Scelbi "8080" Software Gourmet Guide & Cook Book

SCELBI COMPUTER CONSULTING INC.

SCELBI '8080' SOFTWARE GOURMET GUIDE AND COOK BOOK

AUTHOR:

ROBERT FINDLEY

Scelbi Computer Consulting, Inc.
1322 Rear - Boston Post Road
Milford, CT 06460

IMPORTANT NOTICE

The information in this manual has been carefully reviewed and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies or for the success or failure of various applications to which the information contained herein might be applied.

ACKNOWLEDGEMENT

The author wishes to thank the following members of the staff at Scelbi Computer Consulting, Inc. for their dedicated assistance in the preparation of this book.

Miss Gabrielle Tingley
Raymond Edwards

PRINTING: 9 8 7 6 5 4 3 2

SCELBI '8080' SOFTWARE GOURMET GUIDE
AND COOK BOOK

****
TABLE OF CONTENTS
****

INTRODUCTION

Have you tried cooking up a program lately on your 8080 microcomputer, and you just can't seem to get the right mixture of instructions? Or did that math recipe your friend gave you turn out to have too many bugs in it, and leave a sour taste in your mouth? Don't toss your computer in the sink and grind those bad listings up in the garbage disposal. Here's a book that will help take you from a novice that burns the bits to a gourmet chef that can make the sweetest APPLEcations program pie imaginable.

Before throwing together your favorite dish, a thorough knowledge of the basic ingredients, namely the 8080 instruction set, is essential. Every chef that's worth his salt knows exactly what each ingredient will do for him. Begin creating your masterpiece by mixing in a little of this routine and a little of that routine. Spice up the program with a few of your own special application routines, and before baking, add a personal touch by folding in the input/output driver routines for the peripherals in your system. Bake thoroughly with your assembler and there you have it! Your programming masterpiece, ready to feed into your computer's memory for hours of tasty enjoyment.

Is your taste for math routines? Or manipulating data tables and character strings? Or maybe you wish to do some real time programming. Or set up your system to operate the peripherals under interrupt control. Whatever your requirements may be, there is certain to be some ideas, techniques, and routines in this book to aid you in programming for your specific application.

THE 8080 CPU INSTRUCTION SET

The 8080 CPU has quite a comprehensive instruction set which consists of 78 basic instructions. When the possible permutations are considered, the total comes to 244 instructions.

The instruction set allows the programmer to direct the operation of the CPU to transfer and perform mathematical and logical operations on the data stored in the various memory and register elements that make up the computer. These software related elements are the memory, the program counter, the accumulator register, six general registers, the stack pointer register, and the input and output ports. As the instructions are executed by the CPU, the results of certain operations are monitored and indicated by the CPU status "flags."

The memory is the program storage element for the CPU. When a program is to be executed, its instructions must first be stored in the proper locations in memory. The memory is also used to store data which the program will require during its execution. For the 8080, up to 65,536 eight bit bytes of memory can be directly accessed by the CPU.

The program counter is a sixteen bit register used to control the flow of the program from one instruction to the next. When a program is started, the program counter is set to the address in memory of the first instruction to be executed. Unless directed otherwise by the instruction just executed, the program counter will automatically increment to the address of the next instruction in sequence and execute it. When an instruction is executed that directs the program to an address other than the next sequential instruction, the new address is placed in the program counter and program execution is continued with the instruction at the new address.

The next seven CPU registers to be described have been given arbitrary symbols so that they may be referred to in a common language. The accumulator register will be designated by the symbol A. Four of the general registers will be referred to as the B, C, D, and E registers. The remaining two registers, which are used as the primary

memory pointer for the majority of the memory reference instructions, will be referred to as the H register (for the high portion of the memory address), and the L register (for the low portion of the address). Aside from their special function as the memory pointer, the H and L registers exhibit the same characteristics as the other four general registers.

The accumulator is an eight bit register. It is the work horse of the 8080 in that all major mathematical and logical functions can only be performed in the accumulator. The accumulator also performs all the functions of the six individual registers. It is, however, the only register used to transfer data between the computer and an external device via the input/output ports.

The six general registers are, like the accumulator, eight bits in length. They are used individually to store data, transfer data among themselves or with the accumulator and memory, and can perform certain other operations, such as increment and decrement their contents. The general registers can also be grouped in pairs to form pointers to memory for loading and storing from the accumulator, and for performing various stack operations. The registers are grouped as registers B and C, D and E, and, of course, H and L. When used as memory pointers, the high address is designated in registers C, E, B, D, and H, and the low address is designated in registers C, E, and L for each of the respective pairings.

There is also a special memory pointer register used to designate a section of memory for storing the return addresses of subroutines and for temporary storage of register contents. This pointer is the stack pointer. The stack pointer is a sixteen bit register that may point to any location in memory that is to be used for the program stack. The stack refers to the section of memory in which the subroutine return addresses are automatically stored by the CPU and register contents are stored as required by program instructions.

A major function of almost every program is to receive or transmit data between the computer and one or more peripheral devices. With the 8080, this is accomplished with the use of input and output ports. There can be up to 256 input and output ports in an 8080 system. Each input port and output port provides eight parallel data

lines for this communication. The accumulator acts as the gate for the I/O functions, since all data from an input port is received in the accumulator, and all data to be output to an output port must be loaded into the accumulator before the output occurs.

The CPU also has several flip-flops which shall be referred to as flags. These flip-flops are set up as the result of certain operations and are important because they can be tested by many of the instructions, and the instruction's operation may vary as a consequence of the flag's particular status at the time it is tested. There are five basic flags that will be designated as follows:

The "C" flag refers to the carry bit status. The carry bit is a '1' unit register that is used to indicate when the accumulator overflows and underflows. This bit can also be set to a known condition by certain types of instructions. This is important to remember when developing a program because quite often a program will have a long string of instructions which do not utilize the carry bit or care about its status, but which will be causing the carry bit to change its status from time to time. Thus, when one prepares to do a series of operations that will rely on the carry bit, one often desires to set the carry bit to a known state.

The "AC" flag refers to the auxiliary carry bit status. The auxiliary carry is a '1' unit register that is used to indicate when an overflow or underflow from bit 3 occurs. This bit is affected by the addition, subtraction, increment, decrement, and compare instructions. Unlike the other status bits, this cannot be tested by a program. Its only function is in the operation of the decimal adjust accumulator instruction (mnemonic DAA), which adjusts the accumulator from its binary contents to two "BCD" digits, in a manner to be defined later in this chapter.

The "Z" for zero flag refers to a '1' unit register that when desired will indicate whether the value of

the accumulator is exactly equal to zero. In addition, immediately after an increment or decrement of a single register or memory location, this flag will also indicate whether the increment or decrement caused that particular register to go to zero.

The "S" for sign flag refers to a '1' unit register that indicates whether the value in the accumulator is a positive or negative value (based on two's complement nomenclature). Essentially, this flag monitors the most significant bit in the accumulator and is set when it is a one.

The "P" flag refers to the last flag in the group which is for indicating when the accumulator contains a value that has even parity. Parity is useful for a number of reasons and is usually used in conjunction with testing for error conditions on words of data, particularly when inputting data from external sources. Even parity occurs when the number of bits that are a '1' in the accumulator (out of the eight possible) is an even value; i.e., 2, 4, 6, or 8, regardless of their order in the accumulator register.

It is important to note that the Z, S and P flags (as well as the previously mentioned C flag) can all be set to known states by certain instructions. It is also important to note that some instructions do not result in the flags being set so that if the programmer desires to have the program make decisions based on the status of flags, the programmer should ensure that the proper instruction, or sequence of instructions is utilized. It is particularly important to note that "load register" instructions do not by themselves set the flags. Since it is often desirable to obtain a data word (i.e. load it into the accumulator) and test its status for such parameters as whether or not the value is zero or a negative number, etc., the programmer must remember to follow a load instruction by a logical instruction (such as the NDA - "AND the accumulator") in order to set the flags before using an instruction that is conditional in regards to the flag status.

The description of the various types of instructions available with an 8080 CPU unit that follows will provide the machine language code for the instruction given as three octal digits, two hexadecimal digits, and a mnemonic name suitable for writing programs in symbolic type language, which is usually easier than trying to remember machine codes! It may be noted that the symbolic language used throughout this book is a combination of the original mnemonics for the instruction set of the 8008, which is a subset of the 8080 instruction set, plus a version of the mnemonics presented by Intel Corporation for the additional instructions of the 8080. For those readers that may be familiar with the mnemonics as presented by Intel Corporation, Appendix A lists the mnemonics used here along with the equivalent Intel mnemonics. It may be noted that for most instructions, the difference is very slight. If the programmer is not already aware of it, the use of mnemonics facilitates working with an "assembler" program when it is desired to develop relatively large and complex programs. Thus, the programmer is urged to concentrate on learning the mnemonics for the instructions and not waste time memorizing the machine codes. After a program has been written using the mnemonic codes, the programmer can always use a look up table to convert to the machine code if an assembler program is not available. It's a lot easier technique (and subject to less error) than trying to memorize the 244 digital combinations that make up the machine code instruction set!

The machine code for the instructions is presented in both three digit octal and two digit hexadecimal notation. Both formats are provided for those readers that may be familiar with only one or the other. As will be seen with the presentation of the instructions, the three digit octal format allows easier recognition of the type of instruction and the registers involved than does the two digit hexadecimal representation. It may also be less confusing for the novice to deal with numbers from zero to seven, rather than tacking the first six letters of the alphabet onto the numbering system to accommodate the additional values in the hexadecimal notation. Appendix B provides a conversion table from octal to hexadecimal for the octal values from 000 to 377 (00 to FF hexadecimal).

The programmer must also be aware that in this machine some instructions require more than one byte in memory. The "immediate"

type commands and input and output instructions require two consecutive locations in memory. The "jump" and "call" instructions and several other instructions that designate a specific memory location require three bytes in memory. The number of bytes required for each instruction will be indicated in the description when presented.

The first group of instructions to be presented are those that are used to load data from one CPU register to another, or from a CPU register to a byte in memory, or vice-versa. This group of instructions requires just one byte of memory. It is important to note that none of the instructions in this group affect the flags.

LOAD DATA FROM ONE CPU REGISTER
TO ANOTHER CPU REGISTER

MNEMONIC	OCTAL	HEXIDECIMAL
LAA	177	7F
LBA	107	47
.	.	.
.	.	.
LAB	170	78

The load register group of instructions allows the programmer to move the contents of one CPU register into another CPU register. The contents of the originating (from) register are not changed. The contents of the destination (to) register become the same as the originating register. Any CPU register can be loaded into any CPU register. Note that, for instance, loading register A into register A is essentially a NOP (no operation) command. When using mnemonics, the load symbol is the letter L followed by the "TO" register and then the "FROM" register. The mnemonic LBA means that the contents of register A (the accumulator) are to be loaded into register B. The mnemonic LAB states that register B is to have its contents loaded into register A. It can be seen that this basic instruction has many variations. The machine language coding for this instruction is in the same format as the mnemonic code except that the letters

used to represent the registers are replaced by numbers that the machine can use. Using octal code, the seven CPU registers and memory references are coded as follows:

REG A = 7
REG B = 0
REG C = 1
REG D = 2
REG E = 3
REG H = 4
REG L = 5
MEM = 6

Also, since the machine can only utilize numbers, the octal number 1 in the most significant digit of a machine code signifies that the computer is to perform a "load" operation. Thus, in machine code, the instruction for loading register B with the contents of register A becomes 107. The use of hexadecimal code does not lend itself to this convenient method of instruction recognition since the code in the center octal digit is separated between the two hexadecimal digits. It is important to note that the load instructions do not affect any of the flags.

LOAD DATA FROM ANY CPU REGISTER
TO A LOCATION IN MEMORY

LMA	167	77
LMB	160	70
LMC	161	71
LMD	162	72
LME	163	73
LMH	164	74
LML	165	75

This instruction is very similar to the previous group of instructions except that now the contents of a CPU register will be loaded into a specified memory location. The memory location that will receive the contents of the particular CPU register is that whose address is specified by the contents of the CPU H and L registers at the

time that the instruction is executed. The H CPU register specifies the high portion of the address desired, and the L CPU register specifies the low portion of the address into which data from the selected CPU register is to be loaded. Note that there are seven different instructions in this group as any CPU register can have its contents loaded into any location in memory. This group of instructions does not affect any of the flags.

LOAD DATA FROM A MEMORY LOCATION
TO ANY CPU REGISTER

LAM	176	7E
LBM	106	46
LCM	116	4E
LDM	126	56
LEM	136	5E
LHM	146	66
LLM	156	6E

This group of instructions can be considered the opposite of the previous group. Now, the contents of the byte in memory whose address is specified by the H (for the high portion of the address) and L (low portion of the address) registers will be loaded into the CPU register specified by the instruction. Once again, this group of instructions has no affect on the status of the flags.

LOAD IMMEDIATE DATA INTO A CPU REGISTER

LAI	076	3E
LBI	006	06
LCI	016	0E
LDI	026	16
LEI	036	1E
LHI	046	26
LLI	056	2E

An "immediate" type of instruction requires two bytes in order to be completely specified. The first byte is the instruction itself, the

second byte, or "immediately following" byte, must contain the data upon which immediate action is taken. Thus, a load immediate instruction in this group means that the contents of the byte immediately following the instruction byte is to be loaded into the specified register. For example, a typical load immediate instruction would be: LAI 001. This would result in the value 001 being placed in the A register when the instruction was executed. It is important to remember that all immediate type instructions must be followed by a data byte. An instruction such as LDI alone would result in improper operation because the computer would assume the next byte contained data, and if the programmer had mistakenly left out the data, and in its place had another instruction, the computer would not realize the operator's mistake and hence, the program would be fouled up! Note too, that the load immediate group of instructions does not affect the flags.

LOAD IMMEDIATE DATA INTO A MEMORY LOCATION

LMI	066	36

This instruction is essentially the same as the load immediate into the CPU register group except that now, using the contents of the H and L registers as pointers to the desired address in memory, the contents of the immediately following byte will be placed in the memory location specified. This instruction does not affect the status of the CPU flags.

The above rather large group of load instructions permits the programmer to direct the computer to move data about. They are used to bring in data from memory where it can be operated on by the CPU, or to temporarily store intermediate results in a CPU register during complicated and extended calculations, and of course allow data, such as results, to be placed back into memory for long term storage. Since none of them will alter the contents of the four CPU flags, these instructions can be called upon to, for example, set up data, before instructions that may affect or utilize the flags' status are executed. The programmer will use instructions from this set frequently. The mnemonic names for the instructions are easy to remember as they are well ordered. The most important item to remember about the mnemonics is that the TO register is always in-

dicated first in the mnemonic, and then the FROM register. Thus LBA = load to register B from register A.

INCREMENT THE VALUE OF A CPU REGISTER BY 1

INA	074	3C
INB	004	04
INC	014	0C
IND	024	14
INE	034	1C
INH	044	24
INL	054	2C

This group of instructions allows the programmer to add 1 to the present value of any of the CPU registers. This instruction for incrementing the defined CPU registers is very valuable in a number of applications. For one thing, it is an easy way to have the L register successively point to a string of locations in memory. A feature that makes this type of instruction even more powerful is that the result of the incremented register will affect the Z, AC, S, and P flags. (It will not change the carry flag.) Thus, after a CPU register has been incremented by this instruction, one can utilize a flag test instruction (such as the jump and call instructions to be described later) to determine whether that particular register has a value of zero (Z flag), or if it is a negative number (S flag), or even parity (P flag). It is important to note that this group of instructions, and the decrement group (to be described next) are the only instructions that allow the S, Z, and P flags to be manipulated by operations that are not concerned with the accumulator (A) register.

INCREMENT THE CONTENTS OF A MEMORY LOCATION BY 1

INM	064	34

This instruction increments by one the contents of the memory location indicated by the memory pointer registers H and L. This instruction is very useful for example, for incrementing a counter stored in memory without first loading the counter into a register. The re-

sult of the increment operation is indicated by the Z, S, AC, and P flags, in the same manner as that described for the increment register instruction. The C flag is not affected by the operation.

DECREMENT THE VALUE OF A CPU REGISTER BY 1

DCA	075	3D
DCB	005	05
DCC	015	0D
DCD	025	15
DCE	035	1D
DCH	045	25
DCL	055	2D

The decrement group of instructions is similar to the increment group except that now the value 1 will be subtracted from the specified CPU register. This instruction will not affect the C flag, but it does affect the AC, Z, S, and P flags.

DECREMENT THE CONTENTS OF A MEMORY LOCATION BY 1

DCM	065	35

The contents of the memory location designated by the memory pointer registers H and L are decremented by one by the execution of this instruction. The Z, AC, S, and P flags are affected by the results of the decrement while the C flag is not affected.

ARITHMETIC INSTRUCTIONS USING THE ACCUMULATOR

The following group of instructions allows the programmer to direct the computer to perform arithmetic operations between other CPU registers and the accumulator, or between the contents of words in memory and the accumulator. All of the operations for the described addition, subtraction, and compare instructions affect the status of all the flags.

ADD THE CONTENTS OF A CPU REGISTER TO THE ACCUMULATOR

ADA	207	87
ADB	200	80
ADC	201	81
ADD	202	82
ADE	203	83
ADH	204	84
ADL	205	85

This group of instructions will simply add the present contents of the accumulator register to the present value of the specified CPU register and leave the result in the accumulator. The value of the specified register is unchanged except in the case of the ADA instruction. Note that the ADA instruction essentially allows the programmer to double the value of the accumulator (which is the A register)!

ADD THE CONTENTS OF A CPU REGISTER PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACA	217	8F
ACB	210	88
ACC	211	89
ACD	212	8A
ACE	213	8B
ACH	214	8C
ACL	215	8D

This group is identical to the previous group except that now the content of the carry flag is considered as an additional bit (MSB) in the specified CPU register and the combined value of the carry bit plus the contents of the specified CPU register are added to the value in the accumulator. The results are left in the accumulator. Again, with the exception of the ACA instruction, the contents of the specified CPU registers are left unchanged.

SUBTRACT THE CONTENTS OF A CPU REGISTER FROM THE ACCUMULATOR

SUA	227	97
SUB	220	90
SUC	221	91
SUD	222	92
SUE	223	93
SUH	224	94
SUL	225	95

This group of instructions will cause the present value of the specified CPU register to be subtracted from the value in the accumulator. The value of the specified register is not changed except in the case of the SUA instruction. (Note that the SUA instruction is a convenient instruction with which to clear the accumulator.)

SUBTRACT THE CONTENTS OF A CPU REGISTER AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBA	237	9F
SBB	230	98
SBC	231	99
SBD	232	9A
SBE	233	9B
SBH	234	9C
SBL	235	9D

This group is identical to the previous group except that now the content of the carry flag is considered as an additional bit (MSB) in the specified CPU register and the combined value of the carry bit plus the contents of the specified CPU register are subtracted from the value in the accumulator. The results are left in the accumulator. With the exception of the SBA instruction, the contents of the specified CPU register are left unchanged.

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A CPU REGISTER

CPA	277	BF
CPB	270	B8
CPC	271	B9
CPD	272	BA
CPE	273	BB
CPH	274	BC
CPL	275	BD

The compare group of instructions is a very powerful and somewhat unique set of instructions. They direct the computer to compare the contents of the accumulator against another register and to set the flags as a result of the comparing operation. It is essentially a subtraction operation with the value of the specified register being subtracted from the value of the accumulator, except that the value of the accumulator is not actually altered by the operation. However, the flags are set in the same manner as though an actual subtraction operation had occurred. Thus, by subsequently testing the status of the various flags after a compare instruction has been executed, the program can determine whether the compare operation resulted in a match or non-match, and in the case of a non-match, whether the compare register contained a value greater or less than that in the accumulator. This would be accomplished by testing the Z and C flags respectively utilizing a jump or call flag testing instruction (which will be described later).

ADDITION, SUBTRACTION, AND COMPARE INSTRUCTIONS THAT USE BYTES IN MEMORY AS OPERANDS

The five types of mathematical operations: add, add with carry, subtract, subtract with carry, and the compare, which have just been presented for performing the operations with the contents of the CPU registers, can all also be performed with bytes that are in memory. As with the load instructions with memory, the H and L registers must contain the address of the byte in memory that it is desired to add, subtract, or compare to the accumulator. The same conditions for the operations as was detailed when using the CPU registers

apply. Thus, for mathematical operations with a byte in memory, the following instructions are used:

ADD THE CONTENTS OF A MEMORY BYTE TO THE ACCUMULATOR

ADM	206	86

ADD THE CONTENTS OF A MEMORY BYTE PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACM	216	8E

SUBTRACT THE CONTENTS OF A MEMORY BYTE FROM THE ACCUMULATOR

SUM	226	96

SUBTRACT THE CONTENTS OF A MEMORY BYTE AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBM	236	9E

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A MEMORY BYTE

CPM	276	BE

IMMEDIATE TYPE ADDITIONS, SUBTRACTIONS, AND COMPARE INSTRUCTIONS

The five types of mathematical operations discussed can also be performed with the operand being the byte of data immediately after the instruction. This group of instructions is similar in format to the previously described "load immediate" instructions. The same conditions for the mathematic operations as discussed for the operations with the CPU registers apply.

ADD IMMEDIATE

ADI	306	C6

ADD WITH CARRY IMMEDIATE

ACI	316	CE

SUBTRACT IMMEDIATE

SUI	326	D6

SUBTRACT WITH CARRY IMMEDIATE

SBI	336	DE

COMPARE IMMEDIATE

CPI	376	FE

LOGICAL INSTRUCTIONS WITH THE ACCUMULATOR

There are several groups of instructions that allow Boolean logic operations to be performed between the contents of the CPU registers and the A or accumulator register, as well as between contents of locations in memory and the A register. In addition, there are logic immediate type instructions. The Boolean logic operations are valuable in a number of programming applications. The instruction set allows three basic Boolean operations to be performed. These are the logical AND, logical OR, and EXCLUSIVE OR operations. Each type of logic operation is performed on a bit-by-bit basis between the accumulator register and the CPU register or memory location specified by the instruction. A detailed explanation of each type of logic operation, and the appropriate instructions for each type is presented next. The logic instruction set is also valuable because all of them will cause the carry (C) flag to be reset to the '0' condition. This is important if one is going to perform a sequence of instructions that will eventually use the status of the C flag to arrive at a decision, as it allows the programmer to set the C flag to a known state at the start

of the sequence. All other flags are set in accordance with the result of the logic operation and hence, the group often has value when the programmer desires to determine the contents of a register that has just been loaded into the accumulator (since the load instructions do not affect the state of the flags).

THE BOOLEAN "AND" OPERATION AND INSTRUCTION SET

When the Boolean AND instruction is executed, each bit of the accumulator will be compared with the corresponding bit in the register or memory location specified by the instruction. As each bit is compared, a logic result will be placed in the accumulator for each bit comparison. The logic result is determined as follows. If both the bit in the accumulator and the bit in the register with which the operation is being performed are a '1,' the accumulator bit will be left as a '1.' For all other possible combinations (i.e., the accumulator bit = '0' and the other register's bit = '1,' or if the accumulator bit = '1' and the other register's bit = '0,' or if both the accumulator and the other register have the particular bit = '0'), then the accumulator bit will be set to '0.' An example will illustrate the logical AND operation:

INITIAL STATE OF THE ACCUMULATOR:	1 0 1 0 1 0 1 0
CONTENTS OF OPERAND REGISTER:	1 1 0 0 1 1 0 0
FINAL STATE OF THE ACCUMULATOR:	1 0 0 0 1 0 0 0

There are seven logical AND instructions that allow any CPU register to be used as the AND operand. They are as follows:

NDA	247	A7
NDB	240	A0
NDC	241	A1
NDD	242	A2
NDE	243	A3
NDH	244	A4
NDL	245	A5

The contents of the operand register are not altered by a logical AND instruction.

There is also a logical AND instruction that allows a byte in memory to be used as an operand. The address of the byte in memory that will be used is pointed to by the contents of the H and L CPU registers.

NDM	246	A6

And finally, there is also a logical AND IMMEDIATE type of instruction that will use the contents of the byte immediately following the instruction as the operand.

NDI	346	E6

The next group of Boolean logic instructions directs the computer to perform the logical OR operation on a bit-by-bit basis with the accumulator and the contents of a CPU register or a word in memory. The logical OR operation will result in the accumulator having a bit set to '1' if either that bit in the accumulator, or the corresponding bit in the operand register is a '1.' Since the case where both the accumulator bit and the operand bit is a '1' also satisfies the relationship, that condition will also result in the accumulator bit being a '1.' If neither register has a one in the bit position, then the accumulator bit remains '0.' An example illustrates the results of a logical OR operation:

INITIAL STATE OF THE ACCUMULATOR:	1 0 1 0 1 0 1 0
CONTENTS OF THE OPERAND REGISTER:	1 1 0 0 1 1 0 0
FINAL STATE OF THE ACCUMULATOR:	1 1 1 0 1 1 1 0

There are seven logical OR instructions that allow any CPU register to be used as the OR operand. They are:

ORA	267	B7
ORB	260	B0
ORC	261	B1
ORD	262	B2
ORE	263	B3
ORH	264	B4
ORL	265	B5

And, by using the H and L registers as pointers, one can also use a byte in memory as an OR operand:

ORM	266	B6

There is also the logical OR IMMEDIATE instruction:

ORI	366	F6

As with the logical AND group of instructions, the logical OR instruction does not alter the contents of the operand register.

The last group of Boolean logic instructions is a variation of the logic OR. The variation is termed the logical EXCLUSIVE OR. The EXCLUSIVE OR operation is similar to the OR except that when the corresponding bits in both the accumulator and the operand register are a '1,' the accumulator bit will be set to '0.' Thus, the accumulator bit will be a '1' after the operation only if just one of the registers (accumulator register or operand register) has a '1' in the bit position. (Again, the operation is performed on a bit-by-bit basis.) An example provides clarification:

INITIAL STATE OF THE ACCUMULATOR: 1 0 1 0 1 0 1 0

CONTENTS OF THE OPERAND REGISTER: 1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR: 0 1 1 0 0 1 1 0

The seven instructions that allow the CPU registers to be used as operands are:

XRA	257	AF
XRB	250	A8
XRC	251	A9
XRD	252	AA
XRE	253	AB
XRH	254	AC
XRL	255	AD

The instruction that uses registers H and L as pointers to a memory location is:

XRM	256	AE

And the EXCLUSIVE OR IMMEDIATE type instruction is:

XRI	356	EE

As is in the case of the logical OR operation, the operand register is not altered except for the special case when the XRA instruction is used. This instruction, which directs the computer to EXCLUSIVE OR the accumulator (CPU register A) with itself, will cause the operand register - since it is also the accumulator - to have its contents altered (unless it is zero at the time the instruction is issued). This is because, regardless of what value is in the accumulator, if it is EXCLUSIVE ORED with itself, the result will always be zero! The example illustrates:

ORIGINAL VALUE OF THE ACCUMULATOR: 1 0 1 0 1 0 1 0

EXCLUSIVE OR WITH ITSELF: 1 0 1 0 1 0 1 0

FINAL VALUE OF THE ACCUMULATOR: 0 0 0 0 0 0 0 0

This only occurs when the logical EXCLUSIVE OR is performed on the accumulator itself. It can be shown that the results of performing the logical OR or logical AND between the accumulator and itself will result in the original accumulator value being retained.

INPUT INSTRUCTIONS

In order to receive information from an external device, the computer must utilize a group of special signal lines. The 8080 CPU is designed to handle up to 256 groups (each group having eight signal lines) of input signals. A group of signals is accepted at the computer by what is referred to as an input port. The computer controls the operation of the input ports. Under program control, the computer can be directed to obtain the information that is on the group of lines coming in to any input port and bring it into the accumulator. Various types of external equipment, such as a keyboard, can be connected to the input port(s). When it is desired to have information obtained from a specific input port, an input instruction must be used. The input instruction simply identifies which input port is to be operated, and when executed, causes the signal levels on the selected input port to be brought into the A CPU register (accumulator).

INP 333 DB

The input instruction requires two bytes in memory to properly specify the operation. The first byte indicates the input operation and the second indicates the input port number to be accessed. The input number can be from 0 to 377 (octal) and is dependent on the hardware configuration of the peripherals in the system. It is important to note that the input instruction does not affect the condition of the flags. If it desired to test the data read into the accumulator from the input port, one of the logic instructions must be executed, such as an NDA instruction.

OUTPUT INSTRUCTIONS

In order to output information to an external device, the computer utilizes another group of signal lines which are referred to as output ports. The 8080 CPU is capable of servicing up to 256 output ports. Each output port actually consists of eight signal lines. An out-

put instruction causes the contents of the CPU A (accumulator) register to be transferred to the signal lines of the designated output port.

OUT	323	D3

The output instruction, like the input instruction, requires two bytes of memory to define its operation. The first byte indicates the output instruction, the second indicates the port number (0 - 377 octal) to receive the output data. The output instruction does not affect any of the flags.

THE HALT INSTRUCTION

There is one instruction for the computer's instruction set that directs the CPU to stop all operations and to remain in that state until an interrupt signal is received. In a typical 8080 system, an interrupt signal may be generated by an operator pressing a switch or by an external piece of equipment. This instruction is normally used when the programmer desires to have a program terminated, or when it is desired to have the machine wait for an operator to set up external conditions. The machine code instruction that is used for the halt command is:

HLT	166	76

The halt instruction is a single byte instruction and does not affect the status of the CPU flags.

THE NO-OPERATION INSTRUCTION

NOP	000	00

The no-operation, or NOP, instruction directs the computer to consume time by executing a machine cycle that effectively does nothing except advance the program counter to the next memory

address! None of the CPU flags are affected by the operation. The instruction is useful when creating time delays, or as a filler if patches to a program are required (or anticipated). While a specific code is reserved for the NOP, it might be pointed out that a number of other instructions in the 8080 instruction set will also serve the same purpose (such as LAA, LBB, etc.).

THE DISABLE INTERRUPT INSTRUCTION

DI	363	F3

This command instructs the CPU not to accept an interrupt signal from any device in the system capable of generating an interrupt. An interrupt is a signal sent from an external device, whether it be a peripheral or just a switch that is part of the computer's front panel, which, when received by the CPU, causes the following sequence of events. First, the computer completes execution of the current instruction. The next instruction that it executes is taken from the interrupt input lines which is a one byte RESTART instruction, provided by the external device, that is to be executed without incrementing the program counter. For the restart instruction, the current program counter is stored as the return address of a subroutine call and the interrupt is serviced by the routine at the restart location. When the interrupt service routine completes its operation, a return instruction will restore the program counter to the program instruction that would have been executed next, had the interrupt not occurred. The disable interrupt instruction would be used at times in a program when the operation to be performed affects data critical to the execution of the interrupt service routine. Interrupts are automatically disabled at the time an interrupt is received to avoid overlapping interrupts, possibly causing the program to miss an interrupt. The disable interrupt is a one byte instruction and does not affect the flags.

THE ENABLE INTERRUPT INSTRUCTION

EI	373	FB

The enable interrupt instruction allows the CPU to respond to an interrupt received from an external device, as previously described. This function is automatically actuated by the 8080 when power is applied. It is then entirely up to the program to control. After an interrupt is received, the enable interrupt instruction must be executed when the program is ready to receive another interrupt or no further interrupts will be recognized by the CPU. Also, if the program halts without the interrupts enabled, the computer must be reset before any execution can resume, since it requires an interrupt to get the CPU out of the halted state. Resetting may require the computer to be turned off, and then on, unless a reset function has been built into the computer. The enable interrupt instruction requires one byte of memory and does not affect the flags.

INSTRUCTIONS FOR ROTATING THE CONTENTS OF THE ACCUMULATOR

It is often desirable to be able to shift the contents of the accumulator either right or left. In a fixed length register, a simple shift operation would result in some information being lost because what was in the MSB or LSB (depending on which direction the shift occurred) would just be shifted right out of the register! Therefore, instead of just shifting the contents of a register, an operation termed "rotating" is utilized. Now, instead of just shifting a bit off the end of the register, the bit is brought around to the other end of the register. For instance, if the register is rotated to the right, the LSB (least significant bit) would be brought around to the position of the MSB (most significant bit) in the register which would have been vacated by the shifting of its original contents to the right. Or, in the case of a shift to the left, the MSB would be brought around to the position of the LSB.

Since the carry bit (carry or C flag) can be considered as an extension of the accumulator register, it is often desired that the carry bit be considered as part of the accumulator (the MSB) during a rotate operation. The instruction set for this machine allows two types of rotate instructions. One considers the carry bit to be part of the

accumulator register for the rotate operation, and the other type does not. In addition, each type of rotate can be done either to the right or to the left.

It should be noted that the rotate operations are particularly valuable when it is desired to multiply a number because shifting the contents of a register to the left is a quick way to multiply a binary number by powers of two, and shifting to the right provides the inverse operation.

ROTATING THE ACCUMULATOR LEFT

RLC	007	07

Rotating the accumulator left with the RLC instruction means the MSB of the accumulator will be brought around to the LSB position and all other bits are shifted one position to the left. While this instruction does not shift through the carry bit, the carry bit will be set by the status of the MSB of the accumulator at the start of the rotate operation. (This feature allows the programmer to determine what the MSB was prior to the shifting operation by testing the C flag after the rotate instruction has been executed.)

ROTATING THE ACCUMULATOR LEFT THROUGH THE CARRY BIT

RAL	027	17

The RAL instruction will cause the MSB of the accumulator to go into the carry bit. The initial value of the carry bit will be shifted around to the LSB of the accumulator. All other bits are shifted one position to the left.

ROTATING THE ACCUMULATOR RIGHT

RRC	017	0F

The RRC instruction is similar to the RLC instruction except that now the LSB of the accumulator is placed in the MSB of the accumulator and all other bits are shifted one position to the right. Also, the carry bit will be set to the initial value of the LSB of the accumulator at the start of the operation.

ROTATING THE ACCUMULATOR RIGHT THROUGH THE CARRY BIT

RAR	037	1F

Here, the LSB of the accumulator is brought around to the carry bit and the initial value of the carry bit is shifted to the MSB of the accumulator. All other bits are shifted one position to the right.

It should be noted that the C flag is the only flag that can be altered by a rotate instruction. All other flags remain unchanged.

LOAD IMMEDIATE DATA INTO A REGISTER PAIR

LXB	001	01
LXD	021	11
LXH	041	21

This command loads the register pair indicated with the contents of the next two bytes in memory. Thus, these instructions require three bytes of memory. Typically, the second and third bytes could contain a memory address or a double precision binary value. When the LXB is executed, for example, the contents of the N+1 location will be placed in the C register and the contents of the N+2 location will go in the B register (N is the location of the machine code for the first byte of the three byte LXB instruction). The other two instructions load E or L with the second byte of the instruction and D or H with the third, for the designated register pair. None of the flags are affected by this instruction.

LOAD IMMEDIATE DATA INTO THE STACK POINTER

LXS	061	31

This instruction is also a three byte instruction which loads the stack pointer with the contents of the next two sequential bytes of memory. The data loaded into the stack pointer is the address of memory to be used for the stack. The contents of the N+1 byte are loaded into the low address portion of the stack pointer and the N+2 byte is loaded into the high address portion. No flags are affected by the execution of this instruction.

STORE THE ACCUMULATOR DIRECTLY IN MEMORY

STA	062	32

The contents of the accumulator can be stored directly into the memory location indicated by this three byte instruction. This may be used to store information in a fixed memory location, such as within a specific data table set up by the program. The N+1 byte contains the low address and the N+2 byte contains the high address of the memory location that is to receive the data in the accumulator. It is important to note that this instruction only refers to the accumulator. It cannot be applied to the other registers. Also, its operation does not affect the flags.

LOAD THE ACCUMULATOR DIRECTLY FROM MEMORY

LTA	072	3A

This three byte instruction performs the opposite function of the previously defined STA instruction. It loads the accumulator with the contents of the memory location specified in the second and third bytes of the instruction. As is the standard setup, the N+1 byte contains the low address portion and the N+2 byte contains the high address portion of the location from which the accumulator is to be loaded. The CPU flags are not affected by this instruction.

COMPLEMENT THE ACCUMULATOR

CMA 057 2F

The contents of the accumulator are complemented. That is, the bits equal to '1' are changed to '0' and those equal to '0' are changed to '1.' The CPU flags are not affected by this instruction. This instruction is useful in forming the "two's complement of a binary number, by taking the complement of the accumulator and then incrementing the accumulator. The two's complement is the binary format for the negative value of a binary number. It is often used to add the negative value to a number rather than subtracting the positive value.

DECIMAL ADJUST THE ACCUMULATOR

DAA 047 27

The decimal adjust accumulator instruction is a one byte instruction that adjusts the contents of the accumulator to two binary-coded-decimal digits, one digit in the four least significant bits and one digit in the four most significant bits. This instruction is used following the addition of two pair of BCD digits to adjust the result to the proper BCD code. This instruction operates in the following manner.

The least significant half of the accumulator is checked for a BCD value of zero to nine, and the AC flag is checked for a '0' condition. If both of these conditions exist, this half is left as is. However, if this half is greater than nine, or the AC flag is '1,' the accumulator will be incremented by six, thereby adjusting the least significant half to the proper BCD code. The most significant half is then checked for a BCD value of zero through nine and the C flag is checked for a '0' condition. If both conditions are true, the process is complete. Otherwise, if either condition is not met, the most significant half of the accumulator is incremented by six, with the carry flag being set to a '1' if an overflow from the most significant half occurs. The Z, S, and P flags are also affected by this operation.

SET THE CARRY FLAG

STC	067	37

The condition of the carry flag is set to '1.' This instruction affects no other flag but the carry. It is often required in some mathematical and rotate operations to have the carry flag set to '1.' This is a one byte instruction.

COMPLEMENT THE CARRY FLAG

CMC	077	3F

The condition of the carry flag is reversed from its current state by this one byte instruction. If the carry is a '1,' it is changed to '0.' If the carry is a '0,' it is changed to '1.' No other flags are affected by this instruction.

POP FROM THE STACK INTO A REGISTER PAIR

POPB	301	C1
POPD	321	D1
POPH	341	E1

The contents of the stack indicated by the stack pointer are loaded into the low order register (namely, register C, E, or L) of the register pair referred to in the instruction; the contents indicated by the stack pointer + 1 are loaded into the high order register (register B, D, or L) of the named register pair. At the completion of the instruction, the stack pointer will be incremented by two from its initial value. The CPU flags will be unaffected. This instruction requires only one byte in memory to load two registers with the stack contents. It is often used at the completion of an interrupt routine to restore the contents of the registers before returning from an interrupt, or to recall an address or data value temporarily stored in the stack.

POP THE ACCUMULATOR AND STATUS FLAGS FROM THE STACK

POPS	361	F1

The accumulator is loaded with the contents of the stack as indicated by the stack pointer, and the status flags are set to the conditions indicated by the contents of the stack at the address of the stack pointer + 1. The bit definition of the byte that restores the status flags is as follows:

BIT 0 = CARRY FLAG
BIT 1 = ALWAYS 1
BIT 2 = PARITY FLAG
BIT 3 = ALWAYS 0
BIT 4 = AUXILIARY CARRY FLAG
BIT 5 = ALWAYS 0
BIT 6 = ZERO FLAG
BIT 7 = SIGN FLAG

At the completion of the operation, the stack pointer contains the address of the initial stack pointer + 2. This instruction affects all of the flags in that they will be set to the conditions as read from the stack. This instruction is also used at the completion of an interrupt routine to restore the accumulator and status flags before exiting the interrupt.

PUSH A REGISTER PAIR ONTO THE STACK

PUSB	305	C5
PUSD	325	D5
PUSH	345	E5

The contents of the register named in the instruction (register B, D, or H) are stored in the stack at the address of the current stack pointer less 1 and the other register of the pair is stored in the stack at the stack pointer less 2. The stack pointer is decremented by two from its initial value when the instruction is complete. None of the status flags are affected by the operation. These instructions are

often used to save the contents of the registers at the time an interrupt is received to allow their proper restoration at the completion of the interrrupt routine, and to temporarily save the registers in the stack for later recall by a program.

PUSH THE ACCUMULATOR AND STATUS FLAGS ONTO THE STACK

PUSS	365	F5

The status flags are stored in the stack at the address of the current stack pointer less 1 in the format described for the POPS instruction. The accumulator contents are stored in the stack at the current stack pointer less 2 address. The stack pointer is decremented by two and the CPU flags are unaffected by this instruction. This instruction is typically used at the beginning of an interrupt routine and when it is desired to temporarily save the status flags and the accumulator for later reference.

EXCHANGE THE CONTENTS OF REGISTERS H AND L WITH D AND E

XCHG	353	EB

The contents of register H are exchanged with register D and the contents of register L are exchanged with register E. This one byte instruction is a convenient means of changing the memory pointer in register pair H and L from one table area to another. The status flags are not affected.

EXCHANGE THE CONTENTS OF REGISTERS H AND L WITH THE STACK

XTHL	343	E3

The contents of register L are exchanged with the contents of the stack indicated by the stack pointer, and the contents of register H

with the stack contents at the stack pointer + 1 address. This instruction differs from the POPH instruction in that the data that is popped from the stack into H and L is replaced in the stack by the initial contents of registers H and L and the stack pointer is unchanged as a result of the operation. The status flags are not affected.

LOAD THE STACK POINTER FROM REGISTERS H AND L

SPHL	371	F9

This command loads the stack pointer with the address stored in registers H (containing the high portion of the address) and L (containing the low portion of the address). This one byte instruction does not affect any of the status flags.

DOUBLE ADD THE DESIGNATED REGISTER PAIR TO THE REGISTER PAIR H AND L

DADB	011	09
DADD	031	19
DADH	051	29

These one byte instructions add the contents of the designated register pair (B and C, D and E, or H and L) with the register pair H and L, and store the result of the addition in registers H and L. The addition is performed as a "double precision" operation in which the low order register (C, E, or L) is added to register L, and the high order register (B, D, or H) is added to register H. If the addition of the low order registers results in an overflow, the value one is added to the sum of the high order registers. This instruction is typically used to add a displacement to a memory address stored in registers H and L. None of the flags are affected.

DOUBLE ADD THE STACK POINTER TO THE REGISTER PAIR H AND L

DADS	071	39

The low portion of the address in the stack pointer is added to register L and the high portion is added to register H. If the addition of the low portion to register L results in an overflow, one is added to the result in register H. This instruction may be used to load the stack pointer into registers H and L by first loading registers H and L with all zeros and then performing the DADS instruction. The flags are not affected by this one byte instruction.

STORE THE ACCUMULATOR AT THE ADDRESS IN REGISTERS B AND C

STAB	002	02

This instruction uses the register pair B and C as the memory pointer for storing the contents of the accumulator in the designated memory location. It is important to note that this instruction only refers to storing the contents of the accumulator, and not any of the other registers, using B and C as the memory pointer. Also, this one byte instruction does not affect the flags.

STORE THE ACCUMULATOR AT THE ADDRESS IN REGISTERS D AND E

STAD	022	12

This one byte instruction operates the same as the STAB instruction except that the register pair D and E is used as the memory pointer. These instructions allow efficient use of the register pairs as memory pointers for transferring data from one area in memory to another.

LOAD THE ACCUMULATOR FROM THE ADDRESS IN REGISTERS B AND C

LDAB	012	0A

This instruction loads the accumulator with the contents of the

memory location designated in register pair B and C. The flags are not affected by the operation of this one byte instruction.

LOAD THE ACCUMULATOR FROM THE ADDRESS IN REGISTERS D AND E

LDAD	032	1A

The accumulator is loaded from the memory location indicated by the address in register pair D and E. The status flags are not affected by this one byte instruction.

INCREMENT THE DESIGNATED REGISTER PAIR

INXB	003	03
INXD	023	13
INXH	043	23

This command increments the register pair designated in the mnemonic as though they were single sixteen bit registers. The value in the low order register (C, E, or L) is incremented by one and if this register overflows as a result, the high order register (B, D, or H) will be incremented by one. This group of instructions makes it convenient to increment a memory pointer or double precision binary value held in one of the register pairs since it automatically carries an overflow from the least significant half over to the most significant half. These one byte instructions do not affect the CPU flags.

INCREMENT THE STACK POINTER

INXS	063	33

This one byte instruction increments the stack pointer by one. It is often used to move the stack pointer up in the stack to allow a program to access information held in the stack. None of the status flags are affected.

DECREMENT THE DESIGNATED REGISTER PAIR BY ONE

DCXB	013	0B
DCXD	033	1B
DCXH	053	2B

The designated register pair is decremented by one by decrementing the low order register (C, E, or L) first and if an underflow occurs, the high order register is decremented by one. These instructions provide a convenient means of decrementing memory pointers and double precision binary values stored in the register pairs. These one byte instructions do not affect the status flags.

DECREMENT THE STACK POINTER BY ONE

DCXS	073	3B

This instruction decrements the stack pointer by one, allowing the stack pointer to be moved to access data in the lower section of the stack. This one byte instruction does not affect the flags.

STORE REGISTERS H AND L IN MEMORY

SHLD	042	22

This three byte instruction stores the contents of register L in the memory location designated in the instruction, and register H in the memory location + 1. The second byte of the instruction contains the low portion of the memory address and the third byte contains the high portion of the memory address. This instruction allows the program to store a memory pointer, for example, in a predetermined location in memory. None of the CPU flags are affected by this instruction.

LOAD REGISTERS H AND L FROM MEMORY

LHLD	052	2A

The contents of the memory location designated in the instruction are loaded into register L, and the contents of the memory location + 1 are loaded into register H. The memory address in this three byte instruction is specified in the same format as that in the SHLD instruction. The CPU flags are not affected by this operation.

JUMP INSTRUCTIONS

The instructions discussed so far have all been sort of direct action instructions. The programmer arranges a sequence of these types of instructions in memory and when the program is started, the computer proceeds to execute the instructions in the order in which they are encountered. The computer automatically reads the contents of a memory location, executes the instruction it finds there, and then automatically increments a special address register called a "program counter" that will result in the machine reading the information contained in the next sequential memory location. However, it is often desirable to perform a series of instructions located in one section of memory, and then skip over a group of memory locations and start executing instructions in another section of memory. This action can be accomplished by a group of instructions that will cause a new address value to be placed in the program counter. This will cause the computer to go to a new section of memory and to continue executing instructions sequentially from the new memory location.

The jump instructions in this computer add considerable power to the machine's capabilities because there is a series of conditional jump instructions available. That is, the computer can be directed to test the status of a particular flag (C, Z, S, or P), and if the status of the flag is the desired one, then a jump will be performed. If it is not, the machine will continue to execute the next instruction in the current sequence. This capability provides a means for the computer to make decisions and to modify its operation as a function of the status of the various flags at the time that the program is being executed.

In a manner similar to immediate types of instructions, the jump instructions require more than one byte of memory. A jump instruc-

tion requires three bytes to be properly defined. The jump instruction itself is the first byte; the second byte must contain the low address portion of the address of the byte in memory that the program counter is to be set for (in other words, the new location from which the next instruction is to be taken); the third byte must contain the high address (page) of the memory address that the program counter will be set to, hence, the page or high order portion of the address that the computer will jump to to obtain its next instruction.

THE UNCONDITIONAL JUMP INSTRUCTION

JMP	303	C3

Remember, the jump instruction must be followed by two more bytes that contain the low, and then the high (page) portion of the address that the program is to jump to!

JUMP IF THE DESIGNATED FLAG IS TRUE (CONDITIONAL JUMP)

JTC	332	DA
JTZ	312	CA
JTS	372	FA
JTP	352	EA

As with the unconditional jump instruction, the conditional jump instructions must be followed by two bytes of information - the low portion, then the high portion of the address that program execution is to continue from if the jump is executed. The jump if true group of instructions will only jump to the designated address if the condition of the appropriate flag is true (logical '1'). Thus the JTC instruction states that if the carry flag (C) is a logical '1' (true), then the jump is to be executed. If it is a logical '0' (false), program execution is to continue with the next instruction in the current sequence of instructions. In a similar manner, the JTZ instruction states that if the zero flag is true, then the jump is to be performed. Otherwise, the next instruction in the present sequence is executed. Likewise for the JTS and JTP instructions.

JUMP IF THE DESIGNATED FLAG IS FALSE
(CONDITIONAL JUMP)

JFC	322	D2
JFZ	302	C2
JFS	362	F2
JFP	342	E2

As with all jump instructions, these instructions must be followed by the low address and then the high address of the memory location that program execution is to continue from if the jump is executed. This group of instructions is the opposite of the jump if the flag is true group. For instance, the JFC instruction commands the computer to test the status of the carry (C) flag. If the flag is false, i.e. a logic '0,' then the jump is to be performed. If it is true then program execution is to continue with the next instruction in the current sequence of instructions. The same procedure holds for the JFZ, JFS, and JFP instructions.

LOAD THE PROGRAM COUNTER WITH REGISTERS H AND L

PCHL	351	E9

The program counter is loaded with the address contained in register H (for the high portion) and register L (for the low portion). This instruction is actually a special form of a jump instruction, in which the program has formulated the address to jump to in the H and L registers and, rather than storing that address as the second and third bytes of a jump instruction to be executed, PCHL will effectively perform the same function in a single byte. The flags are not affected by this instruction.

SUBROUTINE CALLING INSTRUCTIONS

Quite often when a programmer is developing computer programs, the programmer will find that a particular algorithm (sequence of instructions for performing a function) can be used many times in different parts of the program. Rather than having to keep entering the

same sequence of instructions at different locations in memory (which would not only consume the time of the programmer but would also result in a lot of memory being used to perform one particular function), it is desirable to be able to put an often used sequence of commands in one location in memory. Then, whenever the particular algorithm is required by another part of the program, it would be convenient to jump to the section that contained the often used algorithm, perform the sequence of instructions, and then return back to the main part of the program. This is a standard practice in computer operations. The frequently used algorithm can be designated as a subroutine. A special set of instructions allows the programmer to "call" - in other words specify a special type of jump to - a subroutine. A second type of instruction is used to terminate a sequence of instructions that is to be considered a subroutine. This special terminator will cause the program operation to revert back to the next sequential location in memory following the instruction that called the subroutine. A great deal of computer power is provided by the instruction set in this machine for calling and returning from subroutines. This is because, in a manner similar to the conditional jump instructions, there are a number of conditional calling commands and a number of conditional return commands in the instruction set.

Like the jump instructions, the call instructions all require three bytes in order to be fully specified. The first byte is the call instruction itself. The next two bytes must contain the low and high portions of the starting address of the subroutine that is being called.

When a call instruction is executed, the CPU will save the current value of its program counter by storing it in the stack. The address in the program counter is stored in the stack in the same manner that a PUSH instruction stores a register pair. The high portion of the program counter is stored in the stack pointer less 1 location, and the low portion of the program counter is stored in the stack pointer less 2 location in the stack. The stack pointer is decremented by two as a result of a subroutine call.

The return instruction that terminates a subroutine only requires one byte. When the CPU encounters a return instruction, it causes the address stored in the stack to be popped off the stack into the

program counter, thus returning the execution of the program to the calling routine.

THE UNCONDITIONAL CALL INSTRUCTION

CAL	315	CD

This instruction, followed by two words containing the low and then the high order of the starting address of the subroutine that is to be executed, is an unconditional call. The subroutine will be executed regardless of the status of the flags. The next sequential address after the CAL instruction is saved in the stack.

THE UNCONDITIONAL RETURN INSTRUCTION

RET	311	C9

This instruction directs the CPU to unconditionally POP the next address from the stack and place it in the program counter. Program execution will continue with the instruction at this address.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

CTC	334	DC
CTZ	314	CC
CTS	374	FC
CTP	354	EC

In a manner similar to the conditional jump if true instructions, these instructions (which must all be followed by the low and high portions of the called subroutine's starting address) will only perform the call if the designated flag is in the true (logical '1') state. If the designated flag is false then the call instruction is ignored and program execution continues with the next sequential instruction.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

RTC	330	D8
RTZ	310	C8
RTS	370	F8
RTP	350	E8

These one byte instructions will cause a subroutine to be terminated only if the designated flag is in the logical '1' (true) state.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

CFC	324	D4
CFZ	304	C4
CFS	364	F4
CFP	344	E4

These instructions are the opposite of the previous group of calling commands. The subroutine is called only if the designated flag is in the false (logical '0') condition. Remember, these instructions must be followed by two bytes which contain the low and then the high part of the starting address of the subroutine that is to be executed if the designated flag is false. If the flag is true, the subroutine will not be called, and program operation will continue with the next instruction in the current sequence.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

RFC	320	D0
RFZ	300	C0
RFS	360	F0
RFP	340	E0

These one byte instructions will terminate a subroutine (pop the stack up one level) if the designated flag is false. Otherwise the instruction is ignored and program operation is continued with the next instruction in the subroutine.

THE SPECIAL RESTART SUBROUTINE CALL INSTRUCTIONS

There is a special purpose instruction available that effectively serves as a one byte subroutine call (remember that it normally requires three bytes to specify a subroutine call). This special instruction allows the programmer to call a subroutine that starts at any one of eight specially designated memory locations. The eight special memory locations are at locations: 000, 010, 020, 030, 040, 050, 060, and 070 on page zero. There are eight variations of the restart instruction - one for each of the above addresses. Thus, the one byte instruction can serve to call a subroutine at the specified starting location (instead of having two additional bytes to specify the starting address of the subroutine.) The main purpose of the restart instruction is to provide a one byte instruction that a device operating in the interrupt mode can use to direct the computer to the interrupt service routine. It is also convenient as a quick call to an often used subroutine, or as an easy way to call short starting routines for large programs - hence the name for the type of instruction. The eight restart instructions, along with the starting address of the subroutine that each will automatically call, is as follows:

INSTRUCTION (MNEMONIC)	MACHINE CODE	SUBROUTINE STARTING ADDRESS
RST 0	307 C7	000 000
RST 1	317 CF	000 010
RST 2	327 D7	000 020
RST 3	337 DF	000 030
RST 4	347 E7	000 040
RST 5	357 EF	000 050
RST 6	367 F7	000 060
RST 7	377 FF	000 070

INFORMATION ON INSTRUCTION EXECUTION TIMES

When programming for real-time applications, it is important to know how much time each type of instruction requires to be executed. With this information the programmer can develop timing

loops or determine with substantial accuracy how much time it takes to perform a particular series of instructions. This information is especially important when dealing with programs that control the operation of external devices that require events to occur at specific times.

Along with the list of mnemonics and machine codes, Appendix A provides the nominal instruction execution time for each instruction used in an 8080 system. The table shows the number of cycle states required by the instruction, followed by the nominal time required to perform the entire instruction. The final column indicates the number of times the memory is accessed for the execution of the instruction. Since each state executes in 0.5 microseconds, the total time required to perform the instruction as shown in the table is obtained by multiplying the number of states by 0.5 microseconds. (The times given in Appendix A assume the system contains static memory with access times of less than 0.5 microseconds. For slower memories, up to 1.0 microsecond may have to be added for each time an instruction accesses memory. In the case of dynamic memory, the actual execution time may vary randomly due to the refresh cycles required.) Knowing the exact time required by the CPU to execute each instruction allows algorithms to be programmed to have specific events occur at precisely timed intervals. This concept is discussed in greater detail in chapter three on programmed time delays.

USING THE 8080 STACK

When a subroutine is called, the return address must be saved somewhere for the computer to refer to when returning from the subroutine. For the 8080, the return address is stored in the stack. The stack is a table area set aside in a section of read/write memory. It is used to store the return addresses for the subroutine calls, and it is also a convenient place to temporarily save the contents of CPU registers. The location of the stack in memory is indicated by the contents of the stack pointer. The stack pointer is a register pair capable of addressing any of the 64K possible memory locations.

Manipulating the stack pointer and the contents of the stack is achieved with the use of a number of instructions specifically related to stack operations. These instructions, which have been defined in chapter one, are summarized in the following list. As the use of the stack is described, the application of these instructions to various programming techniques will be indicated.

MACHINE CODE	INSTRUCTION MNEMONIC	DESCRIPTION
061 XXX XXX	LXS ADDRESS	Load stack pointer immediate
371	SPHL	Load stack pointer with contents of register pair H and L
063	INXS	Increment the stack pointer
073	DCXS	Decrement the stack pointer
071	DADS	Add current stack pntr to register pair H & L. Result in H & L.
3X5	PUSX	Store contents of indicated register pair in stack
3X1	POPX	Load indicated reg. pair from stack
343	XTHL	Exchange contents of register pair H&L with last entry in stack. Stack pointer remains unchanged.

There are two instructions that are used to set up the stack pointer. The load the stack pointer immediate instruction, mnemonic LXS ADDR, sets the stack pointer to the two byte address in the immediate portion of the instruction. This allows a program to define the stack at a specific location in memory. The other instruction, mnemonic SPHL, loads the stack pointer from the contents of registers H and L. This instruction provides the program with a means of setting the stack pointer to an address formulated by the program. For example, a program could have several table areas assigned for storing or retrieving data. The selection of the proper table area to use might depend on several factors the program must consider. Once the table area to use is selected, the address could be placed in the stack pointer by first loading it in registers H and L and performing the SPHL instruction. The stack instructions may then be used to transfer data to and from the selected table area.

The stack stores its data, whether it be the return address of a subroutine or the contents of a register pair, by moving down to the lower numbered memory locations from the current address stored in the stack pointer. This requires the stack pointer to be initialized to the high address plus one of the area to be used for storing return addresses and data. The reason for setting the stack pointer to the high address plus one can be seen by examining the method used to store data on the stack.

When data is stored in the stack, the stack pointer is first decremented and then the first data byte is stored in the memory location. The stack pointer is then decremented again and the second data byte is stored in the stack in memory. By automatically decrementing the stack pointer in this manner, it is positioned to receive more data, to store the return address of another subroutine call, or to read back the data just stored when either a return or pop instruction is executed. The following example illustrates the method of storing the return address of a subroutine call in the stack. The return address to be stored is location 157 page 002.

	STACK POINTER	STACK CONTENTS	MEMORY ADDRESS OF STACK
BEFORE	001 000	000	000 375

CALL		000	000 376
		000	000 377
		000	001 000
AFTER	000 376	000	000 375
CALL		157	000 376 -
		002	000 377
		000	001 000

When data is read from the stack, by performing a return from a subroutine or popping the data off the stack into a register pair, the reverse procedure is followed. That is, as data is read from the stack, the stack pointer will be automatically incremented. First, the byte contained at the address of the current stack pointer is read from the stack. The stack pointer is then incremented to point to the next byte, which is then read from the stack. The process is completed by incrementing the stack pointer again to position it for the next stack operation, whether it be to read or write data in the stack.

When allocating memory space to be used for the stack to store return addresses and register contents, some rules must be followed. The first rule is that the stack must reside in "read and write" memory. Otherwise, the data will not be stored when written into memory resulting in erroneous information being read back from the stack by a return or pop instruction. The other rule is that the area in memory used for the stack must be large enough to hold as many levels of subroutine calls and push instructions as will be made at any one time by the program. For example, if the maximum number of subroutine calls made in succession is ten, and within those subroutines four register pairs are pushed onto the stack, the stack must have at least 034 (octal) memory locations assigned for its use. If not, the stack operations will write on the portion of the program that resides just below the stack area.

The use of the stack for storing the return address of a subroutine is an automatic function of the 8080. Storing and loading registers and flag status from the stack, on the other hand, is a function controlled completely by the program. The push and pop instructions provide a great deal of power to the instruction set for temporarily saving registers and status without bothering with the exact location in memory to be used to store them. Suppose the contents of regis-

ters D and E are used as a pointer to a data table, and, at some point, this pointer must be saved for use later in the routine. By executing a PUSD instruction, the pointer is saved on the stack and all that is required to recall it is a POPD instruction.

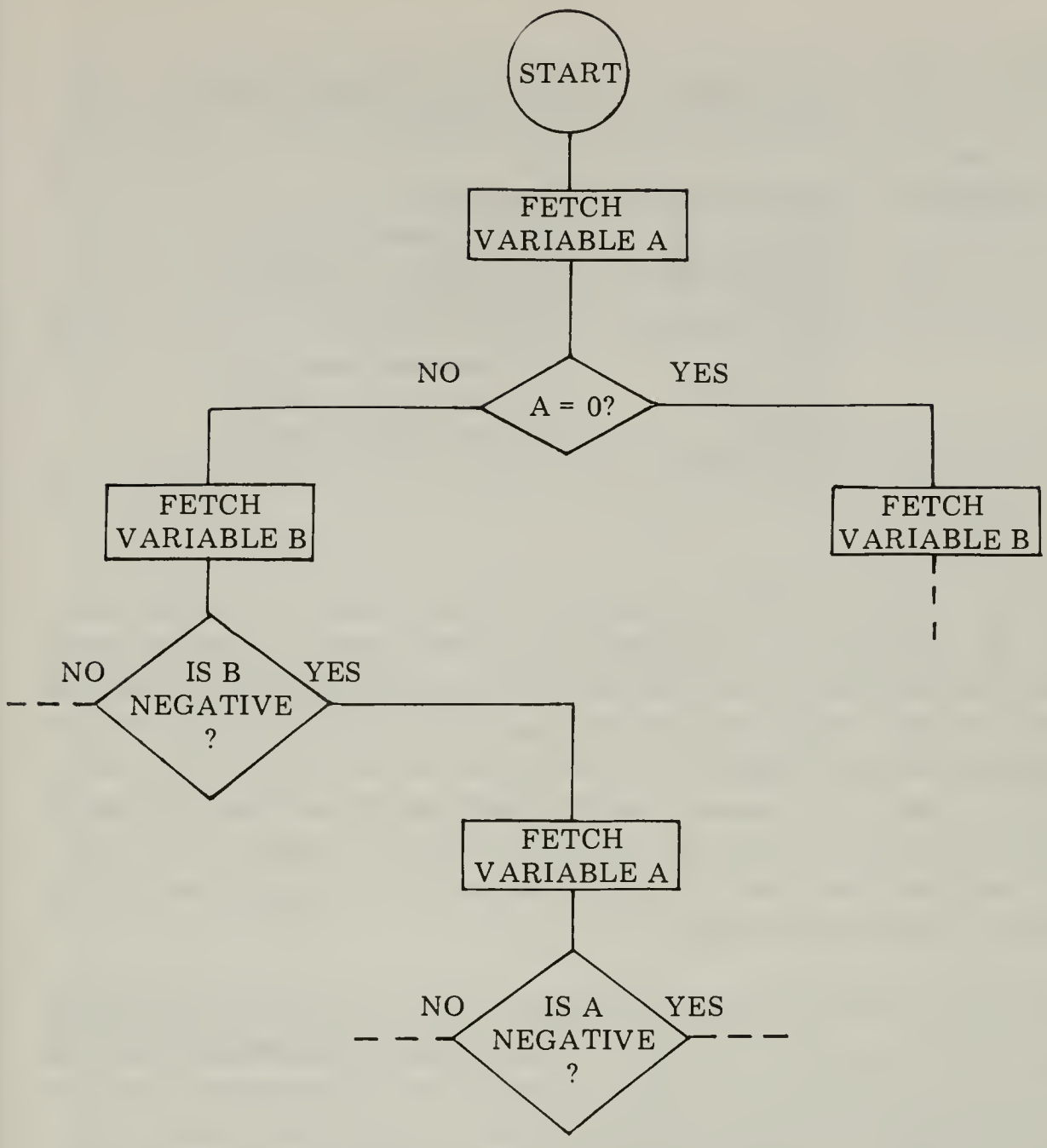
Going one step further, when the contents of all the registers must be saved, as might be the case before calling a subroutine, the most convenient means of doing so is to execute each push instruction just before the subroutine call. Then, upon returning from the subroutine, the registers may be restored by executing the pop instructions in the reverse order of the push instructions. This procedure is illustrated below.

```
PUSH        Save registers H and L
PUSD        Save registers D and E
PUSB        Save registers B and C
PUSS        Save the accumulator and status
CAL SUBR    Call the subroutine
POPS        Restore the accumulator and status
POPB        Restore registers B and C
POPD        Restore registers D and E
POPH        Restore registers H and L
```

Of course, if it is often required to save and then restore all the registers when calling a specific subroutine, it would save program steps to have the subroutine save the registers when it is called and restore them before returning. This is often done when writing an I/O subroutine, since it is most often to be used as a general I/O routine by many programs.

Saving the flag status in the stack can also save program steps. For example, when the condition of several variables in a program are tested in determining which logic path to follow, it may be necessary to check the same variable several times during the process. The partial flow chart on the following page illustrates this point.

First, it is determined whether variable A is equal to zero. If not, variable B is checked for the condition of its most significant bit, indicating whether it is negative or positive. If negative, then variable A is checked again for a negative condition. The following programs



were both written to perform this function, however, one uses the stack to save the status of variable A for the second test, while the other redetermines variable A's status.

```
START,    LTA VARIA     Fetch variable A
          NDA           Set flag status for test
          JTZ AZERO     If zero, jump to AZERO
          LTA VARIB     Fetch variable B
          NDA           Set flag status for test
          JFS BPOS      If positive, jump to BPOS
          LTA VARIA     Fetch variable A
          NDA           Set flag status for test
```

```
        JTS ABNEG       If negative, jump to ABNEG
        . . .

START,  LTA VARIA       Fetch variable A
        NDA             Set flag status for test
        JTZ AZERO       If zero, jump to AZERO
        PUSS            Save variable A status
        LTA VARIB       Fetch variable B
        NDA             Set flag status for test
        JFS BPOS        If positive, jump to BPOS
        POPS            Fetch variable A status
        JTS ABNEG       If negative, jump to ABNEG
        . . .
```

Adding up the number of memory locations used by each routine, it can be seen that by using the stack, two memory locations were saved. This may not seem like much, but suppose the other decision branches of the program also rechecked the status of variable A. By comparing the use of these two techniques for each of the other logic branches, it will be observed that for each additional branch that rechecks the status of variable A, three memory locations are saved. Thus, one can see how effective the use of the stack can be for storage of registers and status.

When the stack contains information that is to be examined or altered by the program, there are several ways of positioning the stack pointer to the desired location to read the information from the stack. If the data is at the current location of the stack pointer, it is simply a matter of popping the data into the required register pair. If, however, the stack pointer is not indicating the location in the stack of the desired information, the stack pointer can be moved by one of the following methods.

If the data is located at a higher address than the current address in the stack pointer, a series of pop instructions can be used to move the stack pointer to the proper location. Each pop will transfer the contents of the stack at that point to the register pair indicated by the instruction, with the final pop resulting in the transfer of the desired data.

In cases where the stack pointer is to be moved to a lower address

without disturbing the contents of the stack, or to move the stack pointer up or down without disturbing the contents of the CPU registers, the INXS and DCXS instructions can be used. These instructions increment and decrement the stack pointer one location for each execution. The stack pointer can be moved to the desired location without changing the contents of any memory location, CPU register, or flag status by performing either of these instructions as many times as necessary.

If the location desired for the stack pointer is known as a relative displacement from the current location of the stack pointer, the new location can be set up in the following manner. The displacement value is stored in registers H and L. The stack pointer is then added to registers H and L by performing a DADS instruction. Then, by transferring the new contents of registers H and L into the stack pointer, by the execution of the SPHL instruction, the stack pointer will indicate the desired location in the stack. The following routine adds a displacement of 020 to the stack pointer to position it for a transfer of data to registers D and E.

```
BEFORE ROUTINE

REGISTER L = 020
REGISTER H = 000
STACK POINTER = 011 376

                ...     The displacement is in registers H and L
                DADS    Add the stack pointer to registers H and L
                SPHL    Set up the stack pointer from reg H and L
                POPD    Load registers D and E from the stack
                ...

AFTER THE ROUTINE

REGISTER L = 016
REGISTER H = 012
STACK POINTER = 012 020
```

There are several facts to be pointed out in this routine. First, one may note that the initial value stored in the stack pointer when added to the displacement value in registers H and L creates the carry

condition which causes register H to be incremented. Next, the final address in registers H and L is two less than the final address in the stack pointer. This is due to the POPD instruction which incremented the stack pointer to read the data off the stack and into registers D and E.

There is one stack instruction that does not change the stack pointer as a result of its execution. This instruction, XTHL, exchanges the contents of the register pair H and L with the contents of the stack as indicated by the stack pointer. At the completion of the operation, the stack pointer indicates the same location in the stack as before the instruction was executed. This instruction allows access to the contents of the stack without losing the contents of registers H and L or changing the position of the stack pointer. Since the contents of registers H and L are moved to the stack by the XTHL instruction, they can be restored by a second execution of the XTHL instruction.

Manipulating the stack pointer must be a very precise operation, especially in a subroutine, where the return address is stored in the stack. When a subroutine changes the stack pointer, to store or retrieve data from another portion of the stack or another data area, the stack pointer must be restored to its initial location before the return can be properly executed. Failure to restore the stack pointer will result in the subroutine returning to an erroneous address. If the location of the stack pointer is not critical upon returning from the subroutine, and there is a register pair that is not used by the subroutine, the return address can be carried by the subroutine by popping it into the unused register pair at the beginning of the subroutine. Then, when it is time to return, the stack pointer can be set to an appropriate location in the stack area. The return address can then be pushed onto the stack and by executing a return instruction, the subroutine will return to the proper address in the program.

The proper use of the stack instructions is important for efficient programming. Saving and recalling information in the stack can save instructions, execution time, and memory requirements for a given program. Specific applications of the stack instructions will be presented in various routines throughout the remainder of this book.

GENERAL PURPOSE ROUTINES

Whenever one writes a program, there are usually several basic operations that are included in the program in the form of subroutines. Although the overall objective of various programs may be to perform totally unrelated tasks, the same group of subroutines may be called on to provide these basic functions. These subroutines perform such functions as clearing a section of memory, moving a block of data from one section of memory to another, manipulating multiple precision numbers, comparing data, creating software delays, and generating random numbers.

This chapter contains a number of routines that may be used to aid in fulfilling the subroutine requirements of many programs. These routines may be set up as subroutines, as presented here, if their operation is required by different portions of a program. Or, they may be revised to be used in line with the main instruction sequence of a program, should their operation be required only once by the program. Such revision requires either eliminating the return instruction at the end of the routine, or changing the conditional return instructions within the routines to conditional jump instructions.

The following list is a summary of the subroutines presented in this chapter.

Clearing a section of memory
Transferring a section of memory
Operations with multiple precision numbers
    Incrementing a multiple precision number
    Decrementing a multiple precision number
    Rotating a multiple precision number
    Complementing a multiple precision number
    Multiple precision addition and subtraction
    Comparing two multiple precision numbers
Checking for a value within given limits
Programmed time delays
Random number generators

CLEARING A SECTION OF MEMORY

When setting up a program for entering data or storing the results of a calculation in a section of memory, it is often desirable to clear the memory locations to be used for the storage. This operation is achieved by filling the memory locations with all zeros. One way to do this would be to perform a series of LMI 000 followed by INXH instructions until the entire section is cleared. This method is fine if the area to be cleared consists of only two or three memory locations and the clearing operation is required in only one or two different portions of the program. However, if a lengthy table area must be cleared, such as an input buffer (which may store 72 characters or more for a single line of input), this method would be highly impractical. Even for shorter tables, if they are to be cleared by different routines throughout a program, this method would use up more memory locations than are necessary.

An alternative is to use a subroutine that, when called, will clear as many locations in a table area as defined by the calling program. The routine listed below will fill up to 256 memory locations with zeros. The calling program must set the memory pointer registers H and L to the lowest address of the section to be cleared, and register B to the binary count of the number of memory locations in the section. The routine begins at the label CLRMEM. The address of the last location cleared is contained in registers H and L upon returning.

```
CLRMEM,   LMI 000      Load memory with 000
          DCB          Decrement the location counter
          RTZ          If counter = 0, operation complete
          INXH         Otherwise, advance memory pointer
          JMP CLRMEM   And continue clearing
```

TRANSFERRING A SECTION OF MEMORY

Programs of varying applications often have the similar requirement of transferring information in one section of memory to

another. For example, an editor program may transfer text from the input buffer to the main text buffer, while a calculator program may transfer a multiple precision value from a storage area to a working area in memory. In either case, the programming to perform this function is basically the same. The start address of the section of memory to be transferred and the section of memory to receive the data are set up along with either a count of the number of memory locations to be transferred, or the address of the last location to be transferred.

The first transfer routine uses register B as a binary counter for the number of locations to be transferred. This counter is set up by the calling program, which also sets registers H and L to the lowest address of the section to be transferred, and registers D and E to the lowest address of the section to receive the data. The memory pointers indicate the addresses of the last locations affected by the transfer. This routine begins at the label MOVEIT.

```
MOVEIT,   LAM          Fetch contents to be moved
          STAD         Store contents in new location
          DCB          Decrement location counter
          RTZ          If counter = 0, return
          INXH         Otherwise, advance FROM pointer
          INXD         Advance TO pointer
          JMP MOVEIT   Continue transfer
```

The next transfer routine checks the contents of registers B and C with the contents of the FROM registers H and L to determine when the last location has been transferred. This method allows more than the maximum of 256 locations to be transferred as is the case with the previous transfer routine. When this routine is called, registers H and L must be set to the lowest address of the section to be transferred, registers D and E must be set to the lowest address of the section to receive the data, and registers B and C must be set to the last address of the section being transferred. The routine begins at the label MOVEAD and returns with registers H and L and D and E pointing to the last locations involved in the transfer.

```
MOVEAD,   LAM          Fetch contents to be moved
          STAD         Store contents in new location
          LAH          Fetch high portion of FROM address
          CPB          Is FROM page = page limit?
          JFZ MOVAD1   No, continue transfer
          LAL          Yes, fetch low part of FROM address
          CPC          Is FROM low = low address limit?
          RTZ          Yes, return to calling program
MOVAD1,   INXH         No, advance FROM pointer
          INXD         Advance TO pointer
          JMP MOVEAD   Continue with next transfer
```

MULTIPLE PRECISION ROUTINES

When dealing with numerical data, it is often necessary to use more than one eight-bit byte to represent a binary number. Since a single byte can only represent the values from 0 to 255, one would be quite limited in the type of calculations that could be performed. This problem is solved by manipulating the data in several bytes as though they were one long register or memory location, N times 8 bits long (N = the number of bytes used to represent the data value). For example, by using two bytes as though they were a single 16 bit register, the decimal values from 0 to 65,535 may be represented in binary format. This form of representation is referred to as multiple precision.

In order to perform operations that consider several bytes as one, there must be some link to carry the effects of an operation on one byte over to the next. This link is the carry flag. The carry flag indicates whether an operation on one byte of a multiple precision operation should carry over to the next byte. When the addition of a number to a low order byte of a multiple precision value creates an overflow, the carry flag will be set to a '1' and will be included in the addition of the next higher byte of the multiple precision value. Similarly, when a subtraction requires a borrow for the MSB of a multiple precision byte, the carry will be set and create a borrow from the LSB of the next higher byte of the multiple precision number.

The subroutines to be described next perform a variety of multiple

precision operations on values stored in memory. These operations include incrementing, decrementing, rotating left, rotating right, and complementing a single multiple precision value, and adding, subtracting, and comparing a pair of multiple precision values with each other. For these routines, the multiple precision value(s) is assumed to be stored in consecutive memory locations with the least significant byte in the lowest address. For several of the operations, routines that use a multiple precision value stored in the CPU registers will also be presented.

INCREMENTING A MULTIPLE PRECISION VALUE

There are a number of different reasons why a multiple precision value may have to be incremented. It may be to advance a memory pointer that is stored in memory, or to increment an event counter, or to simply add one to a binary value. For whatever reason, the basic process consists of incrementing the least significant byte and, if it goes to zero as a result, the next byte will be incremented. If this byte also goes to zero, the third byte will be incremented. This process ends when a byte does not go to zero when incremented or when the most significant byte has been incremented.

The first instruction sequence may be used to increment a double precision value by simply loading the registers H and L with the double precision value and using the INXH instruction to increment it. As one may recall, the INXH instruction automatically increments the H register if register L goes to zero when incremented.

```
LHLD DBLPCN     Load H and L with the double precision value
INXH            Increment as a double precision value
SHLD DBLPCN     Save incremented double precision value
```

The next routine increments a multiple precision value stored in memory as indicated by registers H and L. The number of bytes used for the multiple precision number is set in register B when this subroutine is called. There are two important facts one should be aware of when this subroutine returns. First, the contents of registers H and

L cannot be assumed to be pointing at any one particular byte since it returns when a byte is not incremented to zero, not after each byte has been incremented. Also, the Z flag will be set to '1' upon returning when the entire value has gone to zero. If any of the bytes do not go to zero, the Z flag will be '0' when the return is executed. This routine begins at the label INCMEM.

```
INCMEM,   INM          Increment memory contents
          RFZ          If result not 0, return
          DCB          Decrement precision counter
          RTZ          If last byte incremented, return
          INXH         Otherwise, advance memory pointer
          JMP INCMEM   And continue incrementing
```

The following subroutine may be used to increment a triple precision value stored in registers C, D, and E, with register C containing the least significant byte and register E containing the most significant byte. This routine begins at the label INCREG.

```
INCREG,   INC          Increment the least significant byte
          RFZ          If not zero, return
          IND          Increment the next byte
          RFZ          If not zero, return
          INE          Increment the most significant byte
          RET          Return
```

DECREMENTING A MULTIPLE PRECISION VALUE

The procedure for decrementing a multiple precision value is similar to that for incrementing except for the criteria used to determine when the succeeding bytes should be decremented. The next byte is only decremented when the byte being decremented goes from zero to 377 (octal), in which case a borrow is required from the next byte. The DCM instruction does not condition the flags to indicate the change from zero to 377, so a different instruction sequence must be used. This sequence uses the SUI 001 instruction to decrement a byte because it will cause the carry flag to be set to '1' when the zero transition occurs.

The following subroutine decrements a multiple precision value stored in memory. The calling routine must set registers H and L to the least significant byte, and register B to the number of bytes used for the multiple precision number. The contents of registers H and L cannot be assumed to point to any one particular memory location upon returning from this routine. The routine begins at the label DCRMEM.

```
DCRMEM,   LAM        Fetch memory contents
          SUI 001    Decrement the byte
          LMA        Save memory contents
          RFC        If no borrow required, return
          DCB        Decrement precision counter
          RTZ        If counter = 0, return
          INXH       Otherwise, advance memory pointer
          JMP DCRMEM And continue decrementing
```

The next routine decrements a triple precision value stored in registers C, D, and E. The registers are set up in the same manner as in the INCREG routine. Each register is loaded into the accumulator to be decremented allowing the carry flag to indicate whether the next register should be decremented. This routine begins at the label DCRREG.

```
DCRREG,   LAC        Load the L.S. byte into the ACC
          SUI 001    Decrement the least significant byte
          LCA        Save the least significant byte
          RFC        If no borrow, return
          LAD        Load the next byte into the ACC
          SUI 001    Decrement this byte
          LDA        Save the byte
          RFC        If no borrow, return
          LAE        Load the M.S. byte into the ACC
          SUI 001    Decrement the most significant byte
          LEA        Save the most significant byte
          RET        Return
```

ROTATING A MULTIPLE PRECISION VALUE

The next pair of subroutines may be used to rotate a multiple pre-

cision value stored in memory to the left, for the routine labeled ROTATL, or to the right, for the routine labeled ROTATR. As the reader familiar with the properties of binary numbers already knows, a binary number can be multiplied by two by simply shifting each bit one position to the left and loading the LSB with a '0.' And conversely, by shifting each bit of a binary number to the right one bit position and setting the MSB to '0,' the binary value is divided by two. When rotating a multiple precision number, it is necessary to carry the bit shifted out of a byte over to the next byte. This is done by using the rotate instructions which include the carry flag as part of the accumulator when rotating either left or right. Thus, for a rotate left operation, the MSB shifted out of the lower order byte will be shifted into the LSB of the next byte.

The first routine listed is the ROTATL subroutine. When this routine is called, registers H and L must point to the least significant byte of the value to be rotated, and the number of bytes used for the multiple precision value must be in register B. One should note that the initial instruction clears the carry flag. This creates the '0' bit, which must be loaded into the LSB of the multiple precision value. If the calling program desires to check for a '1' rotated out of the MSB of the value at the completion of the rotate operation, the carry flag will be properly conditioned upon returning to the calling program.

```
ROTATL,   NDA        Clear the carry flag
  ROTL,   LAM        Fetch the byte to be rotated
          RAL        Rotate the byte with the carry
          LMA        Save the rotated contents
          DCB        Decrement precision counter
          RTZ        If counter = 0, return
          INXH       Otherwise, advance memory pointer
          JMP ROTL   And continue rotating left
```

The ROTATR subroutine rotates the designated multiple precision value to the right. Registers H and L must indicate the most significant byte of the value when calling this routine, since it works from the most significant byte down to the least significant byte. Register B must be set to the number of bytes in the multiple precision value. Once again, the carry flag is cleared initially to provide the '0' to be shifted into the MSB of the value.

```
ROTATR,   NDA        Clear the carry flag
  ROTR,   LAM        Fetch the byte to be rotated
          RAR        Rotate right with the carry
          LMA        Save rotated contents
          DCB        Decrement precision counter
          RTZ        If counter = 0, return
          DCXH       Otherwise, decrement pointer
          JMP ROTR   And continue rotating right
```

COMPLEMENTING A MULTIPLE PRECISION NUMBER

The complement of a binary value is performed by changing each bit to the opposite condition of its current state. If a bit is a '1' it is changed to a '0,' if a bit is a '0' it is changed to a '1.' This type of complement is often referred to as the "one's complement" of a binary number. The one's complement is used, for example, to complement data received from an input device if it is in the opposite state of that required by the program. The complement of the inputted data may be derived by a simple CMA instruction just after the input instruction is executed.

Another form of binary complement is the "two's complement" which is formed by complementing each bit of the binary number and then adding a '1' to it. The two's complement is generally used when the negative value of a binary number is desired. Or, it may be used to form the negative of a subtrahend value that may then be added to the minuend, which effectively subtracts the subtrahend from the minuend. This procedure is used in the chapter on floating point arithmetic.

The following routine forms the two's complement of a multiple precision binary number stored in memory. When this routine is called, registers H and L must indicate the least significant byte of the multiple precision value to be complemented, and register B must contain the number of bytes defined for the value. The first byte is complemented, and then the binary value '1' is added to it. Then, each of the succeeding bytes is complemented, followed by the addition of the binary value '0' plus the carry flag. When the routine returns, the two's complement replaces the initial value in memory. This routine begins at the label COMPLM.

```
COMPLM,   LAM          Fetch the least significant byte
          CMA          Complement the least significant byte
          ADI 001      Form two's complement
MORCOM,   LMA          Store byte in memory
          DCB          Decrement precision counter
          RTZ          If counter = 0, return
          INXH         Otherwise, advance pointer
          LAM          Fetch next byte
          CMA          Complement byte
          ACI 000      Add possible carry from previous byte
          JMP MORCOM   Continue operation
```

MULTIPLE PRECISION ADDITION AND SUBTRACTION

Addition and subtraction are common functions often required when dealing with multiple precision values that represent numeric data. Both operations work from the least significant byte up to the most significant byte using the carry flag as the link between bytes. When the addition of two bytes results in an overflow from the MSB, the carry flag is set and is included in the addition of the LSB's of the next pair of bytes. Similarly, if the subtraction of a pair of bytes results in a borrow required from the next byte of the minuend, the carry flag is set, which causes a borrow from the LSB of the next byte of the subtraction.

The routines presented next perform the addition and subtraction of a pair of multiple precision values. Two addition routines are listed. One adds two multiple precision values stored in memory, and the other adds a triple precision value stored in registers C, D, and E to a value stored in memory. These two methods are listed to illustrate the difference in the sequences required for each. The subtraction routine operates on two values stored in memory.

The first addition routine is labeled ADDER. This routine adds the multiple precision value indicated by registers D and E to the value indicated by registers H and L with the result of the addition stored over the value indicated by registers H and L. The pointers D and E and H and L must be set to the least significant byte of each of the multiple precision numbers, and register B to the binary count of the number of bytes for each multiple precision value. The carry flag will

indicate whether an overflow from the MSB of the most significant byte has occurred upon returning from this routine. The calling routine may have to check this flag since such an overflow would usually indicate an error condition. At the completion of this routine, both pointers will be pointing at the most significant bytes of their respective multiple precision values.

```
ADDER,    NDA          Clear the carry flag
ADDMOR,   LDAD         Fetch byte from D and E pointer
          ACM          Add byte from H and L with carry
          LMA          Save result in H and L area
          DCB          Decrement precision counter
          RTZ          If counter = 0, return
          INXH         Otherwise, advance H and L pointer
          INXD         Advance D and E pointer
          JMP ADDMOR   Add next pair of bytes
```

The next addition routine adds a triple precision value stored in registers C, D, and E to a triple precision value stored in memory as indicated by registers H and L. When this routine is called, the H and L pointers must be set to the least significant byte of the triple precision value in memory, and the other triple precision value must be stored in registers C (containing the least significant byte), D, and E (containing the most significant byte). The result of the addition is stored in place of the value in memory, and registers H and L will be pointing to the most significant byte of the result upon returning. Also, the carry flag will indicate whether an overflow from the MSB of the most significant byte has occurred when the routine returns to the calling program. This routine begins at the label ADDREG.

```
ADDREG,   LAC          Fetch the L.S. byte from register C
          ADM          Add the L.S. byte from memory
                       without the carry
          LMA          Save the sum in memory
          INXH         Advance the memory pointer
          LAD          Fetch the next byte from register D
          ACM          Add the next byte from memory
                       with the carry
          LMA          Save the sum in memory
          INXH         Advance the memory pointer
          LAE          Fetch the M.S. byte from register E
```

ACM	Add the M.S. byte from memory with the carry
LMA	Save the sum in memory
RET	Return with carry flag indicating possible overflow

There are several significant differences between the two routines just presented. First, the ADDER routine requires that the carry flag be reset at the start of the routine. This is because the "add with the carry" is the only addition instruction used. Therefore, the carry must be zeroed for the addition of the first pair of bytes. The ADDREG routine, on the other hand, uses the "add without the carry" instruction as its first addition instruction, so the initial condition of the carry need not be preset.

Another difference is the number of bytes that can be used to represent a number. The ADDER routine can handle up to 256 bytes per number, while the ADDREG routine is limited by the number of CPU registers that can be used to hold the multiple precision value. ADDREG can be expanded to accommodate quadruple precision numbers by using register B to store the fourth byte. One final difference is the amount of time required by the routines to perform the addition. The ADDREG routine makes a single pass through the instruction sequence to perform the addition, while ADDER loops through the same instructions for each pair of bytes to be added. Therefore, even though the ADDER routine uses fewer instructions, it requires considerably more time to perform an addition than the ADDREG routine.

The subtraction routine, labeled SUBBER, subtracts two multiple precision values stored in memory. The register pair D and E must indicate the least significant byte of the minuend; register pair H and L must indicate the least significant byte of the subtrahend; and register B must contain the binary count of the number of bytes in each multiple precision value when this routine is called. The result of the subtraction is stored in place of the subtrahend, and the pointers will indicate the most significant bytes of their respective values upon returning to the calling program. The carry flag will indicate whether a borrow was required by the subtraction of the MSB of the most significant byte. This routine begins at the label SUBBER.

```
SUBBER,   NDA          Clear the carry flag
SUBMOR,   LDAD         Fetch minuend byte
          SBM          Subtract subtrahend byte with carry
          LMA          Save result in subtrahend area
          DCB          Decrement precision counter
          RTZ          If counter = 0, return
          INXH         Otherwise, advance subtrahend pointer
          INXD         Advance minuend pointer
          JMP SUBMOR   Continue subtraction
```

COMPARING TWO MULTIPLE PRECISION VALUES

It is often desired to determine whether one number is larger or smaller in magnitude than another. This fact may change the manner in which a program is to deal with the two numbers. For example, when subtracting two numbers it is usually necessary to subtract the larger from the smaller, and, if indicated, to change the sign of the result. The following routine may be used to compare two multiple precision numbers stored in memory.

This same routine may be used to compare alphabetic information, such as one name against another, to check for duplication or, if the character set is well ordered (as is the case with the ASCII code), to place the names in alphabetical order. (This application will be covered in the chapter on search and sort routines.)

The compare routine presented next compares the multiple precision value indicated by registers D and E against the value indicated by registers H and L. These register pairs must be initially set to the most significant byte of the values to be compared. For comparing alphabetical information, the pointers must begin at the first character of each name. Register B must be set to the binary count of the number of bytes to be compared. Upon returning, the carry and zero flags will be set to indicate the outcome of the comparison. The calling program must check these flags in the following order. First, if the value indicated by D and E is less than the value indicated by H and L, the carry flag will be set. If the carry flag is reset and the zero flag is also reset, the value indicated by D and E is greater than the value indicated by H and L. Finally, if the carry flag is reset and the zero flag is set, the two values are equal in magnitude. The regis-

ter pairs will indicate either the least significant byte of each multiple precision value, if the two values are equal, or the pair of bytes at which the comparison failed, upon returning to the calling program. This routine begins at the label CPRMEM.

(The two instructions marked by the @@ may be changed to INXD and INXH for comparing alphabetic data if the data is arranged with the first character in the lowest address for each of the character strings being compared.)

```
CPRMEM,   LDAD          Fetch byte from D and E
          CPM           Compare to H and L
          RTC           If D and E are less than H and L,
                        return with carry set
          RFZ           If D and E are greater than H and L,
                        return with C = 0 and Z = 0
          DCB           Decrement precision counter
          RTZ           If counter = 0, both equal - return
                        with C = 0 and Z = 1
          DCXD    @@    Otherwise, decrement D and E
          DCXH    @@    Decrement H and L
          JMP CPRMEM    Continue comparison
```

The following listing illustrates a possible instruction sequence for calling the CPRMEM routine and then checking the result of the compare operation upon its return.

```
. . .          Instructions leading up to the compare
LXD NMBR1      Set pointers D and E to one value to be compared
LXH NMBR2      Set pointers H and L to value to be compared against
LBI 005        Set precision counter to five bytes per number
CAL CPRMEM     Compare NMBR1 against NMBR2
JTC LESTHN     NMBR1 less than NMBR2, jump to LESTHN
JFZ GRTRTN     NMBR1 greater than NMBR2, jump to GTRTHN
. . .          NMBR1 equal to NMBR2, begin processing for
               equal values
```

CHECKING FOR VALUE WITHIN LIMITS

Another type of comparison often required is to check whether

the value of a byte of data falls within expected limits. One frequent application is in checking the code received from an input device. For example, a calculator program may check each character input for a legal digit code when it expects to be receiving only digital information, or a control program may check inputs from a sensing device to determine whether the parameter it is checking is within allowable limits. When the data being checked is to fall within sequential limits (limits defined by an upper and lower bound), the following type of routine may be used.

The routine compares a byte of data against the lower limit and then the upper limit + 1 of the boundaries in which the data byte must fall. The reason for checking the upper limit + 1 is to allow the condition of the carry flag, upon returning, to indicate whether the byte falls within the designated limits. When the routine returns to the calling program with the carry set, the byte is not within the limits. When this routine is called, the data byte to be checked must be in the accumulator.

The routine listed below checks for the ASCII code for the digits zero through nine, namely 260 through 271. If one desires to use this program to check for the ASCII code for the alphabet characters A through Z, for example, the immediate portion of the compare instructions would simply be changed to 301 (ASCII 'A') and 333 (one greater than ASCII 'Z'). This routine begins at the label LMTCHK.

```
LMTCHK,   CPI 260     Is data byte less than ASCII 0?
          RTC         Yes, not in limits, return with C = 1
          CPI 272     Is data byte greater than ASCII 9?
          CMC         If so, return with C = 1. Otherwise,
          RET         Byte is in limits, return with C = 0
```

PROGRAMMED TIME DELAYS

The computer is designed to execute a program stored in its memory as rapidly as it possibly can. It does not hesitate between instructions, as a human would, to contemplate the next operation it should perform. However, there are certain types of programs that require a

hesitation, or delay, between one operation and the next. One type of program might be a display program that outputs a frame of characters or pattern to a video device and then must wait a specific amount of time before outputting the next frame or pattern. Or, a delay may be required after outputting a control command, which turns on a motor driven device, to allow the motor to get up speed before a data transfer may be initiated with the device. A programmed delay may also be required between outputting each bit of a serial data pattern to allow the program to control the data transmission rate. By inserting programmed time delay sequences, one may affect these REAL TIME program applications.

As pointed out at the end of chapter one, each instruction requires a specific number of cycles and, therefore, a specific amount of time, to execute. By knowing the exact time for each instruction, a delay may be created by programming a group of instructions whose total execution time is as close to the desired delay time as possible. (For the 8080 with a clock frequency of two megahertz, one should be able to program a delay to within two or three microseconds of the desired time.) Also pointed out is the fact that, depending on the type of memory used in one's system, the actual timing for the instructions may vary from those presented in Appendix A. Before getting into the time delay programming, it is necessary to understand the method used by the hardware to access memory, and the differences between various types of memories so that one will be able to discern the actual timing for one's own system.

(The following description is presented in very general terms to enlighten the reader in the basic theory of memory acquisition. It is not intended to educate the reader in the specific details of hardware requirements for accessing memory. One should refer to the manual supplied by the hardware manufacturer for one's particular computer for details on memory access.)

When a computer must access a memory location, to read an instruction or data from it, or to write data into it, the address of the memory location is first placed on the memory address buss. Then, after the memory has been given time to select the memory location accessed, the contents of the location may be read from the data buss by the CPU, or the contents of the data buss may be written

into the memory location. The length of time required to access a memory location for reading or writing is referred to as the speed of the memory. Depending on the speed of the memory, the delay required between sending the address and accessing a memory location may vary. If the normal delay in the CPU instruction time is sufficient for the memory to react to the address selection, the data may be read or written in the following cycle. If, however, the memory cannot react fast enough, one or more wait cycles must be executed before reading from or writing to the memory location.

A wait cycle is essentially a "do nothing" state, that the hardware controls, to allow time for slow memory to access the proper location. A single wait cycle for the 8080 takes 0.5 microsecond to execute. The number of wait cycles used by a specific microcomputer for the memory access delay should be detailed in the hardware manual. The reader should refer to the description on memory read and write operations, which should be included in the hardware manual, to determine the exact number of wait cycles used when accessing memory. Knowing the number of wait cycles will allow one to program exact time delays on a system that uses ROM or static RAM memory.

The use of ROM or static RAM memory allows one to determine the exact timing required for each instruction, although it may be different from the times presented in Appendix A. Each time the CPU accesses memory for an instruction, the additional delay for the number of wait cycles used for each memory access must be added to the time required by that instruction. The final column in Appendix A indicates the number of times each instruction accesses memory. For example, instructions such as "load register to register," or "rotate the accumulator" require only one memory access. So, to calculate the actual time, if one or more wait cycles are used, the wait delay time is multiplied by the number of memory accesses and is added to the time given. Instructions such as "load register to memory" and "add immediate," that require two memory accesses must have two times the wait delay added to their indicated time. Instructions such as "call," and "exchange H and L with the stack" require five memory accesses to execute, which means that five times the wait delay for memory access must be added to the timing given.

The use of dynamic RAM memory in a system makes it difficult to calculate the instruction timing accurately. The reason for this is that the dynamic RAM memory requires a refresh cycle at least once every one or two milliseconds. (This time may vary for different types of dynamic memory.) A refresh cycle means that within the allotted time, each memory address must be accessed with a read cycle in order for the memory to maintain its current contents. This refresh process may interrupt the timing of the CPU instructions, since the refresh circuitry may be accessing a memory location at the same time the CPU may require a memory access. In this case, the CPU would have to wait for the refresh read to complete, thereby extending the time required for the instruction to execute. It is, therefore, only possible to calculate a minimum time delay for a given instruction sequence, and not the maximum when using dynamic RAM memory.

With a knowledge of the timing necessary for the instructions, one may begin to program time delays of specific duration. In programming a delay, one should strive to use as few instructions as possible, and care must be taken to insure that the instructions used in the delay do not interfere with the operation of the main program. Several forms of delay routines will now be presented. Unless stated otherwise, the times given are those listed in Appendix A, which assume no wait delay has been added.

For very short delays, in the order of two to twenty microseconds, several instructions that fall in the direct sequence of the program may be used. Suppose a delay of six microseconds is required at a certain point in a program. A "no operation" instruction requires two microseconds to execute, so the desired six microsecond delay may be derived by using three NOP instructions at the point in the program requiring the delay. (If the memory access circuitry adds two wait cycles to each memory access, the NOP instruction would take three microseconds, and only two NOP instructions would have to be inserted to create the six microsecond delay.)

Another method used to create short delays is to insert a call instruction that calls a location that contains a return instruction. This sequence would delay 8.5 microseconds for the call, plus 5.0 microseconds for the return, for a total of 13.5 microseconds. To

conserve memory, the return instruction may be part of an existing routine. It need not be set up as a return specifically for this delay.

For longer delays, the method of inserting the delay instructions in sequence with the main program would begin to waste a great deal of memory. An alternative is to use a subroutine that will form a timing loop to delay the desired amount of time. The following routine allows control of the delay time by selection of an initial value for register B. The delay is created by forming a program loop that decrements register B until it reaches zero, and then returns. The larger the initial value of register B, the longer the delay; the exception to this is that the initial value of zero will create the longest delay.

```
DELAY,    DCB          Decrement delay counter
                       (2.5 U'SEC)
          RTZ          If counter = 0, return
                       (2.5 U'SEC : 5.5 U'SEC)
          JMP DELAY    Else, continue looping
                       (5.0 U'SEC)
```

The amount of time used by this routine is calculated by adding up the time required for each instruction every time it is executed. The execution time of each instruction is given in parenthesis after each comment. The two times given for the RTZ instruction indicate the time required when the zero flag is false followed by the time required when the zero flag is true. The following formula may be used to calculate the delay time for a given value of register B. If register B is initially zero, the value of 256 must be substituted in this equation.

```
                 LBI CAL  DCB RTZ  JMP              DCB RTZ
DELAY TIME = 3.5 + 8.5 + (2.5 + 2.5 + 5.0)*((REG B)-1)+2.5 + 5.5
```

One fact observed in this formula is that the times required for the instructions LBI and CAL, which set up register B to the required constant and then call the delay subroutine, must be included in the calculation of the delay time. The time for LBI can be excluded if re-

gister B is set up previous to the time the delay is to begin.

The time delay that can be created by this program loop runs from a minimum of 20 microseconds for register B equal to one, to a maximum of 2,570 microseconds, for register B equal to zero, in increments of ten. This incremental factor is controlled by the loop DCB, RTZ, JMP. Should it be desired to expand the increment, and thereby extend the maximum delay possible, additional instructions may be added to this loop. For example, if the incremental factor is desired to be twelve microseconds rather than ten, a NOP instruction can be inserted between the RTZ and JMP instructions, which will add two microseconds to the loop without altering the basic operation of the routine.

When the actual delay required by a program does not equal one of the incremental times generated by the delay loop, the delay may be adjusted by setting register B to the closest incremental time without exceeding the time desired, and then adding one or two instructions to the calling sequence to bring the total delay to within one or two microseconds. For example, suppose a delay of 425 microseconds is needed. Selecting a value of 41 for register B will provide a delay of 420 microseconds. The additional 5 microseconds can be made up by adding an instruction sequence, such as INB, DCB to the calling routine before the CAL DELAY instruction. These additional instructions will add the necessary five microseconds to the total delay.

Substantially longer timing loops can be derived by nesting delay loops. Using two registers, one can set up a delay loop within a delay loop. Then, when one loop goes through a complete cycle, the second loop will be decremented once, thereby multiplying the time required for the inside loop by the initial value -1 of the register in the outside loop. The following routine, which includes the calling sequence, illustrates this method of nesting delay loops. Register B is used as the counter for the inside loop, and register C is used for the outside loop. Here again, the greater the initial value of the registers, the longer the delay, with the exception of the value zero, which creates the longest delay. The exact timing for this routine will be discussed following the listing.

```
                . . .
                LBI XXX         Set initial inside loop
                                (3.5 U'SEC)
                LCI YYY         Set outside loop
                                (3.5 U'SEC)
                CAL DLYLOP      Call delay loop
                                (8.5 U'SEC)
                . . .
DLYLOP,         DCC             Decrement outside loop
                                (2.5 U'SEC)
                RTZ             If zero, return
                                (2.5 U'SEC : 5.5 U'SEC)
DLYLP1,         DCB             Decrement inside loop
                                (2.5 U'SEC)
                JTZ DLYLOP      If zero, go to outside loop
                                (5.0 U'SEC)
                JMP DLYLP1      Else, continue inside loop
                                (5.0 U'SEC)
```

Calculation of the amount of time this routine will require for execution can be made from the formula given next. This formula is shown in two forms. One indicates the instruction sequence that is executed, and the second provides a condensed version for use in making the actual calculation.

```
                LBI   LCI   CAL
DELAY TIME = 3.5 + 3.5 + 8.5 +

                DCC  RTZ                    DCB  JTZ   JMP   DCB  JTZ
                [2.5 + 2.5 + ((INIT B) -1)*(2.5 + 5.0 + 5.0) + 2.5 + 2.5] +

                          DCC RTZ          DCB  JTZ JMP   DCB JTZ
                [((REG C)-2)*(2.5+ 2.5+ (255*(2.5+ 5.0+ 5.0))+2.5+ 5.0)]+

                DCC RTZ
                2.5 + 5.5
```

DELAY TIME = (((REG C) -2)*3200)+(((INIT B) -1)*12.5)+36

The first formula has two sections enclosed in brackets. The first bracketed section indicates the time for the first pass through the inside loop and the second bracketed section indicates the time for all

successive passes. The reason for the separation of these times is that on the first pass through the inside loop, the value of register B will be as initialized by the calling program. After the first pass, register B will always be zero when the inside loop is entered. This formula is only valid for initial values of register C from 2 through 256. (In actual operation of the subroutine, register C is initialized to zero when 256 is used in the formula.) If register C is initially set to one, the execution time is simply the sum of the times not enclosed in the brackets, which is 23.5 microseconds. For values of register C from 2 to 256, the time delay can be set within the limits of 36 to 816,023.5 microseconds in intervals of 12.5 microseconds. If finer selection is required, the technique discussed previously of inserting instructions in the calling sequence may be used.

RANDOM NUMBER GENERATORS

The purpose of a random number generator is to provide a non-repeating series of random numbers. These random numbers may be applied to a number of different programs. For instance, when a game such as dice or black jack is programmed, the program must provide a random assortment of numbers for the roll of the dice or draw of a card. This is accomplished by using some form of a random number generator routine. Another application for random number generators is in creating random patterns for testing devices, such as a computer's memory, which may be sensitive to various random bit patterns.

Two methods of programming random number generators will now be presented. The first is a very simple method that may be used when the numbers are required only in response to an input from the program operator. The second method uses a routine that will produce a new random number each time it is called.

When a program requires a random number only in response to an input received from the operator, the random number may be derived by constantly incrementing a register until the input is received. This may be accomplished by forming a program loop that increments the register and then checks the status of the input device for

an input from the operator. If the status indicates there is no input received, the routine will loop back to increment the register again. Since the program loop is so short, probably in the range of 30 to 50 microseconds, it would be impossible for a human to select the precise time to input a character, thereby stopping the loop when a specific value is in the register being incremented. So, for programs that require random numbers in response to operator inputs, the following routine may be used to generate the random numbers. The CHKINP routine called in this program is assumed to check the status of the input device and return with the sign flag set to '1' if a character has been entered on the keyboard, or set to '0' if a character has not been entered. When the character has been received, the value in register C may be used as the random number for the program.

```
RNDMLP,   INC          Increment random number
          CAL CHKINP   Check for character entered
          JFS RNDMLP   Not entered, increment again
          . . .        When entered, use register C for
                       random number
```

When a program requires random numbers at various times throughout its operation, not necessarily in response to an input, the following routine may be used. This routine generates a pseudo-random data pattern of eight bit bytes. This random number generator is not a true generator because it repeats itself every 501 numbers. However, the program that uses it can make the pattern more random by using a trick that will be described shortly.

This random number generator uses two memory locations, other than those required by the routine itself, to save the random number and an incrementing addend. Each time the routine is called, the random number created last is used in generating the next random number. It is operated on by the series of instructions in this routine and then the addend is added to it to create the new random number. At the same time, the addend will be incremented either once or twice, depending on the result of the addition of the random number to the addend, as indicated in the listing. The new random number will be saved in memory and returned to the calling program in the accumulator.

The trick referred to previously for increasing the randomness of the numbers generated is to have the calling program alter the contents of the addend at a point in the program that is occasionally executed. For example, if the program enters a subroutine once for every twenty or thirty times it calls the random number generator, an instruction sequence should be added to the subroutine to alter the addend. It may be altered by incrementing once, or adding five, or resetting the addend to zero. No matter what method is used to change the value of the addend, the result will be that of altering the data pattern generated, since the normal sequence of the addend will be disrupted.

The listing for the random number generator is given below. It begins at the label RANNUM. The random number is assumed to be stored in the memory location 000 100 with the addend in location 000 101. The contents of registers H and L will indicate the location of the random number upon returning to the calling program. The instruction sequence that follows this routine may be used by the calling program to alter the addend by adding five to it.

```
RANNUM,   LXH 100 000   Set pointer to random number
          LAM           Fetch random number
          RLC           Perform a series of
          XRM           Operations on it to
          RRC           Create a new random number
          INL           Advance pointer to addend
          INM           Increment addend
          ADM           Add addend
          JTP SKIP      If true parity, increment addend once
          INM           Otherwise, increment it twice
SKIP,     DCL           Set pointer to random number
          LMA           Save new random number
          RET           Return to calling program


          . . .         Sequence to add 5 to addend
          LTA ADDEND    Fetch addend
          ADI 005       Add five to addend
          STA ADDEND    Save new addend
          . . .         Continue processing
```

CONVERSION ROUTINES

The real power provided by a computer is exemplified by its capability to operate with unlimited variations of character codes by simply changing a program. It can accept information in one form, convert it to another for processing, and then output it in the same format as initially received, or in an entirely different format. One may be aware of various other devices that perform such conversions, however, the input and output codes are most likely fixed, allowing no variation. A computer may be programmed to utilize a variety of codes for input, processing, and output.

The need for code conversion and the type of conversion required is governed by two factors. The code used by the peripheral devices for transmitting and receiving data is one factor. If the input device transmits one code, and the output device must receive a different code, conversion from one to the other must be made at some point in the program. The other factor is the format required by the program to process the data. If this format is the same code as received from the input device, no conversion is required. Otherwise, the appropriate conversion must be made before the program can proceed with its processing.

The codes used by different I/O devices to transmit and receive data can vary from a standard code, used to represent a specific character set, to a code that has been designed into a special purpose device. Several of the standard codes used to represent alphanumeric information are ASCII, BAUDOT, EBCDIC, and HOLLERITH. ASCII and BAUDOT are commonly used on keyboard and printer or display devices, such as CRT terminals and teletypewriter machines. EBCDIC is generally used for mass storage devices, such as magnetic tape units, and HOLLERITH is generally associated with card reader/punch devices. The special purpose codes may be derived, for example, when interfacing a calculator-type keyboard to enter both numeric data and the functions that are to operate on the data.

The code used by a program to process the information input may be the same code as received, or it may require some conversion to a format convenient for the computer to operate with. When a pro-

gram deals with the manipulation of text, such as an editor program, the character code received by the program is often used for storing the text information. As each character is input, the code is stored as the representation to be used by the program for that character. For programs that deal with numeric data, for arithmetic operations or designating digital information, conversion from the character code received to the binary or decimal equivalent may be required. A calculator program might receive the data as ASCII encoded decimal digits that must be converted to the binary equivalent for processing, and then back to ASCII digits to output the answer. A monitor program may require the conversion of the coded octal or hexadecimal input to the binary equivalent for defining memory addresses and their contents.

As mentioned previously, there are a number of standard codes used to transfer data from a peripheral device to a computer and vice versa. For the following discussion on conversion from one code to another, the ASCII and BAUDOT codes will be used. Their contrasting formats aid in describing various methods of code conversion. Therefore, to preface the conversion routines, a brief discussion of each code is presented.

The ASCII code is a seven bit code that has codes representing the entire alphanumeric character set plus punctuation marks and a number of non-printing control characters. An eighth bit is often added to this code. This bit can be used to provide parity, for error checking, or it can be set to a constant '1' or '0' condition for all characters. The ASCII code for the printing characters and several of the control characters are presented in Appendix D, in both octal and hexadecimal notation. The code presented in Appendix D, and used throughout this book whenever ASCII is discussed, assumes the eighth bit is always set to '1.'

As the reader may notice by examining appendix D, the ASCII code is a well-ordered code. The letters of the alphabet are represented in sequential order from 301 (octal) for A to 332 (octal) for Z. The numbers are similarily ordered from 260 (octal) to 271 (octal) for the numbers 0 through 9, respectively. This coding for the numbers allows ease of conversion from ASCII to binary-coded-decimal by simply dropping the 4 most significant bits of the ASCII

code. The convenience of the ASCII code in this format is in sharp contrast to the BAUDOT code, discussed next.

The BAUDOT code is a five bit code used to represent the alphanumeric character set plus several punctuation marks and control characters. Appendix E contains the octal representation of the BAUDOT code. The code presented assumes the three most significant bits are all '0.' The reader unfamiliar with BAUDOT may question how 5 bits can be used to represent over 32 characters. The answer is quite simple. Each of the letters of the alphabet shares its code with a numeral or punctuation mark. Two separate control characters are used to determine which of the two possible characters is being transmitted, one indicating the letters and the other indicating the figures (numerals or punctuation marks). The proper mode (letters or figures) must be set by outputting the corresponding control character before the output of one or more of the characters of that mode. For example, if a sentence consisting entirely of letters were to be typed on a BAUDOT keyboard, the letters control character would be entered first, followed by the letters that make up the words of the sentence, and then to end the sentence, the figures control character would be entered followed by the period, which shares its code with the letter 'M.' It should be noted that the codes for the space, carriage return, line feed, and null characters are common to both modes.

Examination of the BAUDOT code in Appendix E reveals the obvious scrambled pattern of character codes. There is no set pattern that would lend itself to ease of recognition of the BAUDOT letters, as there is with the ASCII code. And, conversion of the BAUDOT code for the numerals to the equivalent BCD values is certainly not as trivial as conversion of the ASCII digits, described previously.

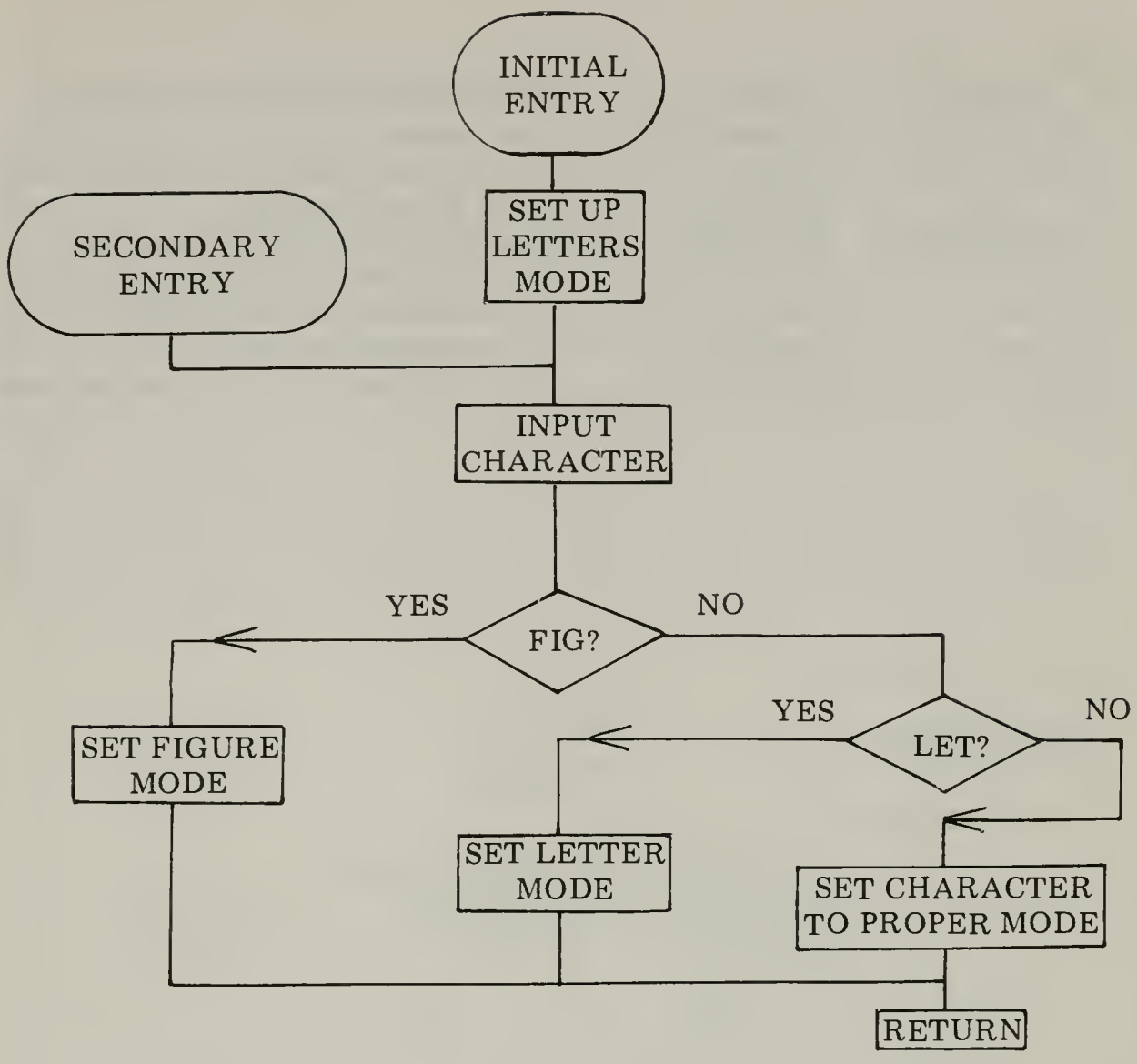
Programs that operate with the BAUDOT code to represent the character set, or perform conversion to or from another code, must have some means of differentiating between the two characters a BAUDOT code may represent. This can be accomplished by defining one of the three most significant bits as a mode designator. That is, one of these three bits would be set to '0' for the letter mode and to '1' for the figure mode. For this discussion, bit five will be so designated. The following pair of routines may be used to encode and

decode the BAUDOT characters according to this method for separating the letters from the figures.

The first routine is used to encode the BAUDOT characters as they are input. There are two entry points for this routine. The first, labeled BAUDIN, is used when the input of characters is to be initialized. The initialization is done by outputting a letters control character to the printer device before calling the input routine to receive a character from the keyboard, and setting register C to indicate the letters mode. The other entry point at label INBAUD is used after the initialization has been completed. The contents of register C are used to encode the characters as they are input, and it is conditioned by the receipt of the letters or figures control character. The routine returns to the calling program which must process the character contained in register A without disturbing register C (which contains the mode indicator). The listing and flow chart for this routine are now presented.

```
BAUDIN,   LAI 037      Load letters code into accumulator
          CAL OUTPUT   Call routine to send BAUDOT char
          CAL LETCOD   Initialize register C to letters
INBAUD,   CAL INPUT    Now accept BAUDOT chars fm mach
          CPI 033      See if figures code
          JTZ FIGCOD   Go set up '1' as sixth position bit
          CPI 037      See if letters code
          JTZ LETCOD   Go set up '0' as sixth position bit
          ADC          Add in status of sixth bit position
          RET          Return to process data
FIGCOD,   LCI 040      Set sixth bit in C to = '1'
          RET          Return to main routine
LETCOD,   LCI 000      Set sixth bit in C to = '0'
          RET          Return to main routine
```

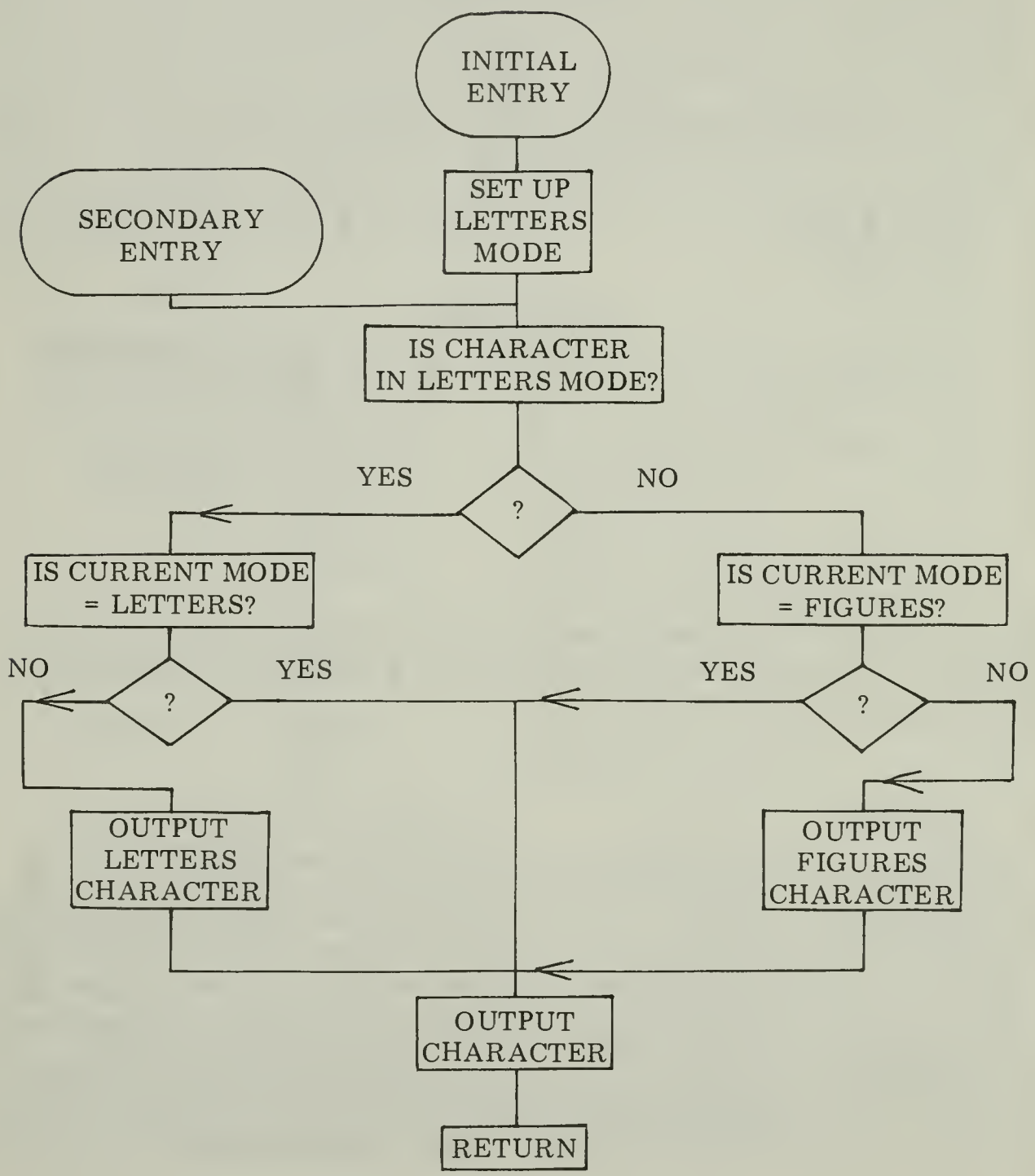
The following BAUDOT output routine also has two entry points. The BAUDOT entry point is called when the initial character of a string of characters is to be outputted. This entry point sets up the output device and register C to the letters mode before outputting the character. After the first character, the subsequent characters are output by using entry point OTBAUD. OTBAUD first checks the character to be output for a change from the current mode and, if different, the proper mode control character will be output before



the character. The character to be output must be stored in register B before calling either of these entry points. The flow chart is on the following page.

```
BAUDOT,   LAI 037        Load letters code into accumulator
          CAL OUTPUT     Call routine to send BAUDOT char
          LCI 000        Set indicator for letters in C
OTBAUD,   LAB            Move character from B to accumulator
          NDI 040        See if sixth bit =1, if yes = figures
          JTZ LTCHAR     Character, if not = letters character
          NDC            If figures see if last out also figures
          JTZ LASLET     If 0 here then last was a letters
OUTCOD,   LAB            Put present character in accumulator
          CAL OUTPUT     Send the BAUDOT character
          RET            Return to calling routine
```

```
LASLET,   LAI 033        Since last was letter put figure code
LASFIG,   CAL OUTPUT     Send code
          LCB            Save latest in register C for comparison
          JMP OUTCOD     Send current character
LTCHAR,   LAI 040        Set mask and see if last was letters
          NDC            By comparison of sixth bit position
          JTZ OUTCOD     If 0 here, last was also letters
          LAI 037        If not, send letters code first
          JMP LASFIG     By using above rtn to send letters code
```

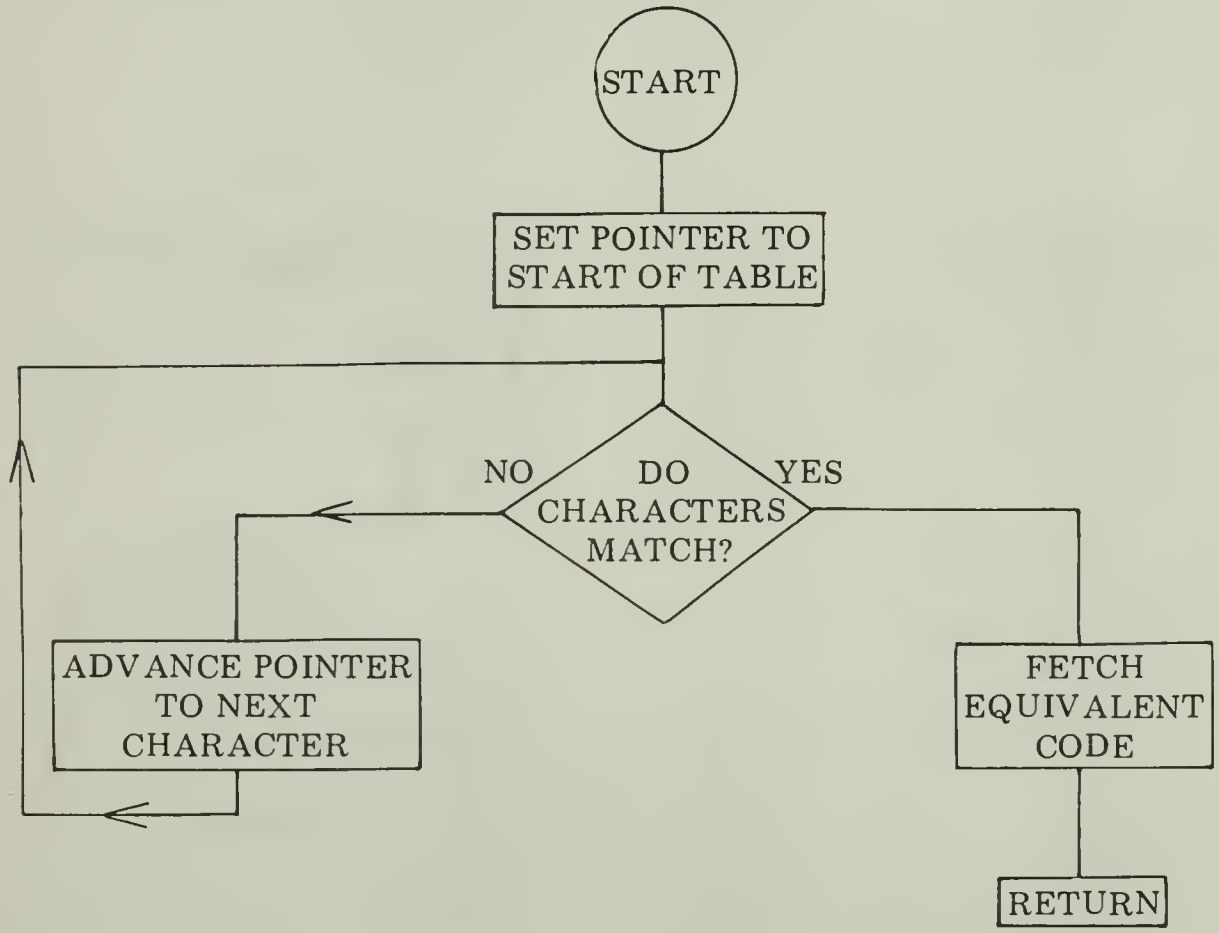


Using the ASCII and BAUDOT codes as the sample codes, two methods of code conversion will now be presented. The first method uses a look-up table. The table consists of both the ASCII and BAUDOT codes for each character of the character sets. The entries in the table are arranged in pairs. The first entry of a pair contains the ASCII code for the character, and the second entry contains the BAUDOT code for the same character. In cases where there is no equivalent BAUDOT code for a character, an appropriate substitute may be inserted (for example, the BAUDOT code for the left and right parenthesis, (and), may be substituted as the equivalent code for the ASCII left and right brackets, [and]), or the BAUDOT null character may be used when no suitable substitute is available. Sample entries for this table are presented next.

```
ADDRESS                OCTAL
                       CODE

07 000     ASBDTB,     301          ASCII A
07 001                 003          BAUDOT A
07 002                 302          ASCII B
07 003                 031          BAUDOT B
.   .                   .
.   .                   .
.   .                   .
07 076                 240          ASCII SPACE
07 077                 004          BAUDOT SPACE
07 100                 241          ASCII !
07 101                 055          BAUDOT !
07 102                 242          ASCII "
07 103                 061          BAUDOT "
07 104                 243          ASCII #
07 105                 064          BAUDOT #
.   .                   .
.   .                   .
07 136                 257          ASCII /
07 137                 075          BAUDOT /
07 140                 333          ASCII [
07 141                 057          BAUDOT ( (substitute)
07 142                 335          ASCII ]
07 143                 067          BAUDOT ) (substitute)
07 144                 250          ASCII (
07 145                 057          BAUDOT (
07 146                 251          ASCII )
```

```
07 147                  062         BAUDOT )
 .   .                   .
 .   .                   .
07 174                  277         ASCII ?
07 175                  071         BAUDOT ?
07 176                  300         ASCII @
07 177                  000         BAUDOT NULL (substitute)
07 200                  377         ASCII RUBOUT
07 201          BDASTB, 000         BAUDOT NULL
```

The conversion program that uses this table begins at one end of the table and compares the character code to be converted against the entries in the table of the same character set. For conversion from ASCII to BAUDOT, the ASCII code to be converted is compared to the ASCII entries in the table. When a match is found, the BAUDOT entry of the pair is returned as the BAUDOT equivalent. A similar process is used to convert BAUDOT to ASCII. The flow chart presented next indicates the logic used for conversion in either direction.



Since the conversion begins at one end of the table, when substitute characters are used, the true code conversion pair must be located such that it will be found before the substitute codes are encountered. For example, in the table given previously, the ASCII and BAUDOT pairs for left and right parenthesis must be placed in the table so that conversion from BAUDOT to ASCII will result in the ASCII code for the left and right parenthesis being returned and not the ASCII code for the left and right brackets. The correct location for these codes is illustrated in the table sample.

The listings for the conversion routines from ASCII to BAUDOT, and vice versa, using the look-up table are given next. While examining these routines, it should be noted that they both assume that the code to be converted is contained in the accumulator when the routine is called. It may also be noted that the ASCII to BAUDOT routine searches the table from the low address end while the BAUDOT to ASCII routine begins at the high address end of the table.

```
ASBAUD,     LXH ASBDTB      Set pointer to low address end of table
  FASCII,   CPM             Compare ASCII code to ACC contents
            INXH            Adv pointer before conditional jump
            JTZ FNDBDO      If match, do conversion
            INXH            Else, adv pointer to next ASCII code
            JMP FASCII  ††  And continue looking for match
FNDBDO,     LAM             Fetch BAUDOT equivalent into ACC
            RET             And return to calling program

BAUDAS,     LXH BDASTB      Set pointer to high address end of tbl
FBAUDO,     CPM             Cmpr BAUDOT code to ACC contents
            DCXH            Decr pointer before conditional jump
            JTZ FNDASC      If match, do conversion
            DCXH            Else, decr pntr to next BAUDOT code
            JMP FBAUDO ††   And continue looking for match
FNDASC,     LAM             Fetch ASCII equivalent into ACC
            RET             And return to calling program
```

Both routines just presented assume that the code to be converted is a valid code that is included in the look-up table. If for some reason the code in the accumulator may not be a valid code, an end of table test should be added to the conversion routines. The following routine may be inserted into either routine to test for the end

of the table by replacing the instruction marked by the †† with this instruction sequence. The immediate portion of the LAI instruction must be set to the value that register L will contain when the end of the table has been reached. For the ASBAUD routine, this would be the low address value of BDASTB+1, and for the BAUDAS routine the low address value would be ASBDTB-1. If the end of the table is reached, the accumulator should be loaded with an error code to indicate to the calling program that the conversion code was not found.

```
LBA             Save code being searched for
LAI XXX         Set up end of table low address
CPL             Set Z flag to test for end of table
LAB             Restore code if test fails
JFZ FZZZZZ      Not end, continue search at
                FASCII or FBAUDO
XRA             End of table, ret with ACC cleared
RET             Or use LAI instr to set error code
```

Another method of code conversion is to form a pointer out of the character code to be converted. This pointer is then used to point to the corresponding code in a conversion table. The conversion table contains a list of the conversion codes. Each entry is located at the address in the table to which the code to be converted will point when the pointer is formed.

In the following example, the conversion from ASCII to BAUDOT is made by setting the two most significant bits of the ASCII code to zero, thereby forming a pointer to the corresponding BAUDOT code in the conversion table. This table must therefore begin at location 000 of the page on which it resides. If it does not, a displacement constant must be added to the pointer to properly adjust it. For this routine, it is assumed that the table begins at location 000.

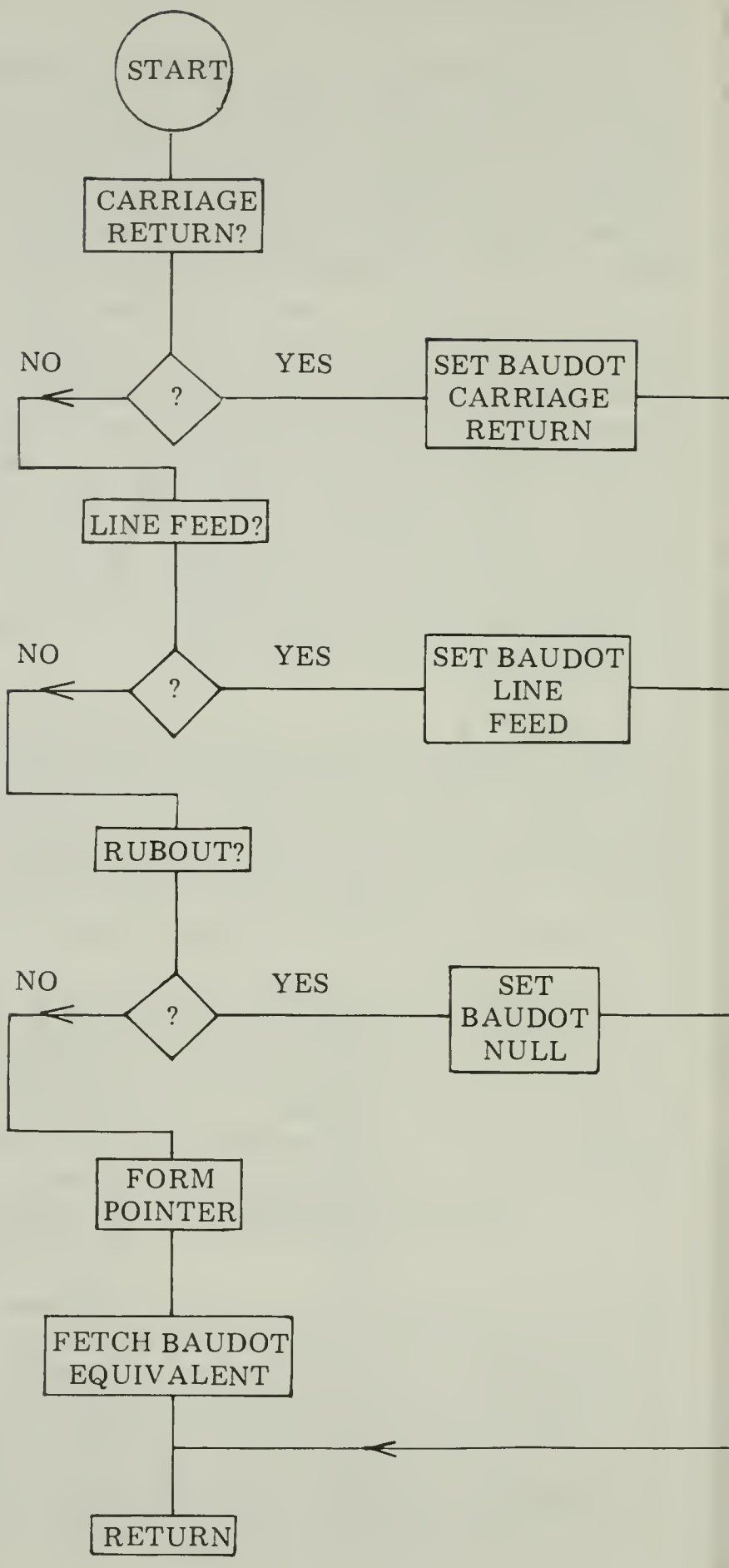
The conversion table uses 64 memory locations that contain the BAUDOT codes for the characters in the order corresponding to the pointer formed by the equivalent ASCII code. As in the previous look-up table, the use of substitute characters is required at the locations in the table for which no BAUDOT equivalents exist. Therefore, the first table entry is the null character, since an @ does not exist in the BAUDOT code. The next entry is the BAUDOT code for

an A, then B, and so on.

A special condition arises when the characters such as carriage return, line feed, and rubout are to be converted. In forming the pointer for these characters, it is noted that the carriage return forms a pointer to the same location as the letter M, the line feed forms a pointer to the same location as the letter J, and rubout forms a pointer to the same location as the ?. Since in this case, only three characters of this type need to be converted to BAUDOT, the conversion routine can check for their individual codes before forming the pointer. This eliminates the possibility of erroneous conversion. However, if the codes being converted have eight or more codes that overlap in this fashion, it would be more efficient, in memory usage, to expand the conversion table from 64 entries to 128, and to zero only the MSB of the ASCII code to form the pointer. This means that there will be more substitute characters contained in the table, but the actual conversion routine will not have to check each code to be converted for special characters.

The conversion routine shown next uses the pointer technique to convert from ASCII to BAUDOT with special consideration given to the carriage return, line feed, and rubout characters. This routine assumes that the ASCII code of the character to be converted is contained in the accumulator when the routine is called. The flow chart for ASBDPT is given following the listing.

```
ASBDPT,   LHI XXX      Set H to page of conversion table
          CPI 215      Carriage return?
          JTZ CARRET   Yes, fetch BAUDOT carriage return
          CPI 212      Line feed?
          JTZ LINFED   Yes, fetch BAUDOT line feed
          CPI 377      Rubout?
          JTZ RUBOUT   Yes, fetch BAUDOT null
          NDI 077      Mask off 2 MSB of ASCII to
          LLA          Form pointer to conversion table
          LAM          Fetch equivalent BAUDOT code
          RET          Return to calling program
CARRET,   LAI 010      Set BAUDOT carriage return in ACC
          RET          Return
LINFED,   LAI 002      Set BAUDOT line feed in ACC
          RET          Return
RUBOUT,   XRA          Set BAUDOT null in accumulator
          RET          Return
```


START
CARRIAGE RETURN?
NO
?
YES
SET BAUDOT CARRIAGE RETURN
LINE FEED?
NO
?
YES
SET BAUDOT LINE FEED
RUBOUT?
NO
?
YES
SET BAUDOT NULL
FORM POINTER
FETCH BAUDOT EQUIVALENT
RETURN

004 000	000	NULL FOR @
004 001	003	A
004 002	031	B
. .	.	INSERT BAUDOT CODES C TO Y FROM APPENDIX E
004 032	021	Z
004 033	057	(FOR [
004 034	000	NULL FOR \
004 035	062	) FOR]
004 036	000	NULL FOR ↑
004 037	000	NULL FOR ←
004 040	004	SPACE
004 041	055	!
004 042	061	”
004 043	064	#
004 044	051	$
004 045	000	NULL FOR %
004 046	072	&
004 047	053	’
004 050	057	(
004 051	062	)
004 052	000	NULL FOR *
004 053	000	NULL FOR +
004 054	054	,
055 043	043	-
004 056	074	.
004 057	075	/
004 060	066	0
004 061	067	1
004 062	063	2
004 063	041	3
004 064	052	4
004 065	060	5
004 066	065	6
004 067	047	7
004 070	046	8
004 071	070	9
004 072	056	:
004 073	076	;
004 074	000	NULL FOR <
004 075	000	NULL FOR =
004 076	000	NULL FOR >
004 077	071	?

There are several facts one must consider when choosing which method is to be used for code conversion. The first fact is whether

the conversion will be made in both directions, from code A to code B for input and then code B back to code A for output, or only one direction. If the conversion is only in one direction, the pointer method would shorten the table space required since only one code is included in the table area. For conversion in both directions, either method results in approximately the same memory requirement, unless the table for the pointer method has gaps of unused locations, as would be the case if the code forming the pointer has a non-sequential bit pattern.

In programs that require speed of conversion, the pointer method is the obvious choice. It provides the correct code with just a single pass through the instruction sequence. The look-up table method will remain in a loop until the correct code is found. This means that it could take up to sixty or more times longer than the pointer method to make a single conversion.

Another common type of conversion is the conversion of numeric data from one number base to another. The typical conversion of this type is from decimal to binary and binary to decimal. The reason this conversion is most common is that the decimal number system is used in the real world for most mathematical and numeric applications, while the computer is generally designed to operate most efficiently with binary numbers. Thus, to allow the real world and the computer to operate in their desired number systems, the conversion of decimal to binary, and vice versa, is required.

The first conversion routine converts a number designated by decimal digits in binary-coded-decimal format to the equivalent triple precision binary value. The decimal digits are contained in a table, labeled DECMAL, with one BCD digit stored per word. The table BINVAL consists of three consecutive memory locations used to store the binary number. This triple precision representation allows conversion of decimal values from 0 to 16,777,215. This routine works from the most significant decimal digit to the least significant decimal digit.

The major part of the conversion is done by a subroutine that multiplies the current contents of BINVAL by ten, and then adds one decimal digit to this new value. This subroutine, labeled TIMS10,

performs the multiplication by a series of rotate and addition operations, as explained in the commented portion of the listing. The listing for this subroutine is presented next. The reader may note that several of the subroutines presented in chapter three are used by this subroutine to aid in performing its function.

```
TIMS10,   PUSS          Save digit to be added
          LXH BINVAL    Set pointer to binary storage
          LXD WRKARA    Set pointer to working area
          LBI 003       Set precision counter
          CAL MOVEIT    Move binary value to work area
          LXH WRKARA    Set pointer to work area
          LBI 003       Set precision counter
          CAL ROTATL    Multiply work area X2 by rotate left
                                    (Total = X2)
          LXH WRKARA    Set pointer to work area
          LBI 003       Set precision counter
          CAL ROTATL    Multiply work area by X2 again
                                    (Total = X4)
          LXH WRKARA    Set pointer to work area
          LXD BINVAL    Set pointer to original binary value
          LBI 003       Set precision counter
          CAL ADDER     Add original to work area (Total = X5)
          LXH WRKARA    Set pointer to work area
          LBI 003       Set precision counter
          CAL ROTATL    Multiply work area X2
                        (Total now = X10)
          POPS          Fetch decimal digit from stack
          LXH BINVAL    Set pointer to binary table
          LMA           Load binary table with decimal digit
          INXH          And remainder with zeros
          LMI 000
          INXH
          LMI 000
          LXH BINVAL    Set pointer to binary table
          LXD WRKARA    Set pointer to work area
          LBI 003       Set precision counter
          CAL ADDER     Add decimal digit to binary value X10
          RET           Return with sum in binary table

BINVAL,   000           Storage for binary value lst signif byte
          000
          000           Most significant byte
WRKARA,   000           Temporary work area
          000
          000
```

```
DECMAL,   000     Decimal digit storage-units digit
          000     Ten's digit
          000     Hundred's digit
          000     Thousand's digit
          000     Ten thousand's digit
          000     Hundred thousand's digit
          000     Million's digit
DECML8,   000     Ten million's digit
```

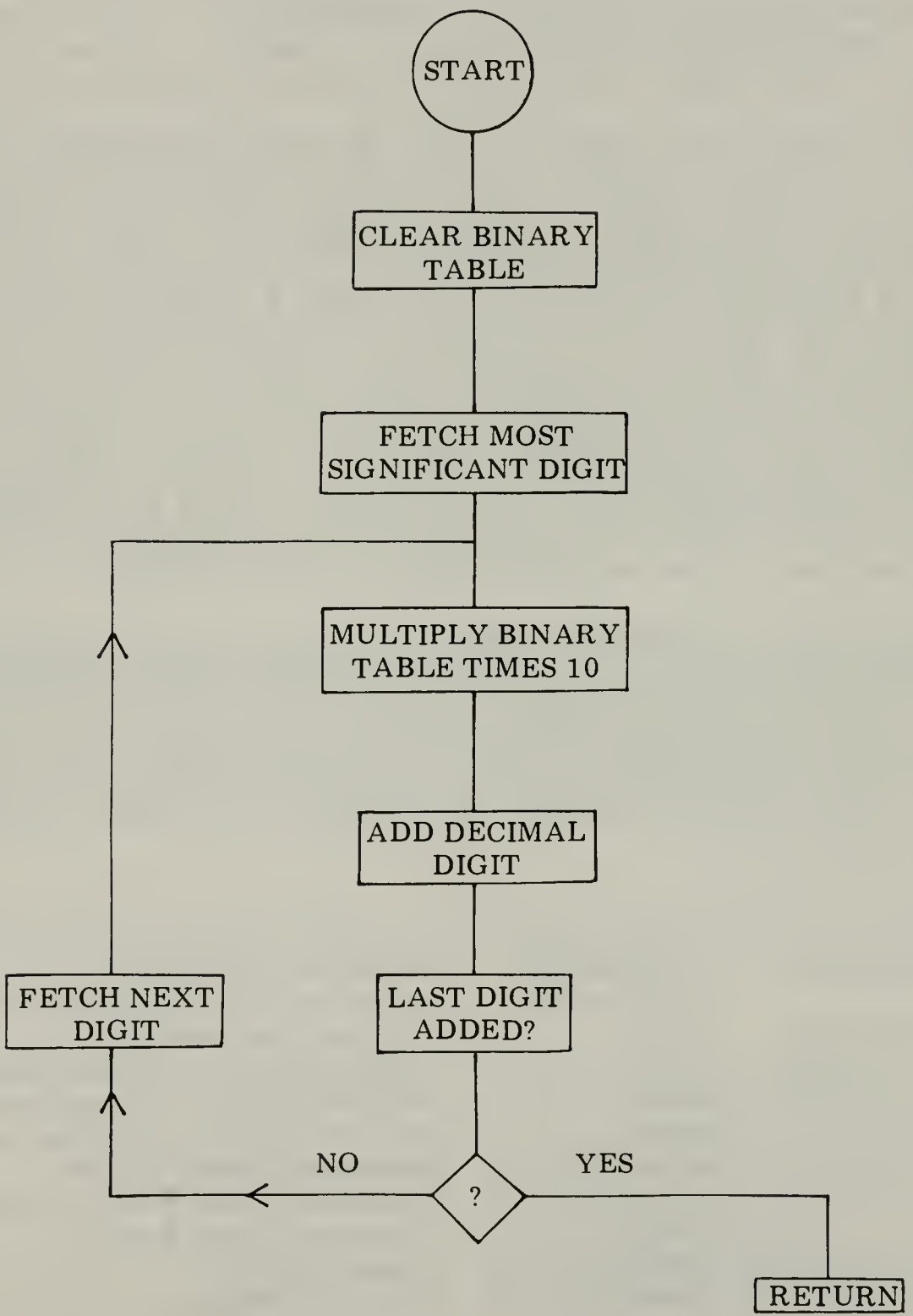
The DCTOBN routine, presented next, fetches the BCD digits from the DECMAL table for conversion to binary by the TIMS10 subroutine. First, using the CLRMEM subroutine, the three words used for the binary number storage are cleared. The routine then fetches one decimal digit at a time, beginning with the most significant digit, and calls the TIMS10 subroutine to add it to the binary value. When the conversion is complete, the routine returns to the calling program. Once again, this routine assumes the decimal digits are stored in the DECMAL table in BCD format, one digit per byte, before being called. This routine begins at the label DCTOBN.

The astute reader will undoubtably notice that the TIMS10 subroutine may be inserted in place of the CAL TIMS10 instruction, rather than being set up as a subroutine. This has been done for two reasons. First, by calling out this subroutine, the portion of the routine that performs the actual conversion is accentuated. Also, as will be pointed out in the chapter on floating point arithmetic, this subroutine may be used to convert decimal numbers directly to binary as they are entered by the operator.

The listing for DCTOBN is presented next followed by the flow chart.

```
DCTOBN,   LXH BINVAL    Set pointer to binary
          LBI 003       Number storage and load
          CAL CLRMEM    Area with zeros to clear
          LXH DECML8    Set pointer to MSD of decimal table
          LCI 010       Set digit counter
DBCNVT,   LAM           Fetch digit from decimal table
          PUSH          Save pointer in stack
          CAL TIMS10    Add digit to binary equivalent
          POPH          Restore printer to decimal table
```

```
DCXH        Decrement pointer to next digit
DCC         Decrement digit counter
JFZ DBCNVT  If not zero, continue conversion
RET         Otherwise, conversion is complete
```



The next routine performs the reverse function of the DCTOBN routine. It converts the triple precision binary value in BINVAL to the equivalent eight digit decimal number, which is then stored in the DECMAL table. This routine is called BNTODC.

BNTODC uses a subroutine labeled DCEQVL, to perform the actual conversion of the binary value to decimal. The conversion is made by subtraction of a binary constant equal to a decimal power of ten. When this subroutine is called, the memory pointer registers H and L must contain the address of the least significant byte of the power of ten to be subtracted. The indicated power of ten is then subtracted from the binary number being converted until the result of the subtraction requires a borrow for the MSB. This is indicated by the carry flag being set after the subtraction is made. (The subtraction portion of this subroutine is actually a slightly modified version of the SUBBER routine in chapter 3.) When the borrow occurs, the current power of ten is added back to the binary value to correct for the last subtraction. Register C contains the decimal value for the power of ten just subtracted when the subroutine returns. For example, if the binary value for one million can be subtracted five times from the binary number before the borrow occurs, the value of five would go in the seventh digit of the decimal equivalent.

This subroutine, like the TIMS10 subroutine in the previous conversion routine, can be placed in line with the BNTODC instruction sequence by replacing the CAL DCEQVL instruction with it. It is presented as a subroutine to bring out the significance of its operation to this conversion routine. The listing for the DCEQVL subroutine is given below.

```
DCEQVL,   LXD BINVAL     Set pointer to binary value
          LXB 000 003    Clear 'C,' set 'B' to precision counter
DCLOOP,   NDA            Clear carry flag initially
 DCLP1,   LDAD           Fetch binary value from D & E pointer
          SBM            Subtract value fm H & L pntr w/carry
          STAD           Save difference in H and L pntr area
          DCB            Decrement precision counter
          JTZ INCRVL     If pre cntr = 0, subtraction is complete
          INXH           Otherwise, advance H and L
          INXD           Advance D and E
          JMP DCLP1      Continue subtraction
```

```
INCRVL,   INC          Add 1 to decimal counter
          DCXH         Reset pointer to start of
          DCXH         Decimal constant
          LXD BINVAL   Reset pointer to start of binary table
          LBI 003      Set precision counter
          JFC DCLOOP   If borrow not needed, continue subtr
          DCC          Otherwise, decr decimal count to
          XCHG         Correct for extra loop and
          CAL ADDER    Add deci constant back to binary vlu
          RET          Return with decimal digit in C
```

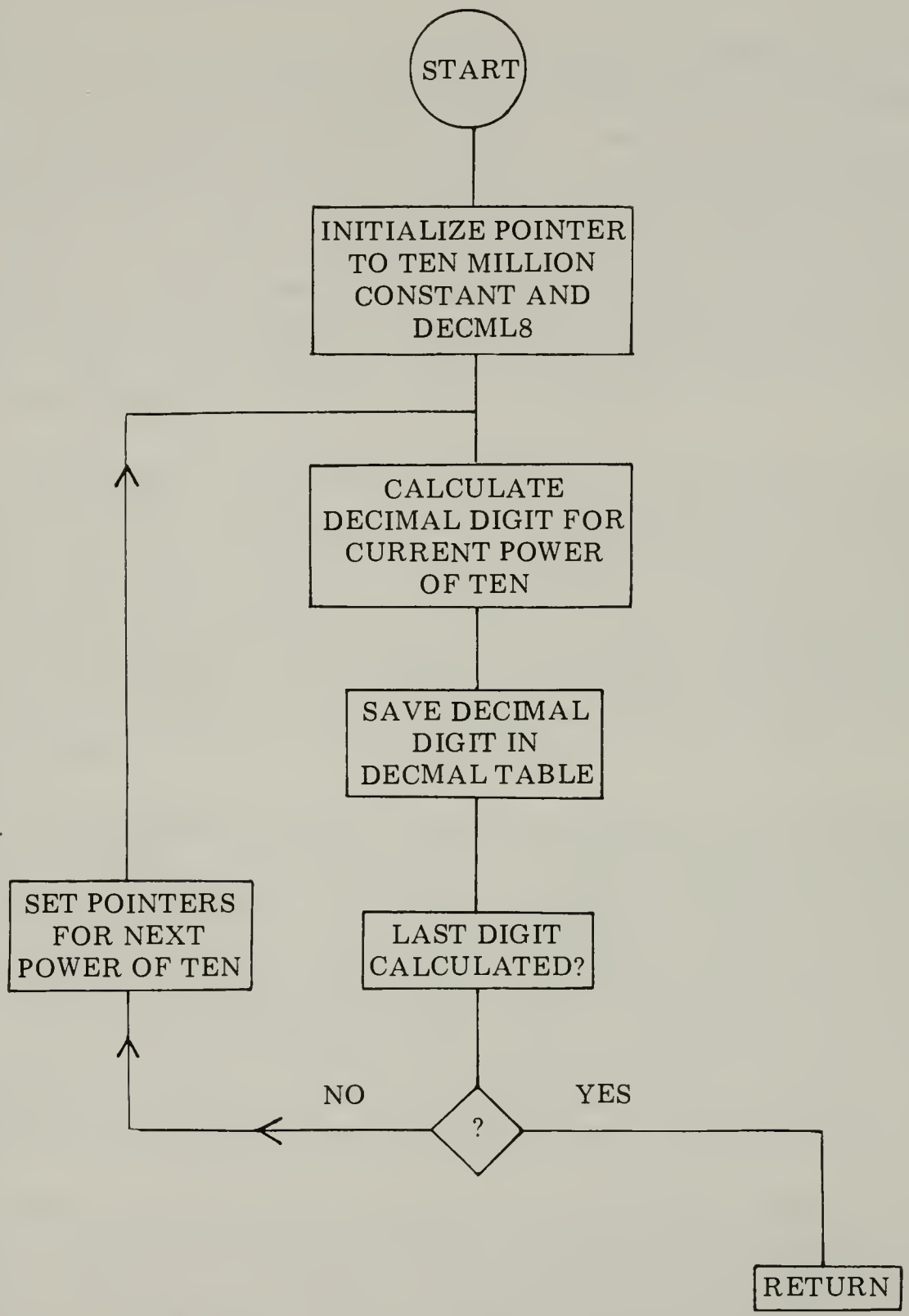
The BNTODC routine sets up and keeps track of the current power of ten being subtracted from the binary value by DCEQVL. As each power of ten, beginning with ten million and working down to one, is subtracted, and the value of the respective decimal digit is returned in register C, BNTODC stores the decimal digit value in DECMAL and advances to the next lower power of ten. When the decimal value of one has been subtracted, the routine returns with the decimal equivalent stored in the DECMAL table. It is important to note that the value in BINVAL will be zero when the conversion is complete. The calling program must save the original value of BINVAL if it is required after the conversion.

There is one location in the listing that must be filled in by the programmer when using this routine. This location is indicated by the XXX in the CPI instruction. The value that must be inserted here is the low portion of the address at which DECMAL is located. When the DECMAL pointer reaches this location, the conversion is complete.

The listing and flow chart are presented next along with the table of binary constants for the decimal values from ten million to one.

```
BNTODC,   LXH TENMIL   Set pntr to binary equiv of ten million
          LXD DECML8   Set pntr to MSD of DECMAL table
BNDC,     PUSH         Save binary equivalent pointer in stack
          PUSD         Save DECMAL pointer in stack
          CAL DCEQVL   Calculate decimal value of current digit
          POPH         Fetch DECMAL pointer
          LMC          Store decimal digit in DECMAL table
          XCHG         Move DECMAL pointer to D and E
```

```
        POPH        Fetch binary equivalent pointer
        INXH        Advance pointer to next binary
        INXH        Equivalent constant
        INXH
        LAE         Fetch pointer to DECMAL table
        CPI XXX     Cmpr to low addr prtn of DECMAL
        RTZ         If equal, conversion is complete
        DCXD        Otherwise, decr DECMAL table pntr
        JMP BNDC    And continue conversion


TENMIL, 200         Ten million in binary
        226
        230
ONEMIL, 100         One million in binary
        102
        017
HUNTHO, 240         One hundred thousand in binary
        206
        001
TENTHO, 020         Ten thousand in binary
        047
        000
ONETHO, 350         One thousand in binary
        003
        000
HUNDRED,144         One hundred in binary
        000
        000
   TEN, 012         Ten in binary
        000
        000
   ONE, 001         One in binary
        000
        000
```


START
INITIALIZE POINTER TO TEN MILLION CONSTANT AND DECML8
CALCULATE DECIMAL DIGIT FOR CURRENT POWER OF TEN
SAVE DECIMAL DIGIT IN DECMAL TABLE
LAST DIGIT CALCULATED?
NO
?
YES
SET POINTERS FOR NEXT POWER OF TEN
RETURN

DECIMAL ARITHMETIC ROUTINES

When using a computer to process mathematical data, such as data entered by an operator, and after processing, output for the operator to read, the decimal numbering system is most often the base used. This representation allows the operator to enter and read the data in a form most widely accepted and easily understood, since it is usually drummed into everyone from the time they are born. The computer, on the other hand, is generally designed to operate most efficiently with numbers in binary format. Therefore, there must be some means made available to allow the operator and the computer to communicate in a common number system.

As discussed in the previous chapter, conversion routines from one number base to another are often used. Routines, such as those presented, make it possible to input and output numbers in decimal notation while performing the actual calculations in binary notation. However, inaccuracies can creep into the most elementary calculation as a result of the conversion! For example, the subtraction of 2.1 from 5.0 may be output as 2.8999 rather than 2.9 due to conversion errors.

For applications where the operation required can be performed as decimal addition and subtraction, it would be far more accurate to perform these simple mathematical calculations in the same format as that used for input and output. The 8080 provides for this operation with an instruction that adjusts the contents of the accumulator to two binary coded decimal digits after an addition or subtraction has been executed. This instruction also conditions the carry flag to provide for multiple precision decimal calculations. This instruction is the decimal adjust accumulator instruction, mnemonic DAA.

Presented in this chapter are several routines that perform addition and subtraction of decimal numbers by utilizing the capability of the DAA instruction to operate with BCD digits. The format used to represent each number will be the same for all the routines. Four bits are required to define each BCD digit. Therefore, two digits will be stored in a single eight-bit byte, with the least significant digit of the pair in the least significant half of the byte. These operations work

with multiple precision values, allowing up to 256 bytes to be assigned for each number. For the routines presented, the bytes used to represent each number must be stored in a table of sequential memory locations, with the byte containing the least significant digit pair in the lowest address of the table.

The first routine is the decimal addition routine. Using the DAA instruction, this routine adds the BCD digits contained in one table in memory to the BCD digits contained in another table. When calling this routine, the least significant digit pair of one table must be indicated by the register pair D and E, and the least significant digit pair of the other table must be indicated by the register pair H and L. The contents of this second table will be replaced by the result of the addition when the routine returns to the calling program. Also, register C must be set to the binary count of the number of bytes per table.

This routine first clears the carry flag to initialize the carry before the addition of the first pair of digits. Each pair of digits in the H and L table is then added to the corresponding digit pair in the D and E table. The result of the addition of each pair of digits is adjusted to the proper decimal value by use of the DAA instruction before being stored in the second table. If the addition of the most significant digits results in the generation of an overflow, indicated by the carry flag being set, this condition of the carry flag will be maintained upon returning to the calling program. Since this condition may indicate an error, the calling program can check the carry and take whatever action may be appropriate. The listing for this routine is presented next. This routine begins at the label DECADD.

```
DECADD,   NDA          Clear the carry flag
 DCAD1,   LDAD         Fetch byte of first number
          ACM          Add byte of second number
          DAA          Decimal adjust
          LMA          Save byte of sum
          INXH         Increment 1 pointer
          INXD         Increment 2 pointer
          DCC          Last byte added?
          JFZ DCAD1    No, continue addition
          RET          Yes, return
```

It may be noted that the decimal addition routine just presented is similar to the ADDER subroutine in chapter three. The only major difference between the two routines is the addition of the DAA instruction in this routine. However, the same cannot be said for the subtraction routine, SUBBER, and the decimal subtraction routine to be described next.

The decimal subtraction routine subtracts the subtrahend value in one table in memory from the minuend value contained in another table in memory. This subtraction is performed by converting the subtrahend to the complemented BCD value and adding the minuend to it. This method is similar to forming the two's complement of a binary number and adding it to the binary minuend, which effectively subtracts the original value of the two's complemented number from the other.

When the number is broken down into two digit pairs, as in these routines, the actual complement of each BCD digit pair can be in one of two forms. These two forms are referred to as the 100's and 99's complement of the BCD value. The 100's complement is used when the result of the subtraction of the previous pair of digits does not require a borrow from this pair. The 99's complement is used when the subtraction of the previous pair does require a borrow from this pair. For the 100's complement, the value being complemented is subtracted from the hexidecimal value 9A which is formed by adding one to the BCD value 99. The 99's complement of the number is formed by subtracting the number to be complemented from the BCD value of 99. The result of the subtraction of either form is used as the BCD complement of that digit pair of the subtrahend.

When the complemented value is added to the minuend, the result is the same as subtracting the initial subtrahend from the minuend. However, by adding the complement, the accumulator, carry flag, and auxiliary carry flag are conditioned for the proper execution of the DAA instruction.

After the DAA instruction is executed, the carry flag will indicate the borrow, or underflow, requirement of the next pair. If the carry flag is set to one, there is no borrow required, and the 100's complement must be formed for the next pair. If the carry flag is reset to

zero, a borrow is required and the 99's complement must be formed. It is important to note that for this operation the carry indicates the opposite condition for the underflow, since addition of the complement is executed rather than subtraction of the actual subtrahend value.

The following calculation illustrates the process just discussed. The two four digit numbers are subtracted by forming the 100's or 99's complement, whichever is indicated, of the subtrahend, and adding it to the minuend. The least significant byte of the subtrahend is 100's complemented, since there is no borrow required. The second byte is 99's complemented, since the result of the subtraction of the first pair of digits requires a borrow from the second pair. The calculation presented is carried out two digits at a time.

4827 - 3246 = ?	
9A - 46 = 54	First, form 100's complement of the least significant pair of the subtrahend.
27 + 54 = 81	Minuend plus $\overline{\text{subtrahend}}$ = difference. No carry indicates next pair must be 99's complemented.
99 - 32 = 67	Form 99's complement of the next pair of the subtrahend.
48 + 67 = 115	Minuend + $\overline{\text{subtrahend}}$ = difference. The hundred's digit is dropped when result is formed. If an additional digit pair were to be subtracted, the presence of the hundred's digit here would mean the next subtrahend pair would be 100's complemented.
4827 - 3246 = 1581	Final result.

The subtraction routine listed next performs the decimal subtraction in the manner just described. When this routine is called, the contents of register pair H and L must indicate the least significant digit pair of the minuend; register pair D and E must indicate the least significant digit pair of the subtrahend; and register C must contain the binary count of the number of bytes for each number.

The result of the subtraction is stored in the subtrahend table at the completion of the subroutine's operation. Before returning, the carry flag will be set if a borrow is required for the subtraction of the most significant digit, or reset if the borrow is not required, to inform the calling program of a possible error condition. The listing for the DECSUB routine is given next.

```
DECSUB,   XCHG      Set pointer to subtrahend
          STC       Set carry for no borrow
DCSB1,    LAI 231   Load accumulator with BCD 99
          ACI 000   Add 0, to set up 100's or 99's comp
          SBM       Subtract subtrahend byte
          XCHG      Set pointer to minuend
          ACM       Add minuend
          DAA       Adjust accumulator
          XCHG      Set pointer to subtrahend
          LMA       Save result
          INXH      Increment subtrahend pointer
          INXD      Increment minuend pointer
          DCC       Last byte subtracted?
          JFZ DCSB1 No, continue subtraction
          CMC       Condition carry for error checking
          RET       Return with result in subtrahend
```

The next pair of routines uses these basic decimal addition and subtraction routines to perform the actual computation. These routines add the capability to perform addition and subtraction of signed decimal numbers. The sign and magnitude of the numbers to be added or subtracted are checked to determine whether the operation actually calls for an addition or subtraction, and to set up the proper sign for the result of the operation.

The two numbers to be operated on by these routines must be stored in two tables, referred to by the labels ONE and TWO. Table ONE is used to store one addend for the signed addition routine, and the minuend for the signed subtraction routine. Table TWO must contain the other addend for the signed addition routine, and the subtrahend for the signed subtraction. For both routines, the result of the respective operation is stored in table TWO upon returning to the calling program. The contents of table ONE remain unchanged.

The number of bytes in each table can be varied to allow for the number of digits desired per number. For these routines, the tables must be of equal length. The tables used in the routines presented here are four bytes long, allowing eight digits per number. If the length of the tables are changed, the instructions whose comments are marked with a **, must be changed to indicate the new word count.

The sign of each number is set up in separate memory locations and indicates a negative number if the sign bit (bit 7) of this byte is set, or a positive number if the sign bit is reset. The remaining bits in each sign byte must be all zeros, since there are several locations in the routine in which the sign bytes are checked as being equal, which involves the contents of the entire byte, not just the sign bit. Also, making the remaining bits equal to zero is consistant with the format used by these routines to set and reset the sign bit of the result. The sign bytes that refer to the sign of the values in table ONE and table TWO are labeled SIGN1 and SIGN2.

A third table, labeled SHFT, is used to allow the contents of table ONE and TWO to be shifted into TWO and SHFT, respectively. This is required when the magnitude of TWO is greater than ONE, and the signs of each are such that the contents of ONE must be subtracted from the contents of TWO. In order to return with the answer in TWO without changing the contents of table ONE, the table contents must be shifted as indicated. The SHFT table is then set up as the minuend and table TWO as the subtrahend when the decimal subtraction routine is called. The relative position of the three tables to each other must be maintained as indicated in the listing for the proper operation of the SHIFT routines.

The SHIFT routine performs the shift operation from ONE and TWO to TWO and SHFT in a manner similar to that discussed for the MOVEIT subroutine in chapter three. The difference here is that the process begins at the high address of the area shifted rather than at the low address end. The instruction marked by the †† in the SHIFT listing sets the byte count for the number of bytes to be shifted. This value is equal to twice the number of bytes per table, and, therefore, must be changed accordingly if the length of these tables is changed.

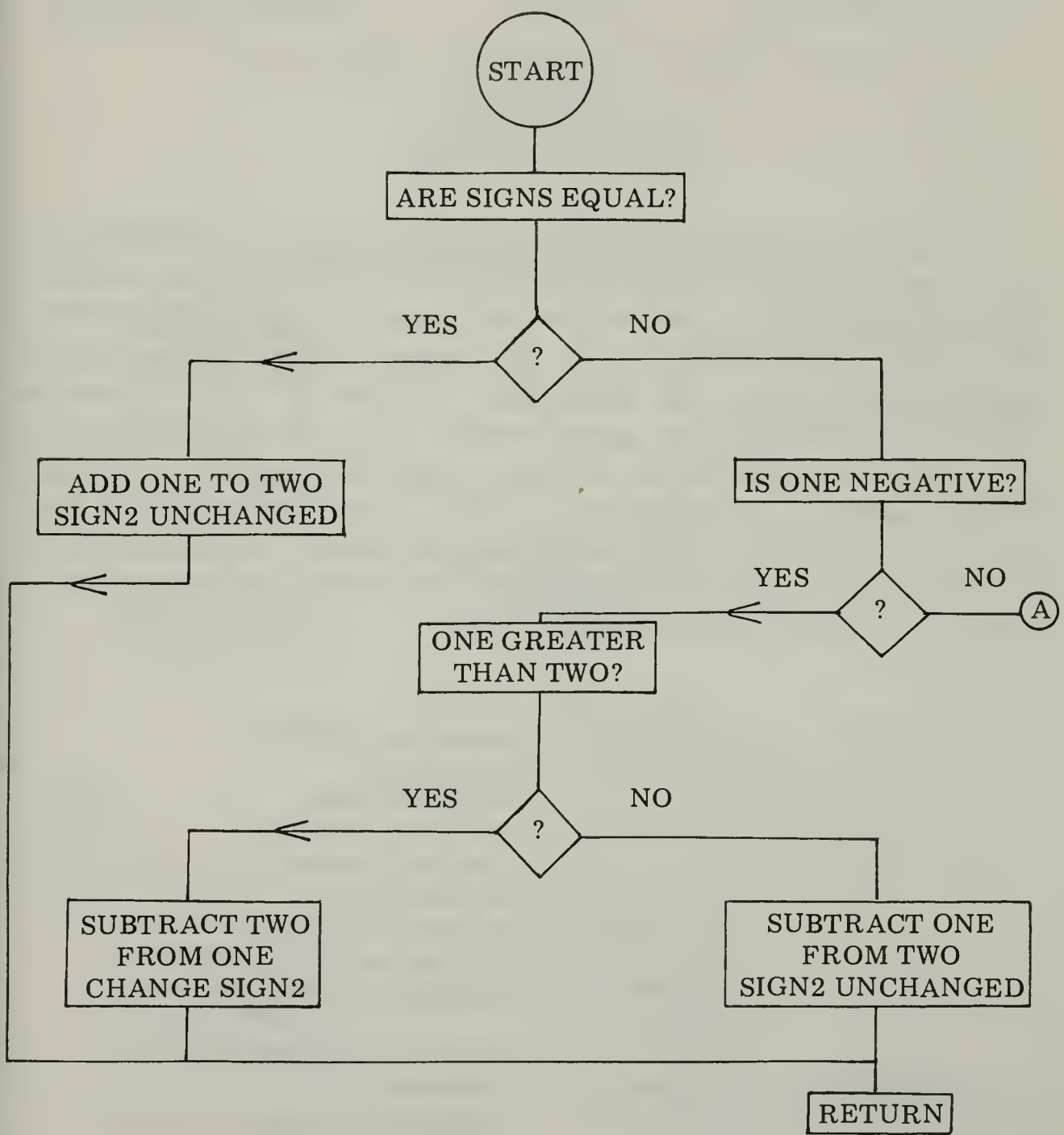
In the process of determining which operation is actually called for, addition or subtraction, the relative magnitudes of the two numbers must be known. This is determined by the CMPR subroutine. Its operation is basically the same as that of the CPRMEM subroutine in chapter three. The only difference in this routine is that the flags are set up to allow the calling program to determine the relationship of the two numbers, rather than the compare subroutine jumping to special routines as a result of one number being less than, greater than, or equal to the other. The entry point CMPR12 is used by the signed addition and subtraction routines to set up the table pointers and the byte counter.

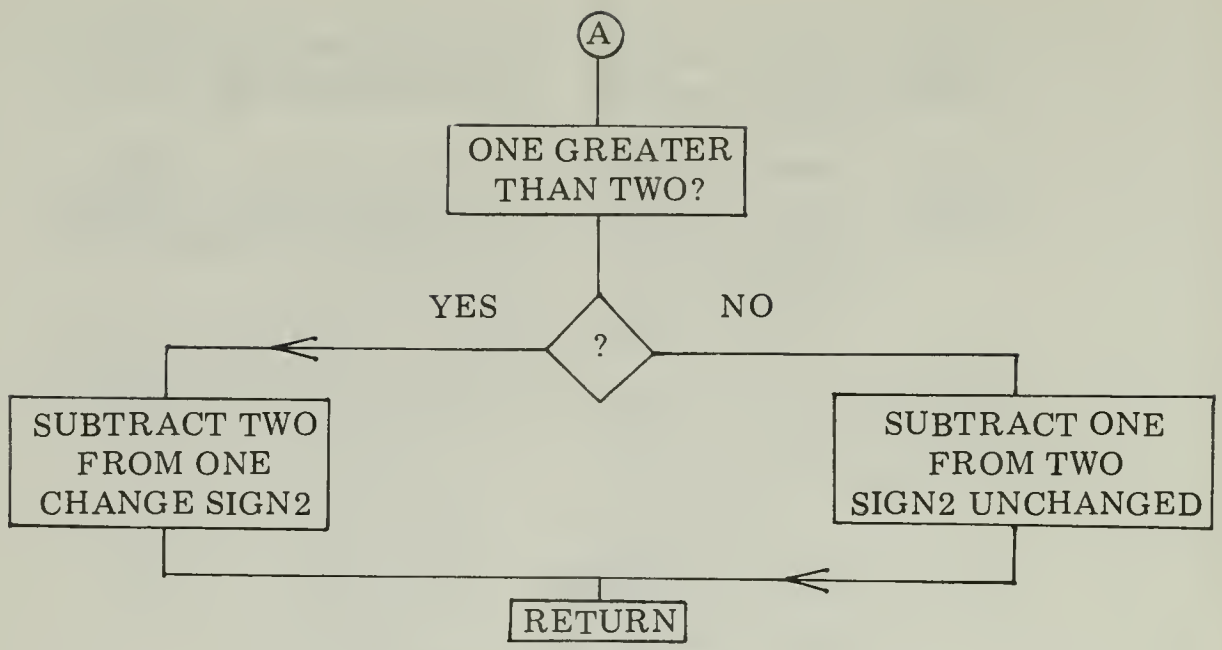
The signed addition routine, beginning at the label SGNADD, adds the contents of table ONE to table TWO, and returns with the answer in table TWO. The calling routine simply loads table ONE, SIGN1, table TWO, and SIGN2 with the desired values before calling this routine. When the sign of each is the same, the addition is performed as indicated. If the signs are different, the number of smaller magnitude is subtracted from the larger number, and the sign of the larger is set as the sign of the answer. The actual computation is done by one of the previous addition or subtraction subroutines. The condition of the carry flag upon returning to the calling program will indicate whether an overflow or underflow has occurred as a result of the operation, signalling a possible error condition. The operation of the signed addition routine is illustrated in the flow chart that follows the source listing.

```
SGNADD,   LXH SIGN1        Set pointer to sign of 1
          LTA SIGN2        Fetch sign of 2
          CPM              SIGN1 = SIGN2?
          JTZ SAR2         Yes, add two numbers and return
          JTC SAR3         ONE is negative, TWO is positive
SAR1,     CAL CMPR12       Is ONE greater than TWO?
          JTC SB12         No, subtract ONE from TWO
          XRA              Yes, clear accumulator and
          STA SIGN2        Change SIGN2 to positive
          JMP SB21         Subtract TWO from ONE

SB12,     CAL SHIFT        Shift ONE & TWO to TWO & SHFT
          LXH SHFT         Set up pointers for subtracting
          LXD TWO          Original ONE from TWO
          LCI 004     **   Set byte counter
```

```
                CAL DECSUB      Subtract ONE from TWO
                RET             Return to calling program

SAR2,           LXH TWO         Set up pointers for addition
                LXD ONE         Of ONE to TWO
                LCI 004     **  Set byte counter
                CAL DECADD      Add ONE to TWO
                RET             Return

SAR3,           CAL CMPR12      Is ONE greater than TWO?
                JTC SB12        No, subtract ONE from TWO
                LAI 200         Yes, change SIGN2
                STA SIGN2       To negative
SB21,           LXH ONE         Set up pointers to subtract
                LXD TWO         TWO from ONE
                LCI 004     **  Set byte counter
                CAL DECSUB      Subtract TWO from ONE
                RET             Return

CMPR12,         LXH TWOM        Set pointers to M.S. bytes
                LXD ONEM        Of ONE and TWO
                LCI 004     **  Set byte counter
CMPR,           LDAD            Fetch No. 1 byte
                CPM             Compare to No. 2 byte
                RFZ             If not equal, return w/flags
                                Indicating condition
                DCXH            Decrement pointer No. 2
                DCXD            Decrement pointer No. 1
                DCC             Last byte compared?
                JFZ CMPR        No, compare next byte
                RET             Return w/Z flag set indicating equal

SHIFT,          LXH TWOM        Set up FM pointer
                LXD SHFTM       Set up TO pointer
                LBI 010     ††  Set byte counter
MOVLOP,         LAM             Fetch FM byte
                STAD            Move TO new location
                DCXH            Back up FM pointer
                DCXD            Back up TO pointer
                DCB             Decrement byte counter. Last moved?
                JFZ MOVLOP      No, continue shifting
                RET             Yes, return

SIGN1,          000             Sign of table ONE
SIGN2,          000             Sign of table TWO
ONE,            000             Least significant byte of ONE
                000
                000
```

```
ONEM,   000     Most significnat byte of ONE
 TWO,   000     Least significant byte of TWO
        000
        000
TWOM,   000     Most significant byte of TWO
SHFT,   000     Least significant byte of SHFT
        000
        000
SHFTM,  000     Most significant byte of SHFT
```

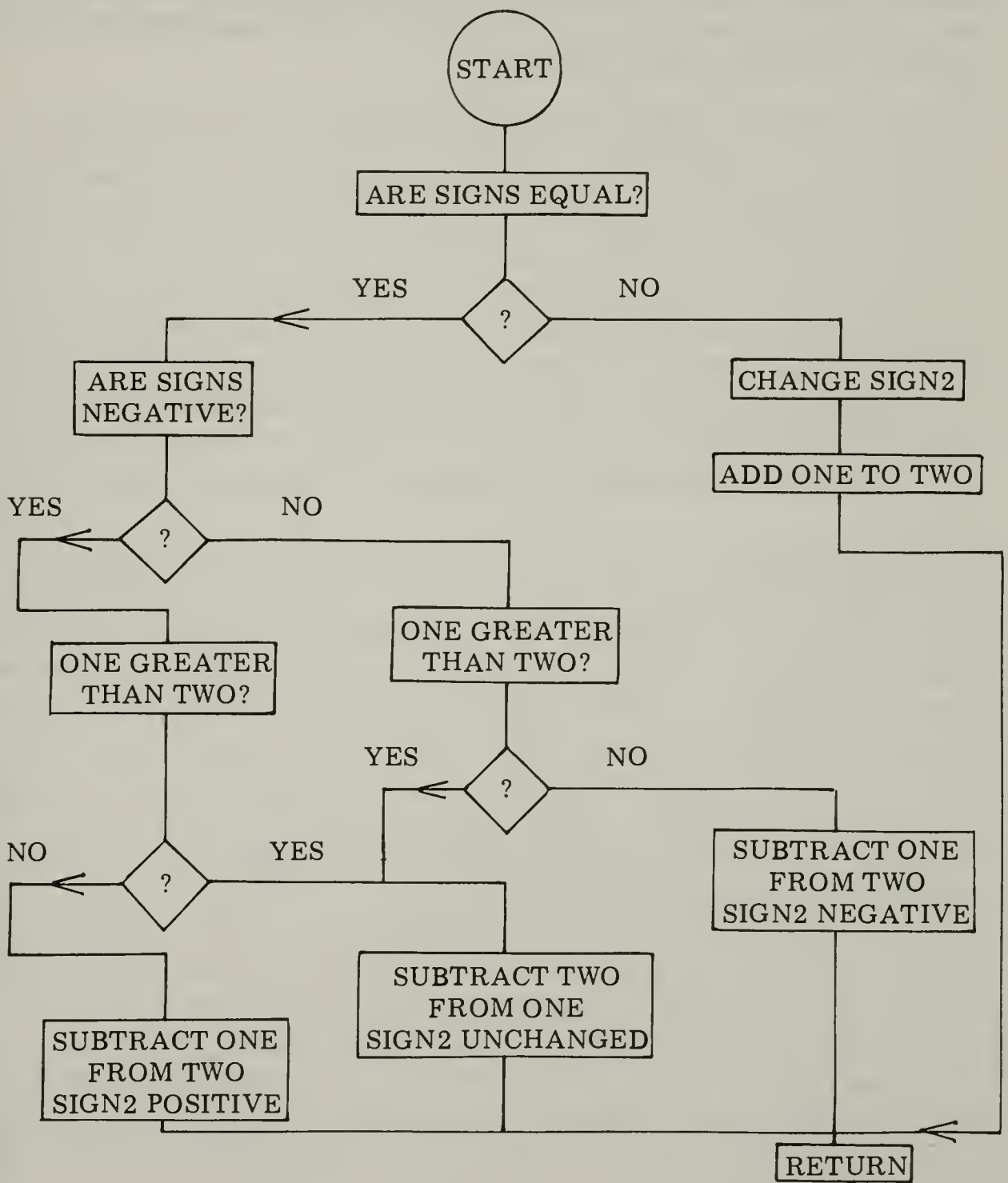




The signed subtraction routine, starting at the label SGNSUB, subtracts the contents of table TWO from the contents of table ONE. The calling program must set the contents of table ONE, SIGN1, table TWO, and SIGN2 with the desired values before calling SGNSUB. The sign and magnitude of each of the numbers is examined to determine the actual operation to be performed. Several of the routines in the signed addition routine are used here. Since the decimal addition or subtraction routine is the last operation executed, the condition of the carry flag will indicate whether an error condition has occurred. The listing and flow chart for the signed subtraction routine are presented next.

```
SGNSUB,   LXH SIGN1      Set pointer to SIGN1
          LTA SIGN2      Fetch SIGN2
          CPM            Both signs same?
          JFZ DIFSGN     No, different signs
          NDA            Yes, both positive?
          JTS NEG        No, both negative
          CAL CMPR12     Both pstv. Is ONE grtr than TWO?
          JFC SB21       Yes, subtract TWO from ONE
          LAI 200        No, set sign of TWO negative
          STA SIGN2
          JMP SB12       Subtract ONE from TWO
DIFSGN,   LTA SIGN2      Fetch sign of 2
          ADI 200        Change sign to opposite condition
```

```
                STA SIGN2
                JMP SAR2        Add ONE to TWO
NEG,            CAL CMPR12      ONE greater than TWO?
                JTZ NEG1        Equal, sign positive, result = 0
                JFC SB21        Yes, result neg. Sub. TWO from ONE
NEG1,           XRA             No, result pos. Sub. ONE from TWO
                STA SIGN2       Set SIGN2 positive
                JMP SB12        Subtract ONE from TWO
```



Using these routines as a base, expanded decimal arithmetic programs can be written. One possible addition might be to include a decimal point by specifying either a fixed number of digits in tables ONE and TWO to be to the left or right of the decimal point, or setting up a memory location to define a floating location for the decimal point. When the decimal point is set, the program must then ensure that each time a number is loaded into one of the tables, it is properly positioned for the location of the decimal point. These operations may also be expanded to perform multiplication and division by applying several of the techniques to be presented in the next chapter on floating point arithmetic.

FLOATING POINT ROUTINES

The concept of programming a computer to perform complex mathematical operations is often considered, by those unfamiliar with the techniques involved, to be too complicated a task to attempt. However, if one takes the time to break the program down into its basic functions, it is seen that the overall operation is not quite so difficult.

As one should be aware of by now, the digital computer is capable of performing mathematic operations with numbers of considerable magnitude. This is possible by representing numbers as multiple precision values in which more than one register or memory location is used to hold the numeric information. However, by increasing the number of locations assigned to represent a number, one could reach a point where the least significant bits become too insignificant with respect to the total value. A more practical representation would be to condense the size to a required number of significant digits and indicate the overall magnitude of the value by a power of the number base. This representation is referred to as floating point format.

Floating point format allows one to define a number as a product of two values. The first value contains the significant digits of the number. This value is referred to as the "mantissa." It should contain as many significant digits as needed to properly define its relative value. The second value contains the power to which the number base is to be raised. This value, called the exponent, indicates the magnitude of the significant digits of the mantissa. For example, the decimal value 1,000,000 would require a triple precision binary number to be properly represented. However, this same value can be defined as "1 X 10**6," or, in floating point notation, "1.0 E+6." This form contains the mantissa, 1, which is the single significant digit of this value, and the exponent, 6, which indicates the power of ten, or the number of places the decimal point should be moved to the right. One should easily realize that this shorter notation also requires fewer memory locations to represent the indicated value - one location to contain the significant digit (1) and a second to hold the exponent value (6).

One more major advantage this notation has over the individual multiple precision value is the capability to represent fractional numbers. By providing a sign bit for the exponent, negative as well as positive, values of the exponent can be expressed. Remember, a negative exponent forms the reciprocal of the power. For base ten, the exponent value -1 would indicate a value of 1/10 times the mantissa. The negative exponent moves the decimal point in the mantissa one place to the left for each integer value of the exponent.

This notation can be used to represent binary numbers as well. The binary mantissa contains the significant bits of the binary value, and the binary exponent will indicate the power of two to which the mantissa is raised, thereby indicating the location of the decimal point (or, to properly refer to it, the binary point). The same properties apply to the exponent for the binary numbers as decimal. If the exponent is positive, the binary point in the mantissa is actually located to the right by the number of places indicated by the exponent. A negative exponent shifts the binary point to the left. Putting it in more relative terms, if the mantissa is shifted to the right, the exponent must be incremented. Shifting the mantissa to the left means the exponent must be decremented. The following illustrates three ways of expressing the same number in binary floating point format.

101.0 E+0 = 5 X 1 = 5

.101 E+3 = 5/8 X 8 = 5

101000.0 E-3 = 40 X 1/8 = 5

This notation may be used to represent a wide range of values with a minimum number of memory locations. One or more memory locations may be set up to store the mantissa and the exponent. The number of locations used will depend on the number of significant bits desired to express each quantity.

The floating point routines presented in this chapter operate with binary floating point numbers in the following format. Each number will be stored in four memory locations. The first location will con-

tain the exponent with the most significant bit indicating the sign of the exponent. The sign bit will indicate a positive number if reset to zero and a negative exponent if set to one. The next three locations will be used to store the mantissa as a triple precision binary number with the most significant bit of the most significant byte used to indicate the sign of the mantissa. The binary point will always be implied to be to the right of the sign bit in the mantissa. One should note that there is no implied binary point in the exponent since the exponent is always assumed to be an integer value. This format is illustrated below.

EXPONENT	MSB	MANTISSA	LSB
SEEEEEEE	S.MMMMMMM	MMMMMMMM	MMMMMMMM
MEM LOC N+3	MEM LOC N+2	MEM LOC N+1	MEM LOC N

The order for storing the data in the memory locations should be noted. The exponent is stored in the highest address of the four locations used to store the floating point number. Also, since the sign bit takes up one bit for both the mantissa and the exponent, the number of bits used to represent each value is 23 (decimal) and 7, respectively.

Before presenting the floating point routines, it should be noted that various locations on page 00 are used for data storage. This data includes pointers and counters required at different times, several temporary storage tables, and two areas which are used frequently as operating registers. These two areas shall be referred to as the floating point accumulator and the floating point operand. The floating point accumulator is used as the accumulator of the floating point routines in performing calculations and storing the results of the operations performed. The floating point operand is used to store and manipulate the number operated on by the accumulator. These two locations have the same format as that defined for the floating point numbers just described. The floating point accumulator and operand shall be abbreviated as FPACC and FPOP throughout the remainder of this chapter.

The first routine to be presented is used to adjust the floating

point numbers to a common format for proper operation of the other floating point routines. In order for the floating point arithmetic routines to operate with the highest degree of accuracy possible, the value in the FPACC must be adjusted to a standard representation before the operations are performed. This representation is referred to as the "normalized" value. A number is considered to be normalized when the most significant bit with a '1' value in the mantissa is in the bit position just to the right of the implied binary point. If this bit is not a '1,' the number is normalized by shifting the mantissa to the left until the most significant '1' is just to the right of the implied binary point. For each bit position shifted to the left in the mantissa, the corresponding exponent must be decremented to maintain the actual value of the number. The resultant value of the mantissa will be a number greater than or equal to one half, and less than one. This process is illustrated below:

BEFORE NORMALIZATION	0.000110111000110000011010 E+0
AFTER NORMALIZATION	0.110111000110000011010000 E-3

The process of normalizing a floating point number is required for several reasons. First, it sets the values up in a common format with which the other routines can work effectively. Also, when data is entered by a human operator, it is generally received as decimal values that must be converted to binary. As pointed out previously, normalization of the binary equivalent is an important step in the conversion process. Finally, normalizing a number allows more significant digits in the mantissa. By insuring that one is using the most number of significant digits possible, the accuracy of the calculations will be increased.

In the input routine to be presented later, the normalization routine is called to normalize fixed point binary numbers that have been converted by a decimal input routine. These binary numbers are stored with the implied binary point to the right of the least significant bit. When the normalization routine is called for this purpose, register B must contain the octal value of 27 (23 decimal) which is used as the exponent of the value being normalized. (Register B is set to zero if the number to be normalized is already in the proper format.) Then, as the mantissa is shifted left, the exponent will be decremented to the proper value. The following example shows the number five being normalized to the proper notation.

BEFORE NORMALIZATION	00000000 00000000 00000101. E+27
AFTER NORMALIZATION	0.1010000 00000000 00000000 E+3

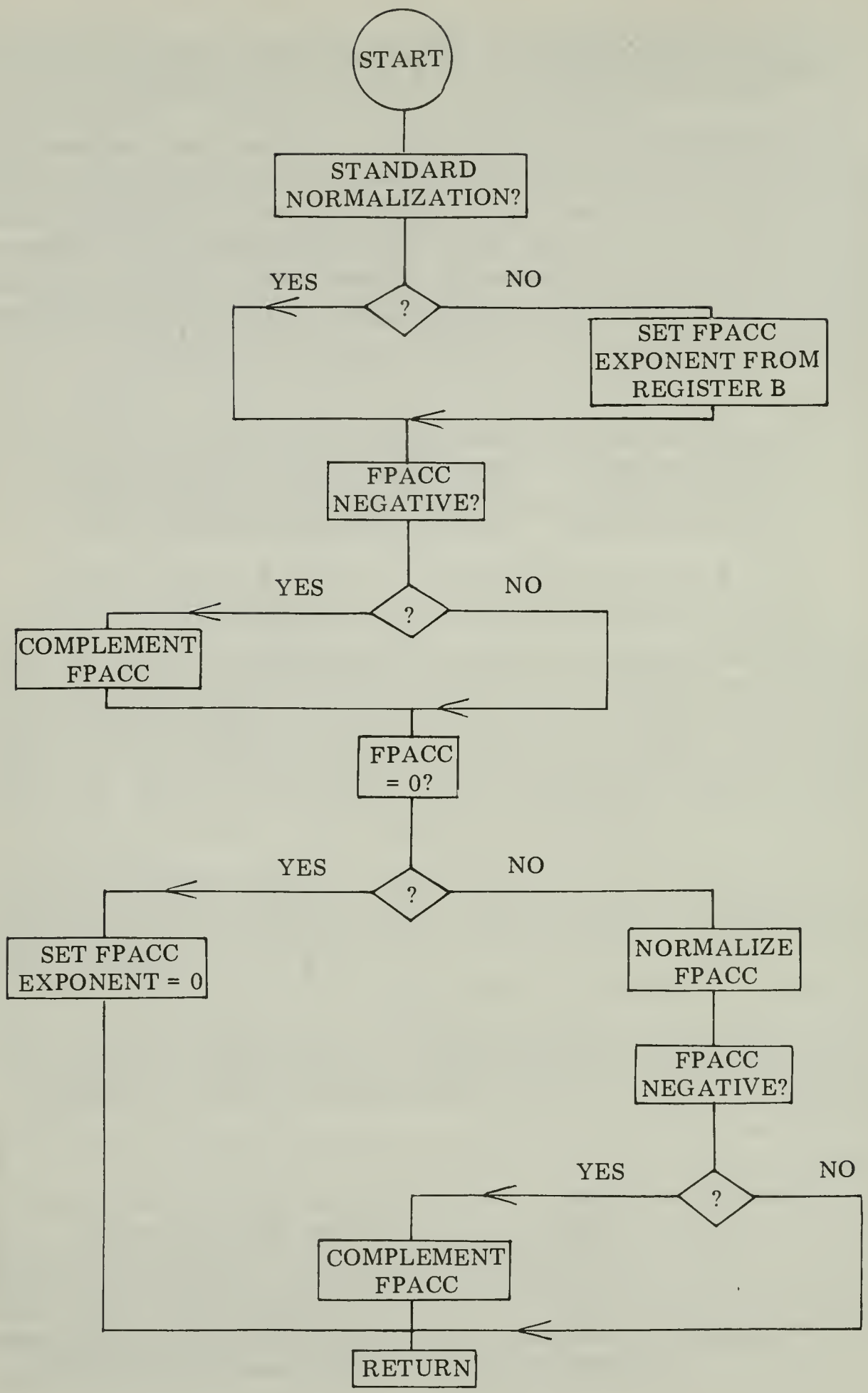
The normalization routine is written to operate with positive mantissa values. If the number to be normalized is negative, this routine will convert it to its two's complement form before normalizing, and then complement it again after the normalization. The following example illustrates the process for normalizing the value -5, as converted by the decimal input routine. Note that in the final result, the MSB of the mantissa is a '1,' indicating a negative number.

INITIAL VALUE	11111111 11111111 11111011
COMPLEMENTED	00000000 00000000 00000101 E+27
NORMALIZED	0.1010000 00000000 00000000 E+3
COMPLEMENTED	1.0110000 00000000 00000000 E+3

One special test that must be made by this routine is to check for an initial mantissa value of zero. If the mantissa is initially all zeros, and the normalization routine is allowed to perform its normal sequence, it would become caught in an endless loop looking for the first '1' bit. Therefore, to eliminate this possibility, the FPACC mantissa is initially checked for a value of zero and, if found, the FPACC exponent is zeroed and the routine returns.

The listing and flow chart for the normalization routine are presented next. It should be noted that the routine uses four memory locations for the mantissa in the initial stages of the process. This is necessary to handle some special cases that occur in the multiplication routine that require the additional precision. For the routines that do not use the additional byte, the least significant byte-1 of the mantissa must be set to zero before calling the FPNORM routine.

```
FPNORM,   LAB          Check register B for special case
          NDA          Set flags after load operation
          JTZ NOEXCO   If B was '0' do standard normalization
          LLI 127      Otherwise set EXPONENT of FPACC
          LMB          To value found in B at start of routine
NOEXCO,   LLI 126      Pntr to MS Byte FPACC MANTISSA
          LAM          Get MS Byte FPACC MANTISSA
```


START
STANDARD NORMALIZATION?
YES
?
NO
SET FPACC EXPONENT FROM REGISTER B
FPACC NEGATIVE?
YES
?
NO
COMPLEMENT FPACC
FPACC = 0?
YES
?
NO
SET FPACC EXPONENT = 0
NORMALIZE FPACC
FPACC NEGATIVE?
YES
?
NO
COMPLEMENT FPACC
RETURN

```
            LLI 100         Change pointer to SIGN storage adrs
            NDA             Set flags after previous LAM operation
            JTS ACCMIN      If MSB = 1, then minus number
            XRA             If MSB = 0, then positive number
            LMA             Set SIGN storage to 000 value
            JMP ACZERT      Proceed to see if FPACC = zero
ACCMIN,     LMA             Orig. FPACC negative, set SIGN
            LBI 004         Set precision cntr (using extra word)
            LLI 123         Pointer to FPACC (using extra word)
            CAL COMPLM      Two's comp FPACC (using extra word)
ACZERT,     LLI 126         Check to see if FPACC contains zero
            LBI 004         Set a counter
LOOK0,      LAM             Get a part of FPACC
            NDA             Set flags after load operation
            JFZ ACNONZ      If find anything, FPACC is not zero
            DCL             Otherwise move pntr to next part
            DCB             Decrement the loop counter
            JFZ LOOK0       If not finished check next part
            LLI 127         If reach here, FPACC was zero
            XRA             Make sure EXPONENT of FPACC =0
            LMA             By placing zero in it
            RET             Can then exit NORMALIZATION rtn
ACNONZ,     LLI 123         If FPACC has value, set up pntr and
            LBI 004         Precision vl (4 to handle special cases)
            CAL ROTATL      Rotate FPACC to the LEFT
            LAM             Get MSB from MANTISSA
            NDA             Set flags after load operation
            JTS ACCSET      If MSB=1, have found MSB in FPACC
            INL             If not, adv pntr to FPACC EXPON
            DCM             And decrement value of EXPONENT
            JMP ACNONZ      Continue in the rotating left loop
ACCSET,     LLI 126         Compensate for last rotate when MSB
            LBI 003         Found leaving room for SIGN IN MSB
            CAL ROTATR      FPACC MNTSA by rotating right once
            LLI 100         Set pointer to original SIGN storage
            LAM             Get original SIGN indicator value
            NDA             Set flags after load operation
            RFS             Finished if value in FPACC is positive
            LLI 124         Orig SIGN negative, so set pointer to
            LBI 003         LSByte of FPACC - also precision cntr
            CAL COMPLM      Two's comp NORMALIZED FPACC
            RET             End of NORMALIZATION routine
```

In studying the listing, one may have noted that several of the subroutines of chapter three are used in this routine. These routines are the ROTATL, ROTATR, and COMPLM subroutines. Throughout the

remainder of the floating point routines, these and other subroutines, such as MOVEIT, CLRMEM, ADDER, and SUBBER, will be called upon to perform their various functions.

The next routine to be discussed is the floating point addition routine. The basic function of this routine is carried out by the ADDER subroutine. However, there are a number of conditions that must be considered before the actual addition is performed.

First, the FPACC and the FPOP are tested for a value of zero. If both values are zero, or only the FPOP is zero, the routine can be exited immediately since the answer is already in the FPACC. (Remember, the result of all floating point operations is returned in the FPACC!) If the FPACC is zero, the contents of the FPOP are transferred to the FPACC before returning.

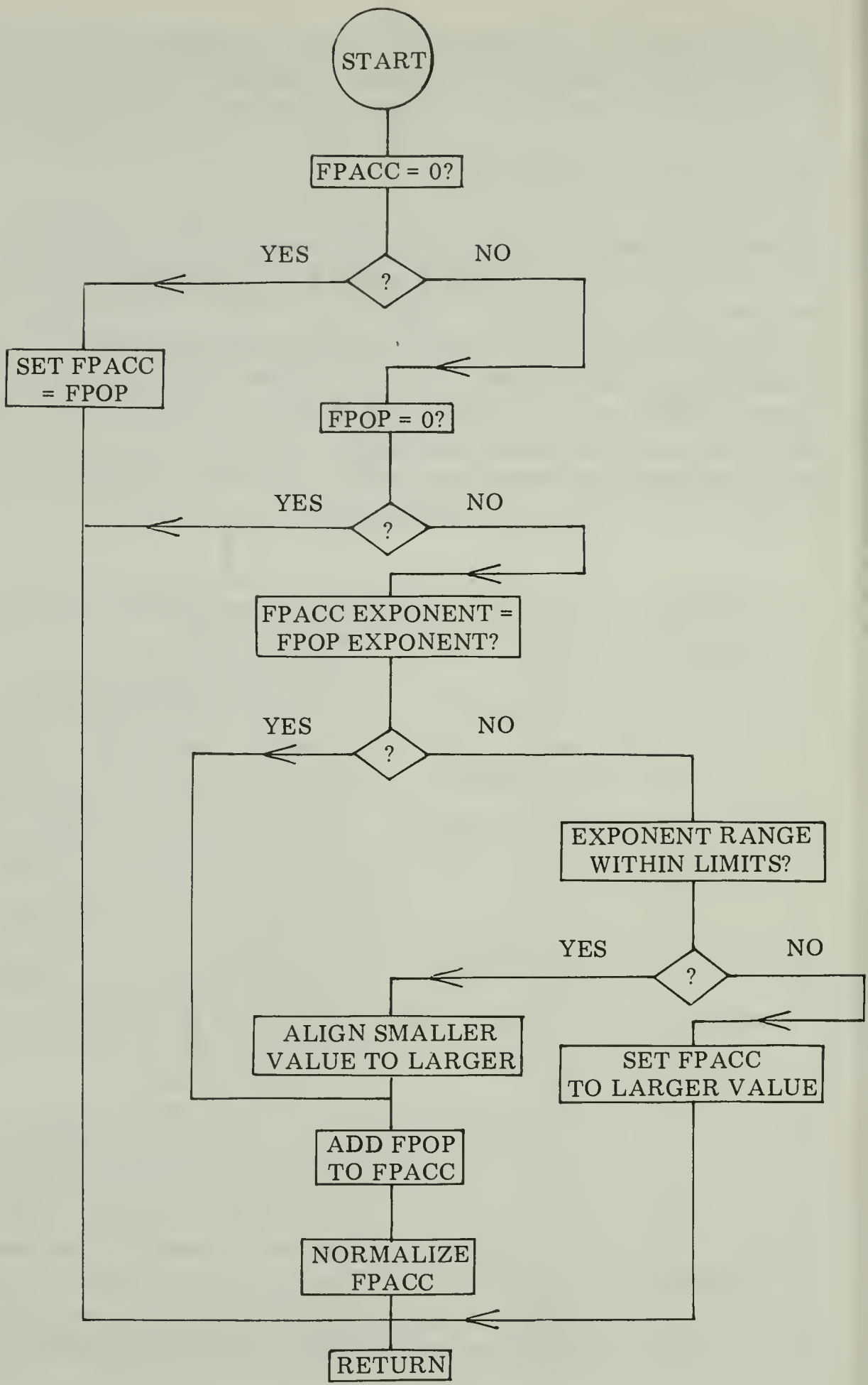
Should both numbers contain values other than zero, as is most likely the case when FPADD is called, the relative magnitude of one number to the other must be compared. With both numbers expressed in floating point notation, the range of values can vary quite a bit. For the addition routine, there is a limit in which the relative magnitude of the two numbers must fall. If one value is so much larger than the other that the significant digits of the smaller are outside the range of the significant digits of the larger, the addition would result in no change in the larger number. The answer would simply be equal to the larger number. This range is equal to the number of bits used to represent the value of the mantissa. For the floating point format used by these routines, the allowable limit on the difference between the two exponents is 27 (octal). If the difference is greater than 27, the number of greater magnitude is returned in the FPACC as the answer.

Assuming that the two numbers fall within the allowable range, the mantissas must be properly aligned before the addition can be executed. The two numbers are aligned when the exponents of each are equal. This alignment is made by shifting the mantissa of the smaller value to the right, while incrementing its exponent until the exponent is equal to the exponent of the larger. Of course, if the exponents are equal at the start, this is not necessary. The only special consideration in this procedure is when the mantissa being

shifted is negative. In this case, a '1' must be shifted into the MSB of the mantissa to maintain the negative condition. This is accomplished by setting the carry flag and calling the second entry point, ROTR, of the ROTATR subroutine, which will not clear the carry flag at the start of the rotate operation.

The final operation before the addition is performed is to shift both the FPACC and the FPOP one bit to the right to maintain any overflow from the addition within the FPACC. This eliminates the necessity of checking the carry flag for an overflow condition at the end of the operation. Also, so that the least significant digits of the aligned FPACC and FPOP are not lost as a result of this shifting operation, quad-precision is used for both the shifting and the addition. The result is normalized before returning. The flow chart and listing for the FPADD are presented next.

```
FPADD,    LLI 126        Set pointer to MS Byte of FPACC
          LBI 003        Set loop counter
CKZACC,   LAM            Fetch part of FPACC
          NDA            Set flags after loading operation
          JFZ NONZAC     Finding anything means FPACC not 0
          DCB            If part = 0, decrement loop counter
          JTZ MOVOP      If FPACC = 0, move FPOP to FPACC
          DCL            Not finished checking, decrement pntr
          JMP CKZACC     And test next part of FPACC
MOVOP,    XCHG           Save pointer to LS Byte of FPACC
          LHD            Set H equal to zero for sure
          LLI 134        Set pointer to LS Byte of FPOP
          LBI 004        Set loop counter
          CAL MOVEIT     Move FPOP into FPACC as answer
          RET            Exit FPADD subroutine
NONZAC,   LLI 136        Set pointer to MS Byte of FPOP
          LBI 003        Set loop counter
CKZOP,    LAM            Get MS Byte of FPOP
          NDA            Set flags after load operation
          JFZ CKEQEX     If not zero then have a number
          DCB            If zero, decrement loop counter
          RTZ            Exit subroutine if FPOP equals zero
          DCL            Else decr pntr to next part of FPOP
          JMP CKZOP      And continue testing for zero FPOP
CKEQEX,   LLI 127        Check for equal exponents
          LAM            Get FPACC exponent
          LLI 137        Change pointer to FPOP exponent
          CPM            Compare exponents
```


START
FPACC = 0?
YES
?
NO
SET FPACC = FPOP
FPOP = 0?
YES
?
NO
FPACC EXPONENT = FPOP EXPONENT?
YES
?
NO
EXPONENT RANGE WITHIN LIMITS?
YES
?
NO
ALIGN SMALLER VALUE TO LARGER
SET FPACC TO LARGER VALUE
ADD FPOP TO FPACC
NORMALIZE FPACC
RETURN

```
                JTZ SHACOP      If same can set up add operation
                CMA             If not the same, two's complement
                INA             The value of the FPACC exponent
                ADM             And add to FPOP exponent
                JFS SKPNEG      If positive, go to alignment test
                CMA             If negative perform two's complement
                INA             On the results
SKPNEG,         CPI 030         See if greater than 27 octal
                JTS LINEUP      If not, can perform alignment
                LAM             If not alignable, get FPOP exponent
                LLI 127         Set pointer to FPACC exponent
                SUM             Subtract FPACC exp from FPOP exp
                RTS             FPACC expo greater so just exit rtn
                LLI 124         FPOP grtr, set pntr to FPACC LSByte
                JMP MOVOP       Put FPOP into FPACC & exit routine
LINEUP,         LAM             Align FPACC & FPOP, get FPOP exp
                LLI 127         Change pointer to FPACC exponent
                SUM             Subtract FPACC exp from FPOP exp
                JTS SHIFTO      FPACC greater so go to shift operand
                LCA             FPOP greater so save difference
MORACC,         LLI 127         Pointer to FPACC exponent
                CAL SHLOOP      Call shift loop subroutine
                DCC             Decrement difference counter
                JFZ MORACC      Continue aligning, if not done
                JMP SHACOP      Set up for add operation
SHIFTO,         LCA             Shift FPOP rtn, save diff cnt (negative)
MOROP,          LLI 137         Set pointer to FPOP exponent
                CAL SHLOOP      Call shift loop subroutine
                INC             Increment difference counter
                JFZ MOROP       Shift again if not done
SHACOP,         LLI 123         First clear out extra room, set up pntr
                LMI 000         To FPACC LS Byte-1 and set to zero
                LLI 133         Set up pointer to
                LMI 000         FPOP LS Byte-1 and set to zero
                LLI 127         Prepare to shift FPACC right once
                CAL SHLOOP      Set pointer, then call shift loop rtn
                LLI 137         Shift FPOP right once, 1st set pntr
                CAL SHLOOP      Call shift loop subroutine
                LDH             Set up pointers, set D = 0 for sure
                LEI 123         Pointer to LS Byte of FPACC
                XCHG            Exchange pointers for ADDER rtn
                LBI 004         Set precision counter
                CAL ADDER       Add FPACC to FPOP using quad-prsn
                LBI 000         Set B for standard normalization
                CAL FPNORM      Normalize the results of the addition
                RET             Exit FPADD rtn w/results in FPACC
SHLOOP,         INM             Shifting loop for alignment, incr exp
                DCL             Decrement the pointer
```

```
            LBI 004         Set precision counter
FSHIFT,     LAM             Get MS Byte of floating point number
            NDA             Set flags after load operation
            JTS BRING1      If number is minus, need to shift in a 1
            CAL ROTATR      Otherwise, perform N'th prsn rotate
            RET             Exit FSHIFT subroutine
BRING1,     RAL             Save '1' in carry bit
            CAL ROTR        Do rotate right w/o clearing carry bit
            RET             Exit FSHIFT subroutine
```

Floating point subtraction may be derived by simply forming the two's complement of the value contained in the FPACC and then jumping to the FPADD routine, as the following FPSUB routine illustrates.

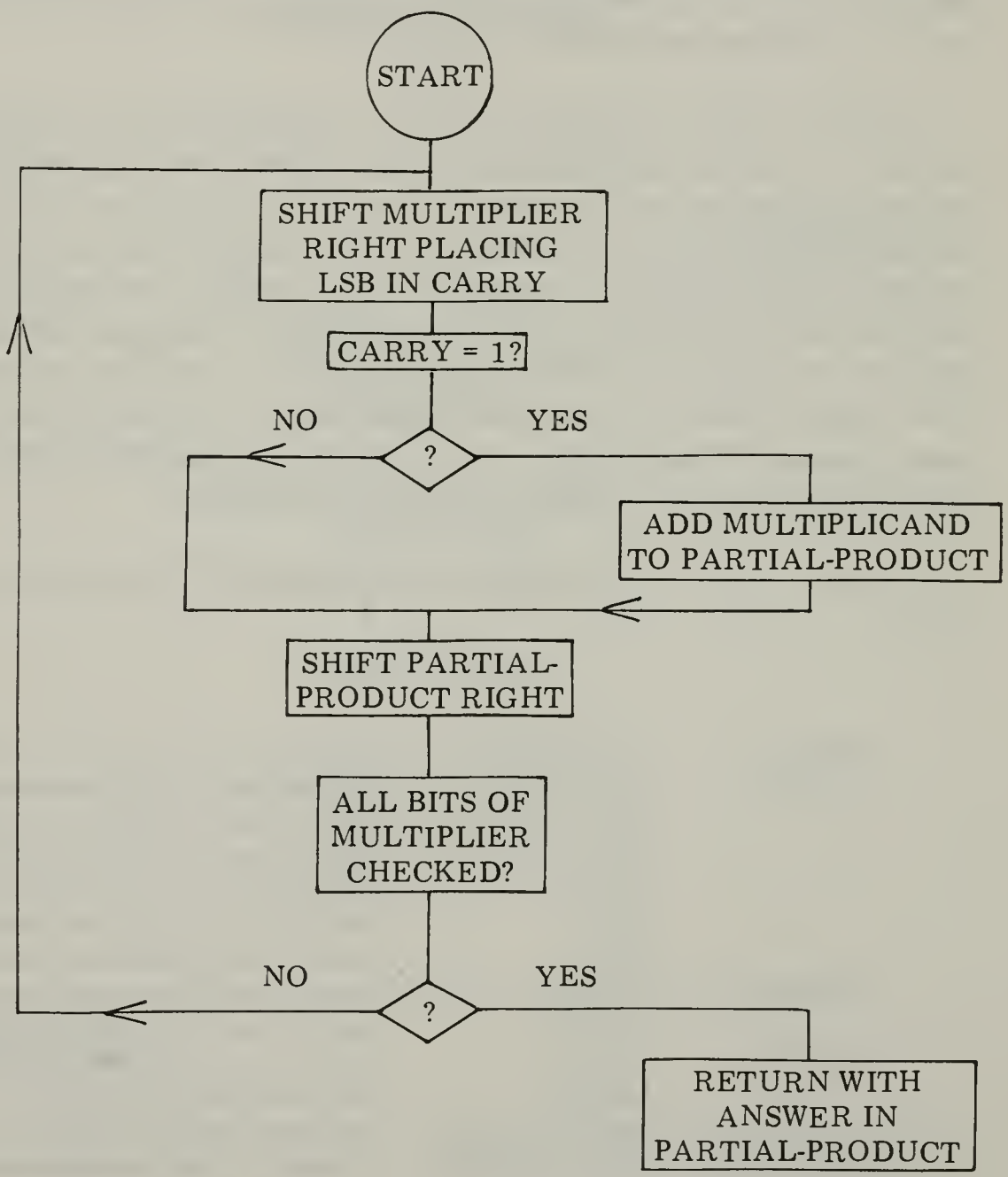
```
FPSUB,      LLI 124         Set pointer to LS Byte of FPACC
            LBI 003         Set precision counter
            CAL COMPLM      Perform two's complement on FPACC
            JMP FPADD       Sub accomplished now by adding!
```

Floating point multiplication is essentially carried out by a series of shifting and addition operations. As presented previously, a binary number is multiplied by two by simply shifting it one bit position to the left. By utilizing this fact with the proper addition function, one can create a multiplication algorithm for multiple precision binary numbers. This algorithm would operate in the following manner.

The two numbers to be multiplied shall be referred to as the multiplier and the multiplicand. A third register, called the partial-product, shall be used to store the product as it is being calculated. First, examine the LSB of the multiplier. If it is a '1,' add the multiplicand to the partial-product register. After the addition, or if the LSB was zero, shift the multiplicand to the left one bit position (multiplying it by two). Examine the bit to the left of the LSB of the multiplier and, if it is a '1,' add the current value of the multiplicand to the partial-product. Then, shift the multiplicand to the left again. The process continues for each bit of the multiplier, working up to the MSB. Each time the multiplier bit is equal to '1,' the current multiplicand is added to the partial-product. The multiplicand is always shifted left following the examination of each bit of the multiplier (and addition to the partial-product if the bit is '1'). The result of the

multiplication is contained in the partial-product register when the operation is complete.

The algorithm just described performs multiplication of standard binary numbers. Using this basic procedure, a multiplication algorithm for the mantissa in floating point format can be written. The following flow chart illustrates the process to be used to multiply the floating point values. The only major difference between the algorithm above and the process used by this floating point multiplication routine is that the partial-product is shifted right for each bit examined, rather than shifting the multiplicand to the left.



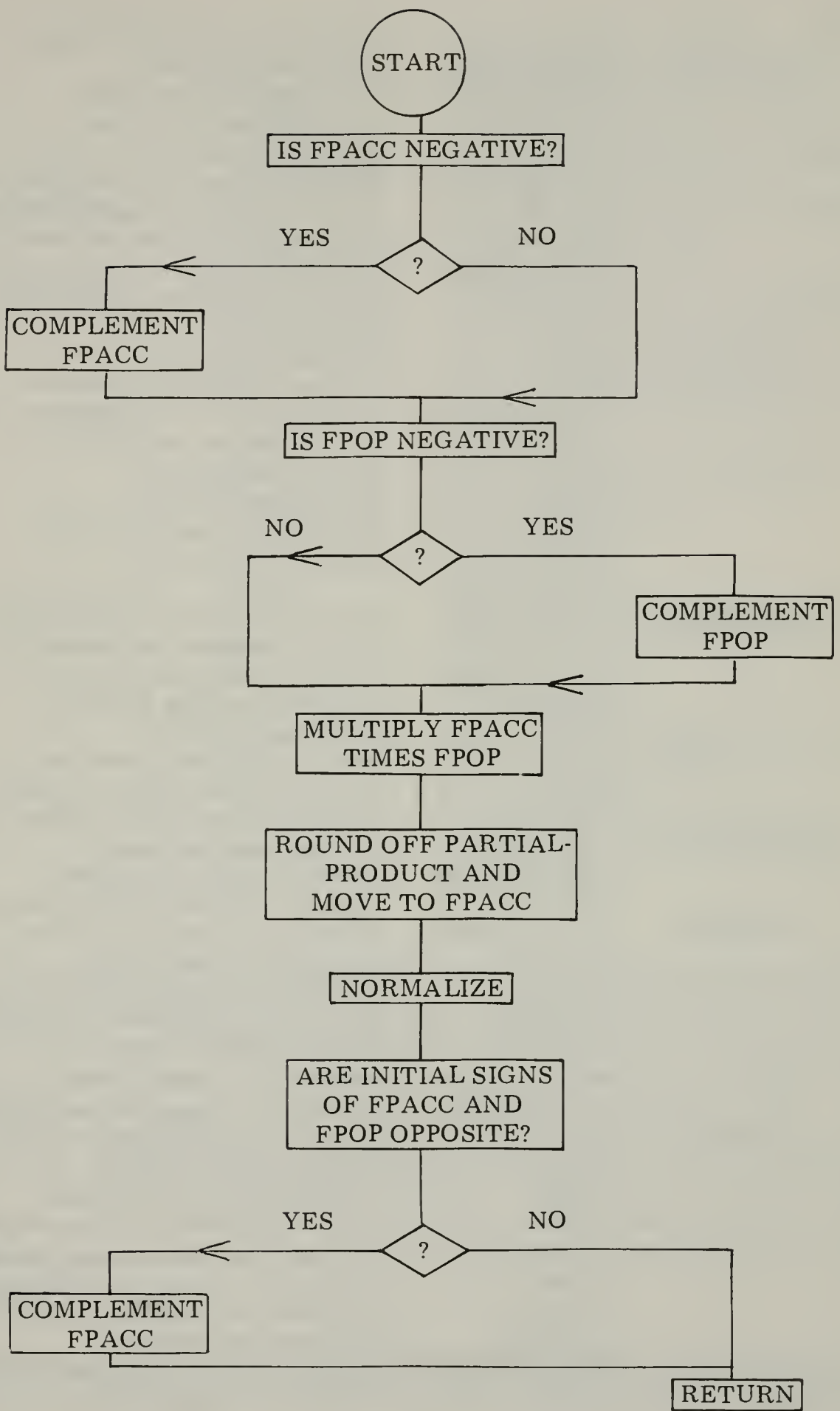
The exponent portion of the binary floating point numbers is manipulated in the same manner as the exponent of decimal floating point numbers for multiplication. They are simply added together.

The signs of the mantissa of both the multiplier and the multiplicand must be examined before the multiplication is executed. Since the multiplication algorithm only works for positive numbers, if either value is negative it must be two's complemented before multiplying. Also, following the laws of multiplication, if the two values are the same sign, the result will be positive; if the signs are opposite, the result will be negative. This condition must be tested at the beginning, and, if the result is to be negative, the final value must be two's complemented before returning.

It should also be noted that if the partial-product is rotated right for each bit of the multiplier, it is necessary for the partial-product register to contain twice as many bits as the multiplier. Although the partial-product register contains more precision than the program is designed to handle, it is necessary to maintain the required significant bits for the answer. At the completion of the multiplication algorithm, the 24th bit of the partial-product is used to round off the final result, which is then normalized to the proper 23 bit floating point format. This manner of handling the partial-product allows maximum precision for the multiplication routine.

The flow chart and listing for the FPMULT routine is now presented.

```
FPMULT,   CAL CKSIGN   Set up rtn & check sign of numbers
ADDEXP,   LLI 137      Set pointer to FPOP exponent
          LAM          Fetch FPOP exponent into ACC
          LLI 127      Set pointer to FPACC exponent
          ADM          Add FPACC exp to FPOP exp
          INA          Add one for algorithm compensation
          LMA          Store result in FPACC exponent
SETMCT,   LLI 102      Set bit counter storage pointer
          LMI 027      Set bit cntr to 23 decimal (27 octal)
MULTIP,   LLI 126      Set pointer to MS Byte FPACC
          LBI 003      Set precision counter
          CAL ROTATR   Rotate multiplier right into carry flag
          CTC ADOPPP   If carry = 1, add multiplicand to PP
```


START
IS FPACC NEGATIVE?
YES
NO
?
COMPLEMENT FPACC
IS FPOP NEGATIVE?
NO
YES
?
COMPLEMENT FPOP
MULTIPLY FPACC TIMES FPOP
ROUND OFF PARTIAL-PRODUCT AND MOVE TO FPACC
NORMALIZE
ARE INITIAL SIGNS OF FPACC AND FPOP OPPOSITE?
YES
NO
?
COMPLEMENT FPACC
RETURN

```
                LLI 146         Set pointer to PP MS Byte
                LBI 006         Set precision counter
                CAL ROTATR      Shift partial product right
                LLI 102         Set pointer to bit counter
                DCM             Decrement value of bit counter
                JFZ MULTIP      If bit cntr is not 0, repeat algorithm
                LLI 146         Set pntr to partial product MS Byte
                LBI 006         Set precision cntr, rotate partial prod
                CAL ROTATR      Once more to make rm for rounding
                LLI 143         Set pntr to 24th bit in partial-prod
                LAM             Fetch 24th bit
                RAL             Position MSB portion
                NDA             Set flags after rotate operation
                CTS MROUND      If 24th bit is '1' do rounding process
                LLI 123         Now set pointer to FPACC
                XCHG            Save FPACC pointer
                LHD             Ensure that H is 000
                LLI 143         Set pointer to partial product
                LBI 004         Set precision counter

EXMLDV,         CAL MOVEIT      Move answer from PP into FPACC
                LBI 000         Set B for standard normalization
                CAL FPNORM      Normalize the answer
                LLI 101         Set pointer to sign indicator
                LAM             Fetch sign indicator
                NDA             Set flags after load operation
                RFZ             If sign has value, result +, exit rtn
                LLI 124         If sign zero, set FPACC LSByte pntr
                LBI 003         And set precision counter
                CAL COMPLM      And complement the answer
                RET             Before exiting the FPMULT routine
CKSIGN,         CAL CLRWRK      Clear work locations for multip
                LLI 101         Set pointer to sign storage
                LMI 001         Place initial value of 1 in sign storage
                LLI 126         Set pointer to MS Byte of FPACC
                LAM             Fetch MS Byte of FPACC
                NDA             Set flags after load operation
                JTS NEGFPA      If number is minus, do two's comp
OPSGNT,         LLI 136         Set pointer to MS Byte of FPOP
                LAM             Fetch MS Byte of FPOP
                NDA             Set flags after load operation
                RFS             If positive, return to calling routine
                LLI 101         If minus, set pointer to sign storage
                DCM             Decrement value of sign indicator
                LLI 134         Set pointer to LS Byte of FPOP
                LBI 003         Set precision counter
                CAL COMPLM      Perform two's complement of FPOP
                RET             Go back to calling routine
```

```
NEGFPA,   LLI 101      Set pointer to sign storage
          DCM          Decrement value of sign indicator
          LLI 124      Set pointer to LS Byte of FPACC
          LBI 003      Set precision counter
          CAL COMPLM   Complement the value in the FPACC
          JMP OPSGNT   Go check sign of FPOP
CLRWRK,   LLI 140      Clear partial prod work area (140-147)
          LBI 010      Set pointer and counter
          CAL CLRMEM   Go clear PP area
CLROPL,   LBI 004      Clear additional room for multiplicand
          LLI 130      At 130 to 133, first set counter & pntr
          CAL CLRMEM   Go clear room for multiplicand
          RET          Return to calling program when done
ADOPPP,   LLI 141      Pointer to LS Byte of partial product
          LDH          On page 00
          LEI 131      Pointer to LS Byte of multiplicand
          LBI 006      Set precision counter
          CAL ADDER    Perform addition
          RET          Exit subroutine
MROUND,   LBI 003      Set precision counter
          LAI 100      Add 1 to 23rd bit of partial product
          ADM          Here
CROUND,   LMA          Restore to memory
          INL          Advance pointer
          LAI 000      Clr ACC without disturbing carry flag
          ACM          And propogate rounding
          DCB          In partial product
          JFZ CROUND   Finished when counter equals zero
          LMA          Restore last byte of partial product
          RET          Exit subroutine
```

The procedure for division can almost be considered the reverse of that for multiplication. The division algorithm consists of a series of subtraction and shifting operations. This algorithm is illustrated in the following flow chart. This algorithm is written for division of numbers in floating point format rather than straight binary. For operating with numbers in standard binary format, the most significant bits of the divisor and dividend would have to be properly aligned, and the location of the binary point in the quotient would have to be accounted for in cases where the result is not a pure integer.

Rather than verbalize the operation as presented in the flow chart, a sample division of two floating point numbers using this algorithm

will be presented in a step-by-step fashion. This illustration will divide the binary equivalent of the value 15 (decimal) by 5. The numbers are presented as 4 bit values to keep the illustration short. However, in the FPDIV routine, the operation is carried out 23 times for each significant bit of the mantissa of the dividend. Once again, this algorithm assumes the numbers are in normalized floating point format.

```
0.1111      Original DIVIDEND at start of routine.

0.1010      DIVISOR (Note floating point format.)
---------
0.0101      This is the REMAINDER from the subtraction
            operation. Since the result was POSITIVE, a '1'
            is placed in the LSB of the QUOTIENT register.

            0.0001  QUOTIENT after 1'st loop.
```

NOW BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

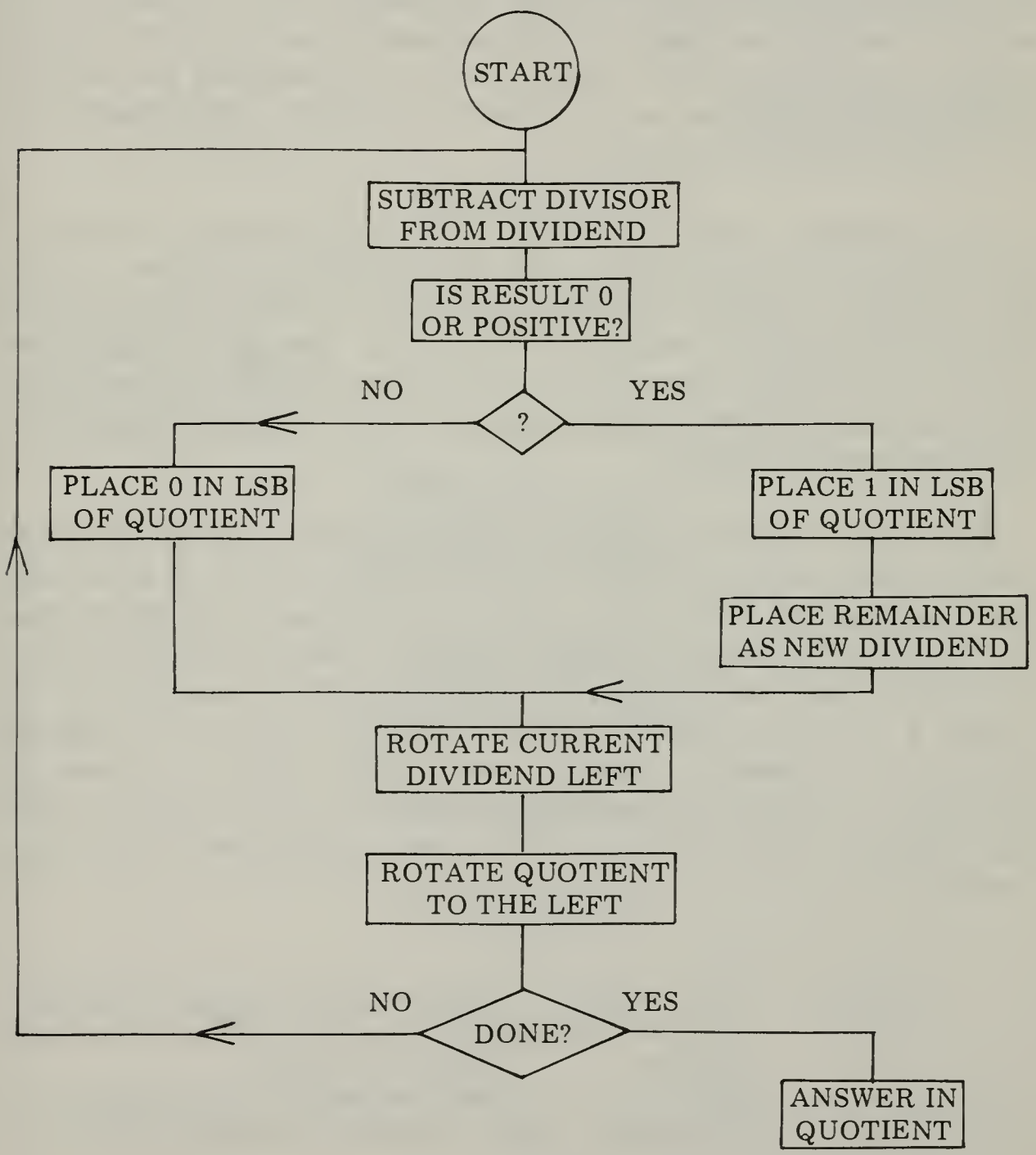
```
0.1010      New DIVIDEND (which is the previous REMAIN-
            DER rotated once to the LEFT.)
0.1010      DIVISOR (Does not change during routine).
---------
0.0000      RESULT of this subtraction is zero and thus
            qualifies to become a NEW DIVIDEND. Also,
            QUOTIENT LSB gets a '1' for this case!

            0.0011  QUOTIENT after 2'nd loop.
```

AGAIN BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

```
0.0000      New DIVIDEND (which is the last remainder
            rotated once to the left).
0.1010      DIVISOR (still same old number).
---------
1.0110      RESULT of this subtraction is a minus number
            (note that the SIGN bit changed). Thus, old
            DIVIDEND stays in place and QUOTIENT
            gets a '0' in LSB!

            0.0110  QUOTIENT after 3'rd loop
```

NOW BOTH QUOTIENT, AND IN THIS CASE, THE OLD DIVIDEND, ARE ROTATED LEFT

0 . 0 0 0 0	Old DIVIDEND rotated once to the left.
0 . 1 0 1 0	Same old DIVISOR
- - - - - - - - -	
1 . 0 1 1 0	RESULT of this subtraction is again a minus. Old DIVIDEND stays in place. QUOTIENT gets another '0' in LSB
0 . 1 1 0 0	QUOTIENT after 4'th loop

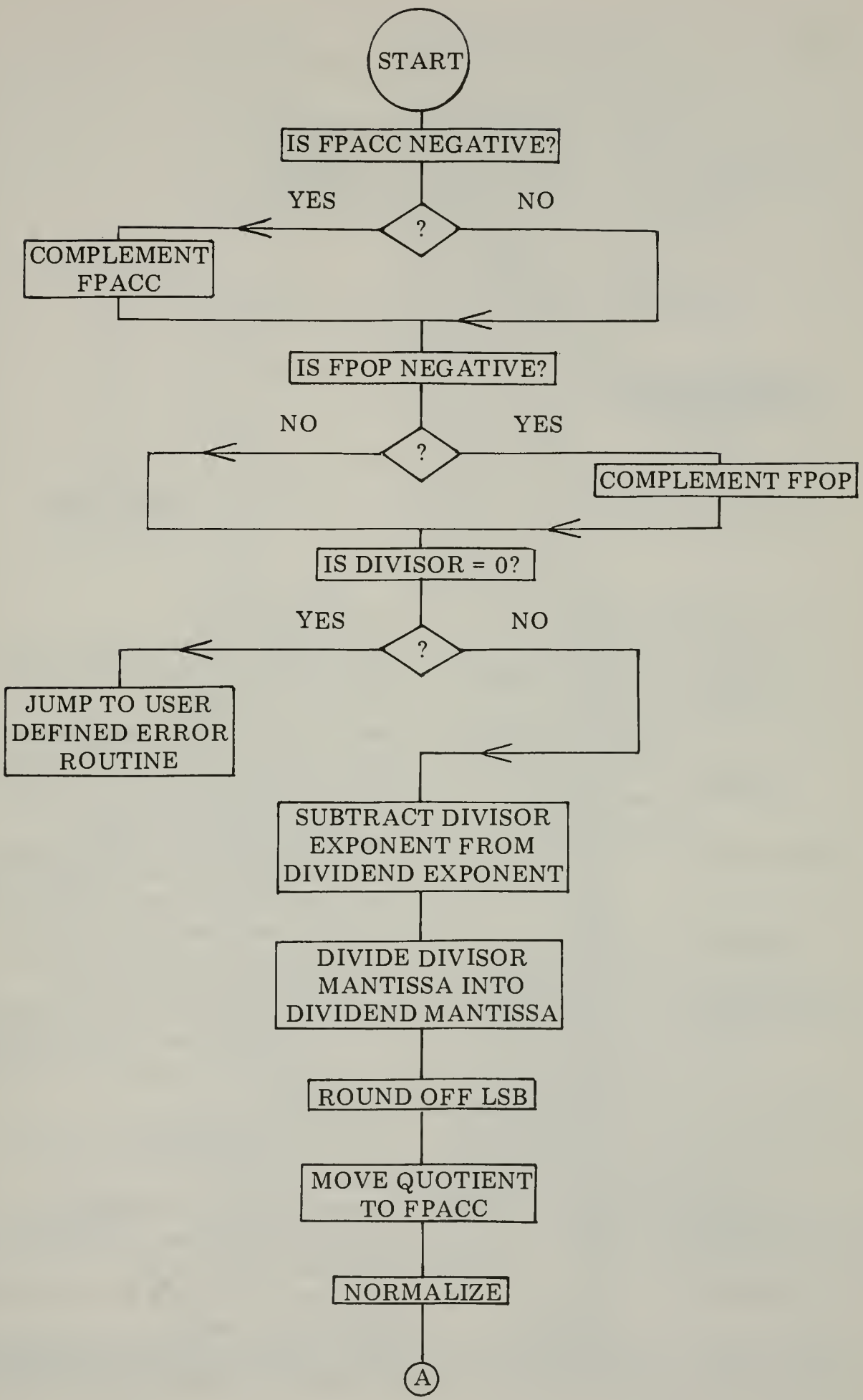


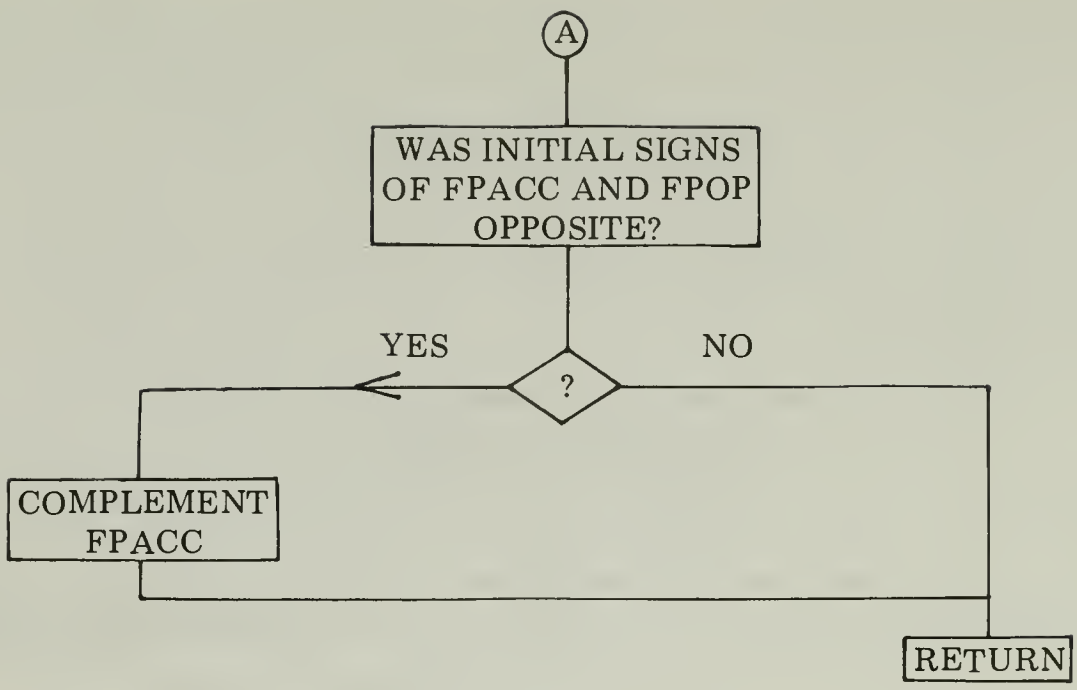
With only four significant bits in the dividend, the calculation illustrated ends after the fourth loop. The answer is contained in the quotient. The exponents are the next quantity that must be dealt with, since the values are represented in floating point notation. Just as in division of decimal floating point numbers, the exponents of the binary counterparts are subtracted, DIVIDEND exponent minus the DIVISOR exponent. In the example given, the dividend would have an exponent of four for the normalized binary value of 15 (decimal) and the divisor would have a binary exponent of three. The algorithm as presented requires a compensation factor of +1 after subtracting the exponents in order to have the correct floating point result. Thus, the exponent of the quotient in the previous example would be (4-3)+1=2. This can be verified by moving the implied binary point in the quotient two places to the right - the binary value of 3 would indeed be observed.

In the division algorithm, just as in the multiplication, the sign of the dividend and divisor must be positive for the algorithm to operate properly. If either is negative, it must be two's complemented before the division is performed. Also, if the signs are the same, the sign of the quotient must be positive. If the signs are opposite, the quotient must be two's complemented before exiting the routine to make the answer negative.

The FPDIV routine is listed next along with the flow chart. As one examines the listing, one should observe that two other conditions are considered by the routine. If the quotient has a remainder after the final loop through the divide algorithm, which would result in a '1' in the 24th bit position, the quotient will be rounded off by adding a '1' to the 23rd bit. Also, if a divide by zero is attempted (which is an illegal operation), the FPDIV routine is exited by jumping to a special user provided error routine. This routine should take whatever action the user determines appropriate for this error condition.

```
FPDIV,    CAL CKSIGN    Set up registers & ck sign of numbers
          LLI 126       Pointer to MSByte of FPACC (divisor)
          LAI 000       Clear accumulator
          CPM           See if MS Byte of FPACC is zero
          JFZ SUBEXP    If find anything proceed to divide
```


START
IS FPACC NEGATIVE?
YES
NO
?
COMPLEMENT FPACC
IS FPOP NEGATIVE?
NO
YES
?
COMPLEMENT FPOP
IS DIVISOR = 0?
YES
NO
?
JUMP TO USER DEFINED ERROR ROUTINE
SUBTRACT DIVISOR EXPONENT FROM DIVIDEND EXPONENT
DIVIDE DIVISOR MANTISSA INTO DIVIDEND MANTISSA
ROUND OFF LSB
MOVE QUOTIENT TO FPACC
NORMALIZE
A




```
            DCL         Decrement pointer
            CPM         See if MSByte of DIVISOR is zero
            JFZ SUBEXP  If find anything, proceed to divide
            DCL         Decrement pointer
            CPM         See if LSByte of DIVISOR is zero
            JTZ DERROR  If DIVISOR = 0, have error condition
SUBEXP,     LLI 137     Set pntr to DIVIDEND (FPOP) exp
            LAM         Fetch DIVIDEND exponent
            LLI 127     Set pntr to DIVISOR (FPACC) exp
            SUM         Sub DIVISOR exp fm DIVIDEND exp
            INA         Compensate for division algorithm
            LMA         Store exponent results in FPACC exp
SETDCT,     LLI 102     Set pointer to bit counter storage
            LMI 027     Set it to 27 octal (23 decimal)
DIVIDE,     CAL SETSUB  Subtract DIVISOR from DIVIDEND
            JTS NOGO    If result negative, put 0 in QUOTIENT
            LEI 134     If + or 0, move remain to DIVIDEND
            LLI 131     Set pointers
            LBI 003     And precision counter
            CAL MOVEIT  Move REMAINDER to DIVIDEND
            STC         Set carry flag
            JMP QUOROT  Rotate it into the QUOTIENT
NOGO,       STC         Set carry flag and then complement to
            CMC         Reset carry flag to zero
QUOROT,     LLI 144     Set pointer to LS Byte of QUOTIENT
            LBI 003     Set precision counter
            CAL ROTL    Move carry bit to LSB of QUOTIENT
            LLI 134     Set pointer to LSByte of DIVIDEND
```

```
                LBI 003         Set precision counter
                CAL ROTATL      Rotate DIVIDEND left
                LLI 102         Set pointer to bits counter
                DCM             Decrement bits counter
                JFZ DIVIDE      If not finished, continue algorithm
                CAL SETSUB      Do 1 more divide for rounding oper
                JTS DVEXIT      If 24th bit = 0, no rounding
                LLI 144         When 24th bit = 1, set pntr to
                                QUOTIENT LS Byte
                LAM             Fetch LS Byte of QUOTIENT
                ADI 001         Add 1 to 23rd bit
                LMA             Restore to LS Byte
                LAI 000         Clear ACC while saving carry flag
                INL             Adv pntr to NS Byte of QUOTIENT
                ACM             Add with carry
                LMA             Restore NS Byte
                LAI 000         Clear ACC while saving carry flag
                INL             Adv pntr to MS Byte of QUOTIENT
                ACM             Add with carry
                LMA             Restore MS Byte
                JFS DVEXIT      If MSB of MSByte =0, prepare to exit
                LBI 003         Otherwise set precision counter
                CAL ROTATR      Move quotient right to clear sign bit
                LLI 127         Set pointer to FPACC exponent
                INM             Increment exponent for rotate right
DVEXIT,         LLI 143         Set pointers to transfer
                LEI 123         QUOTIENT to FPACC
                LBI 004         Set precision counter
                JMP EXMLDV      Exit routine at EXMLDV
SETSUB,         LLI 131         Set pntr to LSByte of working register
                XCHG            Save pointer
                LHD             Set H = 0 for sure
                LLI 124         Set pointer to LS Byte of FPACC
                LBI 003         Set precision counter
                CAL MOVEIT      Move FPACC value to working register
                LEI 134         Set pointer to LSByte of FPOP (dividend)
                LLI 131         Reset pntr to LS Byte (divisor)
                LBI 003         Set precision counter
                CAL SUBBER      Subtract DIVISOR from DIVIDEND
                LAM             Get MSByte of result from subtraction
                NDA             And set flags after load operation
                RET             Before return to calling routine
DERROR,         CAL DERMSG  ** User defined error routine for handling
                JMP USERDF      Attempted divide by zero, exit as
                                directed
```

The floating point routines presented to this point, when assem-

bled into the object code will reside in approximately 2½ pages of memory. Additional memory is required for the data areas on page 00 which are used to store various counters and data values. The routines may be easily modified to change the page on which these data areas reside. The locations used on page 00 are listed in the following table for reference purposes.

100	SIGN Indicator
101	SIGNS Indicator (Multiply & Divide)
102	Bits Counter
123	FPACC Extension
124	FPACC Least Significant Byte (LS Byte)
125	FPACC Next Significant Byte (NS Byte)
126	FPACC Most Significant Byte (MS Byte)
127	FPACC Exponent
130-133	Working Area
134	FPOP Least Significant Byte
135	FPOP Next Significant Byte
136	FPOP Most Significant Byte
137	FPOP Exponent
140-147	Working Area

The floating point routines just presented are extremely powerful routines that can be of considerable value to someone who requires such mathematical calculations on an 8080 based microcomputer. These routines provide the capability to handle binary numbers equivalent to six or seven significant decimal digits raised to plus or minus the 38th power of ten. Using these routines as a base, a wide variety of mathematic operations can be performed by loading FPACC and FPOP with the numbers in normalized floating point format and calling the proper routine. By developing programs that make a series of calculations in this manner, one can solve quite sophisticated equations, such as the expansion series for calculating the sine and cosine functions.

The range of values encompassed by these routines, as presented, may well be within the requirements of most applications. However,

should it be desired to increase the number of significant digits by expanding the precision of the mantissa, or to extend the exponent to a double or triple precision value, the interested programmer should have little difficulty in making the required modification to these routines.

As pointed out in the chapter on conversion routines, one of the most common requirements of a program that deals with binary numbers is the conversion to and from decimal. This is because it is most often necessary to communicate with a human operator. And, since it is inevitable that the operator is most familiar with expressing numbers as decimal values, the conversion must be made. Therefore, to illustrate a method for converting from floating point decimal to floating point binary, and then back to floating point decimal, and also to provide a complete floating point mathematical program, the following three routines are included.

The first of these routines performs the conversion of decimal floating point numbers to floating point binary. The overall requirement of this routine is to receive the decimal number in floating point format, normalize the mantissa portion to an all integer value, and convert to the equivalent floating point binary value.

Floating point decimal values may be expressed in various forms as indicated below.

123.45
or
1.2345 E+2

As either of these formats is received, the mantissa portion is converted to binary, while keeping track of the exponent to provide the proper normalized decimal value. Unlike the binary normalization, which shifts the binary point to the left of the MSB to provide a purely fractional mantissa, decimal normalization maintains the decimal point to the right of the least significant digit to provide a purely integer mantissa. Thus, the example above would be normalized to:

12345 E-2

The conversion of the decimal mantissa to binary is accomplished by the routine labeled DECBIN, which is a version of the TIMS10 subroutine presented in chapter four. As indicated in chapter four, this subroutine converts each digit by first multiplying the binary equivalent of the digits already received by ten and adding the BCD value of the latest digit input to the new binary value.

Once the mantissa is converted, the decimal exponent is input and converted to binary. At this point, it is necessary to normalize the mantissa of the binary equivalent by calling the FPNORM routine, with register B set to 27 (octal). Then, using the FPMULT routine, the normalized binary equivalent is multiplied by 10 (for each unit of a positive decimal exponent received), or by 0.1 (for each unit of a negative exponent received).

The input and output portions of this routine require that the user provide driver routines for the specific input and output devices associated with one's system. The requirement for the INPUT routine is to return to the calling routine with the ASCII code for the character entered on the input device, such as an electronic keyboard, in the accumulator. The routine to output characters to a display device, such as a mechanical printer or video display, must accept the character to be output as an ASCII character stored in the accumulator. This output routine, labeled ECHO, is called to "echo" the characters received from the input device back to the display device. The character is also saved in register B when ECHO is called, and, therefore, if ECHO requires the use of register B, its contents must be saved and then restored before returning. The reader may refer to the chapter on input and output routines for methods of creating these routines.

The decimal to binary input routine listing is presented next. One should note in the listing that both formats illustrated previously are allowed as legal entries, and the routine accounts for positive and negative mantissas and exponents. Also, the operator has the option to cancel the current input by entering a rubout character. Several locations on page 00 are used to store the input characters and save counters and indicators. These locations will be summarized later in this chapter.

```
FPINP,   LXH 150 000    Set pointer to input storage
         LBI 010        Set a counter
         CAL CLRMEM     Clear input registers
         LLI 103        Set pntr to counter/indicator storage
         LBI 004        Set a counter
         CAL CLRMEM     Clear counter/indicator storage
         CAL INPUT      Now bring in a char from I/O device
         CPI 253        Test to see if it is a + sign
         JTZ SECHO      If yes go to ECHO and continue
         CPI 255        If not +, see if it is a - sign
         JFZ NOTPLM     If not + or -, test for valid character
         LLI 103        If minus, set pntr to INPUT SIGN
         LMA            Make it non-zero by depositing char
SECHO,   CAL ECHO       Output character as echo to operator
NINPUT,  CAL INPUT      Fetch a new character from I/O device
NOTPLM,  CPI 377        See if character is code for rubout
         JTZ ERASE      If yes prepare to start over
         CPI 256        If not, see if character is a period (.)
         JTZ PERIOD     If (.) process as decimal point
         CPI 305        If not, see if char is E for exponent
         JTZ FNDEXP     If E process as exponent indicator
         CPI 260        If not, see if character is valid number
         JTS ENDINP     If none above, terminate input string
         CPI 272        Still checking for valid number
         JFS ENDINP     If not terminate input string
         LLI 156        Have nbr, set pntr MSByte of INPUT
         LBA            Save character in register B
         LAI 370        Form a mask and check to see if input
         NDM            Registers can accept large number
         JFZ NINPUT     If not, ignore present input
         LAB            If o.k., restore character to ACC
         CAL ECHO       And echo number back to operator
         LLI 105        Set pointer to digit counter
         INM            Increment value of digit counter
         CAL DECBIN     Perform decimal to binary conversion
         JMP NINPUT     Get next character for mantissa
PERIOD,  LBA            Subroutine to precess (.) - save in B
         LLI 106        Set pointer to (.) storage indicator
         LAM            Fetch contents
         NDA            Set flags after load operation
         JFZ ENDINP     If (.) already present, end input
         LLI 105        Otherwise set pointer to digit counter
         LMA            And reset digit counter to zero
         INL            Advance pointer back to (.) storage
         LMB            And put a (.) there
         LAB            Restore (.) to accumulator
         CAL ECHO       And echo it back to operator
         JMP NINPUT     Get next character in number string
```

```
ERASE,    LAI 274        Put ASCII code for < in ACC
          CAL ECHO       Display it
          CAL SPACES     Display a few spaces
          JMP FPINP      Start the input string over
FNDEXP,   CAL ECHO       Subroutine to process exp, echo E
          CAL INPUT      Get next part of exponent
          CPI 253        Test for a + sign
          JTZ EXECHO     If yes, proceed to echo it
          CPI 255        If not, test for a - sign
          JFZ NOEXPS     If not, see if a valid character
          LLI 104        If - , set pntr to EXPONENT SIGN
          LMA            Set EXPONENT SIGN minus indicator
EXECHO,   CAL ECHO       Echo character back to operator
EXPINP,   CAL INPUT      Get character for exponent portion
NOEXPS,   CPI 377        See if code is for rubout
          JTZ ERASE      If yes, prepare to reenter entire string
          CPI 260        Otherwise check for valid decimal no.
          JTS ENDINP     If not end input string
          CPI 272        Still testing for valid number
          JFS ENDINP     If not, end input string
          NDI 017        Valid no., form mask & strip ASCII
          LBA            Char is pure BCD, save in register B
          LLI 157        Set pointer to input exponent
          LAI 003        Set accumulator = 3
          CPM            See if 1st exp num greater than 3
          JTS EXPINP     Yes, ignore input (limits exp to < 40)
          LCM            If o.k., save previous exp value in reg C
          LAM            And also place it in the accumulator
          NDA            Clear the carry bit
          RAL            Multiply X 10 algorithm:1st mult X2
          RAL            Multiply by two again
          ADC            Add original value
          RAL            Multiply by two once more
          ADB            Add new num to complete decimal to
          LMA            Binary conversion for exponent
          LAI 260        Restore ASCII code by adding 260
          ADB            To BCD value of the number
          JMP EXECHO     And echo number, look for next input
ENDINP,   LLI 103        Set pntr to mantissa SIGN indicator
          LAM            Fetch SIGN indicator
          NDA            Set flags after load operation
          JTZ FININP     If nothing in indicator, number is +
          LLI 154        Set pntr to LSByte of input mantissa
          LBI 003        Set precision
          CAL COMPLM     Perform two's complement of number
FININP,   LLI 153        Set pointer to input storage LSByte -1
          XRA            Clear accumulator
          LDA            Clear register D
```

```
        LMA         Clear input storage location LSByte -1
        LEI 123     Set pointer to FPACC LSByte -1
        LBI 004     Set precision counter
        CAL MOVEIT  Move input to FPACC
        LBI 027     Set FPNORM mode by setting bit cnt
        CAL FPNORM  In B & call FPNORM routine
        LLI 104     Set pntr EXPONENT SIGN indicator
        LAM         Fetch EXPONENT SIGN indicator
        NDA         Set flags after load operation
        LLI 157     Set pntr to decimal exponent storage
        JTZ POSEXP  If exponent positive, jump ahead
        LAM         If exponent negative, fetch it to ACC
        CMA         And form two's
        INA         Complement
        LMA         Then restore to storage location
POSEXP, LLI 106     Set pntr to period indicator
        LAM         Fetch contents of accumulator
        NDA         Set flags after load operation
        JTZ EXPOK   If nothing, no decimal point involved
        LLI 105     If have decimal point, set pntr to digit
        XRA         Counter then clear accumulator
        SUM         Sub digit cntr from 0 to give negative
EXPOK,  LLI 157     Set pntr to decimal exponent storage
        ADM         Add in compensation for decimal pnt
        LMA         Restore compensated value to storage
        JTS MINEXP  If compensated value - , jump ahead
        RTZ         If compensated value 0, finished!
EXPFIX, CAL FPX10   Compenstated decimal exp is +, multip
        JFZ EXPFIX  FPACC X 10, loop till decimal exp =0
        RET         Exit with converted value in FPACC
FPX10,  LEI 134     Mult FPACC X 10 subrtn, set pntr to
        LDH         FPOP LSByte, set D = 0 for sure
        LLI 124     Set pntr to FPACC LS Byte
        LBI 004     Set precision counter
        CAL MOVEIT  Move FPACC to FPOP (include exp's)
        LLI 127     Set pointer to FPACC exponent
        LMI 004     Place FP form of 10 (dec.) in FPACC
        DCL         Place FP form of 10 (dec.) in FPACC
        LMI 120     Place FP form of 10 (dec.) in FPACC
        DCL         Place FP form of 10 (dec.) in FPACC
        XRA         Place FP form of 10 (dec.) in FPACC
        LMA         Place FP form of 10 (dec.) in FPACC
        DCL         Place FP form of 10 (dec.) in FPACC
        LMA         Place FP form of 10 (dec.) in FPACC
        CAL FPMULT  Multiply orig binary no (in FPOP) X10
        LLI 157     Set pntr to decimal exponent storage
        DCM         Decrement decimal exponent value
        RET         Return to calling program
```

```
MINEXP,   CAL FPD10      Compensated decimal exp is -, multip
          JFZ MINEXP     FPACC by 0.1, loop till dec exp is 0
          RET            Exit with converted value in FPACC
FPD10,    LEI 134        Mult FPACC by 0.1 rtn, pntr to
                         FPOP LS Byte
          LDH            Set D = 0 for sure
          LLI 124        Set pointer to FPACC
          LBI 004        Set precision counter
          CAL MOVEIT     Move FPACC to FPOP (include exp)
          LLI 127        Set pointer to FPACC exponent
          LMI 375        Place FP form of 0.1 (dec.) in FPACC
          DCL            Place FP form of 0.1 (dec.) in FPACC
          LMI 146        Place FP form of 0.1 (dec.) in FPACC
          DCL            Place FP form of 0.1 (dec.) in FPACC
          LMI 146        Place FP form of 0.1 (dec.) in FPACC
          DCL            Place FP form of 0.1 (dec.) in FPACC
          LMI 147        Place FP form of 0.1 (dec.) in FPACC
          CAL FPMULT     Mult original value (in FPOP) by 0.1
          LLI 157        Set pntr to decimal exponent storage
          INM            Increment value
          RET            Return to calling program
DECBIN,   LLI 153        Decimal to binary conversion
          LAB            Restore character to accumulator
          NDI 017        Mask to leave pure BCD number
          LMA            Place current number in temp storage
          LXD 150 000    Set pntr to working area LS Byte
          LLI 154        Set another pntr to LSB input registers
          LBI 003        Set precision counter
          CAL MOVEIT     Move original value to working area
          LLI 154        Set pntr to LSByte of input storage
          LBI 003        Set precision counter
          CAL ROTATL     Rotate left (X2)  (total =X2)
          LLI 154        Set pointer to LSByte again
          LBI 003        Set precision counter
          CAL ROTATL     Rotate left (X2)  (total =X4)
          LEI 150        Set pntr to LSByte of rotated value
          LLI 154        & another to LSByte of original value
          LBI 003        Set precision counter
          CAL ADDER      Add original to rotated (total = X5)
          LLI 154        Set pointer to LS Byte again
          LBI 003        Set precision counter
          CAL ROTATL     Rotate left (X2)  (total = X10)
          LLI 152        Set pointer to clear working area
          XRA            Clear accumulator
          LMA            Deposit MSByte of working area
          DCL            Decrement pointer to MS Byte
          LMA            Put zero there too
          LLI 153        Set pntr to current digit storage
```

```
        LAM         Fetch latest BCD number
        LEI 150     Set pntr to LS Byte of working area
        STAD        Deposit latest BCD number in LSByte
        LLI 154     Set up pointer
        LBI 003     Set precision counter
        CAL ADDER   Add in latest no. to complete DECBIN
        RET         Return to calling program
```

The next routine converts the floating point binary number in the FPACC to its floating point decimal equivalent, and outputs it to the display device as ASCII characters in the following format:

0.1234567 E+07

First, the normalized binary value is converted to a binary value, in which the binary exponent is within the range of -4 to -1. As this is done, the decimal exponent is generated. Once the binary exponent is properly adjusted, the decimal mantissa is output by multiplying the adjusted binary mantissa by ten for each decimal digit. Each multiplication causes the next decimal digit to be pushed out into the most significant byte+1 of the binary mantissa in its BCD code. As each digit is pushed out, its ASCII code is formed and the ECHO routine is called to output the digit. When the mantissa has been output, the decimal exponent is converted, using the method described in chapter four for binary to decimal conversion, and then output.

The listing for the FPOUT routine is presented next.

```
FPOUT,      LLI 157     Set pntr to decimal exponent storage
            LMI 000     Clear decimal exponent location
            LLI 126     Set ptr MSByte FPACC MANTISSA
            LAM         Fetch MSByte FPACC MANTISSA
                        To accumulator
            NDA         Set flags after load operation
            JTS OUTNEG  If MSB=1 have negative number
            LAI 253     Otherwise set ASCII code for +
            JMP AHEAD1  Go display + sign
OUTNEG,     LLI 124     Neg number, set pntr LSByte FPACC
            LBI 003     Set precision counter
```

```
         CAL COMPLM    Perform two's complement on FPACC
         LAI 255       Set ASCII code for - sign
AHEAD1,  CAL ECHO      Display sign of mantissa
         LAI 260       Set ASCII code for 0
         CAL ECHO      Display 0
         LAI 256       Set ASCII code for (.)
         CAL ECHO      Display (.)
         LLI 127       Set pntr to FPACC exponent
         DCM           Subtract one from exponent
DECEXT,  JFS DECEXD    If compen exp is +, mult mntsa by 0.1
         LAI 004       If compenstated exponent is negative
         ADM           Add 4 (decimal) to exponent
         JFS DECOUT    If exponent now 0 or +, output mntsa
         CAL FPX10     Otherwise, multiply mantissa by 10
DECREP,  LLI 127       Set pointer to FPACC exponent
         LAM           Get exponent after multiplication rtn
         NDA           Set flags after load operation
         JMP DECEXT    Repeat above test for 0 or + condition
DECEXD,  CAL FPD10     Multiply FPACC by 0.1
         JMP DECREP    Check status of FPACC exp after mult
DECOUT,  LEI 164       Set pntr to LSByte of output registers
         LDH           Make D = 0 for sure
         LLI 124       Set pointer to LS Byte of FPACC
         LBI 003       Set precision counter
         CAL MOVEIT    Move FPACC to output registers
         LLI 167       Set pntr to MSByte+1 of output regis
         LMI 000       And clear that location
         LLI 164       Set pntr to LS Byte of output register
         LBI 003       Set precision cntr, perform 1 rotate
         CAL ROTATL    Oper to compen for space of sign bit
         CAL OUTX10    Mul output X10, overflw to MSByte+1
COMPEN,  LLI 127       Set pointer to FPACC exponent
         INM           Increment exponent
         JTZ OUTDIG    Output digits when compen done
         LLI 167       Binary exp compensation rotate loop
         LBI 004       Set pntr to out. MSByte+1 & set cntr
         CAL ROTATR    Perform compen rotate right operation
         JMP COMPEN    Repeat loop till binary exponent =0
OUTDIG,  LLI 107       Set pntr to output digit counter
         LMI 007       Set digit counter to 7 to initialize
         LLI 167       Set pntr to MSD in out. reg MSByte+1
         LAM           Fetch BCD form digit to be displayed
         NDA           Set flags after load operation
         JTZ ZERODG    See if first digit is a 0
OUTDGS,  LLI 167       If not, pntr to MSByte+1 (BCD code)
         LAI 260       Form ASCII code by adding 260 (oct)
         ADM           To the BCD code
         CAL ECHO      And display ASCII coded decimal no.
```

```
DECRDG,   LLI 107       Set pntr to output digit counter
          DCM           Decrement value of output digit cntr
          JTZ EXPOUT    When it =0, do exp output routine
          CAL OUTX10    Otherwise multip output register X 10
          JMP OUTDGS    And output next decimal digit
ZERODG,   LLI 157       If 1st digit, set pntr to MS Byte
          DCM           Dcr vl to compen for skipping display
          LLI 166       Of 1st digit, then set pntr to MSByte
          LAM           Of output registers, fetch contents
          NDA           Set flags after load operation
          JFZ DECRDG    Check to see if entire mantissa is 0
          DCL           Check to see if entire mantissa is 0
          LAM           Check to see if entire mantissa is 0
          NDA           Check to see if entire mantissa is 0
          JFZ DECRDG    Check to see if entire mantissa is 0
          DCL           Check to see if entire mantissa is 0
          LAM           Check to see if entire mantissa is 0
          NDA           Check to see if entire mantissa is 0
          JFZ DECRDG    Check to see if entire mantissa is 0
          LLI 157       If entire mantissa =0, set pntr to
          LMA           Decimal exp storage & set it to zero
          JMP DECRDG    Before proceeding to finish display
OUTX10,   LLI 167       Multip regis by 10 to push out BCD
          LMI 000       Code of MSD, 1st clr outpt MSByte+1
          LLI 164       Set pntr to LSByte of output registers
          LXD 160 000   Set another pntr to working area
          LBI 004       Set precision counter
          CAL MOVEIT    Move original values to working area
          LLI 164       Set pntr to original value LS Byte
          LBI 004       Set precision counter
          CAL ROTATL    Start multiply by 10 rtn  (total = X2)
          LLI 164       Reset pointer
          LBI 004       And counter
          CAL ROTATL    Multiply by two again  (total = X4)
          LLI 164       Set pntr to LS Byte of rotated value
          LEI 160       And another to LSByte of original vlu
          LBI 004       Set precision counter
          CAL ADDER     Add orig value to rotated (total = X5)
          LLI 164       Reset pointer
          LBI 004       And counter
          CAL ROTATL    Multiply by 2 once more (total = X10)
          RET           Finished multip output registers by 10
EXPOUT,   LAI 305       Set ASCII code for letter E
          CAL ECHO      Display E for exponent
          LLI 157       Set pointer to decimal exponent
          LAM           Fetch decimal exponent to ACC
          NDA           Set flags after load operation
          JTS EXOUTN    If MSB equals 1, value is negative
```

```
                LAI 253         If val is pos, set ASCII code for + sign
                JMP AHEAD2      Go to display the sign
EXOUTN,         CMA             For negative exp, perform two's comp
                INA             In standard manner
                LMA             And restore to storage location
                LAI 255         Set ASCII code for  -  sign
AHEAD2,         CAL ECHO        Display sign of exponent
                LBI 000         Clear register B for use as a counter
                LAM             Fetch decimal exponent value
SUB12,          SUI 012         Subtract 10 (decimal)
                JTS TOMUCH      Look for negative result
                LMA             Restore pos result, maintain count of
                INB             How many times 10 (dec) can be
                JMP SUB12       Subtracted to  obtain MSD of exponent
TOMUCH,         LAI 260         Form ASCII char for MSD of exp by
                ADB             Adding 260 to count in register B
                CAL ECHO        And display MSD of exponent
                LAM             Fetch remainder from dec exp strg
                ADI 260         And form char for LSD of exponent
                CAL ECHO        Display LSD of exponent
                RET             Exit FPOUT routine
```

This final routine ties the FPINP and FPOUT routines together along with the floating point mathematical routines FPNORM, FPADD, FPSUB, FPMULT, and FPDIV to create an operating program that may be used as a floating point calculator program. All that is required by the reader is to supply the I/O driver routines, as previously described. The program will allow one to enter and receive data in the following format:

27.6E-2 X -5 = -0.1380000E+01

```
LOOK-UP TABLE   AAA             Low addr for start of FPADD subrtn
                BBB             Page addr for start of FPADD subrtn
                CCC             Low addr for start of FPSUB subrtn
                DDD             Page addr for start of FPSUB subrtn
                EEE             Low addr for start of FPMULT subrtn
                FFF             Page addr for start of FPMULT subrtn
                GGG             Low addr for start of FPDIV subrtn
                HHH             Page addr for start of FPDIV subrtn
FPCONT,         LXS 000 001     Initialize stack pointer
                CAL CRLF2       Display a few CR/LF's for I/O device
                CAL FPINP       Let operator enter a FP decimal num
                CAL SPACES      Display a few spaces after number
```

```
        LLI 124         Set pointer to LS Byte of FPACC
        LXD 170 000     Set pointer to temp number storage
        LBI 004         Set precision counter
        CAL MOVEIT      Move FPACC to temp storage area
NVALID, CAL INPUT       Fetch operator from input device
        LBI 000         Clear register B
        CPI 253         Test for + sign
        JTZ OPERA1      Go set up for + sign
        CPI 255         If not + then test for  -  sign
        JTZ OPERA2      Go set up for  - sign
        CPI 330         If not above, text for X (multiply) sign
        JTZ OPERA3      Go set up for X sign
        CPI 257         If not above, text for / (divide) sign
        JTZ OPERA4      Go set up for / sign
        CPI 377         If none of above, test for rubout
        JFZ NVALID      If none of above ignore current input
        JMP FPCONT      If rubout start new input sequence
OPERA4, INB             Set up register B based on above tests
        INB             Set up register B based on above tests
OPERA3, INB             Set up register B based on above tests
        INB             Set up register B based on above tests
OPERA2, INB             Set up register B based on above tests
        INB             Set up register B based on above tests
OPERA1, LLI 110         Set pntr to look-up table addr storage
        LMB             Place look-up addr in storage location
        CAL ECHO        Display the operator sign
        CAL SPACES      Display a few spaces after oper sign
        CAL FPINP       Let operator enter 2nd FP decimal no.
        CAL SPACES      Provide a few spaces after 2nd number
        LAI 275         Place ASCII code for = in accumulator
        CAL ECHO        Display = sign
        CAL SPACES      Display a few spaces after = sign
        LLI 170         Set pntr to temp number storage
        LXD 134 000     Set another pntr to LS Byte of FPOP
        LBI 004         Set precision counter
        CAL MOVEIT      Move 1st number inputted to FPOP
        LLI 110         Set pntr to look-up table addr storage
        LLM             Bring in low order addr of look-up tbl
        LHI XXX         XXX=page this routine located on!
        LEM             Bring in an addr stored in look-up tbl
        INXH            Residing on this page (XXX) at loca-
        LDM             tions ****+B and ****B+1 & place it
        LLI Z+1         In register D & E then change pntr to
        LME             Addr part of ins labeled result below
        INXH            And transfer the look-up tbl contents
        LMD             So it becomes the addr prtn of the ins
        LHI 000         Labeled result, then restore
        LDH             Registers H and D to zero
```

```
*Z* RESULT,   CAL DUMMY   Call subrtn indicated by current addr
              CAL FPOUT   Display results of the calculation
              JMP FPCONT  Go back & wait for nxt problem input
CRLF2,        LAI 215     Subroutine to provide CR & LF's
              CAL ECHO    Put ASCII cd for CR in ACC & display
              LAI 212     Put ASCII code for LF in ACC
              CAL ECHO    Then display
              LAI 212     Display LF again
              CAL ECHO    Display LF
              RET         Return to calling routine
SPACES,       LAI 240     Set up code for spaces in ACC
              CAL ECHO    Display a space
              LAI 240     Do it again, put code for space in ACC
              CAL ECHO    Display a space
              RET         Return to calling routine
```

The three routines, FPINP, FPOUT, and FPCONT, as presented, require approximately three pages of memory. This requirement may be shortened by forming various subroutines for several common instruction sequences, and by using the stack to store various temporary data values. This has not been done here to maintain clarity of operation. However, the ambitious reader should have little difficulty in shortening the program. For reference purposes, the following list is given of the data areas on page 00 used by these routines.

103	Input MANTISSA sign storage
104	Input EXPONENT sign storage
105	Input DIGIT COUNTER
107	Output DIGIT COUNTER
110	Temporary storage for control OPERATOR
150-153	Input working area
154-156	Input storage area (for DECBIN conversion)
157	Input EXPONENT (decimal equivalent)
160-163	Output working area
164-167	Output storage (for BINDEC conversion)
170-173	Temporary number storage

This floating point program has been assembled and is presented in appendix F as a memory dump. The routines have been origined to reside on pages 01 through 06 with the data areas as described on page 00, and the user defined I/O routines designated to reside on

page 07. The actual start location of each routine is indicated in the list that follows the memory dump. The order in which the routines have been presented for explanation is not the same order in which they have been assembled. However, the instructions within each routine are exactly as listed in the text of the chapter.

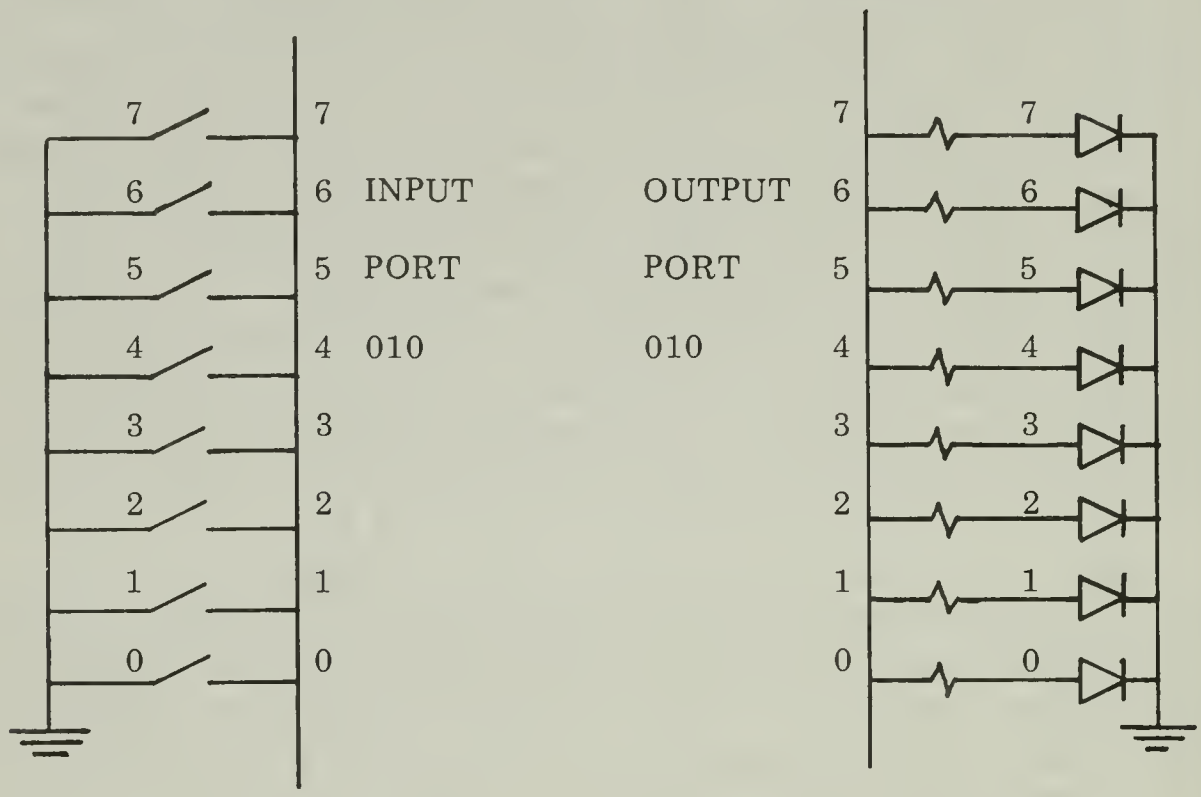
INPUT/OUTPUT PROCESSING

Writing a program to communicate with a peripheral device is as important as almost any other programming task one may have to perform. Nearly every program one can think of requires some form of input or output. The input data may be received from a group of sensors that make up a burglar alarm system, or it may be entered through a keyboard device for a variety of control or data entry purposes, or from a bulk storage device, such as magnetic tape, for loading programs or reading large blocks of data. The output data may be used to turn relays or lights on and off, to send characters to a display device (such as a mechanical printer or video display), or to store programs or data on a bulk storage device. No matter what the task, it is important to be able to write effective I/O driver programs.

Before the various forms of I/O routines are presented, it is important to understand the input/output setup of the 8080. The 8080 CPU handles input and output by transferring the data through input and output ports. Up to 256 input and output ports may be provided. Each port is accessed by an individual input or output instruction. These instructions, as defined in chapter one, require two bytes of memory. The first byte indicates an input (333 octal) or output (323) instruction, and the second byte indicates the port number, 000 through 377 (octal). The data is transferred over eight parallel data lines that provide communication between the CPU and the peripheral device through the accumulator. When an input instruction is executed, the data received from the designated input port is stored in the accumulator. And, data to be sent to an output port must be loaded into the accumulator before performing the output. It is important for one to note that the execution of either an input or output instruction does not affect the condition of the status flags. If it is desired to set the condition of the flags with respect to the data received or outputted, one must perform an instruction such as NDA, or a rotate instruction, or whatever would be appropriate to test the data.

The first type of I/O processing to be discussed is one that would be used in conjunction with the simplest form of input and output devices. The input device might be a group of switches, or sensors,

that provide a '1' or '0' to each of the input data lines to indicate an open or closed position. The output device might consist of a group of lamps that are turned on by outputting a '1,' or off by outputting a '0.' The schematic below illustrates this configuration.



As indicated by this schematic, input port 010 has eight switches, numbered 0 through 7, connected to its corresponding eight data leads. These switches might be sensor switches in a burglar alarm system that monitor the opening and closing of doors throughout a building. Assuming that the switches are closed when the doors are properly secured, the following input routine may be used to test for a door being opened.

```
SWTEST,   INP 010        Input switch conditions
          NDA            Set flags after input
          JTZ SWTEST     All closed, continue testing
          . . .          One or more doors open, alarm condtn
```

This routine illustrates the simplicity of inputting information from an input port. The data read into the accumulator by the input instruction indicates the open (1) or closed (0) condition of the switches connected to input port 010. By providing an NDA instruc-

tion after the input, the CPU status flags are conditioned to indicate whether one of the switches is opened. For this example, the zero flag will be set to '1' if all the switches are closed. Should any of the switches become open, the data lead corresponding to that switch will go to a '1' condition, and the zero flag will be reset by the NDA instruction since the accumulator will not be zero.

One should be aware that it is not necessary to use all eight data leads of an input port. Suppose that there are only five switches, 0 through 4, connected to the input port, and the other three leads were not used. The same type of test could be performed on these five leads by simply changing the NDA instruction to NDI 037. The Z flag would, again, indicate the possible open condition of one or more of the five switches. If only one data lead was required, by connecting it to bit 7 of the input port, the S flag could be used in testing for a '1' or '0' condition. In this case, the NDA would be followed by a JTS or JFS instruction.

Over on the output port side of the schematic, a set of eight lights, shown here as LEDs, are connected to the eight output port data leads, numbered 0 through 7. Each light is turned on by outputting a '1' to the corresponding data lead. The light is turned off by outputting a '0.' For example, to turn on the odd numbered lights and turn off the even numbered lights, one would load the accumulator with a bit pattern of "10101010" and output it to output port 010, as listed below. (Note that the output port number is the same as the input port number. This configuration is often used to minimize the hardware decoding logic required to select an input and output port.)

```
. . .
LAI 252          Load accumulator with bit pattern
OUT 010          Output pattern to lights
. . .
```

If these lights were connected to the control panel of the burglar alarm system described above, they could be used to indicate which of the doors have been opened by including an output instruction in the test program as follows:

```
SWTEST,   INP 010       Input switch conditions
          OUT 010       Output to control panel lights
          NDA           Set flags after load
          JTZ SWTEST    All closed, continue testing
          . . .         One or more doors open, alarm condtn
```

After inputting the data from the switches, the routine immediately outputs the same data to the lights. In so doing, any light that turns on will indicate that the corresponding door is open. The program then tests for a door open, just as before, and either continues testing if the doors are all closed, or performs whatever logic may be necessary when a door is found to be open (i.e. sounding an alarm, calling the police, etc.).

Naturally, the switches and lights used in this example may be replaced by a wide variety of devices for an even greater number of applications. For instance, the input may come from heat, light, or pressure transducers, or from such devices as analog-to-digital converters, which transform an analog signal to a proportional digital binary, or BCD, value. An output port may drive relays, 7 segment displays, alarms, or digital-to-analog converters.

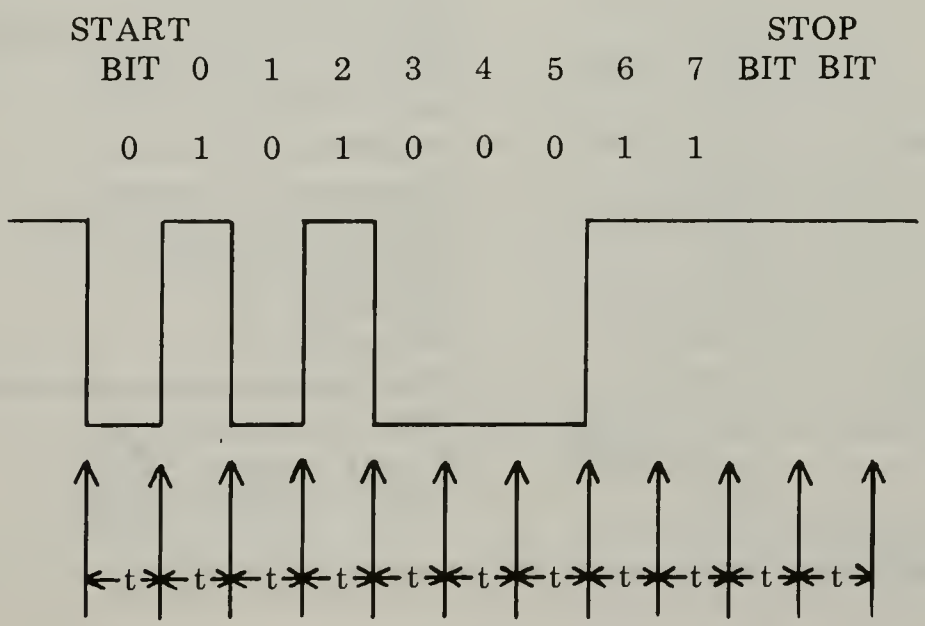
A very novel application for a simple output device is to connect a speaker to one bit of an output port and have the computer synthesize different frequencies to create music. The different tones are generated by outputting alternate ones and zeros with an appropriate delay (as described in chapter three) in between each output. The shorter the delay, the higher the frequency, and vice versa. By outputting a given set of tones in the proper sequence with each tone lasting the proper duration, a musical tune can be played by the computer. Many 8080 based microcomputers have been known to play such intriguing songs as "Mary had a little lamb" and "A bicycle built for two." Looking at this application from a more scientific viewpoint, this form of frequency synthesis may be used to generate any number of different waveforms for a multitude of technical applications.

One such technical application is in the generation of asynchronous serial data. Serial data is data that is sent one bit at a time with each bit lasting a specific amount of time before the next bit is

output. Asynchronous serial data is a short group of bits output in serial form. Each group of bits generally represents a single character of one of the standard character sets (i.e. ASCII, BAUDOT) although random data patterns may be transmitted in this fashion. The reason it is referred to as asynchronous is because the beginning of the group of bits may occur at any time, although once started, the timing of each bit in the group must meet the specified time. The timing diagram shown next illustrates the manner in which the ASCII code for the letter 'E' (11000101 in binary) is transmitted as asynchronous serial data.

As noted in the timing diagram, the character code for the 'E' is preceeded by a start bit. This bit is used to inform the receiving device that a character is being transmitted. The character code then follows the start bit, beginning with the least significant bit. The character transmission is completed by adding one or more stop bits to the end of the code. The stop bits are added to allow time for the receiving device to prepare to receive another character.

The timing diagram also indicates that there is a specific amount of time, 't,' for the duration of each bit. This timing is often referred to by the number of bits that could be transmitted in one second at this rate, rather than the amount of time used for each bit. The standard bit per second, or BAUD, rates used for transmitting ASCII code range from 110 BAUD for many keyboard and printer devices, to 9600 BAUD for high speed devices.



The computer may be used to generate serial data in this form, by outputting one bit at a time, and providing a programmed delay between each bit to create the proper timing. The routine listed next outputs eight bit characters as asynchronous serial data with two stop bits. The timing generated by this routine outputs data at a rate of 110 bits per second. This corresponds to a delay between bits of 9.09 milliseconds, although the actual delay of this routine is 9.08 milliseconds, which is less than 0.2% error. The timing may be calculated by adding up the number of cycles per instruction (indicated in the lefthand column, for a microcomputer with two wait cycles added for each memory access, as described in chapter three) for each instruction executed between the output of each bit and then multiplying the total by 0.5 microseconds.

This routine may be used to output ASCII characters to a printer or other type of device that receives asynchronous serial data at 110 bits per second. The character to be output must be in the accumulator when this routine is called. This routine uses registers B, C and D in its operation. Therefore, if their contents must be maintained, a proper push and pop instruction may be added to the beginning and end, respectively, of this routine to save and then restore them. Each bit to be output is rotated into bit zero of the accumulator and output to output port 001.

```
        PRINT,  NDA             Clear the carry to
                RAL             Set up start bit
                OUT 001         Output start bit
 6              RAR             Restore character in accumulator
27              CAL TIMER       Delay for one bit time
11              LDI 010         Set data bit counter
27      PRINT1, CAL BITOUT      Output data bit and delay
 7              DCD             Decrement bit counter
16              JFZ PRINT1      If not zero, output next bit
 7              LBA             Save character in B
11              LAI 001         Set up stop bit
14              OUT 001         Output stop bit
 7              LAB             Restore character to accumulator
27              CAL TIMER       Delay for one stop bit
27              CAL TIMER       Delay for second stop bit
                RET             Return to calling program

14      BITOUT, OUT 001         Output bit to port 001
```

```
 6                RRC          Position for next output
27                CAL TIMER    Delay one bit
16                RET          Return

11      TIMER,    LCI 003      Set up outside counter
27      TIME1,    CAL MORE     Jump to inside counter
 7                DCC          Decrement outside counter
16                JFZ TIME1    If not zero, more delay
27                CAL DUMMY    Dummy call to use up time
 7                LBA          Save character in B
16      DUMMY,    RET          Return to do next output

11        MORE,   LBI 240      Set inside counter
 7        MORE1,  DCB          Decrement inside counter
 7                INB          Increment inside counter
 7                DCB          Decrement inside counter for test
16                JFZ MORE1    If not zero, continue inside loop
16                RET          Else, return to outside loop
```

The type of peripheral devices discussed up to this point require nothing more than a simple input or output instruction to transfer the information. When a transfer is to be made, the program does not care what state the peripheral is in previous to the transfer. However, for many peripherals, the process of transferring data between it and a computer under program control requires some type of hand-shaking. This means that a program must check whether the device is ready to make a data transfer and, when so indicated, perform the logic necessary to make the transfer. In general, there are two methods used to provide the program control. One method is to have the program continuously input the status bit of the peripheral, often referred to as the "programmed data transfer," or PDT, bit, until it indicates the device is ready for a data transfer. The other method is for the peripheral device to send a signal to the computer when it is ready for a data transfer. This signal is called an interrupt. Once an interrupt is received, the method of data transfer is similar to that for the PDT operation. As will be indicated, the major difference between the two modes is that under PDT operation the program must continuously check the status of the device, while under interrupt operation the program is free to perform other operations while waiting for the interrupt from the peripheral.

Whether a peripheral device is designed to generate interrupts or

operate strictly in the PDT mode, there is generally a PDT bit associated with it. A device that generates interrupts will have a PDT bit to provide the option of operating in the PDT mode and, when operating under interrupt, to identify itself as the device that generated the interrupt, should there be more than one interrupting device in the system. It is, therefore, important to understand how to check the PDT bit of a device. Any peripheral that is designed to operate with a PDT bit will have a status output. This output may contain only the PDT bit, or it may include several other status leads to indicate error conditions that may occur in the peripheral. These status leads are connected to an input port allowing the status to be read in by an input instruction. After the status has been read into the accumulator, the condition of the PDT bit must be transferred to the CPU flags. This will allow the program to perform a conditional jump either to continue to check the PDT bit for the device to be ready, or to the routine that handles the data transfer, should the PDT indicate the device is ready. If the PDT bit is located in the most significant bit of the input port, the sign flag will indicate the condition of the PDT bit by performing an NDA instruction, as illustrated in the following instruction sequence. This instruction sequence assumes that the PDT bit is true when the device is ready for the transfer.

```
CKPDT,    INP XXX      Input the device status
          NDA          Set the sign flag to condition of PDT
          JFS CKPDT    If sign false, PDT is not set
          . . .        If sign true, device is ready
          . . .        For data transfer
```

If the PDT bit is located in a bit position other than bit 7, an "and immediate" instruction with all bits zero except the bit corresponding to the PDT bit will set the zero flag to the opposite condition of the PDT bit. For example, if the PDT bit is located in the least significant bit, and it is true when the device is ready, the following program would check the PDT bit at bit 0 until it indicated that the device was ready.

```
CKPDT,    INP XXX      Input the device status
          NDI 001      Set zero flag opposite of PDT
```

```
JTZ CKPDT       If zero true, device not ready
. . .           If zero false, device is ready for
. . .           Data transfer
```

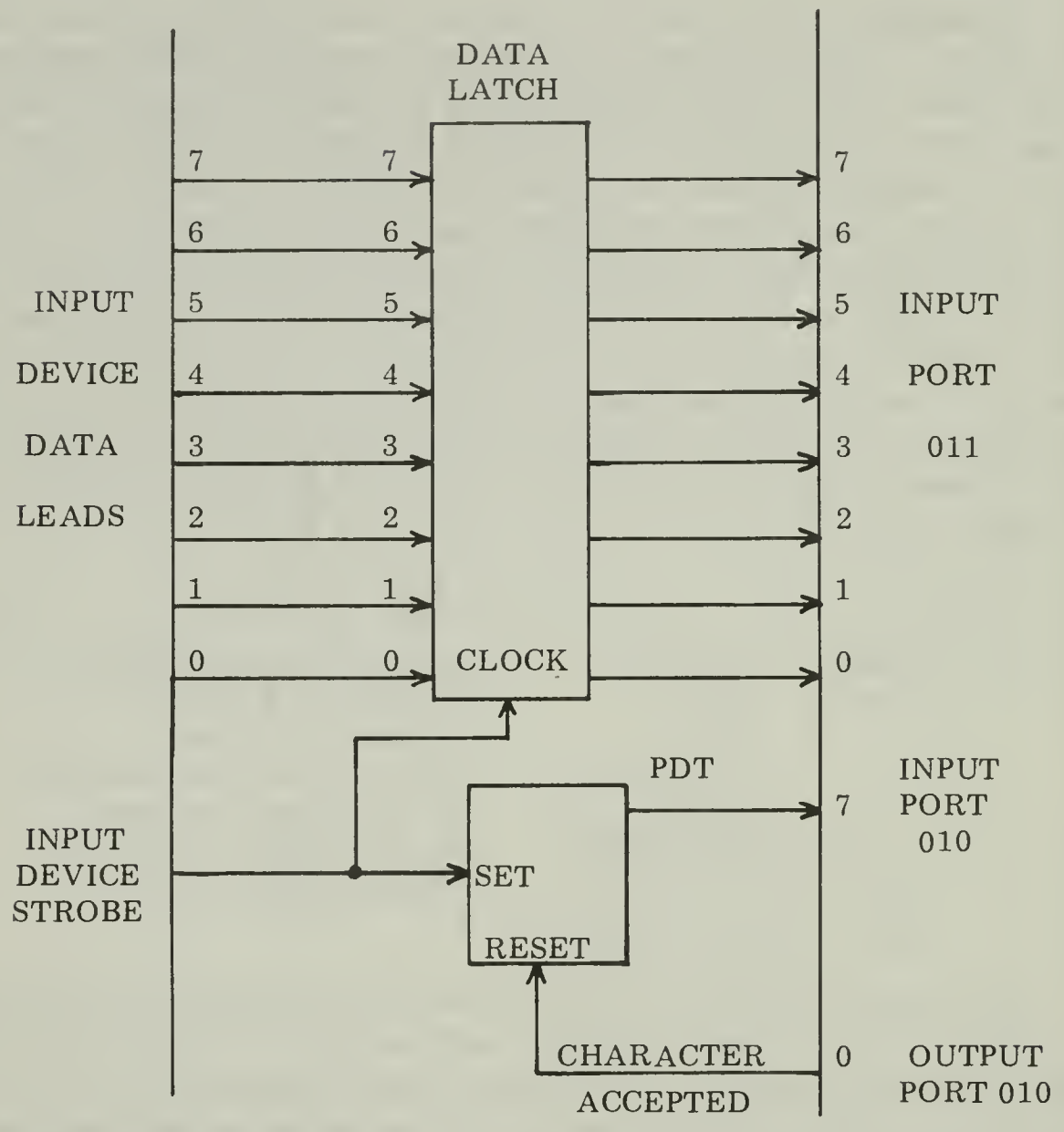
There are times when it is known that a PDT bit must change within a certain amount of time. For instance, after outputting a character to a display device there is usually a specific maximum time limit for the device to accept it and the PDT bit to come true again. If this time limit is surpassed, it might indicate a problem with the display device. This possible error may be monitored by the program by inserting a counter in the PDT test loop. The counter would be calculated to allow only a given amount of time to elapse before the PDT bit must return, otherwise an error routine would be entered to inform the operator of a possible problem. The following format may be used to include a timer in the PDT checking routine. The exact timing of this loop may be calculated as discussed in chapter three.

```
LPSET,    LBI YYY       Set up timing loop counter
CKPDT,    INP XXX       Input the device status
          NDA           Set flags after input
          JTS PDTSET    Have PDT, continue processing
          DCB           No PDT, decrement timer
          JFZ CKPDT     Not zero, continue test
          . . .         Time out, possible error
```

PDT operation, for most input devices, generally follows the same basic procedure. When a program requires data from an input device, it reads the status of the device and checks the condition of the PDT bit. If the PDT bit indicates the device has data available, the program can proceed to input the data. Once the data has been read in, the device may require a "character accepted" signal from the program to reset the PDT bit.

This procedure is typical of many interfaces that latch the data in from a device and then set a PDT bit. The schematic on the following page illustrates this form of interface. The data is entered into the latches by the input device by setting up the data at the input to the latches and then pulsing the strobe line of the latches. This same

strobe signal sets the PDT bit. After the data has been read by the program, the reset line is pulsed by the program outputting a "character accepted" signal.



A program to control this type of interface is listed next. In this case, the PDT bit is connected to bit 7 of input port 010. This allows the program to check for the PDT bit by performing an input from port 010, setting up the flags from the data received, and testing for the condition of the S flag. The data is entered through input port 011. Once the data has been accepted, the PDT is reset by outputting a "character accepted" signal.

```
PDTINP,   INP 010        Input device status
          NDA            Set flags after input
          JFS PDTINP     If sign = zero, no PDT bit
          INP 011        PDT = one, input data
RESET,    PUSS           Save input data
          LAI 001        Set up output pulse
          OUT 010        Output "character accepted"
          XRA            Set up end of pulse
          OUT 010        Reset character accepted
          POPS           Restore data to accumulator
          RET            Return to calling program
```

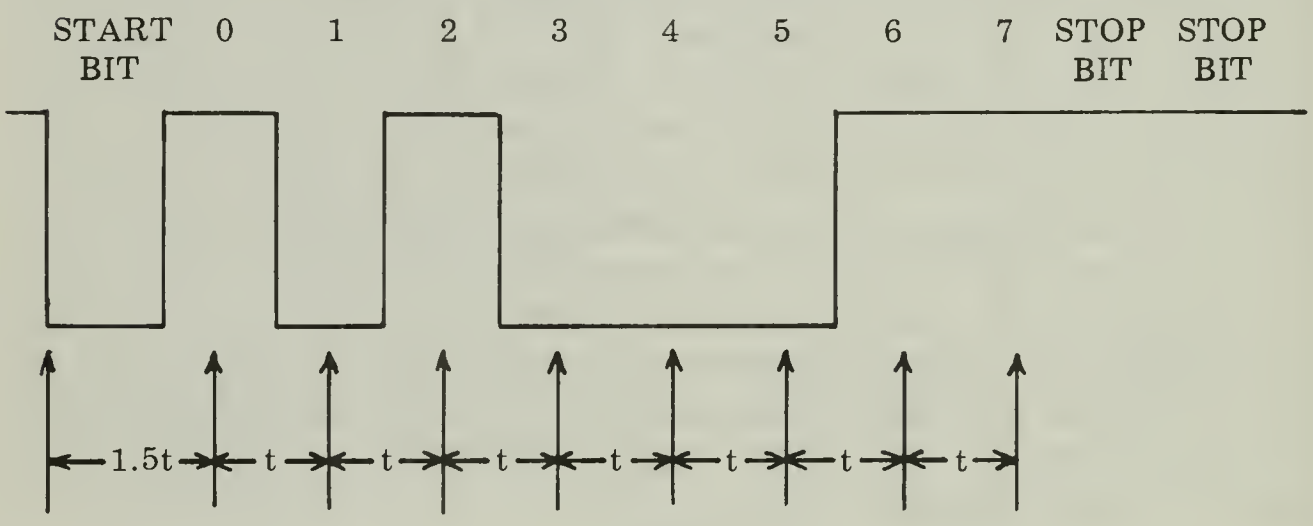
In the program listed here, the "character accepted" signal is derived by loading the accumulator with 001 and outputting it to output port 010, then loading the accumulator with 000 and outputting it to output port 010. This effectively creates a pulse on the least significant data lead of output port 010 to provide the "character accepted" signal. Some interfaces, however, use the output port strobe as the reset for the PDT, rather than a data lead from an output port. If this is done, all that would be required to send a "character accepted" signal would be to execute a single output instruction to output port 010 without being concerned with the contents of the accumulator, since the data leads of the output port would not be connected to anything. A third possibility is that the interface resets the PDT when the input is executed, by using the input port strobe for input port 011. In this case, the input routine may be exited just after the data is inputted.

The label RESET has been included in this routine to point out the portion of the routine that resets the PDT bit. This portion may be required as an initial reset for the input device at the start of a program that uses the device to receive data. Quite often, when dealing with such devices, it is necessary to output a reset during the initializing stages of the program. This guarantees that the device status will indicate the true status when the device is first called upon to input some data. For the other cases described above, in which the output or input strobe is used to reset the PDT, the corresponding instruction should be executed to initialize the input device.

Another type of routine that may be considered to test the status before inputting the data is a program that inputs asynchronous

serial data. The start bit of the asynchronous data could be considered its PDT bit. The input routine would test for the presence of the start bit and, when received, the data bits that follow may be read in by sampling the data at the proper time intervals.

Sampling the data is performed by providing a programmed delay until the mid-point of each bit is reached, and then inputting the value of the bit as it appears on the input data lead. The arrows in the timing diagram shown below indicate when each bit should be sampled by the program. The left most arrow indicates when the start bit is first detected. The next arrow indicates a delay time equal to one and a half bits before sampling bit zero of the data. Each subsequent sample is taken after a delay time of one bit.



The program listed next may be used to receive asynchronous serial data, eight bits at a time, such as that generated by the PRINT routine, previously presented in this chapter. The timing provided in this routine reads the data at 110 BAUD. By altering the delay, as described in chapter three, this timing may be changed to input data over a wide range of BAUD rates. Also, like the PRINT routine, the timing indicated assumes that two wait cycles are added for each memory access. The number of cycles for each instruction is indicated in the lefthand column. The delay time between sampling may be calculated by adding up the number of cycles for each instruction executed between inputs and multiplying by 0.5 microseconds. The major portion of the delay is provided by the same "timer" subroutine used in the PRINT routine.

The data is input through bit 7 of the input port. This allows the program to simply test the sign flag for the start bit. Then, as each bit is inputted, the previous bits are added to it, and they are shifted to the right. When the last bit has been inputted, an additional delay of one bit time is added to make sure the input data is into the stop bit before returning to the calling program. If this final delay was not provided, and the last data bit of the input was a '0,' the calling program could call SRLINP again while the last data bit is still at the input port. If this occurs, SRLINP would input the last data bit and assume it to be the start bit of a new character, which would result in the input of erroneous data. The data received is returned to the calling program in the accumulator.

```
      SRLINP,   INP 001        Input to look for start bit
 6              NDA            Set flags after input
16              JTS SRLINP     S=1, start not here yet
 6              XRA            Have start, clear accumulator
27              CAL HAFBIT     Delay 1/2 bit time
27              CAL TIMER      Delay one bit time
11              LDI 010        Set data bit counter
27    SRLIN1,   CAL NEXBIT     Input data bit
11              NDI 377        Added to use up time
 6              DCD            Decrement bit counter
16              JFZ SRLIN1     If not zero, continue input
 7              LAB            Move data to accumulator
 6              RLC            Correct for extra rotate
27              CAL TIMER      Delay into stop bit
16              RET            Return to calling program

14    NEXBIT,   INP 001        Input the data bit
11              NDI 200        Mask off unused input
 6              ADB            Add previous input
 6              RRC            Position for next input
16              JMP TIMER      Delay and return

11    HAFBIT,   LBI 364        Set 1/2 bit delay constant
16              JMP MORE1      Delay 1/2 bit and return
```

PDT operation of a parallel output device is generally straightforward. When the PDT bit is checked, and indicates the device is ready to accept data, the data may be output. Upon receipt of the data by the device, the PDT bit will change state to indicate that the device

is busy processing the data. Once the processing is completed, the PDT will return to its device ready status and wait for the next output from the program. Therefore, if the program is to output more than one character, the PDT bit must be monitored after each character is output to determine when the device is ready to accept the next character.

The following routine might be used to output a line of text to a printer that accepts ASCII characters as eight bit parallel data. This routine fetches the characters one at a time from a buffer and outputs them to the printer. When a carriage return is detected in the character string, it is output, followed by a line feed, and then the program returns to the calling program. The PDT bit is assumed to be in bit 0 of the status input from the printer. Also, when the routine is called, registers H and L are assumed to be pointing to the start of the character string.

```
LINOUT,   CAL CKPDT     Wait for printer PDT
          LAM           Fetch character to output
          CPI 215       Character = carriage return?
          JTZ FINISH    Yes, complete output
          OUT 002       No, output character
          INXH          Advance string pointer
          JMP LINOUT    Wait for PDT

FINISH,   OUT 002       Output carriage return
          CAL CKPDT     Check printer PDT
          LAI 212       Set line feed character
          OUT 002       Output line feed
          RET           Return to calling program

CKPDT,    INP 002       Input printer PDT
          NDI 001       Set flags after input
          JTZ CKPDT     PDT not set, wait
          RET           PDT set, return
```

This method of checking the PDT bit of an I/O device to transfer data is commonly used when it is not required to perform other functions while waiting for a data transfer from a peripheral. When there is no background program to be running while waiting for a peripheral, it is of no consequence for the program to remain in a

loop for long periods of time, testing the PDT bit of a peripheral. In order for the data to be transferred as rapidly as possible, the program must maintain constant attention to the PDT bit. This is also necessary if the data is only available for a given length of time, as is the case when using a card reader to enter data. The program must read a character from the card reader when it is available. Once the reader starts reading a card, it does not stop in between each character while the program reads it. The program must be ready for each character when it is available, otherwise, the character is lost.

Another method of transferring data under program control is to have the I/O device send an interrupt signal to the computer when it is ready for a data transfer. This signal interrupts the program currently in progress and directs the CPU to an interrupt service routine. The interrupt routine performs the logic necessary to transfer the data to or from the peripheral and then returns to the original program as though it had not been interrupted at all. This method of operation is known as interrupt processing.

Interrupt processing is analogous to a postman being interrupted while delivering the mail. As the postman is placing the mail in a row of mailboxes, someone walks up to him and taps him on the shoulder. The mailman completes filling the current mailbox, makes a mental note as to which mailbox is to be filled next, and turns to the person. The mailman is given a letter and is asked to send it. The mailman takes the letter and stores it in his mailbag. He then returns to the job of filling the mailboxes, beginning with the box he remembers as the next one to be filled. This is similar to the procedure followed by a computer when an interrupt is received from a peripheral.

When an interrupt signal is received by a computer, the current instruction being executed is completed and the address of the next instruction to be executed is saved. The interrupt service routine is then entered and the first function it must perform is to save the pertinent information for the interrupted program, such as the contents of the CPU registers and status flags. This information must be saved so that it may be properly restored before returning to the interrupted program. The interrupt service routine is now ready to perform the steps necessary to transfer the data between the peripheral and the computer. Once the transfer is completed, the CPU registers and

status must be restored to their contents at the time the interrupt was received. The interrupted program may now resume operation beginning with the next instruction to be executed in the interrupted routine.

Before presenting the specific method of interrupt processing used for the 8080, several features of the 8080 designed for this mode of operation should be discussed. These features make interrupt processing with the 8080 easy and effective.

There are two instructions used specifically in dealing with interrupts. One enables the computer to accept an interrupt signal received from a peripheral, and the other causes the computer to ignore any interrupt. These instructions are aptly called ENABLE INTERRUPTS, mnemonic EI, and DISABLE INTERRUPTS, mnemonic DI. They provide the ability to control when a program may or may not accept interrupts from an I/O device.

When writing a program to operate with interrupts, there are several times when it may not be desired to accept interrupts. One is during the initialization of the program, before all the necessary pointers, counters, and tables used by the program have been set up. If an interrupt is received before the program is ready to accept it, the program may receive or transmit erroneous data. To avoid such an occurance, the first instruction of the program should be the disable interrupt instruction. Then, after the initialization is complete, the interrupts may be enabled since the program is now ready to deal with the interrupts properly.

Another time that interrupts must be disabled is upon receipt of an interrupt. This is to allow the program enough time to respond to the first interrupt before receiving a second. Since this is always required when an interrupt is received, the 8080 CPU automatically disables interrupts when an interrupt is received. It is, therefore, not necessary to include a disable interrupt instruction in the interrupt routine. When the interrupt routine is finished and is ready to return to the interrupted program, an enable interrupt instruction must be executed before the return, to allow the receipt of future interrupts. If it is desired to allow "nesting" of interrupts, the interrupt routine can enable interrupts when it has completed its initial steps and can,

itself, be interrupted. The process of nesting interrupts will be discussed later in this chapter.

It may also be necessary to disable interrupts when a section of the program is changing information vital to the function of the interrupt routine. This information might be the address for storing or retrieving data to be transferred, or a flag indicating the progress of the program to the interrupt routine. For whatever reason, the program must disable interrupts before the change is made, make the change to the information, and, upon completion, enable the interrupts. This will provide the smooth transition of information needed by the interrupt routine.

Another feature of the 8080 is the separate set of input data lines that are used by the CPU to input the interrupt instruction. This interrupt instruction is executed after the interrupt is acknowledged, and the current instruction of the interrupted program is completed. The CPU inputs the instruction to be executed from these special data lines. The instruction placed on these lines is controlled by the peripheral hardware. The instruction that has been especially created for use as the interrupt instruction is the RESTART.

As presented in chapter one, the restart instruction is a one byte subroutine call that calls one of eight restart locations on page 00 in memory. Since it is a subroutine call, the current contents of the program counter will be pushed onto the stack. This stores the address of the instruction in the interrupted program that is to be returned to at the completion of the interrupt routine.

The final feature, which fills in the required functions to process interrupts effectively, is the ability to store the CPU registers and status flags at the beginning of the interrupt routine, and then restore them at its completion. This guarantees that the logic of the interrupted program is not disrupted by the occurrence of the interrupt. This function is provided by the PUSH and POP instructions, which allow one to store and recall the contents of the CPU registers and the flag status byte.

The procedure for receiving interrupts on an 8080 based microcomputer follows the basic steps outlined above. When interrupts are

enabled and an interrupt is received from a peripheral, the CPU automatically disables interrupts. When the current instruction being executed is completed, the CPU takes the next instruction to execute from the interrupt instruction input. This instruction will be one of the eight restart instructions that performs a subroutine call to the restart location specified. The contents of the program counter are pushed onto the stack, storing the return address to be used at the completion of the interrupt service routine.

The interrupt routine then begins execution at the restart location. There are eight memory locations provided in between restart addresses. However, most interrupt routines will require more than eight memory locations. Therefore, to provide enough room for the interrupt routine, a jump instruction may be used at the restart location to direct the CPU to the interrupt service routine elsewhere in memory.

The first step the interrupt routine must perform is to save the contents of the CPU registers and flag status. This is required to guarantee that, upon returning to the interrupted program, the registers and status are restored to the same values they contained at the time of the interrupt. This is accomplished by executing the PUSH instructions, which store the registers and status on the stack. For the convenience of having all of the registers available for the interrupt routine, they should all be pushed onto the stack. However, if the memory availability is limited, only those registers that are used by the interrupt service routine should be saved. This generally includes the accumulator and status, plus whatever other registers may be needed.

The interrupt routine is now ready to perform the logic required to service the interrupting device. The I/O portion of the interrupt service routine is usually a combination of the PDT routine for the device being controlled, and a routine that checks and stores the data for an input, or sets up the data to be output. Since the interrupt routine is meant to operate independently from the main program, it must perform its own checks and manipulate the data into and out of memory, as well as driving the peripheral. In order to accomplish this, and to provide a flexible interrupt routine, a link between the main program and the operation of the interrupt service routine must

be established.

One method of establishing the link is through the use of an interrupt table area. This table area normally includes at least three items, namely, a memory pointer, a data counter, and an in-progress flag. The memory pointer is used by the interrupt routine to indicate where input data is to be stored, or where output data is to be found. As the interrupt routine stores or outputs each byte of data, the memory pointer is advanced to the next location. The data counter indicates to the interrupt routine the amount of data to be received or sent. The routine decrements this counter each time it inputs or outputs some data and, when the counter reaches zero, the operation is complete. If necessary, the end of the operation may also be indicated by the receipt or transmission of a terminating character, such as a carriage return or line feed, which would terminate the operation before the data counter reached zero. The completion of the operation is then signalled by resetting the in-progress flag. The in-progress flag is set by the main program when the input or output is initiated. Then, when the interrupt routine is finished with the I/O operation, the in-progress flag is reset. The main program periodically checks this in-progress flag and when it is reset, the main program knows that the I/O operation is complete.

The in-progress flag may also serve another purpose. The interrupt routine can test this flag when an interrupt is received to determine whether an interrupt from the peripheral is expected. If it is expected, the interrupt routine can service the interrupt normally. If the interrupt is not expected, the interrupt may be ignored by resetting the I/O device, if necessary, and returning to the interrupted program, or by entering an error routine, which informs either the main program or the computer operator of the erroneous interrupt.

After the interrupt routine completes its operation, it prepares to return control to the interrupted program. The contents of the CPU registers and flag status must be restored to their values at the time the interrupt occurred. This is accomplished by using the POP instructions to pop the values off the stack and into the registers and status byte. The pop instructions must be executed in the reverse order of the push instructions at the start of the routine. The final step before returning to the interrupted program is to enable the

interrupts to allow future interrupts to be received. A listing of the common instructions is presented next as a summary of the interrupt routine procedure just described.

```
        ORG 000 0X0     Restart location X for interrupt
        PUSS            Save accumulator and status
        PUSH            Save registers H & L, if used by interrupt rtn
        PUSD            Save registers D & E, if used by interrupt rtn
        PUSB            Save registers B & C, if used by interrupt rtn
        JMP INTRTN      Jump to the interrupt routine

        ORG XXX XXX     Start of interrupt routine
INTRTN, . . .           Insert interrupt routine to
        . . .           Service interrupting device
        . . .
        . . .           End of interrupt routine
        POPB            Restore registers B and C
        POPD            Restore registers D and E
        POPH            Restore registers H and L
        POPS            Restore accumulator and status
        EI              Enable interrupts
        RET             Return to interrupted program
```

Interrupt processing for an input device is not exactly the same as that for an output device. The reason for this difference is that an interrupt from an input device indicates that the input device has a character or some data available for the program. The program may read the data in, process it, and then wait for another interrupt. For an output device, an interrupt indicates that the device has accepted the previous output and is ready to receive another character. Therefore, an output device must initially receive an output from the program before it generates an interrupt. Also, after the last character is received by the output device, a final interrupt will be generated, which must be ignored. This difference is further illustrated by the following input and output interrupt routines.

The input interrupt service routine presented next stores characters as they are inputted in a buffer area in memory until either the buffer is filled or a carriage return is received. This routine might be used to input characters from a keyboard, or data from a paper tape reader. This routine uses a table area which contains the input buffer

pointer, data counter, and in-progress flag. This table is listed below, followed by the table setup routine of the main program. The table setup routine initializes the contents of the table when an input sequence is to begin. Several facts about this routine and the table contents should be noted.

First, the in-progress flag in the first byte of the table is represented by the sign bit, not the contents of the entire byte. Therefore, the remaining seven bits in this byte may be used to signal error conditions or intermediate progress status, if this type of information is required by either the interrupt routine or main program.

Next, the input buffer pointer is stored in the second and third bytes of the table, with the low portion of the address in the second byte, and the page in the third. The address that must be initially loaded into these locations is the start address of the input buffer minus one. Setting this pointer to the location before the start of the input buffer is necessary because the input interrupt routine increments the input buffer pointer before storing the character received, not after.

Finally, one should note the use of the disable interrupt and enable interrupt instructions in the SETINT routine before and after the table is set up. This prevents an interrupt from being acknowledged while the contents of the input interrupt table are being initialized. However, the necessity of disabling interrupts while setting up this table may be eliminated by setting up the data counter first, and working backwards. In this way, the in-progress flag will not be set until the other pertinent data has been loaded.

INTERRUPT INPUT TABLE

```
FLAGIN,     000          In-progress flag, sign bit
            000          Low portion, input buffer pointer
            000          Page portion, input buffer pointer
            000          Data counter

            . . .        Set up routine for input
SETINT,     DI           Disable interrupts during setup
            LXH FLAGIN   Set pointer to table area
```

```
LMI 200         Set in-progress bit
INXH            Advance pointer
LMI LLL         Set low portion of input buffer
INXH            Advance pointer
LMI PPP         Set page portion of input buffer
INXH            Advance pointer
LMI XXX         Set data counter
EI              Enable interrupts when ready
. . .           Continue main program
```

The input interrupt service routine is listed shortly, followed by the flow chart. In this listing, the input is performed by a single input instruction followed by an output, which resets the input device's PDT bit. When implementing the routine, the instructions marked with ** should be replaced with those required to operate the specific input device being driven.

It is assumed in this routine that only one device in the system can generate an interrupt to this RESTART location. Therefore, it is not necessary to check for the PDT bit of the input device. If one desires to check the PDT bit, as an error checking measure, this routine should include an instruction sequence before the actual data is inputted, which inputs the PDT bit of the input device and tests its status. If the PDT bit is not set properly, an error routine should be entered. Otherwise, the routine should proceed to input the data and continue with the normal interrupt processing.

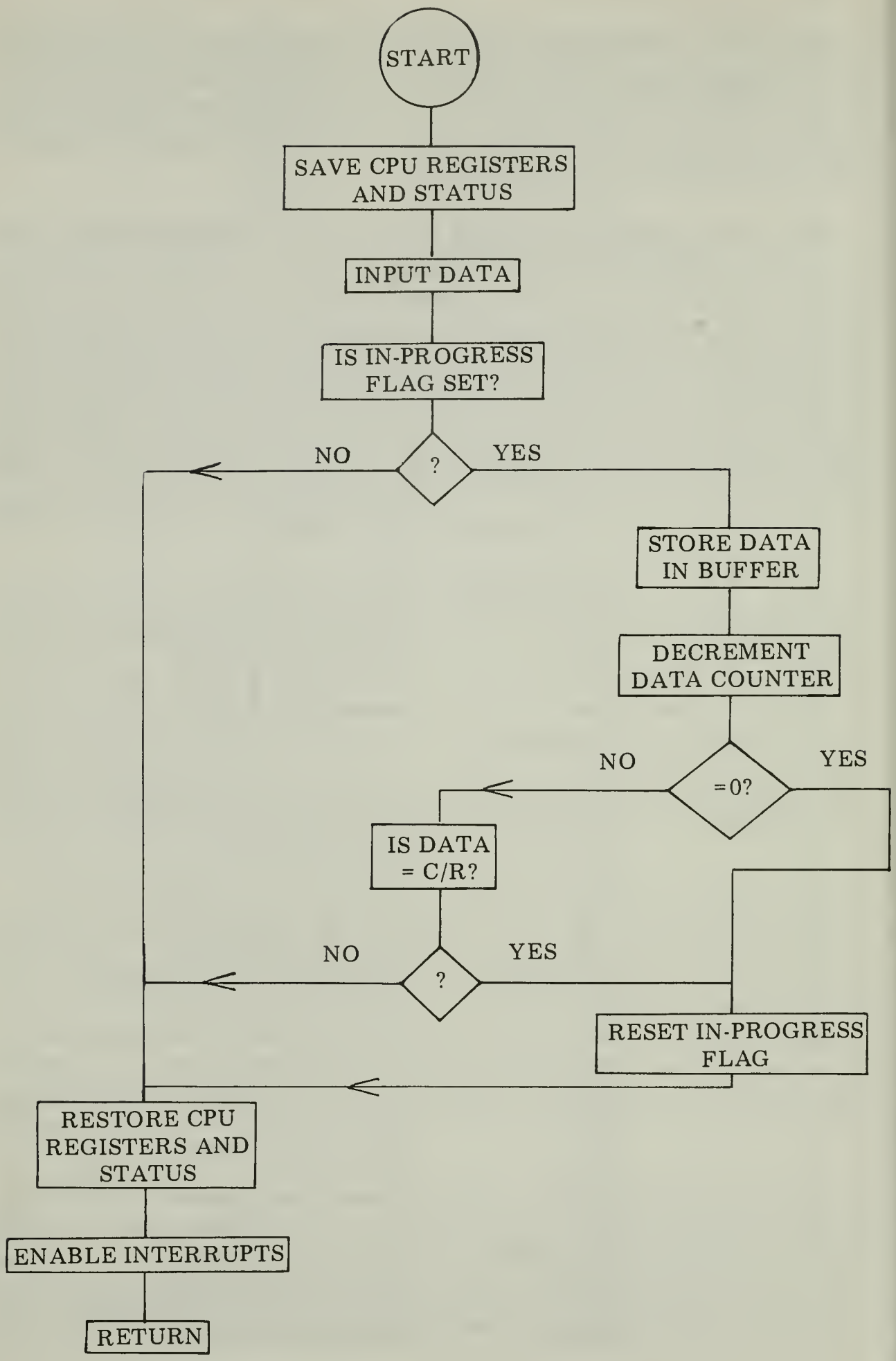
After the data has been inputted, the in-progress flag is checked to determine whether an input is expected by the interrupt program. This routine ignores an unexpected interrupt by simply returning to the interrupted program without storing the character inputted. It should be noted that by performing the input sequence before checking the in-progress flag, the input device will be properly reset (in this case, by the OUT 001 instruction) whether the interrupt was expected or not.

Assuming the interrupt was expected, the character received is stored in the input buffer. The new input buffer pointer is then stored in the input interrupt table. The data counter is decremented and, if zero, the in-progress flag is reset and the interrupt service routine is exited. If it is not zero, the character just received is tested for

a terminating character. In this routine, the input may be terminated by a carriage return, ASCII code 215. If it is a carriage return, the in-progress flag is reset to end the input operation and the interrupt routine is exited. If it is not a carriage return, the in-progress flag remains set when the routine is exited.

The short instruction sequence following the interrupt service routine listing may be used by the main program to check for the completion of the input operation. When the sign bit of the in-progress byte is reset, the main program will branch to the appropriate routine, referred to here as CMPTIN, to examine the data received. The contents of the interrupt input table may be used by the main program in examining the data input, since the input buffer pointer indicates the location of the last character received, and the data counter indicates either the number of unused locations in the input buffer, or, if equal to zero, that the entire buffer is filled.

```
                ORG 000 0X0     Restart X location
                PUSS            Save accumulator and status
                PUSH            Save registers H and L
                PUSD            Save registers D and E
                PUSB            Save registers B and C
                JMP INTINP      Jump to input interrupt routine

                ORG XXX XXX     Input interrupt service routine
INTINP,         INP 001     **  Input character from device
                OUT 001     **  Reset input device
                LBA             Temporarily save data
                LXH FLAGIN      Set pointer to in-progress flag
                LAM             Fetch in-progress flag
                NDA             Set status of flag byte
                JFS EXITIN      Not expected, ignore interrupt
                INXH            Advance table pointer
                LEM             Fetch low portion of input pointer
                INXH            Advance table pointer
                LDM             Fetch page portion of input pointer
                INXD            Advance input pointer
                LAB             Fetch input character
                STAD            Store character in input buffer
                INXH            Advance table pointer
                DCM             Decrement character count
                JTZ FININP      If zero, input finished
                DCXH            Decrement table pointer
                LMD             Save new page of input pointer
                DCXH            Decrement table pointer
```


START
SAVE CPU REGISTERS AND STATUS
INPUT DATA
IS IN-PROGRESS FLAG SET?
NO
?
YES
STORE DATA IN BUFFER
DECREMENT DATA COUNTER
NO
=0?
YES
IS DATA = C/R?
NO
?
YES
RESET IN-PROGRESS FLAG
RESTORE CPU REGISTERS AND STATUS
ENABLE INTERRUPTS
RETURN

```
            LME         Save new low address of input pointer
            LAB         Fetch input character again
            CPI 215     Is it a carriage return?
            JFZ EXITIN  No, exit interrupt routine
FININP,     LXH FLAGIN  Set pointer to flag byte
            LMI 000     Reset in-progress flag
EXITIN,     POPB        Restore registers B and C
            POPD        Restore registers D and E
            POPH        Restore registers H and L
            POPS        Restore accumulator and status
            EI          Enable interrupts
            RET         Return to interrupted program

            . . .       Sequence to check in-progress
            . . .       Flag for end of operation
            LTA FLAGIN  Fetch in-progress flag
            NDA         Set flags after fetch
            JFS CMPTIN  Sign reset, input complete
            . . .       Sign set, continue other processing
```

Operation of an output device under interrupt control requires a different sequence of events from that for an input device. As pointed out before, the main reason for this difference is that the output device generates an interrupt after a character has been outputted by the program, while the input device generates an interrupt to indicate that it has a character available. The output routine presented next illustrates the different approach that must be taken for an output device.

This output interrupt routine outputs a string of characters stored in a buffer in memory. This routine may be used to output messages to a printer or video display, or to output data to a low or medium speed storage device. (High speed devices generally use a method of "direct memory access," in which the data is transferred directly from memory to the storage device, or vice versa, under control of a hardware interface.)

The interrupt output table is the same type of table used to provide the exchange of information between the main program and the input interrupt service routine. The organization of the table is the same as the input table, with the in-progress flag, output buffer pointer, and data counter. However, when the table is initialized, the buffer pointer is set to the actual start address of the output buffer,

rather than the start address minus one as in the input table.

Aside from setting up the table, the initialization routine checks the in-progress flag to determine whether an output is currently being executed. This may occur when a program uses the same output device to display messages from a number of different routines, such as error and advisory messages in a system monitor program. This check eliminates the possibility that an output will be initiated before a previous one is finished. This operation has not been included in the input routine since it is less likely that two separate inputs will be required at the same time. However, if the possibility does exist, a similar instruction sequence should be added to the input initialization routine before the disable interrupt instruction.

When the in-progress flag is reset, the output may be initiated. First, the output table is set up with the required information. While this table is being loaded it is not necessary to disable interrupts since the output device should not generate an interrupt until after the first character has been sent. Once the proper information is contained in the table, the first character is actually output by this routine. This output is performed by the OUT 002 instruction in this listing. This output starts the output sequence which is then carried on by the interrupt service routine. For implementation of this routine on one's own system, the instructions in this routine and in the interrupt service routine marked with the ** should be changed to the instruction sequence necessary to drive the specific output device used.

OUTPUT INTERRUPT TABLE

```
FLGOUT,     000             Output in-progress flag
            000             Low portion of output buffer pointer
            000             Page portion of output buffer pointer
BFRCNT,     000             Output buffer counter

                            Output initialization routine
            . . .
TSTOUT,     LTA FLGOUT      Input in-progress flag of output device
            NDA             Check in-progress flag
            JTS TSTOUT      If output in-progress, wait
            LXD BFROUT      Pointer to start of output buffer
```

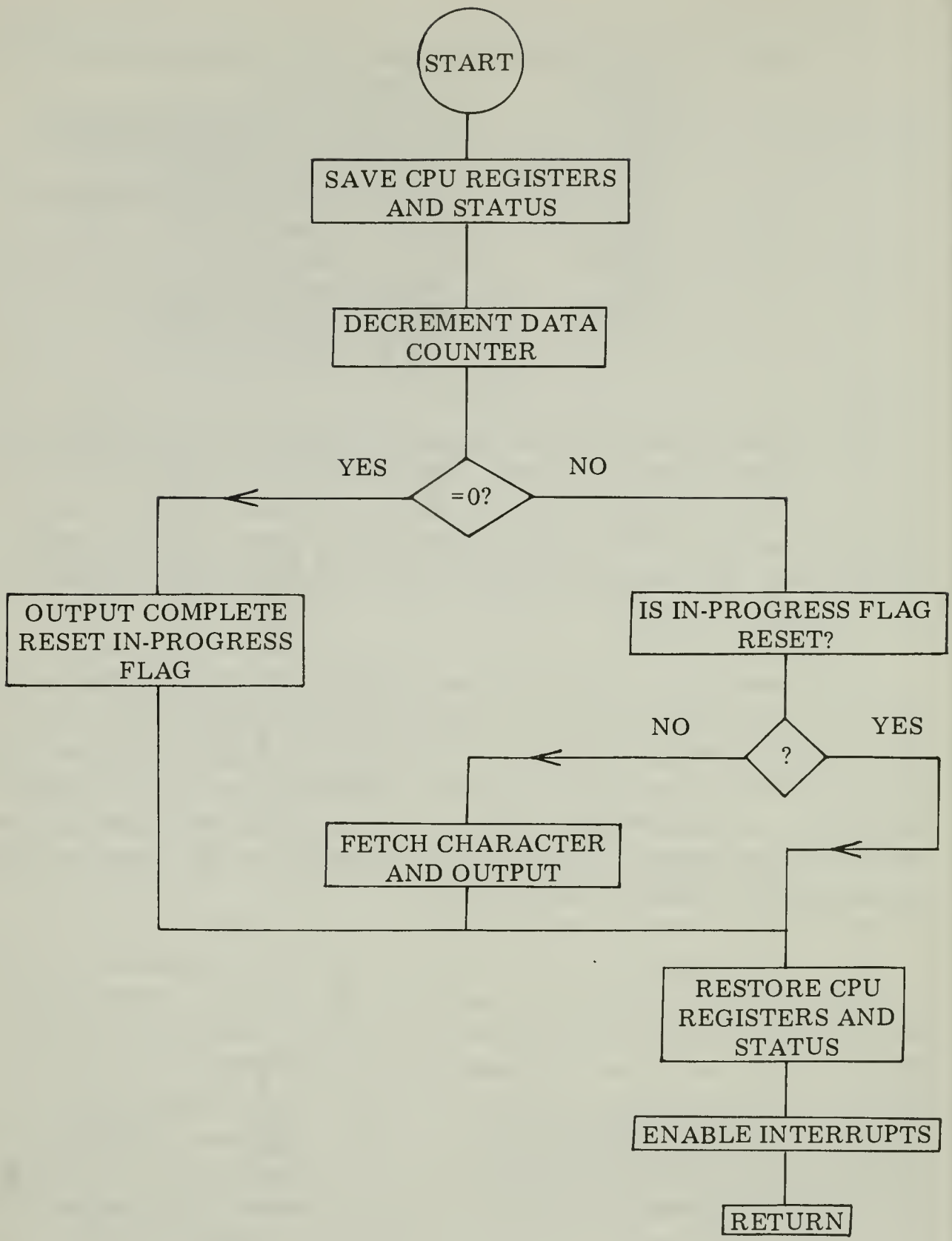
```
LXH BFRCNT      Set pointer to table
LMI XXX         Set data counter
DCXH            Decrement table pointer
LMD             Set page portion of output buffer
DCXH            Decrement table pointer
LME             Set low portion of output buffer
DCXH            Decrement table pointer
LMI 200         Set in-progress flag
LDAD            Fetch character to output
OUT 002      **  Output character to device
. . .           Continue main program
```

The output interrupt service routine is entered upon receipt of an interrupt from the output device. After storing the registers, the data counter is decremented once and checked for zero. When it reaches zero, the last character has been outputted, and the output operation is complete. The in-progress flag is reset and the routine returns to the interrupted program.

If the counter is not zero, the in-progress flag is checked to make sure that the output routine is expecting an interrupt. As in the input interrupt service routine, this is indicated by the in-progress flag being set. If it is reset, the interrupt may be ignored by simply returning to the interrupted program, or an error routine may be entered, which signals either the main program or the operator that an unexpected interrupt was received.

If the routine makes it by these tests, the next character may be outputted. In this routine it is assumed that only the output device can generate interrupts that direct the CPU to this restart location. Thus, a PDT test is not necessary before outputting the character. However, if it is felt that such a test should be performed before outputting a character, the required instruction sequence for testing the PDT bit may be included. The routine then restores the CPU registers and status and returns to the interrupted program. The listing and flow chart for this output interrupt routine are presented next.

```
ORG 000 0X0     Restart X location
PUSS            Save accumulator and status
PUSH            Save registers H and L
PUSD            Save registers D and E
```




```
                JMP INTOUT      Jump to output interrupt routine

                ORG XXX XXX     Start of output interrupt routine
INTOUT,         LXH BFRCNT      Set pointer to output counter
```

```
                DCM             Decrement counter
                JTZ FLGRST      =0? Yes, reset in-progress and exit
                DCXH            Decrement table pointer
                LDM             Fetch page portion of output buffer
                DCXH            Decrement table pointer
                LEM             Fetch low portion of output buffer
                DCXH            Decrement table pointer
                LAM             Fetch in-progress flag
                NDA             Is in-progress flag reset?
                JFS EXITOT      Yes, ignore interrupt
                INXD            Advance output buffer pointer
                LDAD            Fetch next character
                INXH            Advance table pointer
                LME             Save new low address of buffer pointer
                INXH            Advance table pointer
                LMD             Save new page of buffer pointer
                OUT 002     **  Output character
EXITOT,         POPD            Restore registers D and E
                POPH            Restore registers H and L
                POPS            Restore accumulator and status
                EI              Enable interrupts
                RET             Return to interrupted program

FLGRST,         XRA             Clear to reset flag
                STA FLGOUT      Reset in-progress flag
                JMP EXITOT      Exit interrupt routine
```

Up to this point, only one device has been considered to generate an interrupt on a restart level. When an interrupt is received, the interrupt service routine simply performs the indicated input or output for the single device. This may not always be the case, since an I/O controller quite often controls an input and an output device, and generates an interrupt on the same restart level for both devices. Or, there may be more than eight interrupting devices connected to the system, in which case it would be necessary to service several devices on the same level. In order to operate more than one device on an interrupt level, the interrupt service routine must determine which device generated the interrupt by "polling" each device when an interrupt is received.

Polling means that the interrupt service routine checks the status of each device that could have generated the interrupt on that restart level. This is done by checking the PDT bit of each of the devices assigned to that level. When a PDT bit is found to be set, the appro-

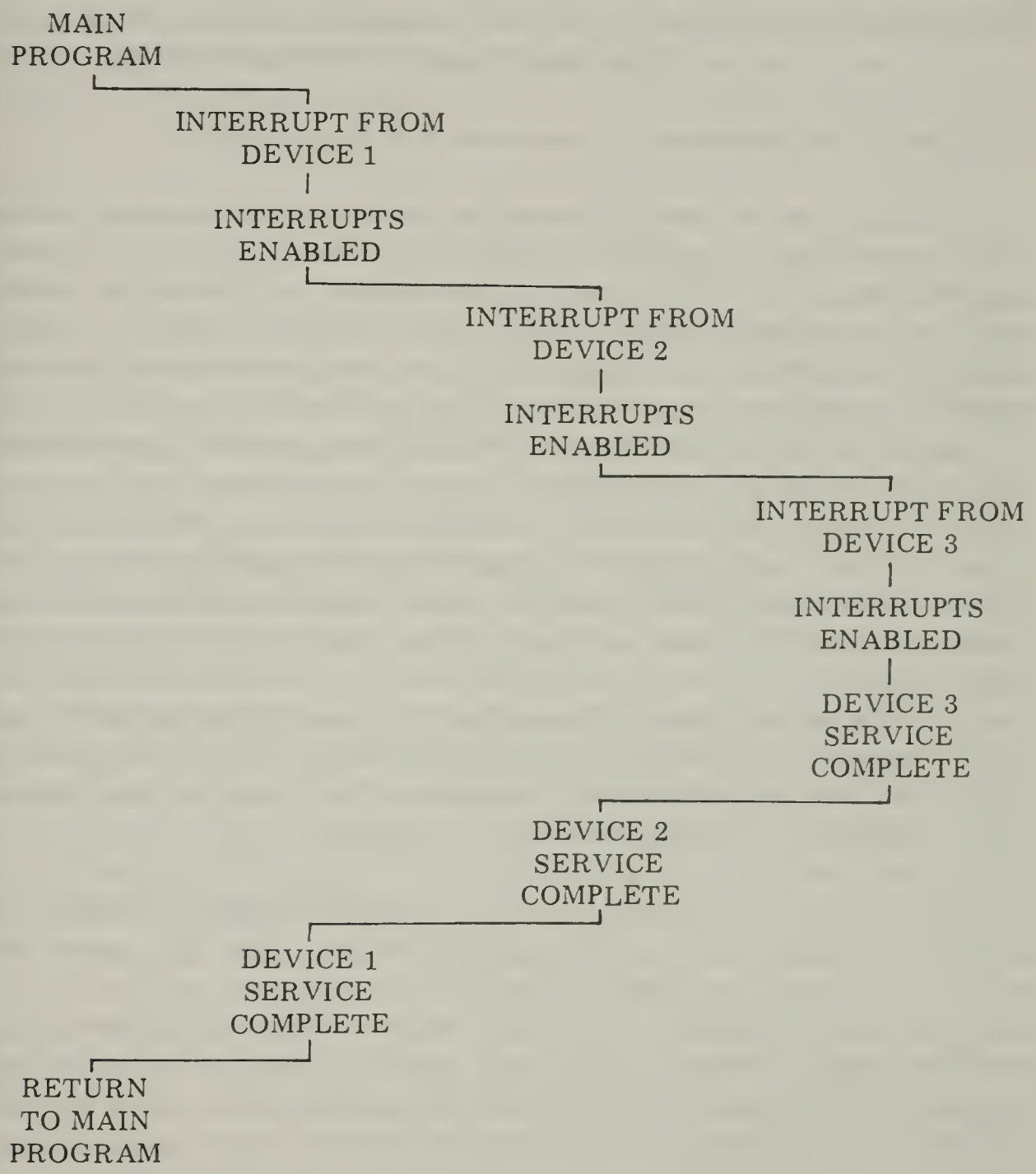
priate service routine is entered to execute the I/O for that device. At the conclusion of the service routine, the normal exit routine is performed to return to the interrupted program.

The following listing is an example of a polling routine that checks the status of three possible interrupting devices. This instruction sequence should be inserted in an interrupt routine just after the registers and status have been saved. The labels DVICE1, DVICE2, and DVICE3 refer to the interrupt service routines, which perform the I/O logic for the designated device. All three devices generate interrupts that call a common restart location. This routine tests the PDT bit of each device, assumed to be in the sign bit of the device status, and jumps to the proper service routine when a PDT bit is set. If none of the possible devices have the PDT bit set, this routine ignores the interrupt and returns to the interrupted program by jumping to the exit routine. This condition may be treated as an error condition, if necessary, rather than ignoring it.

```
. . .                   Polling routine
INP 000                 Input PDT of device 1
NDA                     Set flags to test PDT
JTS DVICE1              If PDT set, service device 1
INP 002                 Input PDT of device 2
NDA                     Set flags to test PDT
JTS DVICE2              If PDT set, service device 2
INP 003                 Input PDT of device 3
NDA                     Set flags to test PDT
JTS DVICE3              If PDT set, service device 3
JMP EXIT                None set, ignore interrupt
. . .
```

The use of several interrupting devices in a system may require the capability of the interrupt service routines to allow receipt of an interrupt from one device while another device is currently being serviced. This means that the service routine of the first interrupting device must enable interrupts before it has completed its operation. Then, if an interrupt from a second device occurs before this routine is finished, the current interrupt routine being executed becomes the interrupted program of the second interrupt. Allowing interrupts to overlap in this manner is referred to as NESTING interrupts.

The illustration below shows how the flow from one interrupt routine to another would proceed if three interrupting devices generated interrupts within a short period of time, to create the nesting of three levels of interrupts.



The 8080 stack plays an important role in nesting interrupts. Saving the return addresses and the CPU registers and status in the stack, as each interrupt is received, allows the interrupt routines to

interrupt each other without setting up special pointers and data storage areas for each interrupt level or device. The only restriction on the number of nesting levels, as far as the stack is concerned, is the amount of memory provided for use by the stack. An interrupt routine that saves all the registers uses 10 (decimal) memory locations in the stack. Therefore, for every interrupt nesting level one can expect, there must be 10 (decimal) memory locations available in the stack, plus that required for the other uses of the stack by the main program (i.e. subroutine calls, temporary data storage).

Deciding when to enable interrupts in an interrupt service routine to allow nesting is generally determined by the speed of the device being serviced. A low speed device may allow interrupts to be enabled immediately upon entering the interrupt service routine, since it is likely that the data transfer is not required immediately. A medium speed device, or a device that has a limited amount of time to transfer the data, may require the data transfer to be performed before the interrupts are enabled. Then, the remainder of the service routine may be executed while the interrupts are enabled. If a high speed device is being operated in the interrupt mode, it is very likely that it should not allow interrupts to be enabled until the end of its service routine. If it were to enable interrupts, a slower device might delay the execution of the high speed device's service routine to the point that a second interrupt from the high speed device would be received before the initial interrupt had been serviced completely. Therefore, one should carefully consider which routines, and where in the routines, the interrupts are to be enabled.

This method of selecting when to enable interrupts is a means of setting a priority for the interrupting devices. The high speed devices would have the highest priority, since they do not allow themselves to be interrupted until the service routine is finished. The medium speed devices, which may enable interrupts after several operations of the service routine have been completed, would be considered a middle priority. The low speed devices would be the lowest priority because they may be interrupted at any time during the interrupt service routine.

This system of priorities may be augmented by the computer's hardware if a priority interrupt interface is used, which fields the

interrupts from the interrupting devices and allows the higher priority interrupts through first, before those of lower priority. The general rule used by these interfaces is that the higher priority devices use the higher numbered restart locations, and the lower priority devices use the lower numbered restart locations. This type of interface greatly eases the burden on the interrupt software of setting up priorities for the interrupts received.

When deciding whether to operate a computer system's peripherals under PDT control or interrupt, one should consider the type of programs (along with the number of peripherals) to be used in the system. If the programs are the type that receive an input and then output a response to a single terminal, the PDT mode would be the easiest to implement, and would provide sufficient performance. Programs of this type include games, editors, and small system monitors. For programs that provide keyboard entry and storage or retrieval from a bulk storage device to enter and store mailing lists, for example, one should consider interrupt processing. This would allow the data entry and bulk storage to be performed simultaneously. However, if the speed of the storage device is fast enough to store each entry with a minimal delay imposed between entries, the PDT mode may work just as well. For programs that operate a number of peripherals simultaneously and, in essence, are running more than one or two programs at a time, the interrupt mode of operation is a necessity. Such "multi-programmed" systems might be used to control several terminals at once, while monitoring a burglar or fire alarm system. Therefore, one should carefully consider the overall requirements for the type of programs to be run when setting up the I/O portions of one's system.

SEARCH AND SORT ROUTINES

The capability of a computer to manipulate data stored in its memory is another reason why the computer is such a powerful machine. The speed with which it can search large blocks of data and extract information, or sort the data into alphabetical order, or into common groupings, is far beyond the capability of a human. The information may represent a wide variety of data. For example, the data may be lists of names and addresses that are to be searched for specific geographic regions or sorted into alphabetical order. Or, the data may be numerical information, such as test grades or data gathered for a research project, which is to be analyzed to obtain the average or derive other information about the data through statistical analysis. In order for the computer to perform these tasks, the information to be processed must be arranged in memory in a specific format, and programs must be written to perform the desired operation.

The data to be manipulated must be arranged in some form of table in memory. The table may contain a number of entries. Each entry may consist of one or more bytes of memory, depending on the maximum size of a single entry and the format specified for an entry. The two types of tables to be discussed in this chapter are commonly referred to as FIXED FORMAT and FREE FORMAT tables. In a fixed format table, the data is arranged in a standard fashion for each entry. The same number of bytes is assigned to each entry, no matter how many bytes an entry may actually take up. A free format table allows the size of each table entry to follow the data pattern of the entry. If the first entry requires four bytes and the next requires six, in a free format table, the first entry will only use the four bytes and the second will use six. There are advantages to both formats, depending on the application. These will be discussed as the search routines for each format are presented.

In order to provide a means of comparing the two formats, two search routines will be presented that perform the same function. However, one routine utilizes a search table of fixed format and the other utilizes a free format search table. The function performed by these search routines is that of receiving a command input from a

keyboard and searching a control table for the same command. If the command is found, the start address of the command routine is taken from the table and is used to jump to that command routine. If the command is not found in the table, the search routine returns to wait for another command input. These routines have many practical applications since, as will be seen, the entries in the control table may be easily modified to represent as many commands as one may need for a specific program. Any program that allows an operator to input commands to direct its operation may find one of these routines useful.

The following control tables shall be used to illustrate the operation of the fixed format and free format search routines. The commands in these tables are: GO, LIST, MEDIAN, AVG, COUNT, and CLEAR. These commands might be used to direct the computer to aid in conducting an experiment. The GO command could initiate a ten second sampling interval, during which time a sensor is monitored to detect the occurrence of an event. A count of the number of times the event occurs within the ten second interval is stored in the computer by the GO command routine. The LIST routine might be used to print out the counts stored for each ten second interval up to that time, allowing one to examine the raw data for possible patterns that may develop. The MEDIAN and AVG commands could calculate the median and average values of the counts stored for each interval, and output the value to a printer. The COUNT command might be used to indicate the number of ten second intervals that have been initiated up to that time. The CLEAR command could be used to reset the storage area to allow a new set of tests to begin.

In both the fixed format and free format control tables, each entry is divided into two FIELDS. The first field consists of the character string that defines the command name. In the fixed format entry, this field is set to a fixed length. In this case, it is six characters long. For the command names that do not use all six locations available for the name, the unused locations are filled with zeros. In the free format entry, the command field contains the characters for the name plus one more location that is filled with zeros. This extra location is used to indicate the end of the name, as will be discussed shortly. The second field is the same for both formats. This field is two bytes long and contains the start address of the command

in the entry.

The end of each control table is indicated by a zero byte stored immediately following the last entry. By terminating the tables in this way, the search routines may simply check the first character of each entry for a zero byte to determine when the end of the table is reached. Therefore, the number of entries in the control table is completely independent of the operation of the search routine. Modifying the number of commands in the control table is accomplished by adding or deleting the command entries and moving the zero byte to the end of the new control table. The search routine does not have to be changed at all.

The control table for the fixed format search routine is presented next, followed by the control table for the free format routine. One should make note of the differences in length of the two tables, due to the extra zeros that must be added to the fixed format entries. This and other points to be considered when deciding which format to use will be discussed following these tables.

FIXED FORMAT CONTROL TABLE

02 000	307	Code for letter G
02 001	317	Code for letter O
02 002	000	Not used for this command
02 003	000	Not used for this command
02 004	000	Not used for this command
02 005	000	Not used for this command
02 006	001	Page where GO routine starts
02 007	100	Location on page where GO starts
02 010	314	Code for letter L
02 011	311	Code for letter I
02 012	323	Code for letter S
02 013	324	Code for letter T
02 014	000	Not used for this command
02 015	000	Not used for this command
02 016	001	Page where LIST routine starts
02 017	140	Location on page where LIST starts
02 020	315	Code for letter M
02 021	305	Code for letter E
02 022	304	Code for letter D
02 023	311	Code for letter I
02 024	301	Code for letter A

02 025	316	Code for letter N
02 026	001	Page where MEDIAN routine starts
02 027	200	Location on page for MEDIAN
02 030	301	Code for letter A
02 031	326	Code for letter V
02 032	307	Code for letter G
02 033	000	Not used for this command
02 034	000	Not used for this command
02 035	000	Not used for this command
02 036	001	Page where AVG routine starts
02 037	240	Location on page where AVG starts
02 040	303	Code for letter C
02 041	317	Code for letter O
02 042	325	Code for letter U
02 043	316	Code for letter N
02 044	324	Code for letter T
02 045	000	Not used for this command
02 046	001	Page where COUNT routine starts
02 047	300	Location on page where COUNT starts
02 050	305	Code for letter E
02 051	322	Code for letter R
02 052	301	Code for letter A
02 053	323	Code for letter S
02 054	305	Code for letter E
02 055	000	Not used for this command
02 056	001	Page where ERASE starts
02 057	340	Location on page where ERASE starts
02 060	000	**End of table marker**

FREE FORMAT CONTROL TABLE

02 000	307	Code for letter G
02 001	317	Code for letter O
02 002	000	*End of command word marker*
02 003	001	Page where GO routine starts
02 004	100	Location on page where GO starts
02 005	314	Code for letter L
02 006	311	Code for letter I
02 007	323	Code for letter S
02 010	324	Code for letter T
02 011	000	*End of command word marker*
02 012	001	Page where LIST routine starts
02 013	140	Location on page where LIST starts
02 014	315	Code for letter M
02 015	305	Code for letter E
02 016	304	Code for letter D

02 017	311	Code for letter I
02 020	301	Code for letter A
02 021	316	Code for letter N
02 022	000	*End of command word marker*
02 023	001	Page where MEDIAN routine starts
02 024	200	Location on page for MEDIAN
02 025	301	Code for letter A
02 026	326	Code for letter V
02 027	307	Code for letter G
02 030	000	*End of command word marker*
02 031	001	Page where AVG routine starts
02 032	240	Location on page where AVG starts
02 033	303	Code for letter C
02 034	317	Code for letter O
02 035	325	Code for letter U
02 036	316	Code for letter N
02 037	324	Code for letter T
02 040	000	*End of command word marker*
02 041	001	Page where COUNT starts
02 042	300	Location on page where COUNT starts
02 043	305	Code for letter E
02 044	322	Code for letter R
02 045	301	Code for letter A
02 046	323	Code for letter S
02 047	305	Code for letter E
02 050	000	*End of command word marker*
02 051	001	Page where ERASE starts
02 052	340	Location on page where ERASE starts
02 053	000	**End of table marker**

As mentioned before, the length of the two tables differs because of the variation in the number of characters for each command name. If, however, all of the names were six characters long, the fixed format table would be shorter than the free format, since the command name field in the free format table would require seven bytes to store each name - six for the name and one for the terminating zero byte.

Another consideration when deciding which format to use is the type of input programming required to enter the commands. There are several different methods that may be used to input and store the command to be searched for in the control table.

One method is to initially clear out the input buffer area by filling

it with zero bytes. Then, as each character is entered, it is stored in the input buffer. When a carriage return is entered, the input is terminated and the contents of the input buffer may be used to search for the command. If the command entered does not fill the input buffer, the unused locations will contain zero bytes. This method is best suited for the fixed format, since the input buffer will contain the same contents as the command name field of the matching command in the control table.

The following routine could be used to clear the input buffer and store the characters in the buffer as discussed above. This routine uses the CLRMEM subroutine in chapter three to clear the input buffer. The INPUT routine, which is called, must input a character from the input device (such as an ASCII keyboard) and return with the character in the accumulator. Along with the test for the carriage return, to terminate the input and return, the character count is checked and when the input buffer is full, any additional characters that may be inadvertently entered before the carriage return are ignored. The initial instruction sequence may be used as a control routine to call the individual routines, including the search routine to be presented later.

```
                                Main program calling sequence
NEXCMD,     CAL CLEARB          Clear input character string buffer
            CAL INCMND          Fetch the cmd string fm input device
            CAL SRCHFX          Search table & perform cmd inputted
            JMP NEXCMD          Repeat loop for next cmd by operator

                                Clear input buffer subroutine
CLEARB,     LXH INPBFR          Set page pointer to start of buffer
            LBI 006             Set clearing counter
            CAL CLRMEM          Clear the input buffer
            RET                 Return to calling program

                                Fetch input command string
INCMND,     LXH INPBFR          Set pointers to start of input buffer
            LBI 006             Set cntr for maximum size of buffer
INCHAR,     CAL INPUT           Call subroutine to input chars fm I/O
            CPI 215             See if character was a carriage return
            RTZ                 If so make no entry
CHECK,      INB                 Exercise register B (cntr) to set flags
            DCB                 According to original contents
            JTZ INCHAR          Ignore new character if cntr was 000
```

```
        DCB         Otherwise decrement value of counter
        LMA         And store value in buffer
        INXH        Advance buffer pointer
        JMP INCHAR  And loop to fetch next char from I/O
```

Another method of inputting the characters is to leave the input buffer contents as is at the start of the input routine. As each character is received, it is stored in the input buffer. When a carriage return is received, the input is terminated by storing the carriage return in the input buffer and returning. Thus, the input buffer area must be assigned one byte more than the maximum number of characters assigned for a command name. This method is more advantageous for the free format search routine because it sets up the command entered in a similar format to that used in the command name field of the free format control table entries.

An input routine that operates in this manner is listed next. The only real difference between this routine, labeled INCTRL, and the previous INCMND routine is the instruction sequence that stores the carriage return as the terminating character in the input buffer before returning. Also, one should note the absence of the routine that clears the input buffer before inputting the command. This saves quite a few memory locations. As in the previous listing, the initial instruction sequence is a sample control routine for directing the operation of the command search function.

```
                            Main program calling sequence
NEXCMD,     CAL INCTRL      Fetch the cmd string fm input device
            CAL SRCHFR      Search table & perform cmd inputted
            JMP NEXCMD      Repeat loop for next cmd by operator

                            Fetch input command string
INCTRL,     LXH INPBFR      Set pointer to start of character buffer
            LBI 006         Set cntr for maximum no. usable chars
INCHAR,     CAL INPUT       Call subroutine to input char fm I/O
            CPI 215         See if character was a carriage return
            JFZ CHECK       If not, check for buffer overflow
            LMA             If yes, store C/R as last char in buffer
            RET             And return to calling routine
CHECK,      INB             Exercise register B (cntr) to set flags
            DCB             According to original contents
```

```
JTZ INCHAR      Ignore new character if cntr was 000
DCB             Otherwise decrement value of counter
LMA             And store character in buffer
INXH            Advance buffer pointer
JMP INCHAR      And loop to fetch next char from I/O
```

The search routine for a fixed format control table, listed next, compares the contents of the input buffer to the command name field on a character-by-character basis. This is done by using a compare sequence, beginning at the CMATCH label, which is similar to the CPRMEM subroutine presented in chapter three. This routine is included in the search routine sequence, rather than calling it as a subroutine, to conserve memory.

If the characters in the input buffer do not match the command name field of an entry in the control table, the NXWORD routine is entered to advance the control table pointer to the start of the next entry. At this point, the first character of this entry is checked for the zero byte, which indicates the end of the control table. If the zero byte is not found, the routine jumps to the compare routine to check for a match between the new control entry and the input buffer. When the zero byte is encountered, it indicates that the entire table has been searched and no match has been found. The routine then returns to the control routine to initiate a new command entry.

It may be desirable at this point to have the search routine print out a message if no match is found, to inform the operator of the error. This may be done by changing the RTZ instruction in the SETNXW routine to a JTZ, which jumps to a message output routine to print the message before returning to the main control program.

When a match is found, the FOUND routine is entered. This routine takes the address from the address field of the matching control entry and uses it to jump to the command routine by placing the address in registers H and L, and then transferring H and L to the program counter, using the PCHL instruction. After the command routine completes its operation, it may return to the main control program by simply executing a return instruction.

Since this routine compares the entire input buffer against the

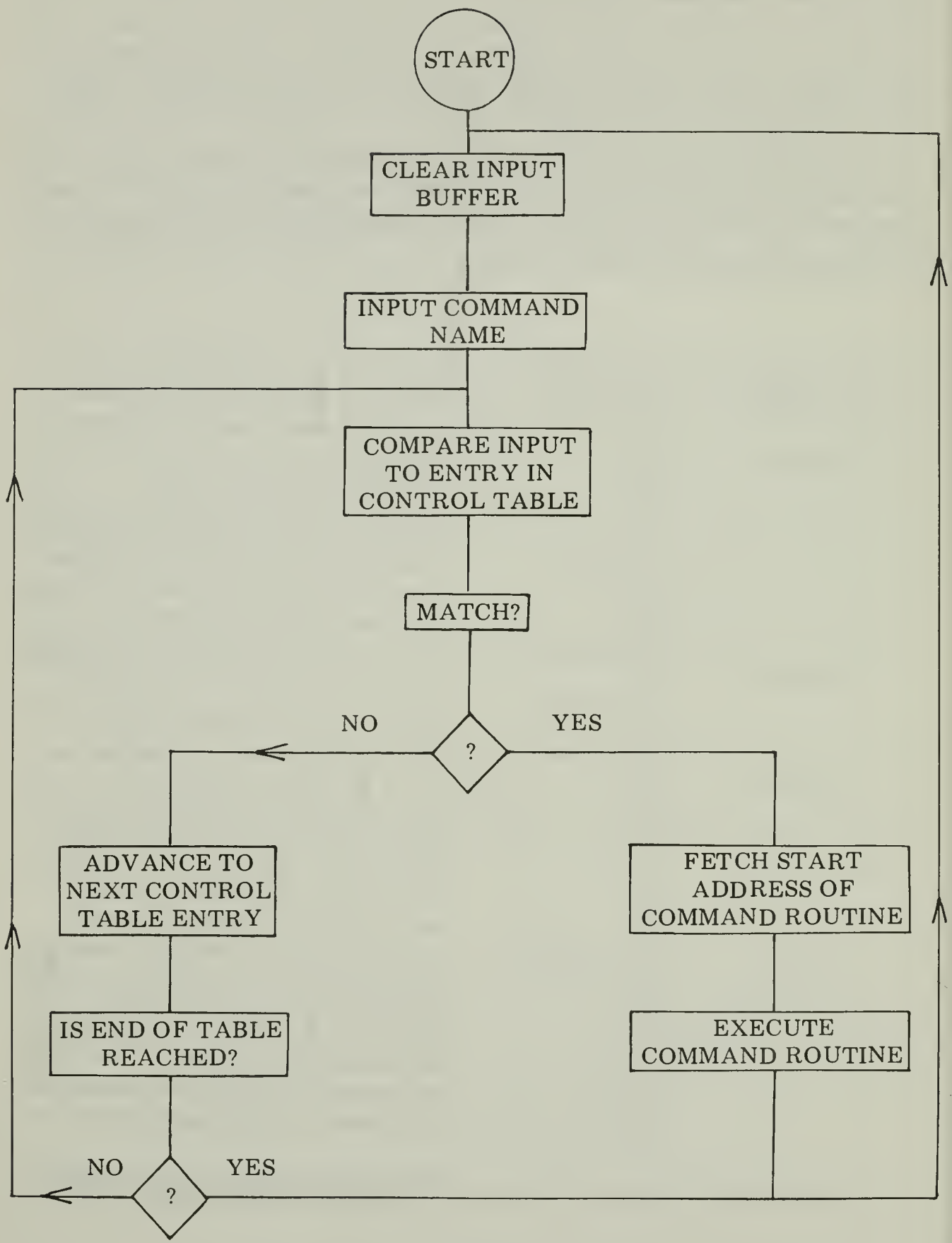
command name field of the control table entries, it is not necessary for it to test the input buffer or command name for a terminating character. However, a counter must be set to the number of characters in the command name field so that the routine will know when all of the characters have been compared. In this routine, this counter is set to six. If one changes the length of the command name field, this counter must also be changed to reflect the new length.

The listing for this fixed format search routine is presented next, followed by the flow chart. The flow chart also includes the logic flow of the main control routine, previously presented, when used in conjunction with this search routine.

```
SRCHFX,   LXH CMDTBL    Set pointer to starting address of table
INITBF,   LXD INPBFR    Set pointer to start of input buffer
          LBI 006       Set control word field size counter
CMATCH,   LDAD          Get char fm bfr (form char match loop)
          CPM           See if have character match
          JFZ NXWORD    If no match, go to next block in table
          DCB           If match, decrement field size counter
          JTZ FOUND     If cntr =0, all chars in field matched
          INXH          Char mtched but not finished, adv pntr
          INXD          Increment input buffer pointer
          JMP CMATCH    Loop to see if next character matches
NXWORD,   DCB           Decr field size counter to find end of
          JTZ SETNXW    Current control field, jump when find
          INXH          Otherwise advance table pointer
          JMP NXWORD    And loop to look for end of CW field
SETNXW,   INXH          At end of control word need to
          INXH          Adv pntr over the ADDRESS field
          INXH          To start of next control word field
          LAM           And fetch 1st character in new block
          NDA           Set the flags after the load operation
          RTZ           Return if end of table (no match)
          JMP INITBF    Loop back to check block in table
FOUND,    INXH          Adv pntr to 1st part (page) of address
          LDM           And extract page to store temp
          INXH          Adv pntr to location on page address
          LEM           And store it temporarily
          XCHG          Set mem pointer (H&L) to jump addr
          PCHL          Now jump to address just loaded
```

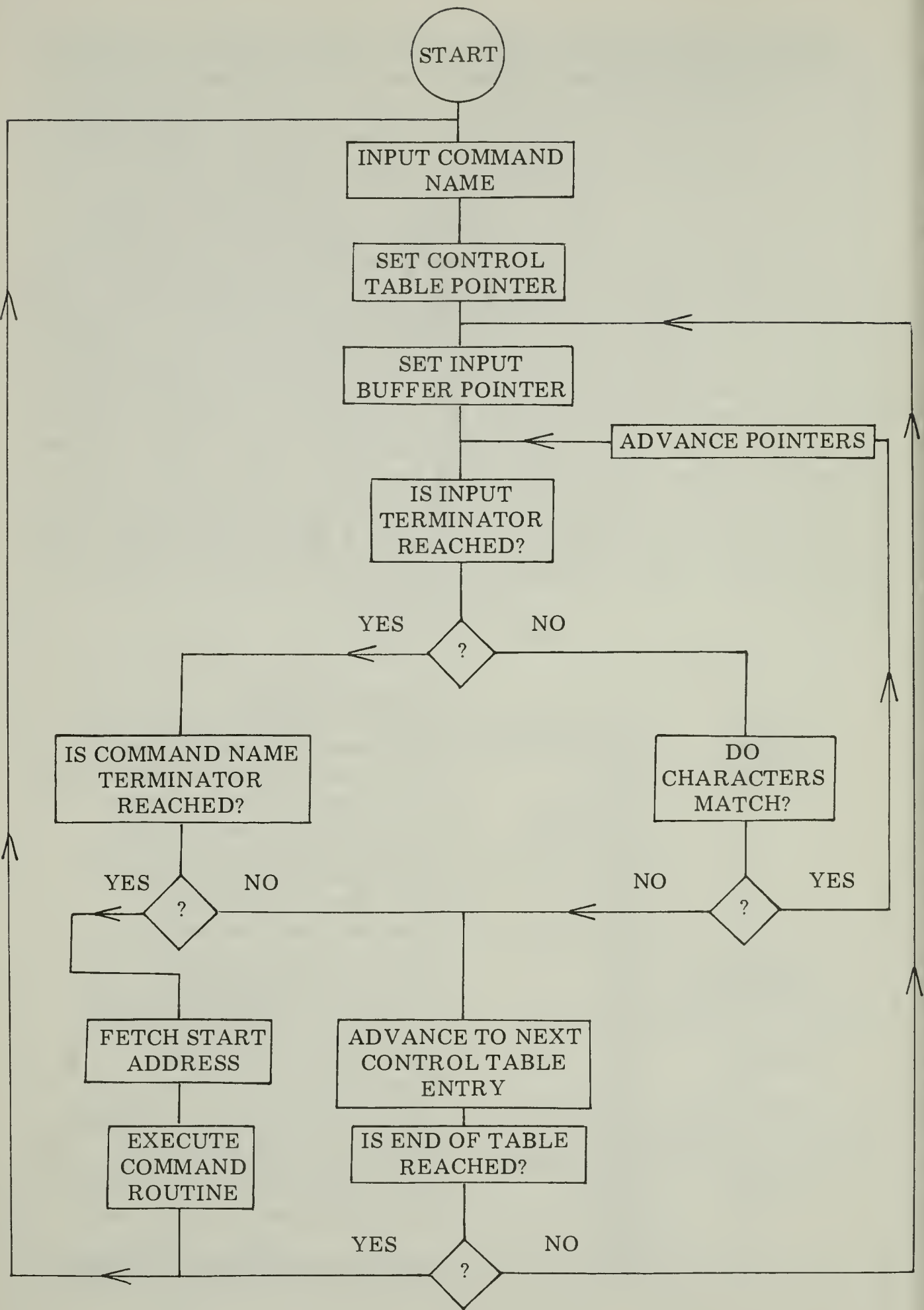
At the conclusion of the routine that the SEARCH routine jumps to when a

match is found, a RET instruction should be executed to return the program to the main calling routine



The free format search routine which follows has the same basic flow of the fixed format routine. That is, it compares the input buffer to an entry in the control table and, if they do not match, it advances the control table pointer to compare the next entry against the input buffer. This continues until either a match is found or the end of the buffer is reached. But instead of comparing a set number of characters to get a match, this routine compares the contents of the input buffer, up to the terminating carriage return, against each command name field in the control table. If the input buffer and command name field match up to the carriage return in the input buffer, the corresponding location in the command name field is checked for its terminating character, a zero byte. If the zero byte is found, the same FOUND routine as in SRCHFX is entered to fetch the address from the control table entry and jump to the proper command routine. Here again, if the suggested main control routine for the free format table is used, the command routines may return to the main control routine by executing a return instruction. The listing for this search routine is shown next, followed by the flow chart.

```
                                Table search routine
SRCHFR,   LXH CMDTBL    Set pointers to starting addr of table
INITBF,   LXD INPBFR    Set pointers to start of char buffer
CMATCH,   LDAD          Get char fm bfr (form chr match loop)
          CPI 215       See if symbol for carriage return
          JTZ LCHAR     If so, go to last character routine
          CPM           See if have match condition in table
          JFZ NXWORD    If not, go to next block in table
          INXH          If yes, advance table pointer
          INXD          Otherwise advance buffer pointer
          JMP CMATCH    Loop to test next character
LCHAR,    LAM           Test for end of control field
          NDA           By seeing if two MSB's are both 0
          JTZ FOUND     If so, found matching control word
NXWORD,   LAM           Test for end of control field
          NDA           By seeing if two MSB's are both 0
          JTZ SETNXW    If so, have marker, go to next block
          INXH          Otherwise, advance table pointer
          JMP NXWORD    And continue looking
SETNXW,   INXH          At end of control word field need to
          INXH          Adv pntr over the ADDRESS field
          INXH
          LAM           And fetch 1st character in new block
```


START
INPUT COMMAND NAME
SET CONTROL TABLE POINTER
SET INPUT BUFFER POINTER
ADVANCE POINTERS
IS INPUT TERMINATOR REACHED?
YES
NO
?
IS COMMAND NAME TERMINATOR REACHED?
DO CHARACTERS MATCH?
YES
NO
NO
YES
?
?
FETCH START ADDRESS
ADVANCE TO NEXT CONTROL TABLE ENTRY
EXECUTE COMMAND ROUTINE
IS END OF TABLE REACHED?
YES
NO
?

```
        NDA         Set flags after load operation
        RTZ         Ret if end of table (no match found)
        JMP INITBF  Loop back to check next block in tbl
FOUND,  INXH        Advance table pointer to address
        LDM         Fetch page address
        INXH        Advance pointer
        LEM         Fetch location on page
        XCHG        Now set mem pntr (H&L) to jmp addr
        PCHL        Jump to the address

                    After processing cmd, ret to main rtn
```

The process of sorting data into some specified order, or into groups with common attributes, is another of the computer's powerful capabilities. For example, it is often desired to arrange a list of names and addresses in alphabetical order according to the last names. Or, one may want to sort out persons living in a common geographical location by sorting the addresses according to the zip code. In order to program these types of sort functions, the data must first be organized into carefully structured tables.

Proper structuring may be accomplished by creating fields in which specific items are to be located. These fields are set up in a similar manner as the fields in the search table entries. For the sort routine to be discussed, a fixed format table will be used. This table will contain a list of names which are to be arranged in alphabetical order. For illustrative purposes, the following names will be used as table entries.

BROWN, L.R.
DAMATO, P. J.
ANDERSON, B.
DARBY, P.
MATTOX, R. T.
MATTHEWS, K. D.
JONES, A. M.

Each element of the names in this list must have a specific field assigned to it. Looking at the last names, one may observe that the longest name is eight characters long. However, there are many names that use more than eight characters. Therefore, a field of four-

teen bytes for the last name will be used to accommodate the longer names. One byte fields will be set up for each of the initials. This makes a total of sixteen bytes for each entry. The delimiters (the comma and period) are not assigned any location in the tables since a fixed format is to be used and the inclusion of these punctuation marks would only serve to take up table space. The delimiters are used though, when entering the names to be stored in the table.

The following program is one possible means of entering the names into the table in the properly formatted fields. This routine accepts the names as keyboard entries in the format illustrated above. Each field is accepted and stored in its proper location in the table for that entry. A name input is terminated by entering a carriage return. The delimiters (comma and period) terminate a field entry and advance the pointer to the next field in the entry. If a field is not filled up by the name, such as a last name with less than fourteen characters or one with no middle initial, the unused locations in the field are filled with zero bytes. After the final name has been entered, the operator may input an asterisk to indicate the completion of the input operation.

The character input is expected to be ASCII characters. As pointed out in the chapter on conversion codes, the ASCII character set is a well ordered set of character codes. The letters A through Z are coded in consecutive order from 301 to 332 (octal), and the numbers 0 through 9 are coded in order from 260 to 271 (octal). This is especially useful when sorting into alphabetical or numerical order, since the lower the code for the character, the lower it will place the entry in the sorted list.

Before listing this routine, several comments about its operation must be given. First, the characters are received by calling a routine referred to by the label INPUT. This routine must be provided by the user to accept characters from the input device associated with one's system. This routine must return the ASCII code for the character inputted in the accumulator upon returning. As a visual indication for the operator to verify the characters inputted, this INPUT routine should also output the characters received to the system display device before returning. The contents of registers H and L must not be altered by this routine. If H and L must be used, to point to a

conversion table or whatever, they should be saved and then restored before returning. One should refer to chapter seven on input and output processing for methods of programming this INPUT routine.

Before this routine is called, the table area must be properly set up. This is accomplished by storing a zero byte in the first location in the table. The zero byte is used to indicate the location for storing the next name inputted. Initially, this byte must be set up at the start of the table by the calling program. Then, as each name is entered into the table, the zero byte is moved up to the location immediately following the last name entered in the table. Before each name input is initiated, the location of this byte in the table is checked. If the zero byte is not found within the limits of the table area, it is assumed that the table is filled. At this point, a routine, to be supplied by the user and referred to here as TOMUCH, would be entered. TOMUCH should output a message to the operator indicating that the table is filled. The limits of the table area in this sample routine begin at page 04 location 000, and end at page 07 location 377. Thus, when the table pointer is advanced to the start of page 10, the table is full.

After the zero byte is found and the program is ready to accept a name input, it calls the input routine to fetch the first character. There are a number of special codes checked when this first character is entered. One is an asterisk, which is to be inputted when the list contains all of the names desired. When this character is received, the routine returns to the calling program, which may then call upon the sort routine to sort the names into alphabetical order. The other special codes checked for are the carriage return, comma, or period. If any of these codes are received as the first character, they are ignored. This is because they indicate either the end of an entry or field, which would not be valid at this time since there are no characters stored as yet for this name.

Once a valid character is entered for the first character, the characters that follow are checked for a comma or carriage return. If the comma is received, the remainder of the last name field is filled with zero bytes, and the portion of the routine that accepts the initials is entered. If a carriage return is received, the remainder of the entire entry is filled with zeros, and a new name input is initiated. If the

character entered is neither of these two characters, it is entered as the next character in the last name, up to the fourteenth character. If more than fourteen characters are entered for the last name, the excess characters are simply ignored.

If when the first initial is to be entered, a carriage return is received, the two initial fields are zeroed and a new name input is begun. If a comma is received, it is ignored. Otherwise, the character is stored as the first initial and the routine jumps to input the second initial. When the second initial is to be entered, receipt of a period is ignored, and a carriage return results in a zero byte being stored for the second initial. Any other input is stored as the second initial. The routine then checks for a full table and, if not filled, begins a new name input sequence.

```
        ACCEPT,   LXH SRTTBL      Initialize sort table pointer
        NOTFND,   LAM             Fetch first location of an entry
                  NDA             Set flags after load operation
                  JTZ FNDEND      And test for end of storage area
                  LXD 020 000     If not end, advance pointer
                  DADD            To next entry by adding 020 to pntr
        CKPAGE,   CAL CKPAGS      Go see if out of area boundary (04-07)
                  JMP NOTFND      Keep looking for end of storage area
        FNDEND,   LBI 016         Set up last names field counter
                  CAL INPUT       And fetch character from input rtn
                  CPI 252         Check for * code (finished indicator)
                  JFZ NOTDON      Proceed if not * code
                  XRA             If * code, place a 000 byte at
                  LMA             Start of block as an ending marker
                  RET             And exit subroutine
        NOTDON,   CPI 215         Test for carriage return code
                  JTZ FNDEND      And ignore if 1st character in field
                  CPI 256         Test for (.) code
                  JTZ FNDEND      And ignore if 1st character in field
                  CPI 254         Test for (,) code
                  JTZ FNDEND      And ignore if 1st character in field
                  LMA             If none of above, put char in field
                  DCB             Decrement the field size counter
                  INXH            Advance storage pointer
    *   NEXTIN,   CAL INPUT       And fetch the next char in last name
                  CPI 215         Test for carriage return
                  JTZ HAVECR      Finish block if have C/R here
                  CPI 254         Test for comma
                  JTZ HAVECM      Finished last name field if have comma
```

```
                LMA         Otherwise place char in last name field
                INXH        Advance storage pointer
                DCB         Decrement last names field size cntr
                JTZ FULFLD  And see if field is filled
                JMP NEXTIN  If not, get next character in last name
    HAVECR,     XRA         If have C/R, put a 000 byte in mem
                LMA         For rest of current entry
                LAL         Fetch memory pointer to accumulator
                NDI 017     Mask off 4 most significant bits
                JTZ NOTFND  Prepare for next entry if done
                INXH        Otherwise advance pointer
                CAL CKPAGS  Go check if out of upper boundary
                JMP HAVECR  & continue putting 000 bytes in entry
    HAVECM,     XRA         If have comma put 000 butes in rest
                LMA         Of last name field
                INXH        Advance field pointer
                DCB         Decrement last names field counter
                JFZ HAVECM  If not done, continue to clr rest of fld
*   FULFLD,     CAL INPUT   Get character for 1st initial of name
                CPI 254     Test for comma
                JTZ FULFLD  Ignore comma at this point
                CPI 215     Test for carriage return
                JFZ SAVIN1  If not carriage return, store character
                XRA         But, if carriage return, put in 000 byte
                LMA         For both initial fields
                INXH        By above instruction, then adv pntr
                JMP SAVIN2  And then following this jump cmd
    SAVIN1,     LMA         Store 1st initial in 1st initial field
                INXH        Then advance storage pointer
*   INITF2,     CAL INPUT   Look for second initial
                CPI 256     Check for period
                JTZ INITF2  Ignore a period
                CPI 215     Test for a carriage return
                JFZ SAVIN2  If not C/R store second initial
                XRA         If was C/R place 000 byte in memory
    SAVIN2,     LMA         Store character or 000 substitute
                INXH        Advance pointer to next entry
                CAL CKPAGS  Go check if out of page boundary
                JMP FNDEND  If not, go process next input
    CKPAGS,     LAI 010     Test to see if still in
                CPH         Storage area (pages 04-07)
                JTZ TOMUCH  Display message if table filled
                RET
```

It may be desired to provide some kind of entry correction capability to this name input routine. One way to accomplish this might

be to designate another special code that, when received, would reset the table pointer to the start of the current entry and initiate a new name input. The subroutine listed below may be called immediately following the instructions marked with the asterisk. This subroutine (labeled CHKRUB) checks for a RUBOUT code (377 octal) and, if entered, resets the table pointer, essentially erasing the current name input from the table, and jumps to the start of the name input sequence at the FNDEND label. Otherwise, it simply returns to continue the input. The first instruction listed is the call to be inserted after the instructions marked by the asterisk.

```
            CAL CHKRUB      Insert after * marked instructions

CHKRUB,     CPI 377         Check for rubout code
            RFZ             If not rubout, return
            INXS            Advance stack pointer to
            INXS            Eliminate latest return address
            LAL             Fetch low address of table pointer
            NDI 360         Remove 4 least significant bits
            LLA             Restore table pointer to start
            JMP FNDEND      Of table entry and begin new input
```

Now that one has defined the format for storing the data, and developed a means of entering it, a routine may be written to sort the data as desired. The main objective of the sort routine is to examine the contents of the field (or fields) that contains the information pertinent to the sort operation, and rearrange the table contents into the desired order or groups. There are a number of techniques used to do this, the choice of which generally depends on the type of data and sorting operation to be performed. The sort routine presented next arranges the table contents into alphabetical order by using what is referred to as a "ripple" sorting technique.

The term ripple is derived from the manner in which the routine moves through the table to sort the entries into alphabetical order. Beginning with the first entry (N), the sort routine compares it to the next entry (N+1) in the table. If the first entry is lower in alphabetical order than the second, the two entries are left as is and the routine advances to check the order of the second entry (new N) against the third (new N+1). If the first entry is greater than the

second, the routine will swap the two entries so that the entry that was initially the second entry would now be the first.

As the procedure continues, if the Nth entry is found to be greater than the N+1 entry, the two entries are exchanged in the table. Then, rather than advancing to the next entry, the routine backs up to compare the N-1 entry against the new N entry. This is because if the initial N+1 entry was lower than the N entry, it may also be lower than the N-1 entry. Therefore, the routine will continue to transfer the lower entry down through the table until the entry before it is lower in alphabetical order, or the entry is moved to the beginning of the table. The routine then starts back up through the table once again. This type of movement up and down through the table gives a "ripple" effect as the routine compares and shifts the entries around.

The operation of the sort routine may be illustrated by examining its procedure for arranging the sample list of names, given at the start of this section, into alphabetical order. The routine initially compares the first entries and finds the order to be correct, since the B in BROWN comes before the D in DAMATO. When the next pair of entries is compared, however, it is found that ANDERSON should be before DAMATO. The routine will therefore shift the second and third entries around, as illustrated in the table below.

BROWN, L. R.
ANDERSON, B.
DAMATO, P. J.
DARBY, P.
MATTOX, R. T.
MATTHEWS, K. D.
JONES, A. M.

Now that the second and third entries are in the proper order with respect to each other, the routine backs up to compare the first entry against the second. The second entry is now found to be less than the first (ANDERSON should come before BROWN), so the routine swaps these two entries, as shown next, and begins comparing the entries once again, starting with the first two entries.

ANDERSON, B.
BROWN, L. R.
DAMATO, P. J.

DARBY, P.
MATTOX, R. T.
MATTHEWS, K. D.
JONES, A. M.

On this pass through the table, the routine will proceed all the way up to MATTOX, R. T. before finding another entry out of order. Note that in comparing MATTOX, R. T. and MATTHEWS, K. D., the routine must work up to the fifth character in the last names to determine the proper order of the two names. If the last names were the same, it must go up to the initials to check whether the two entries are in order. Upon finding these two names in the wrong order, the routine will exchange them.

ANDERSON, B.
BROWN, L. R.
DAMATO, P. J.
DARBY, P.
MATTHEWS, K. D.
MATTOX, R. T.
JONES, A. M.

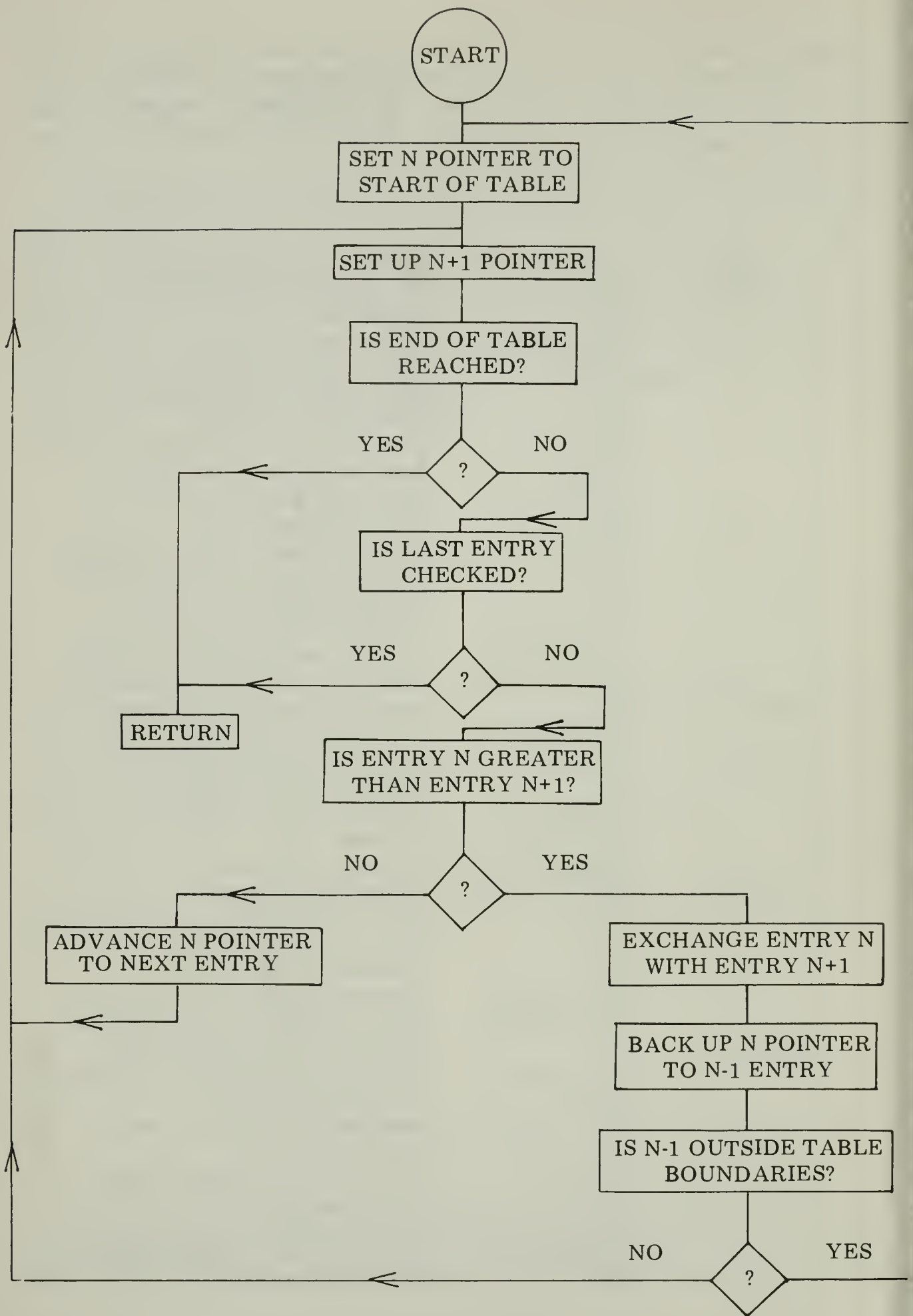
The routine then backs up in order to compare DARBY, P. and MATTHEWS, K. D. Finding these to be in order, it moves forward again until MATTOX, R. T. is compared to JONES, A. M. These two entries are swapped and the routine again backs up and compares MATTHEWS, K. D. to JONES, A. M. which it finds must also be swapped. Finally, after determining that DARBY, P. and JONES, A. M. are in the right order, the routine advances until the end of the table is reached. The resulting table will contain the names in the following order.

ANDERSON, B.
BROWN, L. R.
DAMATO, P. J.
DARBY, P.
JONES, A. M.
MATTHEWS, K. D.
MATTOX, R. T.

A routine that performs this type of ripple sorting is listed next, along with a flow chart of its operation. In checking for the start and end of the allowable table area, this routine assumes that the table

begins at page 04 location 000, and ends at page 07 location 377. If the entire table is not filled, the last entry must be followed by a zero byte, as discussed previously. The instruction sequence used here to compare the two entries is similar to the CPRMEM routine presented in chapter three.

```
SORT,    LXD SRTTBL    Set pointer to start of table
INITBK,  LLE           Move pointer from D and E
         LHD           To H and L
         LXB 020 000   Add entry length count to
         DADB          H and L for N+1 pointer
         LAH           Move page to accumulator to
         CPI 010       Check for end of table
         RTZ           If end, return, sort operation complete
         LBI 020       Set up entry length counter
         LAM           Fetch 1st character in N+1 entry
         NDA           If character is zero byte,
         RTZ           Return, sort operation is complete
         LDAD          Otherwise, fetch char from N entry
         CPM           Compare to character in N+1
         JTZ CKNEXT    Characters match, check next char
         JFC XCHANG    N+1< N, exchange entry contents
         JMP FINEND    N+1> N, order O.K. try next entry
CKNEXT,  DCB           Decrement entry length counter
         JFZ NOTFIN    If not done continue comparing
BACKER,  XCHG          Move N+1 pointer to D and E
         LAE           Adjust N+1 pointer back to
         NDI 360       The start address of the
         LEA           N+1 block
         JMP INITBK    Compare next entry
NOTFIN,  INXH          Advance N+1 pointer
         INXD          Advance N pointer
         LDAD          Fetch character from N
         CPM           Compare character in N+1
         JTZ CKNEXT    Character matched, check next char
         JFC XCHANG    N+1< N, exchange entry contents
FINEND,  DCB           Decrement entry length counter
         JTZ BACKER    End of entry, change N+1 pointer to N
         INXH          Advance N+1 pointer
         JMP FINEND    And continue to look for end
XCHANG,  LAL           Fetch N+1 pointer
         NDI 360       Restore pointer to start of entry
         LLA           For exchanging with N entry
         LAE           Fetch N pointer
         NDI 360       Restore pointer to start of entry
         LEA           For exchanging with N+1 entry
```


START
SET N POINTER TO START OF TABLE
SET UP N+1 POINTER
IS END OF TABLE REACHED?
YES
?
NO
IS LAST ENTRY CHECKED?
YES
?
NO
RETURN
IS ENTRY N GREATER THAN ENTRY N+1?
NO
?
YES
ADVANCE N POINTER TO NEXT ENTRY
EXCHANGE ENTRY N WITH ENTRY N+1
BACK UP N POINTER TO N-1 ENTRY
IS N-1 OUTSIDE TABLE BOUNDARIES?
NO
?
YES

```
                LBI 020         Set entry length pointer for exchange
NOTYET,         LDAD            Fetch character from N
                LCM             Fetch character from N+1
                LMA             Store N+1 character in N
                LAC
                STAD            Store N character in N+1
                INXD            Advance N pointer
                INXH            Advance N+1 pointer
                DCB             Decrement entry length counter
                JFZ NOTYET      If not zero, continue exchange
                LXB 320 377     Set pointer to N-1 start address
                DADB            By adding -40
                LAH             Fetch current page
                CPI 003         Make sure still in storage area
                JTC SORT        Went back too far - go to start
                XCHG            Else, start next sort with
                JMP INITBK      Current N-1 entry
```

This method of sorting may be aided in a number of ways to increase the efficiency of its operation. For example, the name input routine could be revised to separate the table into several sections, one for names beginning with the letters A through J, another for K through R, and another for S through Z. As each name is entered, the first letter could be checked and the name stored in the proper section of the table.

Another possibility is to revise the ripple sequence in the following manner. When a name is found to be out of alphabetical order, the start address of the current N+1 entry could be saved. Then, after the entry is backed up to the proper location in the table, the sort may resume by recalling the saved N+1 address and using it as the N address of the next entry to compare. This would avoid the time consuming process of retracing the sort up through the section already known to be in the proper order.

The sort function could also be revised to limit itself to the contents of just one field in an entry. By setting the pointers and field length counter to a specific field within each entry, the sort operation could arrange the entries according to some classification, such as the zip code of an address, or a special code set up by the programmer to classify each entry.

The techniques and routines discussed in this chapter may be utilized to create rather sophisticated programs designed specifically to fill one's requirements. By combining these with other programming functions presented in this book, one may develop programs that give the computer the capability to perform various operations for a wide variety of applications.

APPENDIX A

8080 INSTRUCTION SET

This table presents the entire instruction set of the 8080 CPU. The first column contains the mnemonics used throughout this book for each instruction. The second column contains the equivalent Intel mnemonics. The machine code, in both octal and hexadecimal representation, is then given, followed by the number of machine cycles and instruction execution time for the instruction. The final column indicates the number of times the instruction accesses memory, either to read or write (as described in chapter three). In the last three columns for the conditional call and return instructions, the two values given in each column represent the value when the condition is met, followed by the value when the condition is not met.

MNEMONIC USED HERE	INTEL EQUIVALENT	MACHINE CODE OCT	HEX	CYCLE STATES	EXEC TIME	MEM ACC
LAA	MOV A,A	177	7F	5	2.5	1
LAB	MOV A,B	170	78			
LAC	MOV A,C	171	79			
LAD	MOV A,D	172	7A			
LAE	MOV A,E	173	7B			
LAH	MOV A,H	174	7C			
LAL	MOV A,L	175	7D			
LBA	MOV B,A	107	47			
LBB	MOV B,B	100	40			
LBC	MOV B,C	101	41			
LBD	MOV B,D	102	42			
LBE	MOV B,E	103	43			
LBH	MOV B,H	104	44			
LBL	MOV B,L	105	45			
LCA	MOV C,A	117	4F			
LCB	MOV C,B	110	48			
LCC	MOV C,C	111	49			
LCD	MOV C,D	112	4A			
LCE	MOV C,E	113	4B			
LCH	MOV C,H	114	4C			
LCL	MOV C,L	115	4D			
LDA	MOV D,A	127	57			
LDB	MOV D,B	120	50	5	2.5	1

LDC	MOV D,C	121	51	5	2.5	1
LDD	MOV D,D	122	52			
LDE	MOV D,E	123	53			
LDH	MOV D,H	124	54			
LDL	MOV D,L	125	55			
LEA	MOV E,A	137	5F			
LEB	MOV E,B	130	58			
LEC	MOV E,C	131	59			
LED	MOV E,D	132	5A			
LEE	MOV E,E	133	5B			
LEH	MOV E,H	134	5C			
LEL	MOV E,L	135	5D			
LHA	MOV H,A	147	67			
LHB	MOV H,B	140	60			
LHC	MOV H,C	141	61			
LHD	MOV H,D	142	62			
LHE	MOV H,E	143	63			
LHH	MOV H,H	144	64			
LHL	MOV H,L	145	65			
LLA	MOV L,A	157	6F			
LLB	MOV L,B	150	68			
LLC	MOV L,C	151	69			
LLD	MOV L,D	152	6A			
LLE	MOV L,E	153	6B			
LLH	MOV L,H	154	6C			
LLL	MOV L,L	155	6D	5	2.5	1
LMA	MOV M,A	167	77	7	3.5	2
LMB	MOV M,B	160	70			
LMC	MOV M,C	161	71			
LMD	MOV M,D	162	72			
LME	MOV M,E	163	73			
LMH	MOV M,H	164	74			
LML	MOV M,L	165	75			
LAM	MOV A,M	176	7E			
LBM	MOV B,M	106	46			
LCM	MOV C,M	116	4E			
LDM	MOV D,M	126	56			
LEM	MOV E,M	136	5E			
LHM	MOV H,M	146	66			
LLM	MOV L,M	156	6E	7	3.5	2
LAI DDD	MVI A,DDD	076	3E	7	3.5	2
LBI DDD	MVI B,DDD	006	06			
LCI DDD	MVI C,DDD	016	0E			
LDI DDD	MVI D,DDD	026	16			
LEI DDD	MVI E,DDD	036	1E			
LHI DDD	MVI H,DDD	046	26			
LLI DDD	MVI L,DDD	056	2E	7	3.5	2
LMI DDD	MVI M,DDD	066	36	10	5.0	3

INA	INR A	074	3C	5	2.5	1
INB	INR B	004	04			
INC	INR C	014	0C			
IND	INR D	024	14			
INE	INR E	034	1C			
INH	INR H	044	24			
INL	INR L	054	2C	5	2.5	1
INM	INR M	064	34	10	5.0	3
DCA	DCR A	075	3D	5	2.5	1
DCB	DCR B	005	05			
DCC	DCR C	015	0D			
DCD	DCR D	025	15			
DCE	DCR E	035	1D			
DCH	DCR H	045	25			
DCL	DCR L	055	2D	5	2.5	1
DCM	DCR M	065	35	10	5.0	3
ADA	ADD A	207	87	4	2.0	1
ADB	ADD B	200	80			
ADC	ADD C	201	81			
ADD	ADD D	202	82			
ADE	ADD E	203	83			
ADH	ADD H	204	84			
ADL	ADD L	205	85	4	2.0	1
ADM	ADD M	206	86	7	3.5	2
ACA	ADC A	217	8F	4	2.0	1
ACB	ADC B	210	88			
ACC	ADC C	211	89			
ACD	ADC D	212	8A			
ACE	ADC E	213	8B			
ACH	ADC H	214	8C			
ACL	ADC L	215	8D	4	2.0	1
ACM	ADC M	216	8E	7	3.5	2
SUA	SUB A	227	97	4	2.0	1
SUB	SUB B	220	90			
SUC	SUB C	221	91			
SUD	SUB D	222	92			
SUE	SUB E	223	93			
SUH	SUB H	224	94			
SUL	SUB L	225	95	4	2.0	1
SUM	SUB M	226	96	7	3.5	2
SBA	SBB A	237	9F	4	2.0	1
SBB	SBB B	230	98			
SBC	SBB C	231	99			
SBD	SBB D	232	9A			
SBE	SBB E	233	9B			
SBH	SBB H	234	9C			
SBL	SBB L	235	9D	4	2.0	1
SBM	SBB M	236	9E	7	3.5	2

CPA	CMP A	277	BF	4	2.0	1
CPB	CMP B	270	B8			
CPC	CMP C	271	B9			
CPD	CMP D	272	BA			
CPE	CMP E	273	BB			
CPH	CMP H	274	BC			
CPL	CMP L	275	BD	4	2.0	1
CPM	CMP M	276	BE	7	3.5	2
ADI DDD	ADI DDD	306	C6	7	3.5	2
ACI DDD	ACI DDD	316	CE			
SUI DDD	SUI DDD	326	D6			
SBI DDD	SBI DDD	336	DE			
CPI DDD	CPI DDD	376	FE	7	3.5	2
NDA	ANA A	247	A7	4	2.0	1
NDB	ANA B	240	A0			
NDC	ANA C	241	A1			
NDD	ANA D	242	A2			
NDE	ANA E	243	A3			
NDH	ANA H	244	A4			
NDL	ANA L	245	A5	4	2.0	1
NDM	ANA M	246	A6	7	3.5	2
NDI DDD	ANI DDD	346	E6	7	3.5	2
ORA	ORA A	267	B7	4	2.0	1
ORB	ORA B	260	B0			
ORC	ORA C	261	B1			
ORD	ORA D	262	B2			
ORE	ORA E	263	B3			
ORH	ORA H	264	B4			
ORL	ORA L	265	B5	4	2.0	1
ORM	ORA M	266	B6	7	3.5	2
ORI DDD	ORI DDD	366	F6	7	3.5	2
XRA	XRA A	257	AF	4	2.0	1
XRB	XRA B	250	A8			
XRC	XRA C	251	A9			
XRD	XRA D	252	AA			
XRE	XRA E	253	AB			
XRH	XRA H	254	AC			
XRL	XRA L	255	AD	4	2.0	1
XRM	XRA M	256	AE	7	3.5	2
XRI DDD	XRI DDD	356	EE	7	3.5	2
INP DDD	IN DDD	333	DB	10	5.0	2
OUT DDD	OUT DDD	323	D3	10	5.0	2
HLT	HLT	166	76	7	3.5	1
NOP	NOP	000	00	4	2.0	1
DI	DI	363	F3	4	2.0	1
EI	EI	373	FB	4	2.0	1
RLC	RLC	007	07	4	2.0	1
RAL	RAL	027	17	4	2.0	1

RRC	RRC	017	0F	4	2.0	1
RAR	RAR	037	1F	4	2.0	1
LXB ADDR	LXI B,ADDR	001	01	10	5.0	3
LXD ADDR	LXI D,ADDR	021	11	↓	↓	↓
LXH ADDR	LXI H,ADDR	041	21	↓	↓	↓
LXS ADDR	LXI SP,ADDR	061	31	10	5.0	3
STA ADDR	STA ADDR	062	32	13	6.5	4
LTA ADDR	LDA ADDR	072	3A	13	6.5	4
CMA	CMA	057	2F	4	2.0	1
DAA	DAA	047	27	4	2.0	1
STC	STC	067	37	4	2.0	1
CMC	CMC	077	3F	4	2.0	1
POPB	POP B	301	C1	10	5.0	3
POPD	POP D	321	D1	↓	↓	↓
POPH	POP H	341	E1	↓	↓	↓
POPS	POP SP	361	F1	10	5.0	3
PUSB	PUSH B	305	C5	11	5.5	3
PUSD	PUSH D	325	D5	↓	↓	↓
PUSH	PUSH H	345	E5	↓	↓	↓
PUSS	PUSH SP	365	F5	11	5.5	3
XCHG	XCHG	353	EB	4	2.0	1
XTHL	XTHL	343	E3	18	9.0	5
SPHL	SPHL	371	F9	5	2.5	1
DADB	DAD B	011	09	10	5.0	1
DADD	DAD D	031	19	↓	↓	↓
DADH	DAD H	051	29	↓	↓	↓
DADS	DAD SP	071	39	10	5.0	1
STAB	STAX B	002	02	7	3.5	2
STAD	STAX D	022	12	↓	↓	↓
LDAB	LDAX B	012	0A	↓	↓	↓
LDAD	LDAX D	032	1A	7	3.5	2
INXB	INX B	003	03	5	2.5	1
INXD	INX D	023	13	↓	↓	↓
INXH	INX H	043	23	↓	↓	↓
INXS	INX SP	063	33	↓	↓	↓
DCXB	DCX B	013	0B	↓	↓	↓
DCXD	DCX D	033	1B	↓	↓	↓
DCXH	DCX H	053	2B	↓	↓	↓
DCXS	DCX SP	073	3B	5	2.5	1
SHLD ADDR	SHLD ADDR	042	22	16	8.0	5
LHLD ADDR	LHLD ADDR	052	2A	16	8.0	5
JMP ADDR	JMP ADDR	303	C3	10	5.0	3
JTC ADDR	JC ADDR	332	DA	↓	↓	↓
JTZ ADDR	JZ ADDR	312	CA	↓	↓	↓
JTS ADDR	JM ADDR	372	FA	↓	↓	↓
JTP ADDR	JPE ADDR	352	EA	↓	↓	↓
JFC ADDR	JNC ADDR	322	D2	↓	↓	↓
JFZ ADDR	JNZ ADDR	302	C2	10	5.0	3

JFS ADDR	JP ADDR	362	F2	10	5.0	3
JFP ADDR	JPO ADDR	342	E2	10	5.0	3
PCHL	PCHL	351	E9	5	2.5	1
CAL ADDR	CALL ADDR	315	CD	17	8.5	5
CTC ADDR	CC ADDR	334	DC	17/11	8.5/5.5	5/3
CTZ ADDR	CZ ADDR	314	CC			
CTS ADDR	CM ADDR	374	FC			
CTP ADDR	CPE ADDR	354	EC			
CFC ADDR	CNC ADDR	324	D4			
CFZ ADDR	CNZ ADDR	304	C4			
CFS ADDR	CP ADDR	364	F4			
CFP ADDR	CPO ADDR	344	E4	17/11	8.5/5.5	5/3
RET	RET	311	C9	10	5.0	3
RTC	RC	330	D8	11/5	5.5/2.5	3/1
RTZ	RZ	310	C8			
RTS	RM	370	F8			
RTP	RPE	350	E8			
RFC	RNC	320	D0			
RFZ	RNZ	300	C0			
RFS	RP	360	F0			
RFP	RPO	340	E0	11/5	5.5/2.5	3/1
RST 0	RST 0	307	C7	11	5.5	3
RST 1	RST 1	317	CF			
RST 2	RST 2	327	D7			
RST 3	RST 3	337	DF			
RST 4	RST 4	347	E7			
RST 5	RST 5	357	EF			
RST 6	RST 6	367	F7			
RST 7	RST 7	377	FF	11	5.5	3

APPENDIX B

OCTAL TO HEXADECIMAL

	0	1	2	3	4	5	6	7
00	0	1	2	3	4	5	6	7
01	8	9	A	B	C	D	E	F
02	10	11	12	13	14	15	16	17
03	18	19	1A	1B	1C	1D	1E	1F
04	20	21	22	23	24	25	26	27
05	28	29	2A	2B	2C	2D	2E	2F
06	30	31	32	33	34	35	36	37
07	38	39	3A	3B	3C	3D	3E	3F
10	40	41	42	43	44	45	46	47
11	48	49	4A	4B	4C	4D	4E	4F
12	50	51	52	53	54	55	56	57
13	58	59	5A	5B	5C	5D	5E	5F
14	60	61	62	63	64	65	66	67
15	68	69	6A	6B	6C	6D	6E	6F
16	70	71	72	73	74	75	76	77
17	78	79	7A	7B	7C	7D	7E	7F
20	80	81	82	83	84	85	86	87
21	88	89	8A	8B	8C	8D	8E	8F
22	90	91	92	93	94	95	96	97
23	98	99	9A	9B	9C	9D	9E	9F
24	A0	A1	A2	A3	A4	A5	A6	A7
25	A8	A9	AA	AB	AC	AD	AE	AF
26	B0	B1	B2	B3	B4	B5	B6	B7
27	B8	B9	BA	BB	BC	BD	BE	BF
30	C0	C1	C2	C3	C4	C5	C6	C7
31	C8	C9	CA	CB	CC	CD	CE	CF
32	D0	D1	D2	D3	D4	D5	D6	D7
33	D8	D9	DA	DB	DC	DD	DE	DF
34	E0	E1	E2	E3	E4	E5	E6	E7
35	E8	E9	EA	EB	EC	ED	EE	EF
36	F0	F1	F2	F3	F4	F5	F6	F7
37	F8	F9	FA	FB	FC	FD	FE	FF

APPENDIX C

HEXADECIMAL TO DECIMAL

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
10	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
20	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047
30	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063
40	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
50	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095
60	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

APPENDIX D

ASCII CHARACTER SET

CHARACTERS SYMBOLIZED	OCTAL REP	HEXA REP	CHARACTERS SYMBOLIZED	OCTAL REP	HEXA REP
A	301	C1	!	241	A1
B	302	C2	”	242	A2
C	303	C3	#	243	A3
D	304	C4	$	244	A4
E	305	C5	%	245	A5
F	306	C6	&	246	A6
G	307	C7	’	247	A7
H	310	C8	(	250	A8
I	311	C9	)	251	A9
J	312	CA	*	252	AA
K	313	CB	+	253	AB
L	314	CC	,	254	AC
M	315	CD	-	255	AD
N	316	CE	.	256	AE
O	317	CF	/	257	AF
P	320	D0	0	260	B0
Q	321	D1	1	261	B1
R	322	D2	2	262	B2
S	323	D3	3	263	B3
T	324	D4	4	264	B4
U	325	D5	5	265	B5
V	326	D6	6	266	B6
W	327	D7	7	267	B7
X	330	D8	8	270	B8
Y	331	D9	9	271	B9
Z	332	DA	:	272	BA
[	333	DB	;	273	BB
\	334	DC	<	274	BC
]	335	DD	=	275	BD
↑	336	DE	>	276	BE
←	337	DF	?	277	BF
SPACE	240	A0	@	300	C0
CAR RET	215	8D	RUBOUT	377	FF
LINE FEED	212	8A			

APPENDIX E

BAUDOT CHARACTER SET

CHARACTERS LC	CHARACTERS UC	5 LEVEL CODE BIT POSITION 5 4 3 2 1	OCTAL CODES LC	OCTAL CODES UC
A	-	0 0 0 1 1	003	043
B	?	1 1 0 0 1	031	071
C	:	0 1 1 1 0	016	056
D	$	0 1 0 0 1	011	051
E	3	0 0 0 0 1	001	041
F	!	0 1 1 0 1	015	055
G	&	1 1 0 1 0	032	072
H	#	1 0 1 0 0	024	064
I	8	0 0 1 1 0	006	046
J	’	0 1 0 1 1	013	053
K	(	0 1 1 1 1	017	057
L	)	1 0 0 1 0	022	062
M	.	1 1 1 0 0	034	074
N	,	0 1 1 0 0	014	054
O	9	1 1 0 0 0	030	070
P	0	1 0 1 1 0	026	066
Q	1	1 0 1 1 1	027	067
R	4	0 1 0 1 0	012	052
S	BELL	0 0 1 0 1	005	045
T	5	1 0 0 0 0	020	060
U	7	0 0 1 1 1	007	047
V	;	1 1 1 1 0	036	076
W	2	1 0 0 1 1	023	063
X	/	1 1 1 0 1	035	075
Y	6	1 0 1 0 1	025	065
Z	”	1 0 0 0 1	021	061
SPACE		0 0 1 0 0	004	004
CAR. RET.		0 1 0 0 0	010	010
LINE FEED		0 0 0 1 0	002	002
NULL		0 0 0 0 0	000	000
FIGURES		1 1 0 1 1	033	033
LETTERS		1 1 1 1 1	037	037

APPENDIX F

FLOATING POINT PROGRAM MEMORY DUMP

This is a memory dump of the floating point program as presented in chapter 6. The two columns on the left indicate the page and low address of the first memory location on that line, followed by the contents of the next eight sequential memory locations. The table that immediately follows this memory dump indicates the start and end addresses of the major routines as assembled here.

001 000	164 004 030 005 042 005 311 005
001 010	061 000 001 315 177 001 315 237
001 020	002 315 217 001 056 124 021 170
001 030	000 006 004 315 017 005 315 200
001 040	007 006 000 376 253 312 105 001
001 050	376 255 312 103 001 376 330 312
001 060	101 001 376 257 312 077 001 376
001 070	377 302 036 001 303 010 001 004
001 100	004 004 004 004 004 056 110 160
001 110	315 300 007 315 217 001 315 237
001 120	002 315 217 001 076 275 315 300
001 130	007 315 217 001 056 170 021 134
001 140	000 006 004 315 017 005 056 110
001 150	156 046 001 136 043 126 056 167
001 160	163 043 162 046 000 124 315 000
001 170	000 315 232 001 303 010 001 076
001 200	215 315 300 007 076 212 315 300
001 210	007 076 212 315 300 007 311 076
001 220	240 315 300 007 076 240 315 300
001 230	007 311 056 157 066 000 056 126
001 240	176 247 372 252 001 076 253 303
001 250	263 001 056 124 006 003 315 145
001 260	006 076 255 315 300 007 076 260
001 270	315 300 007 076 256 315 300 007
001 300	056 127 065 362 326 001 076 004
001 310	206 362 334 001 315 232 003 056
001 320	127 176 247 303 303 001 315 276
001 330	003 303 317 001 036 164 124 056
001 340	124 006 003 315 017 005 056 167
001 350	066 000 056 164 006 003 315 164
001 360	006 315 077 002 056 127 064 312

001 370	004 002 056 167 006 004 315 176
002 000	006 303 364 001 056 107 066 007
002 010	056 167 176 247 312 043 002 056
002 020	167 076 260 206 315 300 007 056
002 030	107 065 312 154 002 315 077 002
002 040	303 017 002 056 157 065 056 166
002 050	176 247 302 027 002 055 176 247
002 060	302 027 002 055 176 247 302 027
002 070	002 056 157 167 303 027 002 056
002 100	167 066 000 056 164 021 160 000
002 110	006 004 315 017 005 056 164 006
002 120	004 315 164 006 056 164 006 004
002 130	315 164 006 056 164 036 160 006
002 140	004 315 132 006 056 164 006 004
002 150	315 164 006 311 076 305 315 300
002 160	007 056 157 176 247 372 175 002
002 170	076 253 303 202 002 057 074 167
002 200	076 255 315 300 007 006 000 176
002 210	326 012 372 222 002 167 004 303
002 220	210 002 076 260 200 315 300 007
002 230	176 306 260 315 300 007 311 041
002 240	150 000 006 010 315 223 006 056
002 250	103 006 004 315 223 006 315 200
002 260	007 376 253 312 276 002 376 255
002 270	302 304 002 056 103 167 315 300
002 300	007 315 200 007 376 377 312 007
002 310	003 376 256 312 363 002 376 305
002 320	312 022 003 376 260 372 124 003
002 330	376 272 362 124 003 056 156 107
002 340	076 370 246 302 301 002 170 315
002 350	300 007 056 105 064 315 334 003
002 360	303 301 002 107 056 106 176 247
002 370	302 124 003 056 105 167 054 160
003 000	170 315 300 007 303 301 002 076
003 010	274 315 300 007 315 217 001 303
003 020	237 002 315 300 007 315 200 007
003 030	376 253 312 045 003 376 255 302
003 040	053 003 056 104 167 315 300 007
003 050	315 200 007 376 377 312 007 003
003 060	376 260 372 124 003 376 272 362
003 070	124 003 346 017 107 056 157 076
003 100	003 276 372 050 003 116 176 247
003 110	027 027 201 027 200 167 076 260
003 120	200 303 045 003 056 103 176 247
003 130	312 142 003 056 154 006 003 315
003 140	145 006 056 153 257 127 167 036
003 150	123 006 004 315 017 005 006 027
003 160	315 036 004 056 104 176 247 056

003 170	157 312 200 003 176 057 074 167
003 200	056 106 176 247 312 213 003 056
003 210	105 257 226 056 157 206 167 372
003 220	267 003 310 315 232 003 302 223
003 230	003 311 036 134 124 056 124 006
003 240	004 315 017 005 056 127 066 004
003 250	055 066 120 055 257 167 055 167
003 260	315 042 005 056 157 065 311 315
003 270	276 003 302 267 003 311 036 134
003 300	124 056 124 006 004 315 017 005
003 310	056 127 066 375 055 066 146 055
003 320	066 146 055 066 147 315 042 005
003 330	056 157 064 311 056 153 170 346
003 340	017 167 021 150 000 056 154 006
003 350	003 315 017 005 056 154 006 003
003 360	315 164 006 056 154 006 003 315
003 370	164 006 036 150 056 154 006 003
004 000	315 132 006 056 154 006 003 315
004 010	164 006 056 152 257 167 055 167
004 020	056 153 176 036 150 022 056 154
004 030	006 003 315 132 006 311 170 247
004 040	312 046 004 056 127 160 056 126
004 050	176 056 100 247 372 064 004 257
004 060	167 303 074 004 167 006 004 056
004 070	123 315 145 006 056 126 006 004
004 100	176 247 302 117 004 055 005 302
004 110	100 004 056 127 257 167 311 056
004 120	123 006 004 315 164 006 176 247
004 130	372 140 004 054 065 303 117 004
004 140	056 126 006 003 315 176 006 056
004 150	100 176 247 360 056 124 006 003
004 160	315 145 006 311 056 126 006 003
004 170	176 247 302 217 004 005 312 205
004 200	004 055 303 170 004 353 142 056
004 210	134 006 004 315 017 005 311 056
004 220	136 006 003 176 247 302 236 004
004 230	005 310 055 303 223 004 056 127
004 240	176 056 137 276 312 334 004 057
004 250	074 206 362 257 004 057 074 376
004 260	030 372 276 004 176 056 127 226
004 270	370 056 124 303 205 004 176 056
004 300	127 226 372 322 004 117 056 127
004 310	315 375 004 015 302 306 004 303
004 320	334 004 117 056 137 315 375 004
004 330	014 302 323 004 056 123 066 000
004 340	056 133 066 000 056 127 315 375
004 350	004 056 137 315 375 004 124 036
004 360	123 353 006 004 315 132 006 006

004 370	000 315 036 004 311 064 055 006
005 000	004 176 247 372 012 005 315 176
005 010	006 311 027 315 177 006 311 176
005 020	022 043 023 005 310 303 017 005
005 030	056 124 006 003 315 145 006 303
005 040	164 004 315 164 005 056 137 176
005 050	056 127 206 074 167 056 102 066
005 060	027 056 126 006 003 315 176 006
005 070	334 256 005 056 146 006 006 315
005 100	176 006 056 102 065 302 061 005
005 110	056 146 006 006 315 176 006 056
005 120	143 176 027 247 374 271 005 056
005 130	123 353 142 056 143 006 004 315
005 140	017 005 006 000 315 036 004 056
005 150	101 176 247 300 056 124 006 003
005 160	315 145 006 311 315 237 005 056
005 170	101 066 001 056 126 176 247 372
005 200	222 005 056 136 176 247 360 056
005 210	101 065 056 134 006 003 315 145
005 220	006 311 056 101 065 056 124 006
005 230	003 315 145 006 303 202 005 056
005 240	140 006 010 315 223 006 006 004
005 250	056 130 315 223 006 311 056 141
005 260	124 036 131 006 006 315 132 006
005 270	311 006 003 076 100 206 167 054
005 300	076 000 216 005 302 276 005 167
005 310	311 315 164 005 056 126 076 000
005 320	276 302 336 005 055 276 302 336
005 330	005 055 276 312 124 006 056 137
005 340	176 056 127 226 074 167 056 102
005 350	066 027 315 075 006 372 375 005
005 360	036 134 056 131 006 003 315 017
005 370	005 067 303 377 005 067 077 056
006 000	144 006 003 315 165 006 056 134
006 010	006 003 315 164 006 056 102 065
006 020	302 352 005 315 075 006 372 064
006 030	006 056 144 176 306 001 167 076
006 040	000 054 216 167 076 000 054 216
006 050	167 362 064 006 006 003 315 176
006 060	006 056 127 064 056 143 036 123
006 070	006 004 303 137 005 056 131 353
006 100	142 056 124 006 003 315 017 005
006 110	036 134 056 131 006 003 315 210
006 120	006 176 247 311 315 100 007 303
006 130	160 007 247 032 216 167 005 310
006 140	043 023 303 133 006 176 057 306
006 150	001 167 005 310 043 176 057 316
006 160	000 303 151 006 247 176 027 167

```
006 170     005 310 043 303 165 006 247 176
006 200     037 167 005 310 053 303 177 006
006 210     247 032 236 167 005 310 043 023
006 220     303 211 006 066 000 005 310 043
006 230     303 223 006
```

		START ADDRESS	END ADDRESS
	LOOK UP TABLE	01 000	01 007
	FPCONT	01 010	01 231
	FPOUT	01 232	02 236
	FPINP	02 237	04 035
	FPNORM	04 036	04 163
	FPADD	04 164	05 027
	FPSUB	05 030	05 041
	FPMULT	05 042	05 310
	FPDIV	05 311	06 131
	ADDER	06 132	06 144
	COMPLM	06 145	06 163
	ROTATL	06 164	06 175
	ROTATR	06 176	06 207
	SUBBER	06 210	06 222
	CLRMEM	06 223	06 232
*	DERMSG	07 100	
*	USERDF	07 160	
*	INPUT	07 200	
*	ECHO	07 300	

* USER PROVIDED ROUTINES