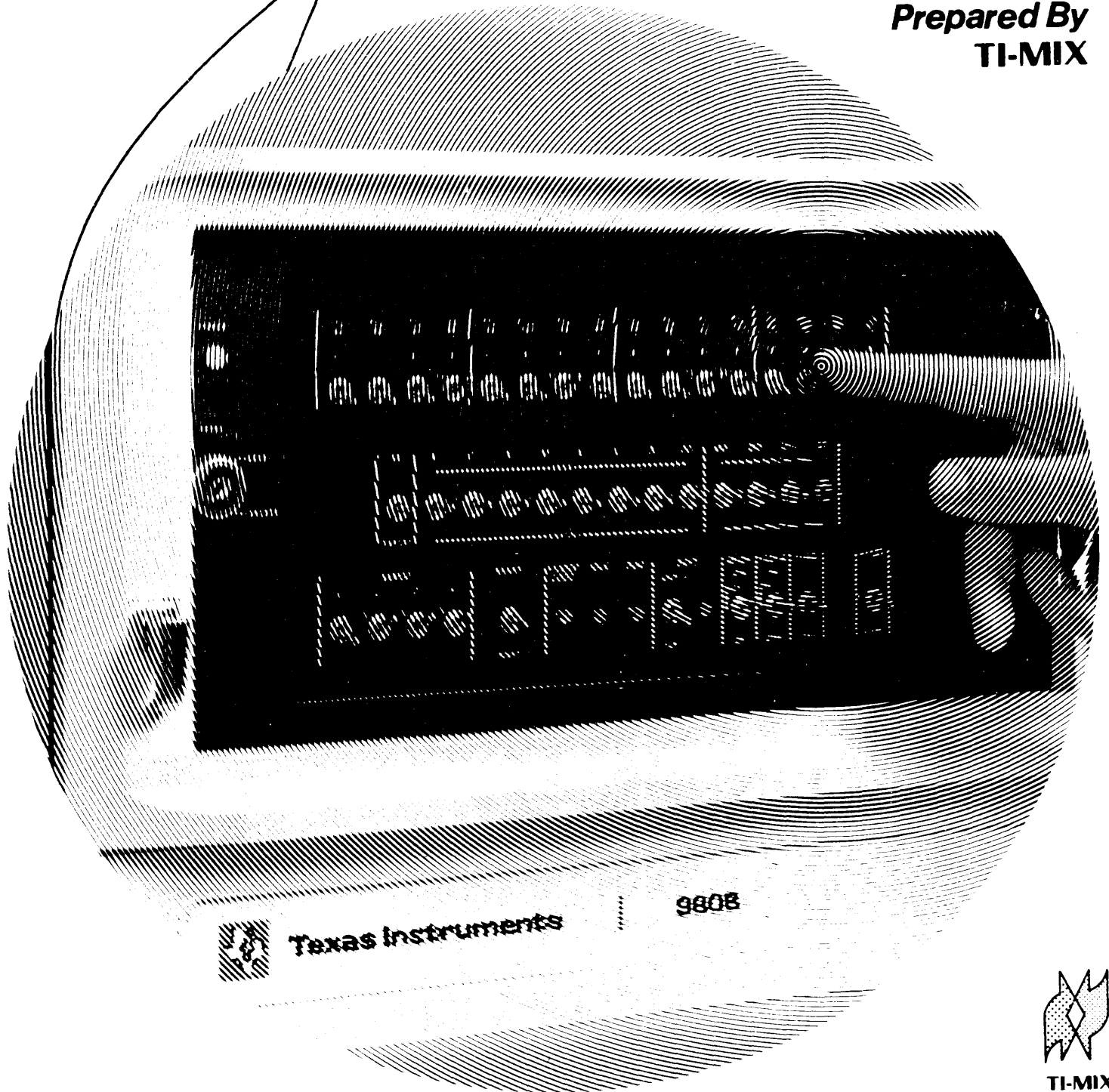


An Introduction to the 980 Series Minicomputer

Prepared By
TI-MIX



Texas Instruments

9808



TI-MIX

To the Reader —

When I first became involved with the TI 980 a few months ago, I decided to write a collection of notes and code samples for my own use. The project eventually evolved into this manual which I hope will be of use to new programmers. The pressures of time and other work caused the effort to be cut short, so many of the code samples have not been tested, and certain statements may be in error. I would sincerely appreciate it if those of you who find errors would assist me in correcting them by responding on the tear-out sheet which is bound in this publication.

I acknowledge my indebtedness to you in advance in addition to those who have already contributed a great deal. In particular, I would like to thank Jerry Junkins of the Equipment Group for supporting the project, Floyd Burton of TI-MIX for publishing the manual, Wilburn Jones of the Services Group for technical illustrations, Jim Simpfenderfer and Jon Jentink of the Equipment Group for reading sections of the manuscript, and Fred Wedemeier of the Equipment Group for answering my incessant questions.

May our efforts be useful to you, the reader, in some way.

S.N.S.

*To Fred -
With thanks for
all your help.
Sue*

About the Author —

Sue Nickerson Stidham has been involved with many aspects of computer technology. She holds a Ph.D. in physical chemistry from the University of Massachusetts (Amherst) and an A.B. in chemistry and physics from Smith College. She was assistant professor of Computer Science at the University of Massachusetts, an instructor in physics at the University at Bridgeport, Connecticut and at Mount Holyoke College. She held a Petroleum Research Fund Fellowship for 2 years.

Dr. Stidham was a National Science Foundation stipend recipient in 1963 at the Winter Institute in quantum chemistry and solid-state physics, Gainesville, Florida, and in 1970 at CASSAD, (computer-assisted system simulation and design), Houston, Texas. She joined Texas Instruments in 1973.

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION TO ASSEMBLY LANGUAGE	
	1-1 Assembly Language Programming: What It Is And Why Learn It	1
	1-2 Assembly Language vs. Machine Language	3
	1-3 Organization of a Computer System	4
	1-4 Octal and Hexadecimal Number Systems	6
	1-5 Hexadecimal Addressing Scheme	7
	1-6 Memory Organization and the Program Counter	8
	1-7 Pseudo-Operations and Assembly	9
	1-8 The Role of the Accumulator in Execution	11
	1-9 Assembling, Loading and Executing the Program	11
	1-10 Finding the Answer	12
2	MACHINE WORDS AND THE BINARY NUMBER SYSTEM	
	2-1 What's in a Word	13
	2-2 Number System Conversion Between Binary and Hexadecimal	14
	2-3 Conversion to and from Decimal	15
	2-3.1 Hexadecimal to Decimal	15
	2-3.2 Decimal to Hexadecimal	15
	2-3.3 Conversion of Fractions	16
	2-4 Number Conversion to and from Octal	17
	2-4.1 Number Conversion Between Binary and Octal	17
	2-4.2 Conversions Between Octal and Hexadecimal	17
	2-4.3 Octal to Decimal	18
	2-4.4 Decimal to Octal	18
	2-5 Conversions Between Binary and Decimal	19
	2-6 Positive and Negative Data Values	19
	2-6.1 The Sign Bit	19
	2-6.2 Negative Numbers	19
	2-6.3 Two's Complement Arithmetic	21
3	INTRODUCTION TO 980 ASSEMBLY LANGUAGE	
	3-1 Elementary Programming Considerations	23
	3-2 The Instruction Format Skeleton	24
	3-3 Data Declarations	25
	3-4 Two Addressing Modes, P-Relative and Immediate	26
	3-5 Transfer of Control: Branch and Skip	28
	3-6 The Assembly Process and Some Assembler Directives	28
	3-7 Register Organization and Definition	30
	3-8 Notation Used in Describing Instructions	31
	3-9 A Basic Group of Instructions	32
	3-9.1 Idle Instruction (IDL)	32

TABLE OF CONTENTS (Continued)

Section	Page
3-9.2 Register-To-Memory and Memory-To-Register Transfers	32
3-9.3 Register-To-Register Instructions	33
3-9.4 Unconditional Branch Instruction	34
3-9.5 Arithmetic Instructions	34
3-9.6 A Sample Program	35
3-9.7 IMO and DMT Instructions	36
3-10 Register Skips and Indexed Branch	37
3-10.1 Register Skips	37
3-10.2 Loops, Counters, and Indexed Branch	40
3-11 Sample Program	42
3-12 The Index Register and its Use	42
3-12.1 Handling an Array: A Typical Problem	42
3-12.2 The Index Register (X)	44
3-12.3 Setting up the Data for an Array Problem	44
3-12.3.1 Incrementing Index Techniques	44
3-12.3.2 Decrementing Index Technique	46
3-12.4 Address Arithmetic	46
3-13 The Bare Machine vs. the Basic Operating System	47
3-14 Use of the Basic Operating System	48
4 TI MODEL 980 ADDRESSING MODES AND THE STATUS REGISTER	
4-1 Summary of Model 980 Characteristics	51
4-2 Effective Address	52
4-3 Address Modes: Immediate; Indirect and Indexed P-Relative	53
4-4 Normal and Extended Format	54
4-5 The Base* Register	56
4-5.1 Base Register Relative Assembly and Execution	56
4-5.2 Base Register Used at Execution Time Only	57
4-6 The Origin Directive	58
4-6.1 Reserving Storage	58
4-6.2 Data Declaration	59
4-6.3 Equate	61
4-6.4 Print Control Directives	61
4-7 The Status Register: Overflow, Carry, and Compare	61
4-7.1 Carry and Overflow	62
4-7.2 Program to Test Carry and Overflow Behavior	63
4-7.3 The Overflow and Carry Tests	65
4-7.4 The Compare Instructions	65
4-7.5 Testing the Compare Indicators	66

TABLE OF CONTENTS (Continued)

Section	Page
5	REGISTER SHIFT AND DOUBLE-LENGTH
5-1	Single Register Shift Operation Format 67
5-2	Circular Shifts 67
5-3	Arithmetic Shifts 68
5-3.1	Arithmetic Right Shift 68
5-3.2	Arithmetic Left-Shift 69
5-4	Logical Shifts 69
5-5	Double Length Register Instructions 69
5-5.1	Multiply and Divide Instructions 69
5-5.2	The Moving Average Problem 72
5-5.3	Shift Operations in a 32-Bit Register 75
5-5.4	Double-Length Shift in Preparation for Divide 76
5-5.5	Division by 2 or any of its Powers 76
5-6	Double Precision Arithmetic 77
5-7	Execution Times and the Space-Versus-Time Tradeoff 78
6	ARRAY TECHNIQUES: SORTS, SEARCHES, AND STACKS
6-1	Array Manipulation Through Indexing 83
6-2	Indexed B-Relative Mode 84
6-3	Variable-Length Arrays 84
6-4	The Exchange Sort (Bubble Sort) Technique 86
6-5	Search Techniques 89
6-6	Sequential Search 91
6-7	Binary Search 92
6-8	Searches Using Hash Technique 93
6-9	Insertion and Deletion in a List 93
6-10	The Pushdown Stack 94
6-11	Indirect Addressing 99
6-12	Summary: Three Methods of Array Handling 100
6-12.1	Indexing Method 101
6-12.2	Self-Modifying Code ("Impure Procedure") 102
7	ASCII DATA, BUFFERS, AND I/O SERVICE CALLS
7-1	Introduction to Character Strings 105
7-2	Character Strings in 980AL: The ASCII Data Declaration 108
7-3	Non-Printing Characters 108
7-4	Buffers 109
7-5	Bytes and Byte Manipulation 110
7-5.1	Move Character String (MVC) 110
7-5.2	Generate Byte Address: The Byte Declaration 112
7-5.3	Compare Logical Characters (CLC) 114

TABLE OF CONTENTS (Continued)

Section	Page
7-6 Conversion From Internal to External Format: Integers	115
7-7 Hash Totals	118
7-8 I/O Service Calls	121
7-8.1 Service Calls vs. The Bare Machine	121
7-8.2 I/O and Program Termination Using Supervisor Service Calls	122
7-8.3 Definition of Service Calls With OPD	123
7-8.4 Program Skeleton: I/O and Program Termination Using Supervisor Calls	124
8 SUBROUTINES	
8-1 The Subroutine: A Labor-Saving Device	127
8-2 The Primitive Subroutine Linkage Problem	129
8-3 Parameter Passing to a Primitive Subroutine	130
8-3.1 Parameter Passing via Registers (Call by Value)	130
8-3.2 In-Line Parameter List (Call by Address)	131
8-3.2.1 Fixed Number of List Entries	131
8-3.2.2 Variable Number of List Entries	132
8-3.3 In-Line Address of Parameter List	133
8-4 Formal Structure of a Subroutine	133
8-5 Entry Points and External Symbols: DEF and REF	134
8-6 Parameter Passing and Common Storage	135
8-6.1 Common Storage	135
8-6.2 Via Registers (Call by Value)	136
8-6.3 In-Line Parameter Lists (Call by Address)	136
8-6.4 Call by Name	136
8-7 Recursive Calls to a Primitive Subroutine	138
8-7.1 Recursion Using the X-Register	139
8-7.2 Recursion Using Indirect Addressing	139
8-7.3 Example of a Recursive Subroutine	140
9 LOGICAL OPERATIONS, BIT MANIPULATION, MASKS, AND FLAGS	
9-1 Truth Tables	143
9-2 Logical Operations	144
9-2.1 Memory-To-Register Logical Operations	144
9-2.2 Register-To-Register Operations	144
9-2.2.1 Complementing Operations	146
9-3 Bit Operations	146
9-4 Masking: One Practical Use of Logical Operations	147
9-5 The Search for a Byte String Delimiter	149
9-6 Tests for Ones and Zeros in the Accumulator	151

TABLE OF CONTENTS (Continued)

Section	Page
10	INPUT/OUTPUT ON THE BARE MACHINE
10-1	Instructions 153
10-2	Two I/O Techniques 153
10-2.1	I/O by Polling 153
10-2.2	I/O by Interrupt 154
10-2.3	Double Buffering 154
10-3	Internal and External Addresses 154
10-4	Read or Write (Low-Speed Data Bus Device) 156
10-5	External Devices: Data, Status, Command Words 158
10-6	Card Reader 158
10-6.1	Reading a Card by Polling 158
10-6.2	Polling the Reader with Autoincrement 159
10-7	High-Speed Paper Tape Reader 160
10-8	High-Speed Paper Tape Punch 161
10-9	733 ASR/KSR Data Terminal 162
10-9.1	RDS Data Word 162
10-9.2	WDS Data Word 163
10-9.3	733 ASR Subcommands 164
10-9.4	Programming Examples 166
10-9.4.1	Write/Read KSR Example 167
10-9.4.2	ASR Subcommand Example 168
10-9.4.3	Write ASR Example 168
10-9.4.4	Read ASR Example 169
10-10	DMAC I/O: The ATI Instruction 170
10-11	Single DMAC Device: Line Printer 170
11	THE INTERRUPT SYSTEM
11-1	Introduction to the Interrupt System 173
11-2	Internal Interrupts 175
11-3	Loading the Status Register 175
11-3.1	The LSB Instruction 175
11-3.2	Register or Instruction 176
11-4	Handling Internal Interrupts 176
11-5	Startup After Power Failure 179
11-6	Memory Protect/Privileged Instruction Feature (MP/PIF) 180
11-6.1	Memory Protect 180
11-6.2	Privileged Instructions 181
11-7	Program Relocation Feature 182
11-8	Data Bus and DMAC Interrupts 182
11-9	DMAC Interrupt 182
11-9.1	Data Bus Interrupt 183

TABLE OF CONTENTS (Continued)

Section	Page
11-9.2 Competing Data Bus Devices	185
11-9.3 Vectored Priority Interrupt Option	185

APPENDIX

A	980AL EXECUTION TIMES
B	980 REGISTER DESIGNATIONS
C	HEXADECIMAL ARITHMETIC
D	STANDARD ADDRESS OF EXTERNAL REGISTERS
E	SPECIALIZED LOW-ORDER MEMORY LOCATIONS
F	BASIC SYSTEM MEMORY MAP
G	SAP ERROR MESSAGES
H	980 OPERATING PROCEDURE
I	ASCII CHARACTERS BY NUMERICAL SEQUENCE
J	OPERATION CODES – NUMERICAL ORDER REGISTER-MEMORY INSTRUCTIONS
K	LOGICAL UNIT INPUT/OUTPUT FUNCTIONS

LIST OF ILLUSTRATIONS

Figure No.		Page
1-1	Major Steps in Writing a Program	2
1-2	Information Flow in a Simple Computer System	4
1-3	Information Flow in a Computer System Equipped with DMAC	4
1-4	Typical Memory Organization for a Simple Program	9
3-1	Adding a Number to Itself (Multiplication) Flowchart	37
4-1	Carry and Overflow Test Program	63
6-1	Exchange Sort Technique	87
10-1	Input/Output by Polling	153
10-2	Four-Port Data Bus and 15 – Port Bus Expander Block Diagram	155
10-3	Data Bus/Device Interfaces, Showing External Registers	155
10-4	RDS Data Word	162
10-5	WDS Data Word	163
10-6	Status Character Bits	165
11-1	The Status Register	174

LIST OF TABLES

Table No.		Page
1-1	Octal, Decimal, and Hexadecimal Number Systems	6
2-1	Comparison of Decimal, Octal, Binary, and Hexadecimal Number Systems	13
2-2	Integral Powers of 16	15
2-3	Negative Integral Powers of 16	17
3-1	Elementary Language	23
4-1	980 AL Subset	52
5-1	Register Shift and Double-Length Instructions	67
5-2	Instruction Execution Times in Microseconds	79
7-1	Untitled	105
7-2	Selected ASCII Characters	107
8-1	Untitled	127
9-1	Untitled	144
10-1	Untitled	153
10-2	Remote Device Control Functions	165
11-1	Interrupt Instructions	173
11-2	Load/Store Status Block (LSB/SSB)	178

SECTION 1

INTRODUCTION TO ASSEMBLY LANGUAGE

1-1 ASSEMBLY LANGUAGE PROGRAMMING: WHAT IS IT AND WHY LEARN IT?

Writing a program, whether in assembly language or a higher level language, actually begins when the programmer devises some kind of abstract process (called an *algorithm*) to accomplish a computational task.

One realization of the abstract algorithm is a *program*, which is an ordered sequence of steps (instructions) which tell a machine to do a given job. The nature of these instructions depends on the type of language being used. Algebraic languages such as FORTRAN permit the user to transmit via a single statement a request for a collection of machine activities; whereas use of assembly language requires the programmer to write a separate instruction for each machine activity. Compare the following program fragments: a FORTRAN statement with its four-instruction counterpart in 980 series Assembly Language (980 AL).

<u>FORTRAN</u>	<u>980 AL</u>
...	...
X=R+S-T	LDA R
...	ADD S
	SUB T
	STA X
	...

The number of assembly language instructions seen above might lead us to suspect that coding a program in 980 AL will result in more work than doing the same job in FORTRAN: in general, that is correct. However, since we have to pay such meticulous attention to detail, we will be in a position to exercise tighter control over the entire process. The real programming world tends to choose a language more or less according to the following rule of thumb:

If the primary objective of the computation is to obtain the numerical result of a set of arithmetic operations, the appropriate choice is probably an algebraic language such as FORTRAN. On the other hand, if the computer is to be used to monitor or control some physical process (for example, a chemical plant or a time-sharing system), a strong case can be made for the use of assembly language. For many types of problems, though, the choice may not be clear cut.

Another realization of the algorithm is a graphic representation (called a *flowchart*) of the logic underlying the process. Ideally, a flowchart is no more than the logical essence of the process; therefore, it is completely independent of (i.e. contains no characteristics of) any programming language. One should be able to use the same flowchart to code equivalent programs in numerous languages (see Figure 1-1).

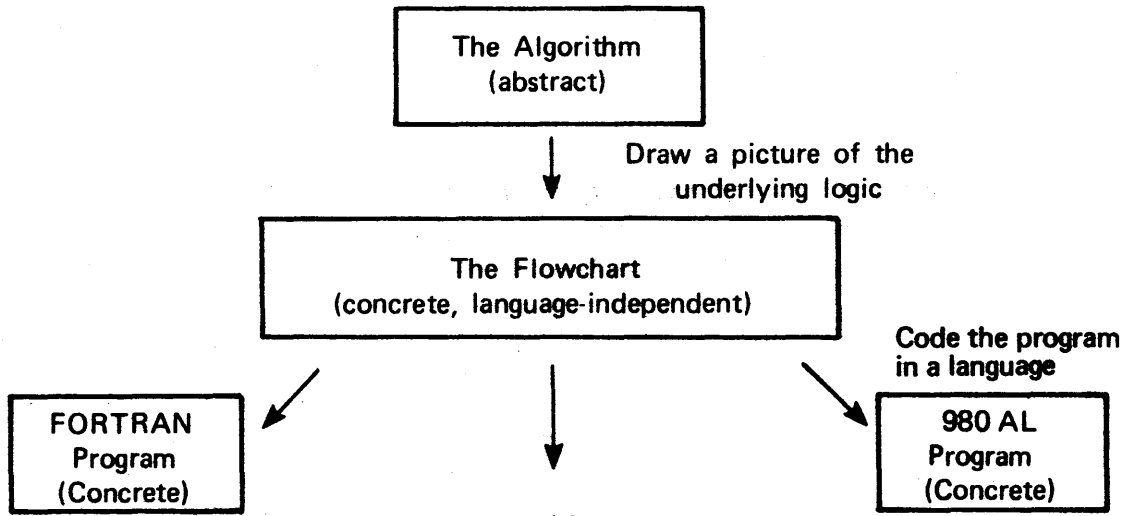


Figure 1-1. Major Steps in Writing a Program

What we've discussed is an ideal situation: it's what we would *like* to have. In practice it works a little differently.

From the very inception of the problem, the programmer will start to incorporate a language-oriented bias — but hopefully as little as possible at each step of the way. A decision should be made very early whether the job should be done in a higher level language (such as FORTRAN) or in an assembly language. In real-world programming some awareness of the advantages and limitations of the language of intended use will creep into the development of the problem solution — often as early as the abstract or algorithmic stage. While this state of affairs cannot be avoided completely, it is to the programmer's advantage to avoid as much language dependence as possible until he comes to the last stage: writing the program.

Why computer people attach so much importance to flowcharts is often puzzling to the novice programmer. The reasons for composing a logical picture of the process may be summarized as follows.

- A flowchart helps the programmer write a better program and helps him do it more efficiently.
- A flowchart helps other people who may have to understand the program, or interface it with other programs, or write the same program for other machines.
- A flowchart makes possible the design and construction of large computer systems.

Knowledge of assembly language is a *must* for computer systems programmers. For others, concerned mostly with user applications programming, the study of assembly language may prove useful for a number of reasons; three of which come to mind easily:

1. With the spread of minicomputers and the increasing amount of automation of laboratory apparatus in fields from physics to psychology, many people are discovering a need for assembly language programming ability.

2. Many users of higher level languages find that a knowledge of assembly language programming is useful in debugging programs when the language diagnostics fail to adequately indicate the source of error.
3. Most importantly, assembly language programming reflects very accurately the machine's architecture and operation — thus it contributes heavily to a person's general understanding of how computers really work.

1-2 ASSEMBLY LANGUAGE vs. MACHINE LANGUAGE.

When a computer is first designed and built, it is provided a rudimentary language in which the user may talk to it; this is the so-called "machine language" associated with that particular model. The machine language is a numerical code to indicate to the machine which of its circuits should be activated in order to accomplish the operation desired by the programmer. The program fragment on page 1 might look something like the following when expressed in machine language:

<u>Assembly language form</u>	<u>Machine language form</u>
...	...
LDA R	0004
ADD S	2006
SUB T	28F5
STA X	80F7
...	...

As we will discuss later, the second line uses 20 as the machine language equivalent of an *add* operation, and the quantity known as S is to be found in some location associated with the number 06.

Once upon a time, all programming was done in machine language, and the programmer had to keep referring to code-tables and memory maps in order to avoid burying himself in tons of numerical garbage. The tedium of this constant look-up process led to the development of assembly language, in which the programmer gave the machine a mnemonic operation code (op-code) such as ADD, and the machine used a special translator program (known as an *assembler*) to look up the corresponding numerical code. (The translation process became known as the *assembly*.) As computers evolved, the assemblers were given more and more of the routine bookkeeping jobs necessary to writing a program. Ultimately, the assemblers were given additional responsibilities, such as selecting and printing error messages designed to give an unlucky programmer some clue why his program failed to assemble (i.e. why the translation process did not work). In general, most current machines come equipped with more sophisticated assemblers; although a few, small, special-purpose computers still must be programmed in machine language.

What we as assembly language programmers will do is to enter our programs into the computer in assembly language and then instruct the computer to translate (or *assemble*) the instructions into machine code.* The time during which this translation process occurs is called *assembly time*. Just translating the program into machine code is not enough: the machine code program then must be loaded into the computer to run (*execute*). The period during which the program runs is called *execution time*.

*Just to add to the confusion of the newcomer, computer people are sometimes ambiguous in their use of the term *machine language*; it usually refers to the machine code discussed above, but it is sometimes used to refer to the assembly language, which is really only one small step away.

1-3 ORGANIZATION OF A COMPUTER SYSTEM.

The two parts essential to any computer system are a *central processing unit* (CPU) which handles arithmetic and control functions and a *memory*. In addition, some means must be available for the operator/programmer to communicate with the machine. In a very crude system, communication can occur through a set of switches and lights on the front panel of the CPU; however, usually at least one input/output (I/O) device is present. The most economical device is often a *terminal* which is actually two devices in one; the keyboard serves as the input device and the printing mechanism serves as the output device. Characters entered through the keyboard are sent as a pulse train to the CPU, which (if the program has such a provision) may be sent back ("echoed") to the printing mechanism with sufficient speed so that the terminal resembles a typewriter in its operation.

Actually, a number of I/O devices may be attached to the computer: a card reader or paper tape reader (or both) for input; and a line printer, card punch, or paper tape punch for output. Other devices like magnetic tape (reel or cassette) may serve for both input and output.

Figure 1-2 is a crude representation of the information flow.

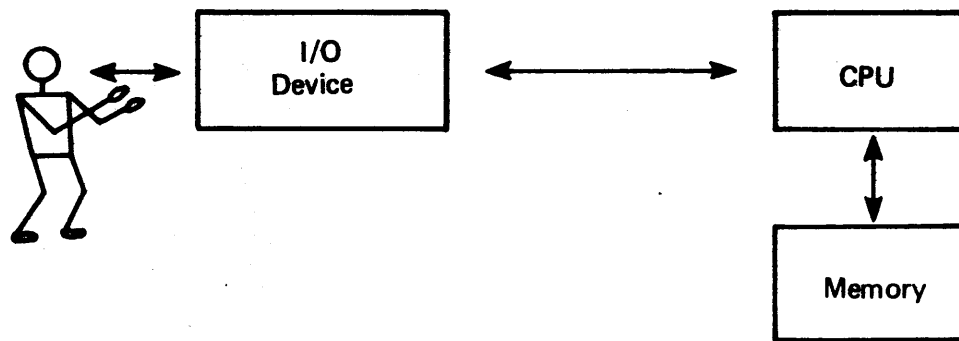


Figure 1-2. Information flow in a Simple Computer System.

A number of computers, like the TI980 series, also have provisions for direct access by an I/O device to the memory using a direct memory access channel (DMAC) as shown in Figure 1-3.

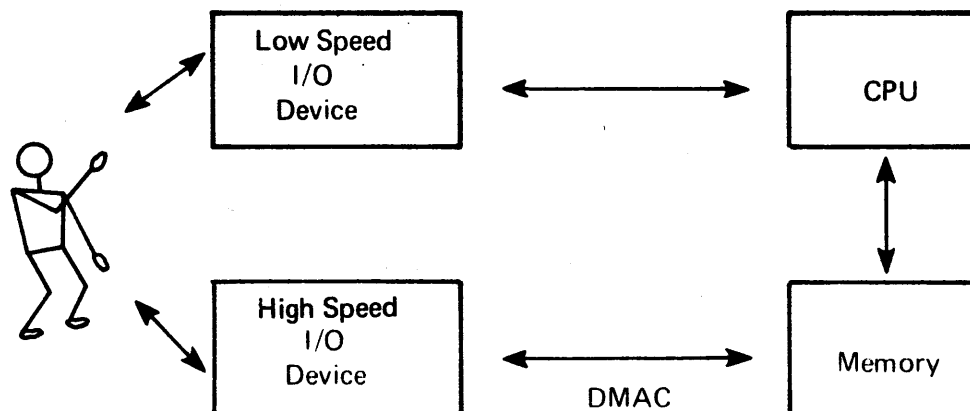
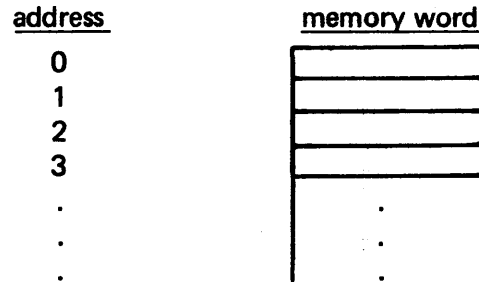


Figure 1-3. Information Flow in a Computer System equipped with DMAC

For now we will ignore the DMAC capabilities of the 980 and consider the system as shown in Figure 1-2.

The memory, simply speaking, consists of an array of pigeon holes in which items of information can be stored. In more formal parlance, these pigeon holes are called *words* or *locations*. For the machine's convenience in referencing a given location, the entire set is numbered, and the counting numbers associated with these locations are called *addresses*. We could conceive of the memory as organized as follows:



Not all computers are supplied with the same size memory; the amount varies, depending on the power resources and compactness of the chassis. Usually the smallest amount supplied is 4096 (or "4k") words. Depending upon the manufacturer of the unit, memory is commonly provided in incremental units of 4k or 8k. The most commonly occurring configurations, in addition to 4k are –

4k	(4096 words)
8k	(8192 words)
16k	(16384 words)
24k	(24576 words)
32k	(32768 words)
64k	(65536 words)

The limit to the amount of memory which can be attached to a computer is usually dictated by the size (number of bits) of a machine word, since a word ultimately must have enough bits to contain a complete address.

Each memory location has an address and is capable of containing either a machine language instruction or a data number. One complexity confronting us in dealing with the 980 computer (and with many other machines, as well) is that we often find ourselves having to use the base 16 (or *hexadecimal*) number system. The reason for the hexadecimal number system will become apparent later when we examine the detailed bit (bit = binary digit) structure of a computer word.

The 980 Computer has other electronic pigeon holes that look quite a bit like memory words except:

- they are physically located in the CPU rather than in the memory
- they are made of expensive high-speed circuitry
- they have names instead of hexadecimal addresses.

Such high-speed pigeon holes are called *registers*, and are used for special purposes which are discussed as each register is introduced:

- accumulator
- program counter (PC or P-register)
- extension arithmetic (E-register)
- index register (X-register)
- base register (B-register)
- link register (L-register)
- maintenance register (M-register)
- storage register (S-register)
- status register (ST).

1-4 OCTAL AND HEXADECIMAL NUMBER SYSTEMS.

Most assembly languages use either an octal (base 8) or a hexadecimal (base 16) number system to represent instructions and data contained internally. The TI980, as well as a number of other machines, mainly uses the hexadecimal system. However, the octal system is in common use, and occasional octal numbers are encountered in programming the 980.

At first inspection, octal numbers look very similar to the familiar decimal numbers except for the fact that the digits 8 and 9 never appear. The concepts *eight* and *nine* do exist, however, and somehow have to be represented. In contrast, hexadecimal numbers have not only 8 and 9, but six more digits as well. (Rather than invent new digit symbols for the six additional, we use the letters A through F.)

In the decimal number system when we wish to count, we write down each of the decimal digits in order 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; at which point we have used up all the single digits and need to have some way of expressing the concept *ten*. We do this by starting over again with the first digit (0) and inscribe a 1 to its left, giving us 10. Then we can run through all our digits again until reaching 19, when we once again run out of digits. We start over again with 0 as the number on the left takes on its next value in the ordered digit string (i.e., 2), giving us 20. The process continues until we run out of digits in both positions at the number 99; whereupon we start both positions over again at zero and inscribe a "1" to the left, giving us 100.

To build up numbers in, for example, the octal system, we do the very same thing except we never have the digits 8 or 9 to work with. Table 1-1 shows what happens:

Table 1-1. Octal, Decimal, and Hexadecimal Number Systems

<u>(decimal)</u> <u>concept</u>	<u>octal</u> <u>representation</u>	<u>decimal</u> <u>representation</u>	<u>hexadecimal</u> <u>representation</u>
zero	0	0	0
one	1	1	1
two	2	2	2
three	3	3	3
four	4	4	4
five	5	5	5
six	6	6	6
seven	7	7	7
eight	10	8	8

Table 1-1. Octal, Decimal, and Hexadecimal Number Systems (continued)

<u>(decimal) concept</u>	<u>octal representation</u>	<u>decimal representation</u>	<u>hexadecimal representation</u>
nine	11	9	9
ten	12	10	A
eleven	13	11	B
twelve	14	12	C
thirteen	15	13	D
fourteen	16	14	E
fifteen	17	15	F
sixteen	20	16	10
seventeen	21	17	11
eighteen	22	18	12
nineteen	23	19	13
twenty	24	20	14
twenty-one	25	21	15
twenty-two	26	22	16
twenty-three	27	23	17
twenty-four	30	24	18
twenty-five	31	25	19
twenty-six	32	26	1A
.	.	.	.
.	.	.	.
.	.	.	.
four thousand			
ninety-five	7777	4095	FFF
four thousand			
ninety-six	10000	4096	1000

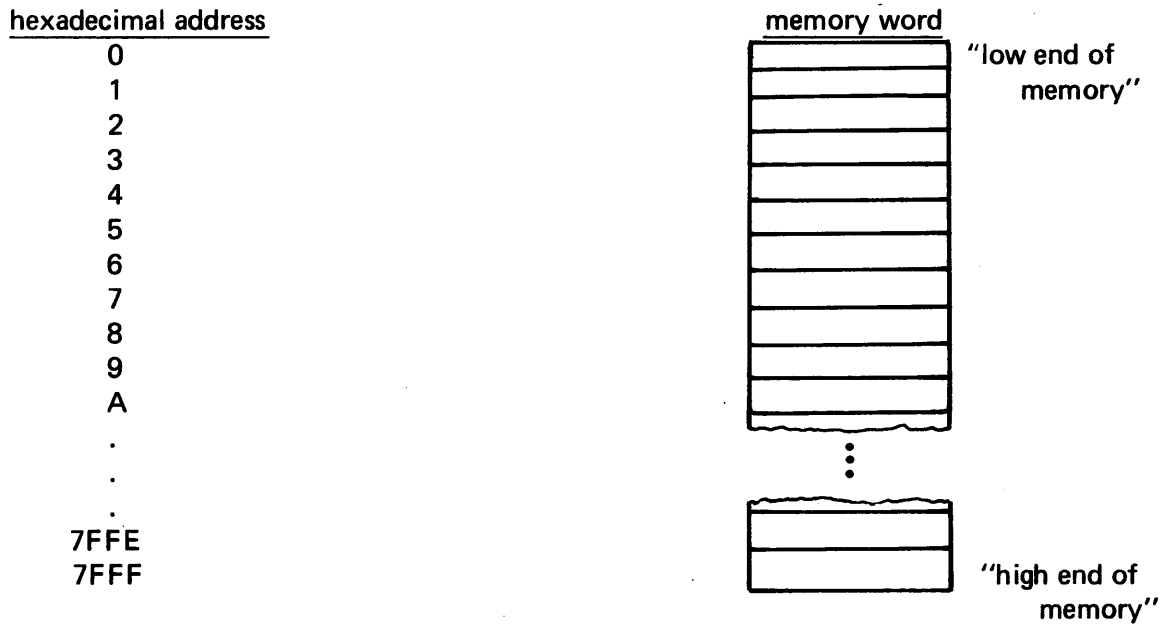
If we examine a number like 4096, we see immediately that it cannot be an octal number because it contains the digit 9; however, if we examine a number like 100, we have no way of knowing to what number base it refers unless we are told explicitly or can decide from the context in which the number appears. One way of indicating which number system is in use is to place a small subscript next to the number. Thus 765₁₀ (or just 765) is seven hundred sixty-five represented in the familiar decimal system; while 765₈ is a representation of some-number-or-other in the octal system and 765₁₆ is yet a different number in the hexadecimal system. Fortunately, a straightforward technique for converting numbers back and forth between number systems is available so we don't have to learn each octal or hexadecimal representation as a special case. But for the time being, let's use numbers in our examples that are sufficiently small that we can determine the hexadecimal representation simply by looking at Table 1-1.

1.5 HEXADECIMAL ADDRESSING SCHEME.

The TI 980 computer uses the hexadecimal number system for internal representations of:

- the machine-language contents of all the memory words
- most of the numbers appearing in assembly language instructions
- the address of all the memory words.

Referring to item c. above, we can again draw a schematic diagram of the memory, this time showing how it *really* looks to the user:



We are assuming the 32k words (actually 32768) are addressed from 0 (which is the same as 0000) to 32767₁₀ (or 7FFF₁₆). The significance of the addressing scheme is this: If we want to examine the contents of the word following location 9, we have to refer to it as location "A" rather than "10". If we want to put some information in the word following location 7FFF, we're out of luck because on a 32k system, there are no more pigeon holes beyond the one at 7FFF.

1-6 MEMORY ORGANIZATION AND THE PROGRAM COUNTER.

Before the computer can be let loose to grind out the problem solution, certain information must appear in the memory locations. This information consists of:

- a. a collection of instructions (i.e., a *program*) for manipulation (or generation) of data (e.g., *add*, *subtract*, etc.)
- b. the data numbers themselves.

Once this information is available, the computer can be told to execute the instructions; and the instructions (if the programmer has written them correctly) will operate on the data.

For now we will consider only those cases in which each instruction of the program and each data number occupies a single word of memory. (Later, in more sophisticated problems these restrictions are relaxed.)

When we instruct the machine to execute the program, we want it to start in the first location containing a program instruction (the *entry point*) and perform all the instructions up to and including some kind of halt instruction. The halt instruction is the last instruction in the program; it keeps the execution from trying to proceed past the locations containing viable program instructions.

Within the machine is a register known as the *program counter*, often called the PC (or P register), which always contains the address of the next instruction the machine is supposed to

execute. If the first program instruction is in location 12F7, *the PC will somehow have to have a 12F7 loaded into it at the beginning of execution. After that, the computer itself will undertake to keep the value of the PC updated. The PC could be visualized as a sliding arrow (see Figure 1.4) that always points to the next instruction to be executed, but in reality it is no such thing. The PC is a register that stays in a fixed position in the CPU but does its "pointing" by holding the address of the next memory location to be executed.

It is reasonable to ask at this point, "if the halt instruction separates the program from the block of data values, how does the PC ever get to point to the data?" The answer is: "during normal program execution, it doesn't."

When the machine needs a number which is stored in the data block, it has the happy capability to "look ahead" and acquire a number out of the data block without ever needing the PC to point to the datum in question. Although the job of updating the contents of the PC is handled by the computer most of the time, there are occasions when we may wish to insert an address of our own choosing: why and how we'll do this are discussed later.

1-7 PSEUDO-OPERATIONS AND ASSEMBLY.

In an earlier discussion, we saw one possible layout of the program instructions and the data values within the computer memory. We, as programmers, can exercise control over which instructions appear in which memory locations by giving the assembler a special assembly-time directive (or *pseudo-op*) with an op-code of ORG (short for *origin*).

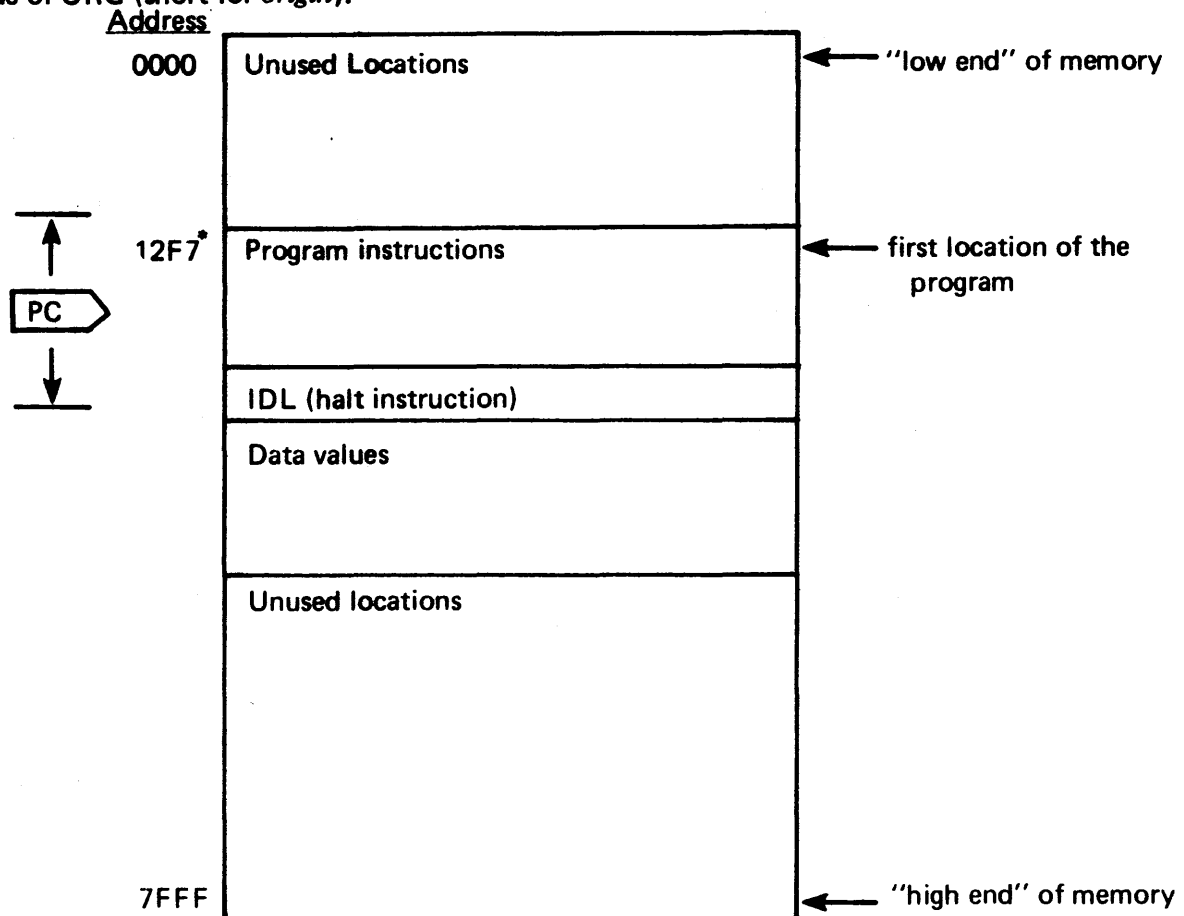


Figure 1-4. Typical Memory Organization For a Simple Program

*The selection of 12F7 is arbitrary

Every time we include within the program an instruction of the form

ORG <address>

the very next instruction the machine finds will be placed in the location specified by <address>. For example, let's look at the program fragment we discussed earlier and convert it into a complete assembly language program which will direct the first instruction to be assembled into location 12F7 and which will place the data numbers 7, 10, and 14 in locations 1300, 1301, and 1302, respectively. Note that the ORG pseudo-op does its job at assembly time; and since it is not needed at execution time, it never appears in the machine-language memory listing.

Assembly-time		Assembly	Execution time
ORG >	12F7	→	01000500
LDA	R		03000501
ADD	S		04000502
SUB	T		
			12F7
			12F8
			12F9

We will use the data declaration (DATA) both to label the addresses with the names R, S, T, and X and to place into those locations the appropriate hexadecimal values for 7, 10, and 14. We can either declare these values to be hexadecimal numbers (by prefixing the number with the symbol >) or we can write them as decimal numbers and let the machine perform the conversion for us.

	Assembly-time*		Execution time
	IDT	SAMPLE	
	ORG	> 12F7	
SAMPLE	LDA	R	0008 12F7
	ADD	S	2008 12F8
	SUB	T	2808 12F9
	STA	X	8008 12FA
	IDL		CE00 12FB
			12FC
			12FD
	ORG	> 1300	12FE
			12FF
R	DATA	7	0007 1300
S	DATA	10	000A 1301
T	DATA	14	000E 1302
X	DATA	0	0000 1303
	END	SAMPLE	

IDT is a pseudo-op which gives the program a name. END is a pseudo-op used to inform the assembler there are no more instructions to be translated. Since END is not part of the executable instructions, it never appears in the machine code listing.

The first two digits of an instruction represent the op-code, and the last two represent the displacement of the location in which the operand (the data value, in this example) is to be found. The machine finds the appropriate operand during execution by adding 08 (displacement) to the current value of the PC. The result is the address where the appropriate number will be found.

*Note that the alphabet letter O is often differentiated from zero by writing a slash through the zero (0).

The fact that each instruction has the same displacement (08) arises from the fact that the values are used in consecutive locations and declared in the same order in a block of consecutive locations which starts eight addresses below the instruction block. (The data address increases by one for each successive location; but then, so does the PC.) The reason the displacement is 08 instead of 09 is that the PC automatically increments by one (i.e., points to the next instruction) before the operand address has been calculated. Thus, we can always think of the PC as being one address higher than the instruction being executed.

1-8 THE ROLE OF THE ACCUMULATOR IN EXECUTION.

Once the program has been translated and properly loaded into the memory, it can be executed (i.e., the instructions can be carried out). All arithmetic operations in 980 AL, and many nonarithmetic operations, involve a very important register known as the *accumulator* (also called the A-register). Like memory words the accumulator has room for four hexadecimal digits; but unlike memory, it is built of high-speed circuitry and is physically located inside the CPU. The accumulator behaves somewhat like a scratchpad for our calculations: ADD really means "add to the accumulator" and SUB means "subtract from the accumulator".

When the sample program (Section 1-7) is executed, the first instruction encountered (location 12F7) is the machine-coded form of LDA R, an abbreviation for "load" the accumulator with (the contents of) the location labeled "R". The machine looks ahead to location R(1300) and dutifully copies the contents (in this case, the number 7) into the accumulator. The program counter is now pointing at the instruction in location 12F8 (i.e., ADD S), so the machine adds *the contents of* location 1301 to the number already in the accumulator. The next instruction causes a subtraction of *the contents of* 1302 from what is in the accumulator. Finally, the value in the accumulator is saved by storing it in some unused memory location (we happened to choose location 1303 and name it X: STA means "store (the contents of) the accumulator in location X". This last instruction frees the accumulator for possible future use; however, we do not use it again in this program, since the next instruction encountered is the IDL (idle) instruction which terminates execution by halting the machine.*

We shall later discuss a way to examine the contents of location 1303 to discover what the answer is. (Remember, it will be in hex!)

It is worth noting that both the *load* and the *store* are merely "copying" operations so that numerical values may be copied back and forth between the accumulator and memory. As is the usual case with copying, the original is not destroyed. When we do an LDA from R, the original value remains in R and a copy of the original is placed in the accumulator. Whatever was in the accumulator before will, of course, be destroyed in the process. The same thing is true of STA X: whatever *was* in location X will be destroyed when the accumulator is copied into it, and the accumulator retains the original from which the copy was made. Locations S and T will remain unchanged, as well.

1-9 ASSEMBLING, LOADING AND EXECUTING THE PROGRAM

Now that we've written a simple program, let's run it on the 980. These directions assume that we have a 980 with at least 8k of memory, a card reader, a line printer and a console Silent 700 ASR Data Terminal with a cassette. We'll assume that the basic operating system has been loaded by the previous user and all we have to do is ready all peripheral devices. The SAPG assembler and our program are both on punched cards.

*If we run this program using the basic operating system, our use of the IDL instruction will be regarded as illegal. Execution will stop because of the attempt to execute a privileged instruction and an error message will be printed out. We'll ignore the error for the time being and as far as stopping goes — that is what we are trying to do.

1. Depress START switch on the 980 front panel.

The operating system will respond with **"*READY*"** printed out at the console.

2. Type in the following logical unit assignments, terminating each line with a carriage return:

```
//ASSIGN,4,KEY.  
//ASSIGN,5,CR.  
//ASSIGN,6,LP.  
//ASSIGN,7,CS1.  
//ASSIGN,10,DUMMY.
```

3. Load the SAPG assembler into the card reader, and type:

```
//EXECUTE,CR.
```

The cards will be read and the system will respond with **"READY SOURCE, HIT CR"**

4. Put the source deck in the card reader and hit carriage return on the console for *pass 1* of the assembly.

5. Repeat step 4 for *pass 2* of the assembly. The assembly listing should appear on the line printer, and the object program will be written on cassette CS1.

6. Place a /* card in the card reader and hit carriage return to write an end-of-file in the object program output and to take control away from the assembler and return it to the operating system.

7. Type on the console:

```
//EXECUTE,CS1.  
//REWIND,7.
```

8. Your program has executed if you receive a privileged instruction message. To the untrained observer it may appear that nothing has happened. The value of X does not appear either at the console or the line printer, which should be no surprise because we haven't put any output instruction in the program.

1-10 FINDING THE ANSWER.

We know that the answer will be stored in location 1303, so we'll set up the address 1303 on the 980 switches: 0001 0011 0000 0011. We'll enter the address into the memory address (MA) register by setting the MA switch to the up position. Now, we can display the contents of the location by setting the memory data (MD) switch to the down (display) position. The contents (i.e., the answer) will be displayed by the 980 lamps. We should see the answer as a binary 3, namely:

0000 0000 0000 0011

The relationship between binary and hexadecimal is explored in the next section.

SECTION 2

MACHINE WORDS AND THE BINARY NUMBER SYSTEM

2-1 WHAT'S IN A WORD?

Now that we have gained some insight into the overall structure and operation of the 980 computer, we are ready to consider some deeper questions, such as:

- Why is the hexadecimal number system so important?
- Is there a limit to the size of data numbers?
- How can a negative number be represented in the 980 memory?

To provide answers, we will have to think about some finer details relating to the structure of the machine.

By their very nature, electronic devices depend on circuit elements conducting at times and not conducting at other times. Stated in the simplest terms this means that the computer "thinks" in an "on-off" type of language. Some computer memories are made of tiny magnetic rings called *cores*, "remembering" information by magnetizing the cores in a pattern of positive and negative field directions. Other memories such as the 980 utilize MOS or solid-state devices which employ thousands of gates. Each gate, analogous to a bit, either *on* or *off* as required for a particular word description.

Both the ON/OFF and PLUS/MINUS notation have something in common with the true/false notation so favored for school quizzes: all these examples provide a *binary* (or two-state) choice. It's very convenient to express any of these choice/pairs in a number system that has only two digits; therefore, we'll select the binary (base 2) number system which uses only the digits 0 and 1.

As it turns out, the computer words we have described in terms of hexadecimal digits are actually composed of 16 binary digits (or *bits*) of information. The ADD S instruction in our earlier example which we wrote:

2008

assumes a bit pattern of

0010 0000 0000 1000

inside the computer. We'll soon see that the two notations can be completely reconciled with each other: the hexadecimal notation is used as a kind of shorthand to represent the more cumbersome binary representation.

In the binary number system, all numbers are represented in terms of the digits 0 and 1. Using the same arguments used to develop the representation of quantities in the hexadecimal number system (Section 1-4) we can write down an expanded table as shown in Table 2-1.

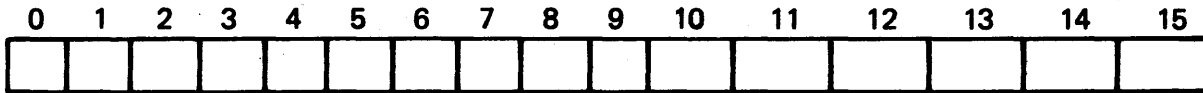
Table 2-1. Comparison of Decimal, Octal, Binary, and Hexadecimal Number Systems

<u>Number</u>	<u>Decimal</u>	<u>Octal</u>	<u>Binary</u>	<u>Hexadecimal</u>
zero	0	0	0	0
one	1	1	1	1
two	2	2	10	2

Table 2-1. Comparison of Decimal, Octal, Binary, and Hexadecimal Number Systems (continued)

<u>Number</u>	<u>Decimal</u>	<u>Octal</u>	<u>Binary</u>	<u>Hexadecimal</u>
three	3	3	11	3
four	4	4	100	4
five	5	5	101	5
six	6	6	110	6
seven	7	7	111	7
eight	8	10	1000	8
nine	9	11	1001	9
ten	10	12	1010	A
eleven	11	13	1011	B
twelve	12	14	1100	C
thirteen	13	15	1101	D
fourteen	14	16	1110	E
fifteen	15	17	1111	F
sixteen	16	20	10000	10

For easy reference the 16-bit positions in a computer word are individually numbered from left to right as follows:



Occasionally, we need to refer to a specific bit position in a computer word; we can do so by specifying one of these numbers.

2-2 NUMBER SYSTEM CONVERSION BETWEEN BINARY AND HEXADECIMAL

Inspection of Table 2-1 indicates that any hexadecimal digit we select can be represented by four binary bits. In binary representation, as in decimal, leading zeros do not alter the value of a number: "one" can be expressed as 1, 01, 001, 0001, and so forth.

We can express any collection of 16 bits as a string of four hexadecimal digits by grouping the bits in fours (starting from the position of the radix) or "binary" point on the right and writing the hexadecimal digit representing each bit group. For example,

$$\begin{array}{cccc} \underline{0011} & \underline{1100} & \underline{0111} & \underline{0101} \\ 3 & C & 7 & 5 \end{array}$$

It also works the other way around. Each hex digit can be "expanded" into its four-bit binary representation.

Fractional numbers are converted the same way as integers except we start with the radix point on the left when marking off the groups of four

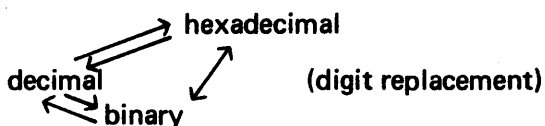
$$\begin{array}{ccc} \underline{.1101} & \underline{0110} & \underline{12} \\ .D68 & 16 & \end{array}$$

to get the equivalent hexadecimal fraction. Note that an incomplete group of four may be filled out with zeros following the least significant bit without altering the value of the fraction.

The ease of the binary-hexadecimal conversion arises from the fact that 16 (the base of the hex system) is itself an integral power of 2 (the base of the binary system).

2-3 CONVERSION TO AND FROM DECIMAL.

Since 10 (the base of the decimal system) is not an integral power of 2 or 16, the conversion of decimal numbers to and from their hexadecimal or binary representations are not as easy as the hex/binary conversion. Nonetheless, it is a job which has to be done. Frequently, we can ask the computer to do it for us; but we should know how to do it ourselves. If we want to convert from decimal to binary, we can do it directly; but it is more convenient for us to convert decimal to hex and then hex to binary by expanding the hex digits into their binary representations:



Regardless of which route is taken, the results are the same if everything is done correctly.

2-3.1 HEXADECIMAL TO DECIMAL. Just as each decimal *place* represents a power of 10, each hex *place* represents a power of 16. Table 2-2 shows the first five powers of 16 for reference.

Table 2-2. Integral Powers of 16

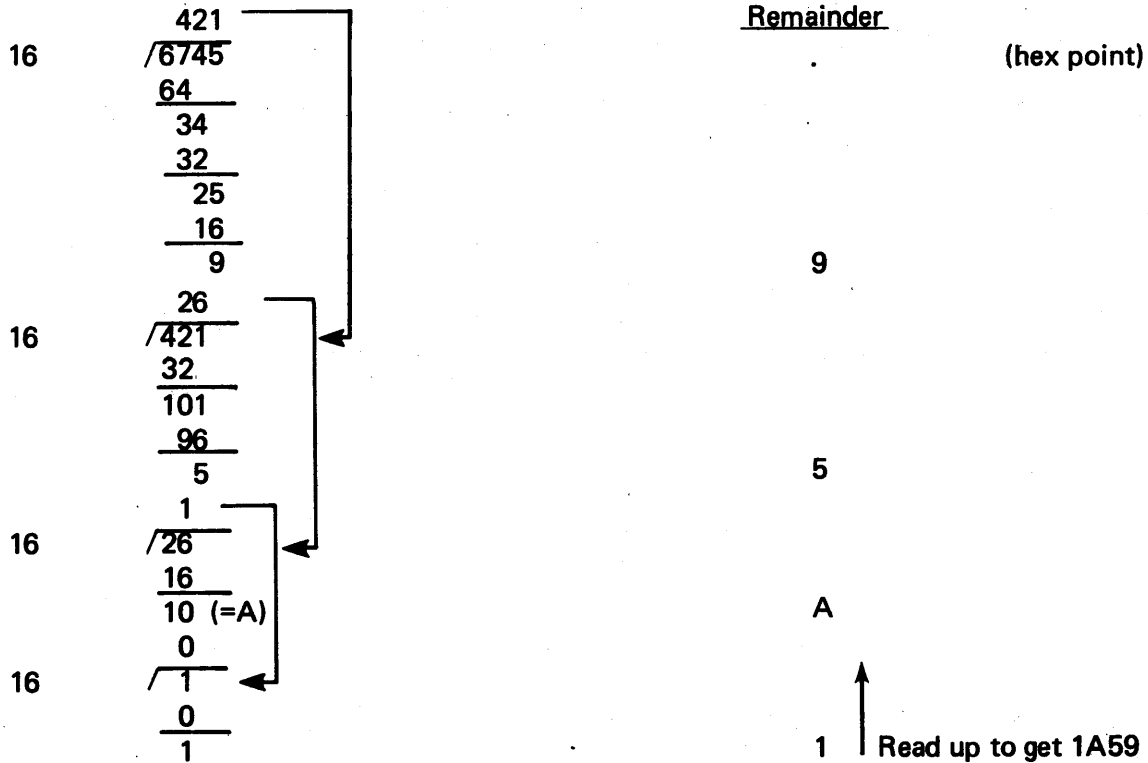
16^0	=	1
16^1	=	16
16^2	=	256
16^3	=	4096
16^4	=	65536

We can convert the number $1A59_{16}$ to decimal by realizing that the number is merely the sum of

$$\begin{array}{r}
 9 \times 16^0 = 9 \times (1) = 9 \\
 5 \times 16^1 = 5 \times (16) = 80 \\
 A \times 16^2 = 10 \times (256) = 2560 \\
 1 \times 16^3 = 1 \times (4096) = 4096 \\
 \hline
 6745_{10}
 \end{array}$$

where all the arithmetic is in decimal. The qualitative reasonableness of the answer can be checked by noting that the decimal representation is larger than the hex representation.

2-3.2 DECIMAL TO HEXADECIMAL. Probably the easiest way to convert decimal to hex is to perform successive divisions by 16 and note the remainder each time. (This method permits all arithmetic to be done in decimal.) The process is continued until a zero appears as the quotient. Let's convert the number we just found back to hexadecimal.



2-3.3 CONVERSION OF FRACTIONS. Converting fractional numbers (for example, 0.887_{10}) involves multiplying by 16, stripping off the integer portion to serve as a digit of the converted fraction, and reading down.

	Carry		0.887
(hex point)	.		
	E = 14		$\begin{array}{r} \times 16 \\ \hline 14192 \end{array}$
	3		$\begin{array}{r} \times 16 \\ \hline 3072 \end{array}$
	1		$\begin{array}{r} \times 16 \\ \hline 1152 \end{array}$
	2		$\begin{array}{r} \times 16 \\ \hline 2432 \end{array}$
			...

giving us $(.E312\dots)_{16}$ for as many places as we have the need or patience to carry out the conversion. Note that there are many cases in which we could continue generating carry digits to infinity without ever arriving at a zero multiplicand (the signal to stop). By stopping too soon we introduce a *truncation error* into the converted value of the number. The truncation error is a necessary consequence of the fact that computer words have a finite number of bits. We must learn how to minimize it, when to put up with it, and always be aware that it may be present.

Conversion in the other direction is accomplished by multiplying by negative powers of 16 (see Table 2-3).

Table 2-3. Negative Integral Powers of 16

16^{-1}	=	0.0625
16^{-2}	=	0.0039062
16^{-3}	=	0.0002441
16^{-4}	=	0.0000152

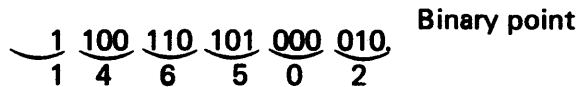
2-4 NUMBER CONVERSIONS TO AND FROM OCTAL.

Many computer systems use the octal number system as their shorthand for binary, this system is one with which every assembly language programmer should be conversant.

2-4.1 NUMBER CONVERSION BETWEEN BINARY AND OCTAL. Inspection of the first eight entries of Table 2-1 indicate that any octal digit we select can be represented by three binary bits (instead of the four we used for conversions to and from hex):

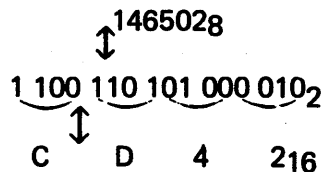
<u>Octal Digit</u>	<u>Binary Representation</u>
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

We can express any collection of bits as a string of octal digits by grouping the bits in threes (starting from the position of the radix or *binary* point on the right), and writing the octal digit that represents each bit-group; for example,



Note that the addition of two leading zeros does nothing to change the value expressed. Each octal digit can be "expanded" into its three-bit binary representation.

2-4.2 CONVERSIONS BETWEEN OCTAL AND HEXADECIMAL. Probably the easiest way is to write out the binary representation and regroup the digits



2-4.3 OCTAL TO DECIMAL. Each octal *place* represents a power of 8. We can convert the number 1567₈ to decimal by realizing that the number is merely the sum of

$$\begin{array}{rcl}
 7 \times 8^0 & = & 7 \times (1) & = & 7 \\
 6 \times 8^1 & = & 6 \times (8) & = & 48 \\
 5 \times 8^2 & = & 5 \times (64) & = & 320 \\
 1 \times 8^3 & = & 1 \times (512) & = & 512 \\
 & & & & \hline
 & & & & 887_{10}
 \end{array}$$

where all the arithmetic has been done in decimal.

Check the qualitative reasonableness of the answer by noting that the decimal representation is smaller than the octal representation.

2-4.4 DECIMAL TO OCTAL. Perform successive divisions by 8 and note the remainder each time. (This method permits doing all arithmetic in decimal.) The process is continued until a zero appears as the quotient.

	Remainder	octal point
$ \begin{array}{r} 110 \\ 8 \overline{) 887} \\ \underline{13} \\ 8 \overline{) 110} \\ \underline{1} \\ 8 \overline{) 13} \\ \underline{0} \\ 8 \overline{) 1} \end{array} $	7 6 5 1	↑ Read up to get 1567.

Working with fractional numbers (for instance, 0.887₁₀) involves multiplying by 8, remembering the carry, and reading down.

	Carry
read down ↓	$ \begin{array}{r} .887 \\ \times 8 \\ \hline 7 \leftarrow .096 \\ \underline{8} \\ 0 \leftarrow .768 \\ \underline{8} \\ 6 \leftarrow .144 \\ \underline{8} \\ \dots \quad 000 \end{array} $

giving us (.706 . . .)₈ for as many places as we have the need or patience to carry out the conversion.

2-5 CONVERSIONS BETWEEN BINARY AND DECIMAL.

The general method from the previous two examples should be clear enough that we need add only that the divisor (or multiplier, depending on the direction of the conversion) is a 2 or one of its powers.

2-6 POSITIVE AND NEGATIVE DATA VALUES.

Since a 980 word is made up of 16 bits, it should be no great surprise that the size of a number that can be stored in a word is limited. But we are able to represent all integers between -32768 and $+32767$

$$-32768, -32767, \dots, -1, 0, 1, \dots, 32766, 32767$$

Since this is the first mention of negative numbers, let's see how the machine differentiates between positive and negative numbers.

2-6.1 THE SIGN BIT. The left most bit position (position 0) in the word is treated as a *sign bit*. If this bit is a 0, the number is positive; if the bit is a 1, the number is regarded as negative. The largest positive number that can be stored in a word is

$$\begin{array}{c} \text{sign bit} \downarrow \\ 0111111111111112 \\ \text{or} \\ 7FFF \\ \text{or} \\ 32767_{10} \end{array}$$

The reason we can safely claim this is the largest value is that if we tried to add 1 to this number, we'd get

$$\begin{array}{c} \text{sign bit} \downarrow \\ 1000000000000000 \\ \text{or} \\ 8000_{16} \end{array}$$

which has a 1 in the sign bit position and is therefore a negative number. It is, of course, mathematical nonsense to claim that one can add two positive numbers and obtain a negative result: so the computer will flag this situation for us (by the overflow bit in the status register).

2-6.2 NEGATIVE NUMBERS

Because of the arithmetic circuitry of the 980 computer, negative numbers have to be stored in a special way called the *two's complement* form. Let's see how we go about representing the number -1810 in a computer word.

First, we'll try to get the binary representation of the corresponding positive number, +18. Expressing the number in hex gives us 12₁₆; and this in turn gives the following 16-bit binary representation:

0000000000010010

Next, we'll perform a binary subtraction of this number from a *seventeen-bit* number consisting of a 1 followed by 16 zeros.

$$\begin{array}{r}
 1\ 0000\ 0000\ 0000\ 0000 \\
 -\ 0000\ 0000\ 0001\ 0010 \\
 \hline
 1111\ 1111\ 1110\ 1110
 \end{array}$$

This, now, is the bit pattern used to represent -18₁₀. Converting the bit pattern to hexadecimal shorthand:

F F E E

which is the value that will appear in an assembled listing of the data value. Note that the bit pattern contains a 1 in left most bit position as a negative number should.

After we become adept at hexadecimal addition and subtraction, we can discard the binary crutch and do it this way

$$\begin{array}{r}
 10000 \\
 -\ 0012 \\
 \hline
 FFEE
 \end{array}$$

where we make use of the addition tables for the hexadecimal number system – in particular

$$E + 2 = 10$$

$$E + 1 = F$$

and

$$F + 1 = 10.$$

Confronted with the number FFEE: we know it is negative, but we would like to know its magnitude (absolute value). This can be determined by finding the number that must be added to FFEE to make the sum equal to 10000₁₆. The number is, of course, 12.

Since positive numbers have a 0 in the left most bit position and negative numbers have a 1 in that position any 4 digit numbers with a left most hex digit between 0 and 7 are positive. Those with 8 through F as their left most hex digit are negative.

Integers in 16-bit, two's complement notation then range across the following limits

<u>Decimal</u>	<u>Hex</u>
+32767	7FFF
.....
+1	0001
0	0000
-1	FFFF
-2	FFFE
.....
-32768	8000

2-6.3 TWO'S COMPLEMENT ARITHMETIC. Subtraction can always be accomplished by adding a complement; for example,

<u>Decimal</u>	<u>Hexadecimal</u>
32767	7FFF
- 2	-0002
<u>32765</u>	<u>7FFD</u>

Using two's complement addition, the convention is to throw away the carry bit if one appears.

32767	7FFF
+ (-2)	+FFFE
<u>32765</u>	<u>7FFD</u>

SECTION 3

INTRODUCTION TO 980 ASSEMBLY LANGUAGE

3-1 ELEMENTARY PROGRAMMING CONSIDERATIONS.

Successful assembly language programs can be produced using a rather small subset of the complete machine instruction set. Such a program is not necessarily the shortest possible in terms of the number of memory locations used: nor does it necessarily execute with great efficiency. But it can be made to work. One argument in favor of a large instruction set is that it often permits use of a single instruction to do a job that might otherwise require a block of instructions.

The intent of this section is to start the novice programmer with a small group of instructions and an assembly language which will prove useful in writing simple programs. Many of the instructions introduced in Table 3-1 will ultimately prove more powerful than these examples indicate, so we will revisit this subset in its full-blown form in Section 4.

Table 3-1. Elementary Language

<u>Instructions:</u>		
LDA, LDE, LDX, LDM	Load from memory	
STA, STE, STX	Store to memory	
ADD	Add	
SUB	Subtract	
IMO	Increment memory by one	
DMT	Decrement memory and test	
RMO	Register move	
REX	Register exchange	
RAD	Register add	
RSB	Register subtract	
RIN	Register increment	
RDE	Register decrement	
RCO	Register two's complement	
BRU	Branch unconditional	
BIX	Branch on incremented index	
SEV	} Conditional skip on register	
SOD		even
SNZ		odd
SZE		not zero
SMI		zero
SPL		minus
		plus
IDL	Idle	
<u>Assembler Directives:</u>		
IDT	Identification	
END	End	
DATA	Data declaration	
ORG	Origin	
EQU	Equate	

Table 3-1. Elementary Language (continued)

Addressing Modes:

Immediate

P-relative

Indexed P-relative

Expressions — “address arithmetic”

The following restrictions apply to this section only, so the new programmer should be prepared to relax them later:

- One assembly language instruction translates into one word of machine code.
- The only numbers we shall use are the integers: positive, negative, and zero.
- Only two of the numerous modes of addressing are used; these are called *immediate* and *program-counter relative* (they will be discussed in another section).
- Programs are assembled and run under control of the basic operating system.
- We shall not try to perform any I/O. All data needed by the machine will be predefined in the program at assembly time, and all results will be found by operator examination of the appropriate memory locations using the display lights on the 980 front panel.
- We shall assume that our source program is input from cards.

3-2 THE INSTRUCTION FORMAT SKELETON.

The formatting details of the various 980 instructions and assembler declaratives differ somewhat, but the basic skeleton is the same for all. The first six-column (field) constitute space reserved for a *label* (i.e., a name we may wish to attach to a particular instruction or data value) and has the same function as the optional *statement number* in FORTRAN. A label must contain no more than six* characters, the first of which must be alphabetic and must appear in column 1.

The other characters may be letters, digits, or selected special symbols. (Some symbols are illegal, namely +, -, *, /, (,), <, and comma, so we'll restrict our labels to letters and digits only to avoid the inadvertent use of a prohibited symbol.) The label is terminated by the first blank.

The next field is the *operation field* containing the mnemonic code or assembly directive. It must be separated from its predecessor by a comma. The first blank encountered assumed to terminate the operand list. Operands may actually be expressions; however, we shall restrict ourselves to single-element operands for the time being.

*During assembly these labels are stored in the symbol table as variable length data. One or two character labels require three words of memory, three or four character labels require four words, and five or six character labels require five words. Therefore, if symbol table overflow occurs, labels should be shortened or omitted where possible.

Since the first blank ‡ encountered after the beginning of the operand field signifies the end of the field, the programmer may include an optional comment on each line. This comment will be ignored by the assembler.

The skeletal format for an instruction is as follows:

<label> † <operation code> † <operand1>, <operand2>, <operand3> † <comment>

Full-line comments may be included if the initial character is either a period (.) or an asterisk (*); both are also ignored by the assembler. For example.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
. THIS IS A COMMENT																															
*SO IS THIS																															
LABEL 1			LDA			VALUE			PUT			VALUE			IN			A													
			RMO			A, X			MOVE			A			TO			X													
			IDL			φ			HALT																						
VALUE			DATA			3 9																									
└──────────┘			└──────────┘			└──────────┘			└──┘																						
label			op code			operands			comments																						

3-3 DATA DECLARATIONS.

The DATA declaration is actually an assembler-directive (pseudo-op) which instructs the assembler to do two things:

- (1) Set aside a memory location and label it in accordance with the programmer's wishes.
- (2) Prestore in that location whatever number the programmer specifies.

For example, a line reading

XYZ DATA 3

assembles into a single word containing bits which represent *three*. The word would be labelled during the assembly process as XYZ so that we can refer to it by the name or symbol XYZ rather than by hexadecimal address.

The data values can be expressed as octal, decimal, or hexadecimal numbers or as an ASCII (character) string.

The assembler distinguishes between the various data types in the following manner:

- A number in octal notation, denoted by at least one leading zero, is translated by the assembler into the corresponding hexadecimal value.
- A number in decimal notation, denoted by a nonzero leading digit, is translated by the assembler into hexadecimal.
- A number in hexadecimal notation is prefixed by the symbol >.

‡A blank or "space" in this notation will be represented in this notation as a "slash b": b.

Each field in the DATA declaration is assembled into a consecutive word; for example,

DATA 13, >5AC, 010, -2

will assemble into the four words

000D
05AC
0008
FFFE

Sometimes, the programmer, may wish to declare an ASCII character string as data. He could look up the character codes and insert them as pairs of hexadecimal digits:

C = C3 b = A0
 A = C1
 T = D4

For example,

DATA >C3C1,>D4A0

is a bit more trouble than necessary in most cases. The assembler would have produced the same result if we had written the ASCII characters surrounded by single quote marks

DATA 'CATb'

The ASCII string notation is fine for printed characters, but action (non-printed) characters such as carriage return and line feed must be entered as the corresponding numeric values.

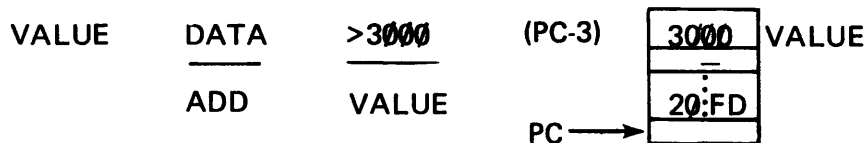
3-4 TWO ADDRESSING MODES, P-RELATIVE AND IMMEDIATE.

The two addressing modes we will probably use most frequently are *program counter relative* (P-relative) and *immediate*.

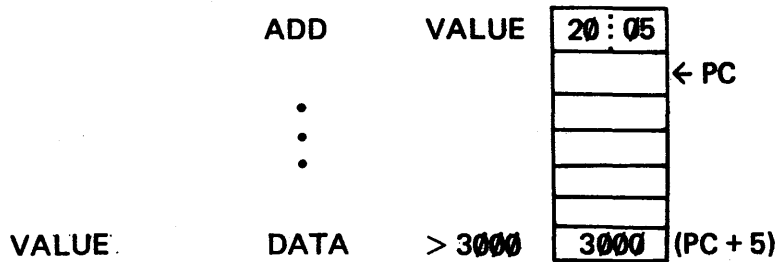
When we use an instruction such as

ADD VALUE

we consult the symbol table to find the location of the quantity labeled VALUE. To provide program relocatability (i.e., the ability to place the program in any set of memory locations which may prove convenient without tying into one specific block of absolute addresses), the assembler computes the "displacement" of VALUE in the negative (backward) or positive (forward) direction from the value the PC has at that instant during the execution phase. (Displacement is simply a count of instructions indicating how far away VALUE is defined and in what direction.) If VALUE is three instructions back from the PC, displacement will be a -3 expressed in two-digit hex form (FD):



whereas, if value is five instructions beyond the PC, the displacement will be expressed as a +5:

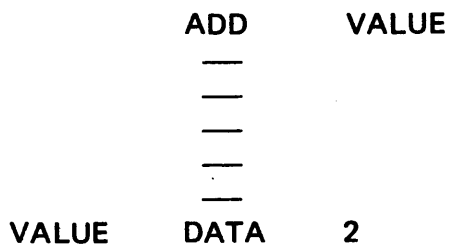


Note that the count can only be as great as two hexadecimal digits will denote – or from 7F in the positive (higher address) direction to 80 in the negative direction. That is to say, we may count ahead 127₁₀ or back 128₁₀.

Actually, the assembler does all the work for us, but we must be careful not to let the label referenced exceed the range the assembler can handle. (If we actually *must* put the label farther away, there is a means of doing it involving an *extended instruction*, which is discussed later. For now, we'll observe the range restriction.) PC relative addressing, then, tells the machine during execution:

1. Extract the displacement.
2. Add it to the program counter (adding a negative, of course, is equivalent to subtracting a positive) to compute an address.
3. Use the operand associated with that address (in case of an ADD instruction, the operand will be the *contents* of that address).

We could use this form of addressing to cover a multitude of cases; however, 980 AL provides a shortcut called an *immediate address*. For example, to ADD the value 2 to the accumulator, we write in P-relative mode:



We could use the immediate mode (thereby saving the storage location containing VALUE) by writing

ADD =2

where the "=" means use the number itself rather than the contents of a location (omitting the equal sign would have caused the machine to add the contents of location 2 rather than the number 2; we are essentially using an *address* itself as the item to be added rather than the contents of the address).

3-5 TRANSFER OF CONTROL: BRANCH AND SKIP.

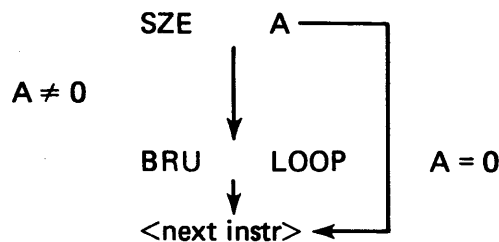
Programs execute sequentially, with increasing PC, unless this order is interrupted by a transfer of control to some instruction other than the next in sequence. This process is called *branching* and can be done unconditionally (e.g., the GO TO statement of FORTRAN) or conditionally, as in the FORTRAN "IF" statement.

There is one unconditional branch instruction in 980 AL: the BRU instruction, in which the address to which branching occurs is assembled as a + or - displacement; i.e., a P-relative mode. (Again the -128 to +127 restriction holds.)

The 980 AL depends on *skip* instructions for most of its conditional branching. For example, the instruction

SZE A

means *skip-on-zero in the A-register*; i.e., if the accumulator contains a zero, skip the next instruction; otherwise execute it. One will often see the skip instruction associated with a branch, as in the following example:



in order to achieve a conditional split in the program flow.

3-6 THE ASSEMBLY PROCESS AND SOME ASSEMBLER DIRECTIVES.

Some operation codes do not really represent executable operations at all (hence the name *pseudo-op*), but serve as instructions to control the course of the translation (or *assembly*) process. Since most pseudo-ops are useful only during assembly, they need not appear in the object code; thus they are not translated and consume no space in the assembled program.

The first group of assembler directives we shall consider are

END
EQU
IDT
ORG

Each program can be given a name, and this name is given by the operand in the identify (IDT) directive, which should appear as the first line of the program:

IDT <name>

where <name> represents a six-character label. If more than six characters are specified, only the first six are used and the remainder discarded.

The END pseudo-op should be the last line of the input program; it indicates to the translator that no more instructions are to be assembled.

Programs are assembled in two passes. During the first pass, the assembler seeks all symbols (labels) in the program and builds a temporary table (the *symbol table*) which lists each symbol and the position in the program where it is defined. After all program instructions have been examined once (pass 1) and all symbols found, the assembler examines all instructions again (pass 2), consults the symbol table to find the actual location of the operands, and writes the appropriate machine code (*object program*) onto some output unit specified by the programmer. At the end of the translation process, execution may be performed by loading the object program into the machine, entering the address of the instruction where execution is to begin (the "entry point"), and relinquishing control to the object program. The PC can be loaded in two ways:

1. Manually by the operator, using the toggle switches on the CPU front panel
2. Automatically by the loader program, if the programmer has included in his END operation the label which identifies the entry point.

We are now in a position to regard the executable program as the filling of a sandwich, with the IDT and END pseudo-ops enclosing the filling. For example,

```

                                IDT <name>
                                <data declarations, if any>
<entry>                        <first executable instruction>
                                —
                                —
                                —
                                —
                                —
                                —
                                —
                                —
                                —
                                IDL 0
                                <data declarations, if any>
                                END  <entry>

```

Often <name> and <entry> are selected to be the same label, but there is no real significance in doing so.

During the assembly process the assembler uses a pointer to the consecutive locations into which the object program is being written. This pointer, called the *location counter*, can be thought of as an assembly-time analog of the PC.

	<u>Location Counter</u>	<u>Program Counter</u>
operates during:	assembly	execution
points to:	instruction being assembled	next instruction to be executed
increments sequentially unless:	altered by an ORG pseudo-op	altered by a branch or skip instruction

The ORG (origin) pseudo-op tells the assembler in which absolute memory location to place the next instruction. Usually a program is assembled so as to be "relocatable" (i.e., it can execute in any block of memory deemed convenient at load time); however, cases may arise in which the programmer wishes to face the program with a specified block of absolute locations.

The instruction

ORG <numeric address>

accomplishes this. This manual uses the ORG instruction in this manner in order to attach the program to some absolute locations so that its structure can be discussed more easily. No other reason for its use, exists, and ORG should usually be omitted in most programming applications. ORG does have one other use in both relocatable and absolute programming: skipping over a specified number of locations which may be used for storage later on. We could write

ORG \$+500

where \$ means *the present value of the location counter*, and the expression there increments the current location counter value by 500 words to find the cell into which the next instruction will be assembled.

The entry point designator can be inserted almost as an afterthought by the use of the EQU (equate) pseudo-op. Instead of punching the entry point label on the card containing the first executable instruction, we can write

<entry point> EQU \$
<First executable instruction>

which is equivalent to writing:

<entry point> <First executable instruction>

It means *equate the entry point label to the current value of the location counter*, where \$ once again means "this location".

3-7 REGISTER ORGANIZATION AND DEFINITION.

The TI 980 has nine 16-bit registers. All are under program control, and eight are directly addressable. Uses of the registers are introduced in the forthcoming sample programs, but for the moment the following list and a brief description will suffice:

Register Number	Usual Program Designation	Function
0	A	Primary arithmetic register (accumulator)
1	E	Auxiliary (extension) arithmetic register
2	X	Index register (address modification)
3	M	Maintenance register (temporary storage and Basic Operating System I/O)
4	S	Storage register (temporary storage)

Register Number	Usual Program Designation	Function
5	L	Link register (subroutine linkage)
6	B	Base register (base address for operands)
status block {	7	Program counter (PC, address of next instruction)
	8	Status register (program status and interrupt enable/disable)

The program counter and status register comprise the *status block*, a register pair used extensively in programming interrupt service routines. The status register differs somewhat from the others in that our attention will be directed toward the state of various bit positions rather than the numerical value the 16 bits of the other registers usually represent.

When a register is exclusively affected by an instruction (LDA, or *load accumulator* for example), the alphabetic register designator is often built into the instruction's mnemonic code. Other kinds of instructions permit us to operate on the contents of the register we specify as an operand (for example, SZE A or *skip on zero in register A*). We may, of course, write SZE 0 instead, but since we usually find it easier to think of the accumulator as register A rather than register 0, we should make the register number, and the alphabetic designator equivalent for use in the program. We state this equivalence through use of the EQU pseudo-op. Defining A to be the symbol for the accumulator (register 0) we write

```
A      EQU      0
```

and so forth for the registers to which we wish to attach symbols. Note there is nothing magical about the use of the letter "A". We could have written any legal label

```
A      EQU      0
                LDA
                SZE      DATA
                A
```

which will change all references to the accumulator when written as an operand. Of course, it will never change the LDA mnemonic.)

3-8 NOTATION USED IN DESCRIBING INSTRUCTIONS.

The actions performed by machine instructions are traditionally depicted in shorthand form. When we speak of a particular register, we refer to it by its "official" designator — the accumulator, for example, is referred to as A, and in general the program counter is called PC or P. When we speak of a register in general (i.e., we do not want to be specific about which one), we use a symbol which denotes the register's role in the instruction. The register-to-register transfers include a source register (the one from which information is taken) and a destination register (the one to which information is transferred), referred to herein as s and d, respectively. In a specific case, we might use A as the source and X as the destination, for example.

In speaking of the label of memory location, we usually refer to m. If we talk about *the number stored at m*, we will write (m)[‡] i.e., "the contents of" m. [Similarly (A) would be "the contents of" A.] The ADD instruction discussed in the next section, when written "ADD m" means: *take the contents of location m and add it to the present contents of the accumulator, leaving the resultant sum in the accumulator.*

[‡]We will use () to mean "the contents of" in this manual. However the () is never used in an assembly language statement.

Our notation to describe this operation would be

$$(m) + (A) \longrightarrow A$$

where the arrow shows where to find the result of the operation. The STA m (*store contents of accumulator in the location labeled m*) instruction would be described as $(A) \longrightarrow m$.

One of the common confusions of the newcomer is to write $(A) \longrightarrow (m)$ instead. This is yet another mode of addressing called *indirect addressing* and is denoted by an asterisk (*) preceding the operand. In indirect addressing the number contained in m, [i.e., (m)] is used as a pointer to another address where the contents of A will actually be stored. Following is an illustration of the difference:

	ORG	1000
PTR	DATA	1003
STA	PTR	stores the contents of A into location 1000 $(A) \longrightarrow m$
STA	*PTR	stores the contents of A into the address pointed to by location 1000, namely into 1003 $(A) \longrightarrow (m)$

Indirect addressing also can be specified by writing a modifier 4 in the second operand field:

STA PTR, 4

Both instructions assemble identically.

3-9 A BASIC GROUP OF INSTRUCTIONS.

The following subsections introduce a basic group of instructions for use in writing simple assembly language programs. These instructions are introduced here in their simplest form and again in Section 4 in their more general form.

3-9.1 IDLE INSTRUCTION (IDL). The IDL instruction is used in this manual whenever we need a program executable instruction to direct the machine to halt. This instruction is *privileged*; i.e., a mode of computer operation (with the memory protect/privileged instruction feature, MP/PIF, enabled) in which the IDL is an illegal instruction. Since MP/PIF is enabled when the 980 computer is operating under the Basic Operating System, these programs cannot be run in that environment without generating a fatal error and consequently being thrown off the system. Encountering the IDL means program execution has ended anyway, so the difference between our making a graceful exit and being thrown off is merely one of aesthetics at this point. Later on we will use a supervisor service call all in place of IDL to improve the aesthetics of the situation.

3-9.2 REGISTER-TO-MEMORY AND MEMORY-TO-REGISTER TRANSFERS. These transfer instructions are essentially *copying* instructions in that they copy data from one place to another without destroying the value from which the copy was made. The memory-to-register direction is called a *load* and the register-to-memory, a *store*.

Only four of the registers (A, E, X, and M) may be copied into with a *load* instruction, and only three of them (A, E, and X) may be copied out of. The others must have values inserted some other way.

The simple sequence

A	EQU	0	
	LDA	NUMBER	MOVE (NUMBER) TO A
	STA	COPY	SAVE IN COPY
	IDL	0	HALT
NUMBER	DATA	>C3	
COPY	DATA	0	

copies hexadecimal C3 into the accumulator, leaving the value in NUMBER unchanged. Then it stores the value in the accumulator into the location named COPY, thereby wiping out the zero value placed there by the assembler during the translation process. The accumulator still contains the value C3 and continues to do so until we destroy it by inserting another value.

We can indicate in symbolic form what happens by the following notation:*

LDA m (m) → A

i.e., the contents of location m goes to the A-register and

STA m (A) → m

i.e., the contents of A is transferred into location m.

The other load-and-store instructions work identically, except different registers are involved.

3-9.3 REGISTER-TO-REGISTER INSTRUCTIONS. Some registers have no load-and-store instructions, yet we need to get information into and out of them. At other times we may already have in one register the number we wish to copy into another, and it takes the machine less time to copy directly between registers than to load a register from memory.

The register-to-register instructions have two operands: s, a source (the register copied from) and d, a destination (the register copied into). These instructions have the general format of

<register operation> s, d

The two specific examples considered in this section are

the <i>register move</i>	RMO	s, d	(s) → d
and the <i>register exchange</i>	REX	s, d	{ (s) → d
			{ (d) → s

where the RMO copies source into destination, leaving a copy in the source: and the REX exchanges the

*Later on we shall write this instruction and other memory referencing instructions in a more general form

LDA m, <mod> (e.a.) → A

where the effective address, e.a., is obtained from modifying m with the <mod> field. When the modifier field is absent, the effective address is the same as m.

the contents of the two registers. The needs of a specific problem may indicate exactly which one of these should be used; however, in one case the programmer can make an arbitrary choice. This case is one in which, say, we wish to copy the X register into E and we no longer care what happens to X. Obviously we could use either instruction

E	EQU	1
X	EQU	2
	RMO	X, E

and we will most often choose whichever instruction is faster for the machine, which turns out to be the RMO instruction.

3-9.4 UNCONDITIONAL BRANCH INSTRUCTION. We may sometimes wish to alter the value in the program counter (P-register) in order to force resumption of execution in some location other than the next sequential one (analogous to the GO TO statements of higher level languages). SAL does this in 980 AL with the *branch unconditional* instruction

BRU y y → PC

where y is the address (usually the label attached to the address) of the location containing the next instruction to be executed. This instruction assembles with the address y converted to a positive or negative displacement from the present portion of the PC.

3-9.5 ARITHMETIC INSTRUCTIONS. There are four arithmetic operations discussed in this section. First are ADD and SUBtract. Both use the contents of the accumulator as the implied operand. The contents of the explicit operand location are either added to or subtracted from the accumulator:

ADD	m	(A) + (m) → A
SUB	m	(A) - (m) → A

and the contents of m remains unchanged in the process. The other two are register-to-register (see Section 3-9.3) instructions which use source and destination register contents as the operands:

RAD	s, d	(s) + (d) → d	Register ADD
RSV	s, d	(d) - (s) → d	Register SUB

The contents of the source register are added to (subtracted from) the contents of the destination register and the result placed in the destination register. The source register contents are unchanged by this operation.

Registers may be incremented or decremented by 1 using the instructions:

RIN	s,d	(s)+1 → d
RDE	s,d	(s)-1 → d

where the source and destination registers may be the same. For example,

RIN X,X

will increment the X register by one unit.

A number in a register will be replaced by its negative (two's complement) if the *register complement* instruction

RCO s,d $-(s) \rightarrow d$

is used.

3-9.6 A SAMPLE PROGRAM. Let's put some of the operations encountered so far into a simple program to evaluate the expression

$$w = x + y - z$$

Since adds and subtracts may be done in any order, it does not particularly matter how we write the code; except that we will store some of our data values before the first executable program statement to illustrate better what happens during the assembly process:

	IDT	EXPRSN	
A	EQU	0	REGISTER DEFINITION
W	DATA	0	
Z	DATA	>1A	HEX VALUE
EXPRSN	EQU	\$	
	LDA	X	EVALUATE EXPRESSION
	ADD	Y	
	SUB	Z	
	STA	W	SAVE VALUE IN W
	IDL	0	HALT
Y	DATA	-9	DECIMAL
X	DATA	027	OCTAL VALUE
	END	EXPRSN	

If we had included on ORG 12F7 pseudo-op, this program would be assembled and loaded as follows:

Address	Contents	Explanation
12F7	0000	value of W
12F8	001A	value of Z
12F9	0005	load from PC + 5
12FA	2003	Add from PC + 3
12FB	28FC	sub from PC - 4
12FC	80FA	store in PC - 6
12FD	CE00	halt
12FE	FFF7	hex value = -9 ₁₀ = y
12FF	0017	hex value = 27 ₈ = x

3-9.7 IMO AND DMT INSTRUCTIONS. The increment-memory-by-one instruction

IMO m

adds 1 to the contents of the memory specified by m. It is equivalent to

```
LDA    m
ADD    =1
STA    m
```

except that since it is one instruction instead of three, it executes faster and does not require use of the accumulator.

The corresponding decrement-memory-by-one involves the same kind of process except that after decrementing, it performs a test-for-zero in the memory cell. If a zero is found, the program skips the next instruction; otherwise, it executes the next instruction.

These instructions are useful in dealing with loops, where the loop counter is stored in memory and must be tested for the exit condition. Since zero triggers the skip, that is the value we shall select to indicate the termination of looping.

Since we don't know how to multiply yet, we could simulate the process by adding a number to itself. Let's assume we want to compute 4X where X is a number stored in VALUE. A flowchart looks like Figure 3-1.

Since the accumulator is initialized with VALUE, we need only add it in three times. So we'll initialize the counter at 3:

START	LDA	=3	
	STA	COUNT	INITIALIZE COUNTER
	LDA	VALUE	GET VALUE
LOOP	ADD	VALUE	ADD VALUE
	DMT	COUNT	DONE?
	BRU	LOOP	NO, GO BACK
	STA	PRDCT	YES, SAVE PRODUCT
	IDL	0	HALT
COUNT	DATA	0	
VALUE	DATA	> 3F	
PRDCT	DATA	0	
	END	START	

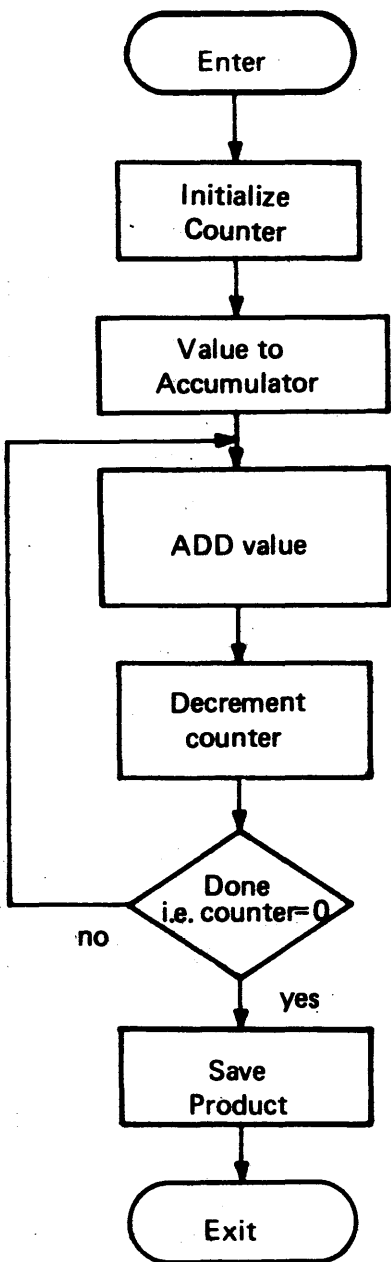
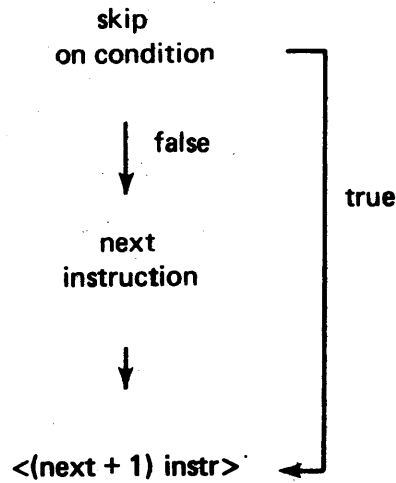


Figure 3-1. Adding a Number to Itself (Multiplication) Flowchart.

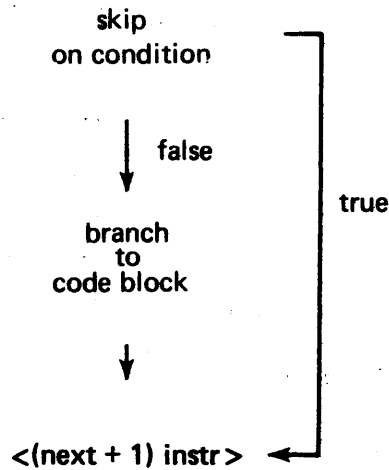
3-10 REGISTER SKIPS AND INDEXED BRANCH.

3-10.1 REGISTER SKIPS. A important part of any programming language is the capacity to test for the presence of conditions and transfer control to whatever portion of the code is appropriate to the condition at hand. The FORTRAN language does this with the logical IF statement; e.g., IF(A.EQ.3.) GO TO 55. Assembly languages generally have several more specialized instructions for either branch-on-condition or skip-on-condition. The 980 has only one branch-on-condition: the BIX instruction. All other conditionals

are the *skip* variety, in which a *true* outcome of a test causes the next instruction* to be skipped and *false* causes it to be executed:



Sometimes it is sufficient to be able to execute only one instruction (i.e., the “next”) before rejoining the mainstream of the program flow. More often though, a programmer will need to execute a block of instructions rather than just one. In this event, it is necessary only to make the “next” instruction an unconditional branch to the desired code block:



*Beware! The assumption here is that each location contains a single instruction. In some cases, two locations are needed to hold one instruction (see *Extended Format*, Section 4-4). When such an extended format instruction is directly preceded by a skip, only the first word of the instruction is skipped, thereby causing execution of data.

Register skips have a mnemonic indication of the test condition in the op-code field and therefore use the name (or number) of the register being tested as the single operand. These instructions are of the form:

<op code> <register>

where <register> can be a register name if the name has been EQU'd to the appropriate number.

For example, if we wish to add the absolute value of the number labeled VALUE to a quantity named SUM, we could do something like this

A	EQU	0	LABEL ACCUMULATOR
	...		
	...		
	LDA	VALUE	GET NUMBER INTO A.
	SPL	A	IS IT POSITIVE?
	[RCO	A,A	NO, CHANGE SIGN OF A.]
	ADD	SUM	YES, ADD IN SUM
	STA	SUM	AND SAVE.

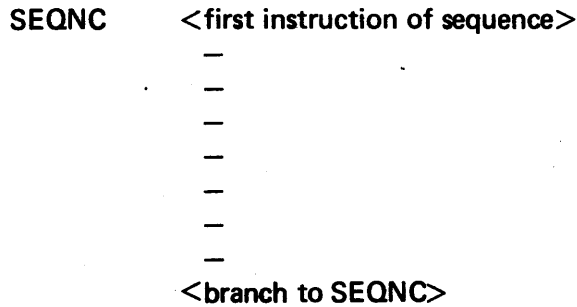
A single instruction [RCO] to complement a register is available, but it is not part of the limited repertoire at our disposal; or we could multiply VALUE by (-1) if we knew how to multiply. Given only the instructions introduced so far, we are forced into the cumbersome alternative of subtracting the negative from zero to convert it to positive:

A	EQU	0	
	LDA	VALUE	GET NUMBER INTO A.
	SPL	A	IS IT POSITIVE?
	BRU	NEGATE	NO, GO TO SIGN CHANGER.
JOIN	ADD	SUM	YES, ADD IN SUM
	STA	SUM	AND SAVE.

in which NEGATE labels another code block:

NEGATE	LDA	= 0	CLEAR A
	SUB	VALUE	SUBTRACT (VALUE)
	BRU	JOIN	BRANCH BACK WITH POSITIVE NUMBER

3-10.2 LOOPS, COUNTERS, AND INDEXED BRANCH. Any instruction sequence which leads directly into at least one repeat of itself is known as a *loop*. The last instruction in the sequence is some kind of branch back to the beginning of the sequence:



If no measures are taken to prevent it, the loop will go on repeating — theoretically forever; thus it is known as an *infinite loop*. Not all infinite loops are truly infinite, since the instructions may involve operating on data in such a way that an overflow error may ultimately terminate execution. Infinite loops have their uses in writing real-time programs or in interrupt techniques. However, for now we'll regard infinite loops as something to be strictly avoided.

To avoid infinite loops, we can build in two kinds of traps:

1. perform the loop a specified number of times, counting each pass and testing the counter between passes to see if the limiting count has been reached.
2. perform the loop until some other program condition of interest occurs: for example, a series approximation carried out term-by-term until the sum (or product) reaches a certain value (or falls within an allowable tolerance). This kind of trap is a bit more dangerous to use than the counter technique described in paragraph 1. above. Convergence may be very slow or (it can happen!) a process the programmer assumed to be convergent is actually divergent. Used with some caution though, the technique is quite acceptable.

In this case the loop struction might resemble

```

SEQNC    <first instruction of sequence>
         -
         -
         -
         -
         -
         <skip if condition met>
         BRU SEQNC
  
```

Traps of the variety described in paragraph 1. above can be managed in two ways: one way involves using the structure described in paragraph 2. above where the condition of the counter is the program condition tested:

```

         <initialize counter>
SEQNC    <first instruction of sequence>
         -
         -
         -
         -
         <increment counter>
         <skip if counter at final value>
         BRU SEQNC
  
```

There is no reason why the process will not work just as well if the counter is run backwards (i.e., initialized at a larger value and decremented until it reaches a predetermined small final value). In fact, it will be more efficient to run the counter backwards on the 980 since the DMT instruction is available. Let's assume that the counter has been prestored in a memory location named COUNT and, provided there is no need to preserve the original value of COUNT, we can decrement the contents of that location with the DMT instruction until it reaches zero and then execute a skip over the BRU.

```

COUNT   DATA <maximum passes through loop>
         -
         -
         -
SEQNC    <first instruction of sequence>
         -
         -
         -
         -
         -
         DMT      COUNT      DECREMENT TEST FOR ZERO
         BRU      SEQNC      NO, BRANCH BACK
         <next instruction>  YES, CONTINUE
  
```

Another way to utilize loops involves using the index register to hold the count instead of a memory cell.

In array problems in which the loop counter and the array pointer assume the same value, the contents of the X register are used for both. The BIX instruction increments the index register and branches upon reaching a zero value. Thus, if this instruction is used, we initialize the X-register with the negative of the desired count and then increase the count until it reaches zero. If we wish to execute a loop twenty times, the structure looks like this:

```

                LDX =-20           INITIALIZE COUNTER
                -
                -
    SEQNC        -
                -
                BIX SEQNC         INCREMENT AND TEST FOR ZERO
    <next instruction >    IF ZERO, CONTINUE

```

3-11 SAMPLE PROGRAM.

A sample program follows which involves adding the counting numbers from 1 to 100 and the result in SUM. Count serves both as a counter and a number to be added.

COUNT	IDT	ADDEM	
SUM	DATA	100	INITIALIZE COUNT
ADDEM	DATA	0	INITIALIZE SUM
LOOP	LDA	= 0	CLEAR ACCUMULATOR
	ADD	COUNT	ADD COUNTING NUMBER
	DMT	COUNT	DECR. COUNT AND TEST. ZERO?
	BRU	LOOP	NO, GO BACK
	STA	SUM	YES, SAVE SUM
	IDL	0	STOP
	END	ADDEM	

3-12 THE INDEX REGISTER AND ITS USE.

3-12.1 HANDLING AN ARRAY: A TYPICAL PROBLEM. Higher level algebraic languages provide standard methods for handling arrays (or *lists*) of data words stored within the computer memory. The 980 AL technique is basically the same: the address of the list (*base address*)* is used along with an incremental pointer (*index*) to indicate which element of the list is to be used. A simple example of an array problem (i.e., adding up all the elements in an array) in two higher level languages provides an introduction to the problem.

In FORTRAN we set aside storage for an array with a DIMENSION statement and then often use the index of a DO loop as a pointer into the array:

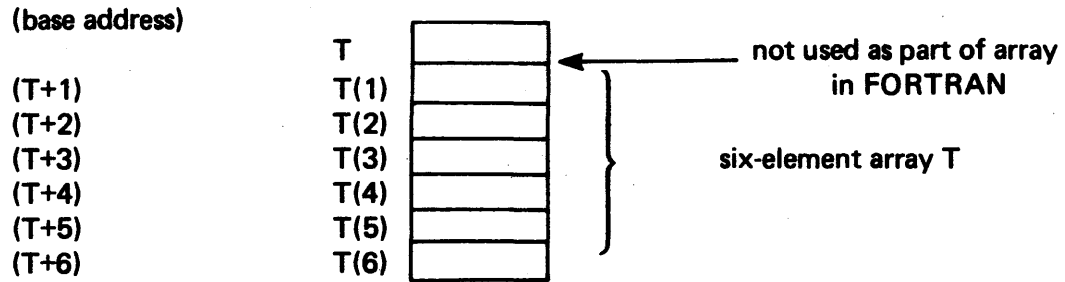
```

                DIMENSION T(6)
                ...
                SUM = 0
                DO 50 I = 1, 6
                SUM = SUM + T (I)
    50  CONTINUE
                ...

```

*The term *base address* of an array as used here is not the same as the base (or B-) register discussed later. However, the base address may be placed in the b-register for convenience in coding some programs. (See Section 4-5.)

where there is a base address (T) to which successive values of the pointer are added to give access to the whole list:



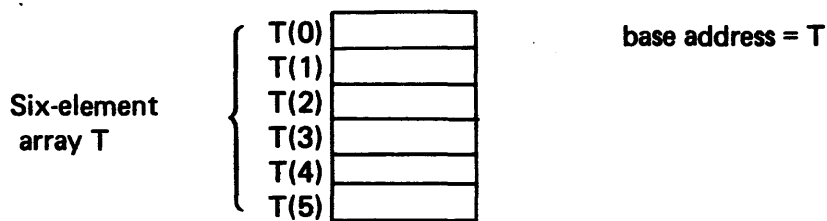
The meaning of T(1) is actually *address of T plus one*.

The FORTRAN compiler accomplishes a number of housekeeping chores that are the responsibility of the programmer in assembly language:

1. Sets up an array of the proper length in a set of memory words chosen so as not to interfere with any other part of the program.
2. Sets up and manages a counter to count the number of passes through the loop and serves as a pointer (or *index*) to the correct element in the list
3. Tests the counter against a predetermined limit with instructions to branch back to the beginning of the loop if the limit has not yet been reached.

Another high level language named BASIC is similar to FORTRAN in many aspects; two notable exceptions are

- a) BASIC permits the base address T to be used as an element of the array.
- b) Indexed loops in BASIC may be incremented or decremented.



Incrementing Loop*

```

DIM T(5)
...
LET S = 0
FOR I = 0 TO 5 STEP 1
LET S = S + T(I)
NEXT I
...

```

Decrementing Loop†

```

DIM T (5)
...
LET S = 0
FOR I = 5 to 0 STEP-1
LET S = S + T(I)
NEXT I
...

```

*Adds elements in the order T(0) first and T(5) last
†Adds elements in the order T(5) first and T(0) last

We can write several slightly different 980 AL equivalents for these fragments. All involve loading the index (handled by the loop control statements in FORTRAN or BASIC) into the 980 index (X) register and arranging in our 980 AL program to change its value at the proper time.

3-12.2 THE INDEX REGISTER (X). To add the contents of T + 4 into the accumulator, we have a 4 in the index register

(X)

=0004

and write

ADD T, X

where the name of the index register (X, if we have EQU'd it to register 2) is used in the second operand (or "modifier") field.

This instruction assembles to

22dd

 (dd are the displacement digits)

which instructs the computer to:

1. Examine the X-register to determine how big the index (or *offset*) is – in this case: 4.
2. Add the offset found to the operand (*base*) address – i.e., add PC to displacement digits (dd) – to compute a quantity known as the *effective address*. In this case the effective address is the address of cell T + 4.
3. Perform the operation; in this case, an ADD to the accumulator using the *contents* of the effective address.

3-12.3 SETTING UP THE DATA FOR AN ARRAY PROBLEM. As was hinted at in Section 3-12.1, there are a number of ways we could set up the array data, and how we manipulate the number in the index-register will be determined by the data set-up. Next we will look at a number of possible data setups and the code that would be used to add the array numbers.

3-12.3.1 INCREMENTING INDEX TECHNIQUES. Given the array:

base address T	n1	index
	n2	0
	n3	1
	n4	2
	n5	3
	n6	4
		5

We could initialize X at zero, increment it, and test it for having reached 5:

	IDT	ARRAY 1	
A	EQU	0	REGISTER DEFINITION
X	EQU	2	REGISTER DEFINITION
M	EQU	5	REGISTER DEFINITION
T	DATA	n ₁ , n ₂ , n ₃ , n ₄ , n ₅ , n ₆ ,	
ARRAY 1	EQU	\$	PROGRAM ENTRY POINT
	LDX	= 0	INITIALIZE INDEX
	LDA	= 0	INITIALIZE SUM
LOOP	LDM	= 5	SET INDEX LIMIT
	ADD	T, X	ADD NEXT ARRAY ELEMENT
	RSU	X, M	TEST INDEX VALUE
	SNZ	M	IS IT EQUAL TO LIMIT?
	BRU	STOP	YES, SAVE SUM
	RIN	X, X	NO, INCREMENT INDEX
	BRU	LOOP	AND GO BACK
STOP	STA	SUM	
SUM	IDL	0	
	DATA	0	
	END	ARRAY 1	

This method is a bit long and cumbersome but does the job.

If we instead considered our *effective base address* to be cell T + 6 (i.e., the cell after the last data value), we could put a negative index in the X-register and decrement it to zero, allowing us to use the BIX instruction:

	IDT	ARRAY 2	
A	EQU	0	
X	EQU	2	
T	DATA	n ₁ , n ₂ , n ₃ , n ₄ , n ₅ , n ₆	
ARRAY 2	EQU	\$	
	LDX	= -6	INITIALIZE INDEX
	LDA	= 0	INITIALIZE SUM
LOOP	ADD	T+6,X	ADD NEXT ARRAY ELEMENT
	BIX	LOOP	IF NOT DONE, INCREMENT AND GO BACK
	STA	SUM	OTHERWISE, SAVE SUM
	IDL	0	
SUM	DATA	0	
	END	ARRAY 2	

which is considerably shorter. We could avoid using address arithmetic (the expression T + 6) in the ADD instruction if we observed that the location T + 6 is really the same as the location ARRAY2 (the LDX instruction). Thus, the ADD instruction could have been written

LOOP	ADD	ARRAY 2,X
------	-----	-----------

3-12.3.2 DECREMENTING INDEX TECHNIQUE. In some array problems (a simple addition problem is not one of them) it may be necessary to work backwards through the array; we could do it in a way similar to the first example:

	IDT	ARRAY 3	
A	EQU	0	
X	EQU	2	
T	DATA	n ₁ , n ₂ , n ₃ , n ₄ , n ₆	
ARRAY 3	LDX	= 5	INITIALIZE INDEX
	LDA	= 0	INITIALIZE SUM.
LOOP	ADD	T,X	ADD NEXT ARRAY ELEMENT.
	SNZ	X	HAS X BECOME ZERO?
	BRU	STOP	YES, SAVE SUM
	RDE	X,X	NO, DECREMENT INDEX,
	BRU	LOOP	AND GO BACK.
STOP	STA	SUM	
	IDL	0	
SUM	DATA	0	
	END	ARRAY 3	

These are, of course, not the only ways to do it.

3-12.4 ADDRESS ARITHMETIC. The 980 permits the programmer use of arithmetic expressions as operands, pursuant to certain conditions. The four operators

- + addition
- subtraction
- * multiplication
- / division

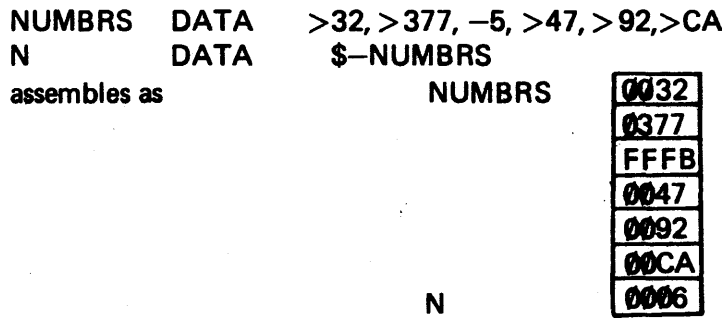
are permitted, but multiplication and division by a label are prohibited in relocatable (nonabsolute)* assemblies. Expressions are evaluated (from left to right) using normal operator procedure (* and / on first scan, + and - on second scan). All symbolic labels must, of course, be defined. Examples of address arithmetic are

<u>Address Expression</u>	<u>Meaning</u>
\$+077	77 cells below the present location
ARRAY-6	6 cells back from the cell named ARRAY
X + > 2A	the absolute address 2C ₁₆ (if X EQU'd to 2)
JOE+TOM*3/BOB	$\frac{3 X \langle \text{address of TOM} \rangle}{\langle \text{address of BOB} \rangle} + \langle \text{address of JOE} \rangle$

Complicated expressions are seldom useful to the programmer.

*Absolute assemblies are those which include the ORG <absolute address> directive.

One useful application involves having the assembler count the number of words appearing as DATA:



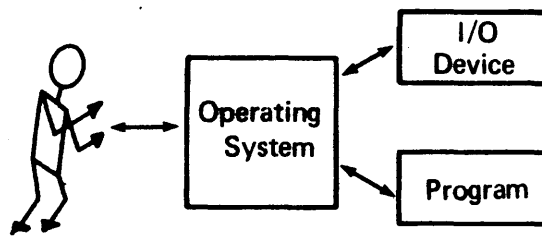
3-13 THE BARE MACHINE vs. THE BASIC OPERATING SYSTEM.

It seems to be true in life that any labor-saving device introduces some complexity into the environment in which it is used. The device saves work in that it assumes tasks which must otherwise be done manually: but in so doing, removes the operator to one level more remote than if he did the job himself. Although it is not necessary that the operator understand intimate engineering details of the device, he should know

- In general how the device should perform
- How to make the device do the job he wants done.

A special purpose device (devoted to doing one job one way) can have a very simple communication procedure (i.e., throwing a switch to turn it on or off – causing the device to do its thing or stop doing it). A multipurpose device must be told what job to do and be informed of the conditions associated with the performance of the task. One can immediately envision a number of tools or appliances which illustrate these facts.

The same sort of thing obtains in a computer environment. The operator can approach a bare machine and tell it how to do each job he wants done, or he can relegate some of the routine operations (such as performing an assembly or execution or causing I/O to happen) to a software utility program called an *operating system*. He must be able to communicate with the operating system in order to tell it what job to do and under what conditions.



This is a very crude simplification, but it makes one think along the right lines.

Operating systems generally insist on their rights, too. It is as if the operating system said; "okay, user, you want me to work for you, but I'm going to do the job the way I was designed to do it and you're not going to meddle. If you don't like that, do it yourself (on a bare machine)." Hence, we are led to the idea of "privileged instructions", which are instructions prohibited to us if we use the operating system. If we try to use them privileged instructions will lead to errors. In other words, we cannot both have our cake and eat it too.

In the 980, instructions associated with I/O, instructions which modify the status register, and the IDL (idle) instruction are privileged when running under the control of the operating system. The operating system is given charge of I/O operations and maintenance of the status register. A program being executed is not really allowed to throw the machine into the idle state when it is finished: it must return control to the operating system which then informs the user, "I am through with that job. What shall I do next?"* Later we will discuss how I/O and program halt are executed legally using a bare 980. We also will investigate how to give these functions to the operating system and how to communicate with that system.

3-14 USE OF THE BASIC OPERATING SYSTEM.

If the basic operating system is resident, switching on power will initialize the system so that when START is hit, the control keyboard/printer will output the message *READY*. The operator then makes the assignments of *logical-unit-numbers* (luns)* to the peripheral units he intends to use. Luns are assigned to devices through use of the //ASSIGN commands to the supervisor.

Let us assume a configuration comprising a 980, a card reader (CR), a line printer (LP), and a Silent 700 keyboard (KEY) device with two cassette units (CS1, CS2).

The SAPG assembler assumes the following lun assignment

- lun 4: Operator communication device
- lun 5: Source input device
- lun 6: Source listing device
- lun 7: Device on which object will be written
- lun 10: Scratch

We type each of the following command lines, terminated by a carriage return.

//ASSIGN,4,KEY.	(control from keyboard)
//ASSIGN,5,CR.	source input from card reader
//ASSIGN,6,LP.	any listing on line printer
//ASSIGN,7,CS2.	object written onto cassette 2
//ASSIGN,10,DUMMY.	assign as dummy (DUMMY or DUM) or else assign to rewindable device to use for interpass storage. We can assign 10 to CS1 and only put the cards through once, or we can use the dummy assignment and load the cards twice.
 	to load assembler
//EXECUTE,CR.	

The keyboard/printer responds with READY SOURCE, HIT CR. Make sure the source device is on-line, loaded, and ready: then HIT CR.

After that pass the keyboard/printer types READY SOURCE, HIT CR. Make sure the source and assembly listing devices are on-line, ready, and (in the case of the source) loaded: hit CR.

The assembly listing should appear on the high speed printer and the object should be written on the cassette tape. Run a /* card through (by pressing CR) to

1. Terminate assembly and return control to supervisor
2. Write an end-of-file and the entry point on the object file.

*The same thing happens in various versions of FORTRAN run under control of an operating system. Some versions of the IBM 7090 FORTRAN prohibit use of the STOP statement. Instead one writes CALL EXIT in which the system subroutine named EXIT returns control to the operating system. Other versions of FORTRAN allow the inclusion of the STOP instruction, but this in turn triggers a call to the EXIT routine.

* Logical-unit-numbers are sometimes abbreviated as LUNS

Assignments need not be changed (line may be reassigned to dummy if not used by the program, but may be left; the object device is automatically rewound, if possible):

```
//EXECUTE,<name of object device>.*  
(e.g.,//EXECUTE,CS2.)
```

will cause the object program on CS2 to load and begin executing.)

```
//EXECUTE. will begin executing the last program loaded into memory
```

In this mode of operation, any I/O done by the program must be handled by service calls to the supervisor (this is discussed in the next section). Control may be taken from the program and returned to the supervisor at any time by setting the following sequence of switches on the 980 front panel:

```
HALT  
RESET  
RUN  
START
```

Cold-start procedures are described in Appendix H.

*If the name of the object device is omitted:

```
//EXECUTE.
```

the object program already loaded into memory will execute.

SECTION 4

TI MODEL 980 ADDRESSING MODES AND THE STATUS REGISTER

4-1 SUMMARY OF MODEL 980 CHARACTERISTICS.

The TI980 is a general purpose minicomputer with the following characteristics:

Word length: 16 bits

Arithmetic mode: two's complement

Addresses and instructions: hexadecimal

Memory size:

Assembly language: 980 AL

Assembler: symbolic assembly program (SAPG)

Priority interrupt: optional

Internal Timer; optional

Peripherals: four low speed on data bus (optionally expandable to 256)
one high speed on DMAC (optionally expandable to eight)

Registers (nine): accumulator (A)
extension arithmetic (E)
index register (X)
maintenance (M)
storage (S)
link (L)
base (B)
program counter (P) or (PC)
program status

Addressing modes:

absolute	}	either program counter (P) relative or base register (B-) relative
immediate		
relative		
indexed		
indirect		
indexed/indirect		
indirect/indexed		

Table 4-1. 980 AL Subset

The instructions and addressing modes discussed in this section are listed in Table 4-1.

Instructions:

SOV	Skip on status register	{ overflow no overflow carry no carry
SNV		
SOC		
SNC		
RCA	Register compare - algebraic	
RCL	Register compare - logical	
CPA	Compare A with storage - algebraic	
CPL	Compare A with storage - logical	
SLT	Skip on status register	{ less than less than / equal equal greater than/equal greater than not equal
SLE		
SEQ		
SGE		
SGT		
SNE		
Extended format instructions		

Assembler Directives:

DATA	repeat from Section 3
ORG	
BSS	Block starting symbol
BES	Block ending symbol
BRS	Base register set
BRR	Base register reset
HED	Page heading
PEJ	Page eject
LIS	List
UNL	Unlist

Addressing Modes:

- Indirect.
- B-Relative

4-2 EFFECTIVE ADDRESS.

Memory-referencing instructions – load, store, ADD, SUB, IMO, DMT – actually use an *effective address*. (e.a.) which is computed from information contained in the operand and modifier fields:

$$\langle \text{op code} \rangle \ m, \langle \text{mod} \rangle$$

where the modifier can be X (for indexed) EQU'd to 2
 BR (for base register relative) EQU'd to 1
 XB for indexed, base register relative EQU'd to 3.

Of course, the digits themselves may be used in the <mod> field.

4-3 ADDRESS MODES: IMMEDIATE; INDIRECT AND INDEXED P-RELATIVE.

The following two address modes are indicated by prefixing a special symbol to the operand:

<u>mode</u>	<u>prefix</u>
immediate	=
indirect	*

The other modes are indicated by use of the <mod>ifier field, as listed in Section 4-2. Here are some concrete examples:

Mode	Example	Action
immediate	LDA = -9	Places the hexadecimal equivalent of -9_{10} in the accumulator
P-relative	STA ANSWER	Stores the contents of the accumulator in the location labeled ANSWER (effective address, location labeled ANSWER)
P-relative, indirect	STA *POINTER	Stores the contents of the accumulator in the effective address; i.e., the cell whose address is the number found in POINTR. In this case the effective address is MEMLOC. Places value 3 in the index register, contents of effective address, BUF + 3 is placed in the accumulator
POINTER	DATA MEMLOC	
X Immediate	EQU 2 LDX = 3	
P-relative, indexed	LDA BUF, X	

Both indexed and indirect modes may be used together in a *pre-indexed* (indexed, then indirect) or *post-indexed* (indirect, then indexed) form. The assembly language expression of these two forms is the same; the order in which the two modes are applied depends on the setting of the *index control bit* (bit 10 of the status register). If that bit is 1, pre-indexing takes place (index before indirect); if the bit is 0, post-indexing (index after indirect) occurs. This bit must be set by the programmer to indicate which order is to be observed.

To pre-index program:

```
<set status10 to 1>
X EQU 2
LDA *ALFA,X
```

Action:

1. compute the effective address of ALFA + (X).
2. LDA with the number pointed to, i.e., (ALFA + (X))

To post-index program:

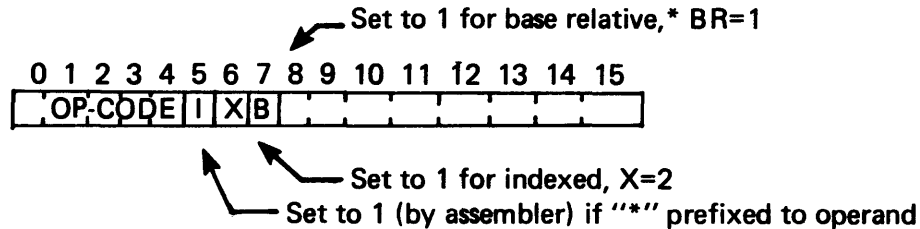
```
< set status10 to 0 >
X EQU 2
LDA *ALFA,X
```

Action:

1. find the number pointed to by ALFA, i.e., (ALFA)
2. ADD (X) to get the effective address, i.e., (ALFA) + (X)

All these instructions assemble with a positive or negative displacement relative to the PC and, therefore, are regarded as P-relative modes. This means that within the machine word, the low order 8 bits represent a count that must be added to the PC in order to find the address of the operand.

In Register-to-memory format instructions, the first 5 bits contain the operation code, and the following 3 bits must be set, depending upon the mode of the operand:



All immediate symbols (=) prefixed to the operand set all 3(I,X,B) bits to 1.

4-4 NORMAL AND EXTENDED FORMAT.

If the B bit is not set, the assembler assumes that addressing is to be done relative to the program counter. Bits 8 through 15 of the instruction contain the offset of the operand (the *displacement*) relative to the current value of the PC. (The PC points to the instruction following the instruction currently being executed.) Since this displacement field is only 8 bits long, the operand must fall within the range

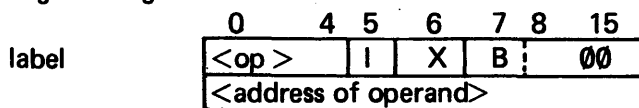
$$(PC) + 80_{16} \leq \text{operand address} \leq (PC) + 7F_{16}$$

where 80_{16} (-128_{10}) is the largest negative number which can be expressed, and $7F_{16}$ ($+127_{10}$) is the largest positive number.

One way to reference an operand outside this range, is to use what is called *extended format*. The programmer must remain aware of this condition and prefix the operation code with the symbol @ when extended format is to be used:

<label> @ <op-code> <operand>, <modifier>

This prefix signals the assembler to translate the instruction into two words of machine code, where the displacement field contains zero and the next word contains the hexadecimal address of the operand relative to the program origin:



In the example:

```

                @LDA    MATCH
                ....
                ....
                ORG    >0121
(location 0121) MATCH  DATA  <data value>
    
```

*We shall defer until later the discussion of the B-relative modes; e.g.,

```

LDX BETA,1 or BR EQU 1
LDX BETA,BR
    
```


the @LDA MATCH instruction will be assembled as

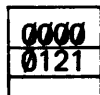


and will execute by placing <data value> into the accumulator.

```

If we write          @LDA    =MATCH
                    .
                    .
                    .
                    .
MATCH               ORG      >121
                   DATA    <data value>
  
```

this assembles as



and will execute by placing 0121 into the accumulator. In this case, the machine sees a zero offset to the PC and loads the contents of the second word; i.e., (PC) + 0.

In the earlier case, the indirect bit is set by the @ prefix and the execution proceeds by using the contents of the PC as a pointer to the data value.

Since the combination of relative addressing and extended format actually leads to an execution which uses an indirect address, the indirect bit is set by the assembler and, therefore, is not available to the programmer. This means that the form

```
@LDA *MATCH
```

is not available.

Extended format instructions may be used in conjunction with indexed operands; for example:

```

X                   EQU      2
                   @LDA    ARRAY,X
                   ...
                   ...
ARRAY              ORG      >121
                   DATA    <data value>
  
```

assembles with both the indirect and index bits set:



and executes with the address at (PC) + 0 (i.e., the address 12110) used as the base address to which the index register contents are added. Since the base address is obtained indirectly before the offset is added, the

post-indexing condition always applies, regardless of the setting of the index control bit (bit 10) of the status register.

Since extended format instructions occupy two words, caution must be used when writing skip instructions directly preceding them; the skip will skip only the first word rather than the entire instruction word-pair.

4-5 THE BASE* REGISTER.

The addressing modes considered so far (other than immediate) are

- P-relative – assembled displacement plus execution time value of the program counter give the effective address
- Indexed P-relative – effective address computed from the P-relative operand address plus the contents of the index-register.

Since the maximum displacement size is 8 bits (including sign), P-relative addressing is restricted to cells closer to the PC than 128₁₀ locations on the negative side and 127₁₀ locations on the positive side.

This restriction can always be overcome by using extended format instructions which reserve an extra word each in order to hold the address; however, if memory space becomes a critical resource in a given problem, the extended format solution may not be the best one. Using the Base register affords us another option.

4-5.1 BASE REGISTER RELATIVE ASSEMBLY AND EXECUTION. If the PC-relative displacement computed by the assembler exceeds the allowable range, the assembler generates a field-size error message unless the assembly is being performed under control of a Base register set directive:

```
BRS    <address of new Base>
```

This directive continues in effect until cancelled by a Base register reset directive:

```
BRR
```

The operand in the BRS directive gives the value related to which the displacements will be computed. These are 8-bit unsigned displacements which must range from 0 to 255₁₀. The program must contain an executable instruction for loading the Base register with the appropriate Base address at execution time.

Let's examine the specific problem of accessing values which exceed the allowable (positive) PC-displacement range. Field-size assembly errors are produced by the following fragment:

```
                LDA    VALUE
                ADD    VALUE +1
                -
                -
                ORG    $+300
VALUE          DATA  n1, n2
```

*We have already discussed the *base address* of an array, which is distinct from the B- or *base* register. This distinction is maintained in the text through the use of the lowercase symbol "base" in reference to an array address and "Base" meaning the contents of the B-register.

unless the BRS option has been specified:

A	EQU	0	
B	EQU	6	SPECIFY ASSEMBLY TIME
	BRS	VALUE	BASE REGISTER VALUE
	@LDA	= VALUE	SPECIFY EXECUTION TIME BASE VALUE
	RMO	A,B	AND MOVE TO BASE
	—		
	—		
	LDA	VALUE	DISPLACEMENT 0, RELATIVE TO B = VALUE
	ADD	VALUE +1	DISPLACEMENT 1, RELATIVE TO B = VALUE
	—		
	—		
	BRR		
	STA	ALPHA	ALPHA MUST BE WITHIN PC RANGE
	ORG	\$+300	
VALUE	DATA	n ₁ , n ₂	

4-5.2 BASE REGISTER USED AT EXECUTION TIME ONLY. The Base register is available for executing programs or program fragments which have not been assembled under BRS control (i.e., normal assembly). In this execution-only context, it is used by those instructions shown here with the letters "BR" in the <mod> field. The first operand of these instructions must assemble to an 8-bit unsigned number between 0 and 255₁₀ that will serve as a displacement relative to the execution time value in the B-register.

Recounting the previous example to show the format and context of the use of the B modifier:

BR	EQU	1	
	@LDA	=VALUE	GET ADDRESS OF EXECUTION TIME BASE VALUE
	RMO	A,B	MOVE TO BASE REGISTER
	—		
	—		
	LDA	0, BR	DISPLACEMENT 0, RELATIVE TO B =VALUE
	ADD	1, BR	DISPLACEMENT 1, RELATIVE TO B =VALUE
	—		
	—		
	—		
	STA	ALPHA	ALPHA IN PC RANGE
	ORG	\$+300	
VALUE	DATA	n ₁ , n ₂	

In this case the base address of the array VALUE is actually the value appearing in the Base register, and the offset appears as an absolute quantity in the operand field.

Although a mode of addressing is available that makes simultaneous use of the B-and X-registers (indexed Base Relative), discussion of that mode is deferred until Section 6. If we want to handle an array problem in pure B-relative mode, we can use an offset that is always 0 and increment (or decrement) the B-register in order to access all the array members. For example, to add an array (ARRAY) of six numbers, assuming that the index (X) register is not available:

	A	EQU 0	
	B	EQU 6	
	BR	EQU 1	
KNTR	DATA	0	
ARRAY	DATA	n ₁ , n ₂ , n ₃ , n ₄ , n ₅ , n ₆	
	@LDA	=ARRAY	SET ADDRESS OF ARRAY
	RMO	A,B	IN BASE REGISTER
	LDA	=6	SET COUNT
	STA	KNTR	IN MEMORY
	LDA	=0	INITIALIZE SUM
	ADD	0, BR	ADD NEXT ARRAY ELEMENT
	RIN	B,B	INCREMENT BASE VALUE
	DMT	KNTR	DONE?
	BRU	\$-3	NO, GO BACK

Here the base address of the array and the Base contained in the register are initially the same.

4-6 THE ORIGIN DIRECTIVE.

Although this pseudo-op was discussed previously in Section 3-6, it is mentioned again here

```
ORG <expression>
```

where <expression> may be absolute;

```
e.g. ORG >2000
```

in which case the assembler produces code tied to absolute machine addresses; or the expression may be such that relocatable code results. A typical relocatable expression might be

```
ORG $+500
```

in which \$ means *the present location* (i.e., pointed to by the location counter) so that the net result of the above expression is to skip the next 500 words before continuing the assembly. This provides one means of reserving storage areas.

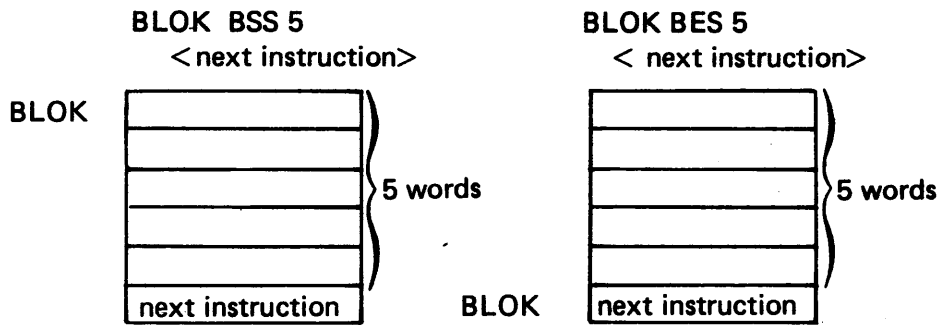
4-6.1 RESERVING STORAGE. In addition to the possible use of the ORG directive discussed above, storage blocks may be reserved and labeled using the following:

```
<label> Block Starting Symbol -  
BSS <number>
```

which reserves a <number> long block of memory words and labels the first word with <label>.

```
<label> Block Ending Symbol -  
BES <number>
```

which reserves a <number> long block of memory and labels the word *following the last word* with <label>. This declarative is particularly useful in reserving space for arrays which use negative offsets and a BIX (see Section 3-10.2). For example:



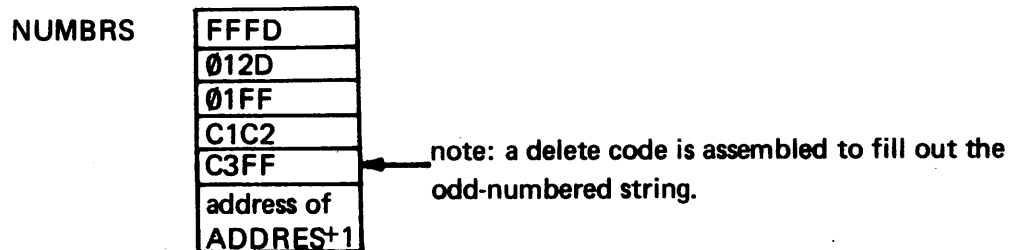
4-6.2 DATA DECLARATION. A data list is assembled into consecutive words by the pseudo-op

```
<label> DATA <list>
```

where <list> contains expressions separated by commas. These expressions may be in decimal, hexadecimal (shown by prefix ">"), octal (shown by prefix of zero), or ASCII (shown by enclosure in single quote marks). For example,

```
NUMBRS DATA -3,>12D,0777,'ABC',ADDRESS+1
```

assembles as



The last cell is an example of our ability to specify a label memory location and to have the assembler record the actual address of that location. This facility is especially useful when we generate relocatable code and do not wish to restrict ourselves to any particular set (actual or relative) of memory locations.

The example

```
HERE DATA HERE
```

takes whatever address appears in the location counter and stores it in the word.

For instance if we write

```
ORG >1500
HERE DATA HERE
```

HERE is the label of cell 1500, and the assembler stores 1500 in the word, as well*

1500	1500
1501	
1502	

One useful application is that in which the assembler generates a word count so that the programmer need not manually count (and therefore risk miscounting). Consider an array in which we have label the beginning and ending word

```
ARRAY DATA 3,4,5,6,7,8,9,-12,22
```

these assemble as

ARRAY	3	(1500)
	4	
	5	
	6	
	7	
	8	
	9	
	-12	
	22	
COUNT	9	(1509)

..and we could count manually (there are nine) and store 9 in count as shown:

```
COUNT DATA 9
```

or, we can have the assembler calculate the count for us:

```
ARRAY DATA 3,4,5,6,7,8,9,-12,22
COUNT DATA COUNT-ARRAY
```

If ARRAY is assigned to the locations starting at 1500 and COUNT follows (falling into 1509), we can generate the count by subtracting the two addresses COUNT-ARRAY (=9) and store that value in the location named COUNT (1509).

*HERE DATA \$ produces the same result

4-6.3 EQUATE. This declarative also was introduced earlier (Section 3-6).

`<SYMBOL> EQU <address>`

permits use of a symbol interchangeably with a number equal to `<address>`. Instructions requiring the use of a register as an operand (the skip-on-zero instruction, for example) may be written as

	<code>SIZE</code>	<code>3</code>	<code>SKIP IF (M) REGISTER (+) = ZERO</code>
		<code>or</code>	
<code>M</code>	<code>EQU</code>	<code>3</code>	<code>DEFINE REG 3 TO BE M</code>
	<code>...</code>		
	<code>SIZE</code>	<code>M</code>	<code>SKIP IF (M) REGISTER = 0</code>

Register definition is not the only use of EQU. Immediate data may be defined the same way – for example, the hexadecimal code for a line-feed/carriage-return pair

`LF CR EQU >0D0A`

which labels the existing hex address with the symbolic name. No memory space is used by the EQU. A common technique for defining the program entry point involves the use of EQU:

`<entry point name> EQU $`

which labels the next machine word with the entry point name.

4-6.4 PRINT CONTROL DIRECTIVES. The directive

`HED <string>`

labels the top of each page with a header identical to `<string>`.

The directive

`PEJ`

is a page-eject instruction.

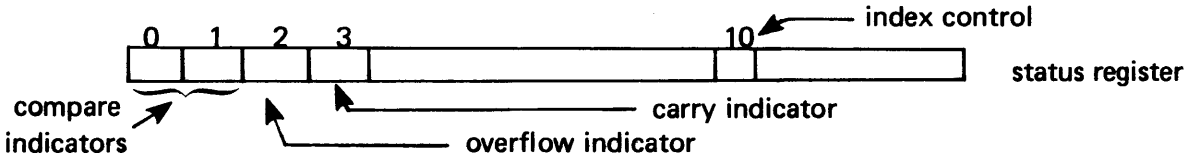
The directives UNL and LIS suppress and restore, respectively, printout of the assembly listing.

4-7 THE STATUS REGISTER: OVERFLOW, CARRY, AND COMPARE.

The first 4 bits of the status register are set by the CPU to reflect certain conditions which arose during the execution of instruction. These bits may be tested under program control for whatever conditions are recorded. The first 2 bits are compare indicators which are set as the result of a compare instruction in the program. The overflow bit is set by an overflow operation (e.g., an integer larger than 16 bits) and the carry bit is set by an add or subtract instruction that results in a carry into the sign bit of a register.

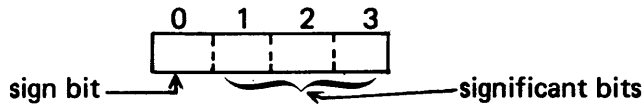
Any instruction empowered to set the carry or overflow bits, should the condition arise, will clear the bit (regardless of the present setting) if the condition does not arise.

In all cases, if the condition of the bit is of interest, it is wise to perform the test immediately and mandatory to perform it before the next operation which could affect the setting of the bit:



- 00 → less than
- 01 → equal to
- 10 → greater than
- 11 → unused

4-7.1 CARRY AND OVERFLOW. Now that we know what a 16-bit arithmetic quantity looks like, let's discuss the meaning of the concepts of carry and overflow. For convenient discussion, and to make our numbers more manageable, imagine for a moment we have a computer with a 4-bit word that uses two's complement arithmetic. This would mean that we have 3 bits of significance and a sign bit



The largest positive number we could hold in this register* is +7 ($=2^3-1$):



and the largest negative number is -8:



Adding (-1) + (-2) generates a carry:

$$\begin{array}{r}
 \text{(1) carry} \\
 1111 \quad (-1) \\
 + 1110 \quad (-2) \\
 \hline
 1101
 \end{array}$$

It appears to become a negative number because we have a carry generated which, by all the rules of arithmetic, rightfully keeps propagating toward the left until it finds a suitable resting place. In this case, the resting place happens to be the sign bit rather than a significant bit.

*The largest positive number is always given by 2^n-1 , where n is the number of bits of significance available (in this case, 3); the largest magnitude negative number in two's complement is $-(2^n)$.

Many computers, including the 980, have an indicator called the carry indicator which can be used to warn the operator when this condition occurs. Thus, a carry condition arises from the carry of 1 into the sign bit (i.e., from bit 1 into bit 0) as the result of an arithmetic operation.

The overflow condition occurs when the result of an operation (regardless of what happened to the sign bit) has too great a magnitude to fit into the available word length. For example, let's add -8 and -8, which should give the result -16.

$$\begin{array}{r} 1000 \\ + 1000 \\ \hline \times 0000 \end{array}$$

In this case the overflow indicator is set, and the carry is clear.

4-7.2 PROGRAM TO TEST CARRY AND OVERFLOW BEHAVIOR. The program shown in the box in Figure 4-1 was used to test the carry and overflow behavior of the 980. The data NUM1 and NUM2 are added and the overflow and carry indicators tested. If an overflow is found, a "1" is saved in location OVFL0; if a carry is found, a "1" is stored in location CARRY. During the first run, the numbers added are 32767 and 1, which produce both carry and overflow. The information outside the box was generated by the SAPG assembler.

		0001				SHEET 0001
	0000	0002	TSTCO	IDT	TSTCO	
	0700	0003		EQU	\$	
0000	8008	0004		LDA	=0	
0001	8008	0005		STA	CARRY	CLEAR CARRY AND
0002	0F01	0006		STA	OVFLO	OVERFLOW FLAGS.
0003	0006	0007		LDE	=1	SET FLAG IN E
0004	2006	0008		LDA	NUM1	ADD THE TWO NUMBERS.
0005	CFE0	0009		ADD	NUM	
0006	8805	0010		SNC		CARRY?
0007	CDE0	0011		STE	CARRY	YES, SET CARRY FLAG
0008	8804	0012		SNV		OVERFLOW?
0009	78FF	0013		STE	OVFLO	YES, SET OVERFLOW
000A	7FFF	0014	NUM1	BRU	\$	FLAG.
000B	0001	0015	NUM2	DATA	32767	
000C	0000	0016	CARRY	DATA	1	
000D	0000	0017	OVFLO	DATA	0	
000E	0000	0018		DATA	0	
	0000			END	TSTCO	
CARRY 000D	NUM1 000B	NUM2 000C		OVFLO 000E		SHEET 0002
TSTCO 0000						
0000 ERRORS						

Figure 4-1. Carry and Overflow Test Program

At the end of execution, the program hangs at the BRU \$ instruction in location A. If the MODE switch is set to HALT, memory locations containing the results can be examined:

To EXAMINE the contents of memory:

1. Enter address into memory address register (MAR):
 set address 000D on switches
 lift the MA switch. MA (↑)

2. Display contents of that memory cell:
 depress MD* switch MD (↓)
 and read contents from the lamps.

3. Display contents of the accumulator:
 depress the A switch. A (↓)

To DEPOSIT new numbers in memory:

1. Enter address into MAR:
 address on switches
 lift MA switch MA (↑)

2. Enter new contents of that cell
 new number on switches
 lift MD‡ switch MD (↑)

To RUN the program with the data just deposited:

1. Enter the entry point address (0000) into the PC:
 address 0000 on switches
 lift PC switch. PC (↑)

2. Set the MODE switch RUN.

3. Depress the START switch.

Here are some examples as handled in the 16-bit word of the 980.

7FFF	(+32767)	carry = set
<u>+0001</u>	<u>+(+ 1)</u>	overflow = set
8000	(-32768)	

7FFF	(+32767)	carry = set
<u>+7FFF</u>	<u>+(+32767)</u>	overflow = set
FFFE	(-2)	

*Depressing the MDI switch, instead, will display the contents and step the MAR by 1 so that successive locations may be examined.

‡Lifting the MDI switch, instead, will enter the number set on the switches and step the MAR by 1 so that successive locations may be deposited in memory.

8000	(-32768)	carry = clear
+ FFFF	+ (-1)	overflow = set
7FFF	(+32767)	
8000	(-32768)	carry = clear
+ 8000	(-32768)	overflow = set
0000	(0)	
8000	(-32768)	carry = clear
+ 7FFF	+ (+32767)	overflow = set
FFFF	(-1)	
8001	(-32767)	carry = set
+ 7FFF	+ (+32767)	overflow = clear
0000	(0)	
FFFF	(-1)	carry = set
+ FFFE	+ (-2)	overflow = clear
FFFD	(-3)	

4-7.3 THE OVERFLOW AND CARRY TESTS. These instructions always test the status register and do not use an operand:

Overflow

SOV	Skip if overflow
SNV	Skip if no overflow

Carry

SOC	Skip on Carry
SNC	Skip if no Carry

4-7.4 THE COMPARE INSTRUCTIONS. Before the compare indicators can be tested, they must be set. The one job (and the only job) of the compare instructions to set these bits. Both logical and algebraic compares are available. Algebraic compares consist of considering the 16-bit comparands to be signed numbers (15 significant bits plus sign). Logical compares regard the comparands as having 16-bits of significance and as positive (i.e., unsigned). For example, the numbers 0001 and FFFF would be in the following relation:

algebraic	0001 > FFFF	(+1 > -1)
logical	FFFF > 0001	(65535 > 1)

The 980 permits the following comparisons:

a. Register with Register

• algebraic:	RCA	s,d	(s) < (d) ⇒ set	"<" on compare indicators
			(s) = (d) ⇒ set	"=" on compare indicators
			(s) > (d) ⇒ set	">" on compare indicators

- logical: RCL s,d same operation, except on 16-bit unsigned quantities

b. Memory with Accumulator

- algebraic: CPA m,<mod> (e.a.) <(A) ⇒ set "<" on compare indicators
(e.a.) =(A) ⇒ set "=" on compare indicators
(e.a.) >(A) ⇒ set ">" on compare indicators

If the <mod> bits are all ones (i.e., IXB = 7), the compare is immediate and uses the low-order 8 bits of the CPA instruction sign extended in a 16-bit comparison with the accumulator. For example,

CPA =3

compares the immediate value 0003 with the current contents of the accumulator.

- logical: CPL m,<mod> same operation except on unsigned numbers

If the <mod> bits are not all ones, the comparison uses all 16 bits of the comparands. If the <mod> bits are all ones (i.e., immediate operand), the comparison is between the low-order 8 bits only of the respective comparands. For example,

CPL m,X

compares the 16 bits in m + (X) to the accumulator; whereas,

CPL =-2

compares the value FE with the low-order 8 bits of the accumulator.

4.7.5 TESTING THE COMPARE INDICATORS. Several skip instructions are provided, requiring no operand, which test the compare indicators and execute a skip exit if the condition tested for is met:

SLT	Skip on less than
SLE	Skip on less than or equal
SEQ	Skip on equal
SGE	Skip on greater than or equal
SGT	Skip on greater than
SNE	Skip on not equal

These mnemonics follow the conventions for the FORTRAN logical operators (.LT., .LE., etc.) making them easy for the FORTRAN programmer to remember.

SECTION 5

REGISTER SHIFT AND DOUBLE-LENGTH

5-1 SINGLE REGISTER SHIFT OPERATION FORMAT.

In some applications it is useful to be able to slide bits around within the A-or E-registers, or even back and forth between registers. This capability is provided by a set of 19-shift instructions (listed in Table 5-1) which cause the entire contents of a designated register to be shifted some specified number of bits to the left or to the right as desired by the programmer. The format for all shift operations is identical:

<op-code> *b* <operand>

where the op code indicates which register is affected, the type of shift (arithmetic, logical, or circular), and in which direction the shift is to occur; and the operand (which we shall call "y" instead of "m" when writing a formal definition of the shift instructions) is the number of bit places through which the register contents are to be shifted.

5-2 CIRCULAR SHIFTS.

The difference between circular shifts and the other varieties is in what becomes of bits that are pushed outside the register boundaries during the shifting process. In circular (*rotational* or *end-around*) shifts, each bit pushed out of the register is sent to the other end of the register to fill the bit position just vacated. Since a circular shift of 15 bits to the right gives the same result as a circular shift of 1 bit to the left, instructions both left and right circular shifts are unnecessary. 980 AL uses the right shift option and has one instruction for each of the registers (except for the PC and the status register):

Table 5-1. Register Shift and Double-Length Instructions

<u>Instructions:</u>	
CRA	Circular Right Shifts of A-, E-, X-, M-, S-, L-, and B- Registers
CRE	
CRX	
CRM	
CRS	
CRL	
CRB	
ARA	Arithmetic Shifts, Right and Left, Accumulator and Double Length
ALA	
ARD	
ALD	
LRA	Logical Shifts, Right and Left, Accumulator and Double Length
LLA	
LRD	
LLD	
MPY	Multiply
DIV	
DLD	Double Length Load, Store. Add, Subtract
DST	
DAD	
DSB	
NRM	Normalize

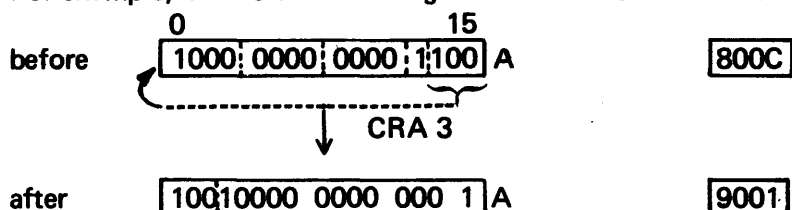
Table 5-1. Register Shift and Double-Length Instructions (continued)

Instructions:

CRA	y	A-register shift
CRE	y	E-register shift
CRX	y	X-register shift
CRM	y	M-register shift
CRS	y	S-register shift
CRL	y	L-register shift
CRB	y	B-register shift

where y is a number or expression giving the shift count.

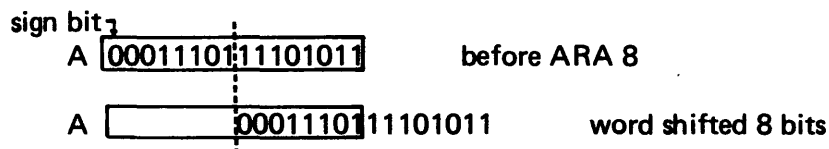
For example, CRA 3 shifts the register contents 3 bits to the right, 1 bit at a time:



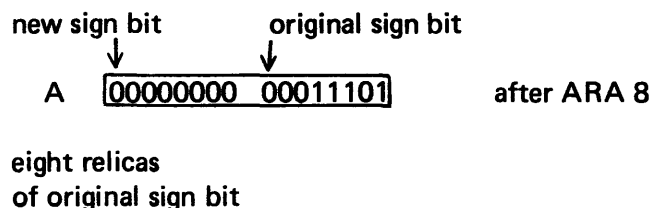
5-3 ARITHMETIC SHIFTS.

The arithmetic shift instructions are generally used in shifting signed numbers within the register. Thus, the sign bit is of particular interest in the way these instructions work. The arithmetic shift instructions apply only to the A-register, since there are no analogues instructions for the other registers.

5-3.1 ARITHMETIC RIGHT SHIFT. The ARA y (Arithmetic Right shift, A register) instruction produces a rightward displacement of y bit positions. For example, ARA 8 produces the following result:

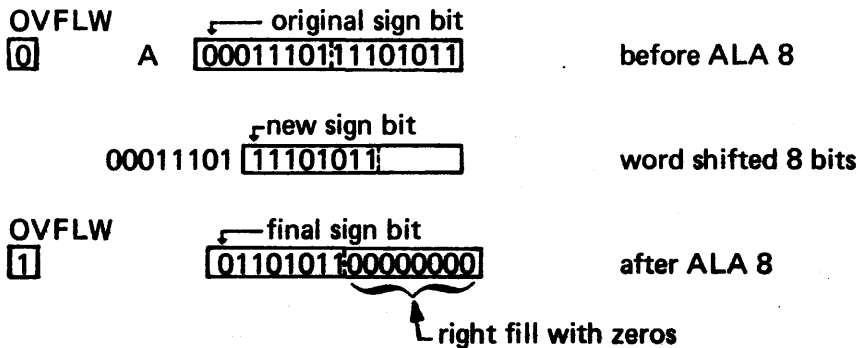


Now the bits that have been shifted off the right end are discarded (the right shift thus is spoken of as an *end-off* shift), and the leftmost 8-bit positions in the A-register are filled with replicas of whatever sign bit appeared in the leftmost position of the original contents of the accumulator – in this example: zero. Upon completion of the ARA 8 operation, the accumulator then contains



This replication of the sign bit through the vacated bit positions is called *sign extension*, and occurs *automatically* during the arithmetic right-shift process.

5-3.2 ARITHMETIC LEFT-SHIFT. In left-shift operations, the bits squeezed out at the left-hand boundary, are discarded, and the register is right-filled with zeros:



Since this is an arithmetic shifts (meaning that the sign bit is of interest), attempting to change the sign bit from 0 to 1 activates the overflow indicator in the status word. The bit itself is not allowed to change and so is forced to agree with the original sign bit.

5-4 LOGICAL SHIFTS.

The A-register (and no other) permits logical shifts to the left or right as well as the arithmetic shifts just introduced. The logical shifts are end-off with the vacated bit positions all being filled with zeros. Since the leftmost bit is not treated as a sign in logical operations, there is no *sign extension* or setting of the overflow indicator as there is in the case of arithmetic shifts. These instructions are of the form:

LLA y logical left shift A
LRA y logical right shift A

5-5 DOUBLE LENGTH REGISTER INSTRUCTIONS.

Some operations affect the double length *super-register* which is formed by *concatenating* (linking together) the A- and E-registers. These instructions include

- MultiPLY and DIVide operations
- Double register shifts (sometimes called *long shifts*)
- Double precision manipulations - load, store, add, and subtract.

Discussion of the double precision operations is deferred until Section 5-6; the others are discussed below.

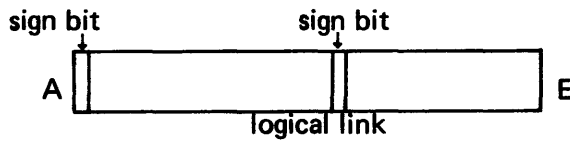
5-5.1 MULTIPLY AND DIVIDE INSTRUCTIONS. Both these instructions involve the accumulator but require the assistance of the E-register (*extension register*).

The MPY instruction is written in the form:

MPY m,<mod>

and has the following effect. The contents fo the (16-bit) accumulator are multiplied by the contents of the (16-bit) word m and the product so formed (regardless of its actual magnitude) is expressed as a 32-bit string.

The rightmost 15 significant bits appear in the E-register and the leftmost reside in the A-register, so it appears that the right end of the A register has been linked to the left end of the E register to make a *super-register* having 32 bits (including 2 *sign bits*):



This linking process is known as *concatenation*, and the double-length register so formed is abbreviated AE since the juxtaposition of those letters reflects the relative positions* the two registers had with respect to each other just as the logical link was forged. When multiplication is complete, the link dissolves. The sign bit of the E register is forced to agree with that of the product. We can write a formal definition of the MPY instruction, then, as follows:

MPY m,<mod> means (A) * (e.a.) → AE

If m is immediate, the signed value of the operand field is used as the multiplier. If both operands are equal to the smallest number that can be expressed by 16 bits (i.e., -2^{-15}), the result is indeterminate, and the overflow indicator is activated. An example of the use of the MPY instruction is given below.

A Random Number Generator

The best method of generating pseudorandom numbers is the *mixed* (additive and multiplicative) *congruence method* in which a "seed" number is multiplied by an appropriately chosen factor, and a constant is added to the product. The pseudorandom number thus produced serves as the seed for producing the next number in the sequence. Although the numbers stream will ultimately repeat (*cycle*), the factors are chosen such that the cycle time is large.

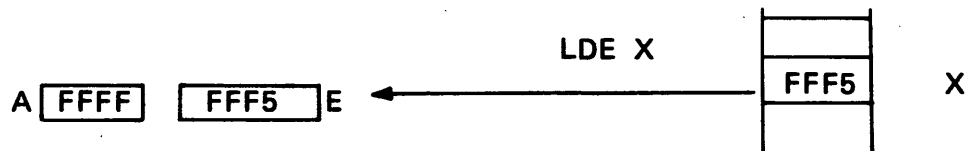
The following routine generates two streams of random numbers (between -32768 and $+32767$) and stores them in arrays RAN1 and RAN2, with the seeds for each stream initialized in the first array position

	IDT	RANDOM	
A	EQU	0	
E	EQU	1	
X	EQU	2	
B	EQU	6	
MULT	DATA	16385	
N	DATA	100	
RANDOM	EQU	\$	
	LDE	N	SET INDEX LIMIT
	RMO	E,B	IN B-REGISTER
	LDX	=0	INITIALIZE INDEX
	LDA	MULT	GET MULTIPLIER
	@MPY	RAN1,X	MULTIPLY BY SEED
	REX	A,E	LOW ORDER BITS TO A
	@ADD	=107	ADDITIVE FACTOR
	STA	RAN1+1,X	SAVE NUMBER IN STREAM 1.

*Beware of taking the word *position* too literally. The builders of the 980 know about the precise physical location of these registers within the main frame, but we probably don't; nor do we care. The two registers could be located at opposite corners of the machine but still appear as neighbors in the electrical circuitry.

	LDA	MULT	
	@MPY	RAN2,X	RETYPE FOR STREAM 2.
	REX	A,E	
	@ADD	=451	
	STA	RAN2+1,X	
	RIN	X,X	INCREMENT INDEX
	RCA	X,B	TEST INDEX
	SEQ		REACHED LIMIT?
	BRU	LOOP	NO, GO BACK
	IDL	0	YES, DONE.
RAN1	DATA	327	SEED 1
	BSS	511	
RAN2	DATA	873	SEED 2
	BSS	511	
	END	RANDOM	

The DIV instruction assumes that the dividend is resident in the 32-bit AE super-register. If the number to be used as dividend was obtained from a previous multiply instruction, it is already 32-bits long and we're ready to divide. However, if the dividend resides in a 16-bit memory location, it is necessary to load it into the E-register and convert it into a 32-bit quantity by a process known as *sign extension*, which is just the replication of the (zero or one) sign bit in the memory word all the way out to the left end sign bit of AE:



Into A must somehow be placed the number FFFF (all binary ones) so the two registers will be ready for the concatenation that will automatically occur when the DIVide circuitry is activated.

The m operand in the

DIV m

instruction contains the divisor and , upon completion of the division, the AE splits apart into its A and E components, where A now contains the quotient having whatever sign is mathematically correct and E contains the remainder which will have the same sign as the original dividend. Formally speaking,

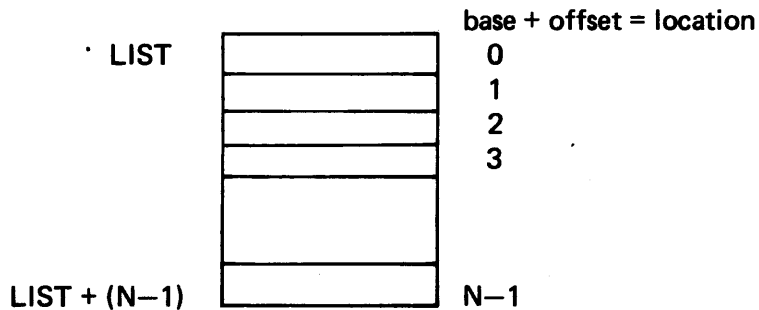
DIV m,<mod> means $(AE)/(m) \Rightarrow$ A (quotient)
E (remainder)

where our formal notation seems to lack any indication concerning the sign of the remainder. In the case of zero remainder, the sign is positive, since negative zero does not exist in two's complement arithmetic. If the magnitude of the dividend is less than that of the divisor (i.e., the result of the operation would be a fraction less than 1) no operation occurs, and the overflow indicator is activated. The sign can be extended in a number of ways

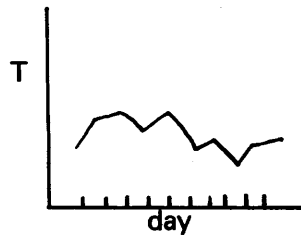
- Multiplication by unity

- Loading the proper bits into the A-register directly
- Through special double-length register shift instructions (Section 5-4.2) which cause replicas of the sign bit to be left in the position of a register vacated as a result of the shift operation.

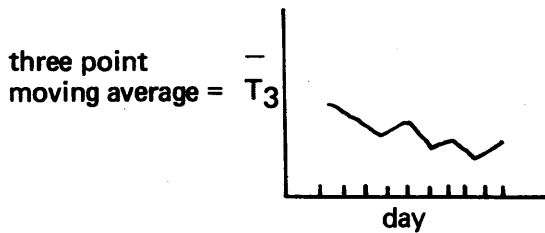
5-5.2 THE MOVING AVERAGE PROBLEM. As an example of a program involving multiplication and division, we can consider the moving average problem in which we have an array of N numbers in the locations from LIST to LIST + (N-1).



Our present problem is to write a 980 AL program to compute a three-point moving average of all values in the array. The moving average technique involves averaging each data value with its neighbors in order to "smooth out" short-term fluctuations. For example, if the data values were daily temperatures recorded at noon through the month of October, a plot of the raw data might appear thus:



whereas the moving average would appear as a smoother curve showing the same trend:

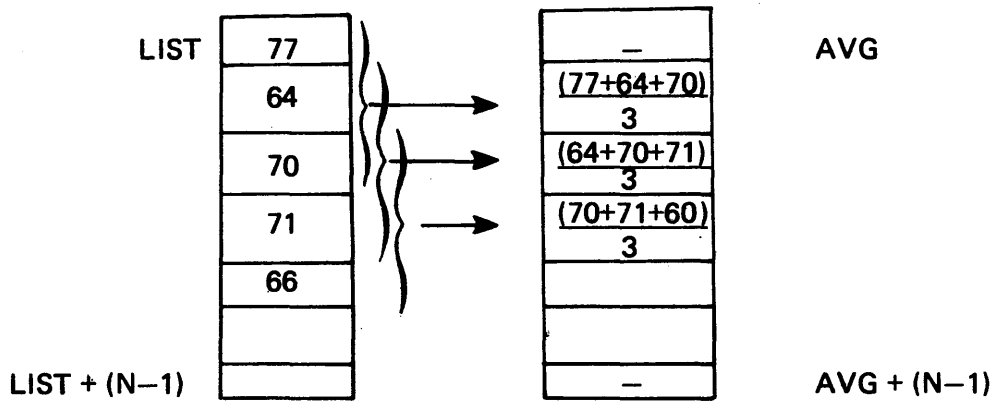


yielding a clearer view of the overall temperature trend during the month, since the short term variations are averaged out.

For the three-point moving average, we can write an algorithm:

$$avg_i = \frac{list_{i-1} + list_i + list_{i+1}}{3}$$

where each of the averages is stored in an element of another array named AVG (note that the endpoints have no corresponding average):

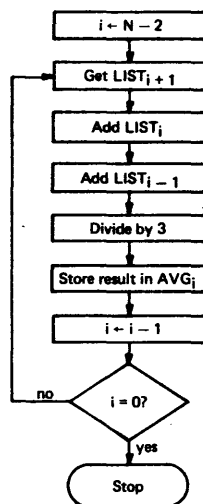


We'll further assume that the number of elements of N is stored in a cell named N

N value of N

We'll design the program so that the index always points to the middle cell of a triad, and the neighboring cells are accessed by address arithmetic:

Flowchart:

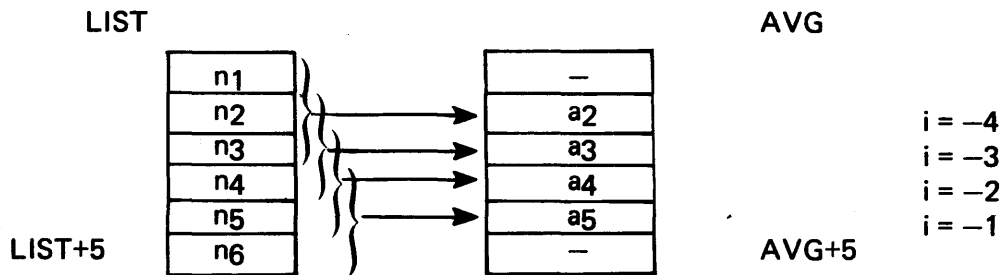


Program:

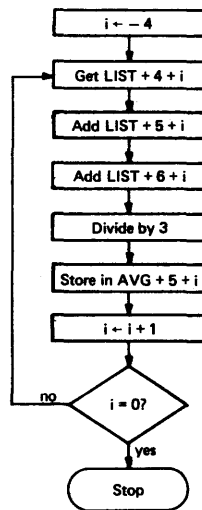
	LDA	N	GET ITEM COUNT.
	SUB	2	COMPUTE MAX INDEX (N-2)
	RMO	A,X	PLACE IN X.
LOOP	LDA	LIST+1,X	ADD ELEMENTS
	ADD	LIST,X	
	ADD	LIST-1,X	
	MPY	=1	SIGN EXTEND.
	DIV	=3	COMPUTE AVERAGE
	STA	AVG, X	AND SAVE
	RDE	X,X	DECREMENT INDEX
	SZE	X	ZERO?
	BRU	LOOP	NO, GO BACK.
	IDL	Ø	YES, QUIT.

We could rewrite the program using a negative index which increments to zero (and, hence, the BIX instruction for the test), but this is a bit awkward unless we know at assembly time the value of N and can work from the top of the array using address arithmetic. A procedure for using the variable N is discussed in Section 6-3.

Consider the example where $N = 6$.



Flowchart:

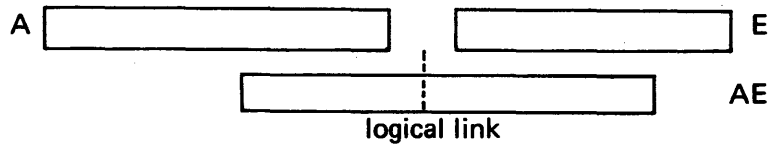


Program:

```

    LOOP      LDX      =-4
             LDA      LIST+4,X
             ADD      LIST+5,X
             ADD      LIST+6,X
             MPY      =1
             DIV      =3
             STA      AVG+5,X
             BIX      LOOP
             IDL      Ø
  
```

5-5.3 SHIFT OPERATIONS IN A 32-BIT REGISTER. The seven double-register shift instructions are often called *long shift* operations because they produce shifts in the 32-bit (i.e. double-length AE register which discussed in dealing with multiplication and division operations. The 32-bit concatenation is formed by making a logical link between the A- and E-registers, that this time the A is on the left and E on the right:



Whenever we write a "double shift" instruction, the machine automatically concatenates the A- and E-registers in this configuration before effecting the shift. At the end of the shift operation, the logical link dissolves and the two registers resume their original identity.

As in single-length register shifts, there are three types of operations:

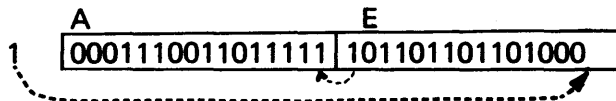
1. Double-length circular shifts. Unlike the single-length instruction group (in which circular shifts are only performed to the right) these are *both* left and right double-length circular shifts; because now that we are dealing with a long bit-string it is faster to use

CLD 1 Circular Left shift (Double) 1 bit

then to use

CRD 31 Circular Right shift (Double) 31 bits

In this case, the bit that drops off the left end of the A-register attaches to the right end of the E-register as each bit moves left one place:

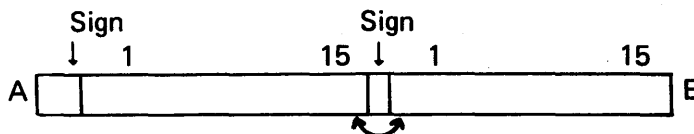


2. Double length logical shifts. The logical shifts behave the same way, except they are *end off* rather than *end around* and the vacated bit positions are filled with zeros, as in single-register logical shifts:

LRD y Logical Right shift Double

LLD y Logical Left shift Double

3. Double length arithmetic shifts. The arithmetic double-length shifts work the same way as their single-length counterparts in that sign extension occurs on a right shift, and zero-fill occurs with a left shift. The principal difference between these and the other long-shift instructions arises from the fact that the two registers maintain the integrity of their sign bits:

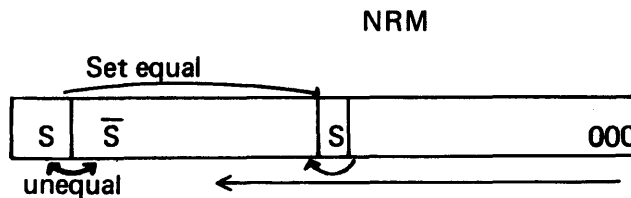


and a shift over the AE boundary will skip the E's sign bit which is forced to agree with A's sign bit. If we

attempt to change A's sign bit on a double-length left shift, the overflow indicator will activate. These instructions are written

ARD y Arithmetic Right shift Double (sign extension).
 ALD y Arithmetic Left shift Double (zero fill).

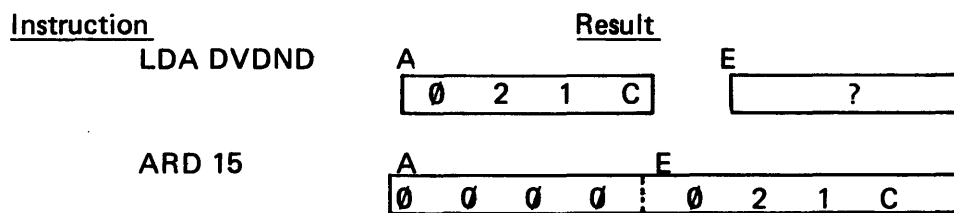
The normalize instruction NRM is also a double-length arithmetic shift. This instruction shifts the AE register to the left until bit 0 of A is unequal to bit 1 of A. The operand field is not used.



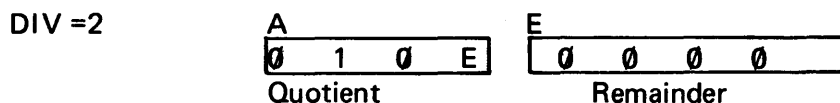
5-5.4 DOUBLE-LENGTH SHIFT IN PREPARATION FOR DIVIDE. If the dividend is a 16-bit number, it must be converted into a 32-bit number in the AE register prior to the issuance of the DIVide instruction; i.e., it must be placed in the E-register with its sign extended through the bits of the A-register. We have already examined one way to do this – namely, multiply the 16-bit dividend in A by unity, which places the 32-bit result in AE. The number is now sign-extended and ready for division. Another way (one could think of several ways) is to load the accumulator with the 16-bit dividend (unless it happens to be there anyway as the result of a previous operation) and do a long arithmetic shift into E, with the result that the garbage in E is replaced by the dividend, and the sign extends through A.

LDA DVDND GET 16 BIT DIVIDEND
 ARD 15 SIGN EXTEND
 DIV DVSR DIVIDE

Pictorially, what happens is this. If (DVDND) = 21C



AE now ready for division



5-5.5 DIVISION BY 2 OR ANY OF ITS POWERS. Since multiplication/division by 10 merely involves moving the decimal point, it follows that the same process can be used to multiply or divide by 2^n where n is any integer power.

If we assume that the right boundary of the accumulator represents the position of the binary point (which it does in integer arithmetic), the arithmetic shift instructions can be used to perform the desired operation:

ALA	n	MULTIPLY BY 2^n
ARA	n	DIVIDE BY 2^n

The result is the shifted number in the accumulator and, in the case of division, the *fractional* portion is expressed in the bits discarded by the end-off shifting process. If we were first to clear the E-register and then perform a long shift by n, the *fraction* would show up in the E register, justified against the left boundary:

LDA	=0	CLEAR E
LDA	DVND	GET DIVIDEND
ARD	<n>	DIVIDE BY 2^n
STA	INTEGR	SAVE INTEGER PORTION
STE	FRAC	SAVE FRACTION

5-6 DOUBLE PRECISION ARITHMETIC.

In Add and Subtract operations we assumed that the operands and the result are all capable fitting into 15 bits of significance plus sign. Some of the world's numbers do not cooperate with this restriction, and we must make provisions to handle them.

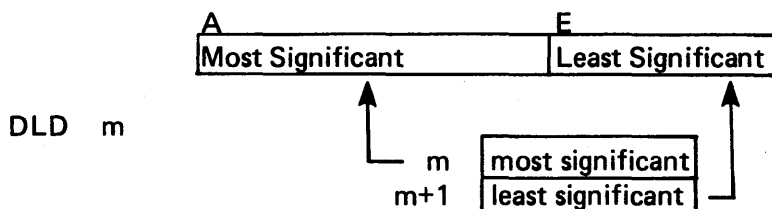
Given numbers larger than 16 bits, we would split them into pieces, each piece capable of fitting into 16 bits and operate on them in a piecewise fashion. Consider, for example, the problem of adding a pair of 24-bit numbers.

First we would have to split the numbers into a low order segment (least significant 15 bits + sign) and a high order segment (the remaining 9 bits with sign extended in another 16-bit word. The low and high order pairs can be added together, respectively, and the low order result tested to see if a carry is generated that must be added to the high order pair.

The 980 has the following four additional double-length instructions which simplify double precision operations:

DLD	m,<mod>	<u>D</u> ouble length <u>l</u> oad
DST	m,<mod>	<u>D</u> ouble length <u>s</u> ore

which loads and stores the A-register from the effective address and the E-register from the effective address + 1. For example:



These instructions handle numbers consistently with the behavior one observes in the operation of the

double-length arithmetic instructions: the MPY and DIV already discussed and two additional ones

DAD	m,<mod>	<u>D</u> ouble length <u>A</u> dd
DSB	m,<mod>	<u>D</u> ouble length <u>S</u> u <u>B</u> tract

5-7 EXECUTION TIMES AND THE SPACE-VERSUS-TIME TRADEOFF.

Having discussed more than half the available 980 instructions, we are now in a position to code just about any problem presented us, except those involving I/O. About half the instructions we have not yet examined are related to I/O, and the other half represent specialized operations which we can program our way around using two (or perhaps more) of the instructions we have learned. The resulting code may be less efficient, but it can be made to work.

Even without knowledge of the additional instructions, we often find several ways to do the same job and should begin to wonder if there is some means of deciding which of several possibilities is best. To address this problem let's examine the two major resources available to us in order to run a program:

- Space within the memory
- Time required for the program to execute.

In most elementary training problems such a sufficient quantity of both resources is available, we seldom will be forced to conserve them. In "real" problems, the situation is often very different. A long program which depends on having large arrays of data stored in memory may require more space in the memory than storage available. In such case anything the programmer can do to reduce the length of the program may mean the difference between being able to solve the problem on the available computer or having to seek out a larger computer. Often the process of pruning the size of the program causes use of instructions requiring a long time to execute so that the price of a shorter program may be a longer running time.

Conversely, if the resource more rapidly depleted is machine time and there is lots of space remaining, it may be possible to rewrite the program in a more verbose fashion so as to gain faster execution. (Of course, if both resources are used up, there is nothing left with which to barter for such an exchange).

If we are given no information concerning the relative speeds with which the operations are carried out (some operations do take longer than others), then, of course, we have no way to judge the speed of the various methods without actually writing them and observing the time used by each. If we were confronted with a machine that demands the same length of execution time for each instruction in the repertoire and do not consider loops, then the program with the largest number of instructions (i.e., taking up the largest amount of space) will also take the longest time to execute — and we lose on both counts.

As a general rule, it is safe to say that — all other things being equal — the shorter program is the better one. But wait! If we carry this principle to an extreme so that the code is written so "tight" as to make debugging and revision a difficult job, we lose again in terms of the programmer time required to produce and maintain the program. How to handle the space/time tradeoff in any given situation is one of the arts of programming.

Execution times for 980 instructions are given in Table 5-2.

TABLE 5-2. INSTRUCTION EXECUTION TIMES IN MICROSECONDS

REGISTER-MEMORY			
Mnemonic	Name	Memory Referencing*	Immediate Addressing
ADD	Add	1.75	0.75
AND	And	1.75	0.75
BIX	Branch on incremented index	1.25	1.25
BRL	Branch and link	1.50	1.50
BRU	Branch unconditional	1.25	1.00
CPA	Compare algebraic	1.75	0.75
CPL	Compare logical	1.75	0.75
DAD	Double add	2.75	1.0
DIV	Divide	2.5 → 7.75	1.50 → 6.75
DLD	Double load	2.75	1.0
DMT	Decrement memory and test	2.75	2.75
DSB	Double subtract	2.75	1.0
DST	Double store	2.75	2.75
IMO	Increment memory by one	2.75	2.75
IOR	Inclusive OR	1.75	0.75
LDA	Load register-A	1.75	0.75
LDE	Load register-E	1.75	0.75
LDM	Load register-M	1.75	0.75
LDX	Load register-X	1.75	0.75
MPY	Multiply	2.25 → 6.25	1.25 → 5.25
STA	Store register-A	2.00	2.0
STE	Store register-E	2.00	2.0
STX	Store register-X	2.00	2.0
SUB	Subtract	1.75	0.75

*Add the following to execution times, when applicable: 0.25 microseconds for indexing, 0.75 microseconds for indirect addressing, and 0.25 microseconds for DAD, DLD, DST, and DSB extended format.

REGISTER SHIFT		
Mnemonic	Name	Time
LTO	Left test for one	$1.0 + \frac{SC}{4}$
LTZ	Left test for zero	
RTO	Right test for one	
RTZ	Right test for zero	
All other instructions**		$0.75 + \frac{SC}{4}$

**Add 0.25 microseconds for ALD and ARD when shift count is zero.

Table 5-2. (concluded)

REGISTER-TO-REGISTER		
Mnemonic	Name	Time
RAD	Register add	1.25
RAN	Register AND	1.25
RCA	Register compare-algebraic	1.25
RCL	Register compare-logical	1.25
RCO	Register complement	1.00
RDE	Register decrement	1.00
REO	Register exclusive OR	1.25
REX	Register exchange	1.50
RIN	Register increment	1.00
RIV	Register invert	1.00
RMO	Register move	1.00
ROR	Register OR	1.25
RSU	Register subtract	1.25

SKIP

All instructions execute in 1.0 microsecond.

MISCELLANEOUS		
Mnemonic	Name	Time
API	Auxiliary processor initiate	AP Controller Dependent
ATI	Automatic transfer instruction	2.5
CLC	Compare logical character string	5.0+2.25/byte
IDL	Idle	1.0
LRF	Load register file	7.0
LSB	Load status block and branch	3.25
LSR	Load status and reset interrupt	3.25
MVC	Move character string	4.75+2.75/byte
NRM	Normalize	1.0 → 8.75
RDS	Read direct single	3.00 → 4.75
SABO	Set register-A bit to one	1.0
SABZ	Set register-A bit to zero	1.0
SMBO	Set memory-bit to one	3.25
SMBZ	Set memory bit to zero	3.25
SRF	Store register file	7.0
SSB	Store status block and branch	3.25
TABO	Test register-A bit for one	1.25
TABZ	Test register-A bit for zero	1.75
TMBO	Test memory bit for one	2.75
TMBZ	Test memory bit for zero	2.75
WDS	Write direct single	3.00 → 5.0

Let's examine three simple code blocks which do the same job and try to evaluate them in terms of space and time consumption. In all cases the job involves preparation of a dividend for division.

Block A)	<u>μsec.</u> 1.75	LDE	DVND	GET DIVIDEND
	2.50	LDA	=0	CLEAR A
		SPL	E	DIVIDEND POSITIVE?
		LDA	=>FFFF	NO, LOAD SIGN EXTENSION
	2.5 → 7.75	DIV	DVSR	YES, DIVIDE
Block B)	1.75	LDA	DVND	GET DIVIDEND
	1.25 → 5.25	MPY	=1	MULTIPLY TO EXTEND SIGN
	2.5 → 7.75	DIV	DVSR	AND DIVIDE
Block C)	1.75	LDA	DVND	GET DIVIDEND
	$0.75 + \frac{15}{4} = 4.50$	ARD	15	SHIFT TO EXTEND SIGN
	2.5 → 7.75	DIV	DVSR	AND DIVIDE

The execution times in microseconds (taken from Table 5-2) are listed above to the left of the code. Ignoring the first and last instructions in each block (they are the same in all cases), we see in blocks A and C a classic example of the space/time tradeoff. The three instructions in block A execute in slightly over half the time of the one instruction in block C.

We find in block B that a multiply (immediate) can consume from 1.25 to 5.25 microseconds. Since we have no way to know how long this multiply will actually take (we'd suspect it to be on the low side because of the simplicity of the *multiply-by-one* operation), we would probably consider it in terms of "best case" and "worst case" situations. If the MPY is done in 1.25 microseconds, block B is obviously the best in terms of the least space and the least time. In the worst case situation (5.25 microseconds) block A is fastest (though longest) and block C is faster than block B (even though the blocks are the same size).

If space is the critical resource, take Block C (or gamble on block B); if time is the critical resource, take block A (or gamble on block B).

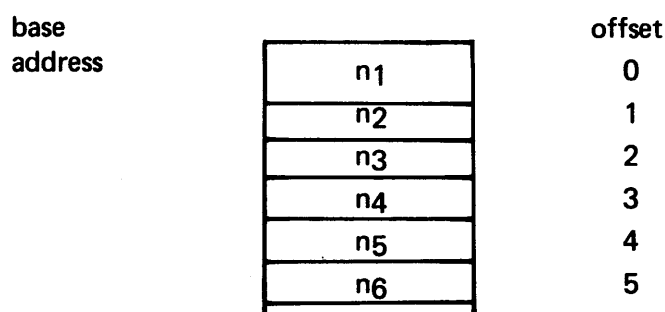
SECTION 6

ARRAY TECHNIQUES: SORTS, SEARCHES, AND STACKS

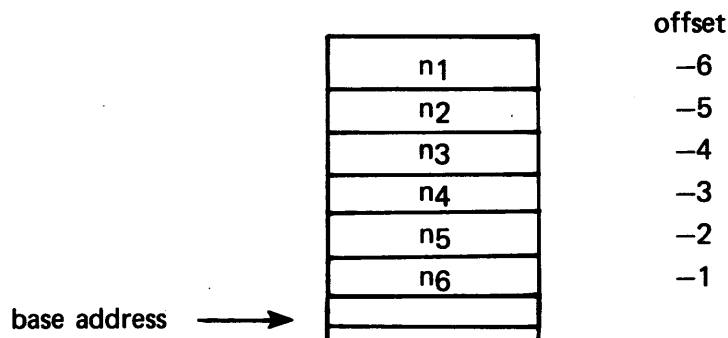
6-1 ARRAY MANIPULATION THROUGH INDEXING.

Use of the X-register to obtain an effective operand address (base + offset) was introduced as the indexed PC-relative mode in Section 3-12.3. A substitute form of indexing using the B-register was explored in Section 4-5.2. In this chapter we discuss a combined addressing mode: indexed B-relative. Then with all these array handling techniques at our disposal, we proceed to some typical array problems encountered in a programming environment.

Arrays can be stored from the base address forward in memory:



or from the base address backward in memory



and indexing may proceed in an incrementing or decrementing fashion, dictated by the position of the base address, the nature of the problem, and the wishes of the programmer. A base address at the top of an array can be converted to an effective base address at the bottom of the array through use of address arithmetic, provided that the precise size of the array is known at assembly time (see Section 3-12.3). Since the precise size of an array is not always known at assembly time (we may not use all the cells for which we have reserved space), it is useful to have a means of coping with such a situation. Indexed B-relative mode (discussed in the next paragraph) is used in the discussion of such variable-length arrays (Section 6-3).

6-2 INDEXED B-RELATIVE MODE.

Standard B-relative addressing, was discussed in Section 4-5, where it was pointed out that one advantage of having a base register is to permit access to memory cells outside the allowable PC-relative range. Normal indexing can occur in a program fragment which is assembled and executed as base register relative (Section 4.5-1):

LDX	=-6	INITIALIZE INDEX
BRS	ARRAY	DECLARE BASE TO ASSEMBLER
@LDA	=ARRAY	LOAD BASE
RMO	A,B	FOR EXECUTION
LDA	=0	INITIALIZE SUM
ADD	ARRAY+6, X	ADD NEXT ARRAY ELEMENT
BIX	\$-1	
BRR		ASSEMBLE PC RELATIVE
STA	SUM	

or using execution time B-relative technique:

LDX	=-6	
@LDA	=ARRAY	
RMO	A,B	
LDA	=0	
ADD	6,XB	SEE FOOTNOTE*
BIX	\$-1	
STA	SUM	

6-3 VARIABLE-LENGTH ARRAYS.

It is not always possible to know at assembly time the number of data values contained in an array during execution time. FORTRAN copes by reserving space (DIMENSION) for some maximum number of elements and then uses as much space as the data requires.

If the data is read in from an external device, the FORTRAN program does either of the following:

1. Requires a count of the actual number, *n*, of data values to be supplied as a *header* value along with the data:

```
    DIMENSION ARRAY (100)
    READ, N
    DO 10 I = 1, N
    READ ARRAY (I)
10  CONTINUE
```

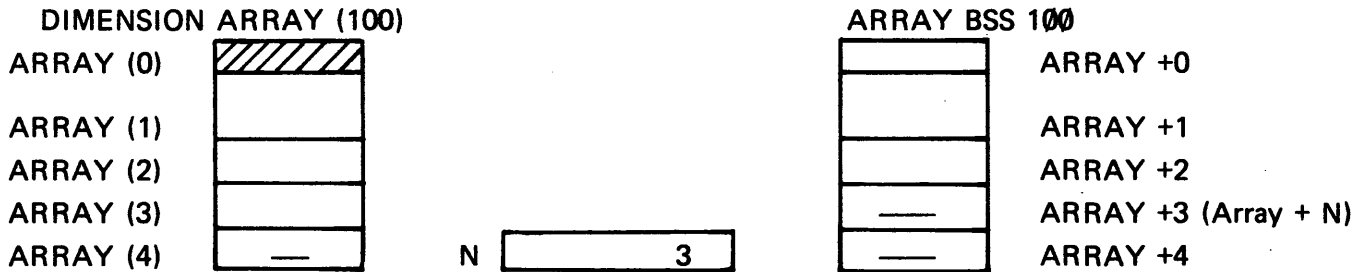
2. Counts the data as it comes in and tests it for a *trailer* value indicating the end of the input sequence. Either a predetermined "ridiculous" value (possibly zero if appropriate to the problem) or an end-of-file can be used:

```
    DIMENSION ARRAY (100)
    DO 10 I = 1, 100
    READ ARRAY (I)
    IF [ARRAY (I).EQ.0] GO TO 20
10  CONTINUE
20  N=I-1
```

*Evaluate by adding the offset (first operand) 6 to the number in B to get the address below the last array element, ARRAY+6. Applying the first index value of -6 returns us to the base address of the array.

In either case — whether N is supplied or computed — its value is available to tell us how many cells in the array contain significant information

The assembly language analog of the DIMENSION statement is the BSS pseudo-op:



Since the ARRAY (0) cell is not available in standard FORTRAN, ARRAY + N represents the first unused cell in the assembly language analogy.

In the case of an N-element array, we would like to be able to write a standard PC-relative indexed loop with a BIX in the spirit of the second example in Section 3-12.3.1, where we would like to make cell ARRAY + N the address to which we apply the negative offset in the index register. We run into trouble because N is a value which must be provided to the assembler, yet it will not be available until execution time. If we write

LDX	N	GET COUNT
RCO	X,X	NEGATE TO FORM INDEX
LDA	=0	INITIALIZE SUM
ADD	ARRAY+N,X	ADD ARRAY ELEMENT ????

observe that the idea has gone sour, as the assembler adds the address of N, not its value, to the base address.

Use of the indexed B-relative mode provides us a clean way out of the dilemma: we can calculate the ARRAY + N address at execution time and place it in the B-register. Indexing then can be done using that address as a base. The following code fragment sums the elements of an array of N values.

Execution Time (μsec)

1.50	@LDA	= ARRAY	GET ADDRESS OF ARRAY
1.75	LDX	N	GET COUNT
1.25	RAD	X,A	COMPUTE ADDRESS OF ARRAY IN
1.00	RMO	A,B	AND PLACE IN B
1.00	RCO	X,X	NEGATE COUNT TO SERVE AS INDEX.
0.75	LDA	=0	INITIALIZE SUM.
2.00	ADD	ARRAY, XB	ADD NEXT ELEMENT
1.25	BIX	\$-1	IF NOT DONE, GO BACK.
<u>2.00</u>	STA	SUM	OTHERWISE, SAVE SUM.
12.50			

The solution to avoid the use of indexed B-relative is to test the value of X after each pass through the loop:

1.75	LDM	N	GET ELEMENT COUNT
0.75	LDX	=0	INITIALIZE INDEX
0.75	LDA	=0	INITIALIZE SUM
2.00	ADD	ARRAY,X	ADD NEXT ARRAY ELEMENT
1.00	RIN	X,X	INCREMENT INDEX
1.25	RCA	X,M	TEST INDEX FOR WRT ELEMENT COUNT
1.00	SGE		INDEX AT LIMIT?
1.00	BRU	\$-4	NO, GO BACK
2.00	STA	SUM	YES, SAVE SUM
<hr/>			
11.50	μsec		

Execution times listed above for each instruction show in this example that the second option is slightly faster.

6-4 THE EXCHANGE SORT (BUBBLE SORT) TECHNIQUE.

Given an array in the computer memory, it is sometimes necessary to rearrange the items into ascending or descending numerical order or, in the case of alphabetic data, into alphabetical order. Alphabetic sorts using the ASCII character set (introduced in Section 7) really prove to be a special case of a standard numeric sorting problem.

Consider an array of $N=5$ numbers which must be sorted into ascending order. The technique illustrated in Figure 6-1 is the exchange sort, in which a sequential pair of numbers is compared and an exchange made if they are not in order. A number of passes through the list may be necessary before complete order is created.

At the end of the pass consisting of $N-1 = 4$ possible exchanges, the largest number has "sunk" to the bottom of the list. then, On the next pass from the top of the list, then, only three comparisons are necessary.

When the last pass single comparison (and exchange, if necessary) is made, we are assured that the numbers are in the proper order.

Pass 1 :	$N-1 = 4$ comparisons:
Pass 2 :	$N-2 = 3$ comparisons
Pass 3 :	$N-3 = 2$ comparisons
Pass 4 :	<u>$N-4 = 1$ comparison</u>

PROCESS COMPLETE

If the original list were in the worst possible order (perfectly *descending*), all $N-1$ passes would have been necessary to order it. If the list were perfectly ordered, it is not affected at all by the process. A partially ordered list may become perfectly ordered at some point before all $N-1$ passes are made. We can save some processing time by testing to see if there is some pass which does not affect the list in any way and quitting as soon as such a pass is made, even though we may not have reached the maximum number of $N-1$. The test consists of clearing a flag at the beginning of a pass, setting it if an exchange is made, and testing it upon completion of the pass.*

*See flow chart page 88

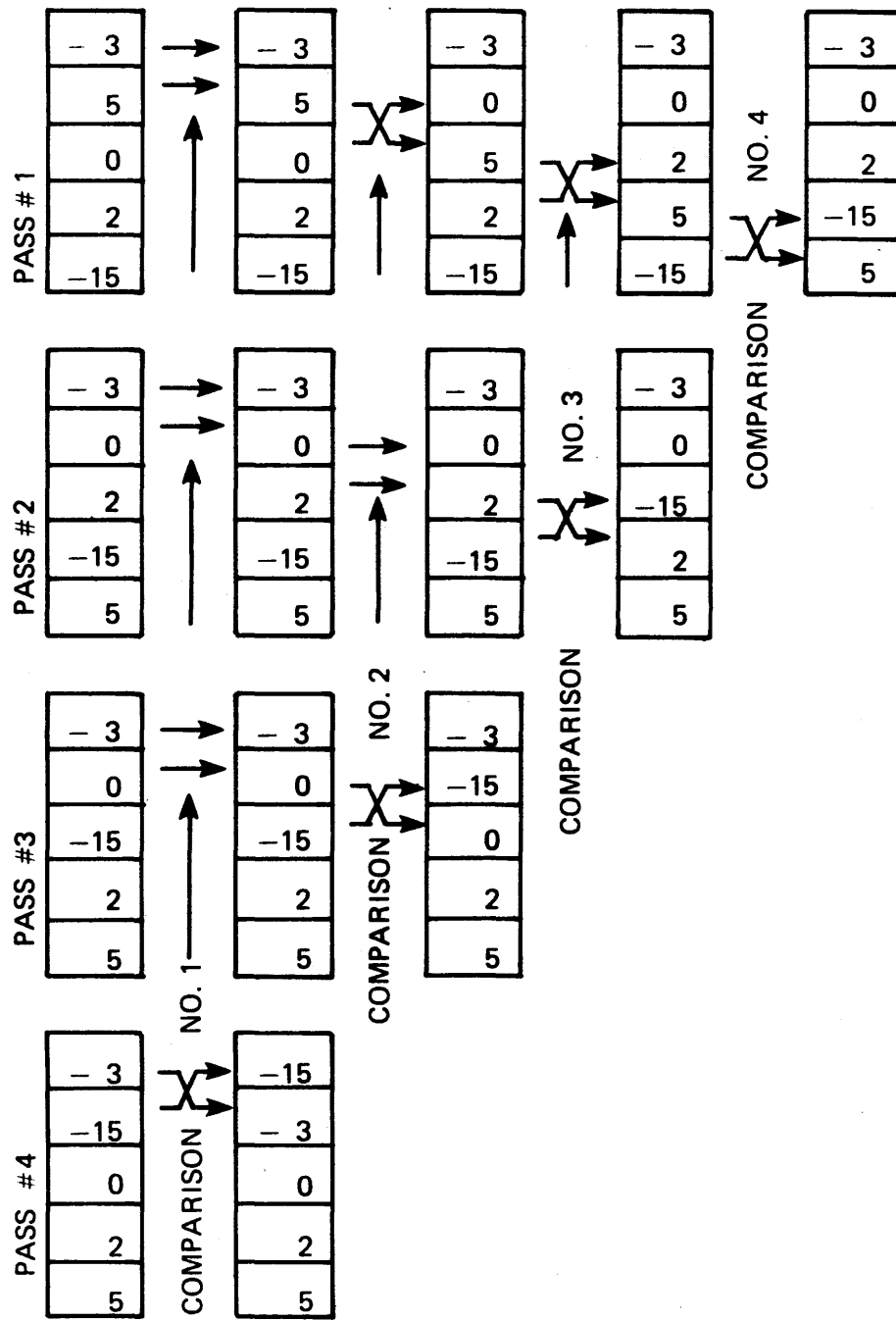
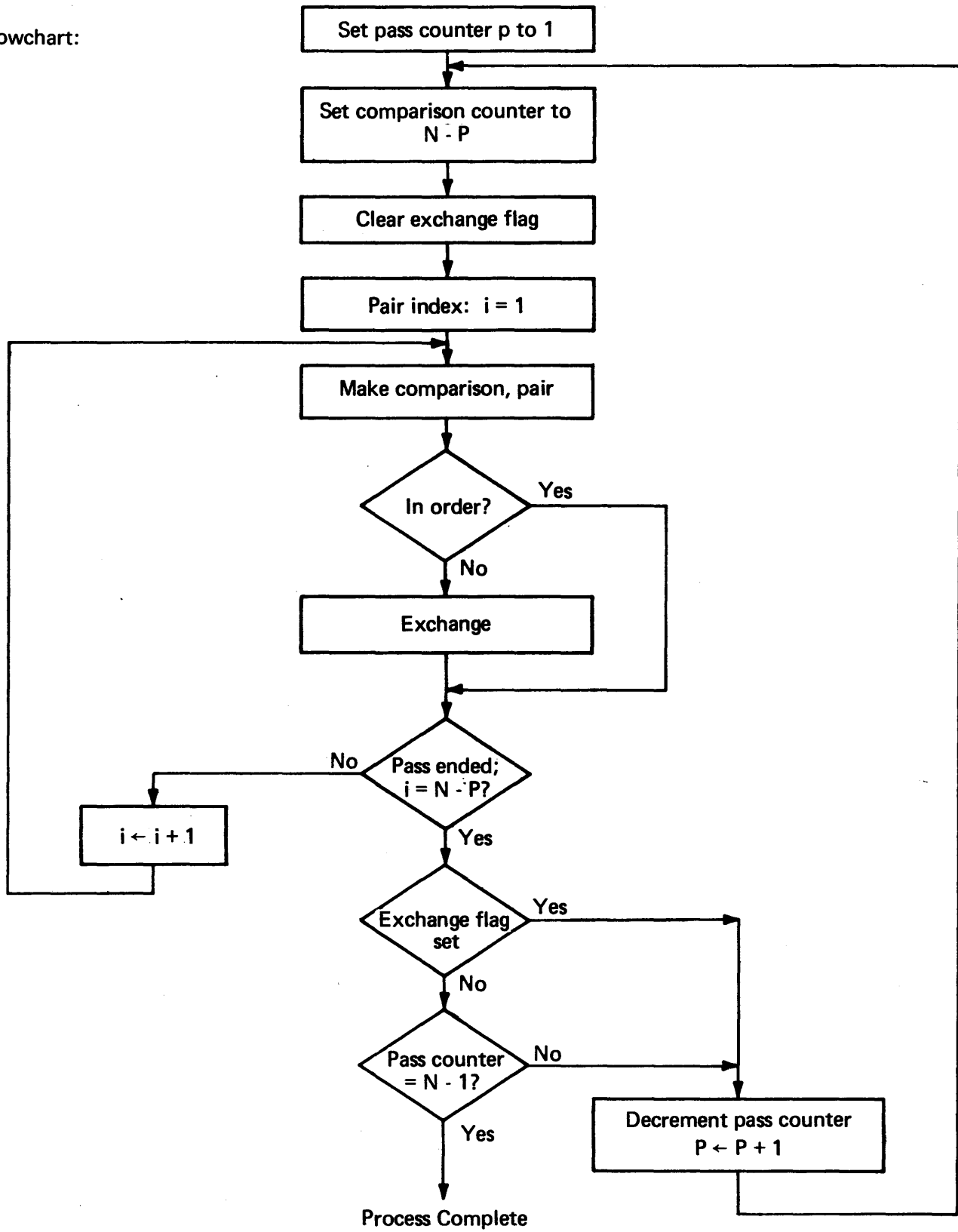


Figure 6-1. Exchange Sort Technique

Flowchart:



Since the number of comparisons is variable, decreasing from pass to pass, we will find useful the indexed B-relative mode (Sections 6-2 and 6-3).

	LDA	=1	INITIALIZE PASS COUNTER
	STA	PASS	AND SAVE.
MORE	LDA	N	GET ELEMENT COUNT.
	SUB	PASS	GET N-PASS = COMPARISON COUNT
	RCO	A,X	AND PLACE ITS NEGATIVE IN X.
	@ADD	=LIST	COMPUTE LIST + (N-PASS)
	RMO	A,B	ADD PLACE IN BASE REGISTER.
	LDA	LIST,XB	MAKE COMPARISON
	CPA	LIST+1,XB	OF PAIR.
	SGT		IN ORDER?
	BRU	NO EXCH	YES, NO EXCHANGE
	LDE	LIST+1,XB	NO, MAKE EXCHANGE
	STE	LIST, SB	
	STA	LIST+1,XB	
	IMO	FLAG	AND SET EXCHANGE FLAG.
NOEXCH	BIX	MORE	IF PASS NOT DONE, GO BACK.
	DMT	FLAG	OTHERWISE, TEST FLAG.
	BRU	QUIT	IF FLAG IS ZERO, PROCESS COMPLETE;
	LDA	N	OR IF PASS = N-1, PROCESS COMPLETE.
	RDE	A,A	
	CPA	PASS	
	SNE		
	BRU	QUIT	PROCESS COMPLETE
	IMO	PASS	IF NOT, INCREMENT PASS COUNTER
	BRU	NXTP	AND GO BACK
QUIT	IDL	Ø	OTHERWISE, IT'S DONE.

6-5 SEARCH TECHNIQUES.

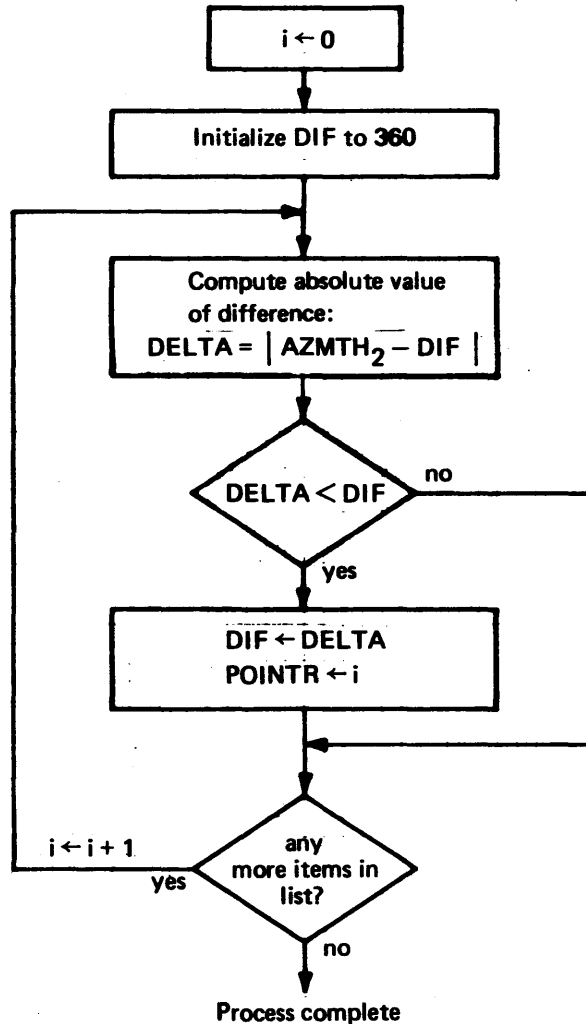
Given a reference item and a list, occasions will arise when we wish to search the list to find a *match* for the reference item. If a match is found, we want to know its position in the list so that we may take some action upon it. If no match is found, some other kind of action is almost always implied. The lists may be ordered or random, depending on the specific requirements of the search technique used.

The meaning of *match* should be elaborated. Sometimes the match must be identical (in which case the process might be termed an *equality search*). That is a special case of the more general problem of finding the list element which is closest to the reference element. For example, if we have a list containing azimuth readings in degrees:

AZMTH	275
	310
	094
	172
	033

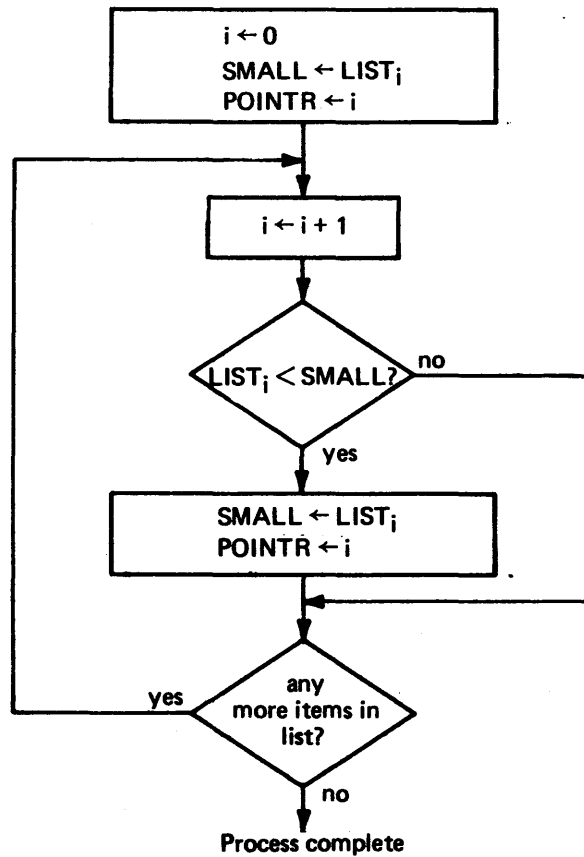
and a reference element REF 090 , visual inspection shows no exact match: the closest match is the 094 entry in AZMTH + 2. We could instruct the computer to find that for us by a search for the smallest difference between REF and the elements of AZMTH. We must take an initial value for the difference (DIF) to be an unrealistically large value (such as 360).

Flowchart:



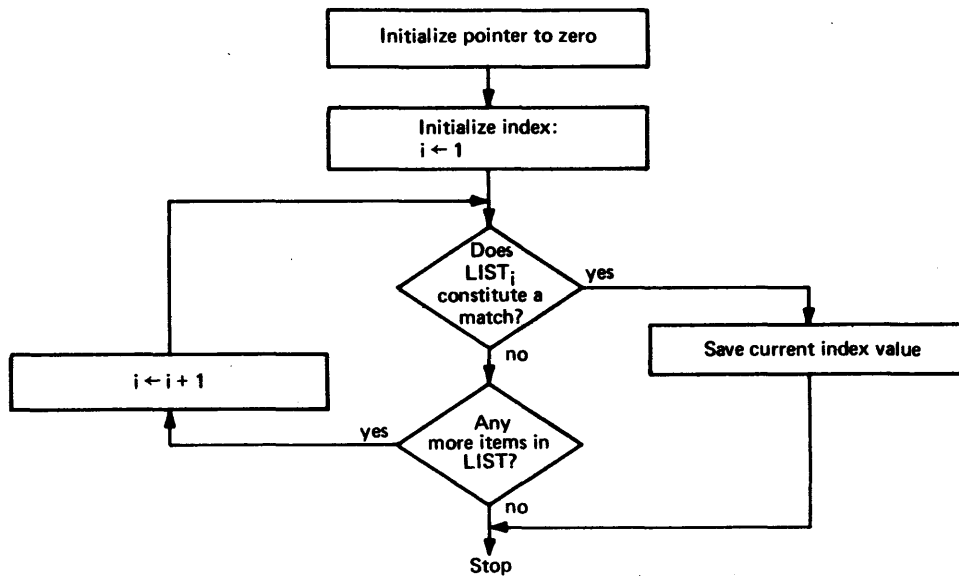
When this process is finished, the pointer will point to the list element closest to the value of REF. In case of duplicate matches, it will point to the earliest one encountered. This same technique is useful for finding the smallest (or largest) element in a list, except we would arbitrarily select the first element as the smallest and then attempt to find one smaller.

Flowchart:



6-6 SEQUENTIAL SEARCH.

The sequential search is a simple technique most appropriately used for short lists. Longer lists are generally better handled by more sophisticated methods. The sequential technique involves examining each list element in turn and comparing it with the item sought. If the list item constitutes a match, the position of the item in the list is saved, and the search may be continued if the list may contain multiple occurrences of the item and we are interested in such occurrences. Here is a flowchart for finding a unique list item.



The index *i* also serves as a count of the number of matches found in the list. We could consider a list having *N* elements. (*N* is an execution time variable (see Section 6-3) or a fixed value known at assembly time.) To simplify the example, we'll use the latter option.

The code for N=10 elements may be written:

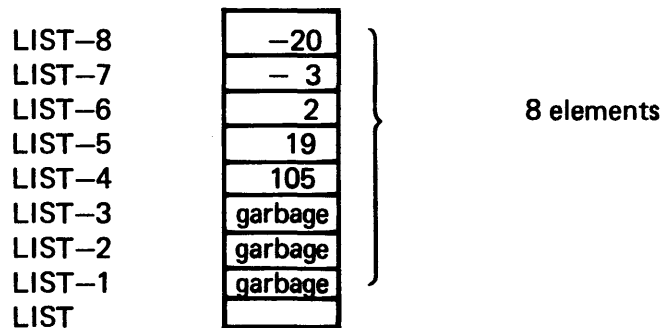
Program:

	LDA	=0	INITIALIZE POINTER INDEX
	STA	POINTR	
SRCH	LDX	=-10	INITIALIZE INDEX
	LDA	LIST+10,X	GET NEXT LIST ELEMENT
	CPA	MODEL	TEST FOR "MATCH"
	SNE		FIND ONE?
	BRU	FOUND	YES, SAVE POINTER
	BIX	SRCH	NO. END OF LIST? NO, GO BACK.
NOFIND	IDL		QUIT
FOUND	STX	POINTR	YES.
	IDL		

6-7 BINARY SEARCH.

A list which is searched sequentially for a match to a reference element need not be in order. However, an ordered list is imperative using the binary search technique. This technique involves examining the dividing line between the upper half and the lower half of the list and deciding in which half the element of interest would have to lie if it were present. This half is further subdivided into half and the process repeated until isolation to a single element is obtained. Either the element constitutes a match or does not constitute a match, and the process ends.

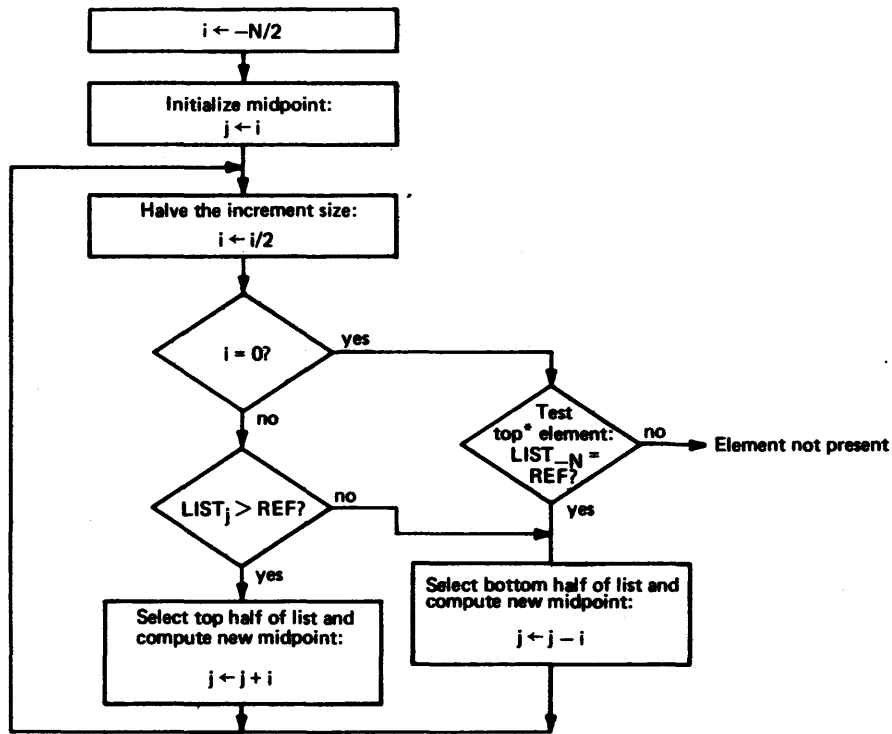
The binary search is most easily performed on a list of N elements where $N = 2^n$; i.e., some integral power of two. It is useful to "pad" the list at the end until this condition is met. Take a list of 5 elements, for example, the next higher power of 2 is 2^3 , so we have an eight-element ordered list. Since one way of dividing the list into half each time involves dividing the index by 2, let's set up the list with a BES symbol and use negative offsets:



If we are looking for the number:

REF	5
-----	---

the process works something like the following, where i is the increment in list size and j is the midpoint of the current list segment.

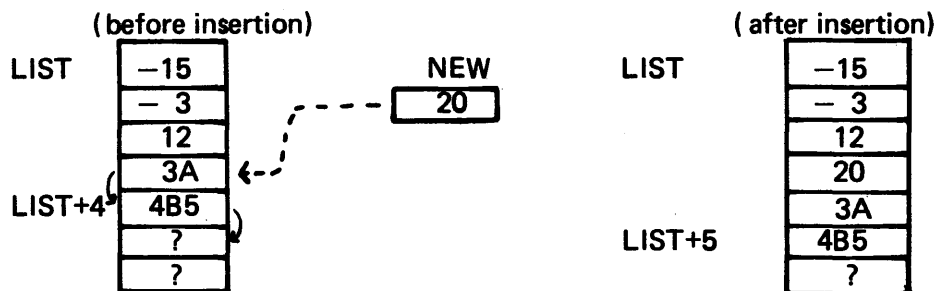


6-8 SEARCHES USING HASH TECHNIQUE.

To use the hash technique list to be searched must be built in a special way, and the method used in building the list is to find an item in it. The *hash total* of an item is the arithmetic sum of all characters constituting the item and is most often used in building lists of alphanumeric characters (such as symbol tables). Further discussion of the hash technique is deferred until character data is discussed in Section 7-7.

6-9 INSERTION AND DELETION IN A LIST.

Let us assume we have a program which is maintaining an ordered list, and we wish to write a code block to insert a new value into the appropriate position in the list:



We could do this by making a new copy of LIST, element by element, inserting NEW at the appropriate time, and continuing to the end of LIST. It is more efficient (usually) to work entirely within the original list, starting with the bottom value and moving it into the empty slot below, moving up to the next value and copying it one slot below, and so forth until the location is freed into which NEW should be copied. We need to maintain a COUNT of the number of items currently in the list:

*The topmost element (LIST_N) is not accessible through values of j; if the increment size cannot be further divided, this is the only cell which has not been tested. We must either perform the test on the cell or make a test [i.e., is j = N - (n - i)] to determine if we should perform the test on the cell. It is more efficient just to make the test in all cases whether appropriate to the data or not.

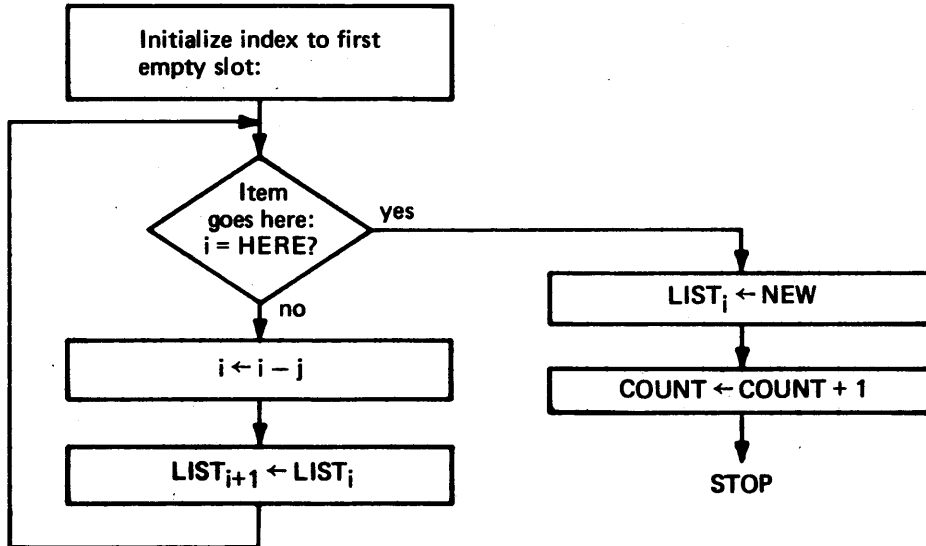
COUNT
5
 (before insertion)

COUNT
6
 (after insertion)

A number of search techniques are available, varying from the simple sequential search to ever increasing degrees of sophistication. Whichever technique we elect to use, however, we will expect from it a pointer named HERE which gives us the value of the offset at which insertion is to take place. In our example, HERE takes on the value of +3.

We must first move every item in the lower part of the list (including the original entry in LIST +3) down one cell.

Flowchart:

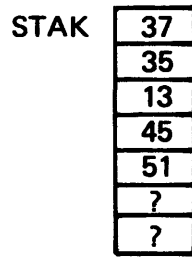


Program:

LOOP	LDX COUNT RMO X,A CPA HERE SNE BRU INSERT RDE X,X LDA LIST,X STA LIST+1,X BRU LOOP	GET INDEX OF FIRST EMPTY CELL. MOVE TO A FOR TEST INSERT POINT? YES, GO TO INSERTION NO, DECREMENT INDEX. MOVE ITEM TO CELL BELOW AND GO BACK
INSERT	LDA NEW STA LIST,X IMO COUNT IDL Ø	INSERT ITEM INCREASE COUNT

6-10 THE PUSHDOWN STACK.

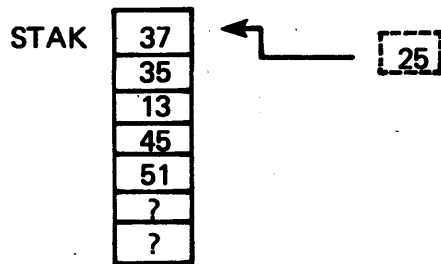
In some computer problems it is convenient to store data values in the form of a *pushdown stack*. For example, given an array (STAK) of five numbers:



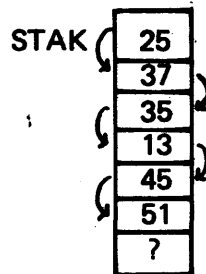
and NEW a new number to be added to the top of the list



push down each old number, starting with the bottom one, into the slot below and insert NEW number



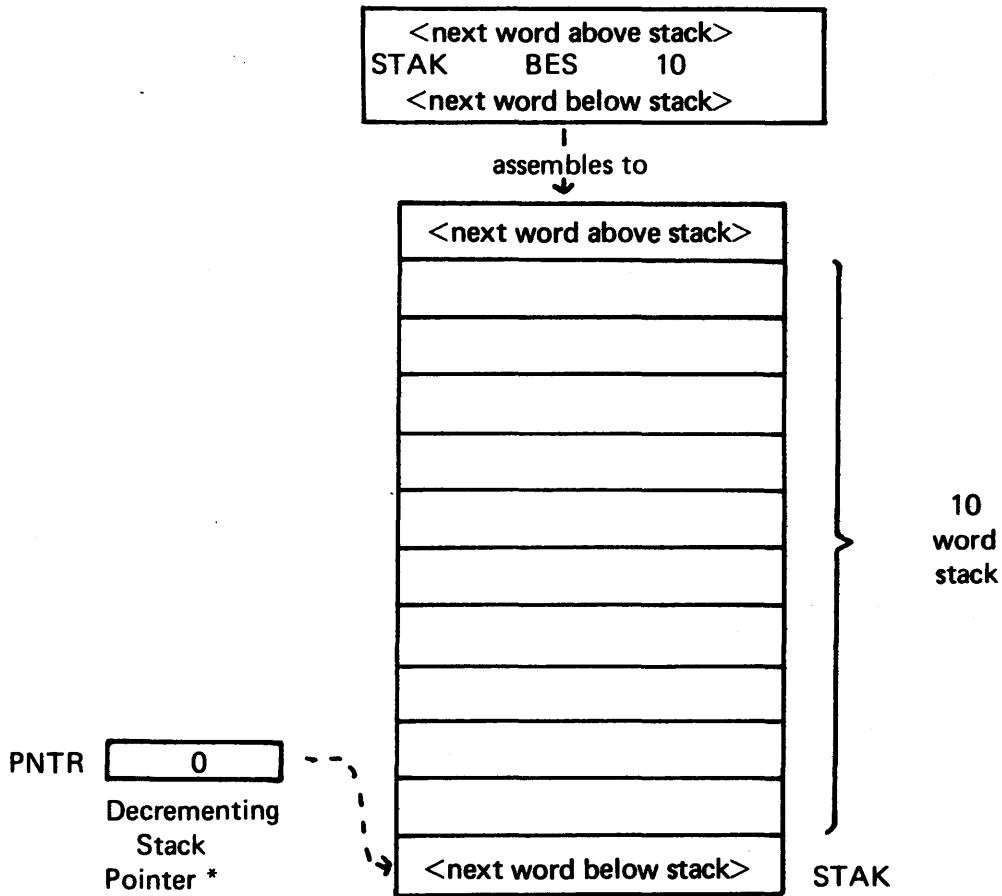
resulting in



Removal of items from the list consists of taking the item from the top of the stack and "popping up" the items below to fill in the space thus vacated. Thus, the pushdown stack is a last in, first out (LIFO) structure.

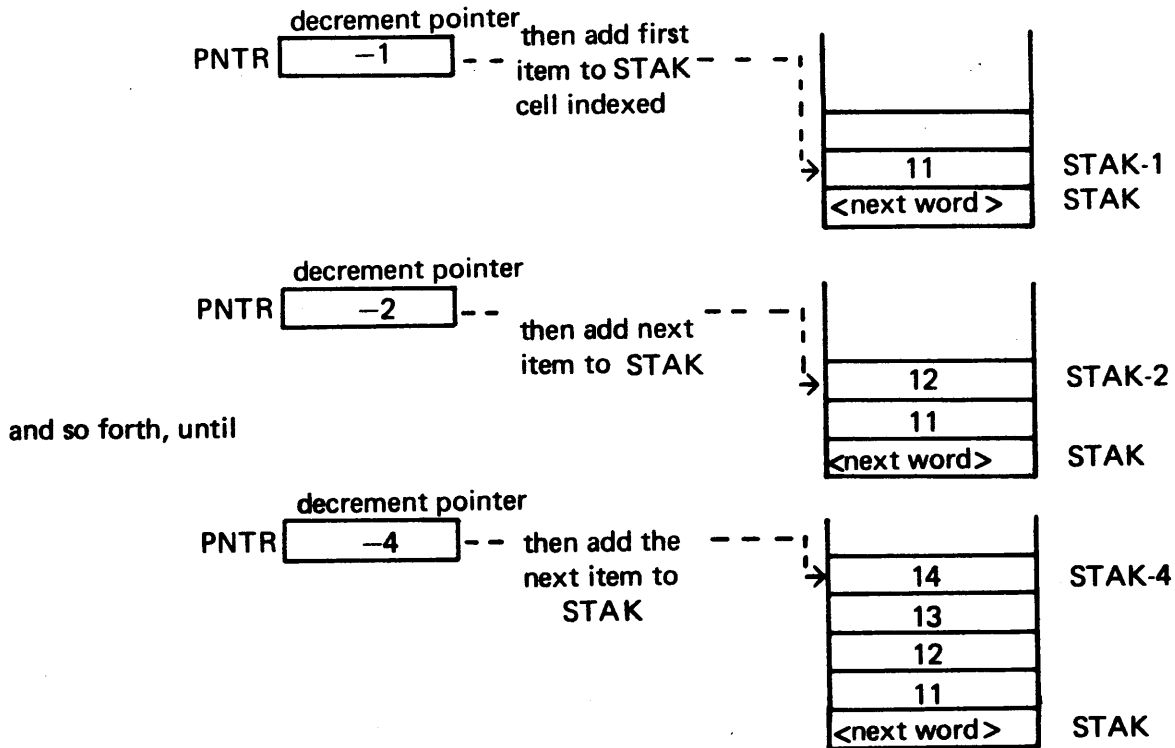
We could interpret this problem as a list insertion (Section 6-9), in which the insertion always takes place at the top of the list. However, moving array elements costs more in overhead than really necessary for most problems requiring use of a stack. As an alternative to moving the items around, let us investigate the technique of leaving the list fixed and moving only a pointer to the list. Given our BIX instruction, one convenient way to building the stack is to define a base address which one cell below the maximum stack address and move a pointer PNTR backwards when adding something to the stack and forwards when popping an item off the stack.

If we define a stack with a *block ending symbol* (BES) to contain 10 words, the BES identifier is attached to the location following the tenth word:

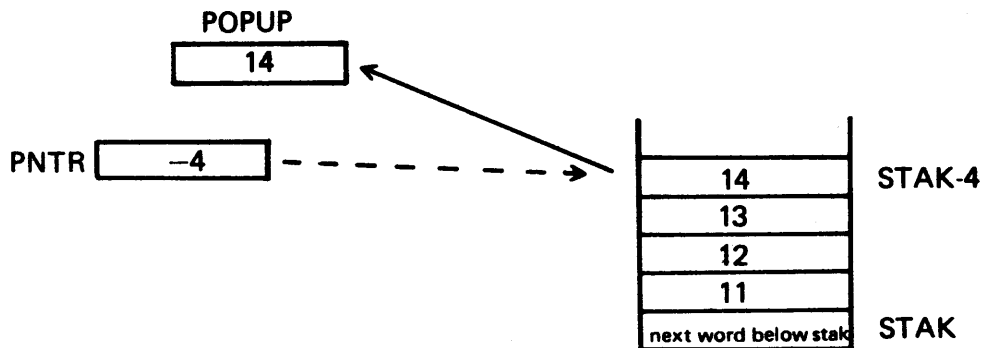


When the stack is empty, the pointer index is = zero. Before adding a number to the stack, we decrement the pointer and then store the stack entry in the cell indexed by adding the value of pointer (negative) to the base address (STAK). Let us add to the stack the sequence PUSH = 11,12,13,14 and see how the stack and the stack pointer work; "14" will be the last in and thus, the first out:

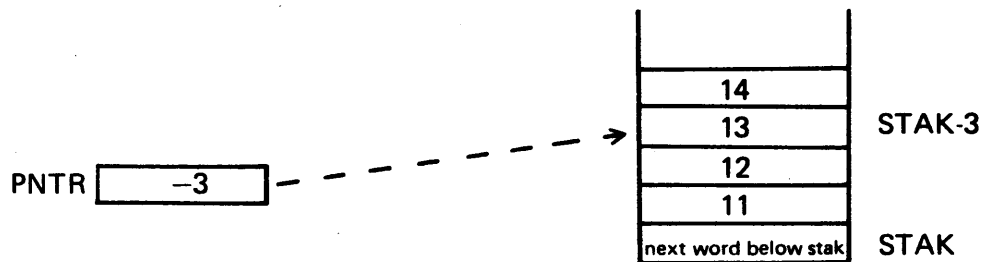
*For neatness' sake, we could use location STAK itself as the cell we call PNTR.



We know that in popping up the stack, the items must come off in reverse order; i.e., 14,13,12,11. Item 14 is the item pointed to already; copy it out into POPUP and increment the stack pointer:



Then increment the stack pointer:



Notice that 14 is still left as garbage in the STAK: it is the stack pointer that moves to define the next item to be removed.

The coding used to control such a stack is somewhat dependent on how the stack items are to be used. If we wish the emptying of a stack to trigger some kind of wrapup routine (named EMPTY), we code the stack control this way:

Pushdown Process:

```

...
LDX      PNTR      GET CURRENT STACK POINTER*
RDE      X,X       DECREMENT TO NEXT STACK CELL
LDA      PUSH      GET NEW ITEM
STA      STAK,X    STORE IN STACK
STX      PNTR      SAVE STACK POINTER
...

```

Popup Process:

```

...
LDX      PNTR      GET STACK POINTER*
LDA      STK,X     GET ITEM TO BE REMOVED
STA      POPUP     SAVE ITEM
BIX      CONTIN    JUST REMOVED LAST ITEM?
...
CONTIN   STX      PNTR      YES, ENTER EMPTY ROUTINE
                                NO, SAVE POINTER

```

EMPTY <process last POPUP value> NOTE THAT PNTR STILL CONTAINS THE VALUE - 1

Note the dangers involved:

1. Stack overflow will wipe out the code resident in locations STAK-11 and earlier; this situation can be guarded against by enabling the memory protect (MP/PIF) feature discussed in Section 11-6.
2. Popping up an already empty stack will lead to the repeated use of STAK-1. If the first instruction in EMPTY is STX, PNTR, popping an empty stack leads to the use of the next word and those cells following as if they were stack items. The counter will take on increasing positive values +1, +2, +3, . . . and will never read zero. This situation can be guarded against by writing a slightly longer POPUP routine:

"Better" Popup Process

```

LDX      PNTR      GET STACK POINTER
SNZ      X         TEST FOR EMPTY
BRU      BARE     NOTHING THERE, GO TO BARE
LDA      STAK,X   OTHERWISE, CONTINUE
STA      POPUP     SAVE ITEM
RIN      X,X      INCREMENT STACK POINTER
STX      PNTR     AND SAVE
...

```

*These instructions are necessary only when the X-register is used for other purposes outside these code blocks.

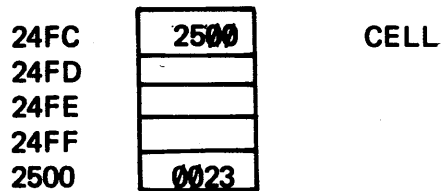
As a general rule, it is better to program the extra instructions needed to avoid flirtation with dangerous situations.

6-11 INDIRECT ADDRESSING.

Indirect addressing involves using contents of the cell specified as the address of (i.e., a pointer to) the cell actually desired. When we write "load A from CELL" we mean "load A with *the contents of CELL*". If we were to write "load A from CELL, indirect", we mean "load A with *the contents of the contents of CELL*". We would write it:

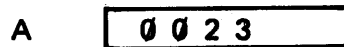
LDA *CELL

Assuming that CELL is the name of location 24FC and contains the value 2500:

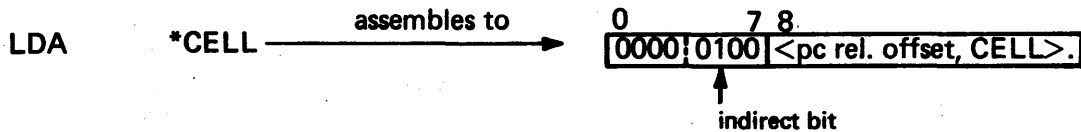


the instruction LDA *CELL would put into the accumulator the contents of the address contained in CELL:

CELL contains 2500, and the contents of location 2500 is 23 which is the number entered into the accumulator:



The LDA *CELL instruction is assembled with bit 5 (i.e., the "I" bit of the IXB modifier triad equal to 1):



This mode of addressing is referred to as *one-level indirect*.

Some machines allow *cascaded* indirect addressing, in which each cell encountered contains the address of the next cell to be referenced until the choice finally stops where the desired cell is found. The 980 has provision only for one level of indirect addressing.

Indirect addressing may seem frightening at first, but it's easy if you keep your wits about you. Here's a "plain-language" example: you wish to talk with me at 10 a.m. tomorrow, and I'd be delighted to talk with you. I'm not sure just where I'm going to be, but I know I won't be at my desk. I'll ask you to come to my desk anyway, so that you can find the note I'll leave there telling you where I am. If you go where the note tells you and find me there, that's one-level indirect addressing. If, instead of finding me, you find a second note that tells you where to look, that's two-levels indirect. In general, when you finally find me (with murder in your eye!) at the end of a trail of N notes (a trail that began at my desk), this analogous to N-level indirect addressing.

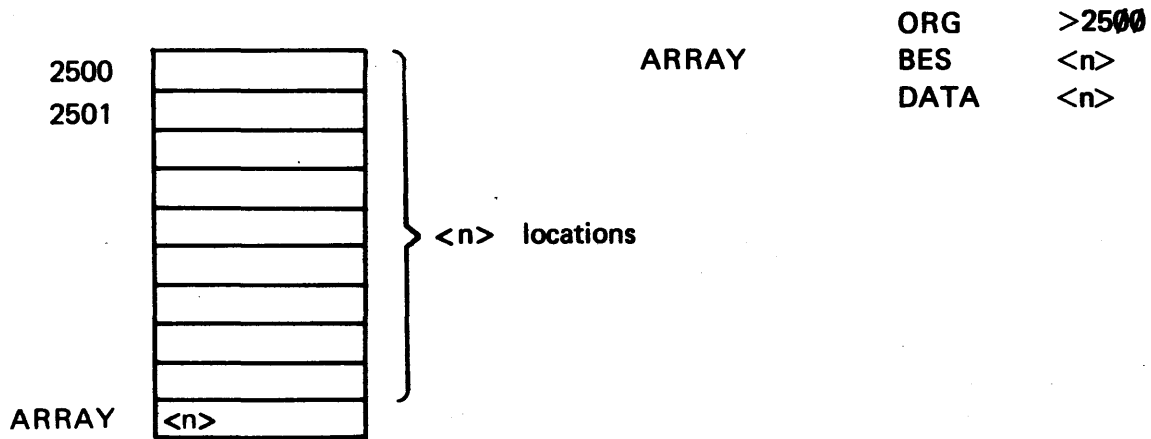
6-12 SUMMARY: THREE METHODS OF ARRAY HANDLING.

Digital computers on today's market range from the primitive to the sophisticated. One basis for rating machines is the addressing modes available that can be used to handle array problems:

	Indexed	Modes Available	
		Indirect	Indexed/Indirect
Sophisticated	x	x	x
Semi-sophisticated	x	x	
Simple		x	
Primitive		impure procedures	

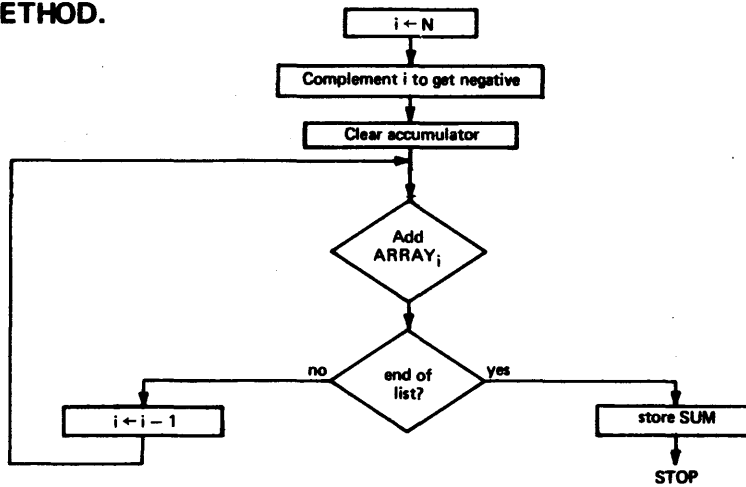
The TI 980 resides in the sophisticated class along with most of the large computers because of its ability to handle both indirect and indexed modes in the same instruction. Many minicomputers fall into the second class, in which both indexed and indirect modes are available (but not simultaneously). Simple machines usually dispense with an index register and rely heavily on indirect addressing. Primitive machines, lacking even the indirect capability, rely on real-time code modification ("impure procedures") to handle arrays. To contrast the three basic methods of array handling (indexed, indirect, and impure procedures) we will consider one problem - the simple case of adding an array of numbers - and program it in the three modes.

In this problem we assume an array of $\langle n \rangle$ numbers and store the value of $\langle n \rangle$ in a location of the same name. The array is labeled ARRAY and out of deference to the operation of BIX, we store it backwards from $ARRAY-1$ to $ARRAY-\langle n \rangle$



6-12.1 INDEXING METHOD.

Flowchart:

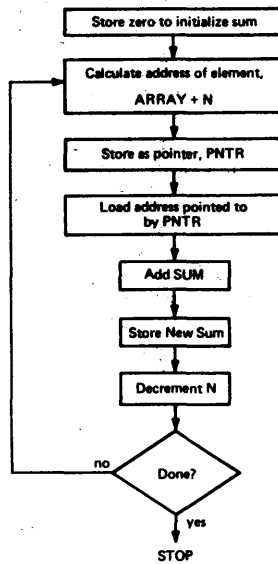


LDX	ARRAY	GET VALUE OF N AS INDEX
RCO	X,X	CHANGE SIGN*
LDA	=0	CLEAR A
ADD	ARRAY,X	ADD ELEMENT
BIX	\$-1	IF MORE, GO BACK
STA	SUM	OTHERWISE, SAVE SUM

A one-level indirect addressing scheme can be used to index through an array as an alternative to address modification; this approach is essentially useful when dealing with a different machine – one which has no index registers available. Let's think about how it might work:

- a. Establish a location named PNTR which will serve as a pointer into ARRAY

Flowchart:



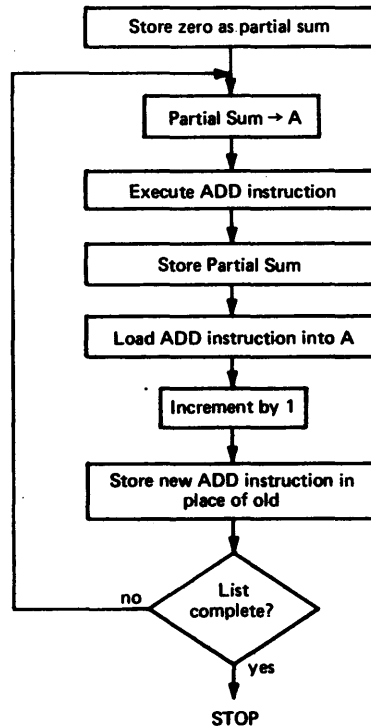
*RCO is a register complement operation:

$$\text{RCO } s,d \quad (s) \times (-1) \rightarrow d$$

	LDA	=0	CLEAR A
	STA	SUM	AND INITIALIZE SUM.
LOOP	LDA	=ARRAY	GET BASE ADDRESS
	SUB	ARRAY	SUBTRACT N TO GET ELEMENT ADDRESS
	STA	PNTR	SAVE AS POINTER.
	LDA	*PNTR	GET ELEMENT POINTED TO
	ADD	SUM	ADD SUM
	STA	SUM	AND SAVE
	DMT	ARRAY	SUBTRACT 1 FROM COUNT, EQUAL 0?
	BRU	LOOP	NO, GO BACK
	...		YES, GO ON

6-12.2 SELF-MODIFYING CODE “(IMPURE PROCEDURE)”. Basically, the computer cannot distinguish between instructions and data (both are hex *numbers*). Any number it is told to execute, the computer will treat as an instruction. Any number it is told to operate on the computer will treat as data. If we tell the computer to operate on an instruction, then, it will do to the instruction exactly what it would do to a data number. Thus, we should be able to load an ADD <operand> instruction into the accumulator and increase it by 1 to get an instruction that means ADD <operand>+1. If we store that ADD instruction and loop back to execute it again, we should once again perform the ADD but this time with the next operand in the array.

Flowchart:



Program:

LDA	=0	CLEAR A
STA	SUM	STORE W SUM
LDA	SUM	GET PARTIAL SUM
ADD	ARRAY-1	ADD IN ELEMENT

STA	SUM	AND SAVE
LDA	\$-2	GET ADD INSTRUCTION
SUB	=1	SUBTRACT 1 FROM ADD INSTRUCTION
STA	\$-4	STORE NEW ADD IN PLACE OF OLD
DMT	ARRAY	DONE?
BRU	LOOP	NO, BRANCH BACK
...		YES, GO ON

This is *not* the nicest way to write a program. Self-modifying code is often hard to follow and hard to debug because we always run the risk of modifying something a way in which we never intended and, therefore, don't expect. The instructions that will be executed are those that actually appear, regardless of whether they were what we intended or not – and it's difficult to discover what a program is doing wrong if you don't know what it is doing at all!

There are some jobs on some machines in which use of self-modifying code is the only reasonable approach, but the rule of thumb is: **AVOID THE USE OF SELF-MODIFYING CODE.**

SECTION 7

ASCII DATA, BUFFERS, AND I/O SERVICE CALLS

7-1 INTRODUCTION TO CHARACTER STRINGS.

The instructions discussed in this section are shown in Table 7-1.

If you already know how alphabet characters are handled by FORTRAN, you can safely skip this section and go on. But if you don't know, the confusing subject of character strings is best introduced in terms of what happens in a higher level language.

In addition to handling numerical data, FORTRAN can be used to manipulate alphabetic (or *character*) data as well – just so long as the characters occur in groups of six* or fewer. We could write in FORTRAN

N = "SALLY"

and if we could examine the contents of the variable N, we could think of it as looking like

N

SALLY	
-------	--

where any of the six character positions we have not used (here, only the last one) are automatically filled with blanks. We could go along very happily with this information alone, unless it occurs to us to wonder how SALLY can possibly be "spelled" in the binary bits (hex digits) that we now know constitutes a computer word.

TABLE 7-1.

<u>Instructions:</u>	
MVC	Move Character String
CLC	Compare Logical Character String
<u>Assembler Directives</u>	
BYTE	Generate Byte Address
OPD	Operation Define

*The limit of six happens to be true for languages implemented on a 48-bit machine. When implemented on a machine having a 16-bit word length (as the 980) the limit is two.

A glance at Table 7-2 shows that each character on a teleprinter (as well as some nonprinting activities such as ringing the bell or returning the carriage) has a 2-hexadecimal-digit (8-binary-bit) representation in *internal code*. Since our "FORTRAN machine" has a word length of 48 bits, that means we can store six 8-bit characters in one word. SALLY looks like this if we could see the binary bits:

1101 0011, 1100 0001, 1100 1100, 1100 1100, 1101 1001, 1010 0000,
 S A L L Y b

D3	C1	CC	CC	D9	A0
----	----	----	----	----	----

Since SALLY uses fewer than the eight characters it is entitled to, the unused space (here, only 1 character position) is automatically filled with blanks. (A blank has a hex shorthand of A0, whereas a zero has a shorthand of B0.)

As far as the computer is concerned, bits are bits and the collection shown above might be some negative number rather than someone's name. As programmers we know that in the context of the problem, the bit collection is supposed to represent SALLY and not the number D3C1CCCD9A0, though the same bit collection could be used to represent either one.

Let's see what happens if we try to add SALLY to JACK:

```

SALLY = D3C1CCCD9A0
JACK = CAC1C3CBA0A0
19E8390987A40
```

It would overflow the word boundary because of the carry into the leftmost position. If we were to discard the overflow bit, the remaining bits could be interpreted as either the number 9E8390987A40 or, searching the complete ASCII character table (see Appendix I), it could be translated into the characters:

<record separator> <end of text> <data link escape> <cancel> z @

This example makes it quite evident that we had best avoid use of character-words in arithmetic operations.*

Since it is possible for the machine to interpret a word as either a binary number or a collection of eight characters, we must specify in the FORTRAN read or write statement how that word is to be regarded. The machine will assume that the word has a numeric meaning unless we tell it otherwise by specifying alpha-*betic* (or "A") format.

For example,

```

N ="SALLY"
WRITE (6, 100) N
100 FORMAT (1X, A6)
END
```

*Remember what your third grade teacher said about adding apples to oranges!

TABLE 7-2. SELECTED ASCII CHARACTERS

Printed Character	Internal Code (after assembly) (leftmost bit = 1)	External Representation (before assembly) (leftmost bit = 0)
0	B0	30
1	B1	31
2	B2	32
3	B3	33
4	B4	34
5	B5	35
6	B6	36
7	B7	37
8	B8	38
9	B9	39
A	C1	41
B	C2	42
C	C3	43
D	C4	44
E	C5	45
F	C6	46
G	C7	47
H	C8	48
I	C9	49
J	CA	4A
K	CB	4B
L	CC	4C
M	CD	4D
N	CE	4E
O	CF	4F
P	D0	50
Q	D1	51
R	D2	52
S	D3	53
T	D4	54
U	D5	55
V	D6	56
W	D7	57
X	D8	58
Y	D9	59
Z	DA	5A
blank	A0	20
carriage return	8D	-
line feed	8A	- "action"
bell	87	-
+	AB	2B
-	AD	2D

would cause the machine to print the character string

SALLY

whereas the program

```
N = "SALLY"  
WRITE (6,100) N  
100 FORMAT (1X, I10)
```

would cause the machine to print whatever integer number the bits inside N represent.

The same holds true of input operations: READ, according to an A format, will cause the machine to expect a character string rather than a numeric value.

7-2 CHARACTER STRINGS IN 980AL: THE ASCII DATA DECLARATION.

If we were to write in FORTRAN

```
M = 24
```

the ultimate effect would be the same as if we had written in 980AL

```
M DATA 24
```

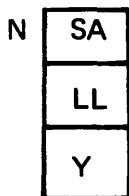
namely, there would be a cell named M containing the numeric value of 24_{10} . Let's see how we express in 980AL the equivalent of the FORTRAN

```
N = "SALLY"
```

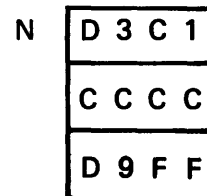
Since the 980 has only 16 bits per word, only two 8-bit characters can be packed into one memory location. We will use single quotes to indicate ASCII data declaration and write

```
N DATA 'SALLY'
```

which would assemble into:



or rather



7-3 NON-PRINTING CHARACTERS.

There are a few cases in which this method of loading character strings will not work. Assume that we are using a teleprinter as an input unit and we wish to have a character string input and then echoed back to us, followed by a carriage return <CR> and line feed <LF> to restore the teleprinter carriage to the first <CR> <LF> print position on the next line.

Our first inclination might be to type in

```
DATA 'SALLY <CR> <LF>'
```

and let the assembler translate the depression of the carriage return and line feed keys into the appropriate internal ASCII codes.

What happens, instead, is that when we depress the <CR> key, the computer assumes we have completed the data line, it sees that we are missing a final quote mark, and gives us an error.

The only way to make the printer respond with a carriage return/line feed during output is to store the appropriate numeric values (0D0A) in the last word to be printed on the line. As the machine attempts to print them, it will instead respond with the appropriate action. An example of this is found in the next section.

7-4 BUFFERS.

When information is taken into the computer from an external input device, it must have some place to go. The programmer is responsible for providing that "some place" by setting aside a block of memory locations known as an *input buffer*. Information sent to an external device for output first will be gathered into a similar block of memory known as an *output buffer*.

Forgetting about input for a moment, let's consider a program whose sole job is to output message characters, do a carriage return and line feed, and stop. We can predefine the contents of the output buffer using the DATA declaration (since no computations are necessary in this example):

```

          BUFOUT          IDT          MESSAGE
          DATA          'THE TIME IS NOW',>0D0A
          <output instruction(s)> *
          IDL
          END
    
```

The assembler would produce the following machine code from this program

BUFOUT	Meaning
D4 C8	TH
C5 A0	Eb
D4 C9	TI
CD C5	ME
A0 C9	bI
D3 A0	Sb
CE CF	NO
D7 FF	W
0D 0A	<CR> <LF>
<output instructions>	-
CE00	<idle>

*We'll defer discussion of the output instruction format to the next section.

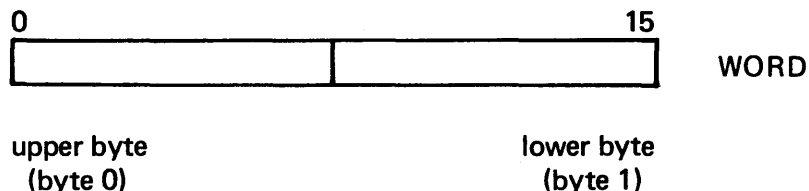
The output instruction(s) must specify what device is to be used for output, where the buffer is located in the memory (BUFOUT), and how many words long it is (nine).

The entire contents of the buffer will be output as one line on the output device:

THE TIME IS NOW

7-5 BYTES AND BYTE MANIPULATION.

In the 980, it is possible to have individual addressing not only of words, but of half words (bytes) as well. Since one *byte* is the proper size to hold one ASCII character, the byte manipulation operations are rather useful in processing character strings:



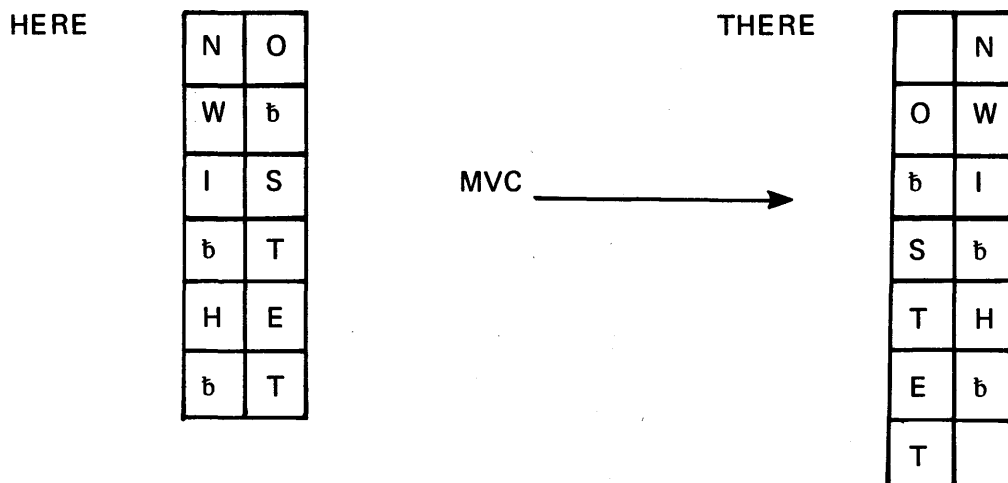
When we wish to specify the location of a byte, we need to specify not only the word, but which half we are talking about.

7-5.1 MOVE CHARACTER STRING (MVC). We can move a character string of some specified number (m) of bytes from one starting location to another using the

<label> MVC

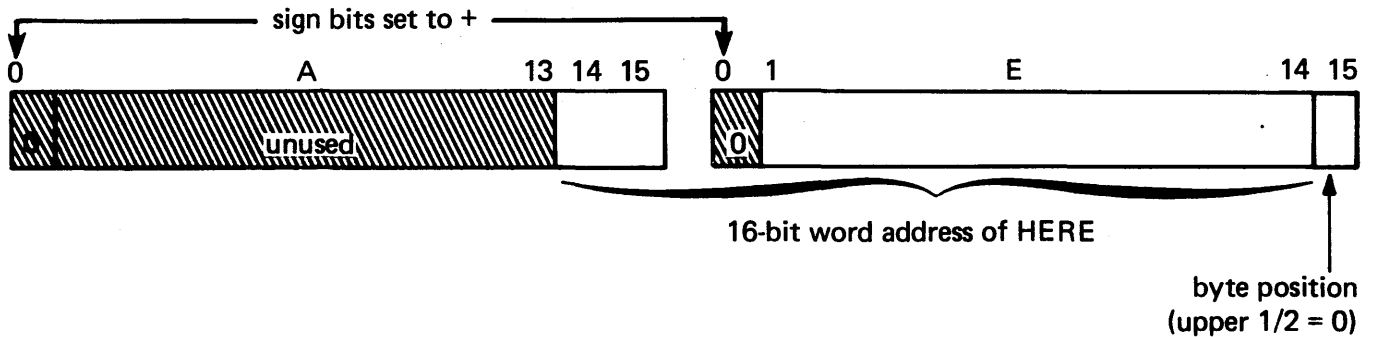
instruction. There is no operand in this instruction, and its use is predicated on the assumption that the A, E, M, S, and X-registers contain information needed to execute the instruction.

Let us consider the example of moving a 12-character byte string from consecutive bytes, starting with the upper half (byte 0) of the word **HERE**, to a block of consecutive bytes, starting with the low order byte (byte 1) of the word **THERE**.

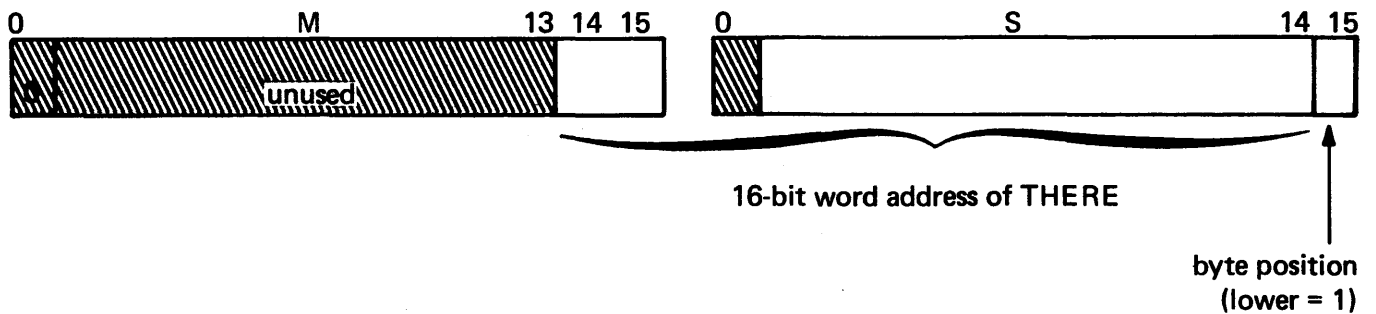


We do the following:

1. Load the byte count $M = 12_{10}$ into the X-register. As the bytes are moved, this count is automatically decremented, reaching 0 when the transfer is complete.
2. Load the word address and byte position of the first source byte into the A- and E-registers according to the following pattern:



3. Load the word address and byte position of the destination block into the M- and S-registers:



(Notice that a sign bit (+) is inserted to indicate that the two upper address bits will spill over into the left register.)

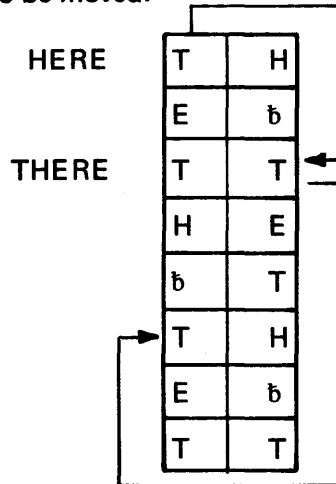
One means of achieving this configuration* is to load the A- and E-registers with data for THERE, long shift to achieve the positioning of the bits, and do a Register Move to get the numbers from A and E into M and S. Then we do the same with the data for HERE, leaving it in the A- and E-registers:

LDE	=0	CLEAR E
LDA	=THERE	DESTINATION WORD ADDRESS TO A
ARD	14	ARITHMETIC SHIFT 14 BITS TO E
SABZ	0	FORCE SIGN BITS TO ZERO
DAD	=1	ADD 1 FOR INDEXING RIGHTMOST BYTE
RMO	A,M	SHIFT DESTINATION
RMO	E,S	TO MS REGISTER
LDE	=0	CLEAR E
LDA	=HERE	SOURCE WORD ADDRESS TO A

*There is an easier way!

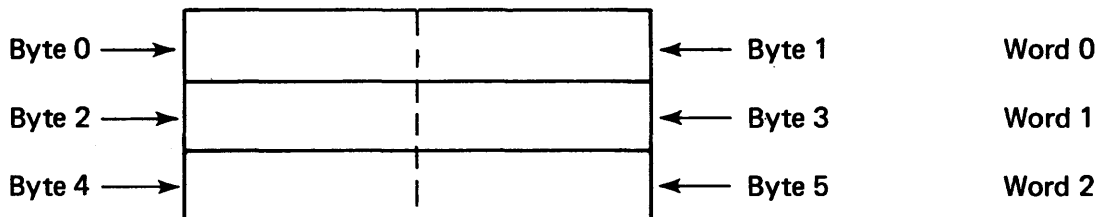
ARD	14	ARITHMETIC SHIFT 14 BITS TO E
SABZ	0	FORCE SIGN BITS TO ZERO
LDX	<byte count>	SET BYTE COUNT AS INDEX
MVC		MOVE BYTE STRING

Beware of the pathological case in which the two blocks **HERE** and **THERE** overlap, since some bytes will be replaced before they get the chance to be moved:



7-5.2 GENERATE BYTE ADDRESS: THE BYTE DECLARATION. The byte manipulation instructions MVC and CLC assume that *byte addresses* appear in AE and MS registers. The program fragment in the previous subsection generated the proper byte addresses through a multiply-by-2 (left shift by 1 bit); generate space for a sign (left shift by another bit); and add 1 for lower byte address, 0 for upper byte address.

The internal scheme for referencing bytes is to start counting using the upper byte of word zero as byte 0:



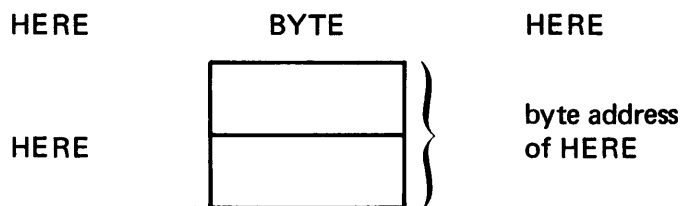
Thus we could reference any byte by specifying its byte address in those instructions (i.e., MVC, CLC) which expect a byte address to be provided. Byte addresses are calculated from word addresses, and vice versa, through the relation:

$$\langle \text{upper byte address} \rangle = 2 * \langle \text{word address} \rangle$$

$$\langle \text{lower byte address} \rangle = 2 * \langle \text{word address} \rangle + 1$$

There is a **BYTE** declaration that works much the same way as the **DATA** declaration (Section 4-6.2) except that it generates the byte addresses for us. Each byte address is assembled into a word pair, with the

least significant bits in the second word and the label (if any) attached to the first word:

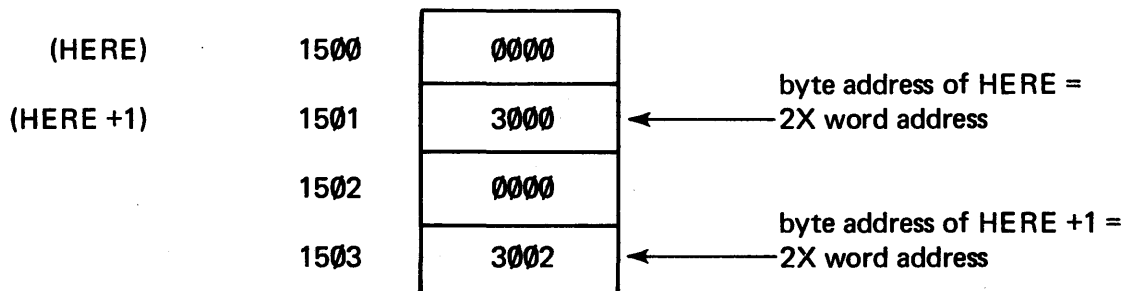


The byte address is calculated by taking 2* operand, so if we had this declaration ORG'd so that HERE is at word address 1500 (hence HERE +1 is at 1501)

```

HERE          ORG      > 1500
              BYTE     HERE
              BYTE
              BYTE     HERE+1
    
```

we would obtain the assembled form:

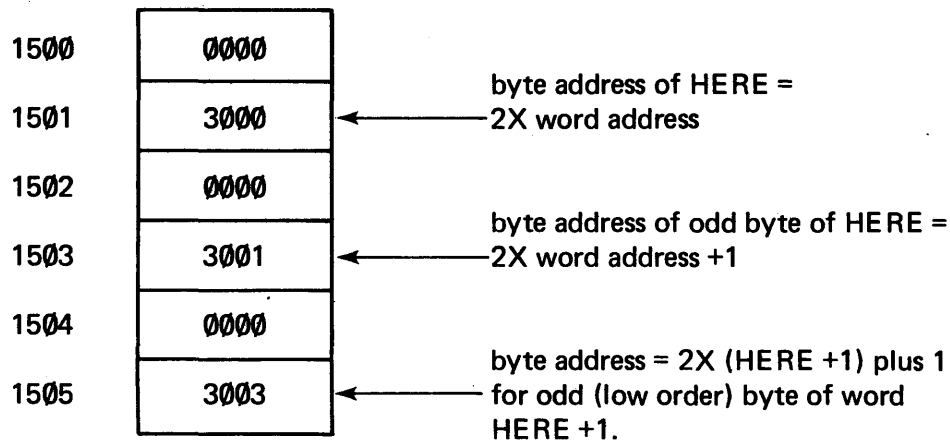


Notice this always gives us the leftmost (or "even") byte. To get the odd byte we add "1" to the byte address, and this is designated by prefixing the expression with a colon; e.g.,

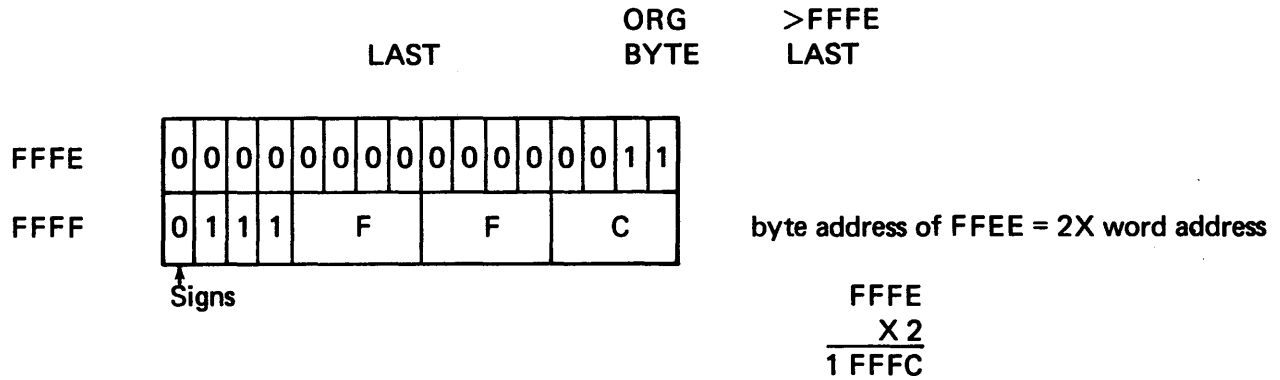
```

HERE          ORG      >1500
              BYTE     HERE
              BYTE     :HERE
              BYTE     :HERE+1
    
```

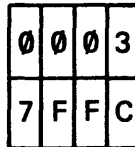
gives



The byte addresses are forced into being signed quantities, with the upper 2 bits of the original word address being forced into the low order bit positions of the upper word in the word pair (c.f. the format of the AE register discussed in Section 7-5.1). This forcing makes no difference to the example above, because the upper 2-address bits are zero anyway. However, consider the example where the word address is FFFE:



or, collecting all the bits into hex notation:



The BYTE directive does all this so that the byte addresses are ready for loading into the AE and MS registers prior to a character operation:

ADDR1	BYTE	HERE	(two words)
ADDR2	BYTE	:THERE	(two words)
<entry>	DLD	ADDR2	
	RMO	0,3	
	RMO	1,4	
	LDA	ADDR1	
	LDE	ADDR1+1	
	MVC		

7-5.3 COMPARE LOGICAL CHARACTERS (CLC). The CLC instruction requires a setup identical to that for the MVC instruction (see Sections 7-5.1 and 7-5.2). CLC performs a byte-by-byte comparison of two byte strings where the *source* string begins in the address found in the AE register, and the *destination* string begins in the address found in the MS register. The bytes are treated as unsigned quantities in the comparison.

The action of the instruction is to set the compare bits in the status register in the same manner as the other compare instructions:

	status code
source < dest	00
source = dest	01
source > dest	10
(not allowed)	11

The compare is terminated by the first nonequal comparison and the X-register contains the number of bytes remaining in the string (i.e., *bytes to go*). If X is initially zero, no comparison is performed and the outcome is set to the "=" code unconditionally.

7-6 CONVERSION FROM INTERNAL TO EXTERNAL FORMAT: INTEGERS.

When an output instruction is issued, the output buffer is regarded as a collection of ASCII characters and is output accordingly. This condition applies regardless of whether the buffer contains a message; for example:

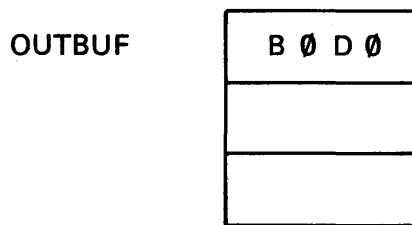
```
OUTBUF DATA 'NOW IS THE TIME'
```

or a number.

Input numbers are converted for us automatically by the assembler during the translation process. For example, if we declare the decimal number:

```
OUTBUF DATA -20272
```

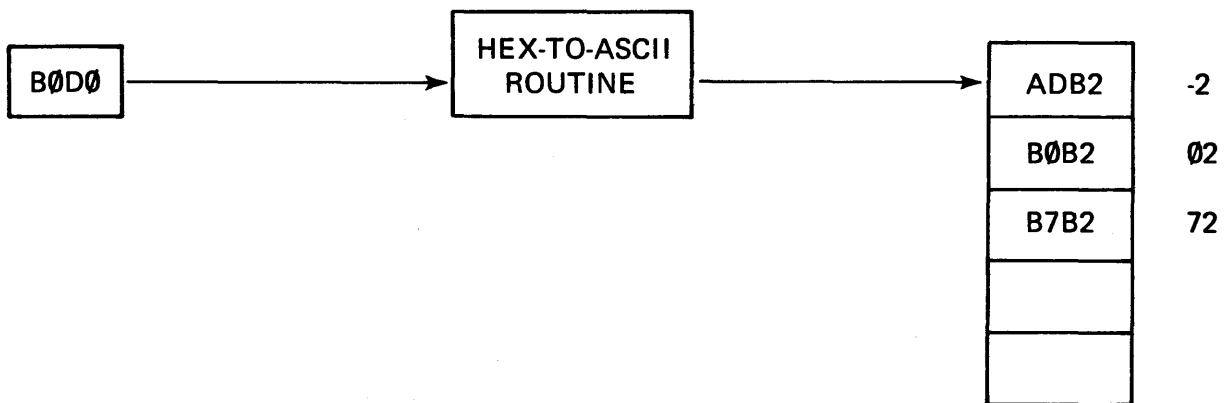
the assembler will treat this as a negative decimal number and translate it to the corresponding hex value:



If we now ask for an output of OUTBUF, the contents will be interpreted as the two characters B0 and D0 and the characters

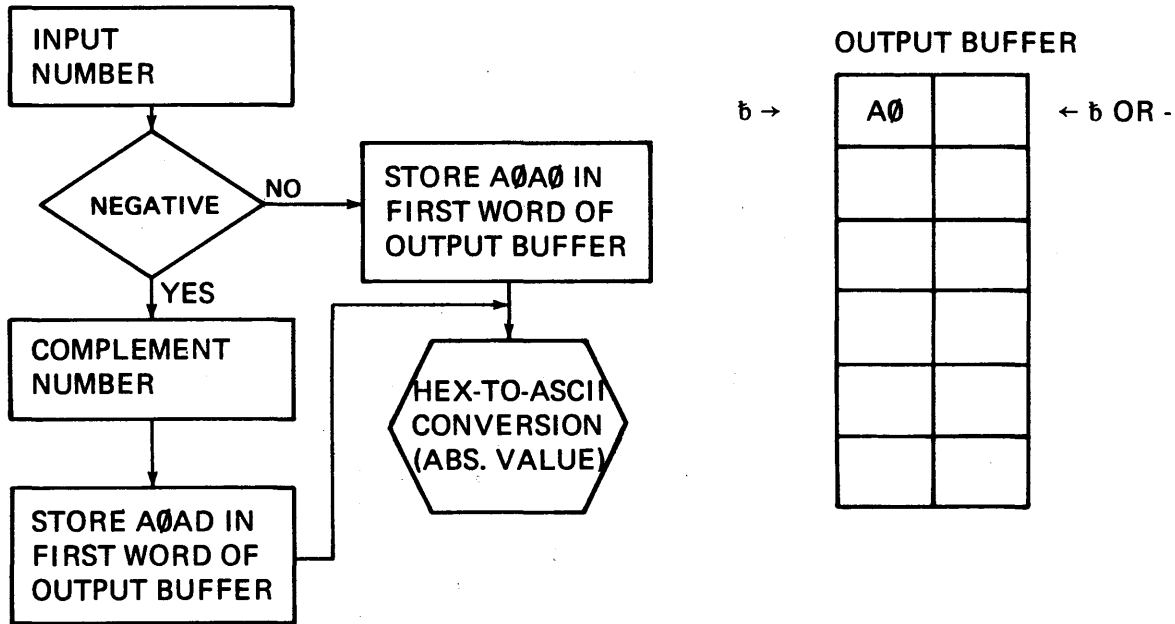
0P

will appear on the output device. This is a far cry from the -20,272 we originally entered. We shall need some kind of a conversion process into which we can feed the hexadecimal internal value of B0D0 and have come out of the buffer the characters representing -20,272:

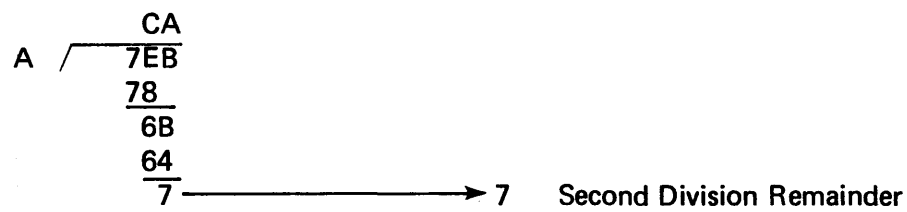
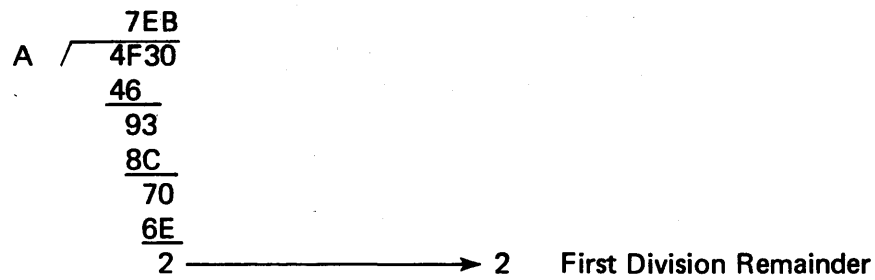


Since the comma is not part of the value of the number, it is not included. We could, of course, count the output characters and move them apart to insert commas in the appropriate places if we desire.

Let us agree that our HEX-TO-ASCII routine will handle only positive numbers. A number to be converted first will be tested for sign and the sign saved. If the number is negative, we will insert a minus character (AD) as the least buffer element.



Complementing the number B0D0 (two's complement) gives 4F30 as the absolute value to be converted; i.e., 20272₁₀. We'll use the conversion scheme discussed in Section 2; i.e., successive division by 10₁₀ (=A₁₆) and collecting the digits of the successive remainder: The tables of hexadecimal arithmetic in Appendix C are useful.



```

A  / 14
    CA
    A
    2A
    28
    2 -----> 2   Third Division Remainder

```

```

A  /  2
    14
    14
    0 -----> 0   Fourth Division Remainder

```

```

A  /  0
    2
    0
    2 -----> 2   Fifth Division Remainder

```

The successive-division process stops when the quotient becomes zero. Since the digits 0 through 9 are represented by the ASCII characters B0 through B9, all we need do is prefix a B to each digit (i.e., add B0 to the digit). As each digit is produced, it is stored in a pushdown stack, STAK. Upon encountering a zero quotient the stack contents are popped up, and each pair of stack entries is packed into the next word of the buffer. This process assumes that STAK has been initialized with blanks, in case of an odd number of stack entries.

The packing process followed here involves shifting the first character of the pair into the left half of the A register and performing an inclusive OR of the second character to the A register. The process could be done by a series of shifts and long shifts, but that would consume twice the time.

STAK	BES	10	
PTR	DATA	OBUF	
ENTRY	LDX	=0	INITIALIZE STACK POINTER
	LDM	=>B0	BLANK CHARACTER IN LOWER B
	LDA	INPT	GET INPUT NO. (ABS. VAL.)
LOOP	MPY	=1	EXTEND SIGN (+)
	DIV	=10	DIVIDE BY 10
	RAD	M,E	ADD B0 TO REMAINDER
	RDE	X,X	DECREMENT STACK POINTER
	STE	STAK,X	AND SAVE DIGIT IN STACK.
	SZE	A	ZERO QUOTIENT?
	BRU	LOOP	NO, GO BACK
AGAIN	IMO	PTR	YES, UPDATE BUFFER POINTER
	LDA	STAK,X	POP FIRST CHARACTER OFF STACK
	RIN	X,X	INDEX SECOND CHARACTER
	CRA	8	SHIFT CHARACTER TO LEFT HALF
	IOR	STAK,X	PACK SECOND CHARACTER ON RIGHT
	STA	*PTR	AND SAVE IN OUTPUT BUFFER

RIN	X,X	INDEX FIRST CHARACTER OF PAIR
SPL	X	STACK EMPTY?
BRU	AGAIN	NO, GO BACK
:	:	YES, BUFFER READY FOR OUTPUT
END	ENTRY	

In the case of -20,272, the buffer now appears as:

OBUF	A0AD
	B2B0
	B2B7
	B2A0

or

b-
20
27
2b

which, when output on a single line, gives

b-20272b.

Problem: For practice, manipulate the output buffer to insert a comma to separate the hundreds from the thousands column.

7-7 HASH TOTALS.

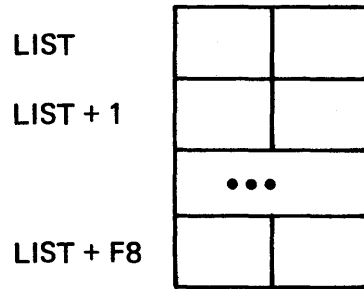
We briefly mentioned in Section 6-8 that one technique of building a list for easy searching, especially if the list contains character data (ASCII, in the case of the 980), is the so-called *hash* technique. Using this technique the hexadecimal codes of the character are added together and the result is used as the pointer to a table. The same algorithm is used later in searching for the item in the table. Consider the four-character word CATb. The ASCII code representation is two 16-bit words (see Appendix I for a complete ASCII table):

C	A
43	41
T	b
54	20

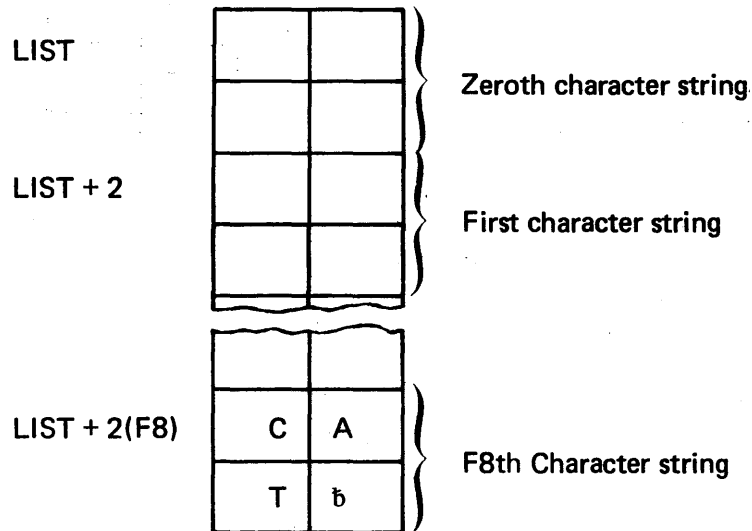
and the hash total of CATb is the sum of the hexadecimal codes:

$$\begin{array}{r}
 43 \\
 41 \\
 + 54 \\
 \hline
 20 \\
 \hline
 F8
 \end{array}$$

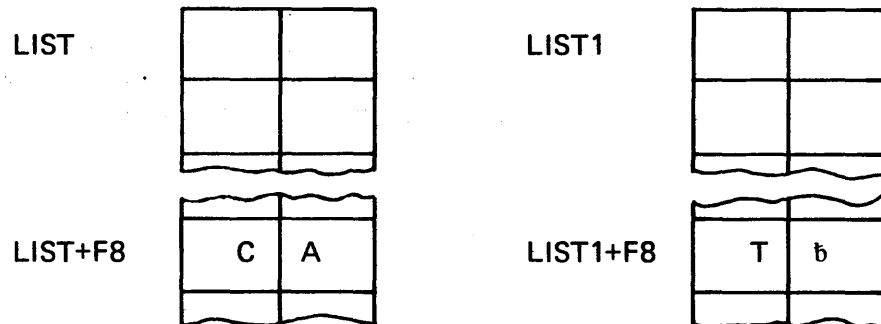
The hash total F8 is always obtained by adding the digits CATb. We can place the item in a list, using F8 as the index:



Since the 16-bit word can only contain two ASCII characters, we either have to use two consecutive words in the same list in order to store the four characters (which will require us to double the size of the index):



or we have to use two lists, say LIST and LIST1, where LIST holds the first two characters and the corresponding item in LIST1 holds the last two:

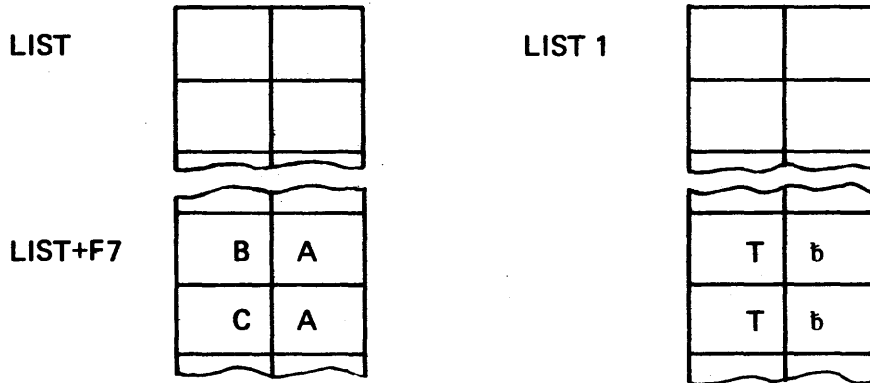


If at some future time we wish to determine if CATb is in the list, we add its characters, index the list, and see if the characters we find there match the ones we are seeking.

If we put BATb in the list, the index assumes a different value.

B	A	or	42
42	41		41
			+ 54
T	b		20
			F7

so that BATb appears in the slot above CATb:



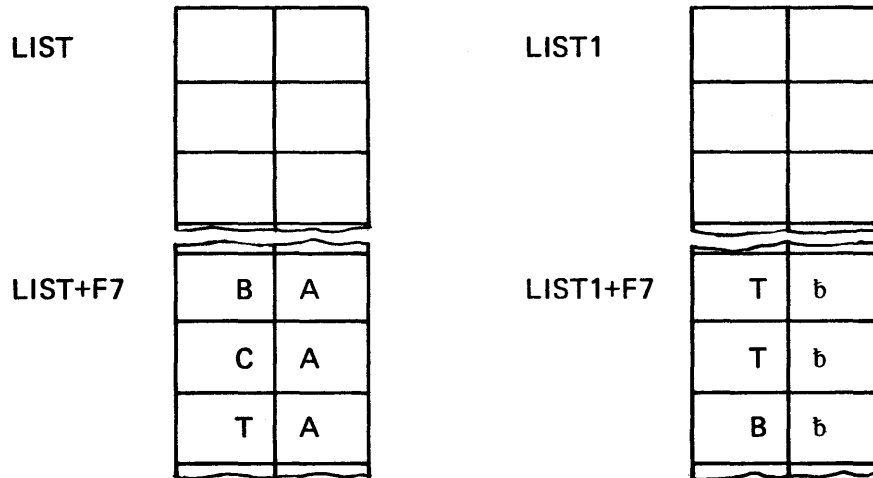
Everything is satisfactory (although the table may be sparsely filled) until we encounter a word that is different, but which has the same total as a word already in the list:

T	A		54
54	41		41
			+ 42
B	b		20
			F7

42	20
----	----

Slot number F7 is full, so we cannot store TABb there.

There are various means to cope with this situation, but we won't concern ourselves with any but the simplest; namely, if the cell we wish to use is already occupied by some other word, we start from that word and perform a sequential search until we find the first empty cell and use that.



To retrieve the word we simply follow the same process: compute the hash total and start searching the list from the cell pointed to by the hash total. We will either find the word before we encounter the first blank, or we can be sure the word is absent. In some cases it may be expedient to scale the hash total before using it as a pointer.

Even though this example is not the most efficient use of the hash technique, it represents an improvement over a sequential search of the entire list starting at the top.

7-8 I/O SERVICE CALLS.

7-8.1 SERVICE CALLS VS. THE BARE MACHINE. The TI980 may be used as a bare machine under direct control of the operator or under the control of a supervisory operating system (or *monitor*). When a number of different jobs are contained in the job stream, it is easier and more efficient to use the operating system. The main differences in programming between the two is the way I/O and program termination is handled.

The supervisor or operating system is itself a program resident in the memory. Since it consumes space, the program and data must be small enough to fit the remaining space

— OR —

- Parts must be capable of being called in as needed from an external device to overlay existing memory

— OR —

- The supervisor may be eliminated and the 980 used as a "bare machine".

Some special purpose programs, especially real-time systems, require that the computer be used as a bare machine or under a special purpose operating system written by the user. For training purposes, though, we should examine a method permitting us to at least get some programs up and running on the machine.

I/O is almost invariably the most complex task a machine is called upon to perform, and this is true of the 980. The author recommends use of the operating system to the beginner, even though it may not be clear

at the outset just what is taking place in the system. The beginner who is unwilling to accept anything on faith (even if temporarily) is certainly free to tackle I/O using the bare machine. The programmer should ultimately, of course, learn both ways.

7-8.2 I/O AND PROGRAM TERMINATION USING SUPERVISOR SERVICE CALLS. In order to avoid changing the //ASSIGN commands, it makes sense to have the user program maintain the same lun (logical unit) assignments used for the assembler. This assumes that the user data is in the same form of input as the SAPG assembler; i.e., in this example, cards. Thus, read-user-data instructions are issued to lun 5, and print-user-results instructions are issued to lun 6.

Physical record blocks (PRB's) must be defined for every device operation: these are blocks of numbers needed for the I/O operation, and the address of the block must appear in the M register at the time the I/O operation is requested.

1. A device must be "opened" before it is used through use of a *physical record block*.
2. A read or a write is specified through use of a *physical record block*.

The format of a PRB is as follows

<label> DATA <lun>,<l.u.op>,<character count>,<buffer label>,0

Since each device must be opened, we'll have two device opener blocks, one each for logical 5 and logical 6 input device.

Since <l.u.op> code = 7 for opening a device, these blocks are

PRBOP5	DATA	5,7,60,
PRBOP6	DATA	6,7,130,

Opening the devices is not enough, so we also need a data block for use by the read/write. The read operation uses one of two codes, depending on whether the input is in ASCII characters (operation code 0) or binary object code (operation code 1). Similarly, the write operation uses operation 2 if ASCII code or operation 3 if object code.

Since we wish to have the I/O in character (ASCII) mode, the two physical record blocks are:

<i>for read:</i>	PRBIN	DATA	5,0,60,IBUF,0
<i>for write:</i>	PRBOUT	DATA	6,2,130,OBUF,0

A table of logical unit operations is given in Appendix K.

Since the supervisor performs the actual I/O and program termination operations for us, we need some means of, first, attracting its attention, and second, telling it what we want.

We attract its attention by trying to execute a special kind of "illegal instruction" known as a *supervisor service call*. The supervisor interprets the particular type of instruction and relinquishes control to one of its resident service programs.

The two service calls we are concerned with at the moment* are the instructions

>C380	(I/O request)
>C381	(Program termination)

which are of the form C38<n>, where <n> :: = 0/1/2/3/4.

Since there is no difference, as we have seen, between instruction-numbers and data-numbers, we could declare a data number

DATA > C380

to be in the location of the supervisor service call for I/O, and

DATA > C381

in place of the IDL instruction. Or we could be a bit more sophisticated in our approach and define a generalized operation which could be used for any type of service call, merely by changing the specific values of the operands. This option is discussed in the next subsection.

7-8.3 DEFINITION OF SERVICE CALLS WITH OPD. One use of the multipurpose *operation define* (OPD) pseudo-op permits us to define a whole family of illegal instructions that can be used as supervisor service calls. It is written

<label> OPD <number>,<format>

where the <label> is chosen to be the same as the actual mnemonic op-code we want used in the executing program. The <number> appearing in the definition is the hexadecimal value we wish to have as the "root" of our family of operations. (Recall that this family is >C380, >C381, >C382, >C383, and >C384.) Thus, the "root" will be >C380. The root value may be modified easily to produce any specific family member by adding to it some digit between 0 and 4. We'll specify the digit as an argument of the <label> operation in the executable program:

<label> <operand digit>
↑
in op code field

The fate of the operand (i.e., how it is to be used in modifying the root) is determined by the <format> of the instruction in the OPD definition.

*The others are

>C382	(set floating point package address)
>C383	(get memory limits)
>C384	(set control status flag).

To select a format it is necessary to know exactly what each instruction type does with its operands at execution time.*

It happens that <format> type-3 operations (register shift instructions) have their operand fields added to their operation codes as a first step of execution. If we use an operand =1, this is added to the basic C380 to get C381, signifying a return to the monitor. If we use an operand of 0, we get C380, signifying an I/O request. The type of request is identified as I or O, data transfer or logical unit open/close, buffer to/from which transfer will occur, and the number of characters, by inspecting the contents the cell pointed to by the M register. The address of the physical record block must be loaded prior to the issuance of the calls by using the sequence:

@LDM =<address of PRB>

A full illustration is given in the next subsection.

7-8.4 PROGRAM SKELETON: I/O AND PROGRAM TERMINATION USING SUPERVISOR CALLS.

nonexecutable declarations	}	SVC	OPD	>C380,3	DEFINE SUPERVISOR CALL
		RDROPN	DATA	5,7,60,IBUF,0	PRB FOR RDR OPEN
		LPOPEN	DATA	6,7,130,OBUF,0	PRB FOR PRINTER OPEN
		RDBLK	DATA	5,0,60,IBUF,0	PRB FOR READ 60 ASCII
		LPBLK	DATA	6,2,130,OBUF,0	PRB FOR PRINT 130 ASCII

<entry point>

—
—
—
—
—
—

@LDM =RDROPN OPEN
SVC 0 CARD READER.

*For reference, the format types are listed here:

<format> :=		Type
	null	Register to memory
	0	Register to memory
	1	Register to memory
	2	Register to register
	3	Register shift
	4	Register skip
	5	Status indicator skip and idles
	6	Data bus I/O
	7	Sense switch skip and register bit
	8	DMAC and auxiliary processor

```
@LDM =LPOPEN      OPEN
SVC 0              LINE PRINTER.
```

—

—

—

—

—

```
@LDM =RDBLK      ISSUE
SVC 0              READ.
[60 characters are now available in IBUF]
```

—

—

—

—

—

```
@LDM =LPBLK      ISSUE
SVC 0              WRITE.
[130 characters have now been sent from OBUF
to the line printer]
```

—

—

—

—

—

```
SVC 1              RETURN TO SUPERVISOR
END
```

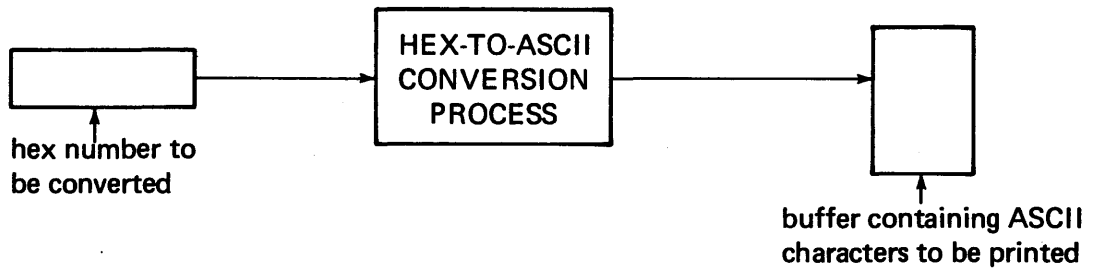
The values in the physical record blocks may be changed under program control. For example, if the size of an output record is to be changed, say to 25, then storing the value 25 in LPBLK+2 prior to the @LDM = LPBLK instruction will do the job.

SECTION 8
SUBROUTINES

8-1 THE SUBROUTINE: A LABOR-SAVING DEVICE.

The instructions discussed in this section are shown in Table 8-1.

In some programming tasks, the same subtask must perform at a variety of points within the code stream. A good example of this is the hexadecimal-to-ASCII conversion discussed in Section 7-6:



If the overall program looked something like this.

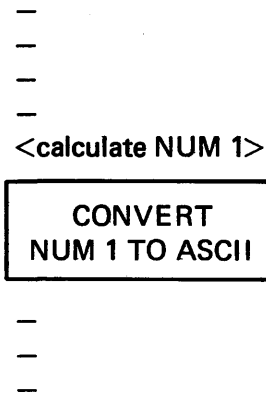
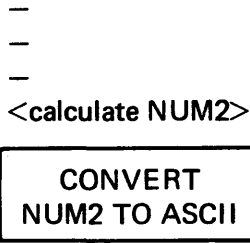
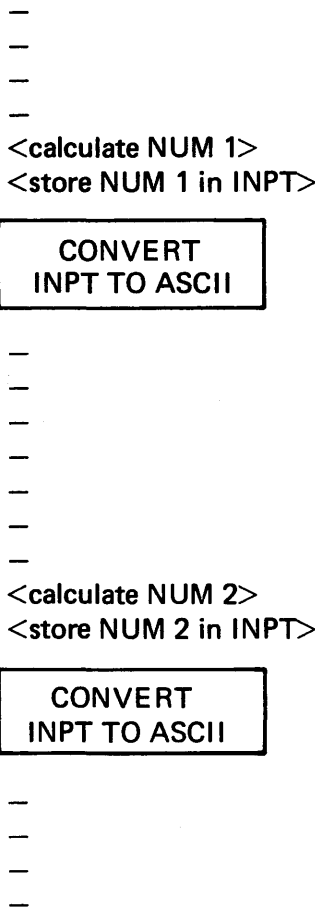


TABLE 8-1.

<u>Instruction</u>	
BRL	Branch and Link
<u>Assembler Declaratives</u>	
DEF	Define Entry Point
REF	Reference External Symbol
COMM	Common



we would feel some justifiable reluctance to place the hex-to-ASCII conversion code in the main program stream every time it is needed. It is more reasonable to code the process once and branch to it whenever the need arises:



Once the value of NUM1 is stored in INPT, the program branches to the hex-to-ASCII code block and performs the conversion. At the end of the code block, a branch back (*return*) to the main code stream can be performed only if the value of the PC has been saved before branching off to (*calling*) the conversion block.

The hex-to-ASCII code block described here is a primitive form of subroutine.*

8-2 THE PRIMITIVE SUBROUTINE LINKAGE PROBLEM.

When a program calls a subroutine, certain data must be recorded:

1. The PC must be given an address to which to go
2. The address to which to return after subroutine execution must be saved
3. If any data values are to be passed between the two programs, provision must be made for their exchange.

Let us first consider the case in which no data exchange occurs. A practical example of such a routine is one in which the time of day is to be printed out. The routine needs access to the real-time clock, carries out the appropriate operations, and prints the result. No data need be sent from the calling program or returned to the calling program on completion of the routine. The skeleton structure is

```
<label> <first instruction of subroutine>
-
-
-
RMO L,P restore PC to return ADDR
```

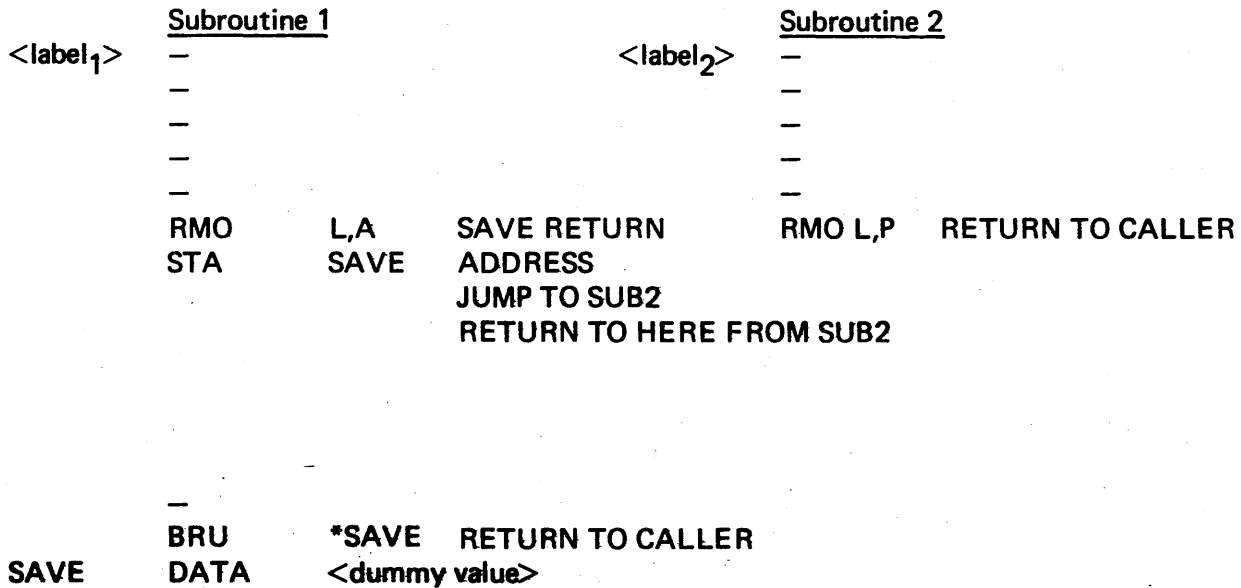
and for the calling program:

```
IDT <name of main program>
HED <page heading>
<entry> - <first instruction>
-
-
-
-
@ BRL <label>
- (control returns here after SBRT execution)
value of -
PC saved -
in LINK -
```

Use of the BRL instruction assumes that the link (L) register is available since the insertion of the present PC contents is saved in the link register, and the entry point of the subroutine is entered into the PC. The register-move (RMO) instruction as the last executable subroutine instruction restores the PC to the return address. It also assumes that the subroutine need not make direct use of the link register during the execution.

*The designation "primitive" is applied because of the assumption that the code block comprising the subroutine is assembled as part of the main program. In Section 8-4 we'll examine a more refined form of subroutine; i.e., one which is assembled separately.

Let's assume that the subroutine must, in turn, issue another subroutine call. The link register will be needed, so it must be freed for use. We can store the contents of the link register in memory, and in order to return, do a branch indirect to that memory location



8-3 PARAMETER PASSING TO A PRIMITIVE SUBROUTINE

The three commonest parameter passing techniques may be classified as

- call by name
- call by address
- call by value

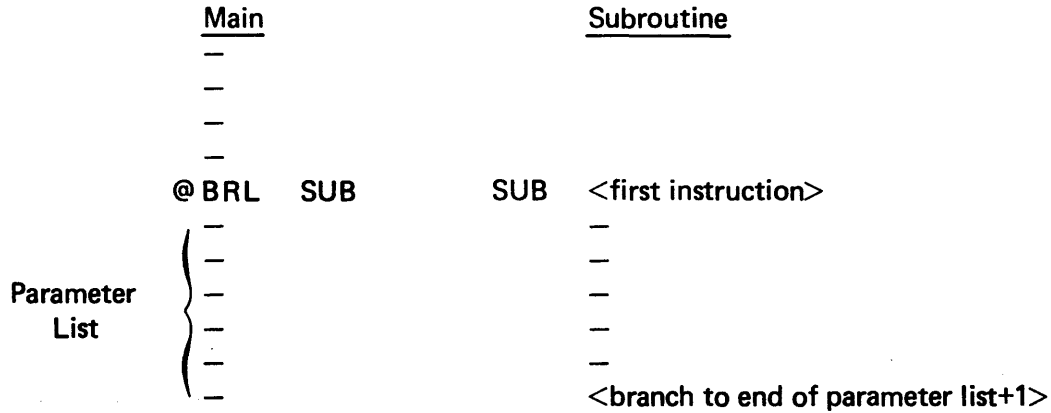
depending on what actually passes across the interface between the calling program and its subroutine.

If the actual value of the parameter (i.e., a copy of it) is sent across, it is a *call by value* technique. If it is the address where the value is to be found, the technique is *call by address*. Call-by-address is inherently dangerous since a subroutine can accidentally change a value in the main program because it has access to the actual storage locations (compare the discussion of COMMON in Section 8-6.1). In call-by-value the subroutine is given only a copy of a number to work with and no way to locate the address of the original value from which the copy was made.

8-3.1 PARAMETER PASSING VIA REGISTERS (CALL BY VALUE)

The only register affected by a BRL subroutine call is the L-register (and, of course, the PC). The other registers may be filled with parameters just prior to the issuance of the call; the subroutine must be cognizant of this fact and behave accordingly, either by using the parameters from the registers directly or storing them for future use. Since the number of registers is limited, this technique will not serve in all applications. Besides a certain amount of unavoidable overhead is involved in the LOAD/STORE operations which must occur on both sides of the interface.

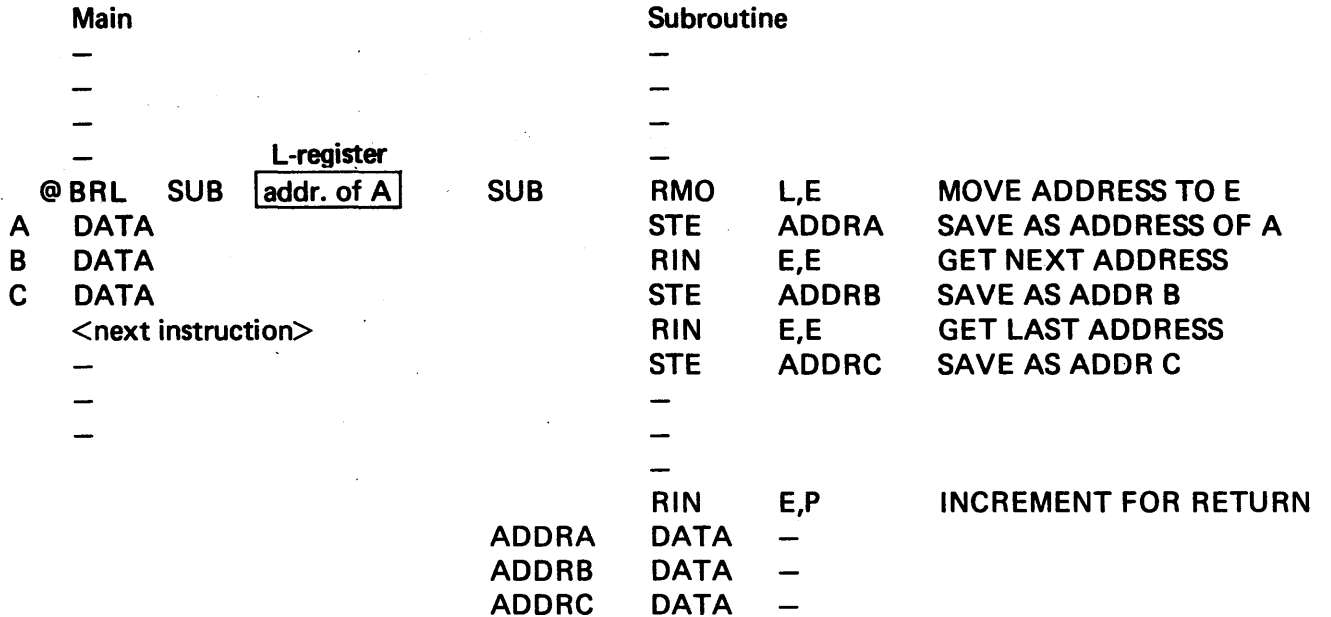
8-3.2 IN-LINE PARAMETER LIST (CALL BY ADDRESS). It is sometimes convenient for the parameter list to be passed to a subroutine to appear in the main program immediately following the subroutine call:



At the time the BRL is executed, the link register will contain not the return address, but the address of the first item in the parameter list.

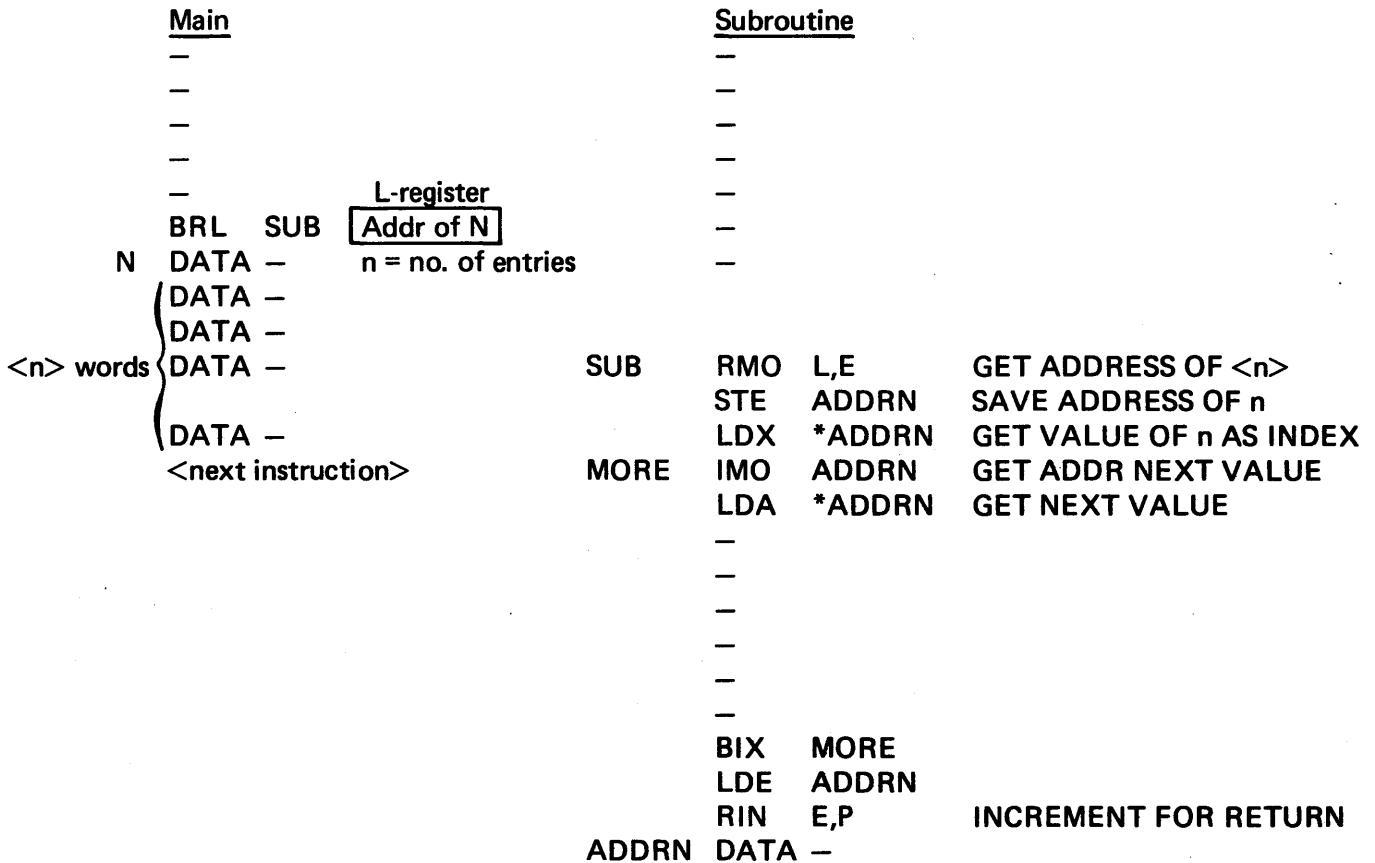
The list may be constructed in one of three slightly different ways as follows. (We'll assume in all cases that the E-register will not be needed during the subroutine execution.)

8-3.2.1 Fixed Number of List Entries. The subroutine "knows" that there will be exactly, say, three items in the parameter list and increments the *return address* accordingly before executing the actual return:

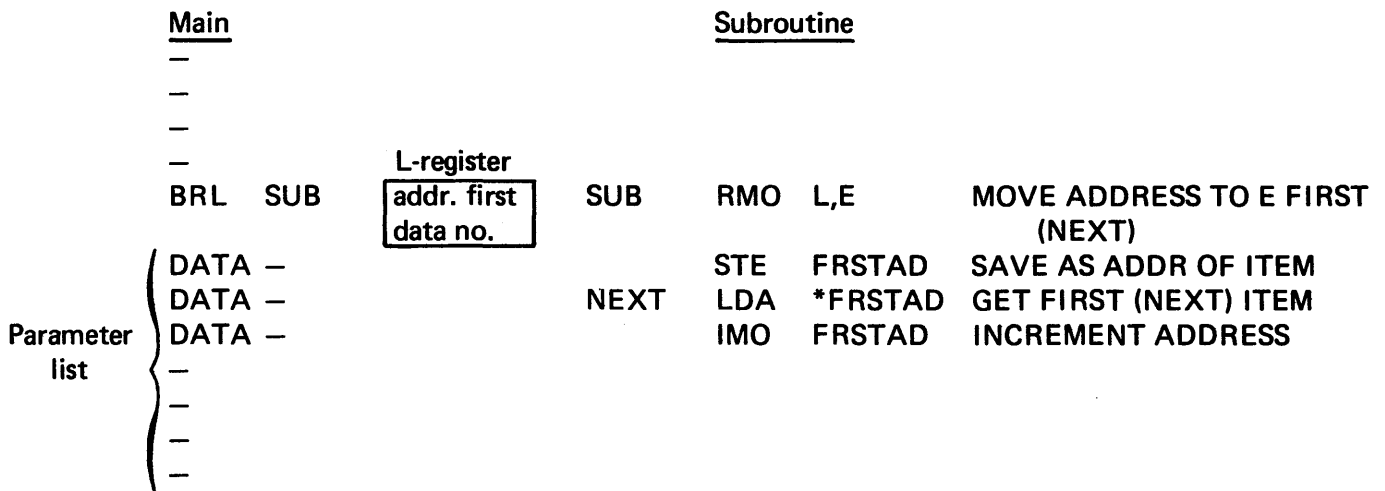


8-3.2.2 Variable Number of List Entries

a. Variable Length List with Header. In this case, the link address gives the number of data items in the following list:



b. Variable Length List with Stopper Value. In this example, the last DATA value stored in the list is a flag indicating there is no more data. We'll use 0 as an example here (note the E-register is freed immediately):



DATA 0	SZE	A	END OF LIST
<next instruction>	BRU	NEXT	NO
	LDE	FRSTAD	YES, GET RETURN ADDR
	RMO	E,P	AND TRANSFER TO PC.
	FRSTAD	DATA	-

8-3.3 IN-LINE ADDRESS OF PARAMETER LIST. It is possible to place the address of the data in the location following the call:

	<u>Main</u>			<u>Subroutine</u>	
	-		SUB	RMO	L,E
	-	L		STE	DBLOK
	-	addr of LISTAD		LDA	*DBLOK
	-			STA	BBLOK
	-			LDA	*BBLOK
	-			-	
	BRL	SUB		-	
LISTAD	DATA	LIST		-	
	<next instruction>			-	
	-			-	
	-			-	
	-			LDE	DBLOK
	-			RIN	E,PC
LIST	DATA	-	DBLOK	DATA	GET ADDR LISTAD
	DATA	-	BBLOK	DATA	INCREMENT FOR RETURN
	DATA	-			
	DATA	-			

Note that we have programmed our way through two levels of indirection: DBLOK contains the address of LISTAD. If we load (indirect) the value pointed to by LISTAD, we get the address of LIST, which we save as BBLOK. (Thus LISTAD and BBLOK contain the same value; i.e., the address of LIST.) If we load (indirect) the contents of BBLOK, we get the first item in the LIST. This list can be managed the same way as an in-line list (Section 8-3.2) with either a fixed number of values or a header or stopper value.

8-4 FORMAL STRUCTURE OF A SUBROUTINE.

Subroutines are usually constructed as standalone programs, assembled separately from the calling program. Although this kind of structure increases their versatility (i.e., the object code may be moved from one job to another without reassembling), it complicates the problem of linkage and parameter passing.

In the "primitive" subroutines discussed in previous sections, linkage is done by the assembler at translation time. In formal subroutines linkage must be done by the loader just prior to execution.

A separately assembled subroutine has its own IDT and END declaratives:

<u>Main</u>		<u>Subroutine</u>
IDT	MAIN	IDT SUB
HED	<page heading>	HED <page heading>
-		-
-		-

```
—  
—  
—  
END
```

```
—  
—  
—  
END
```

and in both the subroutine linking and the calling program there must be some kind of declaration to tell the link editor where the interface points are, so that the linkage can be completed before load time. These declarations are made with the DEF and REF pseudo-ops discussed in the next subsection.

8-5 ENTRY POINTS AND EXTERNAL SYMBOLS: DEF AND REF.

Any mnemonic labels used in a separately assembled routine are *local* to that routine (i.e., other routines will not be able to identify the label unless provision is made for them to do so). This request for communication at load time is accomplished by using the DEF (define entry point) and REF (reference external symbol) assembler declaratives. (An *external symbol* is a symbol defined in a different routine.)

DEF tells the link editor that the subroutine entry point name and address must be recorded, because some external routine will be transferring control to that location in the form of a subroutine call. There may be more than one entry point, in which case the entire list is declared:

```
          IDT SUB  
          DEF <entry point1>, <entry point2>  
          —  
          —  
          —  
          <entry point1>  
          —  
          —  
          —  
          <entry point2>  
          —  
          —  
          <branch to address contained in link register>  
          END
```

When a subroutine entry point is called, the calling program will look for a local symbol having the name of the entry point. In the case of a primitive subroutine, it finds one; but in the case of the standalone subroutine, it does not. To tell the calling program to look elsewhere, we use a REF declarative:

```
          IDT MAIN  
          REF <entry point1>, <entry point2>  
          DEF MAIN          DEFINE MAIN PROGRAM ENTRY POINT‡  
          —  
          —  
          —
```

‡ Since the subroutine has no need to know the address of MAIN, we have not declared it as a subroutine reference.


```

@BRL <entry point1>
-
-
-
@BRL <entry point2>
-
-
END MAIN

```

Extended format is used so that the loader will have a cell in the calling program in which to place the actual load address of the entry points.

8-6 PARAMETER PASSING AND COMMON STORAGE.

Now that the PC can find the subroutine and from there, find its way home after execution, there is the associated problem of how the main program and the subroutine pass parameters. The basic techniques illustrated previously for primitive subroutine data lists (Section 8-3) continue to be valid, and there are several additional methods.

8-6.1 COMMON STORAGE. Use of common storage may preclude the need to pass parameters at all. Common storage is an area set aside at load time which may be accessed by separately assembled programs.

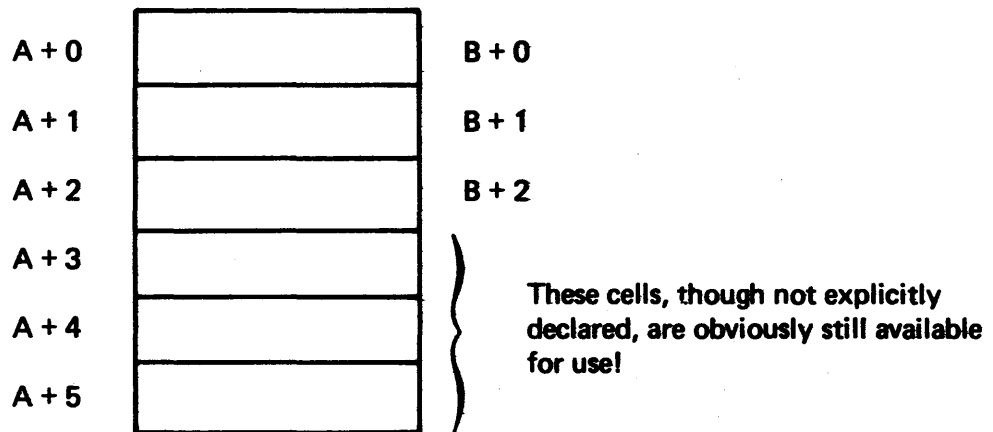
Each routine which accesses these locations should contain a common declarative:

```
<label> COMM <n = size of COMMON>
```

with an optional label. For example, in the following structure:

	<u>Main</u>				<u>Subroutine</u>	
	IDT	MAIN			IDT	SUB
A	COMM	5		B	COMM	3
	-				-	
	-				-	
	@BRL	SENTRY		SENTRY	-	
	-				-	
	-				-	
	-				-	
	END	MAIN		END	END	SENTRY

the loader will set aside five locations (maximum common size declared by any routine) in the common area and allow the main program to reference them with respect to base address A and allow the sub-program to reference them with respect to the base address B.



Both programs access the same memory locations although they may use different names to do so.

8-6.2 VIA REGISTERS (CALL BY VALUE). This technique is the same as that described in Section 8-3.2.

8-6.3 IN-LINE PARAMETER LISTS (CALL BY ADDRESS). This technique is the same as that described in Section 8-3.3.

8-6.4 CALL BY NAME. In the call-by-name technique the name of the variable must be passed across the program interface, and the system must have a way to locate the value corresponding to that label. The routine using the name may address it indirectly through a local name, provided that

- It is declared as an external value with a REF pseudo-op
- Space is reserved for a copy of the address to be supplied by the link editor with a DATA declaration that links the local label with the external label.

For example, expanding on the skeletons presented in Section 8-5 to pass a parameter, PARAM1 from the main program to the subroutine:

```

P1          IDT          SUB
P2          DEF          SUB1
SUB1       REF          PARAM1
           DATA        PARAM1
           DATA        PARAM2
           <first entry point>
           -
           -
           LDA          P1
           -
           -
SUB2       <second entry point>
           -
           -

```

```

                                <branch to link address>
                                END
                                -
                                -
                                -
                                IDT          MAIN
                                REF          SUB1
                                DEF          PARAM1
PARAM1          DATA          <value of parameter>
                                -
                                -
                                -
                                @ BRL      SUB1
                                -
                                -
                                -
                                -
                                -
                                END

```

and further expanding the example to pass PARAM2 back from the subroutine to the main program:

```

                                IDT          SUB
                                DEF          SUB1
                                REF          PARAM1, PARAM2
P1          DATA          PARAM1
P2          DATA          PARAM2
SUB1          <first entry point>
                                -
                                -
                                -
                                LDA          *P1
                                -
                                -
                                -
                                STA          *P2
                                <branch to link address>
                                END

```

where the main program must now DEFINE PARAM2 as well as PARAM1. Alternatively, data may be passed using the external name rather than the local name in conjunction with:

1. An extended format instruction using the external name, with the external name declared in a REF pseudo-op; and
2. A corresponding DEF pseudo-op in the program to which the name is local.

For example:

	<u>Main</u>			<u>Subroutine</u>
	IDT	MAIN		IDT SUB
	REF	SUB1		DEF SUB1
	DEF	PARAM1, PARAM2		REF PARAM1, PARAM2
PARAM1	DATA		SUB1	<first entry point>
PARAM2	DATA			-
MAIN	-			-
	-			-
	@BRL	SUB1		@LDA PARAM1
	-			-
	-			-
	-			@STA PARAM2
	END			<branch to link address>
				END

8-7 RECURSIVE CALLS TO A PRIMITIVE SUBROUTINE.

A subroutine which in the process of executing issues a call to itself, is called a *recursive* subroutine. A few machines have a builtin stack feature in which the return pointer for a subroutine call is automatically saved when a new subroutine call is issued. The 980 does not have the hardware feature, but it can be implemented in software without great effort, using the pushdown stack technique discussed in Section 6-10. If we are using the formal subroutine structure (Section 8-4) and wish to make it recursive, we will probably be most successful if the pushdown stack is built in COMMON.

If we try to make recursive calls with no stack feature, as follows:

	IDT	SUB1
ENPT1	-	
	-	
	-	
CALL	BRL	ENPT1
	-	
	-	
	-	
	-	
	RMO	L,P RETURN

we can never return to the main program because the contents of the link register have been destroyed by the BRL ENPT1 subroutine call. Thus, whenever we execute the return, we will always come back to CALL+1, resulting in an endless loop.

Note that the return addresses (successive contents of the link register) are the items saved in the stack.

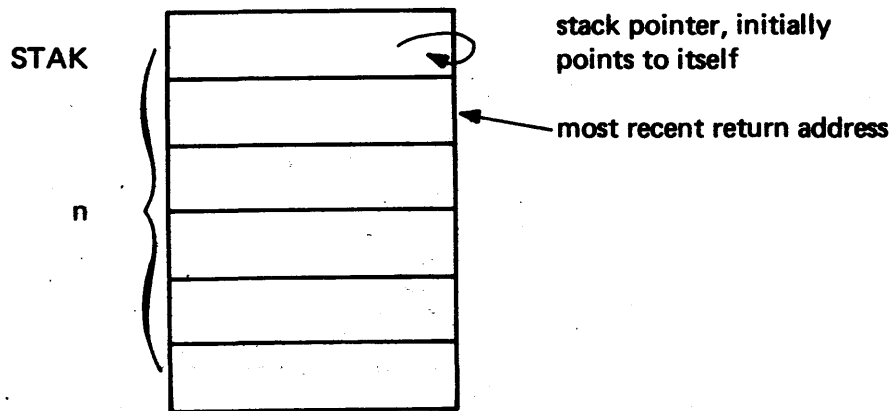
8-7.1 RECURSION USING THE X-REGISTER. This routine assumes the X-register is not needed except to hold the current stack pointer:

<u>Main</u>				<u>Subroutine</u>		
EQU	\$		STAK	BES	20	
-			ENPT1	EQU	\$	
-				RMO	L,A	SAVE RETURN ADDRESS
-				RDE	X,X	
LDX	=0	INITIALIZE SP		STA	STAK,X	IN STACK.
CALL1	BRL	ENPT1	CALLR	BRL	ENPT1	RECURSIVE CALL.
		RETURN TO HERE	RTRN	LDA	STAK,X	POP-UP STACK.
-				RIN	X,X	
-				RMO	A,P	RESTORE PC.
-				END		
END						

Each time the RTRN sequence is executed*, the appropriate return address is entered into the PC for the pending call.

8-7.2 RECURSION USING INDIRECT ADDRESSING. If the X-register must be free for other uses during the subroutine execution, a stack may be built using indirect addressing through the use of a stack pointer. Since we want to use the IMO and DMT instructions, it will be convenient to build the stack from smaller to larger memory addresses (i.e., opposite from the way we did it previously, to try to utilize the BIX instruction).

We will use the base address (STAK+0) to contain the stack pointer and begin the actual stack at STAK+1.



STAK	DATA	STAK	INITIALIZE STACK POINTER
	BSS	<n>	SAVE n STACK LOCATIONS
ENPT2	RMO	L,A	SAVE RETURN ADDRESS
	IMO	STAK	IN NEXT
	STA	*STAK	STACK LOCATION.
-			
-			

*Note: the instructions between ENPT1 and CALLR must make some provision for a branch into the instructions below the CALLR instruction, or none of these instructions will be executed.

```

-
-
BRL      ENPT2
-
-
-
-
RTRN    LDA    *STAK    GET RETURN ADDRESS FROM STAK
        DMT    STAK    POP UP STACK
        RMO    A,P     RESTORE PC
        <error>    TEST FUNCTION NOT USED

```

Note that the DMT instruction is used only for decrementing, since the stack pointer moves only between STAK+1 and STAK+<n>.

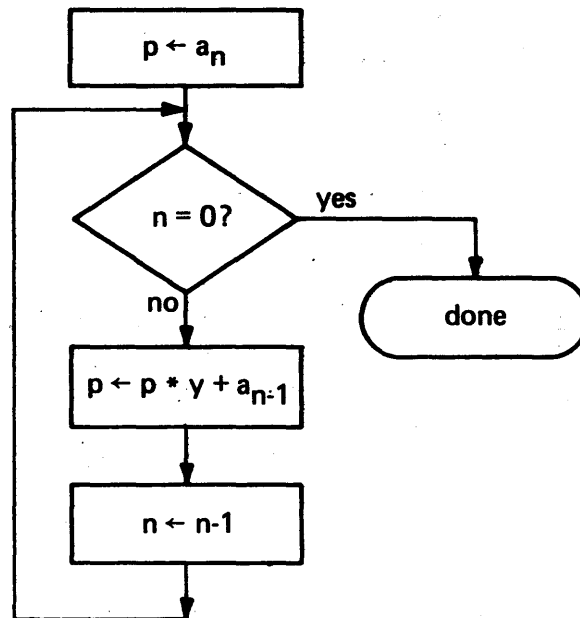
8-7.3 EXAMPLE OF A RECURSIVE SUBROUTINE. As a trivial example of how a recursive subroutine can be written, let's consider the problem of evaluating a polynomial:

$$a_n y^n + a_{n-1} y^{n-1} + a_{n-2} y^{n-2} + \dots + a_0$$

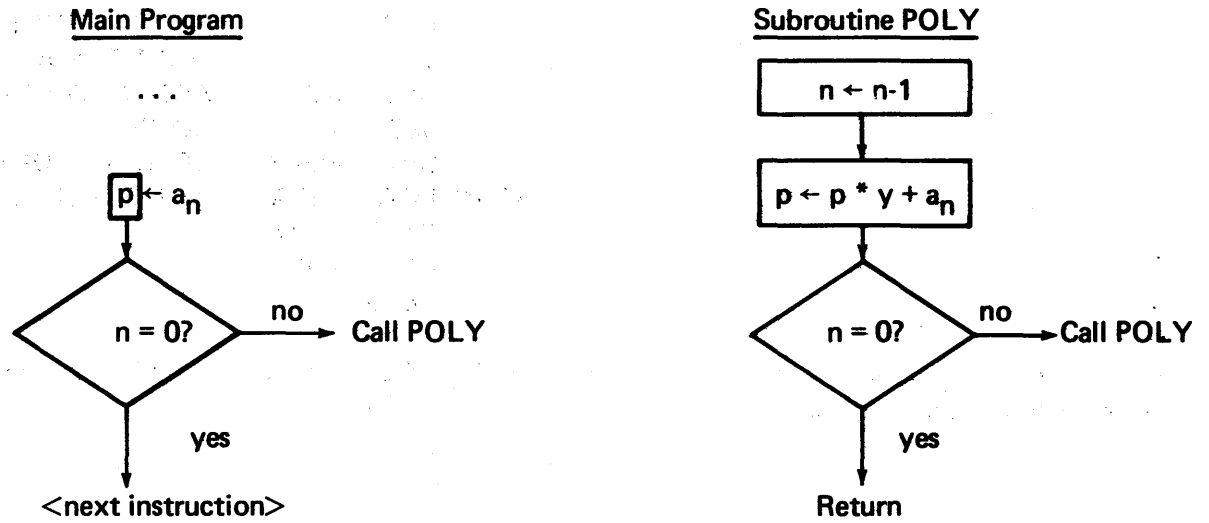
The most efficient algorithm evaluates the equivalent expression:

$$[(a_n * y + a_{n-1}) * y + a_{n-2}] * y + \dots$$

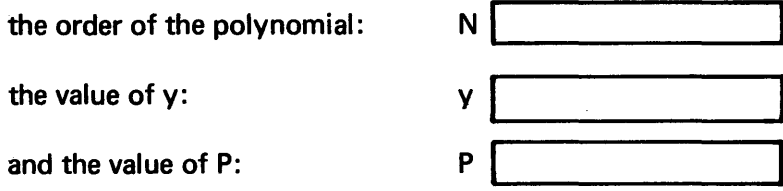
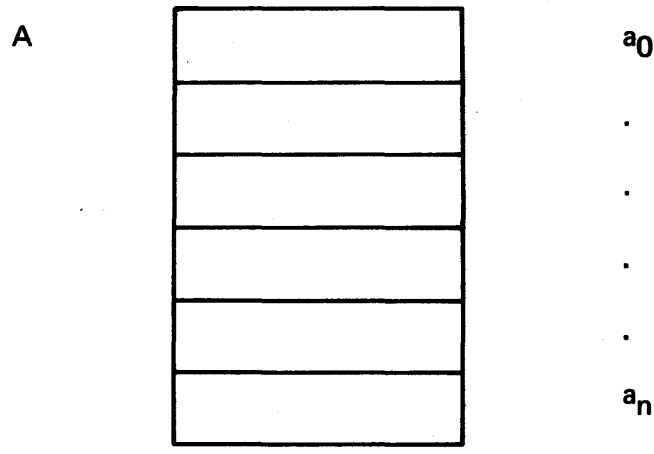
and the flowchart looks something like this:



Encountering this situation in real life, we would probably write it as the simple loop shown above, but we will write it as a recursive subroutine as an illustration of the technique:



We'll assume that all parameters are stored in COMMON – the coefficients:



We'll use the boilerplate of Section 8-7.2 to stack the subroutine calls, since the X-register is used by the subroutine.

<u>Main</u>			<u>POLY Subroutine</u>		
...					
LDA	N	GET ORDER			
LDA	A,X	GET HIGHEST ORDER COEF.	STAK	DATA STAK	THIS IS
STA	P	SAVE AS POLYNOMIAL		BSS 20	
SZE	X	DONE ?	RMO	L,A	THE RETURN

BRL POLY NO, CALL POLY
... YES, CONTINUE

	IMO	STAK	ADDRESS
	STA	*STAK	STACKER.
POLY	RDE	X,X	DECREMENT INDEX
	LDA	P	GET ACCUMULATED VALUE
	MUL	Y	MULTIPLY AND
	ADD	A,X	ADD NEXT COEF.
	SZE	X	DONE?
	BRL	POLY	NO, CALL POLY AGAIN
RTRN	LDA	*STAK	YES, EXECUTE RETURN
	DMT	STAK	
	RMO	A,P	
		<error>	

In this trivial example, each return up to, but not including, the last is made to RTRN; the very last return goes back to the main program.

SECTION 9

LOGICAL OPERATIONS, BIT MANIPULATION, MASKS, AND FLAGS

9-1 TRUTH TABLES.

Given two logical variables A and B, which may have the value 1 (*true*) and 0 (*false*), the truth tables for the AND, OR, XOR (*exclusive OR*) and NOT functions are as follows:

AND (·)

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

Both A and B must be true for the result to be true.

OR (+)

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

Either A or B (or both) must be true for the result to be true.

XOR (⊕)

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Either A or B (but not both) must be true for the result to be true.

NOT (—)

A	\bar{A}
0	1
1	0

If A is true, \bar{A} is false.

TABLE 9-1

<u>Instructions</u>	
AND	AND To accumulator
IOR	OR To accumulator
RAN	Register AND
ROR	Register OR
REO	Register XOR
RIV	Register INVERT (one's complement)
SABO	} Set accumulator bit to 1 or 0
SABZ	
SMBO	} Set memory bit to 1 or 0
SMBZ	
TABO	} Test accumulator bit for 1 or 0
TABZ	
TMBO	} Test memory bit for 1 or 0
TMBZ	
LTO	} Left and right tests for 1 or 0 in accumulator
RTO	
LTZ	
RTZ	

9-2 LOGICAL OPERATIONS.

It is useful to be able to apply logical operations to the bit patterns in memory words and registers. The 980 is capable of six such operations – two of which assume that one logical operand is in memory and one is in a register, and four which assume that both operands are in registers.

9-2.1 MEMORY-TO-REGISTER LOGICAL OPERATIONS. These instructions have the form:

<label> <op-code> m, <mod> (e.a.) \oplus (A) \rightarrow A

where the symbol \oplus represents a bitwise logical operation between the contents of the effective address and the contents of the accumulator. The logical result is stored in the accumulator. The register/memory operations are:

logical AND
 AND $m_1, \langle \text{mod} \rangle$ (e.a.) \wedge (A) \rightarrow A
 logical Inclusive OR
 IOR $m_1 \langle \text{mod} \rangle$ (e.a.) \vee (A) \rightarrow A

9-2.2 REGISTER-TO-REGISTER OPERATIONS. These instructions have the form:

<label> <op-code> s,d (s) \oplus (d) \rightarrow d

where the bitwise logical operation takes place between the source and destination registers, and the result is stored in the destination.

The operations are

RAN	logical AND	$(s) \wedge (d) \rightarrow d$
ROR	logical OR	$(s) \vee (d) \rightarrow d$
REO	logical inclusive OR	$(s) \times (d) \rightarrow d$
RIV	logical NOT, or complement*	$\sim (s) \rightarrow d$

Consider, for example, the instruction:

RAN A,E

which means *take the (bit-wise) logical produce of the number in the A-register (source) with the number in the E-register (destination) and place the result in the E-register (destination).*

The original contents of the A- and E-registers:

```

      ...1110100111  A
        +
A  ...1011010001  E
-----

```

new contents of E:

```

      ↓
      ...1010000001  E

```

RAN A,E

while the contents of A remain unchanged.

When the operand sizes do not match, as when using an immediate operand:

IOR => 1A

the "missing" bits are treated as if they are zeros:

```

      1100011011000100  A
        +
V  00000000000011010
-----
      ↓
      1100011011011110  A

```

IOR => 1A

*This is one's complement (logical negation) as opposed to RCO which produces the two's complement (arithmetic negation).

9-2.2.1 Complementing Operations. There are two complementing operations. One gives the logical NOT operation (one's complement):

RIV s,d (s) → d

and the other places the two's complement of the source into the destination, making it more useful for algebraic calculations:

RCO s,d 0-(s) → d

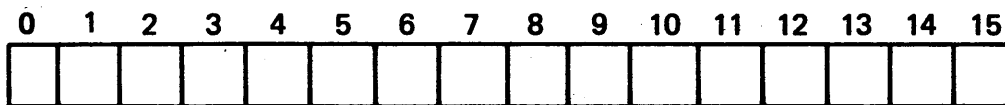
For example,

	0011100011011011	A
RIV A,A	↓	
	1100011100100100	A

9-3 BIT OPERATIONS.

Sometimes it may be useful to assign binary values to individual bits within the A-register or memory word. We say that we *set* a bit when we give it a value of 1 and *clear* it when we give it a value of 0

The bits, as usual, are numbered from left to right starting with 0



The bit settings instructions are

Accumulator

SABO n Set accumulator bit <n> (i.e., set to 1)
 SABZ n Clear accumulator bit <n> (i.e., set to 0)

Memory word, address m

SMBO n, m Set bit <n> in word <m>
 SMBZ n, m Clear bit <n> in word <m>

Having the ability to set or clear bits infers we should also have the ability to test the bits and make decisions based on the bit value.

The *bit skips* are of the same form as the set/clear operations:

Accumulator

TABZ n Test for zero in bit <n>
 TABO n Test for one in bit <n>

Memory location, address m

TMBZ n, m Test for zero in bit <n> of word <m>
 TMBO n, m Test for one in bit <n> of word <m>

If the condition tested for is true, the next instruction is skipped; if the condition is false, the next instruction is executed.



9-4 MASKING: ONE PRACTICAL USE OF LOGICAL OPERATIONS.

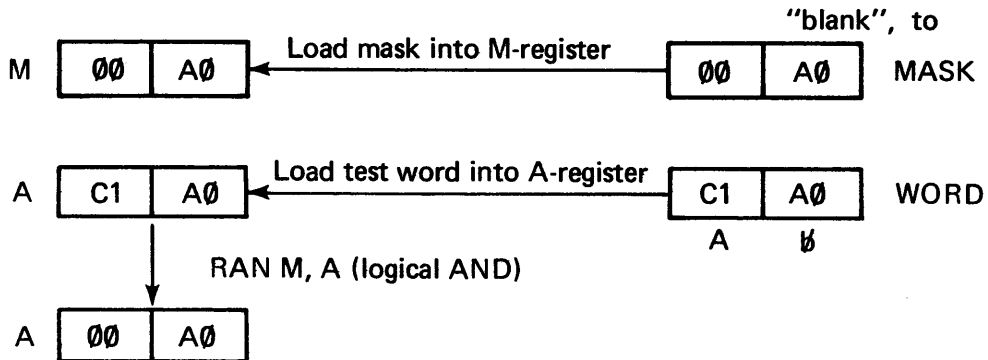
A *mask* is a collection of bits which can be used in conjunction with logical operations to test for the presence of certain bit patterns the programmer may wish to detect as part of a scheme to solve a problem. We can test for such conditions as:

- Even versus odd
- Positive versus negative
- All zeros
- All ones.

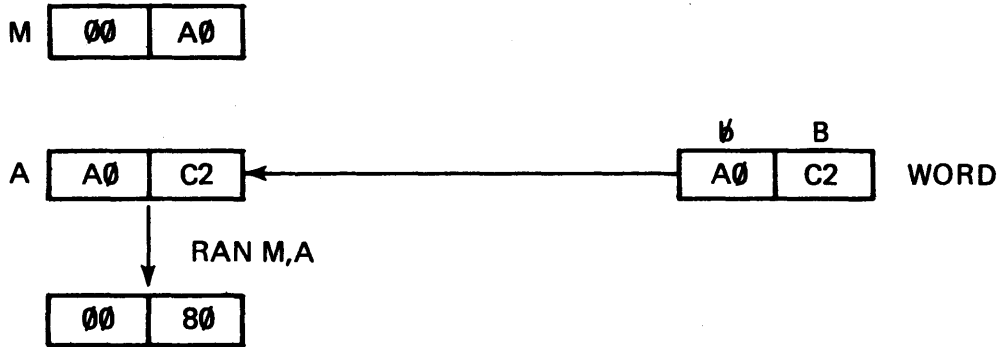
But we already have instructions in the instruction set that make these tests for us, so let's consider a problem not easily solved by the application of an existing instruction.

The problem we'll consider is that of searching for a blank (hex A0) in the lower half of a word. (In Section 9-5 below, we'll use this technique to search for a blank as a byte string delimiter.

We'll use a mask containing the bit pattern we wish to see, and zeros everywhere else. If we take the logical AND operation of this mask with some test word, we'll discover that every zero bit in the mask will cover up (*mask out*) the corresponding bits in the test word. In the following example we'll load the mask into M-register and the test word into A-register:



A blank was present in the appropriate position of the test word, so it "shows through" the mask. Consider the case where a blank either is not present or is positioned incorrectly:



There are two methods that could be used to find out whether or not a blank shone through the mask.

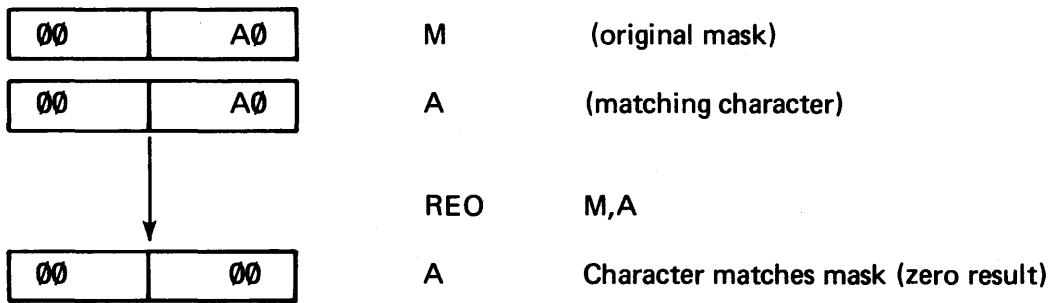
Method 1 Subtract the mask from the accumulator and test the result: If the result is zero, a matching bit pattern has been found. A nonzero result indicates the presence of a mismatch:

RSU	M,A	SUBTRACT MASK FROM A
SNZ	A	WAS THERE A MATCH?
BRU	ELSEWH	YES, GO ELSEWHERE
<next>		NO, PROCESS NO-MATCH COND

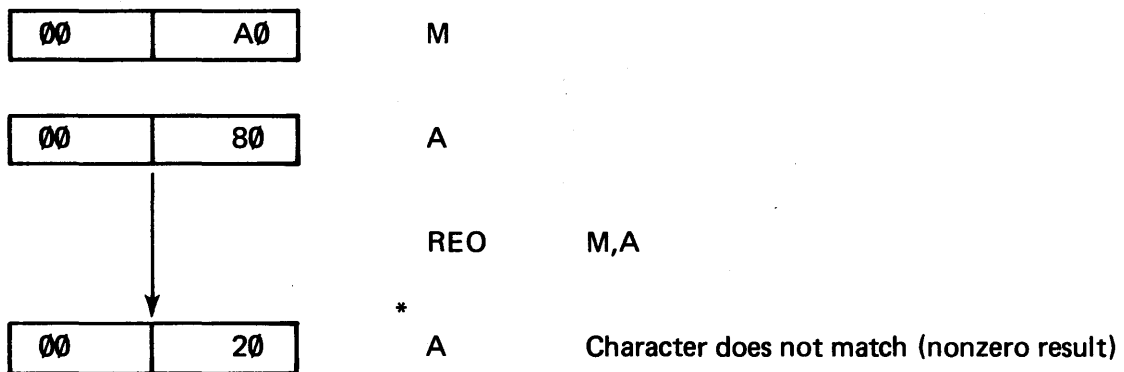
*How did we get 80 in the rightmost 8-bit positions? Watch!

A0 = 1010 0000
 C2 = 1100 0010 logical AND
 1000 0000 = 80

Method 2 Take the logical exclusive-OR (REO) of the accumulator with the original mask and look for a zero result. If we had obtained a matching character as a result of the first operation:



or, using a nonmatching character

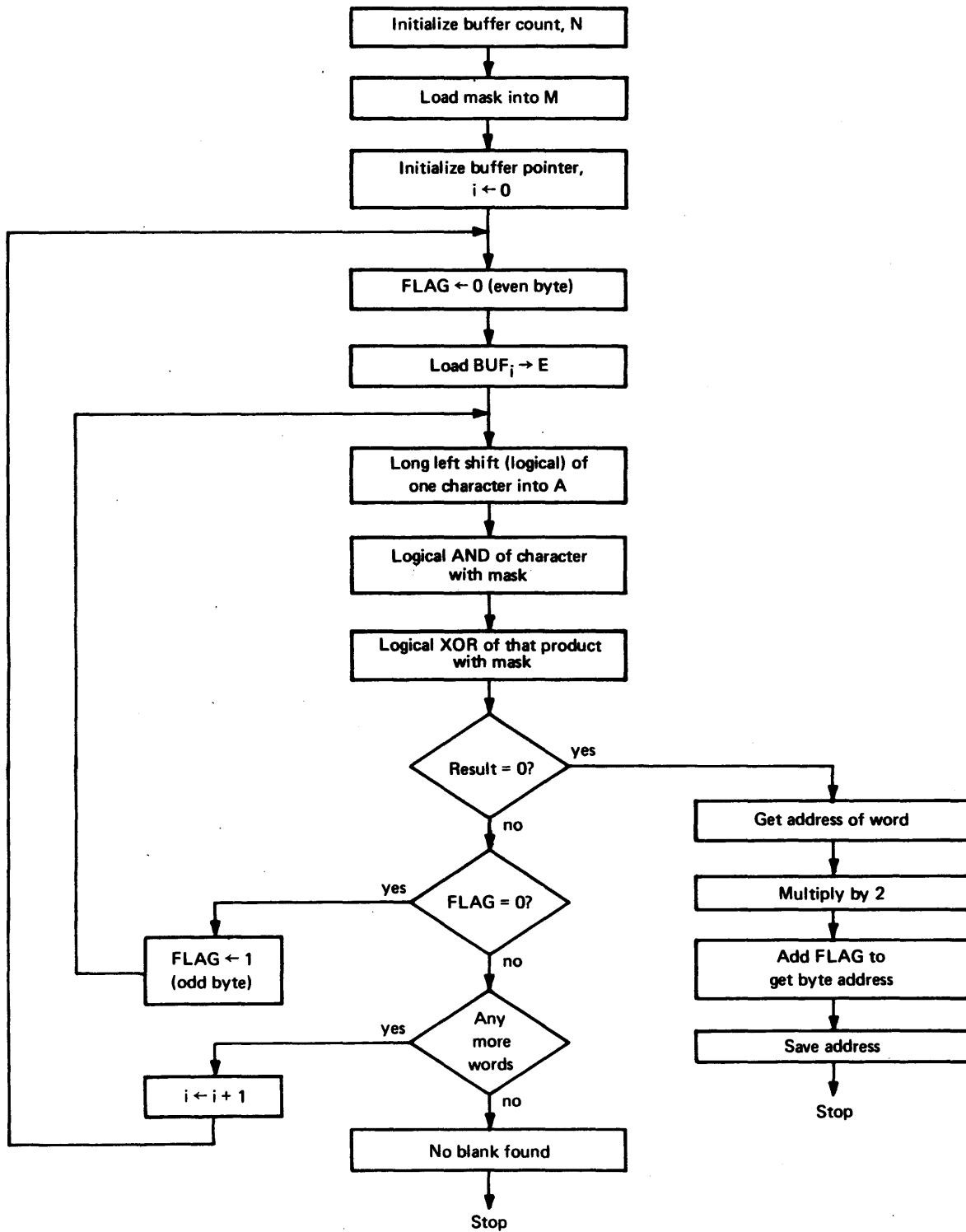


9-5 THE SEARCH FOR A BYTE STRING DELIMITER.

Let's use the masking technique discussed in the last section in the larger context of searching a byte string for a blank. (This application is useful in looking for a blank used as a delimiter in an input buffer.) We cannot conveniently use the CLC (*compare logical character*) to compare the input string against a string of blanks, because the CLC looks for the first mismatch.

*Get it now? A0 = 1010 0000 logical XOR
 80 = 1000 0000
 0010 0000

The following flowchart and code sequence shows how we could shift successive bytes into the low order 8 bits of a register, test them against the mask, and quit as soon as a match is found or the buffer is examined completely:



X	EQU	2	
A	EQU	0	
M	EQU	3	
BYTAD	DATA	0	
FLAG	DATA	0	
MASK	DATA	>A0	BLANK
N	DATA	<buffer size>	
	LDM	MASK	MASK TO M
	LDX	=0	INITIALIZE BUFFER INDEX
LOOP	SMBZ	15,FLAG	FLAG FOR HIGH ORDER CHARACTER
	LDE	BUFFER,X	GET BUFFER WORD
BACK	LLD	8	SHIFT CHARACTER TO A
	RAN	M,A	LOGICAL AND
	REO	M,A	LOGICAL XOR
	SNZ	A	MATCH
	BRU	FOUND	YES, BRANCH TO FOUND
	TMBZ	15,FLAG	NO, READY FOR NEXT WORD?
	BRU	LOWER	NO, PROCESS LOWER HALF OF THIS WORD
	RIN	X,X	YES. INCREMENT INDEX TO GET NEXT WORD
	DMT	N	END OF BUFFER?
	BRU	LOOP	NO, GO BACK AND GET NEXT WORD
EMPTY	<buffer empty routine>		YES. PROCESS NO MATCH IN BUFFER
	—		—
	—		—
	—		—
LOWER	SMBO	15,FLAG	SET FLAG FOR LOWER HALF
	BRU	BACK	AND GO BACK TO SHIFT AGAIN
FOUND	@LDA	=BUFFER	GET ADDRESS OF BUFFER
	RAD	X,A	AND ADD OFFSET TO GIVE WORD ADDRESS
	ALA	1	MULTIPLY BY 2
	LDM	FLAG	GET BYTE OFFSET FLAG INTO REGISTER
	RAD	M,A	ADD FLAG TO GET ODD OR EVEN
	STA	BYTAD	STORE A AS BYTE ADDRESS

9-6 TESTS FOR ONES AND ZEROS IN THE ACCUMULATOR.

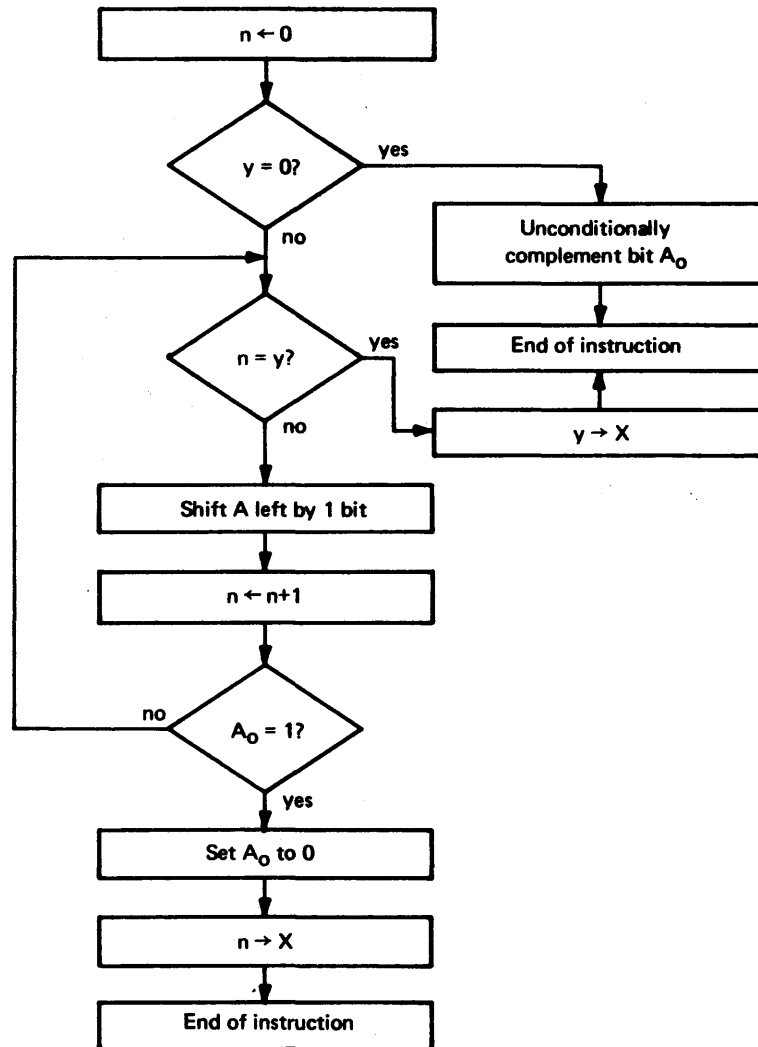
There are four instructions that work similarly to test for set or cleared bits in the accumulator; two instructions test for ones encountered in a left or right logical shift (LTO, RTO); and two instructions test for zeros (LTZ, RTZ). The instruction format is

<op code> y (where y is the number of bits to be tested)

The vacated bits are filled with zeros.

Let's consider the Left Test for Ones instruction (LTO) which works as follows:

Initialize the shift counter:



These instructions are most useful in programming real-time control systems, especially when identifying the source of interrupts (Section 11).

SECTION 10

INPUT/OUTPUT ON THE BARE MACHINE

10-1 INSTRUCTIONS.

The instructions discussed in this section are shown in Table 10-1.

10-2 TWO I/O TECHNIQUES.

Compared with the other operations carried out by a computer, I/O is slow because of the speed mismatch of an I/O device (which is basically mechanical) and the CPU/memory combination (which is totally electronic). This mismatch in speeds is handled two different ways: by polling or, if the computer in question has an interrupt capability, by interrupt (if the interrupt system has been enabled by the programmer). The differences are discussed below.

Table 10-1

<u>Instructions</u>		
RDS		Read Direct Single
WDS		Write Direct Single
ATI		Automatic Transfer Initiate

10-2.1 I/O BY POLLING. An order for the I/O is issued. Then the CPU (which contains a tight loop for the purpose) keeps asking the device, "Are you done yet?" As soon as the device reports (usually by setting a bit somewhere) that it is done, the CPU breaks out of the loop and continues with the program.

The actual instructions used in the polling loop depends on whether we are executing I/O on a low-speed device attached to the data bus or on a high-speed device connected to the direct memory access channel (DMAC) which handles 10^6 words per second (see Figure 10-1).

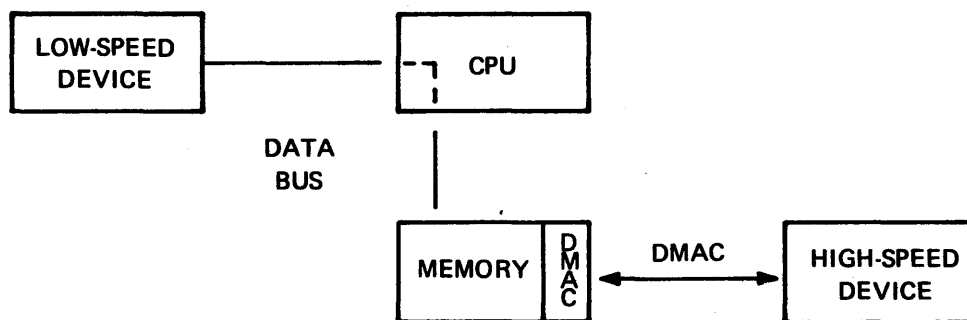


Figure 10-1. Input/Output by Polling

10-2.2 I/O BY INTERRUPT. The CPU issues an order for I/O and tells the device to transmit a signal (interrupt) when it is finished. Meanwhile, the CPU continues with whatever other job it is assigned. If that job depends on the outcome of the I/O, obviously the CPU can't go on but must sit in an idle loop waiting for the interrupt signal to come. In the latter case the CPU has nothing else it can do without waiting for the results of the I/O. There is really nothing to indicate that either I/O method should be chosen over the other. However, if other useful work can be done while the I/O is going on, the interrupt method produces a faster executing program.

A clever programming technique is sometimes available for use in conjunction with the interrupt method; that is the technique of *double buffering*.

10-2.3 DOUBLE BUFFERING. In this technique two buffer areas are defined (let's call them A and B). The computer fills buffer A and then issues a write instruction to transfer the contents of that buffer to the output device. Meanwhile, it continues operation and fills buffer B while buffer A is being transferred. If the interrupt signal comes before buffer B is full, it is not acted upon until buffer B is ready. If buffer B is ready first, the CPU must wait until the transfer of buffer A is complete before issuing a write instruction for the transfer of buffer B. While B is being transferred, the CPU works on filling A again. The process alternates in this manner until all desired output is transferred.

This example assumes that the contents of both buffers were sent to the same output device. If the contents of A were going to one device and the contents of B to another device, this represents not double buffering, but merely the case in which each device has its own internal buffer area.

10-3 INTERNAL AND EXTERNAL ADDRESSES.

The memory address pair at 0098 /0099 is devoted to interfacing a single high-speed DMAC device to the 980. If the DMAC expander option (extra hardware) is available, the address pairs 009A — 00A7 are used to address up to seven other DMAC devices.

The other path for interfacing I/O devices is the data bus, which in the standard 980 configuration has four ports. Each port may be used to connect one low-speed external device, or external expander hardware may be used to connect 15 devices through each port *(see Figure 10-2). Each of the 15 ports can be used to plug in another level of external expander; however, we are limited to an addressing capability for only 256 external registers).

Part of the expander logic is devoted to resolving which of the devices connected to the port is the one being addressed. All 980 standard documentation assumes that each of the various devices has a specific address. (These addresses can be changed by hardware strapping, if necessary, to suit the needs of a particular installation.)

Part of the interface between the device and the port (or expander) is designed to hold numbers that are being transmitted between the device and the CPU. Since such number-holders are traditionally called *registers* and since these are associated with an external device (rather than being part of the standard complement of CPU/memory registers), they are called *external registers*.

*Limited internal expansion capability is available for connecting up to nine more devices.

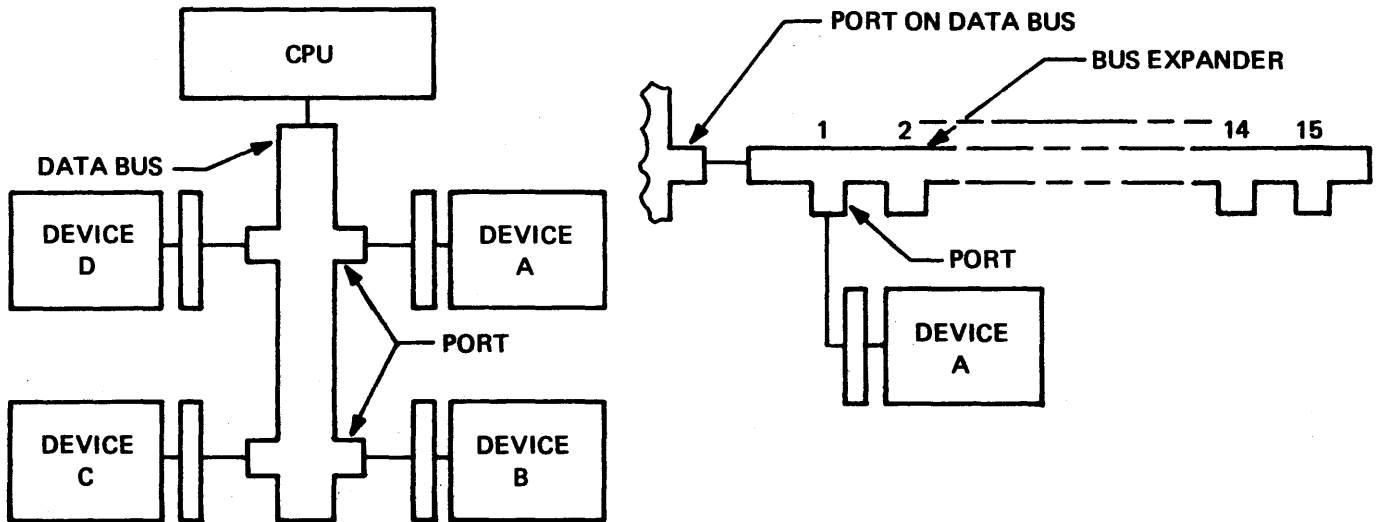


Figure 10-2. Four-Port Data Bus and 15 – Port Bus Expander Block Diagram

The number of external registers for each device varies according to the requirements of the device. The card reader, for example, uses only one external register, while the paper tape reader uses 2. The standard addresses for these registers are shown in Appendix D.

A standard four-port configuration (no expander logic) which we add a card reader and teleprinter, is shown in Figure 10-3.

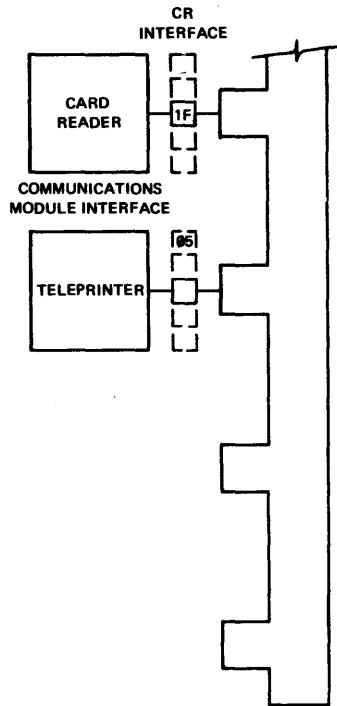


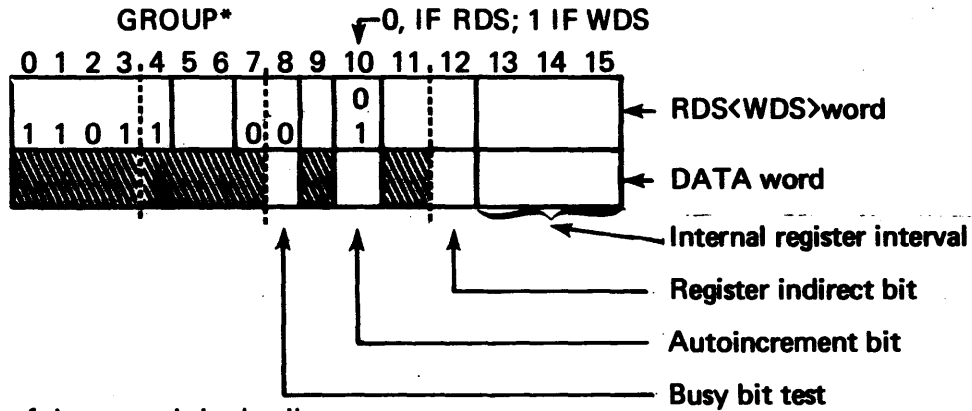
Figure 10-3. Data Bus/Device Interfaces, Showing External Registers

10-4 READ OR WRITE (LOW-SPEED DATA BUS DEVICE).

Reading or writing a single word from an external device connected to the data bus is done using the read-direct-single (RDS) or write-direct-single (WDS) instruction. The instruction must be followed immediately by a DATA word whose rightmost 3 bits specify the internal register to or from which the transfer is to take place. The external register address is specified by the <device address> in the operand field of the RDS/WDS instruction.

RDS/WDS <device address>
 DATA <operand>

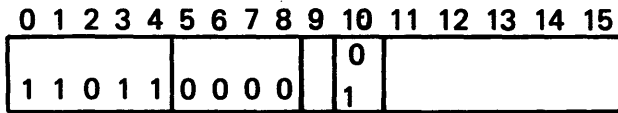
The WDS instruction moves a word from the internal register to the device and the RDS moves a word from the device to the internal register. Let's see what the assembler does to the word pair and what the various fields of the assembled words mean:



Let's examine each of these words in detail.

- Word 1. Bits 0 – 4 contain the op-code for the direct single I/O instruction.
- 7 – 8 contain the op-code for the direct single I/O instruction.
- *Bits 5, 6 contain the group where the device is to be found.

Standard device addresses assume that we have only group zero; so we shall assume zeros in these bits.



The remaining bits 9 and 11-15 contain the external register address.

Note that all our external addresses are such that bit 10 is unused, since the assembly process will set it to zero or one depending on the direction of the transfer. So:

RDS > IF assembles as D81F (bit 10 = 0)

whereas,

WDS > IF assembles as D83F (bit 10 = 1)

Word 2. Bits 0–7 } unused in the 980. To ensure compatibility with earlier models of the 980 series,
 9 } place a one in bit 2. This configuration will translate into hex as

00	–	980 only
20	–	downward compatible

Bit 8 permits the programmer to specify a *device busy* test. If bit 8 is programmed one, and no data transfer takes place, execution proceeds with the next instruction which is usually a BRU \$-2 (or *try again*). If bit 8 is programmed one, and there is a successful data transfer, the next instruction is skipped. If bit is programmed zero, the data lines are sampled for whatever information is on them (i.e., whether the device is ready or not) and the execution proceeds sequentially.

Bit 10* Autoincrement
 Bit 11 Unused in the 980; used as an auto-decrement bit in earlier models.
 Bit 12‡ Register indirect bit
 Bits 13–15 Internal register to/from which a word is to be moved. The designators are the same as those used previously:

A	=	000
E	=	001
X	=	010
M	=	011
S	=	100
L	=	101
B	=	110
PC	=	111

To summarize the I/O word pair to date, let us assume the following conditions:

1. All transfers take place to and from the accumulator (register 0) directly: no indirect, no autoincrement.
2. Make compatible with earlier 980 models.

With device-busy test; "polling"

RDS/WDS	<device addr>	I/O TO A
DATA	> 2080	DEVICE BUSY?
BRU	\$-2	YES, HANG TIL RDY.
<next instruction>		NO, CONTINUE

Without device busy test

RDS/WDS	<dev addr>	I/O TO A
DATA	> 2000	XFER WORD.
<next instruction>		CONTINUE

*If the indirect bit is set, the autoincrement bit is consulted at the end of each transfer. If this bit is also set, the address contained in the register is incremented; otherwise, it is decremented.

‡If the register indirect bit is zero, the register designated in bits 13–15 contains the number. If the register indirect bit is one, bits 13–15 designate a register which points to an internal memory word that will be used instead. Following this transfer the address in the register is incremented or decremented according to the setting of the autoincrement bit (see footnote above).

10.5 EXTERNAL DEVICES: DATA, STATUS, COMMAND WORDS.

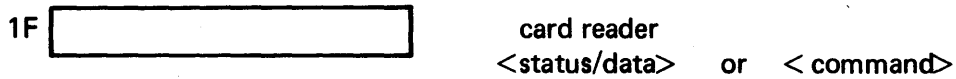
External devices have from one and five external registers as part of their interfaces to the data bus. In general, three functions are served by these registers; they may contain

- Data: the actual number to be transferred
- Status: the status of the device as reported to the CPU
- Command: control word sent from the CPU to the device.

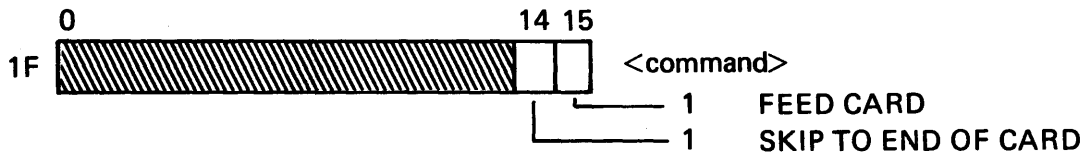
A register may serve more than one function at different times, and the significance of the registers is different for each different device. Hence, we shall discuss the devices individually.

10-6 CARD READER.

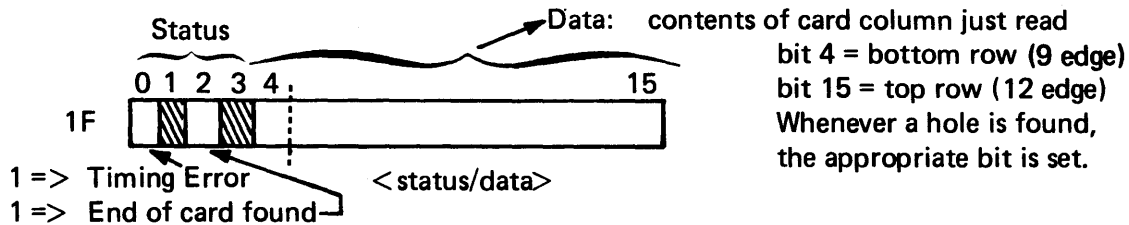
The card reader has only one register, external address 1F which suffices for all CPU/device communications.



It makes no sense to try to write on a card reader; so the device is designed so that any attempt to write on it is interpreted as a command word from the CPU. When written into (and, therefore, viewed as a command register) 1F has the following configuration:



When read from, this register function as a combined status and data register, reading one column at a time.



Some devices have a bit which can be set in the command word to connect or disconnect their program control and/or switch them off or on. No such capability exists for the card reader, for which these functions are handled by the operator through switches on the card reader control panel.

10-6.1 READING A CARD BY POLLING. The first thing to do is feed a card. We'll use the WDS instruction to send the appropriate command to the device

LDA	=1	SET FEED ENABLE BIT
WDS	>1F	WRITE TO CR COMMAND REGISTER
DATA	>2080	FROM ACCUMULATOR. BUSY?
BRU	\$-2	YES, TRY AGAIN
<next instruction>		NO, CONTINUE

Now that the card is fed, we can read the first column.

READCD	RDS > 1F	INITIATE READ
	DATA 2080	TO A. BUSY?
	BRU \$-2	YES, TRY AGAIN.
	<next instruction>	NO, CONTINUE

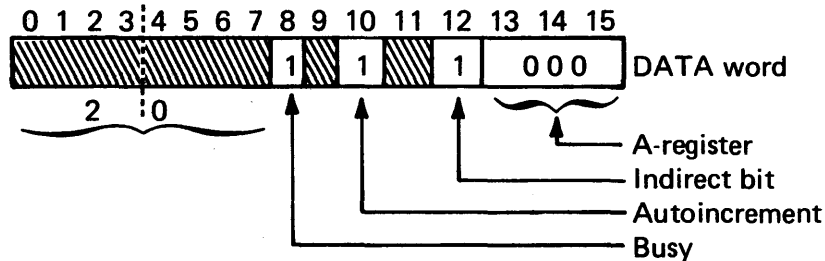
The A-register now contains the status and the data; we can separate the two via a long shift of the character bits into E. A test of the remaining status bits will tell us if there was a timing error or if this is the end of the card.

LRD 12	SHIFT CHARACTER TO E
SZE A	ANY ERROR BITS SET?
BRU ERROR	YES, GO TO ERROR ROUTINE
<next instruction>	NO, CONTINUE.

We can arrange to do this 60 times (for 60 columns of input on a card - or some other number of our choosing), moving each character into a separate word of an input buffer:

	LDA =1	FEED ENABLE
	WDS > 1F	WRITE COMMAND
	DATA > 2080	BUSY?
	BRU \$-2	YES, TRY AGAIN
	LDX =-60	NO, SET INDEX FOR 60 CHARACTERS
READ	RDS > 1F	INITIATE READ
	DATA > 2080	BUSY?
	BRU \$-2	YES, TRY AGAIN
	LRD 12	NO, SHIFT CHARACTER TO E
	SZE A	ANY ERRORS?
	BRU ERROR	YES, GO TO ERROR ROUTINE
	LLD 12	NO RIGHT ADJUST DATA
	STA INBUF+60,X	AND SAVE IN BUFFER
	BIX READ	GOT ALL 60? NO, GO BACK
	<next instruction>	YES, CONTINUE

10-6.2 POLLING THE READER WITH AUTOINCREMENT. We could use A as a pointer to the reception area and save the labor of moving each character individually from A to IBUF by setting bits 10 and 12 of the DATA word to one.



The hex value in this case is

DATA > 20A8

and the program is

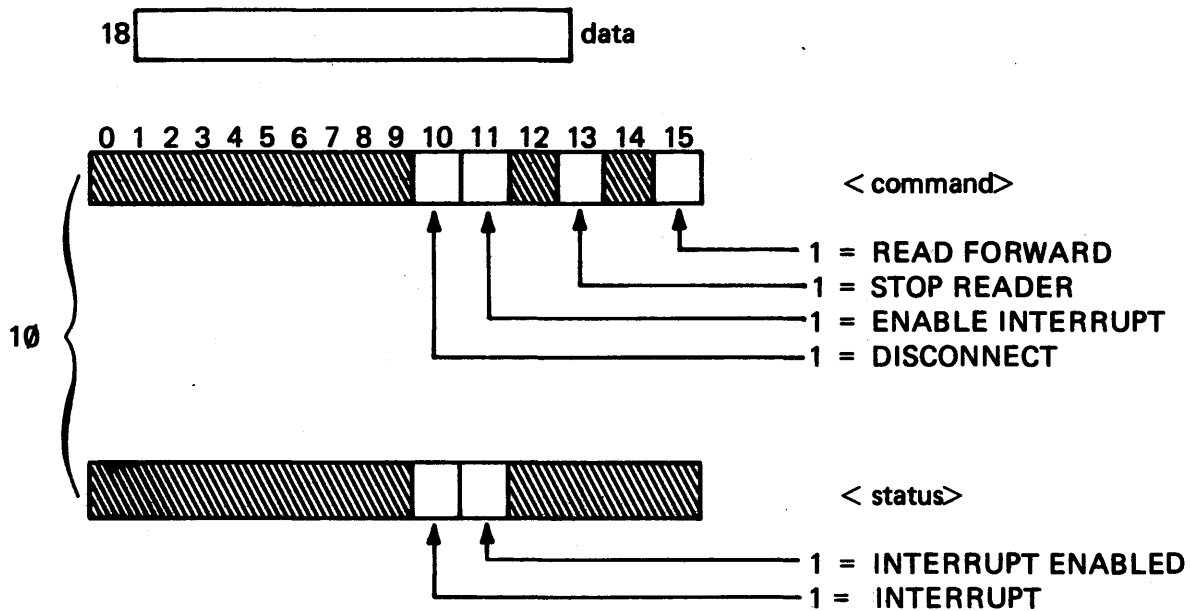
```

<enable feed>
LDX   =-60      INITIALIZE INDEX
LDA   =INBUF    GET ADDRESS OF FIRST BUFFER WORD
READ  RDS   > 1F  READ TO ADDRESS POINTED TO BY
      DATA > 20A8 A AND INCREMENT. BUSY?
      BRU   $-2  YES, TRY AGAIN
      -
      -
      -
      BIX   READ  NO, GOT ALL 60? IF NOT, GO BACK.
<next instruction>

```

10-7 HIGH-SPEED PAPER TAPE READER.

This device has two external registers: data and command/status:



```

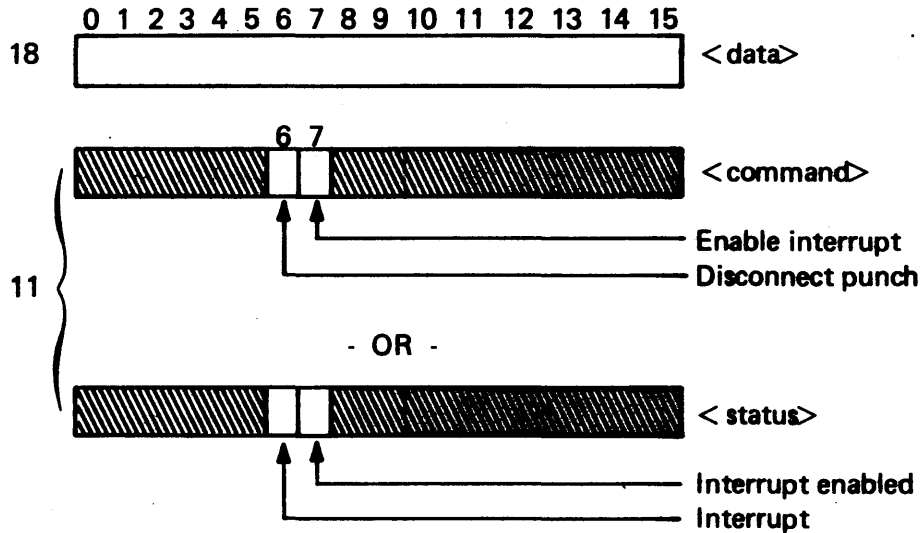
LDA   =1      ENABLE HSR
WDS   > 10    SEND TO READER
DATA  2080    WITH BUSY BIT TEST
BRU   $-2     REPEAT UNTIL READER ON
LDX   =-10    INDEX 10 CHARACTERS
RDS   > 18    READ CHAR, RIGHT ADS
DATA  > 2080  WITH BUSY BIT TEST
BRU   $-2     REPEAT IF NOT READY
STA   IN+10,X SAVE CHARACTER

```

BIX	\$-4	LOOP BACK
LDA	=4	BIT 13 STOPS READER
WDS	> 10	OUT TO COMMAND REGISTER
DATA	2000	NO BUSY TEST

10-8 HIGH-SPEED PAPER TAPE PUNCH.

This device has two external registers: data and command/status:



Except for the disconnect bit, these bits all pertain to use of the interrupt technique, discussed in Section 10-2.2.

This program punches a 12-inch blank leader (10 character 5 per inch): And 10 characters:

LDX	=-120	SET UP FOR 12 INCH
LDA	=0	BLANK FRAMES LEADER
WDS	>18	OUT TO HSP
DATA	>2080	WITH BUSY BIT TEST
BRU	\$-2	HANG, IF BUSY
BIX	\$-3	LEADER FINISHED? NO, GO BACK
LDX	=-10	SET UP FOR 10 CHAR.
LDA	CUT+10,X	GET (RIGHT ADJUSTED) CHAR.
WDS	>18	OUT TO HSP
DATA	>2080	WITH BUSY BIT TEST
BRU	\$-2	HANG, IF BUSY
BIX	\$-4	MORE CHARACTERS, GO BACK
<next>		OTHERWISE, CONTINUE

Note: A blank trailer, similar to the leader, should be punched at the trailing end of the tape.

10-9 733 ASR/KSR DATA TERMINAL.

The Texas Instrument a Model 733 ASR/KSR has one external register number >05 for all status/command data transfer; and communicates on a character - by - character basis via the Full Duplex EIA Communications Module.

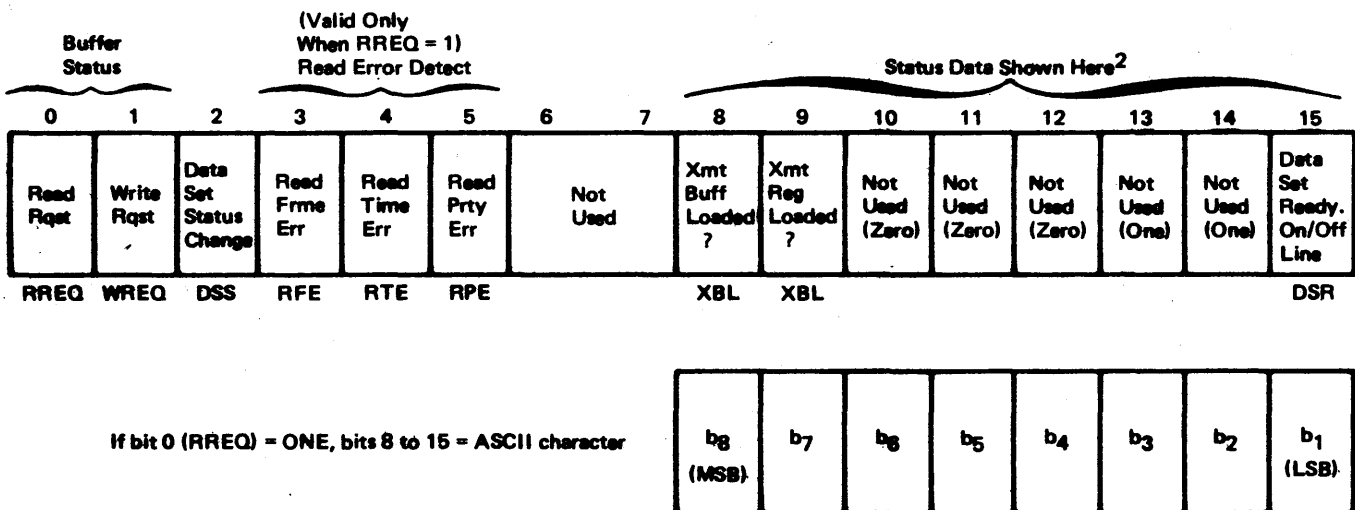


Figure 10-4. RDS Data Word

10-9.1 RDS DATA WORD. The RDS instruction is used to read one word of status information or one word of both status/data. The 16-bit word input by an RDS instruction is described below.

- | | |
|------------|-----------------------|
| <u>Bit</u> | <u>Interpretation</u> |
|------------|-----------------------|
- 0 Read Request (RREQ) zero: Bits 8-15 are status. ONE: A character has been transferred from the terminal to the interface (eq. the operator has hit a key) and is available to the computer in bits 8-15 of the data word. This bit, when set by the interface, causes an interrupt if interrupts are enabled.
 - 1 Write Request (WREQ). zero: The interface is not requesting a new character to be sent (eq., the interface has not finished sending the previous character to the data terminal). ONE: The transmit buffer is empty and ready to receive another character.
 - 2 New Data Set Status (DSS). zero: No change in the data-set-ready (DSR) signal. ONE: A change of state has occurred on the DSR signal, (eq., the keyboard ON-LINE/LOCAL switch has been repositioned).
 - 3 Read Framing Error (RFE). zero: No error. ONE: Improper character framing (eq., wrong baud rate or erroneous stop bit). To clear RFE, issue a CRR*, followed by the receipt of a "proper" character.
 - 4 Read Timing Error (RTE). zero: No error. ONE: Indicates that the communications module (CM) interface received another character from the data terminal before the CPU read the last one. Thus, one or more characters have been lost. This bit is cleared by issuance of a CRR* followed by receipt of a "proper" character.

* See WDS Data Word Section 10-10.2 below.

Bit	Interpretation
5	Read Parity Error (RPE). zero: No error. ONE: Improper character parity generation; to clear bit 5, the programmer should issue a CRR* followed by the receipt of a "proper" character.
6-7	Not Used

If RREQ is a ONE, bits 8-15 are character data; if ZERO bits 8-15 contain the following information:

8	Transmit Buffer Loaded (XBL). ZERO: The character in the transmit buffer has been sent to the transmit shift register to be sent to the 733 ASR/KSR. ONE: Buffer is loaded with characters and data terminal unable to accept another
9	Transmit Shift Register Loaded (XRL). ZERO: Shift register empty. ONE: A character is being serially transmitted to the data terminal.
10-14	Not Used.
15	Data Set Ready (DSR). ZERO: Data Terminal OFF-LINE. ONE: Data Terminal ON-LINE.

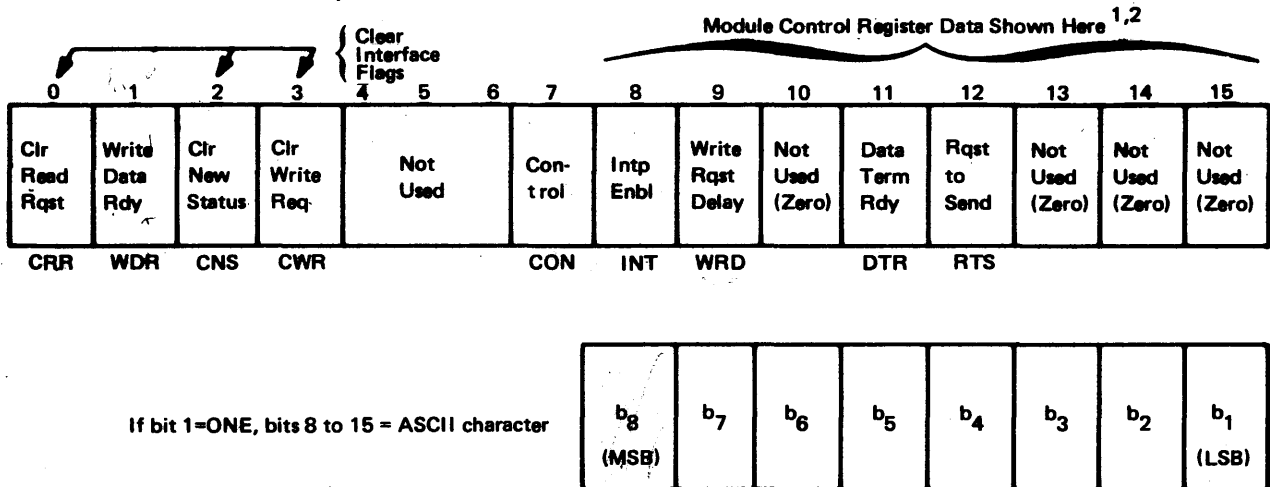


Figure 10-5. WDS Data Word

10-9.2 WDS DATA WORD. The 16-bit WDS data word is used to issue commands, subcommands (see paragraph 10-10.3), and data. A description of the WDS data word bit functions follows.

Bit	Interpretation
0	Clear Read Request (CRR). ZERO: No action. ONE: Clear read request and clear interrupt (if any). The Read Request status bit is set by the EIA interface when it has a character ready to send to the CPU (eq., the operator has pressed a key). This status line should be cleared directly before the program reads the character.
1	Write Data Ready (WDR). ZERO: Notation. ONE: Loads the USASCII character in bits 8-15 of the WDS data word into the interface transmit buffer and initiates transfer.
2	Clear New Status (CNS). ZERO: No action. ONE: Clears DSS flag.

<u>Bit</u>	<u>Interpretation</u>
3	Clear Write Request (CWR). ZERO: No action. ONE: Clears WREQ and associated interrupt (if enabled).
4-6	Not Used.
7	Control (CON). ZERO: No action. ONE: Bits 8, 9, 11, and 12 contain control information.

If bit 1 (WDR) of the WDS data word is set, bits 8-15 contain a character. If bit 7 (CON) is set, the following controls are output to the data terminal.

8	Interrupt Enable (INT). ZERO: Inhibit interrupts. ONE: Enable interrupt on RREQ, WREQ, or DSS.
9	Write Request Delay (WRD). ZERO: No transmission delay. ONE: Causes a 33 msec. delay between character transmission to the printer position of the 733 ASR/KSR data terminal. This is necessary for writing to the printer (300 baud) but not necessary for writing to the cassettes (1200 baud).
10	Not Used.
11	Data Terminal Ready (DTR). ZERO: Maintains the DTR circuit in the OFF condition. ONE: Maintains the DTR circuit in the ON condition which in effect, enables the transmit and receive capabilities of the data terminal.
12	Request to Send (RTS) ZERO: Maintains RTS circuit in OFF condition. ONE: Maintains RTS circuit in ON condition which enables the data terminal to accept characters from the CM interface.
13-15	Not Used.

10-9.3 733 ASR SUBCOMMANDS. The 733 ASR subcommands, or Remote Device Control (RDC) functions, each consist of a USASCII character placed in the least significant half of the data word associated with a WDS instruction. These control instructions are used by the programmer to perform such mechanical tasks as rewind and load cassettes.

The RDC functions are basically divided into two categories: (1) those activated by receipt of a single USASCII control character and (2) those activated by receipt of the nonprintable DLE character, followed by a second predetermined USASCII character code. The first category is known as the Automatic Device Control (ADC), or single-character subset; the second category is known as the DLE, or two-character subset. The two categories of RDC functions and their activation codes are listed in Table 10-1. It should be noted that the RDC logic circuitry automatically disables the teleprinter portion of the 733 ASR from printing the first character received after the DLE character code. This ensures that all two-character functions are, in effect, treated in the same manner as any nonprintable USASCII control character such as those in the ADC (single-character) category.

The command status character, shown in Figure 10-6, is transmitted by the 733 ASR Data Terminal to the computer in response to the RDC request status command (command ">3C" of Table 10-1). The status character bits, numbered from 7 (MSB) to 1 (LSB), correspond to bits 9 through 15 at the computer interface.

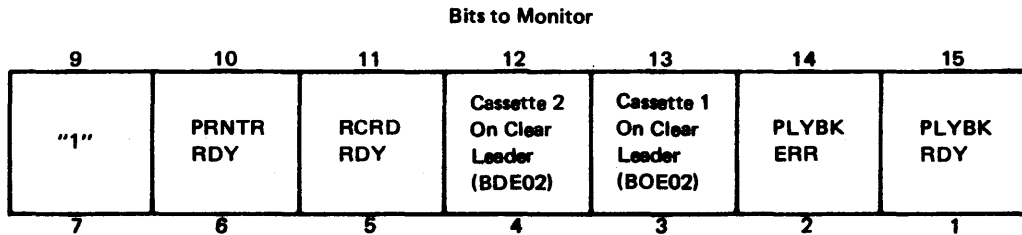


Figure 10-6. Status Character Bits

Table 10-2. Remote Device Control Functions

Single-Character Functions (ADC)		
<u>Function</u>	<u>Hex Code</u>	<u>USASCII Character</u>
Playback On/Keyboard Off (No busy bit necessary)	> 11	DC1
Record On/Printer Off	> 12	DC2
Playback Off/Keyboard On	> 13	DC3
Record Off/Printer On	> 14	DC4
Two-Character Functions ("DLE")		
<u>Function</u>	<u>Hex Code</u>	<u>USASCII Character</u>
Rewind Cassette 1	> 31	1
Rewind Cassette 2	> 32	2
Load Cassette 1	> 33	3
Load Cassette 2	> 34	4
Cassette 1 in Record, 2 In Playback	> 35	5
Cassette 2 in Record, 1 In Playback	> 36	6
Block Forward (1Block = 86 characters)	> 37	7
Block Reverse	> 38	8
Printer On (Non-Operable)	> 39	9
Printer Off (Non-Operable)	> 30	0
Automatic Device Control (Allows ADC Commands) On	> 3A	:
Automatic Device Control (ADC Above) Off	> 3B	;
Request Status Information	> 3C	<

If bit 0 of the RDS data word (RREQ) is a logic one, bits 8 through 15 contain a USASCII character with bit 8 the MSB and bit 15 the LSB. If bit 0 of the RDS data word is a logic zero, the following status data is returned to the computer:

- Transmit Buffer Register Loaded (XBL) - A logic one in bit 8 indicates that the communications module transmit buffer register is presently loaded with a USASCII character and unable to accept another. A logic zero indicates that the USASCII character has been transferred to the transmit shift register and that the transmit buffer register is now ready to accept another USASCII character.
- Transmit Shift Register Load (XRL) - A logic one in bit 9 indicates that the communications module character is in process of being serially transmitted to the 733 ASR/KSR Data Terminal. A logic zero indicates that the module transmit shift register is empty and that no serial transmission is presently in progress.
- Not Used - Bits 10 through 14 of the RDS status data word are not used by the 7333 ASR/KSR Data Terminal. Bits 10, 11 and 12 are wired to logic zero's and bits 13 and 14 are wired to logic one's.
- Data Set Read (DSR) - A logic one in bit 15 indicates that the 733 ASR/KSR Data Terminal is ON-LINE. This circuit is driven by the Data Terminal Ready output line of the 733 ASR/KSR Data Terminal and is maintained in the ON condition as long as the 733 ASR/KSR keyboard ON-LINE switch is engaged and the ASR-to-computer connection is maintained.

10-9.4 PROGRAMMING EXAMPLES. The programming examples in the following paragraphs include writing to both the teleprinter and cassette portions of the 733 ASR, reading from both the keyboard and cassettes, and issuing subcommands. All of the examples address I/O data bus register 05₁₆ in conjunction with the 733 ASR/KSR Data Terminal.

The busy-bit test, associated with both WDS and RDS instructions, should not be used when writing a data word consisting entirely of control data or reading a data word consisting entirely of status data. In addition, the busy-bit test should not be used with the first WDS instruction in a program or when writing the DCI subcommand (playback on) to the terminal. These restrictions make it advantageous to make the first WDS instruction one that writes all control data or issues the DCI subcommand. If this is not possible, the busy-bit test can be replaced with a check on the Write Request (WREQ) flag to determine when the WDS data word character bits have been accepted by the data terminal. The following example illustrates this alternate method:

LDA => 12	SEND DC2 COMMAND (RECORD-ON CHARACTER
@IOR => 5000	LOAD DATA, CLR WRITE REQ
WDS 5	WRITE TO ASR FORM REG A
DATA 0	NO BUSY BIT (1ST WDS)
RDS 5	READ STATUS (BIT 0=0)
DATA 0	REGISTER A
TABO 1	TEST BIT 1 (WREQ=1?)
BRU \$-3	LOOP IF WREQ = 0

·
·
·

In general, the guidelines listed below should be followed in programming the 733 ASR/KSR Data Terminal.

- If the ASR portion of the data terminal is the subject of the program, issue the appropriate subcommand(s) to ready the desired cassette for the I/O data transfer.
- If the busy-bit test is required on the first WDS instruction of the program, use the alternate method shown in the example above:
- Before each WDS instruction to write an USASCII character, clear the Write Request (WREQ) flag. If the teleprinter is being written to, the Write Request Delay (WRD) bit should be set prior to any character transmission to initialize the interface for 300 baud.
- Before each RDS instruction to read an USASCII character, clear the Read Request (RREQ) flag.

10-9.4.1 Write/Read KSR Example. *The following program writes 10 characters (each character is right-justified in a memory word) to the printer and then reads 10 characters from the keyboard. The data word of 0158₁₆ associated with the first WDS instruction disables the I/O data bus interrupts from the data terminal, sets the transmission rate to 300 baud, and sets the Data Terminal Ready and Request to Send control bits. Refer to the WDS and RDS data word formats for aid in following the example.

	@LDA	=> 158	CONTROL DATA, 300 BAUD TERMINL RDY
	WDS	5	OUT TO INTERFACE
	DATA	0	NO BUSY BIT (NO CHARACTER)
	LDX	=-10	
WRITE	LDA	OUT+10,X	GET CHAR, RIGHT ADJUST
	AND	=>7F	MASK OUT MSB
	@IOR	=>5000	LOAD ASCII DATA, CLR WREQ
	WDS	5	WRITE CHARACTER & CONTROL DATA
	DATA	> 2080	BUSY BIT
	BRU	\$-2	
	BIX	WRITE	LOOP BACK
	LDX	=-10	
READ	@LDA	=>8000	CLR READ REQUEST
	WDS	5	OUT TO INTERFACE
	DATA	0	NO BUSY BIT
	RDS	5	READ CHARACTER
	DATA	> 2080	BUSY BIT
	BRU	\$-2	
	AND	=> 7F	SAVE 7-BIT LSB CHARACTER
	IOR	=> 80	OR IN 8TH USASCII BIT (ONE), MARK PARITY
	STA	IN+10,X	
	BIX	READ	
OUT	DATA	' C H A R A C T E R S '	
IN	BSS	10	

*All examples are without interrupts

10-9.4.2 ASR Subcommand Example. The following program puts cassette 1 in the record mode (and cassette 2 in the playback mode), loads cassette 1, and initiates the recording process. Note that the DLE character code (0010₁₆), accompanies each of the two-character functions. The program assumes that a WDS instruction has already been executed and that cassette 1 is positioned on the clear leader at the beginning of the tape.

```

:
:
LDA      =>10          DLE USASCII CODE
@IOR     =>5000        LOAD USASCII DATA, CLR WREQ
WDS      5            ASR
DATA     > 2080       BUSY BIT
BRU      $-2          BRANCH IF NO DATA XFER
LDA      =>35          CASSETTE 1 TO RECORD MODE CODE
@IOR     =>5000        LOAD USASCII DATA, CLR WREQ
WDS      5            ASR
DATA     > 2080       BUSY BIT
BRU      $-2          BRANCH IF NO XFER
LDA      =>10          DLE USASCII CODE
@IOR     =>5000        LOAD USASCII DATA, CLEAR WREQ
WDS      5            ASR
DATA     > 2080       BUSY BIT
BRU      $-2          BRANCH IF NO XFER
LDA      =>33          LOAD CASSETTE 1 CODE
@IOR     =>5000        LOAD USASCII DATA' CLRAR WREQ
WDS      5            ASR
DATA     > 2080       BUSY BIT
BRU      $-2          BRANCH IF NO XFER
LDA      =>12          DC2, RECORD-ON CODE
@IOR     =>5000        LOAD USASCII DATA, CLRAR WREQ
WDS      5            ASR
DATA     > 2080       BUSY BIT
BRU      $-2          BRANCH IF NO XFER
:
:

```

10-9.4.3 Write ASR Example. The following program initiates the cassette recording process, writes 10 USASCII characters to the cassette, and terminates the cassette recording process. Note the alternate method of the busy bit test associated with the first WDS instruction. This program does not use interrupts.

```

LDA      =>12          SEND DC2 (RECORD-ON)
@IOR     =>5000        LOAD USASCII DATA, CLR WREQ
WDS      5            TO ASR
DATA     0            FROM REG A, NO BUSY BIT
Replaces } RDS      5
Busy Bit  } DATA     0
          } TABO     1
          } BRU      $-3

```

```

WRITE   LDX           =-10
        LDA           OUT+10,X
        AND           =>7F           MASK BITS 0-TO-8 TO ZEROES
        @IOR          => 5000       LOAD USASCII DATA, CLR WREQ
        WDS           5
        DATA        > 2080
        BRU           $-2
        BIX           WRITE
        LDA           =>14           SEND DC4 (RECORD-OFF)
        @IOR          =>5000       LOAD USASCII DATA, CLR WREQ
        WDS           5
        DATA        > 2080
        BRU           $-2
        LDA           =>7F           WRITE OVER RECORD-OFF CHARACTER (DELETE)
        WDS           5
        DATA        > 2080
        BRU           $-2
OUT     DATA        ' C H A R A C T E R S '

```

10-9.4.4. Read ASR Example. The following program reads 10 source records from a cassette, assuming each source record is less than or equal to 200 bytes. Note that the count of source record is maintained by keeping track of the number of DC3 characters (playback off) encountered. The DC3 character is used in programs that run under a monitor (supervisor), where each source line is terminated by a carriage return, line feed, DC3, and RUB OUT (delete). Standalone programs can use whatever conventions are necessary to end source lines. This program does not use interrupts; a wait loop is entered prior to reading each character from the cassette.

```

Designate Amt of Records to Read
START   {LDA           =10           AMT RECORDS TO READ
        {STA           COUNT        STORE AMT OF RECORDS
        {LDX           =0           CHARACTER STORAGE POINTER
        LDA           =>11          PLAYBACK ON (DC1), BEGIN READING RECORD
        @IOR          =>D000       CLEAR RREQ & WREQ, WRITE CHARS
READ     WDS           5           WRITE COMMAND & CHARACTER DATA
        DATA        0           NO BUSY BIT (DC1 IS 1ST WDS)
        RDS           5           READ DATA FROM ASR
        DATA        2080        BUSY BIT, REGISTER A
        BRU           $-2
End-of-Record Check {AND           =>7F           MASK OUT 9 MSB
        {CPL           =>13          CHK IF DC3 (PLYBCK OFF) READ
        {SNE
        {BRU           FINISH      BRANCH IF DC3 READ
        IOR           =>80          7 BIT ASCII TO 8 BIT ASCII, MARK PARITY
        STA           TABLE,X     STORE CHARACTER
        RIN           X,X          INCREMENT X-REG
        @LDA          =>8000       SET CLR READ REQ (CRR)
        BRU           READ        RESTART CHARACTER CYCLE
Count of Records Read {FINISH {DMT          COUNT        DECREMENT COUNT, SKIP NEXT IF COUNT = 0
        {BRU          START        GO TO NEXT RECORD
        COUNT        IDL           1           IDLE WHEN COUNT = 0
        TABLE      DATA        0           CONTAINS AMT OF RECORDS
        BSS          1000         STORAGE AREA

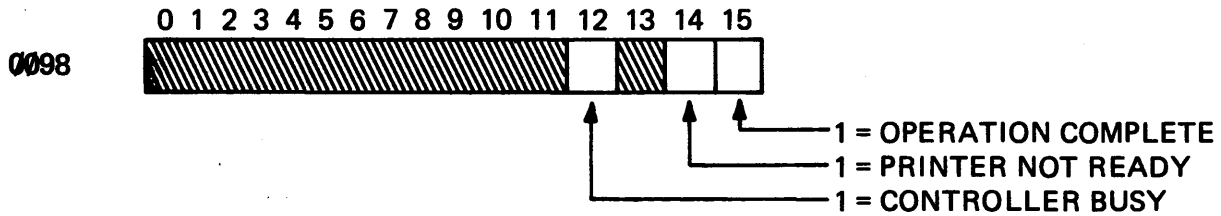
```

10-10 DAMC I/O: THE ATI INSTRUCTION.

Without expansion hardware, we have room for only one DMAC device: the high-speed line printer, the moving head disc or the magnetic tape transport. We won't discuss disc or tape operations herein, and consider the printer to be our only DMAC device. The instruction used for I/O to DMAC devices is an automatic transfer initiate (ATI) instruction rather than the RDS/WDS used for low-speed data bus devices. The command function to the device is built into the ATI instruction at assembly time. There are one or two status words (depending upon the device) and a list containing the buffer address, character count, disk address, and chaining information, as applicable.

10-11 SINGLE DMAC DEVICE: LINE PRINTER.

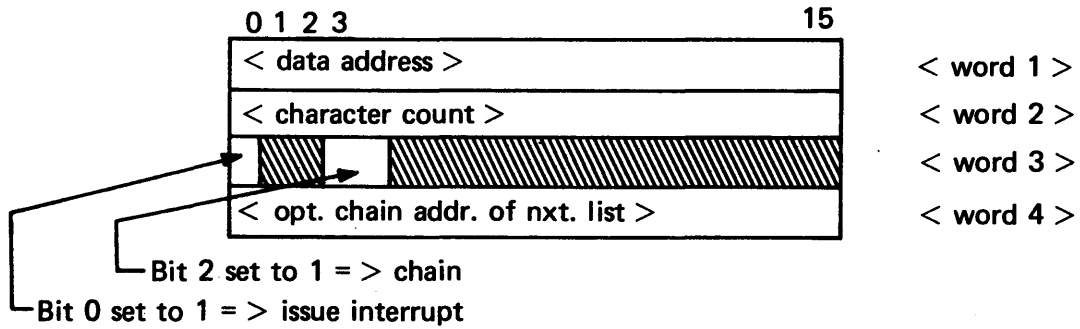
The status word is 0098* in the memory



and these bits will be set by the device.

*If a DMAC expander is used, the printer normally occupies expander position 5 and the status word occupies memory location 00A2.

The program must, in addition, define in memory a LIST, which for the line printer, consists of four words:



The data for the program is set up as follows:

STATUS	EQU	0098
...		
LIST	DATA	< address of buffer> , < character count>
	DATA	< interrupt/chain word> , < address of next list>

and the program itself sets the device status location to some illegal value, usually all zeros. After issuing the ATI, the status is checked for an error during operation. The ATI specifies the port (in the case of a single DMAC device, no expander, the port is 0).

LDA	=0	ILLEGAL VALUE
STA	STATUS	TO BE STORED IN STATUS WORD 0098
ATI	0	ISSUE ATI TO PORT 0
DATA	LIST	
LDA	STATUS	GET STATUS
SNZ	A	TEST FOR CHANGE
BRU	\$/-2	IF NONE, KEEP TESTING
CPL	=1	TEST FOR OK STATUS (OPERATION COMPLETE)
SEQ		CONTINUE IF OK
BRU	ERROR	IF NOT, PROCESS ERROR
<next instruction>		

SECTION 11

THE INTERRUPT SYSTEM

Table 11-1. Interrupt Instructions

LSB	LOAD STATUS BLOCK
SSB	STORE STATUS BLOCK
LRF	LOAD REGISTER FILE
SRF	STORE REGISTER FILE
LSR	LOAD STATUS BLOCK AND RESET INTERRUPT

11-1 INTRODUCTION TO THE INTERRUPT SYSTEM.

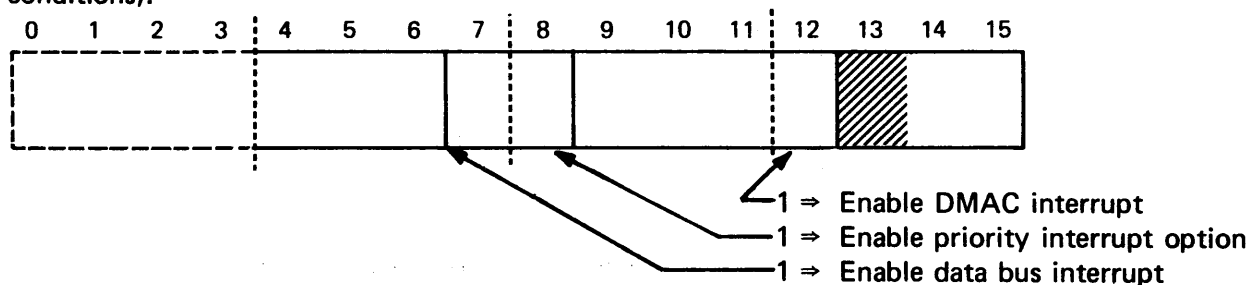
During execution of a computer job, conditions may arise in the system that require reasonably urgent attention. The disastrous effects of a power failure, as an extreme example, might be averted if all register contents could be saved until restoration of power. The program might be able to resume processing at the point of failure if the machine were restored to the condition it was in at the time.

Using the power fail warning (*interrupt*) whatever code is executing at the time is interrupted, and the program branches to a special routine (*interrupt service routine*) provided by the programmer to tell the machine what to do, when and if the condition arises. Not all cases of interruption are that urgent. One way of performing I/O involves issuing a transfer command to an external device along with the request that the device signal via interrupt when data transfer is complete. The CPU thus may proceed with other tasks and not "wait" for the device to complete the transfer.

The T1980 computer has interrupt capability in four levels of decreasing priority:

1. Internal interrupt (power fail, parity errors, etc.)
2. Priority interrupt option
3. DMAC interrupt
4. Data bus interrupt.

A look at the status register (Figure 11-1) reveals that each of the interrupt levels (except internal) corresponds to a bit which may be set under program control to enable that level of interrupt (i.e., the processor will allow interruptions) or disable the level (i.e., the processor will ignore the interrupt conditions).



status bits	meaning
01	
00	<
01	=
10	>
11	not allowed

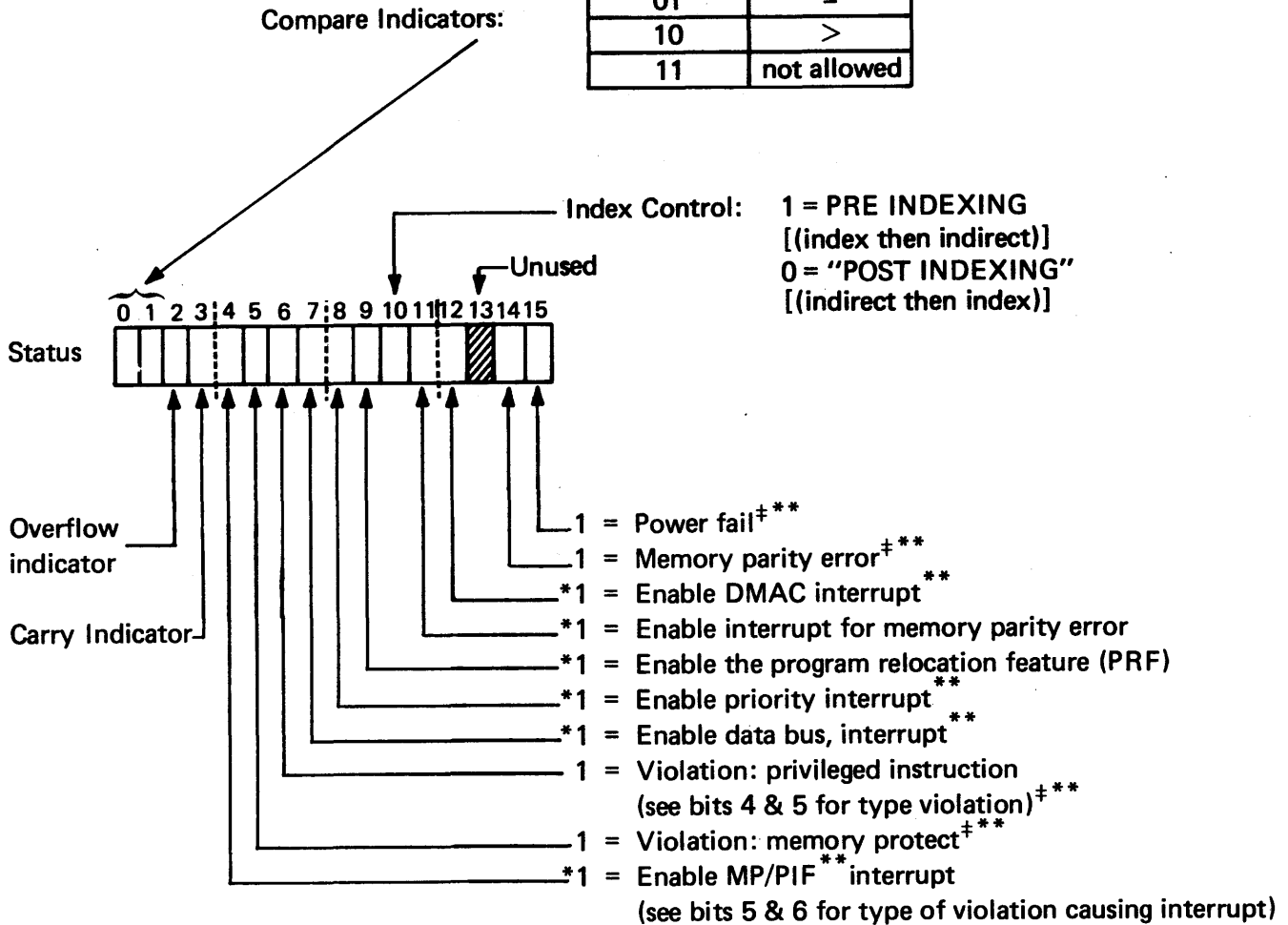


Figure 11.1. The Status Register

When an interrupt condition arises, the CPU, knowing the level of the interrupt, looks at these status register bits (*interrupt mask*) to determine whether the interrupt should be honored or not. The internal interrupt level is always enabled and cannot be disabled by the programmer, although two of the specific types of interrupt that occur at that level may be selectively disabled.

Note that the status register contains two kinds of bits:

1. Bits that are set by the programmer to enable or disable an interrupt feature.
2. Bits that are set by the machine to report a condition (or status). These bits may be interrogated by the program as the basis for a decision.

*Bits marked by * are set by the programmer to call in the interrupt features he desires. The other bits are set by the machine in response to encountering one of the interrupt conditions.

†Bits marked by † are unconditionally cleared by an LSB instruction.

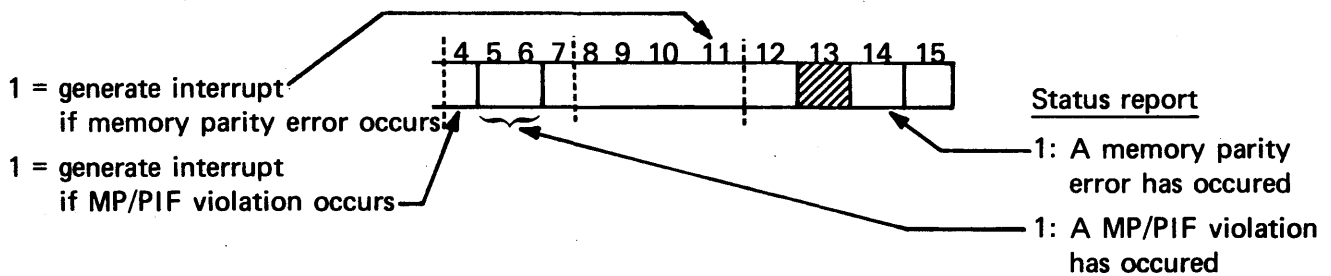
**Bits marked ** are cleared after execution of the instruction in location 0002 (i.e., internal interrupt).

11-2 INTERNAL INTERRUPTS.

An internal interrupt (highest priority) is generated by one of the following conditions:

- may not be disabled
1. Detection of imminent power failure
 2. Detection of an undefined op-code
 3. Detection of a memory parity error (disabled by setting bit 11 to zero); a memory parity error causes the processor to set bit 14 to a one, but no interrupt will be generated unless bit 11 is set to one (enable). Since memory parity errors are undesirable and we should be informed when they occur, this feature should normally be enabled
 4. Violation of the memory-protect/privileged instruction (MP/PIF) feature* (disabled by setting bit 4 to zero): violations are reported in status bits 5 and 6.

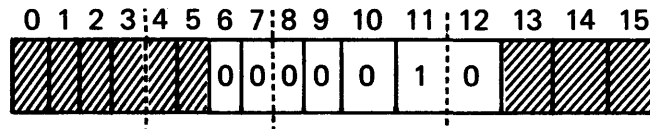
Enable interrupt



11-3 LOADING THE STATUS REGISTER.

When loading the status register let's assume we disable all interrupts except the memory parity error (which we generally prefer to keep enabled) and the power failure and illegal operation interrupts (which we cannot disable even if we wanted to).

At the outset of processing, we should load the status register to reflect the conditions we want:



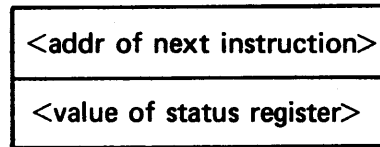
The shaded bits are set by the processor anyway, so we'll assume 0 in those positions. Thus, the number we wish to place in the status register is 0010. (Note that by setting bit 10 to zero, we get post-indexing: indirect followed by indexed in any operation involving both.)

11-3.1 THE LSB INSTRUCTION.

SBLOK	DATA	ENTRY+2	}	DEFINE STATUS BLOCK
STATUS	DATA	>0010		LOAD STATUS BLOCK
ENTRY	@LSB	SBLOK		
....				

*Discussed in Section 11-6.

The load-status block instruction (LSB) is written with an extension indicator (@) to indicate that it uses two words in the assembly listing. Its action is to load a *status block* consisting of two words located at the SBLOK address:



into the PC and status registers, respectively. The next instruction is taken as the one pointed to by the PC, which should be at ENTRY +2, since the LSB instruction uses the two words ENTRY and ENTRY +1. All interrupts (except for internal) are automatically disabled until the instruction pointed to by the new PC is executed. This LSB instruction costs us a bit more effort than necessary. We could load the status word into A and then move it into the status register:

	STATRG	EQU	>08	DEFINE STATUS REGISTER
	STATUS	DATA	>0010	INITIALIZE STATUS VALUE
ENTRY	LDA	STATUS		
	RMO	A,STATRG		PUT STATUS INTO STATUS REG.

and thereby not have to replace the contents of the PC.

The LSB instruction automatically clears the status bits 5, 6, 14, and 15 which indicate violations of MP/PIF, occurrence of a memory parity error, or power failure, since these conditions have presumably been tested for and handled in the program prior to the condition of LSB.

11-3.2 REGISTER OR INSTRUCTION. Another way to set individual bits in the status register is to use the ROR instruction with the status register as the destination and the bits to be set appearing as ones with zeros elsewhere:

LDA	MASK	GET NEW BITS TO BE SET
ROR	A,STATRG	OR TO STATUS

and, of course, bits may be selectively cleared by setting the corresponding bit positions in the mask to zero (with ones elsewhere) and performing a logical AND instruction (RAN).

11-4 HANDLING INTERNAL INTERRUPTS.

When the interrupt comes, it will be serviced at the completion of the instruction currently executing. The processor will always trap to location 0002 for an internal interrupt, and it is in this location where some instruction will be found that tells the machine what to do next (see Appendix E for a list of trap locations). Since the only space available is the word pair at 0002 and 0003, we have room for (at most) a double-length instruction which usually is a store-status-block-and-branch (SSB).

The SSB instruction has an address as its operand, and it is at this address that the processor stores its status block (PC and status register). Immediately following the locations reserved to hold the status block, it will expect to find the appropriate interrupt service routine and will, therefore, commence execution at that point (i.e., at the stored status block +2):

In Trap Location

ORG 0002
@SSB IIHNDL
.....

In User Area

IIHNDL EQU \$
BSS 2 SAVE ROOM FOR S. B.
<first executable instruction of
interrupt service routine>
.....
@LSB IIHNDL Return to
Interrupted Execution

The effect of the interrupt is really just a branch indirect through 0003 to the address IIHNDL, followed by a store of the status block at IIHNDL and resumption of execution with (PC) = IIHNDL+2.

Return from the interrupt service routine is commonly done by an LSB of the saved status block (which contains the old PC value and the status; the machine settable bits are cleared – otherwise another interrupt corresponding to the one just serviced would occur immediately).

This SSB instruction allows no interrupts until the instruction following the SSB has been executed (in this case, it is the instruction at IIHNDL+2).

What instruction should we put into the interrupt service routine starting at IIHNDL+2? In the present example, we have disabled all internal interrupts except power failure, illegal instruction, and memory parity error. Therefore, the interrupt must have been caused by one of these three conditions.

Since we have 1 millisecond grace before the power fails, we should check that condition first:

	TMBO	15,IIHNDL+1	TEST FOR POWER FAIL
	BRU	NXTEST	NO, BRANCH TO NEXT TEST
	@SRF	REGSAV	YES, SAVE REGISTER FILE:A,E,X,M,L,S,B
	IDL	0	WAIT FOR FAIL
REGSAV	BSS	7	
NXTEST	—		
	—		
	—		

If the power is okay, the next test will be for a memory parity error in bit 14. If not set, we have an illegal op code.

NXTEST	TMBO	14,IIHNDL+1	MEMORY PARITY ERROR?
	BRU	BADOP	NO, HANDLE ILLEGAL OPERATION
	—		YES, PRINT ERROR MESSAGE
	—		
	—		
	IDL	1	WAIT FOR OPERATOR INTERVENTION
BADOP	—		BAD OPERATION CODE, MESSAGE
	—		
	—		
	—		
	IDL	2	WAIT FOR OPERATOR INTERVENTION

These interrupts all indicate potentially serious problems; hence the IDL rather than an LSB to resume the interrupted process. Other kinds of interrupts, which are of a routine nature (e.g., an interrupt signalling completion of I/O) and can be totally handled by the machine, use the LSB instruction at this point.

A summary of LSB/SSB instruction behavior is found in Table 11-2.

Table 11-2. Load/Store Status Block (LSB/SSB)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	COMPARE		OVERFLOW	CARRY	ENB MP/PIF	MP VIOLATION	PIF VIOLATION	ENB DATA BUS INTERRUPT	ENB PRIORITY INTERRUPT	ENB PRF (RELOCATION)	PRE-/POST-INDEXING CONTROL	ENB MEM PARITY INTERRUPT	ENB DMAC INTERRUPT	(UNUSED)	MEMORY PARITY ERR INDICATOR	POWER FAIL INDICATOR
Cleared after execution of instruction in trap location 0002 (internal interrupt trap)					0	0	0	0	0	0		X	0		0	0
Until execution of one instruction past LSB, only internal interrupts are enabled					1			0	0			1	0			
After execution of one instruction past LSB/SSB					1			X	X	X		X	X			
Execution of LSB clears unconditionally:						0	0								0	0
LSB is illegal instructions if					1											

ST
EN ST
EN EN
EN
ST

11-5 STARTUP AFTER POWER FAILURE.

Having made provision in the last sub-section for handling the power failure interrupt by storing the register file, let's examine how we can restore the program when power is restored.

At powerup the PC is set to zero, and execution begins in the ~~0000~~ location. Thus, in that location we could store a link to take the program to a register-restore routine. Since we will replace all register contents, including the PC, execution can resume as if nothing happened. Thus, in location 0:

ORG 0	RESTOR	DATA	RESTOR+2,0	GET S, B, HAVING REG FILE LOAD
@LSB RESTOR		@LRF	REGSAV	RELOAD REGISTERS
-		@LSB	IIHNDL	GET SAVED STATUS & PC
-				
-				
-				

dummy status
↙

where the first act of the LSB is to load the register file restore instruction address into the PC and load a dummy status into the status register. Then the status block containing the actual values of the PC and status at the time are interrupted, and execution proceeds with the instruction pointed to by the PC.

The same thing can be accomplished without using the dummy status by writing

ORG 0	RESTOR	@LRF	REGSAV
BRU RESTOR	@LSB	IIHNDL	

thereby saving two memory locations.

Recall, in our use of the basic operating system, that powerup initializes the operating system so that *READY* is printed at the console. We can do the same thing with our program (i.e., initiate execution by depressing the START switch) by making sure we deposit in the IIHNDL status block: (1) a PC value equal to the program entry point and (2) a status equal to the initial value of the status register just prior to termination of the program. The machine, unable to differentiate startup from restart after power failure, will load the register file with garbage (Who cares? They will be initialized by the program anyway!) and place the entry point into the PC. So to include this automatic startup option, we conclude our program with

LDA	=ENTRY	
STA	IIHNDL	SAVE ENTRY POINT
LDA	STATUS	
STA	IIHNDL+1	SAVE INITIAL STATUS
IDL	0	HALT

Following is the entire power fail protect and automatic startup routine.

	ORG 0		
	@LSB	RESTOR	(RE)INITIALIZE
	@SSB	SAVE	
	-		
	-		
	-		
ENTRY	—————→ PROGRAM ENTRY POINT		
	-		
	-		
PROGRAM EXIT	LDA	=ENTRY	GET ENTRY POINT ADDRESS
	STA	SAVE	SAVE FOR RESTART
	LDA	STATUS	GET INITIAL STATUS WORD SETTING
	STA	SAVE+1	SAVE FOR RESTART
	IDL	0	
	-		
	-		
	-		
STATUS	DATA	<appropriate hex number for initial status>	
SAVE	DATA	ENTRY, <initial status>	
	TMBO	15,SAVE	TEST POWER FAIL
	BRU	ERROR	NO, BRANCH TO ERROR IDENTIFYING ROUTINE
	@SRF	REGSAV	YES, SAVE REGISTER FILE
	IDL	1	WAIT FOR FAIL
REGSAV	BSS	7	REGISTER FILE STORAGE A, E, X, M, S, L, B
RESTOR	DATA	RESTOR+2,0	← dummy status
	@LRF	REGSAV	RESTORE REGISTER FILE
	@LSB	SAVE	RESTORE STATUS BLOCK
	END		

NOTES

1. LSB and SSB – turn off interrupts until one more instruction has executed.
2. LSB – loads status block, BR to (PC)
3. SSB – stores status block, continues executing with STATUS BLOCK+2.

11-6. MEMORY PROTECT/PRIVILEGED INSTRUCTION FEATURE (MP/PIF).

When enabled through setting bit 4 of the status word, the MP/PIF provides one or both of the following features.

11-6.1 MEMORY PROTECT. There are cases, as in the development of an operating system, that the user is restricted to a certain area of memory. This sort of restriction is valuable in some circumstances in that it protects the neophyte user from inadvertently trespassing the bounds of the operating system and destroying part of it.

Consider the 980 basic operating system. This system sets the memory protect limits so that the user cannot enter the operating system inadvertently, and in so doing the system preserves its own integrity. Consider what decision you might make about memory protection. A user of your system who inadvertently destroys it would be annoyed at you for not protecting it, and another user will be angry that your system does not permit him the same privileges he has on a bare machine (he is perfectly happy to suffer the consequences of his own actions.) Your decision of "to be or not to be" with regard to memory protection cannot possibly please everyone. Your best decision with respect to building the operating system is probably to protect it, but in such a way that a knowledgeable user can bypass your safeguards if he so wishes.

In the case of a time-sharing system, the ethical question is a bit different. There are potentially a number of users and the system should be designed to protect itself and its users from the misdeeds of others.

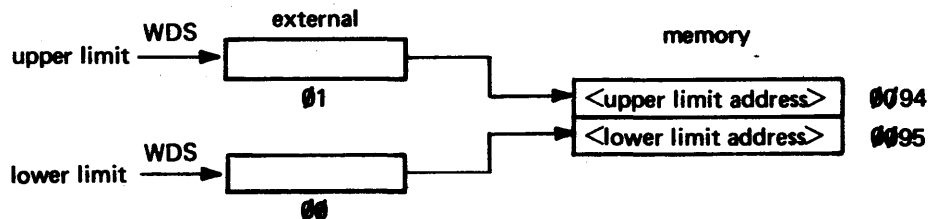
11-6.2 PRIVILEGED INSTRUCTIONS. Hand-in-hand with the question of memory protection goes the concept of privileged instructions. Certain instructions may do as much damage (albeit, a different kind) as permitting unrestricted access to memory. These instructions include:

- Use of idle (IDL) instruction: If the casual or time-shared user wishes to halt, control should return to the operating system.
- Changing the status register: A great deal of the behavior of the machine is determined by the settings in the status register, and allowing casual or time-shared users access to it may court disaster.
- Especially in a time-shared environment, free access to I/O capability can lead, among other things, to the appearance of spurious data in some user files. Therefore, I/O is best handled by the supervisor through a request made to it by the program.

To summarize, when the MP/PIF is enabled (by setting bit 4 of the status word to one) — as it is while operating in the environment of the 980 basic operating system — the user program is prevented from:

1. Read/write/branch into protected memory
2. Changing the status register
3. Performing I/O
4. Bringing the computer to idle.

If a violation of any feature is detected (bits 5 or 6 of the status word) an internal interrupt is generated, the user is informed of his transgression and control is taken away from the user program by the supervisor program. Instructions which are privileged belong to a certain class (see Appendix A); however, the limits of memory protection must be set. Setting the limits is done by writing the lower bound to external register 00 and the upper bound to external register 01. The values written into these registers will be stored by the computer in the low memory locations 0095 and 0094, respectively:



where they will be used as soon as the MP/PIF feature is enabled by setting bit 4 of the status register.

The memory limits may be reset under program control by writing the new limits out to the external address registers. The need for such a scheme becomes apparent if we consider the case in which the lower limit (location 0095) is set to, say, memory address 11D2. This means that any reference to a location lower than 11D2 will result in an error. Thus, we are unable to successfully write a new limit directly into 0095 because that cell is protected along with all the others between 0000 and 11D2.

11-7 PROGRAM RELOCATION FEATURE.

When a user program has been assembled, the monitor may load it anywhere in memory. If the monitor sets bit 9 just prior to transferring control to the user program, the program can be executed by having the monitor place the actual load address minus 1 into the lower limit register and performs LSB to transfer to program. LSB sets bit 9 of the status register and loads PC with absolute entry point:

RELOC	DATA	<absolute entry point>
	...	
	LSB	RELOC

The program now will execute as if it were located in absolute entry point and lower limit +1.

11-8 DATA BUS AND DMAC INTERRUPTS.

Used for I/O operations these interrupts are most useful in situations in which the CPU has other work it could do while waiting for a device to become ready or while waiting for an autonomous I/O transfer to become complete. One salient example is encountered in keyboard input problems in which

1. It may be necessary for the operator to get control away from a program
- OR -
2. The program may wish to maintain an attentiveness to any activity at the keyboard without spending a great deal of time polling the keyboard for inputs that may occur at random intervals.

In the absence of expander logic, up to four data bus devices and only one DMAC device can be connected to 980. With the single DMAC device there is no question of which device requires service; however, if all data bus devices are capable of generating interrupts, some effort is necessary to resolve which device originated the signal. (All the CPU knows to begin with is that an interrupt has been issued by the data bus.)

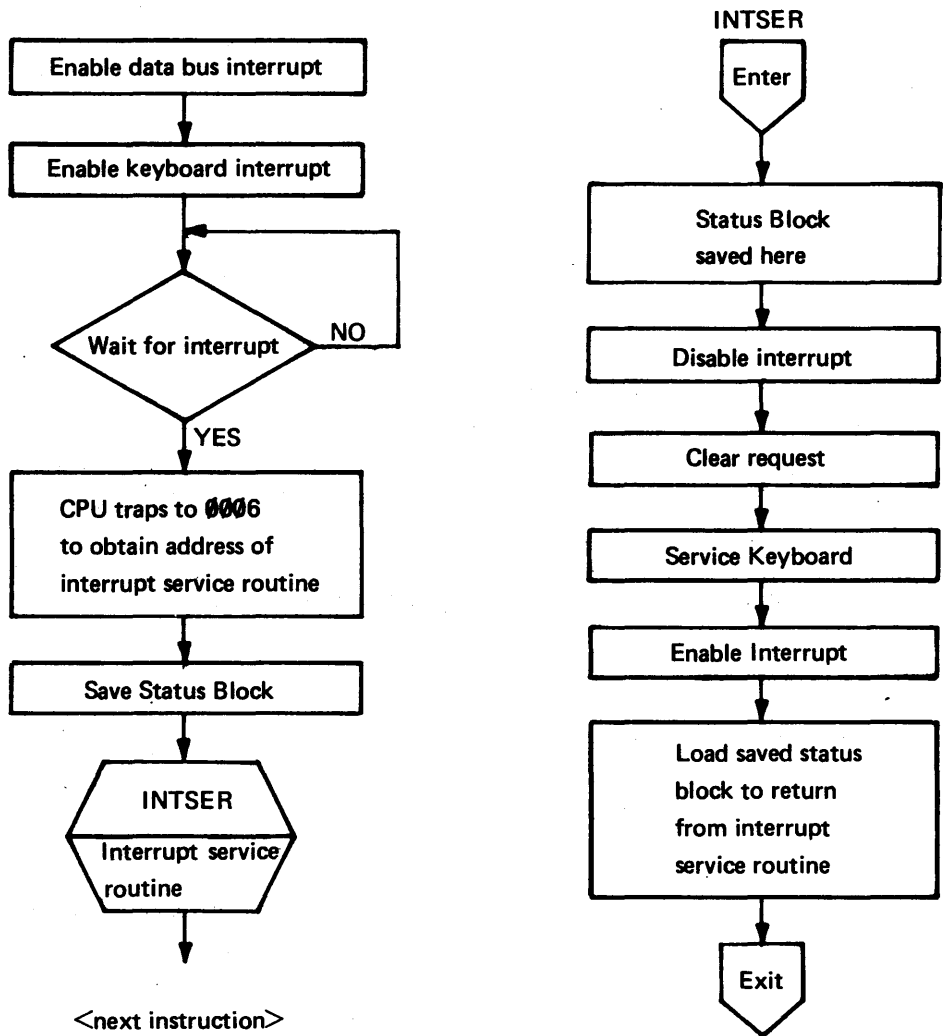
11-9 DMAC INTERRUPT.

If bit 12 of the status register is set (the DMAC interrupt enable bit), an interrupt from the DMAC device will cause the program to trap to memory location 0004. We handle this trap with an SSB just as in the case of data bus interrupts.

The SSB instruction clears status register bits 4, 5, 6, 7, 9, and 12, and the device status information is stored in the appropriate status location(s).

11-9.1 DATA BUS INTERRUPT. If the CPU is to recognize data bus interrupts, the appropriate enable bit (bit 7) of the status register must be set. We need to tell the devices at the other end of the data bus which one (or more) is to generate these interrupts. Thus, for the interrupt scheme to work, we must tell the appropriate devices to generate interrupts and also tell the CPU to recognize the interrupts. For simplicity, let's consider a 980 with a teleprinter, a high-speed paper tape punch, a card reader. In the following example, we will enable the teleprinter interrupt only (the data bus interrupt in the CPU also must be enabled.)

Keyboard Interrupt Service Routine:



The first thing we need to worry about in the interrupt service routine is preventing further interrupts from being serviced until we have finished servicing the current one. (If we were to allow the interrupt service routine to be interrupted, the current stored status block would be overlaid with a new one, and we could never find our way back through the nested interrupts.) We can disable the data bus interrupt by:

1. Disabling the interrupt generation capability of all the devices on the bus
- OR —
2. Disabling the interrupt recognition capability of the mask in the status register.*

We will elect option 2. At least this way an interrupt generated during the period of "disablement" can be "saved up" and honored as soon as the original mask is restored. We will have to clear the interrupt condition from the device. This is usually done automatically by a read of the device status word. If the interrupt request is not cleared, then, as soon as the interrupt has been serviced and the interrupt system is enabled, the persisting request will appear to be a new interrupt.

An example showing the enabling of the keyboard interrupt and the corresponding interrupt service routine is given below.

***ENABLE KEYBOARD AND INTERRUPT**

```

STATUS      EQU      8
DBTRAP      ORG      @SSB      KBHNDL
            ...
            LDA      =>58      ENB KB & INT.—FULL DPLX,
            WDS      >A        AND SEND TO COMMAND REGISTER.
            DATA    >2000
            LDA      =>8        SET DATA BUS INT. ENB BIT AND
            ROR      A,STATUS   OR TO CURRENT STATUS
            —
            —
            —
            BRU      $          STOP

```

***KEYBOARD INTERRUPT SERVICE ROUTINE**

```

KBHNDL      BSS      2          SAVE ROOM FOR STATUS BLOCK
            LDA      LOC      GET ADDRESS OF NEXT ARRAY ELEMENT
            RDS      >2        READ A CHARACTER
            DATA    >20A8     TO THAT ADDRESS AND INCREMENT.
            STA      LOC      SAVE NEXT ADDRESS.
            LSB      KBHNDL    RETURN FROM INTERRUPT
LOC          DATA    INBUF

```

*Done automatically by hardware at the trap location for data bus and DMAC interrupts.

11-9.2 COMPETING DATA BUS DEVICES. Let's consider the situation in which we would like one data bus device capable of interrupting the servicing of another: for example, an interval timer interrupt of a keyboard service routine. As long as we are willing for the keyboard service routine to finish before processing the clock interrupt, we have no problem. If instead, we wish the clock interrupt to be serviced immediately, we must set the interrupt enable bit, which gives us the nested interrupt problem. The way around the problem is to use the vectored priority interrupt option, which we assume is installed on our hypothetical machine.

11-9.3 VECTORED PRIORITY INTERRUPT OPTION. Using this option allows each device (or device group if expander logic is used) – whether on the DMAC or data bus – to have its own trap location. This arrangement has two advantages:

1. It eliminates the software overhead of deciding which of the devices issued the interrupt.
2. We may leave interrupts enabled – disabling only the device we are currently servicing.

This scheme allows one device to interrupt another with no possibility of destroying our return pathway to the beginning of the interrupt chain. If one device is permitted to interrupt another, it must be assigned a higher priority. Appendix E shows the priority traps, located between locations 0008 and 0047. An interrupt trapping to word pair 0008/0009 will interrupt any other interrupt currently being serviced except an internal interrupt which has highest priority.

These vectored interrupts provide a range of decreasing priority that fits between internal interrupt and standard (nonvectored) DMAC interrupts.

1. Internal (highest priority of all)
2. Vectored interrupt option (data bus or DMAC):
 - 0008 (highest priority)
 - 000A
 -
 - 0046 (lowest priority)
 - (0048-0087 are unused)
3. Regular DMAC interrupts
4. Regular data bus interrupts (lowest priority of all).

We shall use an SSB in the trap location to branch to the appropriate service routine. At the conclusion of the service routine is an LSR instruction (like an LSB, except that it resets the interrupt just serviced before returning control to the interrupted program). Assume we use up one data port on the vectored priority interrupt option. We would connect in a board giving us eight double-word trap locations. If we had room for a second board, we could get a second block of eight, giving the sixteen traps that comprise group 0:*

<u>Operation</u>	<u>External Register</u>
Set or Read Interrupts	> 5A
Reset Interrupt	> 5B
Mask or Read Mask	> 58
Unmask	> 59

Each of the 16 bits in these external registers corresponds to the state of one of the 16 priority interrupt locations (0008 – 0026).

*We may have up to three other groups of 16, totaling 64. The other groups each have their own unique set of four external register addresses.

We may perform the following operations on any or all of the 16 locations:

Inhibit Interrupts Write a mask word to external register >58:

(this word may also be read)
vector location 0008



vector location 0026

1 ⇒ inhibit interrupt
0 ⇒ leave in current status

Allow Interrupts Write an unmask word to external register >59:



1 ⇒ enable interrupt
0 ⇒ leave in current status

Create Interrupts Write the appropriate bit pattern into external register >5A: (this word may also be read)



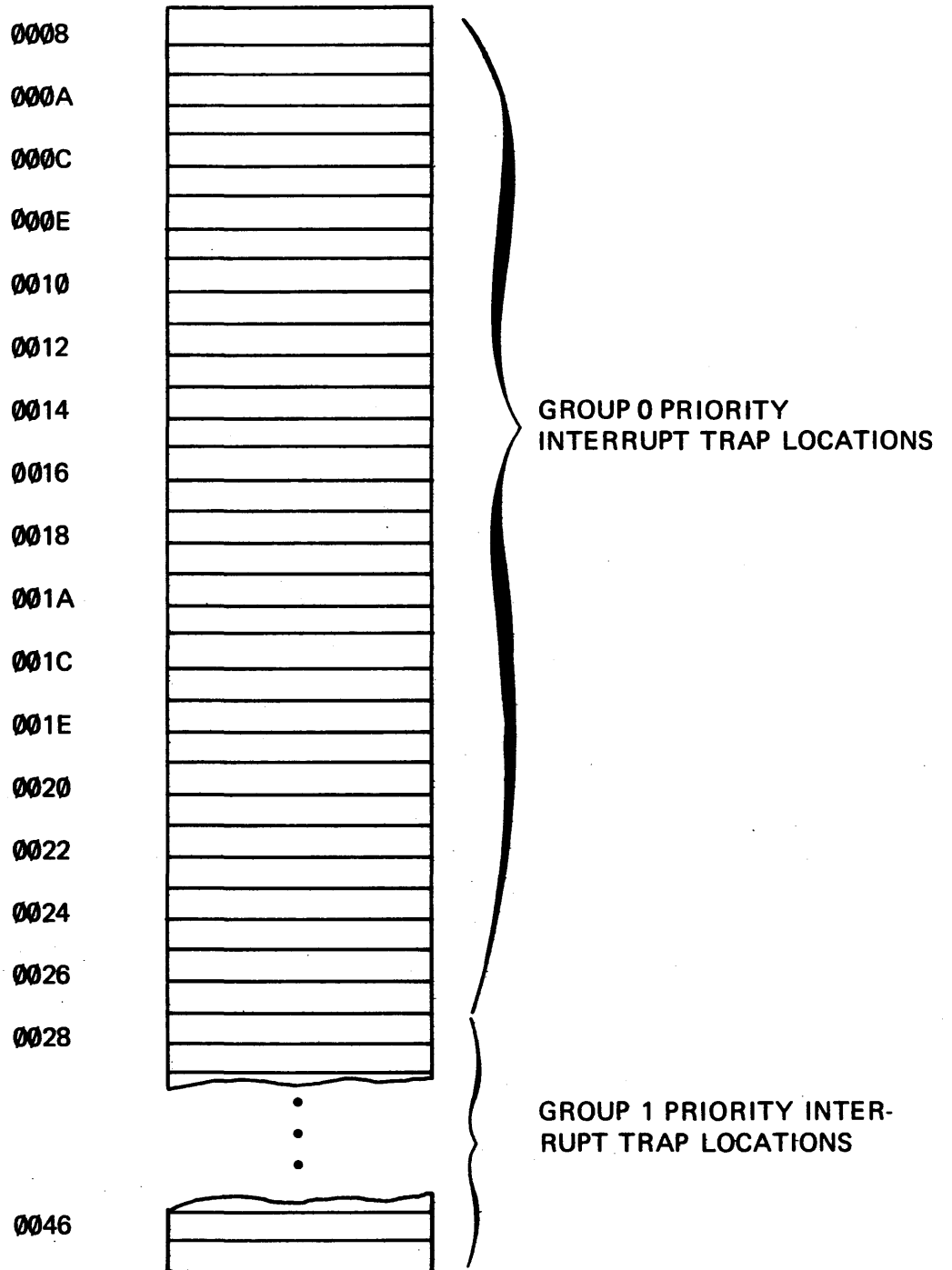
1 ⇒ create interrupt

Reset Interrupts Write appropriate bit pattern in external register >5B:



1 ⇒ reset interrupt

These 16 bits are keyed to the first 16 priority interrupt trap locations:



APPENDIXES

APPENDIX A

980AL EXECUTION TIMES

The following symbols are used in execution time lists.

m	= memory address	\wedge	= logical "and"
<mod>	= modifier	∇	= logical "exclusive or"
e. a.	= effective address	~	= complement
→	= replaces	s	= source register
()	= contents of	d	= destination register
PC	= program counter	y	= immediate operand
AE	= concatenated A and E registers	n	= bit number
\subset	= concatenation symbol	f	= flag
\vee	= logical "or"	dev	= device address

<u>INSTRUCTIONS</u>				<u>EXECUTION TIME (μsec)</u>	
<u>REGISTER/MEMORY TRANSFER</u>				<u>MEM REF*</u>	<u>IMMED</u>
0000	LDA	m, <mod>	(e.a.) → A	1.75	0.75
0800	LDE	m, <mod>	(e.a.) → E	1.75	0.75
1000	LDX	m, <mod>	(e.a.) → X	1.75	0.75
1800	LDM	m, <mod>	(e.a.) → M	1.75	0.75
8000	STA	m, <mod>	(A) → e.a.	2.00	2.00
8800	STE	m, <mod>	(E) → e.a.	2.00	2.00
9000	STX	m, <mod>	(X) → e.a.	2.00	2.00
<u>REGISTER/MEMORY TRANSFER (double-length)</u>					
B000	DLD	m, <mod>	{ (e.a.) → A { (e.a.)+1 → E	2.75†	1.0†
A000	DST	m, <mod>	{ (A) → e.a. { (E) → e.a.+1	2.75†	2.75†
<u>Memory/Memory</u>					
5000	IMO	m, <mod>	(e.a.)+1 → e.a.	2.75	2.75
4800	DMT	m, <mod>	(e.a.)-1 → e.a.; then (e.a.)=0 ⇒ (PC)+1 → PC (i.e., skip)	2.75	2.75
<u>Arithmetic</u>					
2000	ADD	m, <mod>	(e.a.)+(A) → A	1.75	0.75
2800	SUB	m, <mod>	(A) - (e.a.) → A	1.75	0.75
6800	CPA	m, <mod>	(e.a.) : (A) → Status	1.75	0.75

*Add 0.25 μ sec for indexing, 0.75 μ sec for indirect

†Add 0.25 μ sec for extended format

EXECUTION TIME (μsec)

Arithmetic (double length)

				<u>MEM REF*</u>	<u>IMMED</u>
9800	MPY	m, <mod>	(A)*(e.a.) → AE	2.25–6.25	1.25–5.25
5800	DIV	m, <mod>	{ (AE)/(e.a.) → A remainder → E	2.50–7.75	1.50–6.75
B800	DAD	m, <mod>	(AE)+(e.a., e.a.+1) AE	2.75†	1.0†
A800	DSB	m, <mod>	(AE)–(e.a., e.a.+1) AE	2.75†	1.0†

Logical

3000	IOR	m, <mod>	(e.a.) (A) → A	1.75	0.75
3800	AND	m, <mod>	(e.a.) (A) → A	1.75	0.75
6000	CPL	m, <mod>	(e.a.):A → Status	1.75	0.75

REGISTER/REGISTER**

EXECUTION TIME (μsec)

C500	RMO	s,d	(s) → d	1.00
C780	REX	s,d	{ (s) → d (d) → s	1.50
C300	RIN	s,d	(s) +1 → d	1.00
C700	RDE	s,d	(s) –1 → d	1.00
C080	RAD	s,d	(s) +(d) → d	1.25
C000	RSU	s,d	(d) –(s) → d	1.25
C100	RCO	s,d	0 – (s) → d	1.00
C680	RAN	s,d	(s) Δ (d) → d	1.25
C480	ROR	s,d	(s) ∇ (d) → d	1.25
C280	REO	s,d	(s) ∇ (d) → d	1.25
C200	RIV	s,d	~ (s) → d	1.00
C400	RCA	s,d	(s) : (d) signed → Status	1.25
C600	RCL	s,d	(s) : (d) unsigned → Status	1.25

CIRCULAR SHIFTS (single length)

$$.75 + \frac{y}{4}$$

CA00	CRA	y	} End around Right shift, y bits
CA20	CRE	y	
CA40	CRX	y	
CA60	CRM	y	
CB20	CRS	y	
CB40	CRL	y	
CB60	CRB	y	

(double length) End around

CB80	CLD	y	} y bits
CBC0	CRD	y	

**Privileged if d = PC

EXECUTION TIME (μ sec)

LOGICAL SHIFTS (single length)

C8C0	LLA	y	A left shift, zero fill
C840	LRA	y	A right shift, zero fill
(double length)			
C8E0	LLD	y	AE Left Shift, zero fill
C860	LRD	y	AE Right Shift, zero fill

ARITHMETIC SHIFTS

(single length)

C880	ALA	y	A left shift y bits, zero fill	0.75 + y/4
C800	ARA	y	A right shift y bits, sign extend	

(double length)

C8A0	ALD	y	AE left shift zero fill, $E_1 \rightarrow A_{15}$	$\left. \begin{array}{l} y = 0 \quad 1.00 \\ y \neq 0 \quad 0.75 + y/4 \end{array} \right\}$
C820	ARD	y	AE right shift sign ext., $A_{15} \rightarrow E_1$	
CA9F	NRM		AE left shift, zero fill $E_1 \rightarrow A_{15}$ until $A_0 \neq A_1$; $A_0 \rightarrow E_0$, # Shifts $\rightarrow X$	1.0 - 8.75

BRANCH

				<u>MEM REF</u>	<u>IMMED</u>
7800	BRU	m, <mod>	e.a. \rightarrow PC	1.25	1.00
4000	BIX	m, <mod>	(X)+1 \rightarrow X, then (X) \neq 0 \Rightarrow e.a. \rightarrow PC	1.25	1.25
7000	BRL	m, <mod>	(PC) \rightarrow L e.a. \rightarrow PC	1.50	1.50

MEMORY SKIP

4800	DMT	m, <mod>	(e.a.) -1 \rightarrow e.a. then (e.a.) = 0 \Rightarrow (PC) +1 \rightarrow PC	2.75	2.75
------	-----	----------	--	------	------

REGISTER SKIP

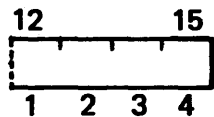
CCE0	SPL	S	(s) \geq 0 \Rightarrow (PC) +1 \rightarrow PC	1.0
CC60	SMI	S	(s) \leq 0 \Rightarrow (PC) +1 \rightarrow PC	
CC00	SZE	S	(s) = 0 \Rightarrow (PC) +1 \rightarrow PC	
CC80	SNZ	S	(s) \neq 0 \Rightarrow (PC) +1 \rightarrow PC	
CCA0	SNO	S	(s) \neq FFFF \Rightarrow (PC) +1 \rightarrow PC	
CC20	SOO	S	(s) = FFFF \Rightarrow (PC) +1 \rightarrow PC	
CC40	SOD	S	(s) = odd \Rightarrow (PC) +1 \rightarrow PC	
CC00	SEV	S	(s) = even \Rightarrow (PC) +1 \rightarrow PC	

STATUS SKIP

CD20	SEQ		skip if status =	1.0
CDA0	SNE		\neq	
CD00	SLT		<	
CDC0	SLE		\leq	
CD80	SGE		\geq	
CD40	SGT		>	

EXECUTION TIME (μsec)

STATUS SKIP

			skip (PC ← PC +1) if status:	1.0
CD60	SOV		= overflow	
CDE0	SNV		≠ overflow	
CF60	SOC		= carry	
CFE0	SNC		≠ carry	
CC10	SSE	SS	skip if ss switches all on	
				← operand field
				ss = 0, ... 15
CC90	SSN	SS	skip if any ss switches off	← switch #

BIT OPERATIONS

SET

DB40	SABZ	n	$A_n \leftarrow 1$	1.0
DB70	SMBO	n,m	$m_n \leftarrow 1$	3.25

CLEAR

DB50	SABO	n	$A_n \leftarrow 0$	1.0
DB60	SMBZ	n,m	$m_n \leftarrow 0$	3.25

SKIP

DB00	TABZ	n	$A_n = 0 \Rightarrow (PC)+1 \rightarrow PC$	1.25
DB10	TABO	n	$A_n = 1 \Rightarrow (PC)+1 \rightarrow PC$	1.25
DB20	TABZ	n,m	$m_n = 0 \Rightarrow (PC)+1 \rightarrow PC$	2.75
DB30	TABO	n,m	$m_n = 1 \Rightarrow (PC)+1 \rightarrow PC$	2.75

CHARACTER (BYTE) OPERATIONS

DF00	MVC		Move byte string	4.75 + (2.75/byte)
DF80	CLC		Compare byte string	5.00 + (2.25/byte)

I/O

D800	RDS	dev	Read from device dev	3.00 – 4.75
D820	WDS	dev	Write on device dev	3.00 – 5.00
D900	ATI	dev	Automatic Transfer/Initiate to/from device dev	2.5

STATUS BLOCK & REGISTER FILE

D8C0	SSB	m	{ (PC) → m; (Status) → m+1; (m+2) → PC	3.25
D880	LSB	m	{ (m) → PC; (m+1) → status	3.25
D890	LSR	m		3.25
D8A0	LRF	m	(M), ..., (m+6) → A,E,X,M,L,S,B	7.0
D8E0	SRF	m	(AEXMLSB) → m, ..., m+6	7.0

EXECUTION TIME (μ sec)

MISCELLANEOUS

CE00	IDL	f	f=0,...,> F	1.0
C900	RTO	y		1.0 + y/4
C940	RTZ	y		1.0 + y/4
C980	LTO	y		1.0 + y/4
C9C0	LTZ	y		1.0 + y/4
DD00	API	dev		AP controller dependent

APPENDIX B

980 REGISTER DESIGNATIONS

	<u>REGISTER NUMBER</u>	<u>USUAL PROGRAM DESIGNATION</u>
	0	A
	1	E
	2	X
	3	M
	4	S
	5	L
	6	B
Status	{ 7	P
Block	{ 8	ST (Status)

APPENDIX C

HEXADECIMAL ARITHMETIC

ADDITION TABLE

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

MULTIPLICATION TABLE

1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	13	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

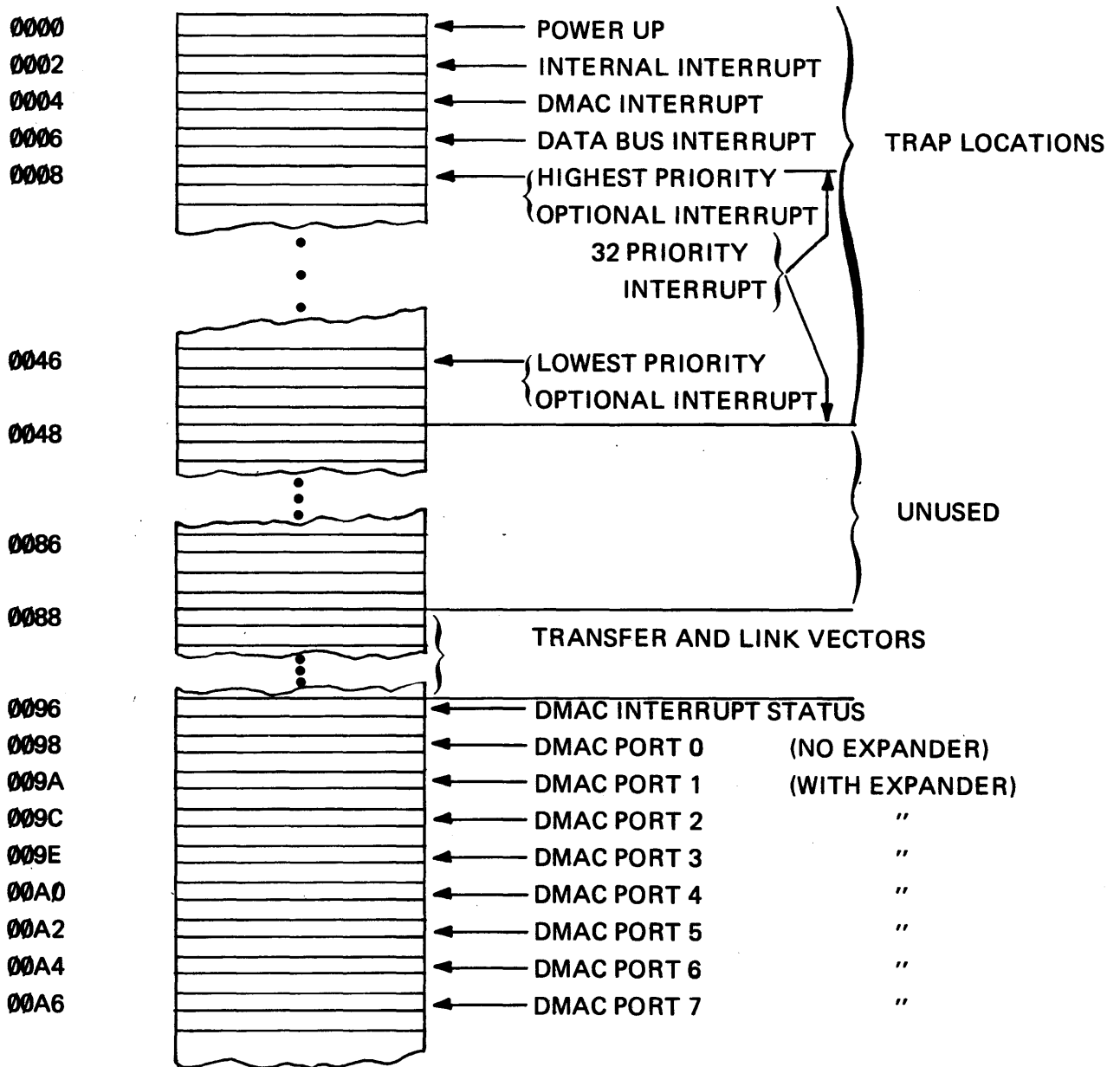
APPENDIX D

STANDARD ADDRESS OF EXTERNAL REGISTERS

<u>EXTERNAL REGISTER NO.</u>		<u>Read/Write</u>
00	<input type="text"/>	Timer Status/Lower Bound MP
01	<input type="text"/>	_____ / Upper Bound MP
02	<input type="text"/>	TTY1 DATA
05	<input type="text"/>	733 ASR/KSR1
0A	<input type="text"/>	TTY1
10	<input type="text"/>	PTR
11	<input type="text"/>	PTP
13	<input type="text"/>	Interrupt Expander
14	<input type="text"/>	Parity
15	<input type="text"/>	Parity
16	<input type="text"/>	Parity
18	<input type="text"/>	Paper Tape Data
1F	<input type="text"/>	CDR
40	<input type="text"/>	CDP
58	<input type="text"/>	Vect. Int. Expander
59	<input type="text"/>	_____ / Unmask
5A	<input type="text"/>	Read INT/Set INT.
5B	<input type="text"/>	_____ / Reset Int.

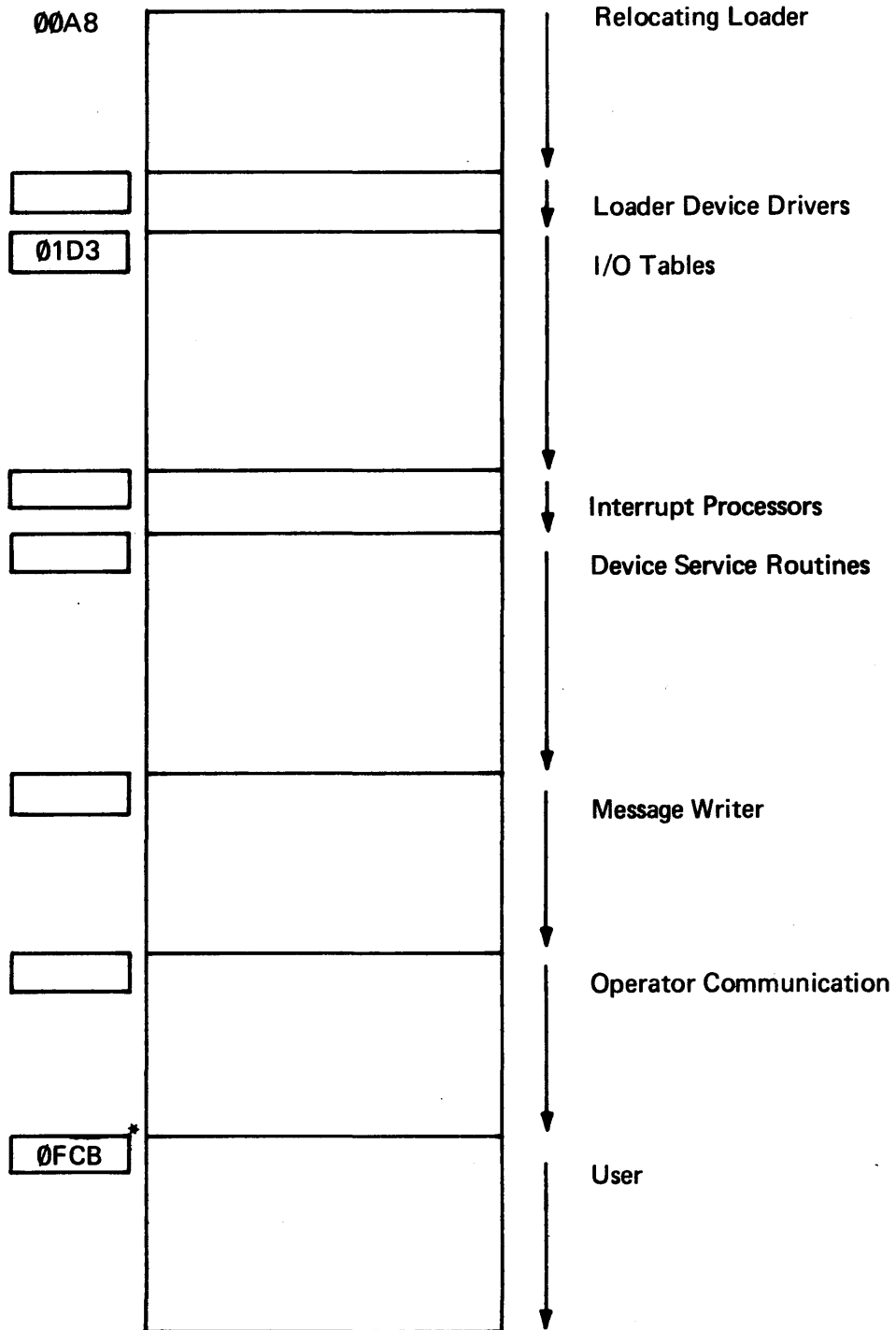
APPENDIX E

SPECIALIZED LOW-ORDER MEMORY LOCATIONS



APPENDIX F

BASIC SYSTEM MEMORY MAP



*This address will vary from system to system. It can be found by examining the lower address limits (LL) register, 0095.

APPENDIX G

SAP ERROR MESSAGES

The various versions of the assembler may detect certain syntax errors in the source program. When this occurs, a diagnostic message or corresponding message number is printed in the listing adjacent to the line in question.

<u>Message Number</u>	<u>Message</u>	<u>Meaning (and Corrective Action)</u>
1	FIELD SZ	Address beyond reach (use @ for extended format)
2	UNDF OP	Undefined operation code (check list of valid op codes)
3	LONG SYM	Symbol > 6 characters
4	MDF O/F	OPD or FRM multiply defined (rename label)
5	FRM > 16	FRM fields contain more than 16 bits
6	CAD > 10	Address expression has 10 elements
7	UNDF SYM	Symbol not defined (label probably omitted)
8	MDF SYM	Symbol multiply defined (rename labels)
9	RELOC	A relocation error (use only one relocatable label in arithmetic expression, or ORG statement can use only one relocatable label)
10	SYM OVF	Too many symbols have been defined (eliminate symbols or divide program)
11	BAD NUM	Numeric element not valid (properly define item in label or address field)
12	NO END	No END assembler directive found (put in END card before end of file)
13	LN > 64	Source line is too long (comment, label, and op code greater than 64 characters)
14	IMP R/D	A REF or DEF symbol used improperly (REF symbol defined inside and outside the program, DEF symbol not defined in the program)
15	X RF USE	A REF symbol appeared invalidly in an unrelocatable expression
16	IXB ERR	Address mode error (improper use of IXB field) (Table 3-1 in ALMI manual)
17	OPD ERR	No such format number (OPD format numbers 0 to 8 explained in paragraph 2-4.17)
18	ADR MODE	Illegal addressing mode (improperly written address)

APPENDIX H

980 OPERATING PROCEDURE

START

- 1) Turn on 980A and all peripherals. Connect peripherals to "ON LINE" position.
- 2) With the MODE switch in HALT position, set RESET switch (↓) on 980A front panel.

If battery pack is installed skip to step (8); otherwise, load IDL (CE00) as follows:

- 3) Set 000F on switches and ENTER into PC: PC (↑)
- 4) Set MODE switch to RUN position (↑).
- 5) Set LOAD switch up (↑).
- 6) Select any arbitrary memory location, and ENTER address into MA(↑).
- 7) Display contents of that address MD (↓) – should be CE00.

Load bootstrap and loader: (Assuming cassette input)

- 8) Set 0006 on switches and enter into PC: PC (↑)
- 9) Ready ASR733 loader cassette (REWIND-LOAD) to READY
- 10) With mode switch set to RUN, set LOAD switch: LOAD (↑). This should result in the reading of the 733 loader.
- 11) Put memory limit in E (use 7FFF if machine has 32k memory).
- 12) Check display of A for 0240.

Load Operating System:

- 13) Ready O. S. cassette
- 14) With machine in RUN mode, set START (↑).

ASR733 responds: *READY*

ASSEMBLY UNDER THE OPERATING SYSTEM

For assembly, object on CS2:

- 1) SAPG in CR
- 2) // ASSIGN, 4, KEY. C/R* control
- 3) // ASSIGN, 5, CR. C/R source
- 4) // ASSIGN, 6, LP. C/R listing
- 5) // ASSIGN, 7, CS2 C/R object
- 8) // ASSIGN, 10, DUM. C/R
- 9) // EXECUTE, CR. C/R

*C/R means Carriage Return

Machine response: Reads in assembler

- 10) Source in CR.

Machine response: READY SOURCE, HIT CR

- 11) Hit C/R

- 12) Pass source thru again for second pass

If no /* card at end of source deck on second pass, no entry point written on cassette and no return to supervisor.

RUN

- 1) Ready data deck in card reader (if input data required)

- 2) // EXECUTE, CS2. C/R

To execute a program already loaded: // EXECUTE. C/R

APPENDIX I
ASCII CHARACTERS BY NUMERICAL SEQUENCE

0000 000	00	Null	0100 000	20	Space	1000 000	40	@	1100 000	60	.
0000 001	01	Start reading	0100 001	21	!	1000 001	41	A	1100 001	61	a
0000 010	02	Start text	0100 010	22	"	1000 010	42	B	1100 010	62	b
0000 011	03	End text	0100 011	23	#	1000 011	43	C	1100 011	63	c
0000 100	04	End transmission	0100 100	24	\$	1000 100	44	D	1100 100	64	d
0000 101	05	Enquiry	0100 101	25	%	1000 101	45	E	1100 101	65	e
0000 110	06	Acknowledge	0100 110	26	&	1000 110	46	F	1100 110	66	f
0000 111	07	Bell	0100 111	27	'	1000 111	47	G	1100 111	67	g
0001 000	08	Backspace	0101 000	28	(1001 000	48	H	1101 000	68	h
0001 001	09	Horizontal Tab	0101 001	29)	1001 001	49	I	1101 001	69	i
0001 010	0A	Line Feed	0101 010	2A	*	1001 010	4A	J	1101 010	6A	j
0001 011	0B	Vertical tab	0101 011	2B	+	1001 011	4B	K	1101 011	6B	k
0001 100	0C	Form feed	0101 100	2C	,	1001 100	4C	L	1101 100	6C	l
0001 101	0D	Carriage return	0101 101	2D	-	1001 101	4D	M	1101 101	6D	m
0001 110	0E	Shift out	0101 110	2E	.	1001 110	4E	N	1101 110	6E	n
0001 111	0F	Shift in	0101 111	2F	/	1001 111	4F	O	1101 111	6F	o
0010 000	10	Data link escape	0110 000	30	0	1010 000	50	P	1110 000	70	p
0010 001	11	Device control 1	0110 001	31	1	1010 001	51	Q	1110 001	71	q
0010 010	12	Device control 2	0110 010	32	2	1010 010	52	R	1110 010	72	r
0010 011	13	Device control 3	0110 011	33	3	1010 011	53	S	1110 011	73	s
0010 100	14	Device control 4	0110 100	34	4	1010 100	54	T	1110 100	74	t
0010 101	15	Negative Acknowledge	0110 101	35	5	1010 101	55	U	1110 101	75	u
0010 110	16	Synchronous idle	0110 110	36	6	1010 110	56	V	1110 110	76	v
0010 111	17	End transmission block	0110 111	37	7	1010 111	57	W	1110 111	77	w
0001 000	18	Cancel	0111 000	38	8	1011 000	58	X	1111 000	78	x
0011 001	19	End Medium	0111 001	39	9	1011 001	59	Y	1111 001	79	y
0011 010	1A	Substitute	0111 010	3A	:	1011 010	5A	Z	1111 010	7A	z
0011 011	1B	Escape	0111 011	3B	:	1011 011	5B	[1111 011	7B	
0010 000	1C	File separator	0111 100	3C	<	1011 100	5C		1111 100	7C	
0011 101	1D	Group separator	0111 101	3D	=	1011 101	5D]	1111 101	7D	
0011 110	1E	Record separator	0111 110	3E	>	1011 110	5E	or ↑	1111 110	7E	~
0011 111	1F	Unit separator	0111 111	3F	?	1011 111	5F	-or ←	1111 111	7F	Delete

In the binary representation, the most significant bit is on the left, the least significant is on the right, and the space indicates the position of the sprocket hole when the characters are punched into paper tape. The hexadecimal representation includes eight bits with the most significant bit set to zero.

NOTE: IN MEMORY, THE MOST SIGNIFICANT BIT IS UNDEFINED AND EITHER A ZERO OR A ONE IS ACCEPTABLE.

APPENDIX J

OPERATION CODES – NUMERICAL ORDER

REGISTER-MEMORY INSTRUCTIONS

<u>Hexadecimal Code for Address Mode Specified</u>								<u>Mnemonic</u>	<u>Name</u>
<u>P</u>	<u>B</u>	<u>PX</u>	<u>BX</u>	<u>PI</u>	<u>BI</u>	<u>PIX</u>	<u>BIX</u>		
0000	0100	0200	0300	0400	0500	0600	0700	LDA	Load Register A
0800	0900	0A00	0B00	0C00	0D00	0E00	0F00	LDE	Load Register E
1000	1100	1200	1300	1400	1500	1600	1700	LDX	Load Register X
1800	1900	1A00	1B00	1C00	1D00	1E00	1F00	LDM	Load Register M
2000	2100	2200	2300	2400	2500	2600	2700	ADD	Add to Register A
2800	2900	2A00	2B00	2C00	2D00	2E00	2F00	SUB	Subtract from Register A
3000	3100	3200	3300	3400	3500	3600	3700	IOR	Inclusive OR with A Register
3800	3900	3A00	3B00	3C00	3D00	3E00	3F00	AND	Logical AND with A Register
4000	4100	4200	4300	4400	4500	4600	4700	BIX	Branch or Incremented Index
4800	4900	4A00	4B00	4C00	4D00	4E00	4F00	DMT	Decrement Memory and Test
5000	5100	5200	5300	5400	5500	5600	5700	IMO	Increment Memory by One
5800	5900	5A00	5B00	5C00	5D00	5E00	5F00	DIV	Divide
6000	6100	6200	6300	6400	6500	6600	6700	CPL	Compare Logical to Register A
6800	6900	6A00	6B00	6C00	6D00	6E00	6F00	CPA	Compare Algebraic to Register A
7000	7100	7200	7300	7400	7500	7600	7700	BRL	Branch and Link
7800	7900	7A00	7B00	7C00	7D00	7E00	7F00	BRU	Branch Unconditional
8000	8100	8200	8300	8400	8500	8600	8700	STA	Store Register A
8800	8900	8A00	8B00	8C00	8D00	8E00	8F00	STE	Store Register E
9000	9100	9200	9300	9400	9500	9600	9700	STX	Store Register X
9800	9900	9A00	9B00	9C00	9D00	9E00	9F00	MPY	Multiply
A000	A100	A200	A300	A400	A500	A600	A700	DST	Double Length Store
A800	A900	AA00	AB00	AC00	AD00	AE00	AF00	DSB	Double Length Subtract
B000	B100	B200	B300	B400	B500	B600	B700	DLD	Double Length Load
B800	B900	BA00	BB00	BC00	BD00	BE00	BF00	DAD	Double Length Add

APPENDIX J (CON'T)
REGISTER SHIFT INSTRUCTIONS

<u>Hexadecimal</u> <u>Code</u>	<u>Mnemonic</u>	<u>Name</u>
C800	ARA	Arithmetic Right Shift Register A
C820	ARD	Arithmetic Right Shift Double Length
C840	LRA	Logical Right Shift Register A
C860	LRD	Logical Right Shift Double
C880	ALA	Arithmetic Left Shift Register A
C8A0	LAD	Arithmetic Left Shift Double
C8C0	LLA	Logical Left Shift Double
C900	RTO	Right Test for Ones
C940	RTZ	Right Test for Zeros
C980	LTO	Left Test for Ones
C9C0	LTZ	Left Test for Zeros
CA00	CRA	Circular Right Shift Register A
CA20	CRE	Circular Right Shift Register E
CA40	CRX	Circular Right Shift Register X
CA60	CRM	Circular Right Shift Register M
CB20	CRS	Circular Right Shift Register S
CB40	CRL	Circular Right Shift Register L
CB60	CRB	Circular Right Shift Register B
CB80	CLD	Circular Left Shift Double
CBC0	CRD	Circular Right Shift Double

APPENDIX J (CON'T)
REGISTER TO REGISTER INSTRUCTIONS

<u>Hexadecimal</u> Code	<u>Mnemonic</u>	<u>Name</u>
C000	RSU	Register Subtract
C080	RAD	Register Add
C100	RCO	Register Complement
C200	RIV	Register Invert
C280	REO	Register Exclusive OR
C300	RIN	Register Increment
C400	RCA	Register Compare Algebraic
C480	ROR	Register Inclusive OR
C500	RMO	Register Move
C600	RCL	Register Compare Logical
C680	RAN	Register AND
C700	RDE	Register Decrement
C780	REX	Register Exchange

SKIP INSTRUCTIONS

CC00	SZE	Skip if Zero
CC10	SSE	Skip if Sense Switch Equal
CC20	SOO	Skip if Ones
CC40	SOD	Skip if Odd
CC60	SMI	Skip if Minus
CC80	SNZ	Skip if Not Zero
CC90	SSN	Skip if Sense Switch Not Equal
CCA0	SNO	Skip if Not ones
CCC0	SEV	Skip if Even
CCE0	SPL	Skip if Plus
CD00	SLT	Skip if Less Than
CD20	SEQ	Skip if Equal
CD40	SGT	Skip if Greater Than
CD60	SOV	Skip if Overflow
CD80	SGE	Skip if Greater Than or Equal
CDA0	SNE	Skip if Not Equal
CDC0	SLE	Skip if Less Than or Equal
CDE0	SNV	Skip if No Overflow
CF60	SOC	Skip on Carry
CFE0	SNC	Skip on No Carry

APPENDIX J (CON'T)
MISCELLANEOUS

<u>Hexadecimal</u> <u>Code</u>	<u>Mnemonic</u>	<u>Name</u>
CA9F	NRM	Normalize
CE00	IDL	Idle
D800	RDS	Read Direct Single
D820	WDS	Write Direct Single
D880	LSB	Load Status Block
D890	LSR	Load Status Block and Reset Interrupt
D8A0	LRF	Load Register File
D8C0	SSB	Store Status Block
D8E0	SRF	Store Register File
D900	ATI	Automatic Transfer Instruction
DB00	TABZ	Test Register A Bit for Zero
DB10	TABO	Test Register A Bit for One
DB20	TMBZ	Test Memory Bit for Zero
DB30	TMBO	Test Memory Bit for One
DB40	SABZ	Set Register A Bit to Zero
DB50	SABO	Set Register A Bit to One
DB60	SMBA	Set Memory Bit to Zero
DB70	SMBO	Set Memory Bit to One
DD00	API	Auxiliary Processor Initiate
DF00	MVC	Move Character String
DF80	CLC	Compare Logical Character String

APPENDIX K

LOGICAL UNIT INPUT/OUTPUT FUNCTIONS

<u>Operation Code</u>	<u>Operation</u>	<u>Function Performed by Logical Unit</u>
00	Read ASCII	One ASCII record is read from the logical unit specified in the PRB, and the data is stored in memory (two characters per word) at the buffer address specified. The number of characters input is placed in PRB Word 2.
01	Read object	One object record is read from the logical unit specified in the PRB, and the data is stored in memory at the buffer address specified in the PRB. The number of characters input is placed in PRB Word 2:
02	Write ASCII	The data in memory at the address specified in the PRB is transferred to the logical unit specified. The number of characters transferred is specified in the PRB, and the maximum number that can be transferred is the number for which the logical unit was opened.
03	Write object	The data in memory is transferred to the logical unit as specified in the PRB. In particular, the number of characters transferred is as specified even though the standard object record length is 64.
04	Rewind	This command is ignored and bit 3 of PRB word 0 is set to one, unless the physical device can be rewound under program control. After a successful rewind, the unit is positioned to read or write the first data record.
05	Backspace	The logical unit is backspaced one record.
06	Forward Space	The logical unit is spaced forward one record.
07	Open	The maximum record size is specified in the PRB. The logical and physical unit is initialized as required. The dictionary for a named disc file is brought into memory.

<u>Operation Code</u>	<u>Operation</u>	<u>Function Performed by Logical Unit</u>
08	Open-rewind	A logical unit open is performed, followed by a logical unit re-wind.
09	Close	Logical unit and physical device termination procedures are performed. Dictionary information is transferred from memory to the applicable disc.
10	Close write end of file	The close process is performed and an end of file record is written. An end of file consists of one record in which the first two characters are /* (except for magnetic tape in which hardware recognizes an end of file).
11	Unload	Magnetic tape units are rewound and placed off — line; otherwise, the command is ignored.
12	Read dictionary	The dictionary of the disc volume for the logical unit specified in the PRB is transferred into memory at the address specified. The dictionary is 5504 words long.
13-16		These operation codes are used for transfer of information between the file management program and the operator communication program.
17	Read 733 status	Read status from the ASR733 status word returned in first word of the buffer address as specified in the PRB:

Status Word Bit

10	Printer Ready
11	Record Ready
13	Cassette on Clear Leader
14	Playback Error
15	Playback Ready

INTRODUCTION TO THE 980 SERIES

TI-MIX MEMBER'S CRITIQUE

To make this manual more useful to you, we will appreciate your comments and recommendations on any improvements to this manual you feel are needed. After using this manual, please take the first opportunity to complete this questionnaire and return it, postpaid, to the TI-MIX staff where your comments will be given every consideration.

MANUAL ORGANIZATION

Were the manual sections well organized?

Yes, No Comment _____

Are there enough examples

Yes No Comment _____

GRAPHICS

What is the quality of the illustrations?

Excellent Satisfactory Poor

Are there enough illustrations throughout the manual?

Yes No Comment _____

Are the tables clear and easy to follow?

Yes No Comment _____

TEXT

What is the quality of the technical writing?

Excellent Adequate Poor

If there are particular paragraphs, instructions, etc., you feel need clarification or rewriting, please identify them and add your comments. _____

GENERAL COMMENTS _____

Respondent _____ Title _____

Company _____

Address _____

City/State/Zip _____

NO POSTAGE NECESSARY IF MAILED IN U.S.A.

Please fold, tape, and mail

CUT ALONG LINE

FIRST CLASS
Permit No. 6189
Houston, Texas

BUSINESS REPLY MAIL

No postage necessary if mailed in the United States

Postage will be paid by

TEXAS INSTRUMENTS INCORPORATED

DIGITAL SYSTEMS DIVISION

P.O. BOX 1444 HOUSTON, TEXAS 77001

ATTENTION: TI-MIX
M/S 784