                       ;pointer.
        etc.
        etc.
```

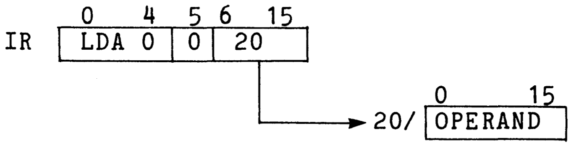
4.2.9 AUTO INDEXING

Now that you are feeling comfortable with indirect addressing, it's time for another one of those "wouldn't it be nice" curves. Wouldn't it be nice if the same instruction that gets the operand from the table would also advance the table pointer, thereby eliminating the need for a separate instruction to do the job: ISZ TEMP. Dare we call on the assembler again? No, this is a job for CPU. It is referred to as auto-indexing, and it works as follows:

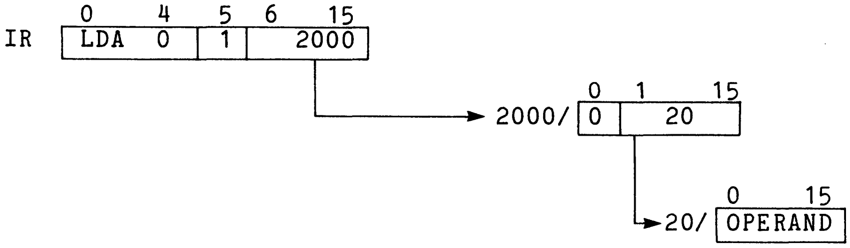
> If at any level in the effective address calculation locations 20-37$_8$ are referenced indirectly (i.e., their content is an address), the content will be automatically incremented or decremented by one before use. The new value is both written back into the auto-indexed location and used as the next level in the indirect addressing chain. Addresses taken from locations 20-27$_8$ are incremented before use, those from 30-37$_8$ are decremented before use. To illustrate, consider the following.

When referenced directly, locations 20-37 are no different from any other location.

4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)

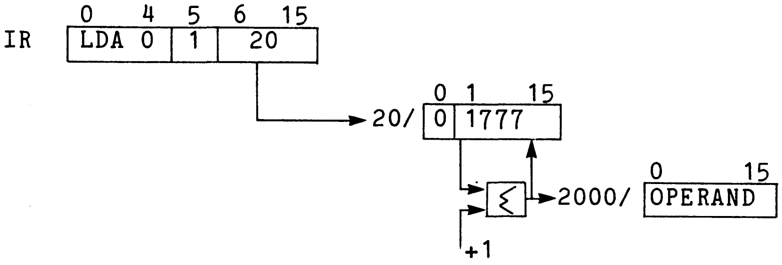


Even after one level of indirect addressing, when locations 20-37 are referenced directly, they are no different from any other addresses.



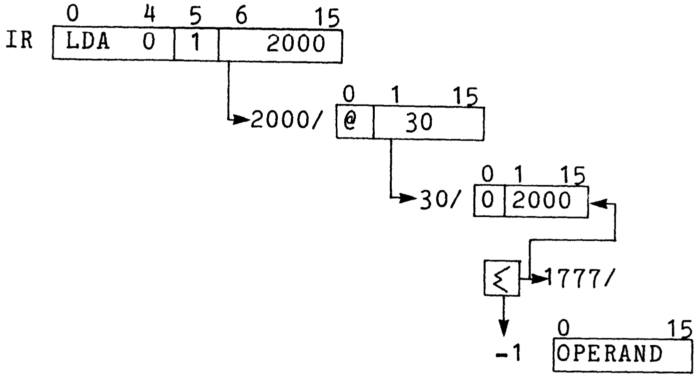
The auto-indexed location may be referenced indirectly at the instruction register level:

First Level Auto-Indexing

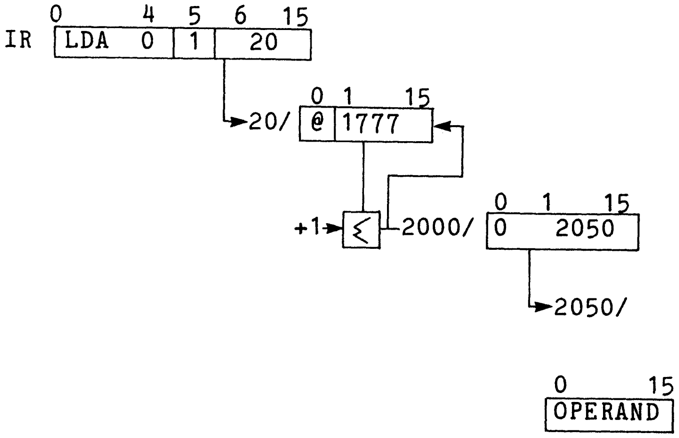


or at any level thereafter. The auto-indexed location may contain a final address:

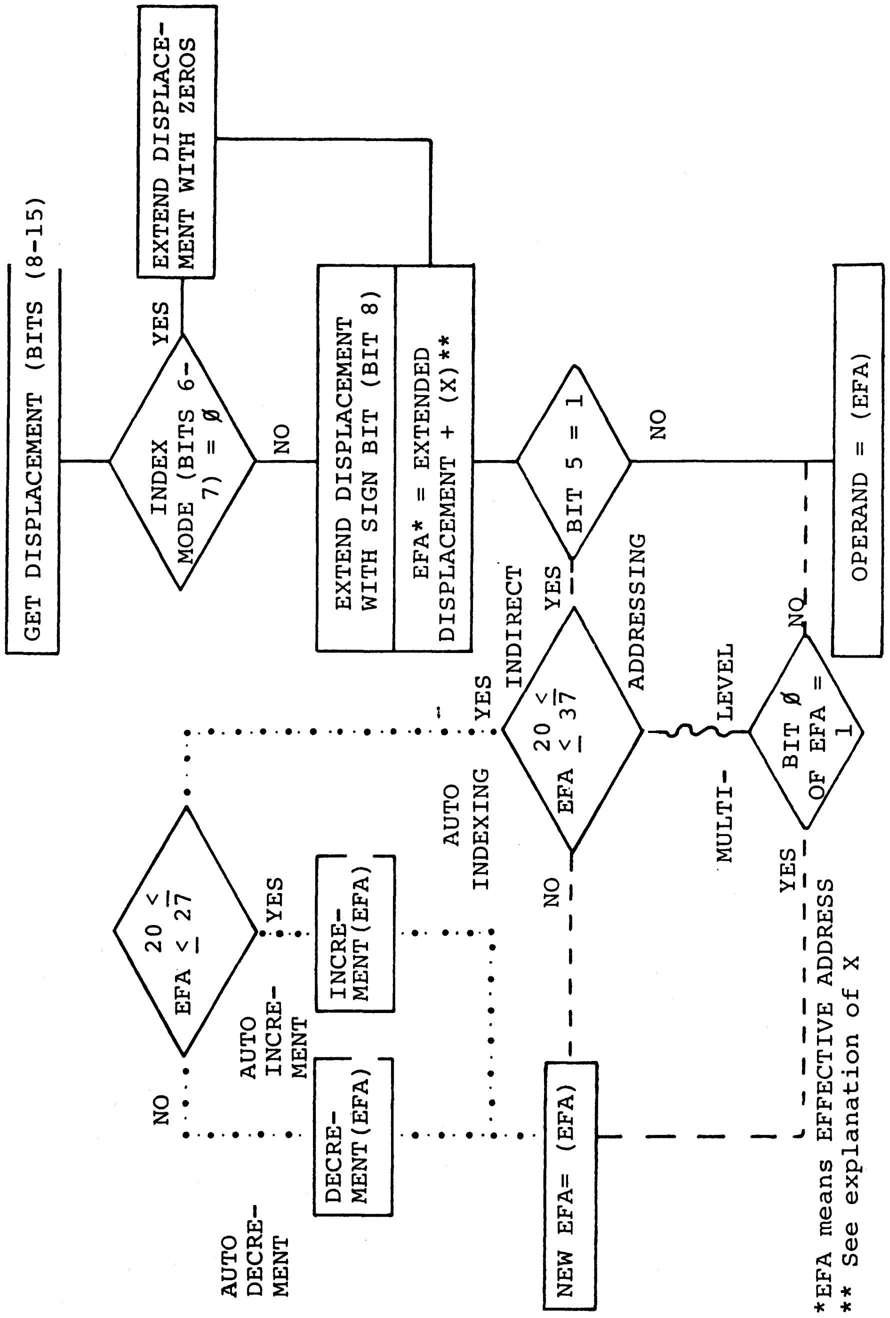
4.2
MEMORY
REFERENCE
INSTRUCTIONS
(Continued)



or another indirect address; they are still auto-indexed locations as long as the chain is not broken.



The following flowchart demonstrates how the computer calculates the address.



*EFA means EFFECTIVE ADDRESS

** See explanation of X

Figure 4.5 Complete Flowchart of Address Calculations

Modifying our program to implement the new technique, we have the following:

```
        LDA     0,ATABL         ;Get the address TABLE.
        STA     0,20            ;Save as auto-index
                                ;pointer.
        DSZ     20              ;Back off for "incre-
                                ;ment before use."
        STA     0,21            ;Input versus Output
                                ;needs
        DSZ     21              ;separate pointers.
GET:    JSR     GCHAR           ;Get the CHARacter.
        STA     0,@20           ;Store character and
        .                       ;advance pointer.
        .
        .
        LDA     1,@21           ;Get the character and
                                ;advance pointer.
        JSR     PCHAR           ;Print the CHARacter.
        ISZ     COUNT           ;advance the tally
                                ;counter.
        etc.
```

Notice that this technique requires additional instructions to back off on the auto-indexed locations before using them.

```
DSZ     20
.
.
.
DSZ     21
```

This can be overcome by using auto-indexed addressing for all references to TABLE and then simply initialize location ATABL to one less than the start address of TABLE. The program would then look like the following:

```
        LDA     0,ATABL ;Get the address of
                        ;TABLE.
        STA     0,20    ;Save as auto-index
                        ;pointer
        STA     0,21    ;for both input and
                        ;output.
GET:    JSR     GCHAR   ;Get the CHARacter
        STA     0,@20   ;advance pointer and
        .               ;store character.
        .
        .
        LDA     1,@21   ;Advance pointer
                        ;and get character.
        JSR     PCHAR   ;Print the CHARacter
        ISZ     COUNT   ;advance the tally
                        ;counter.
        JMP     GET     ;Go get next char-
                        ;acter.
COUNT:  0
ATABL:  TABLE-1         ;Back off for
                        ;"increment before
                        ;use."
TABLE:  0
```

To further enhance your understanding of indirect addressing, try the following program.

1. Fill in the comment column based on your understanding of the instructions.

```
                        Comments

START:  LDA     0,CON   ;
        STA     0,CNT   ;
        LDA     0,VAR   ;
LOOP:   ISZ     TAD     ;
        STA     0,@TAD  ;
        DSZ     CNT     ;
        JMP     LOOP    ;
        HALT            ;
TAD:    @770
CON:    5
CNT     0
VAR:    771
```

(Continued)

4.2 MEMORY REFERENCE INSTRUCTIONS (Continued)

2. Given:

Address/Content	Address/Content
771/7700	774/7730
772/7710	775/TAD
773/7720	776/0

3. Fill in the missing content.

Address/Content	Accumulator = Content
7700/	ACO =
7710/	
7720/	
7730/	

This concludes our immediate coverage of memory reference instructions. However, we will be using MRIs as we continue our coverage of the instruction set.

4.3 ARITHMETIC AND LOGIC INSTRUCTIONS

Thus far in the development of our program, we have been able to input and output characters to a device. Also, we have stored and retrieved these characters from memory by building a table, using a pointer, and keeping a tally. The third category of instruction, the Arithmetic and Logic Class (ALC), will be used to overcome some of the severe restrictions in our program as developed thus far. For instance, wouldn't it be nice if we could pack two 8-bit (ASCII) characters into those 16-bit memory locations instead of wasting 50% of memory? Wouldn't it be nice if we could sense for the presence of a particular character to signify end-of-input? Wouldn't it be nice if we could selectively store or discard characters? For these and other niceties, stay tuned as we present the Arithmetic and Logic Class of instructions.

4.3 ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)

The basic format of ALC instructions as interpreted by the assembler is:

```
FUNCT  ACS,ACD
```

where: ACS means Source Accumulator (0-3)
ACD means Destination Accumulator (0-3)

This information is contained in bits 0 through 7 of the instruction register in the following manner:

1	ACS	ACD	FUNCT	
0	1 2	3 4	5 6 7	8 15

A 1 in bit 0 indicates an ALC.

The mnemonics that the assembler will accept and their associated descriptions are given on the following page.

4.3 ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)

```
Mnemonic                Description
COM     ACS,ACD           ;compute the 1's comple-
                          ;ment of the number in ACS,
                          ;and put the result into
                          ;ACD.
NEG     ACS,ACD           ;compute the 2's comple-
                          ;ment (negative) of the
                          ;number in ACS, and put
                          ;the result into ACD.
INC     ACS,ACD           ;add one (increment) to the
                          ;number in ACS and put the
                          ;result into ACD.
MOV     ACS,ACD           ;copy (move) the number in
                          ;ACS into ACD.
ADD     ACS,ACD           ;add the number in ACS
                          ;to the number in ACD and
                          ;put the answer into ACD.
SUB     ACS,ACD           ;subtract the number in ACS
                          ;from the number in ACD
                          ;and put the answer into
                          ;ACD.  Subtract is per-
                          ;formed by taking the 1's
                          ;complement of the number
                          ;in ACS, adding this to the
                          ;number in ACD, then adding
                          ;1 to the result  (2's
                          ;complement subtraction).
ADC     ACS,ACD           ;add the 1's complement
                          ;of the number in ACS to
                          ;the number in ACD and put
                          ;the answer into ACD.
AND     ACS,ACD           ;perform a logical AND
                          ;operation between the
                          ;number in ACS and the
                          ;number in ACD and put the
                          ;result into ACD.
```

Quite often it is convenient to start with a value of zero in an accumulator. Since we don't have a CLEAR instruction as such, this may be accomplished (without the use of a constant from memory) by subtracting the accumulator from itself. For instance, to clear AC2, use the following:

```
        SUB  2,2
```

4.3
ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)

Another convenient function would be that of comparing two quantities. However, the comparison would be meaningless unless we had a way of testing the outcome. For this purpose the assembler will accept a skip specifier in the form of a three-character mnemonic following ACD. The table below gives the acceptable mnemonics and their meanings.

4.3.1
SKIP FUNCTIONS

Mnemonic	Meaning
(None)	Default condition; no test is made. The next location in the program sequence will be executed.
SKP	(Unconditional SKIP) The next location in the program sequence is unconditionally skipped.
SZR	(Skip on Zero Result) If the 16-bit result from the operation is zero, the next location in the program sequence is skipped.
SNR	(Skip on Nonzero Result) If the 16-bit result from the operation is nonzero, the next location in the program sequence is skipped.
SZC	(Skip on Zero Carry) If the carry bit resulting from the operation is zero, the next location in the program sequence is skipped.
SNC	(Skip on Nonzero Carry) If the carry bit resulting from the operation is nonzero, the next location in the program sequence is skipped.

4.3 ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)

Mnemonic	Meaning
SEZ	(Skip if Either or both are Zero) If either or both (Carry and Result) are zero, the next location in the program sequence is skipped.
SBN	(Skip if Both are Nonzero) If both (Carry and Result) are nonzero, the next location in the program sequence is skipped.

The assembler codes this information into bits 13 through 15 of the instruction as follows:

0	1 2	3 4	5 7	8 12	13 15
1	ACS	ACD	FUNCT		SKIP

Now, to effect a comparison for equality we might use the following program sequence.

Test to see if the input character is a carriage return.

4.3
ARITHMETIC
AND LOGIC
INSTRUCTIONS
(Continued)

```
               .
               .
               .
               DIA      0,TTI     ;Get the character.
               LDA      1,CR      ;Get the ASCII for
                                  ;carriage return.
               SUB      0,1,SNR   ;Test for equality.
               JMP      EQUAL     ;Char = CR.
               .                  ;Execute this pro-
               .                  ;gram sequence only
               .                  ;if Char = CR.
      EQUAL:   -                  ;Execute this pro-
               -                  ;gram sequence
               -                  ;only if Char = CR.
               -
               -
               -
      CR:      215                ;ASCII for carriage
                                  ;return with even
                                  ;parity.*
```

Additional comparisons will be made after we have the entire 16 bits of instruction with which to work. During the execution of the SUB 0,1,SNR instruction, the original content of the source accumulator is unaltered; however, the original content of the destination accumulator is destroyed. This means we have to reload the CR character every time we want to test a new input character. Wouldn't it be nice if we could load it once and perform all tests without having to reload the test character again? Hey assembler, what have you got in your back of tricks?

4.3.2
NO-LOAD FUNCTION

Funny you should ask. It just so happens that if the number sign (#) appears anywhere in an ALC instruction, I will interpret this to mean "do not deliver the result to the destination accumulator." I will therefore use this information to set bit 12 of the instruction.

---

* For a discussion on parity, see page 4-51.

4.3 ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)

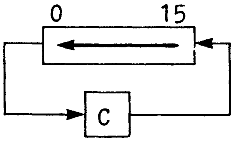
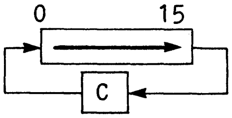
Isn't he wonderful folks? Let's hear it for the assembler. So now the instruction register looks like the following:

0	1 2	3 4	5 7	8 11	12	13 15
1	ACS	ACD	FUNCT		no load	SKIP

4.3.3. SHIFT FUNCTION

Another very common test that is performed is that of testing for positive versus negative numbers; i.e., testing the sign of a number. In signed number representation, the most significant bit (bit 0) is the sign bit. Since we don't have a skip specifier to test bit 0, we will just have to position bit 0 where it can be tested. How about if we move it into the Carry bit? The table below shows how this is done.

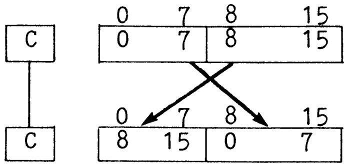
The Shift Field (Bits 8 and 9)

Mnemonic	Effect
(none)	Default value. No effect.
L	All bits are shifted one position to the left: 
R	All bits are shifted one position to the right: 

4.3
ARITHMETIC
AND LOGIC
INSTRUCTIONS
(Continued)

The Shift Field (Bits 8 and 9)

Mnemonic	Effect
S	A byte swap occurs:



The instruction register now looks like the following:

0	1 2	3 4	5 7	8 9	11	12	13 15
1	ACS	ACD	FUNCT	SHIFT		no lead	SKIP

Now, with our new found capabilities we can do the following:

```
        MOVL#    0,0,SNC ;Test the sign.
        JMP      POS     ;If positive (=0) JMP
        next instruction;Do this if negative.
                         ;(≠0)
```

Notice that by combining the shift operation with NO LOAD feature,we can perform the test without destroying the original content of the accumulator. The same technique may also be used to test odd versus even numbers by shifting in the other direction.

```
        MOVR#    0,0,SZC ;Test for odd versus
                         ;even.
        JMP      ODD     ;If odd,JMP.
        next instruction;Do this if even.
```

4.3
ARITHMETIC
AND LOGIC
INSTRUCTIONS
(Continued)

Let us consider now the third possibility: a swap. Remember the Teletype, that 8-bit ASCII device? Since it only deals in eight bit quantities, it will always send and receive information in bits 8 through 15. Therefore, if we have two characters in an accumulator, we would output them in the following manner.

```
LDA      0,@21     ;Advance the pointer and
                   ;get two char.
JSR      PCHAR     ;Print CHAR in low-byte
                   ;position.
MOVS     0,0       ;Reposition the bytes.
JSR      PCHAR     ;Print the second CHAR.
```

Notice in the example above, the No-Load switch (#) is off. Therefore, the result will be delivered to the destination accumulator. The shift left and shift right operations could also be used to multiply a number by two or divide a number by two. There is only one drawback. If the carry bit that gets shifted into the number is a 1, it destroys the integrity of the number. What we need is more control over the carry bit. Aside from the fact that shifting left or right alters carry, carry is also affected by overflow resulting from certain arithmetic operations.

Overflow will result from any of the following:

INSTRUCTION	CONDITION CAUSING OVERFLOW
ADD ACS,ACD	where (ACS) + (ACD) > 2^{16} -1
INC ACS,ACD	where (ACS) = 2^{16} -1
NEG ACS,ACD	where (ACS) = 0
SUB ACS,ACD	where (ACS) $\leq$ (ACD)
ADC ACS,ACD	where (ACS) < (ACD)

4.3
ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)

The effect that overflow has on carry is to complement the value of the carry bit. For this reason, the assembler provides a means for you to force the carry bit used in the operation to a known state before the operation takes place (known as the base value of carry). This base value is established by appending a fourth letter onto the instruction mnemonic. The acceptable letters and their associated base values are given in the table below.

4.3.4
CARRY FUNCTION

If the Letter Is:	Then the Base Value Will Be:
(absent)	(Default value) The present state (1 or 0) of carry at the time the instruction is encountered.
C	The Complement of the present state of carry at the time the instruction is encountered.
Z	Forced as a Zero.
O	Forced as a One.

You must bear in mind that what you are doing is establishing a base value for carry that will be complemented if the arithmetic/logic result produces overflow.

For example:

```
ADD      1,2       ;The base value of the Carry
                   ;bit is whatever the value of
                   ;the Carry bit happens to be
                   ;at the time this instruction
                   ;is encountered.  An overflow
                   ;causes this base value to
                   ;be complemented.

ADDC     1,2       ;The base value of the Carry
                   ;bit is the complement of what-
                   ;ever the value of the Carry
                   ;bit happens to be at the time
                   ;this instruction is encountered.
                   ;An overflow causes this base
                   ;value to be complemented.

ADDZ     1,2       ;The base value of the Carry
                   ;bit is forced to a zero.  An
                   ;overflow causes the Carry bit
                   ;to become 1.

ADDO     1,2       ;The base value of the Carry
                   ;bit is forced to a 1.  An
                   ;overflow causes the Carry bit
                   ;to become zero.
```

Now let's go back to our technique for clearing an accumulator. Realizing that subtraction by two's complement addition will produce overflow and thereby complement the base value established for carry, we can use this to effect the following:

4.3 ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)

```
SUB     1,1       ;Clear AC1 and complement the
                  ;present state of carry.
SUBC    2,2       ;Clear AC2 and preserve the
                  ;present state of carry.
SUBO    3,3       ;Clear AC3 and clear carry.
SUBZ    0,0       ;Clear AC0 and set carry.
```

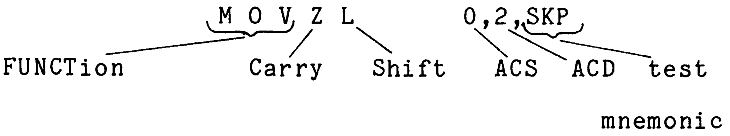
Now our instruction is complete,

0	1 2	3 4	5 7	8 9	10 11	12	13 15
1	ACS	ACD	FUNCT	SHIFT	CARRY	no load	SKIP

and our mnemonic representation looks like the following:

```
FUNCT<C><S><#>     ACS,ACD<,SKIP>
```

where < > denotes optional entries, and # is a floating symbol that may appear anywhere in the instruction. Also notice that contrary to its position in the instruction register, if both Shift and a Carry specifier are given, the carry must precede the shift.

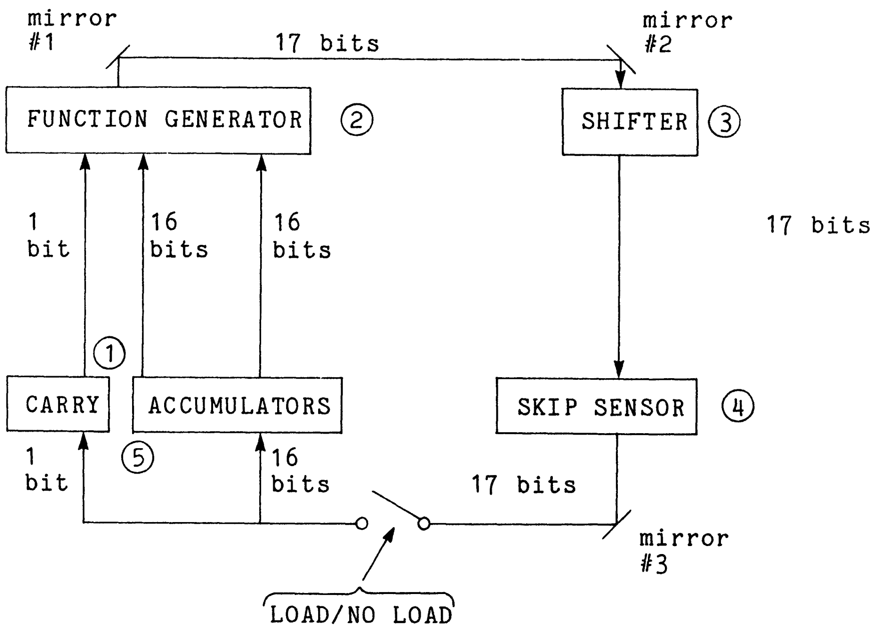


```
SUBO    1,1      ;Clear AC1 and clear Carry.
ADDCS   0,1,SZC  ;Since there can be no
                 ;overflow, and since Swap
                 ;does not affect Carry,
                 ;Carry will get set.
```

This technique will be used in a later discussion of a concept called packing.

4.3
ARITHMETIC
AND LOGIC
INSTRUCTIONS
(Continued)

Wow! With all of this going on at once, how does it ever produce a result? Would you believe it's all done with mirrors? The exact sequence of events can be seen by following the data in a clockwise direction through the diagram below.



Notice that all manipulations of carry (base value versus overflow) are performed within the function generator, and then the 17-bit result is passed on to the shifter.

4.3.5 BYTE MANIPULATION

Now with all the bells and whistles accounted for, let's go back and make some enhancements on our program. First of all, after we get the character, and before we store it in memory, let's pack two 8-bit characters into one 16-bit accumulator. To do this we will need an extra accumulator in which to do the packing, and, secondly, some technique for detecting the fact that two characters have been input. The following program will accomplish the job.

```
TEST:   177777              ;Minus 1 for ISZ
                            ;instruction.
        SUBO     1,1        ;Clear AC1 and clear
                            ;Carry.
        NIOS     TTI        ;Start the Teletype
                            ;reader.
        SKPDN    TTI        ;Is the character
                            ;ready?
        JMP      .-1        ;No, test it again.
        DIAS     0,TTI      ;Yes, get the char-
                            ;acter.
        ADDS     0,1        ;ADD char to AC1
                            ;and swap bits.
        ISZ      TEST       ;Have two characters
                            ;been input?
        MOVS     1,1,SKP    ;Yes, reposition
                            ;bytes.
                            ;AC1 = |1st | 2nd|
        JMP      .-6        ;No, go get second
                            ;character.
        STA      1,@20      ;Store two char-
                            ;acters in the
        .                   ;table.
        .
        .
```

Now let's analyze the program. About the only point of merit is that the last four instructions show the application of the unconditional skip (SKP) feature of an ALC instruction. Aside from that, the program only works for the first two characters. After that, location TEST will not produce a zero result (ISZ TEST) for another 2^{16} characters. It would require no less than two additional instructions to restore location TEST to its initial value of minus one. Secondly, we are using an entire 16 bits to detect whether or not the second character has been input. The same thing could be accomplished with one bit and at the same time greatly simplify the program. The technique is to start with the carry bit in a known state (which we have already done) and then test the state of carry to determine if both characters have been input. Let's use the input subroutine (GCHAR) that we wrote back on page 4-23.

```
SUBO    1,1      ;Clear AC1 and clear
                 ;Carry.
JSR     GCHAR    ;Get the CHARacter.
ADDCS   0,1,SZC  ;Position char. Is it
                 ;second char?
JMP     .-2      ;No, go get second
                 ;character.
MOVS    1,1      ;Yes, reposition bytes.
                 ;AC1 = |1st | 2nd|
STA     1,@20    ;Store two characters
                 ;in the table.
```

The purpose of repositioning the bytes |1st | 2nd| before storing them is to be compatible with some existing software which, when outputting from a table, will always output the high byte first.

4.3.6 Parity and Masking

Before we further modify our program, let's discuss the Teletype parity bit and a technique known as masking. In the field of data transmission (especially serial data transmission), it is imperative that the integrity of the character be checked to ensure that nothing was lost during transmission; so that the character received is indeed the character that was transmitted. For this purpose, Teletype appends onto its 7-bit code an eighth bit called the parity bit. The parity used can be either even parity or odd parity. For even parity, the parity bit will be set when the 7-bit code contains an odd-number of one bits, or clear when the 7-bit code already contains an even number of one bits. This technique allows the receiving device to simply check the number of one bits against the type of parity being used. For even parity, all characters will have an even number of one bits; for odd parity, an odd number of one bits. Since you don't want to be bothered with which characters will have the parity bit set, and which will not, or whether it's even parity or odd, we will use a technique called masking to strip off the parity bit leaving only the 7-bit code. This technique of masking is done with the logical AND instruction and may be used to isolate any number of sequential or randomly located bits within a word. Remember, the logical AND function will save anything that is ANDed with a binary 1, and discard anything that is ANDed with a binary 0.

4.3
ARITHMETIC
AND LOGIC
INSTRUCTIONS
(Continued)

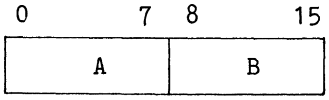
Example: Assume AC2 contains the information we are interested in.

```
MASK1:  177
MASK2:  3400
MASK3:  160000
        LDA     1,MASK1 ;Get the mask.
        AND     2,1     ;Isolate the low-
                        ;order seven bits.
        LDA     0,MASK2 ;Get the mask.
        AND     2,0     ;Isolate bits 5-7.
        LDA     3,MASK3 ;Get the mask.
        AND     2,3     ;Isolate bits 0-2.
```

In many applications, 8-bit words -- bytes -- are sufficient data word blocks, such as for storage of 8-bit Teletype character strings.

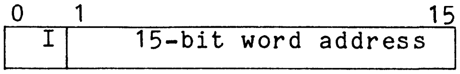
Because the address of any 16-bit word requires only 15 bits, the remaining bit can be used to specify the left or right byte of the contents of a memory location.

A memory capacity of 32K words contains 64K bytes, where each memory cell contains two bytes.



Remember the technique we used to perpetuate indirect addressing:

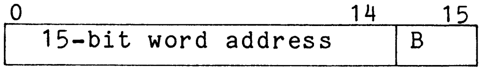
4.3
ARITHMETIC AND LOGIC INSTRUCTIONS (Continued)



Where: I = 0, bits 1-15 = address of operand.
I = 1, bits 1-15 = address of an address.

4.3.7
BYTE POINTERS

Similarly, byte addresses or byte pointers are of the form:



where: B=0 specifies the left byte (A)
B=1 specifies the right byte (B)

Thus, incrementing the byte pointer addresses first the left byte and then the right byte of sequential memory locations.

Right shifting the byte pointer leaves a memory address. Following this with program skipping based on the Carry flag designates the specific byte.

One technique for accomplishing this in our program is as follows:

```
        LDA     2,ATABL ;Get the address
                        ;of TABLE.
        INCZL   2,2     ;Generate a byte
        .               ;pointer.
        .
        .
ATABL:  TABLE-1         ;Minus 1 for auto-
                        ;indexing before
                        ;use.
TABLE:  0               ;The table starts
                        ;here.
```

4.3
ARITHMETIC
AND LOGIC
INSTRUCTIONS
(Continued)

The purpose of the INC is to compensate for the fact that ATABL is initialized to TABLE-1 for auto-indexing purposes. The purpose of the L (shift Left) is to multiply by two. The purpose of the Z (base value of Carry) is to ensure that a zero gets shifted into bit 15. As a result of all this,

In our program we keep track of how many characters were in the table by using a tally counter. Another method that is commonly used is to always end a table with a null (all bits zero) character. With this method, the output routine simply checks each character until it finds the null, and then terminates. After the table has been built, the output routine might look something like the following.

Main Program

```
        .
        .
        .
        LDA     1,ATABL ;Get the address of
                        ;TABLE.
        INCZL   2,2     ;Generate a byte
                        ;pointer.
        JSR     PRINT   ;Go print the table.
        .
        .
        .
ATABL:  TABLE-1
TABLE   0
        .
        .
        .
```

Subroutine Print

```
PRINT    STA     3,SAC3   ;Save the return
                          ;address.
         MOVZR   2,3      ;Address to bits
                          ;1-15, byte pointer
                          ;to Carry.
         LDA     0,0,3    ;Get first two
                          ;characters.
         MOV     0,0,SNC  ;Which byte?
         MOVS    0,0      ;Carry = 0, move
                          ;high to low.
         LDA     3,MASK   ;Get low byte mask.
         AND     3,0      ;Mask out bits 0-8.
         JSR     PCHAR    ;Go print the char-
                          ;acter.
         MOV     0,0,SNR  ;Was character a
                          ;null?
         JMP     @SAC3    ;Yes, return to
                          ;main program.
         INC     2,2      ;No, advance char-
                          ;acter pointer.
         JMP     PRINT+1  ;Go get more char-
                          ;acters.
SAC3:    0                ;Save return address
                          ;here.
MASK:    177              ;Mask to save bits
                          ;9-15.
```

The PCHAR routine is the same one that we used back on page 4-23. The purpose of the AND 3,0 instruction is to ensure that there are zeroes in the high byte when the MOV 0,0,SNR instruction checks for the null byte. The purpose of the MOV 0,0,SNC is to check the state of carry based on the previous MOVZR 2,3, which in turn compensates for the INCZL 2,2 that we did back in the main program. Notice that the combination of INC 2,2 and the MOVZR 2,3 will retain the same address for two go'rounds, but will alternate the state of carry to first print the high byte, then the low.

Ninety-nine percent of the software applications requesting keyboard input from the operator will echo the input back to the printer so that the operator will have "proof" of what key was stuck. One application where the input is not echoed would be "signed in" on a time-sharing system. So that unauthorized users cannot use your identification code, the system does not echo the code as you enter it. A slightly modified version of this is used in the program that follows.

<u>Problem:</u> Write a program that will input characters from the Teletype keyboard and pack them two characters per location in memory. If the character is a carriage return (CR), store a line feed (LF) along with it. Use the ESC character to signify end-of-input. Only after receiving the ESC character are the contents of the table to be echoed.

<u>ALGORITHM</u>

1. Initialize pointers for input and output.
2. Input a character and strip off parity.
3. If the character is an ESC, store a NULL character in the table, terminate the input and go to Step 6.
4. If the character is a CR, store it plus a LF in the table and go back to Step 2.
5. Pack all characters two per location.
6. Output the table.
7. Return to Step 1.

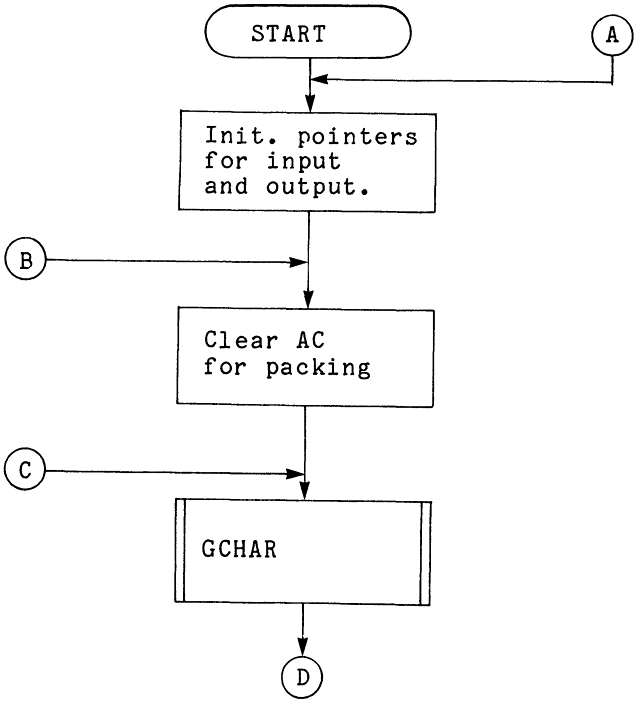


Figure 4.6 Flowchart

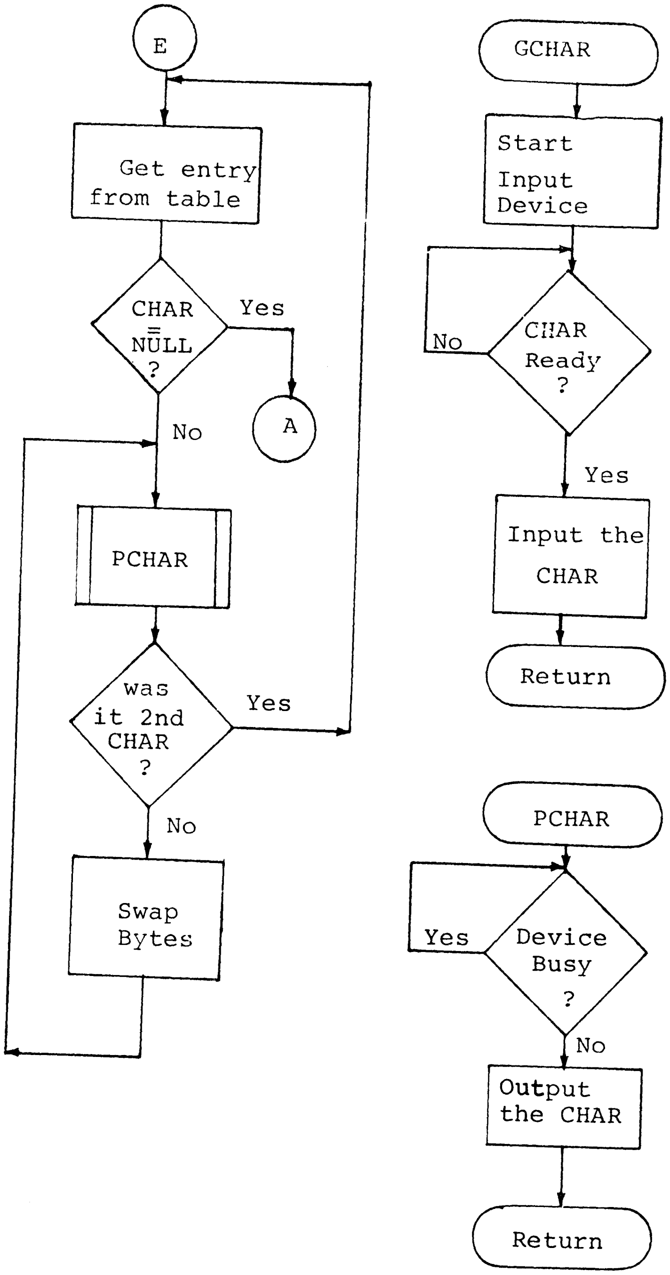


Figure 4.7 Flowchart (Cont.)

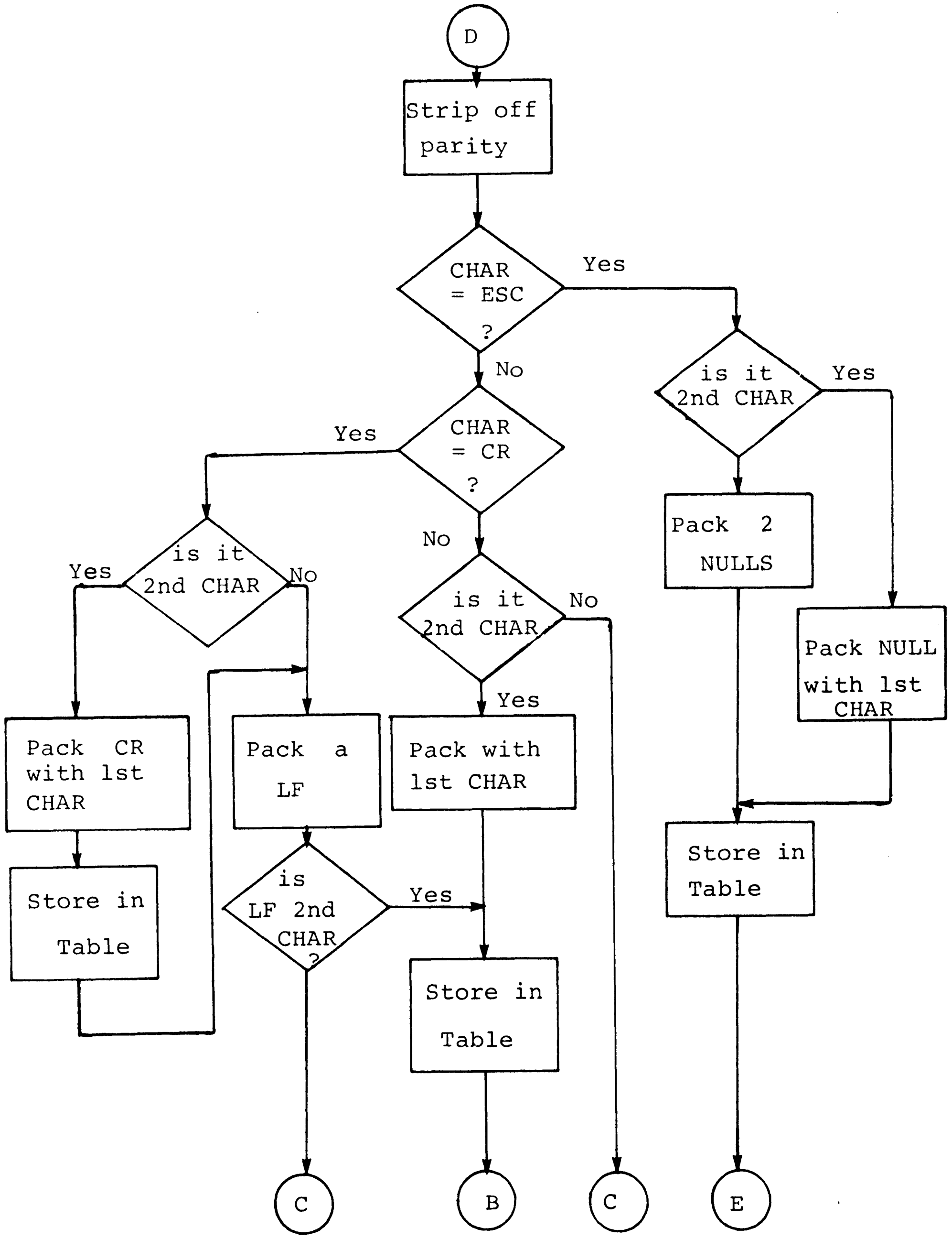


Figure 4.8 Flowchart (Cont.)

4.3
ARITHMETIC
AND LOGIC
INSTRUCTIONS
(Continued)

```
ATABL:  TABLE-1
START:  LDA     0,ATABL ;Get address of
                        ;Table.
        STA     0,20    ;Initialize for
        STA     0,21    ;Input and Output.
FIRST:  SUBO    1,1     ;Clear AC1 and
                        ;Carry.
SECND:  JSR     GCHAR   ;Get a CHARacter.
        LDA     2,C177  ;Get the mask.
        AND     2,0     ;Strip off parity.
        LDA     2,C33   ;Get ASCII ESC.
        SUB#    0,2,SZR ;CHAR = ESC?
        JMP     CR      ;No, try CR.
        SUBC    0,0     ;Yes, NULL to AC0,
                        ;retain carry.
        ADDS    0,1     ;Pack the NULL.
        STA     1,@20   ;Store in table.
        JMP     OUT     ;Go to output
                        ;routine.
C177:   177             ;Mask to strip off
                        ;parity.
C33:    33              ;ASCII ESC.
C15:    15              ;ASCII CR.
CR:     LDA     2,C15   ;Get ASCII CR.
        SUB#    0,2,SNR ;CHAR = CR?
        JMP     CRLF    ;Yes, process it.
        ADDCS   0,1,SZC ;No, Is it 2nd
                        ;Char?
        JMP     SECND   ;No, go get 2nd
                        ;char.
        STA     1,@20   ;Yes, store in
                        ;table.
        JMP     FIRST   ;Go get next char.
C12:    12              ;ASCII LF
CRLF:   ADDCS   0,1,SZC ;Is it 2nd Char?
        JMP     LF      ;No, add a LF.
        STA     1,@20   ;Yes, store in
                        ;table.
        SUBO    1,1     ;Clear AC1 and
                        ;Carry.
LF:     LDA     0,C12   ;Get ASCII LF.
        ADDCS   0,1,SZC ;Is LF 2nd Char?
        JMP     SECND   ;No, go get 2nd
                        ;Char.
        STA     1,@20   ;Yes, store in
                        ;table.
        JMP     FIRST   ;Go get next Char.
OUT:    SUBO    0,0     ;Clear AC0 and
                        ;Carry.
```

```
        LDA     0,@21   ;Get first char-
                        ;acters.
        MOV     0,0,SKP ;Output low byte
                        ;first.
SWAP:   MOVS    0,0     ;Swap for 2nd char-
                        ;acter.
        LDA     2,C177  ;Get low byte mask.
        AND     0,2,SNR ;Byte = NULL?
        JMP     START   ;Yes, go back for
                        ;input.
        JSR     PCHAR   ;No, Print the byte.
        MOVC    0,0,SZC ;Was it 2nd byte.
        JMP     SWAP    ;No, position 2nd
                        ;character.
        JMP     OUT     ;Yes, get more
                        ;characters.
GCHAR:  NIOS    TTI     ;Start INPUT device.
        SKPDN   TTI     ;Character Ready?
        JMP     .-1     ;No, test again.
        DIAC    0,TTI   ;Input Char., idle
                        ;device.
        JMP     0,3     ;Return to main
                        ;program.
PCHAR:  SKPBZ   TTO     ;Device Busy?
        JMP     .-1     ;Yes, test again.
        DOAS    0,TTO   ;No, Output Char
                        ;and Start device.
        JMP     0,3     ;Return to main
                        ;program.
TABLE:  0               ;TABle starts here.

        .END    START   ;Program is load
                        ;and go.
```

The preceding program communicates with the Teletype via programmed instructions; i.e., the program is dedicated to the device. Considering the instruction execution rate (approximately two microseconds per instruction) versus the speed of the Teletype (100 milliseconds per character), the program could have executed approximately 50,000 instructions while waiting for a single Teletype character. Rather inefficient use of CPU time wouldn't you say?
In the following chapter, I/O Device Handling, we will discuss more efficient methods of communicating with I/O devices.

Included are some additional special mnemonic instructions as promised (see page 5-2).

4.3
ARITHMETIC
AND LOGIC
INSTRUCTION
(Continued)

Before we leave the instruction set, we have an unfinished program to write. In our discussion of algorithms and flowcharts, we introduced the SORT routine; a routine for arranging random entries into ascending order. While there are many algorithms for sorting information (depending upon how many entries there are, and whether time is a consideration, et. al.), we have chosen a rather middle-of-the-road approach, suitable for tables of moderate length, in our algorithm and flowcharts (see pages 3-11 through 3-13). Here now is the coded solution to that problem.

```
                .TITL SORT
                .ENT ASORT START SORT DONE
                 SIZE ATABL1 TABL1

        .ZREL
SIZE:12
ATBL1:  TABL1
ATBL2:  0
ASORT:  SORT

        .NREL
START:  LDA     0,ATBL1 ;Get adrs of
                        ;TABL1.
        LDA     1,SIZE  ;Get size of
                        ;TABL1.
        STA     0,20    ;Set pointer to
        DSZ     20      ;TABL1 minus one.
        STA     1,XFER  ;Save size of
                        ;XFER count.
        ADD     1,0     ;Begin TABL2 at
        STA     0,ATBL2 ;TABL1 plus size.
        STA     0,21    ;Set pointer to
        DSZ     21      ;TABL2 minus one.
        LDA     0,020   ;Transfer entry
        STA     0,021   ;From TABL1 to
                        ;TABL2.
        DSZ     XFER    ;All transferred?
        JMP     .-3     ;No, go get next.
SORT:   LDA     0,SIZE  ;Yes, initialize
        NEG     0,0     ;Pass-count and
        COM     0,0
        STA     0,PASS  ;Compare kount
                        ;to
        STA     0,KOUNT ;size minus one.
REPT:   LDA     0,ATBL1 ;Initialize
                        ;pointers
        STA     0,20    ;back to the
        DSZ     20      ;beginning of
        STA     0,21    ;TABL2.
        DSZ     21
FIRST:  LDA     0,020   ;Get first entry.
NEXT:   LDA     1,020   ;Get next entry.
        SUB2#   1,0,SNC ;AC1 less than
                        ;AC0?
        JMP     LESS    ;No, AC0 less
                        ;than AC1.
```

(Continued)

```
GRATR:  STA     1,021   ;Yes, save AC1 in
                        ;TABL2.
        JMP     BUMP    ;Go to bump kount.
LESS:   STA     0,021   ;Save AC0 in TABL2.
        MOV     1,0     ;Move AC1 to AC0.
BUMP:   DSZ     KOUNT   ;One less to com-
                        ;pare.
        JMP     NEXT    ;Not done this pass.
        STA     0,021   ;If done, AC0 to
                        ;TABL2.
        DSZ     PASS    ;Last pass?
        JMP     .+2     ;No, adjust
                        ;pointers.
DONE:   JMP     DONE    ;Yes, done.
        LDA     0,PASS  ;Set new kount.
        STA     0,KOUNT ;From old pass.
        JMP     REPT    ;Go for next pass.
PASS:   0
KOUNT:  0
XFER:   0
TABLE1: 32
        14
        27
        12
        53
        35
        42
        11
        62
        20
        .END    START   ;Load and go.
```

CHAPTER 5

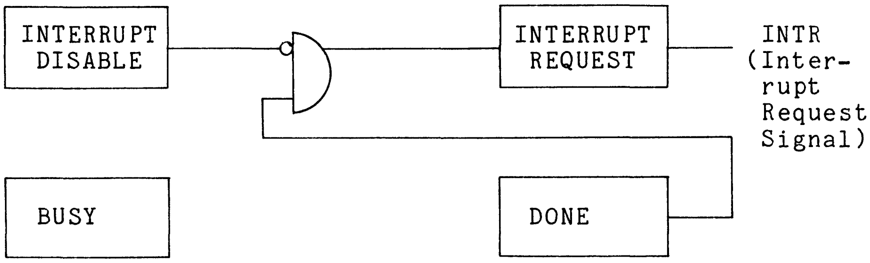
I/O DEVICE HANDLING

PROGRAM INTERRUPTS

Although peripheral devices may be serviced by the processor on a dedicated basis, as previously discussed, this usually results in extremely inefficient use of processor time and/or temporary neglect of all other devices.

To overcome this, a device interrupt and servicing facility is available. This facility provides for enabling and disabling devices from requesting service, establishing 16 levels of priority interrupts, and servicing devices only when they request service.

In addition to the BUSY and DONE flip-flops, every device has an Interrupt Disable flip-flop and an Interrupt Request flip-flop arranged logically as follows:



5.1
PROGRAM INTERRUPTS
(Continued)

Within the processor is an interrupt system status flag (ION). When the flag is reset, indicating that the interrupt system is disabled, no device can interrupt the processor. When the flag is set and the interrupt system is on, selected devices may request service via an interrupt.

The interrupt system is enabled by the instruction INTEN (NIOS CPU) and disabled by the instruction INTDS (NIOC CPU). The status of the interrupt system can be monitored by the ION indicator on the front panel or by the instructions:

SKPBZ CPU	SKIP NEXT INSTRUCTION if interrupts are disabled.
SKPBN CPU	SKIP NEXT INSTRUCTION if interrupts are enabled.

Thus, the following conditions must be met before a device can interrupt the processor.

1. The ION flag must be set. (Interrupts enabled.)
2. The device's Interrupt Disable flip-flop must be reset. (Interrupts allowed from the device.)
3. The device's DONE flip-flop must be set. (Device is ready for service.)

The commands for controlling the ION flag are:

INTEN	Interrupt Enable (set ION flag)
INTDS	Interrupt Disable (reset ION flag)

The command for controlling the individual Interrupt Disable flip-flops is:

```
MSKO    AC      ;MASK OUT
```

5.1 PROGRAM INTERRUPTS (Continued)

When a MSKO AC command is given, the Interrupt Disable flip-flop of every device is effectively connected to one of the 16 bit positions in accumulator AC. If the bit position contains a 1, all Interrupt Disable flip-flops connected to it are set, thus disabling those devices from requesting interrupts. If the bit position contains a 0, all Interrupt Disable flip-flops connected to it are reset, thus enabling those devices to request interrupts.

Because accumulator AC has 16 bit positions, there are 16 possible levels of interrupt priority.

5.1.1 Example

A program is used for dedicated service as a controller for a lathe. However, it will permit only the Teletype keyboard input to request an interrupt. Enable the interrupt request facility for this device. (Assume the TTI Interrupt Disable flip-flop is connected to data line 14 on the I/O bus.)

```
LDA  0,MASK
MSKO 0   ⎫
INTEN    ⎬      DOBS    0,CPU
NIOS TTI ⎭
________
MASK:177775  ;1/111/111/111/111/101
              disables all devices but
              those connected to data
              line 14 on the I/O bus.
```

The preceding example has taken care of two of the four preliminary steps in programmed interrupts. To use the programmed interrupt feature, you must prepare for it by doing the following:

5.1.1
Example
(Continued)

1. Prepare location 0 to hold the return address while in an interrupt routine. This means if you have information in location 0 that you don't want to lose, save it somewhere else.

```
    LDA     0,0
    STA     0,SAVO
```

2. Store in location 1 the address of the interrupt handler routine. The reason for this and the previous step will be detailed as we step through the interrupt sequence.

3. Set ION flip-flop by executing the INTEN instruction. This allows the CPU to acknowledge the interrupt when it occurs.

4. Initiate an operation in the device.

Steps 3 and 4 are handled in the preceding example by the last two instructions shown.

```
        INTEN   ;INTerrupt ENable sets ION
                 flip-flop.

        NIOS TTI;Start the low-speed reader
                 to assemble a character in
                 the device's data buffer.
```

After these preliminary steps have been taken care of, the program continues executing instructions (approx. 50,000 in the case of TTI) while waiting for the interrupt. Every time the program references memory to fetch an instruction, an address, or an operand, it also queries all devices with, "Does anybody want service?" A device requesting service on an interrupt basis does so for one of two purposes; to inform the program that:

a. "I have completed what you told me to do," or,

b. "I was unable to complete what you told me me to do."

5.1.1
Example
(Ccontinued)

The latter is only possible from more sophisticated devices such as magnetic tape drives and magnetic disc drives and will be discussed later under the topic Data Channel. When the interrupt request comes in, the program will complete the instruction currently being executed; then, CPU, what's your job?

> "First, I will clear the ION flag, thereby disabling any further interrupts. This will allow the programmer to determine who is generating this interrupt and handle it accordingly without further interrupts."
>
> "Secondly, I will take the current value in the program counter (PC) and save it in location zero. This will allow the programmer to return to the interrupted program after servicing this device."
>
> "Lastly, I will execute a JMP @1 instruction, thereby transferring program control to what should be an interrupt handler routine."

That's it folks; the hardware has done its thing. The rest is up to you.

What are the types of things that your interrupt handler should do? Perhaps the first thing it should do is to determine who is generating this interrupt. The technique for doing so is partly a function of how many devices are connected to the I/O bus, and secondly the type of interrupt priority structure that you desire. One technique called "polling," will test each device's Done flag, looking for that device whose Done flag is set. This technique establishes the device you test first. The sequence on the following page illustrates this technique.

5.1.1
Example
(Continued)

```
        0/        0        ;Location 0 prepared to
                           ;hold return address.
        1/        HNDLR    ;Location 1 contains the
                           ;address of the interrupt
                           ;handler routine.
         .
         .
         .
HNDLR:            SKPDZ    CPU       ;Highest priority is
                                     ;given to the
                  JMP      PWRDN     ;Power-fail-Auto-
                                     ;Restart option.
                                     ;If the interrupt
                                     ;is from this option,
                                     ;control is trans-
                                     ;ferred to the PoWeR
                                     ;DowN routine.
                  SKPDZ    PTR       ;Next priority is
                                     ;high-speed Paper
                                     ;Tape Reader.
                  JMP      PTRSV     ;Xfer to PTR service
                                     ;routine.
                  SKPDZ    TTI       ;If all devices have
                  JMP      TTISV     ;been tested and
                  JMP      ERROR     ;none of them respond,
                                     ;we should be pre-
                                     ;pared to handle
                                     ;this situation
                                     ;(false interrupt).
```

Depending upon how sophisticated you want to be, the ERROR routine might:

a. Simply HALT the program;
b. Type a suitable message to the operator: ERROR: FALSE INTERRUPT and then HALT; or,
c. Attempt to investigate and correct the situation and ultimately return to the program from whence it came. The polling technique is satisfactory for a system with relatively few devices.

5.1.1
Example
(Continued)

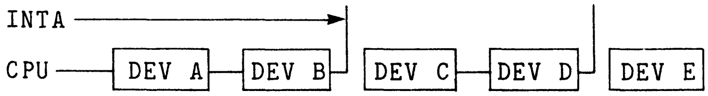
The second and third techniques generally work together. The second technique, called "broadcasting," asks the interrupting device to identify itself by asserting its unique device code. This code is loaded into the specified accumulator and can be used as a displacement in a table of service routine addresses. The broadcasting technique is implemented by the special mnemonic instruction INTA AC. Since the power/fail-auto/restart option, although acting like an I/O device, is not assigned a device code, the broadcasting technique should be preceded by a check on the CPU Done flag.

```
HNDLR:  SKPDZ   CPU     ;Check power failure
                        ;first.
        JMP     PWRDN   ;If yes, go to PoWeR
                        ;DownN routine.
        INTA    0       ;Code of interrupting
                        ;device goes to AC0.
        LDA     2,JTAB  ;Get start address
                        ;of Jump TABle.
        ADD     0,2     ;Add device code as
                        ;a displacement
        JMP     @0,2    ;Jump to the inter-
                        ;rupting device's
                        ;service routine.
JTAB:   ERROR           ;Displacement of
                        ;zero means code
                        ;was zero.
        .
        .
        .
JTAB+10 TTISV           ;Displacement of ten
                        ;indicates inter-
                        ;rupting device was
                        ;Teletype input (key-
                        ;board or reader).
JTAB+11 TTOSV           ;Device code 11 is
                        ;Teletype output
                        ;(printer/punch).
        .
        .
        .
JTAB+30 ERROR           ;No device currently
                        ;assigned code 30.
```

5.1.1
Example
(Continued)

NOTE: The labels JTAB+nn are for demonstration only. The assembler would reject any label containing an arithmetic operator.

The broadcasting technique, also referred to as "who are you," establishes device priority on the basis of electrical proximity; devices closest to the CPU have a higher priority. All devices are connected serially by a hand-holding scheme called a daisy-chain. When a device requests an interrupt, it does so by raising its hand, thereby breaking the chain. If two devices request simultaneously, the closest device to the CPU is serviced first, since the second device never receives the "who are you" signal.



After the device has been identified by either the polling or the "who are you" technique, and before the interrupt system is turned on again by the INTEN instruction, you might want to employ the third technique of priority structure; priority on the basis of who you will allow to request interrupt service. This is done with the MSKO AC instruction. Our sample program (page 5-3) did this before the interrupt system was turned on.

5.1.1
Example
(Continued)

Similar to the jump table that was used with the INTA AC instruction, so too the interrupting device's code can be used as a displacement into a table of mask words. Basically the question being answered is, "If the interrupt is from device X, then, while servicing device X, what other devices do I want to acknowledge?" As previously pointed out, a zero in the mask bit enables the device; a one disables it. (See example, page 5-3.)

Between the interrupt handler routine and the individual device's service routine we have thus far determined who is generating this interrupt and, on that basis, who we will allow to generate further interrupts. Since we are going to allow further interrupts while servicing this one, it becomes extremely important to save the content of location zero. This and other housekeeping chores may include any combination or all of the following:

1. Save all or some combination of the accumulators. If you are always going to save all of them, it could be a function of the handler routine. Or, based on the complexity of the device's service routine, some combination of accumulators might be saved.

2. Save the state of carry. If the program you are coming from relies on carry for branching decisions, then the original state of carry must be returned to that program. Saving carry can be done in conjunction with saving the 15-bit return address now in location 0.

5.1.1
Example
(Continued)

3. Save location 0. Location 0 could contain the return address to the main program or to a previous level of interrupt. After the accumulators have been saved, the return address and carry may be saved in one location by the following instruction sequence:

```
LDA     0,0      ;Get the return
                 ;address.
MOVL    0,0      ;Shift Carry into
                 ;bit 15.
STA     0,SAVE   ;Save both as

           0             14  15
          | ret. address    | C |
```

4. Save the current mask. Since each level of interrupt and the main program has its own priority mask, this information should be saved before proceeding to another level.

5. Save the stack pointer. A stack is just another table set aside in memory where information is generally accessed on a last-in-first-out (LIFO) basis. The stack pointer usually points to the first available location on the stack. The stack technique is used in the accompanying program to do the saving mentioned above.

After saving all the necessary parameters, and before actually servicing the device, the interrupt system is again enabled with an INTEN instruction.

5.1.1
Example
(Continued)

The actual servicing of the device generally consists of a check on the device's status register (if applicable) to determine the reason for the interrupt. If Error is set, the device is telling you, "I was unable to complete what you told me to do." If in processing this interrupt, the interrupting device has not been masked out (MSKO), then the device's Done flag must be cleared prior to enabling the interrupt system.

After servicing the device, and before restoring all of the information that was saved, the interrupt system should be disabled with an INTDS instruction so that the restoration can take place without possible loss of data. From the time the interrupt system is enabled (INTEN), the CPU guarantees you the execution of one instruction before it will acknowledge another interrupt. This one instruction is generally the JMP that returns you to the previous level of program.

The following program incorporates the techniques just discussed. It does not, however, carry the program to the individual device service routine level; rather, it shows the possible handling of false interrupts.

```
                        SAMPLE INTERRUPT HANDLER ROUTINE

                ;LAYOUT OF STACK ENTRY

        000000  SAC3=0  ;SAVE FOR AC3
        000001  SAC0=1  ;SAVE FOR AC0
        000002  SAC1=2  ;SAVE FOR AC1
        000003  SAC2=3  ;SAVE FOR AC2
        000004  SCRY=4  ;SAVE FOR CARRY
        000005  SRTN=5  ;SAVE FOR RETURN ADDRESS (WORD0)
        000006  SMSK=6  ;SAVE FOR CURRENT MASK

        000001          .LOC    1
00001   000400          ISR

        000400          .LOC    400             ;LOAD IN SECOND PAGE
00400   056464  ISR:    STA     3,@ADSTK        ;NO-SAVE AC3 IN STACK
00401   034463          LDA     3,ADSTK         ;AC3 ADDRESS OF STACK
00402   041401          STA     0,SAC0,3        ;SAVE ACCUMULATORS
00403   045402          STA     1,SAC1,3
00404   051403          STA     2,SAC2,3
00405   102560          SUBCL   0,0             ;SAVE CARRY
00406   041404          STA     0,SCRY,3
00407   020000          LDA     0,0             ;SAVE RETURN ADDRESS
00410   041405          STA     0,SRTN,3
00411   020456          LDA     0,CMASK         ;SAVE CURRENT MASK
00412   041406          STA     0,SMSK,3
00413   030455          LDA     2,SIZE          ;PUSH STACK
00414   157000          ADD     2,3
00415   054447          STA     3,ADSTK
00416   061477          INTA    0               ;AC0=DEVICE CODE
00417   030446  ISR1:   LDA     2,AMTAB         ;AC2=ADDR-1 OF MASK
                                                ;TAB
00420   113000          ADD     0,2             ;AC2=ADDRESS OF MASK
00421   031000          LDA     2,0,2           ;AC2=NEW MASK
00422   050445          STA     2,CMASK         ;SET CMASK TO NEW MASK
00423   034443          LDA     3,AJTAB         ;AC3=ADDR-1 OF JUMP TAB
00424   117000          ADD     0,3             ;AC3=ADDR OF ADDR WORD
00425   072177          DOBS    2,CPU           ;MSKO AND TURN ON INT
00425   007400          JSR     @0,3            ;EXIT TO ROUTINE
00427   060277          INTDS                   ;DISABLE INTERRUPTS
00430   034434          LDA     3,ADSTK         ;POP STACK
00431   030437          LDA     2,SIZE
00432   156400          SUB     2,3
00433   031406          LDA     2,SMSK,3        ;AC2=OLD MASK
00434   072077          MSKO    2               ;ISSUE OLD MASK
```

```
00435   061477           INTA     0                 ;GET DEVICE CODE
00436   101004           MOV      0,0,SZR           ;SKIP IF NO INTS
00437   000760           JMP      ISR1              ;PROCESS PENDING INT
00440   054424           STA      3,ADSTK           ;UPDATE POINTER
00441   050426           STA      2,CMASK           ;UPDATE MASK
00442   021405           LDA      0,SRTN,3          ;RESTORE RETURN ADDRESS
00443   040000           STA      0,0
00444   021404           LDA      0,SCRY,3          ;RESTORE CARRY
00445   101220           MOVZR    0,0
00446   021401           LDA      0,SAC0,3          ;RESTORE AC0 THRU AC2
00447   025402           LDA      1,SAC1,3
00450   031403           LDA      2,SAC2,3
00451   036413           LDA      3,@ADSTK          ;RESTORE AC3
00452   060177           INTEN                      ;ENABLE INTERRUPTS
00453   002000           JMP      @0                ;RETURN TO ROUTINE

                 ;ROUTINE TO IGNORE INTERRUPTS.

00454   024405   IGNOR   LDA      1,CLEAR           ;LOAD NIOC COMMAND
00455   123000           ADD      1,0               ;ADD IN DEVICE CODE
00456   040401           STA      0,.+1             ;STORE IN NEXT
00457   000000           0                          ;EXECUTE NIOC COMMAND
00460   001400           JMP      0,3               ;RETURN TO ROUTINE
00461   060200   CLEAR:  NIOC     0

                 ;ERROR HALTS.

00462   063077   ERROR:  HALT
00463   000771           JMP      IGNOR

                 ;STORAGE AND ADDRESS CONSTANTS.

00464   000545   ADSTK:  STACK                      ;ADDRESS OF PUSHDOWN
                                                    ;STACK
00465   000474   AMTAB:  MTAB-1                     ;ADDR-1 OF MASK TABLE
00466   000506   AJTAB:  JTAB-1                     ;ADDR-1 OF JUMP TABLE
00467   000000   CMASK:  0                          ;STORAGE FOR CURRENT
                                                    ;MASK
00470   000007   SIZE:   7                          ;SIZE OF STACK ENTRY
                                                    ;(7 WORDS)
```

```
                ;MASK TABLE.

        177777  ALL=177777                              ;MASK TO DISABLE ALL
                                                        ;INTERRUPTS.
00471   177777  MTAB:   ALL
00472   177777          ALL
00473   177777          ALL
00474   177777          ALL
00475   177777          ALL
00476   177777          ALL
00477   177777          ALL
00500   177777          ALL
00501   177777          ALL
00502   177777          ALL

                ;JUMP TABLE.

        000464  ERR=ERROR

00503   000464  JTAB:   ERR
00504   000464          ERR
00505   000464          ERR
00506   000464          ERR
00507   000464          ERR
00510   000464          ERR
00511   000464          ERR
00512   000464          ERR
00513   000464          ERR
00514   000464          ERR

                ;INITIALIZATION ROUTINE.

00515   024420  INIT:   LDA     1,ASTK                  ;INITIALIZE POINTER
00516   044746          STA     1,ADSTK
00517   126400          SUB     1,1                     ;ZERO CURRENT MASK
00520   044747          STA     1,CMASK
00521   020415          LDA     0,ADERR                 ;AC0=A (ERROR ROUTINE)
00522   024415          LDA     1,MALL                  ;AC1=FULL MASK
00523   030415          LDA     2,M12                   ;AC2=10
00524   034741          LDA     3,AMTAB                 ;MEM(20)=A(MTAB)-1
00525   054020          STA     3,20
00526   034740          LDA     3,AJTAB                 ;MEM(21)=A(JTAB)-1
00527   054021          STA     3,21
00530   042021  INIT1:  STA     0,@21                   ;ENTER IN JTAB
00531   046020          STA     1,@20                   ;ENTER IN MTAB
00532   151404          INC     2,2,SZR                 ;LOOP 10 TIMES
00533   000775          JMP     INIT1
00534   063077          HALT
```

```
00535   000545  ASTK:   STACK   ;ADDRESS OF STACK
00536   000464  ADERR:  ERROR   ;ADDRESS OF ERROR ROUTINE
00536   177777  MALL:   ALL     ;MASK TO ENABLE ALL INTS
00540   177766  M12:    -12     ;MINUS 10

        000043  STACK:  .BLK    5*7

                        .END
```

As a logical extension of the topic interrupts, something should be said about the power monitor-auto restart option.

5.2
POWER MONITOR AND AUTO-RESTART

The optional power monitor warns a program when power is failing by setting the Power Failure flag. If a system contains this option, the monitor will appear as any other I/O device to the interrupt system, except that it does not respond to an INTA command and must be serviced by:

```
        SKPDN          CPU
                 or
        SKPDZ          CPU
```

The first function of the interrupt service routine should be to test this Power Failure flag. If this is the interrupting device, the program has 1 to 2 milliseconds to save the contents of the accumulators, Carry, and the contents of location 0, to put a JMP to the desired restart address in location 0, and then to HALT.

With the power switch in the LOCK position, when POWER UP occurs, the instruction in location 0 will be executed.

Additional Suggestions:

1. If the system is of any size it probably has a Real Time Clock. * PWRDN should record time of failure and PWRUP should print "Power Failed at HH:MM:SS."
2. A location in core should keep track of active I/O devices. PWRUP could then print "The following devices were active:"
3. PWRUP should clear all device flags before enabling interrupts. This could be one instruction: DICS 0,CPU = IORST INTEN.

* PWRDN - power down
PWRUP - power up

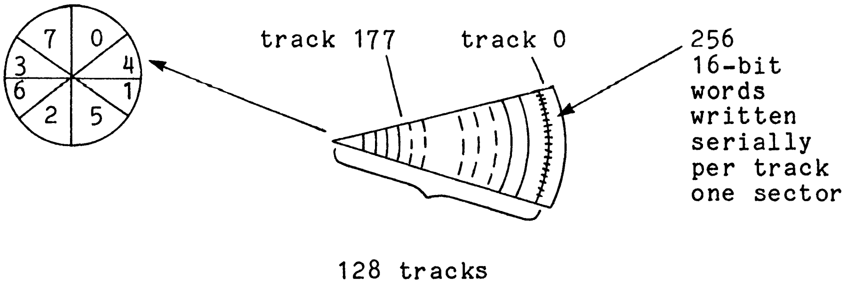
5.3 DATA CHANNEL

The final aspect of I/O device handling allows fast devices direct access to memory (DMA) for high speed data transfers. The term we use for DMA is Data Channel.

As an example of a data channel device, let's look at the fixed-head disc. The mnemonic for fixed-head disc is DSK, the device code is 20, and the priority mask bit is 9. As a data channel device, the DSK is given the memory address involved in the transfer, and the disc address involved in the transfer, and told in which direction the transfer is to take place. The direction is specified as a read (transfer from disc to memory) or a write (transfer from memory to disc). To understand the concept of disc address, we need to know something about how data is stored on the disc.

The disc surface is divided into eight pie-shaped wedges called sectors. Each sector is divided into bands, called tracks, which start toward the outside edge and work toward the center. The tracks are concentric bands as opposed to a phonograph record, which has a single groove that spirals toward the center. On each track within each sector, there are 256 16-bit words recorded serially.

5.3
DATA
CHANNEL
(Continued)



Additionally, one disc controller can handle up to eight disc drives. So, in providing the controller with a disc address, you must specify which disc unit, which track, and which sector. This may be accomplished with the DOA AC,DSK instruction where the content of the specified accumulator provides the following information:

	1 of 8	1 of 128	1 of 8
not used	disc	track	sector
0 1 2	3 4 5	6 7 8 9 10 11 12	13 14 15

The second requirement, providing the disc with the first memory address involved in the transfer, is accomplished with a DOB AC,DSK instruction. In this case, the specified accumulator should contain a zero in bit 0. (A one in bit 0 places the controller in diagnostic mode.)

The third factor, specifying a read or write, is done with the I/O S and P pulses.

5.3 DATA CHANNEL (Continued)

```
Example:        NIOS   DSK        ;initiate a read
                                   operation.

    or          NIOP   DSK        ;initiate a write
                                   operation.
```

Just for the sake of explanation, let us assume the disc has been initiated to do a read operation. After the controller finds the proper unit, track, and sector, the first 16-bit word begins serially shifting into the data buffer register. When the word is fully assembled, it does a parallel transfer into the output data buffer register.

Coinciding with this parallel transfer, the controller raises its data channel request (DCHR) flag. While this is taking place in the disc controller, the CPU continues to fetch and execute instructions. Just as it did for interrupt requests, every time the CPU references memory it also asks, "Does anybody want service?" If both an interrupt request and a data channel request (two different devices) occur simultaneously, the data channel request has a higher priority. If two data channel requests occur simultaneously, a daisy chain priority scheme, similar to the interrupt daisy chain, acknowledges the closest device first. When the CPU acknowledges the DCHR, the requesting device then passes to the CPU the memory address involved and also the direction in which the data transfer is to take place. This information is then followed by the actual data word. At the end of this single word DCH transfer, the disc controller increments its memory address buffer (B buffer) in preparation for the next single word transfer, and decrements its word count buffer (more on that later). Meanwhile, back at the data buffer, the second 16-bit word has been serially shifting in

5.3
DATA
CHANNEL
(Continued)

behind the first. If this second word is ready to do its parallel transfer to the output buffer before the first word is transferred to memory, there is going to be a loss of data as the second word overwrites the first. This is called a "data late" error. On some CPUs, this can be the result of too long an indirect addressing chain. On these CPUs, you are advised to keep your indirect addressing chains short while data channel devices are active.

On the subject of word count (WC), some devices (such as magnetic tape) allow the programmer to specify the number of words to be transferred via data channel. The disc, however, will always transfer a fixed number of words: one sector, or 256 16-bit words. For each DCHR, one 16-bit word is transferred. In between requests from the disc (approx. 8 microseconds), other data channel devices may also be making requests and transferring data or, the CPU may be servicing an interrupt or just executing programs. When the device has completed its data transfers (WC = 0), it will set its DONE flag and generate an interrupt (if enabled). The device service routine should then check the status of the device to determine if the transfers took place properly. In the case of disc, this may be done with a DIA AC,DSK instruction.

```
NOTE:   DOA  AC,DSK loads the disc address
                    buffer, but
        DIA  AC,DSK reads the disc status
                    buffer.
```

The information in the specified accumulator received from the status register is as follows:

5.3 DATA CHANNEL (Continued)

UNUSED						
0	1	2	3	4	5	6

shift register bit 0	first buffer full	second buffer full	write data	write error	data late	no such disc	data error	error
7	8	9	10	11	12	13	14	15

Bits 7 through 10 are for maintenance only and are not discussed further here. Clear, Start and Pulse clear all of these flags.

Bit	Meaning
11	The program has specified Write and the selected track-sector is write-protected. The setting of this bit clears Busy and sets Done, requesting an interrupt if Interrupt Disable is clear.
12	The data channel has failed to respond in time to a request for access (e.g., because of a long instruction or preemption of the channel by faster devices).
13	The disc selected by the program is not connected to the bus. The setting of this bit clears Busy and sets Done, requesting an interrupt if Interrupt Disable is clear.

5.3
DATA
CHANNEL
(Continued)

Bit	Meaning
14	In Read, the cyclic check word read from the disc differed from that computed by the control for the data in the block.
15	Bit 11, 12, 13, or 14 is 1.

Additional information about programming the disc or other I/O devices may be found in another Data General document entitled "Programmers Reference Manual for Peripherals" (DG publication #015-000021).

APPENDIX A

SAMPLE PROGRAMS

The following programs illustrate some of the features described in this document. You should examine them for their operational merit, but also feel free to modify them for your own personal applications. All of the programs are written as independent subroutines with page 0 linkages.

Page	Title
2	GCHAR
3	PCHAR
4	PRINT
7	BNOCT
9	BNDEC

SAMPLE
PROGRAMS
(Continued)

```
;Title:  GCHAR
;Routine to read characters from the
;Teletype.  As each character is read,
;the parity bit is stripped off and the
;character is placed right-justified in
;AC0.  This routine is called through its
;page 0 link as follows:
;       JSR  @AGCHR

AGCHR: GCHAR             ;Page 0 link to GCHAR.

GCHAR: STA      3,RET    ;Save the return
                         ;address.
       NIOS     TTI      ;Start the Teletype.
       SKPDN    TTI      ;Character ready?
       JMP      .-1      ;No, test again.
       DIAC     0,TTI    ;Get Char. and idle
                         ;TTI.
       LDA      3,MSK    ;Get the mask.
       AND      3.0      ;Keep right 7 bits.
       JMP      @.+1     ;Return to calling
                         ;program.
RET:   0                 ;Return address held
                         ;here.
MSK:   177               ;Mask for right 7 bits.
```

SAMPLE
PROGRAMS
(Continued)

```
;Title:  PCHAR
;Routine to print characters on the
;Teletype.  If a character is a
;carriage return (CR), the program
;automatically generates a line feed
;(LF).  The program is called through its
;page 0 link as follows:
;       JSR     @APCHR  ;Page 0 link to
                        ;PCHAR.

PCHAR:  STA     3,RET   ;Save the return
                        ;address.
        JSR     OUT     ;Print the char-
                        ;acter.
        LDA     3,CR    ;Get ASCII CR.
        SUB#    0,3,SZR ;Char = CR?
        JMP     @RET    ;No, return to
                        ;calling program.
        LDA     0,LF    ;Yes, get ASCII
                        ;LF.
        JSR     OUT     ;Print a LF.
        LDA     0,CR    ;Restore the CR.
        JMP     @RET    ;Return to calling
                        ;program.
OUT:    SKPBZ   TTO     ;Device busy?
        JMP     .-1     ;Yes, test again.
        DOAS    0,TTO   ;No, output char.
                        ;and start TTO.
        JMP     0,3     ;Return to PCHAR
                        ;routine.
RET:    0               ;Return address
                        ;to calling
                        ;routine.
CR:     15              ;ASCII carriage
                        ;return.
LF:     12              ;ASCII line feed.
```

SAMPLE
PROGRAMS
(Continued)

```
;Title:  PRINT
;The following routine may be used to
;output text messages packed left to
;right using Assembler pseudo-ops.
;Example:
;     .TXTM 1
; MSG:.TXT/MESSAGE/
;
;Messages packed by .TXT automatically
;end with a NULL byte.  When "PRINT"
;detects the NULL, it substitutes a
;carriage return (CR).  When "PCHAR"
;receives the CR, it automatically
;executes a carriage return (CR) and
;line feed (LF).  To prevent this
;automatic CR LF, the message should end
;with the BELL character as follows:
;     .TXTM 1
; MSG:.TXT/MESSAGE <7>/

;This routine begins by saving the state
;of the machine (accumulators and Carry)
;before calling PCHAR to output the
;message.  At the completion of the
;message, the original state is restored.

;This program is called through its page
;0 link as follows:
;       JSR     @APRNT
;       MSG*2
;       MORE PROGRAM
;The word following the call (MSG*2) is
;a trailing argument byte-pointer to
;the message to be printed.
```

```
;Title: PRINT

APRNT:PRINT

PRINT:  STA     0,SAC0          ;Save AC0
        STA     1,SAC1          ;Save AC1
        STA     2,SAC2          ;Save AC2
        INCL    3,2             ;Combine return
                                ;address with
                                ;Carry.
        STA     2,PC.CRY        ;Save both.
        LDA     1,BELL          ;Get ASCII BELL.
        LDA     2,0,3           ;Get MSG address.
MORE:   MOVZR   2,3             ;Adrs ÷ 2, Byte
                                ;Pointer to
                                ;Carry.
        LDA     0,0,3           ;Get first two
                                ;char.
        MOV     0,0,SNC         ;Which Byte?
        MOVS    0,0             ;C=0, high byte
                                ;first.
        LDA     3,MSK           ;Get low-byte
                                ;mask.
        AND     3,0             ;Mask for bits
                                ;9-15.
        JSR     @APCHR          ;Go print char-
                                ;acter.
        SUB#    0,1,SNR         ;Char=BELL?
        JMP     DONE            ;Yes, done.
        MOV     0,0,SNR         ;Char=NULL?
        JMP     .+3             ;Yes, substitute
                                ;a CR.
        INC     2,2             ;No, bump byte
                                ;pointer.
        JMP     MORE            ;Go get more
                                ;message.
        LDA     0,CR            ;Get ASCII CR.
        JSR     @APCHR          ;Print CR and LF.
```

SAMPLE
PROGRAMS
(Continued)

```
DONE:   LDA     3,PC.CRY        ;Get combined
                                ;PC and Carry.
        MOVZR   3,3             ;Restore Carry.
        LDA     2,SAC2          ;Restore AC2.
        LDA     1,SAC1          ;Restore AC1.
        LDA     0,SAC0          ;Restore AC0.
        JMP     0,3             ;Return to
                                ;calling program.
SAC0:   0                       ;
SAC1:   0                       ;Temporary stor-
                                ;age for ACs.
SAC2:   0                       ;
PC.CRY  0                       ;Combined return
                                ;address and
                                ;Carry.
BELL:   7                       ;ASCII BELL.
MSK:    177                     ;Mask to save
                                ;bits 9-15.
CR:     15                      ;ASCII carriage
                                ;return.
```

SAMPLE
PROGRAMS
(Continued)

```
;Title:  BNOCT
;Binary to octal conversion routine.
;The routine converts a 16-bit binary
;integer to an ASCII Character String
;for output.  The integer to be con-
;verted is assumed to be in AC1.  This
;routine calls "PCHAR:" for the printing
;of the octal digits.  This routine is
;called through its page 0 link as follows:
;       LDA     1,DIGIT
        JSR     @ABOCT

ABOCT:BNOCT             ;Page 0 link to
                        ;BNOCT

BNOCT:  STA     0,SAC0  ;Save AC0
        STA     2,SAC2  ;Save AC2
        MOVL    3,3     ;Combine return
                        ;address with
                        ;Carry.
        STA     3,PC.CRY;Save both.
        SUBZR   2,2     ;Set AC2=100000,
                        ;octal constant.
LOOP:   LDA     0,C60   ;Set AC0=ASCII
                        ;zero.
        SUBO    2,1,SNC ;Subtract octal
                        ;constant from
                        ;integer.
        INC     0,0,SKP ;No underflow,
                        ;inc ASCII Char.
        ADD     2,1,SKP ;If underflow,
                        ;add back.
        JMP     .-3     ;No underflow,
                        ;try subtract
                        ;again.
        JSR     @APCHR  ;Print the digit.
        MOVZR   2,2     ;Generate next
                        ;octal constant.
        MOVZR   2,2     ;
        MOVZR   2,2,SZR ;Last digit con-
                        ;verted?
        JMP     LOOP    ;No, continue.
        LDA     3,PC.CRY;Get return
                        ;address and
                        ;Carry.
        MOVZR   3,3     ;Restore both.
        LDA     2,SAC2  ;Restore AC2.
        LDA     0,SAC0  ;Restore AC0.
```

SAMPLE
PROGRAMS
(Continued)

```
                JMP     0,3       ;Return to calling
                                  ;routine.
        SAC0:   0                 ;Temporary stor-
        SAC2:   0                 ;age for accumu-
        PC.CRY: 0                 ;lators, return
        C60:    60                ;address and
                                  ;Carry.
                                  ;ASCII ZERO.
```

SAMPLE
PROGRAMS
(Continued)

```
;Title:  BNDEC
;Binary to decimal conversion routine.
;This routine converts a 16-bit binary
;integer to an ASCII character string
;for output.  The integer to be con-
;verted is assumed to be in AC1.  This
;routine calls "PCHAR" for the printing
;of the decimal digit in AC0.  This
;routine is called through its page 0 link
;as follows:
;       LDA    1,DIGIT
;       JSR    @ABDEC

ABDEC:BNDEC             ;Page 0 link to
                        ;BNDEC

BNDEC:  STA     0,SAC0  ;Save AC0
        STA     2,SAC2  ;Save AC2
        MOVL    3,3     ;Combine return
                        ;address with
                        ;Carry.
        STA     3,PC.CRY;Save both.
        LDA     3,INST  ;Set up LDA
                        ;command
        STA     3,.+1   ;with decimal
                        ;constant.
LOOP:   0               ;AC2=Power of Ten.
        LDA     0,C60   ;AC0=ASCII ZERO.
        SUBO    2,1,SNC ;Subtract decimal
                        ;constant from
                        ;integer
        INC     0,0,SKP ;No underflow,
                        ;inc ASCII char.
        ADD     2,1,SKP ;If underflow,
                        ;add back.
        JMP     .-3     ;No underflow, try
                        ;subtract again.
        JSR     @APCHR  ;Print the digit.
        ISZ     LOOP    ;Inc LDA command.
        MOVR    2,2,SNC ;Last digit con-
                        ;verted?
        JMP     LOOP    ;No, continue
        LDA     3,PC.CRY;Get return and
                        ;Carry.
        MOVZR   3,3     ;Restore both.
        LDA     2,SAC2  ;Restore AC2
        LDA     0,SAC0  ;Restore AC0
        JMP     0,3     ;Return to calling
                        ;routine.
```

SAMPLE
PROGRAMS
(Continued)

```
         .RDX     10       ;The information
                           ;which follows
                           ;is decimal.
TENS:    10000             ;Decimal 10000 =
                           ; Octal 23420.
         10000             ;Decimal 1000 =
                           ; Octal 1750.
         100               ;Decimal 100 =
                           ; Octal 144
         10                ;Decimal 10 =
                           ; Octal 12.
         1                 ;Decimal 1 = Octal
                           ; 1.
         .RDX     8        ;Return to octal
                           ;input mode.
INST:    LDA      2,.+17   ;This instr, gets
                           ;executed from
                           ;LOOP.
C60:     60                ;ASCII ZERO.
SAC0:    0                 ;Temporary storage
SAC2:    0                 ;for accumulators,
PC.CRY:  0                 ;return address
                           ;and Carry.
```

APPENDIX B

PROGRAMMING TRICKS

1. Clear AC and Carry.

```
SUBO    AC,AC
```

2. Clear AC and preserve Carry.

```
SUBC    AC,AC
```

3. Generate the indicated constants.

```
SUBZL   AC,AC              ;generate +1
ADC     AC,AC              ;generate -1
ADCZL   AC,AC              ;generate -2
```

4. Inclusive OR the content of two accumulators.

```
COM     0,0
AND     0,1
ADC     0,1
```

5. Exclusive OR the content of two accumulators.

```
MOV     1,2
ANDZL   0,2
ADD     0,1
SUB     2,1
```

6. Let ACX by any accumulator whose contents are zero.

```
INCZL   ACX,ACX            ;generate +2
INCOL   ACX,ACX            ;generate +3
INCS    ACX,ACX            ;generate +400
                                         8
```

PROGRAMMING TRICKS (Continued)

7. Without using a constant from memory:

   a. Subtract 1 from an accumulator.

```
⎧NEG     AC,AC
⎩COM     AC,AC
```

   b. Add +3 to an AC.

```
⎧INCR    AC,AC
⎩INCL    AC,AC
```

   c. Complement bit 0 in AC.

```
 ADDOR   AC,AC
```

8. Check if both bytes in an accumulator are equal.

```
MOVS    ACS,ACD
SUB     ACS,ACD,SZR
JMP     ---         ;not equal
---     ---         ;equal
```

9. Check if two accumulators are both zero.

```
MOV     ACS,ACS,SNR
MOV     ACS,ACD,SZR
JMP     ---         ;not both zero
---     ---         ;both zero
```

(This technique does not destroy either accumulator, nor does it alter Carry.)

PROGRAMMING TRICKS (Continued)

10. Check an ASCII character to make sure it is a decimal digit. The character is in ACS and is not destroyed by the test. Accumulators ACX and ACY are destroyed.

```
LDA      ACX,C60          ;ACX=ASCII
                          ;zero
LDA      ACY,C71          ;ACY=ASCII
                          ;nine
ADCZ#    ACY,ACS,SNC      ;skips if
                          ;(ACS)> 9
ADCZ#    ACX,ACX,SZC      ;skips if
                          ;(ACS)≥ 0
JMP      ---              ;not digit
---      ---              ;digit

C60      60               ;ASCII zero
C71      71               ;ASCII nine
```

11. Test an accumulator for zero.

```
MOV      AC,AC,SZR
JMP      ---              ;not zero
---      ---              ;zero
```

12. Test an accumulator for -1.

```
COM#     AC,AC,SZR
JMP      ---              ;not -1
---      ---              ;-1
```

13. Test an accumulator for 2 or greater.

```
MOVZR#   AC,AC,SNR
JMP      ---              ;less than 2
---      ---              ;2 or greater
```

14. Assume it is known that AC contains 0,1,2, or 3. Find out which one.

```
MOVZR#   AC,AC,SEZ
JMP      THREE            ;was 3
MOV      AC,AC,SNR
JMP      ZERO             ;was 0
MOVZR#   AC,AC,SZR
JMP      TWO              ;was 2
---      ---              ;was 1
```

PROGRAMMING
TRICKS
(Continued)

15. Multiply an AC by the indicated value.

```
MOV      ACX,ACX          ;multiply by 1

MOVZL    ACX,ACX          ;multiply by 2

MOVZL    ACX,ACY          ;multiply by 3
ADD      ACY,ACX

ADDZL    ACX,ACX          ;multiply by 4

MOV      ACX,ACY          ;multiply by 5
ADDZL    ACX,ACX
ADD      ACY,ACX

MOVZL    ACX,ACY          ;multiply by 6
ADDZL    ACY,ACX

MOVZL    ACX,ACY          ;multiply by 7
ADDZL    ACY,ACY
SUB      ACX,ACY          ;in ACY

ADDZL    ACX,ACX          ;multiply by 8
MOVZL    ACX,ACX

MOVZL    ACX,ACY          ;multiply by 9
ADDZL    ACY,ACY
ADD      ACY,ACX

MOV      ACX,ACY          ;multiply by 10
ADDZL    ACX,ACX                         10
ADDZL    ACY,ACX

MOVZL    ACX,ACY          ;multiply by 12
ADDZL    ACY,ACX                         10
MOVZL    ACX,ACX

MOVZL    ACX,ACY          ;multiply by 18
ADDZL    ACY,ACY                         10
ADDZL    ACY,ACX
```

APPENDIX C

INSTRUCTION MNEMONICS

NUMERIC
LISTING

000000	JMP
000001	SKP
000002	SZC
000003	SNC
000004	SZR
000005	SNR
000006	SEZ
000007	SBN
000010	#
002000	@
004000	JSR
010000	ISZ
014000	DSZ
020000	LDA
040000	STA
060000	NIO
060100	NIOS
060177	INTEN
060200	NIOC
060277	INTDS
060300	NIOP
060400	DIA
060177	READS
060500	DIAS
060600	DIAC
060800	DIAP
061000	DOA
061100	DOAS
061200	DOAC
061300	DOAP
061400	DIB
061477	INTA
061500	DIBS
061600	DIBC
061700	DIBP
062000	DOB
062077	MSKO
062100	DOBS
062200	DOBC
062300	DORP
062400	DIC
062500	DICS
062600	DICC
062677	IORST
062700	DICP
063000	DOC
063077	HALT
063100	DOCS
063200	DOCC
063300	DOCP
063400	SKPBN
063500	SKPBZ
063600	SKPDN
063700	SKPDZ
073101	DIV
073301	MUL
100000	@
100000	COM
10010	COM#
10020	COMZ
10030	COMZ#
10010	COMO
10050	COMO#
10060	COMC
10070	COML
100110	COM#
100120	COMZL
100130	COMZL#
100140	COMOL
100150	COMOL#
100160	COMCL
100170	COMCL#
100200	COMR
100210	COMR#
100220	COMZR
100230	COMZR#
100240	COMOR
100250	COMOR#
100260	COMCR
100270	COMCR#

NUMERIC
LISTING
(Continued)

100300	COMS	101070	MOVC#
100310	COMS#	101100	MOVL
100320	COMZS	101110	MOVL#
100330	COMZS#	101120	MOVZL
100340	COMOS	101130	MOVZL#
100350	COMOS#	101140	MOVOL
100360	COMCS	101150	MOVOL#
100370	COMCS#	101160	MOVCL
100400	NEG	101170	MOVCL#
100410	NEG#	101200	MOVR
100420	NEGZ	101210	MOVR#
100430	NEGZ#	101220	MOVZR
100440	NEGO	101230	MOVZR#
100450	NEGO#	101240	MOVOR
100460	NEGC	101250	MOVOR#
100470	NEGC#	101260	MOVCR
100500	NEGL#	101270	MOVCR#
100520	NEGZL	101300	MOVS
100530	NEGZL#	101310	MOVS#
100540	NEGOL	101320	MOVZS
100550	NEGOL#	101330	MOVZS#
100560	NEGCL	101340	MOVOS
100570	NEGCL#	101350	MOVOS#
100600	NEGR	101360	MOVCS
100610	NEGR#	101370	MOVCS#
100620	NEGZR	101400	INC
100630	NEGZR#	101410	INC#
100640	NEGOR	101420	INCZ
100650	NEGOR#	101430	INCZ#
100660	NEGCR	101440	INCO
100670	NEGCR#	101450	INCO#
100700	NEGS	101460	INCC
100710	NEGS#	101470	INCC#
100720	NEGZS	101500	INCL
100730	NEGOS	101510	INCL#
100750	NEGOS#	101520	INCZL
100760	NEGCS	101530	INCZL#
100770	NEGCS#	101540	INCOLO
101000	MOV	101550	INCOL
101010	MOV#	101560	INCCL
101020	MOVZ	101570	INCCL#
101030	MOVZ#	101600	INCR
101040	MOVO	101610	INCR#
101050	MOVO#	101620	INCRZR
101060	MOVC	101630	INCZR#

NUMERIC
LISTING
(Continued)

101640	INCOR	102460	SUBC
101650	INCOR#	102470	SUBC#
101660	INCCR	102500	SUBL
101700	INCS	102510	SUBL#
101710	INCS#	102520	SUBZL
101720	INCZS	102530	SUBZL#
101730	INCZS#	102540	SUBOL
101740	INCOS	102550	SUBOL#
101750	INCOS#	102560	SUBCL
101760	INCCS	102570	SUBCL#
101770	INCCS#	102600	SUBR
102000	ADC	102610	SUBR#
102010	ADC#	102620	SUZBR
102020	ADCZ	102630	SUBZR#
102030	ADCZ#	102640	SUBOR
102040	ADCO	102650	SUBOR#
102050	ADCO#	102660	SUBCR
102060	ADCC	102670	SUBCR#
102070	ADCC#	102700	SUBS
102100	ADCL	102710	SUBS#
102110	ADCL#	102720	SUBZS
102120	ADCZL	102730	SUBZS#
102130	ADCZL#	102740	SUBOS
102140	ADCOL	102750	SUBOS#
102150	ADCOL#	102760	SUBCS
102160	ADCCL	102770	SUBCS#
102170	ADCCL#	103000	ADD
102200	ADCR	103010	ADD#
102210	ADCR#	103020	ADDZ
102220	ADCZR	103030	ADDZ#
102230	ADCZR#	103040	ADDO
102240	ADCOR	103050	ADDO#
102250	ADCOR#	103060	ADDC
102260	ADCCR	103070	ADDC#
102270	ADCCR#	103100	ADDL
102300	ADCS	103110	ADDL#
102310	ADCS#	103120	ADDZL
102320	ADCZS	103130	ADDZL#
102330	ADCZS#	103140	ADDOL
102340	ADCOS	103150	ADDOL#
102350	ADCOS#	103160	ADDCL
102360	ADCCS	103170	ADDCL#
102370	ADCCS#	103200	ADDR
102400	SUB	103210	ADDR#
102410	SUB#	103220	ADDZR
102420	SUBZ	103230	ADDZR#
102430	SUBZ#	103240	ADDOR
102440	SUBO	103250	ADDOR#
102450	SUBO#	103260	ADDCR

NUMERIC LISTING (Continued)

103270	ADDCR#
103300	ADDS
103310	ADDS#
103320	ADDZS
103330	ADDZS#
103340	ADDOS
103350	ADDOS#
103360	ADDCS
103370	ADDCS#
103400	AND
103410	AND#
103420	ANDZ
103430	ANDZ#
103440	ANDO
103450	ANDO#
103460	ANDC
103470	ANDC#
103500	ANDL
103510	ANDL#
103520	ANDZL
103530	ANDZL#
103540	ANDOL
103550	ANDOL#
103560	ANDOL
103570	ANDOL#
103600	ANDR
103610	ANDR#
103620	ANDZR
103630	ANDZR#
103640	ANDOR
103650	ANDOR#
103660	ANDCR
103670	ANDCR#
103700	ANDS
103710	ANDS#
103720	ANDZS
103730	ANDZS#
103740	ANDOS
103750	ANDOS#
103760	ANDCS
103770	ANDCS#

ALPHABETIC LISTING

ADC	102000	Add the complement of ACS to ACD; use Carry as base for carry bit.
ADCC	102060	Add the complement of ACS to ACS; use complement of Carry as base for carry bit.
ADCCL	102160	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; rotate left.
ADCCR	102260	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; rotate right.
ADCCS	102360	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; swap halves of result.
ADCL	102100	Add the complement of ACS to ACD; use Carry as base for carry bit; rotate left.
ADCO	102040	Add the complement of ACS to ACD; use 1 as base for carry bit.
ADCOL	102140	Add the complement of ACS to ADC; use 1 as base for carry bit; rotate left.
ADCOR	102240	Add the complement of ACS to ACD; use 1 as base for carry bit; rotate right.
ADCOS	102340	Add the complement of ACS to ACD; use 1 as base for carry bit; swap halves of result.
ADCR	102200	Add the complement of ACS to ACD; use Carry as base for carry bit; rotate right.

ALPHABETIC
LISTING
(Continued)

ADCS	102300	Add the complement of ACS to ACD; use Carry as base for carry bit; swap halves of result.
ADCZ	102020	Add the complement of ACS to ACD; use 0 as base for carry bit.
ADCZL	102120	Add the complement of ACS to ACD; use 0 as base for carry bit; rotate left.
ADCZR	102220	Add the complement of ACS to ACD; use 0 as base for carry bit; rotate right.
ADCZS	102320	Add the complement of ACS to ACD; use 0 as base for carry bit; swap halves of result.
ADD	103000	Add ACS to ACD; use Carry as base for carry bit.
ADDC	103060	Add ACS to ACD; use complement of Carry as base for carry bit.
ADDCL	103160	Add ACS to ACD; use complement of Carry as base for carry bit; rotate left.
ADDCR	103260	Add ACS to ACD; use complement of Carry as base for carry bit; rotate right.
ADDCS	103360	Add ACS to ACD; use complement of Carry as base for carry bit; swap halves of result.

ALPHABETIC LISTING (Continued)

ADDL	103100	Add ACS to ACD; use Carry as base for carry bit; rotate left.
ADDO	103040	Add ACS to ACD; use 1 as base for carry bit.
ADDOL	103140	Add ACS to ACD; use 1 as base for carry bit; rotate left.
ADDOR	103240	Add ACS to ACD; use 1 as base for carry bit; rotate right.
ADDOS	103340	Add ACS to ACD; use 1 as base for carry bit; swap halves of result.
ADDR	103200	Add ACS to ACD; use Carry as base for carry bit; rotate right.
ADDS	103300	Add ACS to ACD; use Carry as base for carry bit; swap halves of result.
ADDZ	103020	Add ACS to ACD; use 0 as base for carry bit.
ADDZL	103120	Add ACS to ACD; use 0 as base for carry bit; rotate left.
ADDZR	103220	Add ACS to ACD; use 0 as base for carry bit; rotate right.
ADDZS	103320	Add ACS to ACD; use 0 as base for carry bit; swap halves of result.
AND	103400	And ACS with ACD; use Carry as carry bit.
ANDC	103460	And ACS with ACD; use complement of Carry as carry bit.

ALPHABETIC LISTING (Continued)

ANDCL	103560	And ACS with ACD; use complement of Carry as carry bit; rotate left.
ANDCR	103660	And ACS with ACD; use complement of Carry as carry bit; rotate right.
ANDCS	103760	And ACS with ACD; use complement of Carry as carry bit; swap halves of result.
ANDL	103500	And ACS with ACD; use Carry as carry bit; rotate left.
ANDO	103440	And ACS with ACD; use 1 as carry bit.
ANDOL	103540	And ACS with ACD; use 1 as carry bit; rotate left.
ANDOR	103640	And ACS with ACD; use 1 as carry bit; rotate right.
ANDOS	103740	And ACS with ACD; use 1 as carry bit; swap halves of result.
ANDR	103600	And ACS with ACD; use Carry as carry bit; rotate right.
ANDS	103770	And ACS with ACD; use Carry as carry bit; swap halves of result.
ANDZ	103420	And ACS with ACD; use 0 as carry bit.
ANDZL	103520	And ACS with ACD; use 0 as carry bit; rotate left.
ANDZR	103620	And ACS with ACD; use 0 as carry bit; rotate right.
ANDZS	103720	And ACS with ACD; use 0 as carry bit; swap halves of result.

ALPHABETIC
LISTING
(Continued)

COM	100000	Place the complement of ACS in ACD; use Carry as carry bit.
COMC	100060	Place the complement of ACS in ACD; use complement of Carry as carry bit.
COMCL	100160	Place the complement of ACS in ACD; use complement of Carry as carry bit; rotate bit.
COMCR	100260	Place the complement of ACS in ACD; use complement of Carry as carry bit; rotate right.
COMCR	100260	Place the complement of ACS in ACD; use complement of Carry as carry bit; rotate right.
COMCS	100360	Place the complement of ACS in ACD; use complement of Carry as carry bit; swap halves of result.
COML	100100	Place the complement of ACS in ACD; use Carry as carry bit; rotate left.
COMO	100040	Place the complement of ACS in ACD; use 1 as carry bit.
COMOL	100140	Place the complement of ACS in ACD; use 1 as carry bit; rotate left.
COMOR	100240	Place the complement of ACS in ACD; use 1 as carry bit; rotate right.
COMOS	100340	Place the complement of ACS in ACD; use 1 as carry bit; swap halves of result.

ALPHABETIC LISTING (Continued)

COMR	100200	Place the complement of ACS in ACD; use Carry as carry bit; rotate right.
COMS	100300	Place the complement of ACS in ACD; use Carry as carry bit; swap halves of result.
COMZ	100020	Place the complement of ACS in ACD; use 0 as carry bit.
COMZL	100120	Place the complement of ACS in ACD; use 0 as carry bit; rotate left.
COMZR	100220	Place the complement of ACS in ACD; use 0 as carry bit; rotate right.
COMZS	100320	Place the complement of ACS in ACD; use 0 as carry bit; swap halves of result.
DIA	060400	Data in, A buffer to AC.
DIAC	060600	Data in, A buffer to AC; clear device.
DIAP	060700	Data in, A buffer to AC; send special pulse to device.
DIAS	060500	Data in, A buffer to AC; start device.
DIB	061400	Data in, B buffer to AC.
DIBC	061600	Data in, B buffer to AC; clear device.
DIBP	061700	Data in, B buffer to AC; send special pulse to device.
DIBS	061500	Data in, B buffer to AC; start device.

ALPHABETIC LISTING (Continued)

DIC	062400	Data in, C buffer to AC.
DICC	062600	Data in, C buffer to AC; clear device.
DICP	062700	Data in, C buffer to AC; send special pulse to device.
DICS	062500	Data in, C buffer to Ac; start device.
DIV	073101	If overflow, set Carry. Otherwise divide AC0-AC1 by AC2. Put quotient in AC1, remainder in AC0.
DOA	061000	Data out, AC to A buffer.
DOAC	061200	Data out, AC to A buffer; clear device.
DOAP	061300	Data out, AC to A buffer; send special pulse to device.
DOAS	061100	Data out, AC to A buffer; start device.
DOB	062000	Data out, AC to B buffer.
DOBC	062200	Data out, AC to B buffer; clear device.
DOBP	062300	Data out, AC to B buffer; send special pulse to device.
DOBS	062100	Data out, AC to B buffer; start device.
DOC	063000	Data out, AC to C buffer.
DOCC	063200	Data out, AC to C buffer; clear device.
DOCP	063300	Data out, AC to C buffer; send special pulse to device.

ALPHABETIC
LISTING
(Continued)

DOCS	063100	Data out, AC to C buffer; start device.
DSZ	014000	Decrement location E by 1 and skip if result is zero.
HALT	063077	Halt the processor (= DOC 0,CPU).
INC	101400	Place ACS + 1 in ACD; use Carry as base for carry bit.
INCC	10460	Place ACS + 1 in ACD; use complement of Carry as base for carry bit.
INCCL	101560	Place ACS + 1 in ACD; use complement of Carry as base for carry bit; rotate left.
INCCR	101660	Place ACS + 1 in ACD; use complement of Carry as base for carry bit; rotate right.
INCCS	101760	Place ACS + 1 in ACD; use complement of Carry as base for carry bit; swap halves of result.
INCL	101500	Place ACS + 1 in ACD; use Carry as base for carry bit; rotate left.
INCO	101440	Place ACS + 1 in ACD; use 1 as base for carry bit.
INCOL	101540	Place ACS + 1 in ACD; use 1 as base for carry bit; rotate left.
INCOR	101640	Place ACS + 1 in ACD; use 1 as base for carry bit; rotate right.

ALPHABETIC
LISTING
(Continued)

INCOS	101740	Place ACS + 1 in ACD; use 1 as base for carry bit; swap halves of result.
INCR	101600	Place ACS + 1 in ACD; use Carry as base for carry bit; rotate right.
INCS	101700	Place ACS + 1 in ACD; use Carry as base for carry bit; swap halves of result.
INCZ	101420	Place ACS + 1 in ACD; use 0 as base for carry bit.
INCZL	101520	Place ACS + 1 in ACD; use 0 as base for carry bit; rotate left.
INCZR	101620	Place ACS + 1 in ACD; use 0 as base for carry bit; rotate right.
INCZS	101720	Place ACS + 1 in ACD; use 0 as base for carry bit; swap halves of result.
INTA	061477	Acknowledge interrupt by loading code of nearest device that is requesting an interrupt into AC bits 10-15 (=DIB-,CPU).
INTDS	060277	Disable interrupt by clearing interrupt On (= NIOC CPU).
INTEN	060177	Enable interrupt by setting Interrupt On (=NIOS CPU).
IORST	062677	Clear all I/O devices, clear Interrupt On, reset clock to line frequence (=DICC 0,CPU).
ISZ	010000	Increment location E by 1 and skip if result is zero.

ALPHABETIC LISTING (Continued)

JMP	000000	Jump to location E (put E in PC).
JSR	004000	Load PC + 1 in AC3 and subroutine at location E (put E in PC).
LDA	020000	Load contents of location E into AC.
MOV	101000	Move ACS to ACD; use Carry as carry bit.
MOVC	101060	Move ACS to ACD; use complement of Carry as carry bit.
MOVCL	101160	Move ACS to ACD; use complement of Carry as carry bit; rotate left.
MOVCR	101260	Move ACS to ACD; use complement of Carry as carry bit; rotate right.
MOVCS	101360	Move ACS to ACD; use complement of Carry as carry bit; swap halves of result.
MOVL	101100	Move ACS to ACD; use Carry as carry bit; rotate left.
MOVO	101040	Move ACS to ACD; use 1 as carry bit.
MOVOL	101140	Move ACS to ACD; use 1 as carry bit; rotate left.
MOVOR	101240	Move ACS to ACD; use 1 as carry bit; rotate right.
MOVOS	101340	Move ACS to ACD; use 1 as carry bit; swap halves of result.
MOVR	101200	Move ACS to ACD; use Carry as carry bit; rotate right.

ALPHABETIC LISTING (Continued)

MOVS	101300	Move ACS to ACD; use Carry as carry bit; swap halves of result.
MOVZ	101020	Move ACS to ACD; use 0 as carry bit.
MOVZL	101120	Move ACS to ACD; use 0 as carry bit; rotate left.
MOVZR	101220	Move ACS to ACD; use 0 as carry bit; rotate right.
MOVZS	101320	Move ACS to ACD; use 0 as carry bit; swap halves of result.
MSKO	062077	Set up Interrupt Disable flags according to mask in AC (=DOB -,CPU).
MUL	073301	Multiply AC1 by AC2, add product to AC0, put result in AC0-AC1.
NEG	100400	Place negative of ACS in ACD; use Carry as base for carry bit.
NEGC	100460	Place negative of ACS in ACD; use complement of Carry as base for carry bit.
NEGCL	100560	Place negative of ACS in ACD; use complement of Carry as base for carry bit; rotate left.
NEGCR	100660	Place negative of ACS in ACD; use complement of Carry as base for carry bit; rotate right.
NEGCS	100760	Place negative of ACS in ACD; use complement of Carry as base for carry bit; swap halves of result.

ALPHABETIC
LISTING
(Continued)

NEGL	100500	Place negative of ACS in ACD; use Carry as base for carry bit; rotate left.
NEGO	100440	Place negative of ACS in ACD; use 1 as base for carry bit.
NEGOL	100540	Place negative of ACS in ACD; use 1 as base for carry bit; rotate left.
NEGOR	100640	Place negative of ACS in ACD; use 1 as base for carry bit; rotate right.
NEGOS	100740	Place negative of ACS in ACD; use 1 as base for carry bit; swap halves of result.
NEGR	100600	Place negative of ACS in ACD; use Carry as carry bit; rotate right.
NEGS	100700	Place negative of ACS in ACD; use Carry as carry bit; swap halves of result.
NEGZ	100420	Place negative of ACS in ACD; use 0 as base for carry bit.
NEGZL	100520	Place negative of ACS in ACD; use 0 as base for carry bit; rotate left.
NEGZR	100620	Place negative of ACS in ACD; use 0 as base for carry bit; rotate right.
NEGZS	100720	Place negative of ACS in ACD; use 0 as base for carry bit; swap halves of result.
NIO	060000	No operation.
NIOC	060200	Clear device.

ALPHABETIC
LISTING
(Continued)

NIOP	060300	Send special pulse to device.
NIOS	060100	Start device.
READS	060477	Read console data switches into AC (=DIA -,CPU).
SBN	000007	Skip if both carry and result are nonzero (skip function in an arithmetic or logical instruction).
SEZ	000006	Skip if either carry or result is zero (skip function in an arithmetic or logical instruction).
SKP	000001	Skip (skip function in an arithmetic or logical instruction).
SKPBN	063400	Skip if Busy is 1.
SKPBZ	063500	Skip if Busy is 0.
SKPDN	063600	Skip if Done is 1.
SKPDZ	063700	Skip if Done is 0.
SNC	000003	Skip if carry bit is 1 (skip function in an arithmetic or logical instruction).
SNR	000005	Skip if result is nonzero (skip function in an arithmetic or logical instruction).
STA	040000	Store AC in location E.
SUB	102400	Subtract ACS from ACD; use Carry as base for carry bit.
SUBC	102460	Subtract ACS from ACD; use complement of Carry as base for carry bit.

ALPHABETIC
LISTING
(Continued)

SUBCL	102560	Subtract ACS from ACD; use complement of Carry as base for carry bit; rotate left.
SUBCR	102660	Subtract ACS from ADC; use complement of Carry as base for carry bit; rotate right.
SUBCS	102760	Subtract ACS from ACD; use complement of Carry as base for carry bit; swap halves of result.
SUBL	102500	Subtract ACS from ACD; use Carry as base for carry bit; rotate left.
SUBO	102440	Subtract ACS from ACD; use 1 as base for carry bit.
SUBOL	102540	Subtract ACS from ACD; use 1 as base for carry bit; rotate left.
SUBOR	102640	Subtract ACS from ACD; use 1 as base for carry bit; rotate right.
SUBOS	102740	Subtract ACS from ACD; use 1 as base for carry bit; swap halves of result.
SUBR	102600	Subtract ACS from ACD; use Carry as base for carry bit; rotate right.
SUBS	102700	Subtract ACS from ACD; use Carry as base for carry bit; swap halves of result.
SUBZ	102420	Subtract ACS from ACD; use 0 as base for carry bit.
SUBZL	102520	Subtract ACS from ACD; use 0 as base for carry bit; rotate left.

ALPHABETIC LISTING (Continued)

SUBZR	102620	Subtract ACS from ACD; use 0 as base for carry bit; rotate right.
SUBZS	102720	Subtract ACS from ACD; use 0 as base for carry bit; swap halves of result.
SZC	000002	Skip if carry is 0 (skip function in an arithmetic or logical instruction).
SZR	000004	Skip if result is zero (skip function in an arithmetic or logical instruction).
@	002000	When this character appears in a memory reference instruction, the assembler places a 1 in bit 5 to produce indirect addressing.
@	100000	When this character appears with a 15-bit address, the assembler places a 1 in bit 0, making the address indirect.
#	000010	Appending this character to the mnemonic for an arithmetic or logical instruction places a 1 in bit 13 to prevent the processor from loading the 17-bit result in Carry and ACD. Thus the result of an instruction can be tested for a skip without affecting Carry or the accumulators.

APPENDIX D

IN-OUT CODES

The table on the next two pages lists the in-out devices, their octal codes, mnemonics, and DG option numbers. 800 series options are for the SUPERNOVA® * only, 8100 for the NOVA® * 1200, 8200 for the NOVA 800, and 4000 series options are for all machines or the NOVA only. Codes 40 and above are used in pairs (40-41, 42-43...) for receiver-transmitter sets in the high speed communications controller.

The table beginning on page D-4 lists the complete Teletype code. The lower-case character set (codes 140-176) is not available on the Model 33 or 35, but giving one of these codes causes the teletypewriter to print the corresponding upper-case character. Other differences between the 33-35 and the 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the computer teletypewriter, and the definitions bear no necessary relation to the use of the codes in conjunction with the software.

---

*SUPERNOVA and NOVA are registered trademarks of Data General Corporation, Southboro, Massachusetts.

IN-OUT
DEVICES

Octal Code	Mnemonic	Priority Mask Bit	Device	Option Number
01	MDV		Multiply-divide	A
02	MAP0			
03	MAP1		Memory allocation and protection	8008
04	MAP2			
05				
06	MCAT	12	Multiprocessor adapter transmitter	4038
07	MCAR	12	Multiprocessor adapter receiver	
10	TTI	14	Teletype input	4010
11	TTO	15	Teletype output	
12	PTR	11	Papertape reader	4011
13	PTP	13	Papertape punch	4012
14	RTC	13	Real-time clock	4008
15	PLT	12	Incremental plotter	4017
16	CRD	10	Card reader	4016
17	LPT	12	Line printer	4018
20	DSK	9	Disc	4019
21	ADCV	8	A/D converter	4032 4033
22	MTA	10 -	Industry compatible magnetic tape	4033
23	DACV	-	D/A converter	4037
24	DCM	0	Data communications multiplexer	4026
25				
26			Other multiplexers and/or control signal options	
27				
30				
31*	IBM1	13	IBM 360 interface	4025
32	IBM2			
33				
34				
35				
36				
37				
40		8	Receiver	4015
41		8	Transmitter	
42				

A SUPERNOVA, 8007; NOVA 1200, 8107; NOVA 800, 8207; NOVA, 4031
* Code returned by INTA

IN-OUT
DEVICES
(Continued)

Octal Code	Mnemonic	Priority Mask Bit	Device	Option Number
43				
44				
45				
46				
47				
50			Second Teletype input	4010
51			Second Teletype output	
52			Second papertape reader	4011
53			Second papertape punch	4012
54				
55				
56				
57				
60			Second disc	4019
61				
62			Second magnetic tape	4030
63				
64				
65				
66				
67				
70				
71*				
72			Second IBM 360 interface	4025
73				
74				
75				
76				
77	CPU		Central processor Power monitor and auto restart	B C

---

* Code returned by INTA
B SUPERNOVA, 8001; NOVA 1200, 8101; NOVA 800, 8201; NOVA, 4001
C SUPERNOVA, 8006; NOVA 1200, 8106; NOVA 800, 8206; NOVA, 4006

TELETYPE
CODE

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed. Repeats on Model 37. Control shift P on Model 33 and 35.
1	001	SOM	Start of heading; also SOM, start of message. Control A.
1	002	STX	Start of text; also FOA, end of address. Control B.
0	003	ETX	End of text; also FOM, end of message. Control C.
1	004	EOT	End of transmission (END); shuts off TWX machines. Control D.
0	005	ENQ	Enquiry (ENORY); also WRU "who are you?" Triggers identification. ("Here is ...") at remote station if so equipped. Control E.
0	006	ACK	Acknowledge; also RU, "Are you...?" Control F.
1	007	BEL	Rings the bell. Control G.
1	010	BS	Backspace, also EEO, format effector. Backspaces some machines. Repeats on Model 37. Control II on Model 33 and 35.
0	011	HT	Horizontal tab. Control on Model 33 and 35.
0	012	LF	Line feed or line space (NRE LINE); advances paper to next line. Repeats on Model 37. Duplicated by control I on Model 33 and 35.

TELETYPE
CODE
(Continued)

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	013	VT	Vertical tab (VTAB). Control C on Model 33 and 35.
0	014	FF	Form feed to top of next page (PAGE). Control L.
1	015	CR	Carriage return to beginning of line. Control M on Model 33 and 35.
1	016	SO	Shift out; changes ribbon color to red. Control N.
0	017	SI	Shift in; changes ribbon color to black. Control O.
1	020	DLE	Data link escape. Control P (DCO).
0	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (XON).
0	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE,AUX ON).
1	023	DC3	Device control 3, turns transmitter (reader) off. Control S (XOFF).
0	024	DC4	Device control 4, turns punch or auxiliary off. Control T (AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, error. Control U.
1	026	SYN	Synchronous idle (SYNC). Control V.
0	027	ETB	End of transmission block; also MM, logical end of medium. Control W.
0	030	CAN	Cancel (CANCL). Control X.
1	031	EM	End of medium. Control Y.

TELETYPE
CODE
(Continued)

Even Parity Bit	7-Bit Octal Code	Char-acter	Remarks
1	032	SUB	Substitute. Control Z.
0	033	ESC	Escape, prefix. This code is also generated by control shift K on Model 33 and 35.
1	034	ES	File separator. Control shift L on Model 33 and 35.
0	035	GS	Group separator. Control shift M and Model 33 and 35.
0	036	RS	Record separator. Control N on Model 33 and 35.
1	037	US	Unit separator. Control shift O on Model 33 and 35.
1	040	SP	Space.
0	041	!	
0	042	"	
0	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	053	*	Repeats on Model 37.

TELETYPE
CODE
(Continued)

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	053	+	
0	054	,	
0	055	-	Repeats on Model 37.
0	056	.	Repeats on Model 37.
1	057	/	
0	060	0	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	Repeats on Model 37.
1	076	>	

TELETYPE
CODE
(Continued)

Even Parity Bit	7-Bit Octal Code	Char-acter	Remarks
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	

TELETYPE
CODE
(Continued)

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	Repeats on Model 37.
0	131	Y	
0	132	Z	
1	133	[	Shift K on Model 33 and 35.
0	134	\	Shift L on Model 33 and 35.
1	135	]	Shift M on Model 33 and 35.
1	136	↑	
0	137	←	Repeats on Model 37.
0	140	`	Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	

TELETYPE
CODE
(Continued)

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	Repeats on Model 37.
1	171	y	
1	172	z	
0	173	{	
1	174	|	

TELETYPE
CODE
(Continued)

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	175	}	On early versions of the Model 33 and 35, either of these codes may be generated by either the ALT MODE or ESC key.
0	176	~	
1	177	DEL	Delete, rub out. Repeats on Model 37.

Keys That Generate No Codes

REPT	Model 33 and 35 only: causes any other key that is struck to repeat continuously until REPT is released.
PAPER ADVANCE	Model 37 local line feed.
LOCAL RETURN	Model 37 local carriage return.
LOC LF	Model 33 and 35 local line feed.
LOC CR	Model 33 and 35 local carriage return.
INTERRUPT, BREAK	Opens the line (machine sends a continuous string of null characters).
PROCEED, BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 20-character message.

DataGeneral

Data General Corporation, 4400 Computer Drive, Westboro, MA 01580
(617) 366-8911