ZENTEC

ZAM

ZENTEC ASSEMBLY METHOD

9000 SERIES REFERENCE MANUAL

Zentec Corporation

88-409-01 REV B.

ZAM

ZENTEC ASSEMBLY METHOD

9000 SERIES REFERENCE MANUAL

The information in this manual is based on the latest specifications available at the time of publication. Every effort has been made to insure its accuracy. However, ZENTEC reserves the right to make changes at any time.

Issued: May 1976

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

SECTION TITLE PAGE

TABLE OF CONTENTS (Continued)

SECTION TITLE PAGE

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

The Zentec 9000 Series Microcomputer Terminal System is an 8-bit microcomputer. It offers a powerful instruction set, including extensive memory referencing and flexible branch-on-condition capability.

There are two models in the Zentec 9000 series: the 9002 and the 9003. The 9003 is a somewhat more powerful version of the 9002. The 9003 can directly address 64K bytes of memory; the 9002 can directly address 16K bytes. The 9003 also includes fully-programmable stacks, allowing unlimited subroutine nesting and full interrupt handling capability.

This manual has been written to help the reader program the Zentec 9000 series. Most of this manual applies equally to the 9003 and the 9002. Differences between the features of the two models are discussed where appropriate. All descriptions within that are unique to the 9003 are printed in *italics*. All programming examples are for the 9003 unless otherwise noted.

All memory addresses used in this manual are hexadecimal, and are denoted in the form X'nnnn'.

SECTION 1

ORGANIZATION OF THE ZENTEC 9000 SERIES

This section provides the programmer with a functional overview of the 9000 series Microcomputer Terminal System. Information is presented at a level that provides a programmer with necessary background in order to write efficient programs.

The programmer can think of the computer as consisting of the following parts:

1. Seven working registers in which all data operations occur, and which provide one means for addressing memory.

2. Memory, which may hold program instructions or data and which must be addressed location by location in order to access stored information.

3. The Program Counter, whose contents indicate the next program instruction to be executed.

4. The Stack Pointer, a register which enables execution of subroutines.

5. Input/Ouptut, which is the interface between a program and external devices.

1.1 WORKING REGISTERS

The 9000 series provides the programmer with an 8-bit accumulator and six additional 8-bit "scratch pad" registers. These seven "working registers" are, by convention, identified by the letters A (Accumulator), B, C, D, E, H, and L.

1.2 MEMORY

The Zentec 9003 contains up to 64K bytes of memory. The Zentec 9002 contains up to 16K bytes of memory. This memory is divided into ROM (Read Only Memory), PROM (Programmable Read Only Memory) and RAM (Random Access Memory) portions.

For addressing purposes the overall memory is divided into 2K (2048) 8-bit byte blocks, as shown in Figures 1-1 and 1-2. The usage of certain 2K memory blocks is preassigned at the factory, but all other blocks can be assigned by the user.

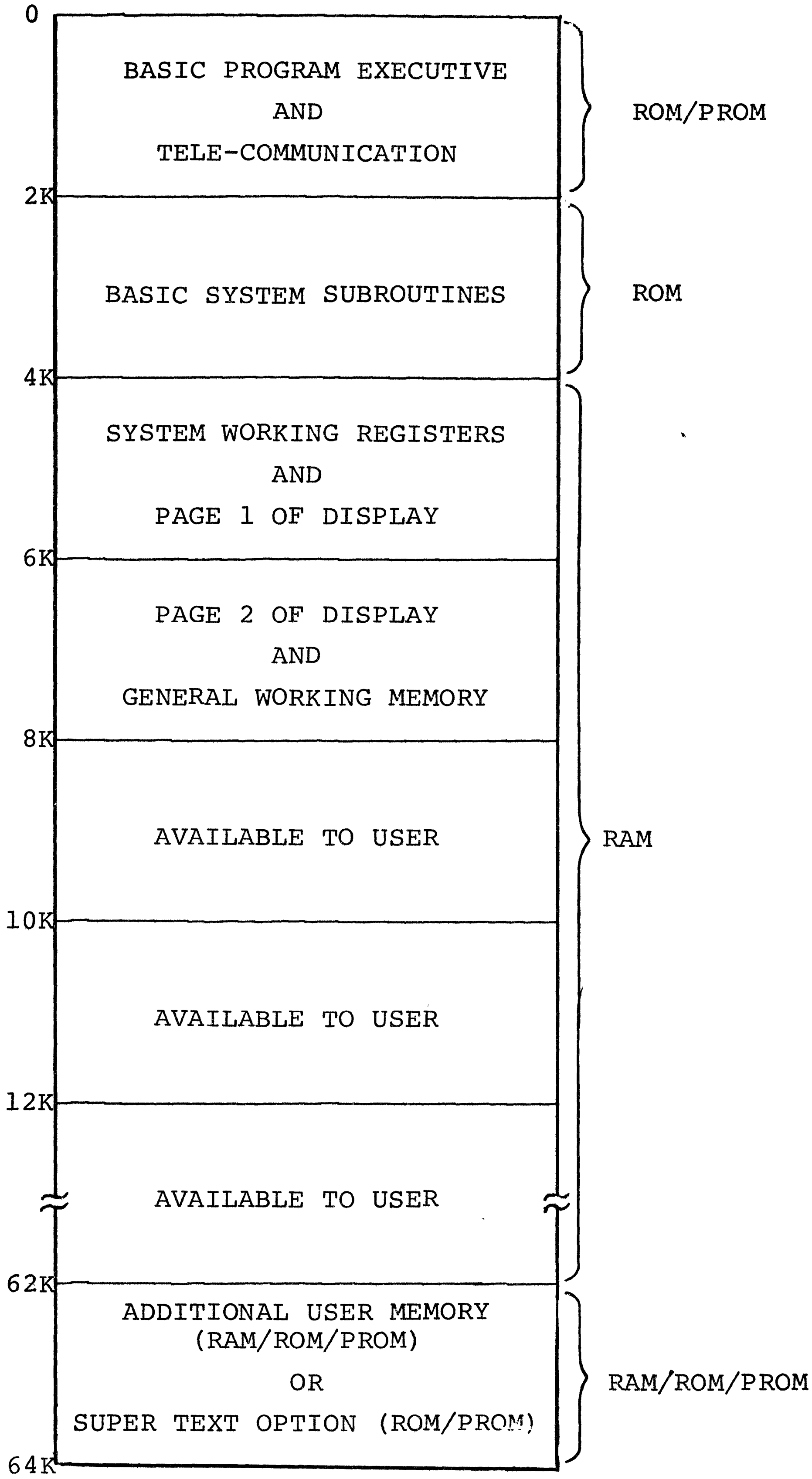

0
BASIC PROGRAM EXECUTIVE AND TELE-COMMUNICATION
ROM/PROM
2K
BASIC SYSTEM SUBROUTINES
ROM
4K
SYSTEM WORKING REGISTERS AND PAGE 1 OF DISPLAY
6K
PAGE 2 OF DISPLAY AND GENERAL WORKING MEMORY
8K
AVAILABLE TO USER
RAM
10K
AVAILABLE TO USER
12K
AVAILABLE TO USER
62K
ADDITIONAL USER MEMORY (RAM/ROM/PROM) OR SUPER TEXT OPTION (ROM/PROM)
RAM/ROM/PROM
64K

FIGURE 1-1

SYSTEM MEMORY ORGANIZATION OF THE 9003

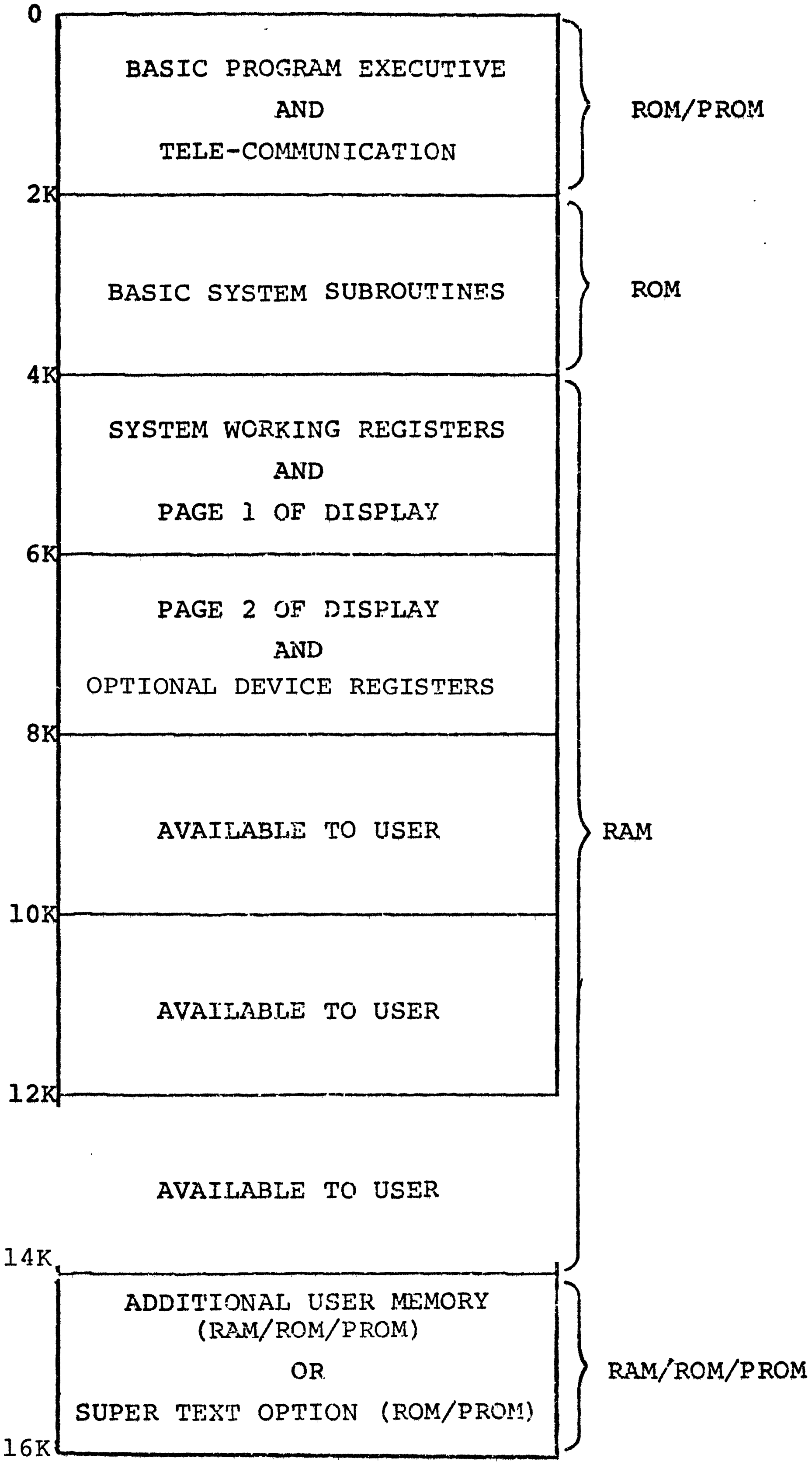

0
BASIC PROGRAM EXECUTIVE AND TELE-COMMUNICATION
ROM/PROM
2K
BASIC SYSTEM SUBROUTINES
ROM
4K
SYSTEM WORKING REGISTERS AND PAGE 1 OF DISPLAY
6K
PAGE 2 OF DISPLAY AND OPTIONAL DEVICE REGISTERS
8K
AVAILABLE TO USER
RAM
10K
AVAILABLE TO USER
12K
AVAILABLE TO USER
14K
ADDITIONAL USER MEMORY (RAM/ROM/PROM) OR SUPER TEXT OPTION (ROM/PROM)
RAM/ROM/PROM
16K

FIGURE 1-2
SYSTEM MEMORY ORGANIZATION OF THE 9002

The block assignments are as follows:

- The first 2K-byte block is committed to ROM or PROM. Locations X'0000'-X'01FF' hold the Basic Program Executive. Locations X'0200'-X'07FF' hold the Tele-Communications package.

- The second block is committed to ROM. These locations (X'0800'-X'0FFF') hold the Basic System Subroutines, designed for both system and user use. These subroutines are described in Appendix B.

- The third block (X'1000'-X'17FF') is a RAM area that contains the system working registers (described in Section 8), system work space and Terminal display Page 1.

- The fourth block (X'1800'-X'1FFF') is a RAM area that contains Terminal display Page 2.

- 9003: Blocks 5 through 32 (X'2000'-X'FFFF') are RAM areas that the programmer may use. The 32nd block (X'F800'-X'FFFF') will contain the Super Text ROM/PROM if that option is installed.

- 9002: Blocks 5 through 8 (X'2000'-X'3FFF') are RAM areas that the programmer may use. The 8th block (X'3800'-X3FFF') will contain the Super Text ROM if that option is installed.

1.3 PROGRAM COUNTER

9003: The Program Counter is a 16-bit register which is accessible to the programmer and whose contents indicate the memory address of the next instruction to be executed.

9002: The Program Counter is a 14-bit register whose contents indicate the memory address of the next instruction to be executed.

1.4 STACK POINTER

9003: A stack is an area of memory set aside by the programmer in which data or addresses are stored and retrieved by stack operations. Stack operations are performed by several of the 9003 instructions, and facilitate execution of subroutines and handling of program interrupts. The programmer specifies which addresses the stack operations will operate upon via a special accessible 16-bit register called the stack pointer.

9002: Seven 14-bit registers provide storage for 7 levels of CALL. The stack automatically stores and restores the program counter upon the execution of a CALL and RETURN.

1.5 INPUT/OUTPUT

9003: The outside world consists of up to 64 input devices and 48 output devices. Each device communicates with the 9003 via data bytes sent to or received from the Accumulator, and each device is assigned a number which is not under control of the programmer. The instructions which perform these data transmissions are described in Section 5.14.

9002: The 9002 may access up to 32 input devices and 24 output devices.

SECTION 2

PROGRAMMING CONCEPTS

This section gives a basic introduction to Zentec 9000 series programming.

2.1 COMPUTER PROGRAM REPRESENTATION IN MEMORY

A computer program consists of a sequence of instructions. Each instruction enables an elementary operation such as the movement of a data byte, an arithmetic or logical operation on a data byte, or a change in instruction execution sequence. Instructions are described individually in Section 5.

A program will be stored in memory as a sequence of bits which represent the instructions of the program, and which we will represent via hexadecimal digits. The memory address of the next instruction to be executed is held in the Program Counter. Just before each instruction is executed, the Program Counter is advanced to the address of the next sequential instruction. Program execution proceeds sequentially unless a transfer-of-control instruction (branch, call, or return) is executed, which causes the Program Counter to be set to a specified address. Execution then continues sequentially from this new address in memory.

Upon examining the contents of a memory byte, there is no way of telling whether the byte contains an encoded instruction or data. It is up to the logic of a program to insure that data is not misinterpreted as an instruction code, but this is simply done as follows.

Every program has a starting memory address, which is the memory address of the byte holding the first instruction to be executed. Before the first instruction is executed, the Program Counter will automatically be advanced to address the next instruction to be executed, and this procedure will be repeated for every instruction in the program. 9003 instructions may require 1, 2, or 3 bytes to encode an instruction; in each case the Program Counter is automatically advanced to the start of the next instruction, as illustrated in Figure 2-1.

In order to avoid errors, the programmer must be sure that a data byte does not follow an instruction when another instruction is expected. Referring to Figure 2-1, an instruction is expected in byte X'201F', since instruction 8 is to be executed after instruction 7. If byte X'201F' held data, the program would not execute correctly. Therefore, when writing a program, do not store data between adjacent instructions that are to be executed consecutively.

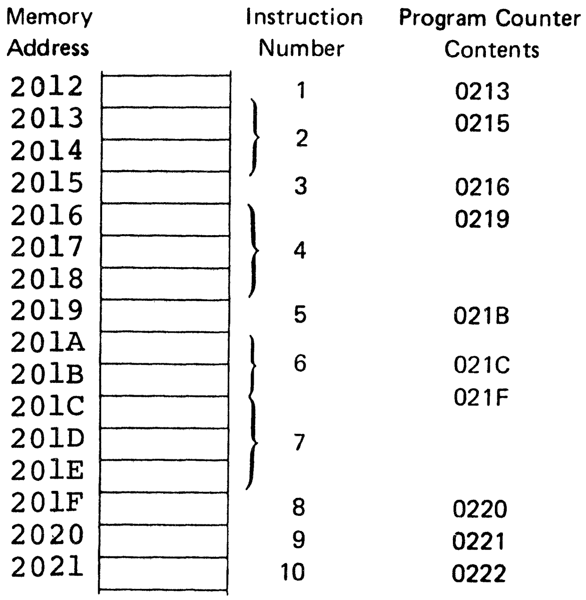


FIGURE 2-1

AUTOMATIC ADVANCE OF THE PROGRAM COUNTER AS INSTRUCTIONS ARE EXECUTED

A class of instructions (referred to as branch instructions) cause program execution to branch to an instruction that may be anywhere in memory. The memory address specified by the branch instruction must be the address of another instruction; if it is the address of a memory byte holding data, the program will not execute correctly. For example, referring to Figure 2-1, say instruction 4 specifies a branch to memory byte X'201F', and say instructions 5, 6, and 7 are replaced by data; then following execution of instruction 4, the program would execute correctly. But if, in error, instruction 4 specifies a branch to memory byte X'201E', an error would result, since this byte now holds data. Even if instructions 5, 6, and 7 were not replaced by data, a branch to memory byte X'201E' would cause an error, since this is not the first byte of the instruction.

Upon reading Section 5, you will see that it is easy to avoid writing an assembly language program with branch instructions that have erroneous memory addresses. Information on this subject is given rather to help the programmer who is debugging programs by entering hexadecimal codes directly into memory.

2.2 MEMORY ADDRESSING

By now it will have become apparent that addressing specific memory bytes constitutes an important part of any computer program.

Addresses are absolute or relocatable, depending upon the effect program relocation has on them. Program relocation is the loading of the object program into memory locations other than those originally assigned by the assembler. An address is absolute if its value does not change upon relocation. An address is relocatable if its value changes upon relocation.

Relocatability is resolved during loading. The subsections to follow show the ways the 9003 instructions can address memory when the program is executing.

2.2.1 Direct Addressing

With direct addressing, an instruction supplies an exact memory address.

The instruction

> *"Load the contents of memory address 1F2A into the Accumulator"*

is an example of an instruction using direct addressing, 1F2A being the direct address.

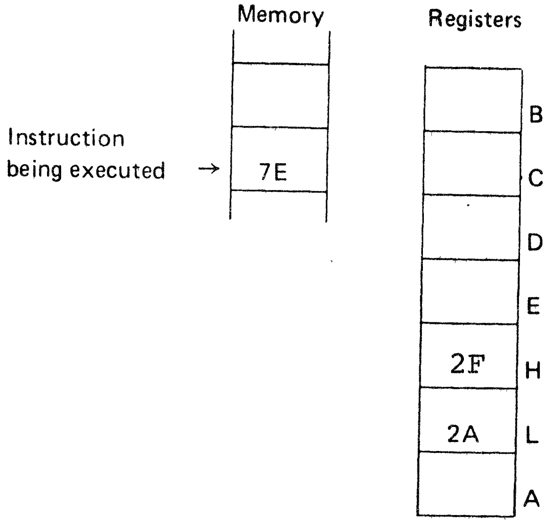
This would appear in memory as follows:

Memory Address	*Memory*	
any	*3A*	
any + 1	*2A*	*instruction being executed*
any + 2	*1F*	

The instruction occupies three memory bytes, the second and third of which hold the direct address.

2.2.2 Register Pair Addressing

A memory address may be specified by the contents of a register pair. For all 9002 and almost all 9003 memory reference instructions, the memory address is specified by the contents of the H and L registers. The H register contains the most significant 8 bits of the referenced address, and the L register contains the least significant 8 bits. A one-byte instruction which will load the Accumulator with the contents of memory byte X'2F2A' would appear as shown below.



In addition, there are two 9003 instructions which use either the B and C registers or the D and E registers to address memory. As above, the first register of the pair holds the most significant 8 bits of the address, while the second register holds the least significant 8 bits. These 9003 instructions, STA and LBA, are described in Section 5. For the 9002 a predefined macro is used to load a register pair. This macro, LHI, is described in Section 5.

2.2.3 Stack Pointer Addressing

Memory locations may be addressed via the 16-bit stack pointer register, as described below.

There are only two stack operations which may be performed; putting data into a stack is called a push, *while retrieving data from a stack is called a* pop.

STACK PUSH OPERATION

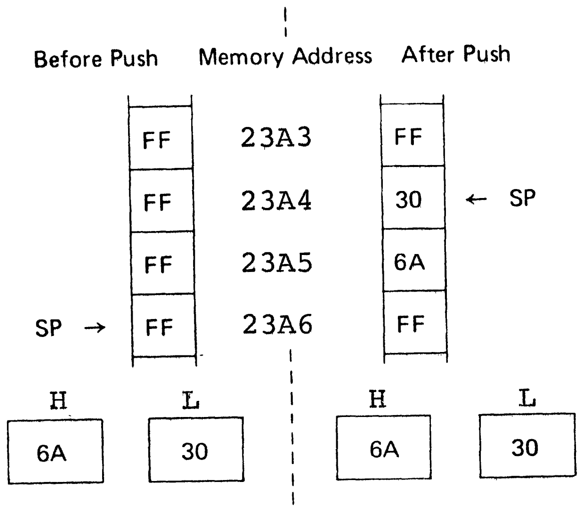
Sixteen bits of data are transferred to a memory area (called a stack) from a register pair or the 16-bit program counter during any stack push operation. The addresses of the memory area which is to be accessed during a stack push operation are determined by using the stack pointer as follows:

1. *The most significant 8 bits of data are stored at the memory address one less than the contents of the stack pointer.*

2. *The least significant 8 bits of data are stored at the memory address two less than the contents of the stack pointer.*

3. *The stack pointer is automatically decremented by two.*

For example, suppose that the stack pointer contains the address X'23A6', register H contains X'6A' and register L contains X'30'. Then a stack push of register pair H and L would operate as follows:



STACK POP OPERATION

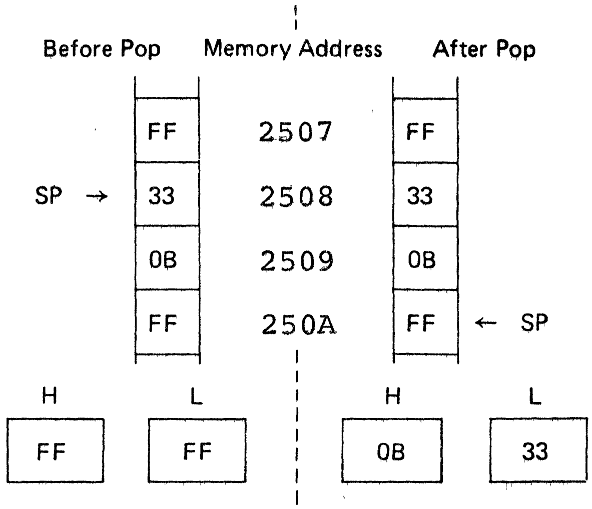
16 bits of data are transferred from a memory area (called a stack) to a register pair or the 16-bit program counter during any stack pop operation. The addresses of the memory area which is to be accessed during a stack pop operation are determined by using the stack pointer as follows:

1. *The second register of the pair, or the least significant 8 bits of the program counter, are loaded from the memory address held in the stack pointer.*

2. *The first register of the pair, or the most significant 8 bits of the program counter, are loaded from the memory address one greater than the address held in the stack pointer.*

3. *The stack pointer is automatically incremented by two.*

For example, suppose that the stack pointer contains the address X'2508', memory location X'2508' contains X'33' and memory location X'2509' contains X'0B'. Then a stack pop into register pair H and L would operate as follows:

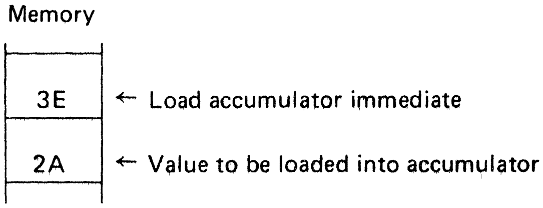


2.2.4 Immediate Addressing

An immediate instruction is one that contains data. The following is an example of immediate addressing:

"Load the accumulator with the value X'2A'."

The above instruction would be coded in memory as follows:

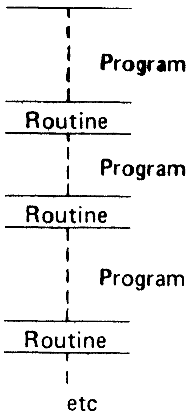

Memory
3E
← Load accumulator immediate
2A
← Value to be loaded into accumulator

Immediate instructions do not reference memory; rather they contain data in the memory byte following the instruction code byte.

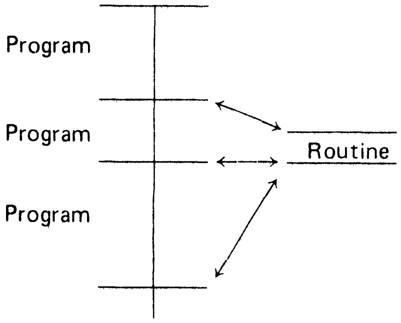
2.3 SUBROUTINES AND USE OF THE STACK FOR ADDRESSING

Before understanding the purpose or effectiveness of the stack, it is necessary to understand the concept of a subroutine.

Consider a frequently used operation such as multiplication. The 9000 series provides instructions to add one byte of data to another byte of data, but what if you wish to multiply these numbers? This will require a number of instructions to be executed in sequence. It is quite possible that this routine may be required many times within one program; to repeat the identical code every time it is needed is possible, but very wasteful of memory:

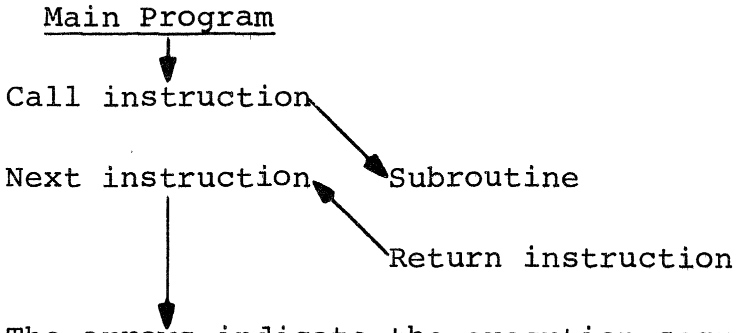


A more efficient means of accessing the routine would be to store it once, and find a way of accessing it when needed:



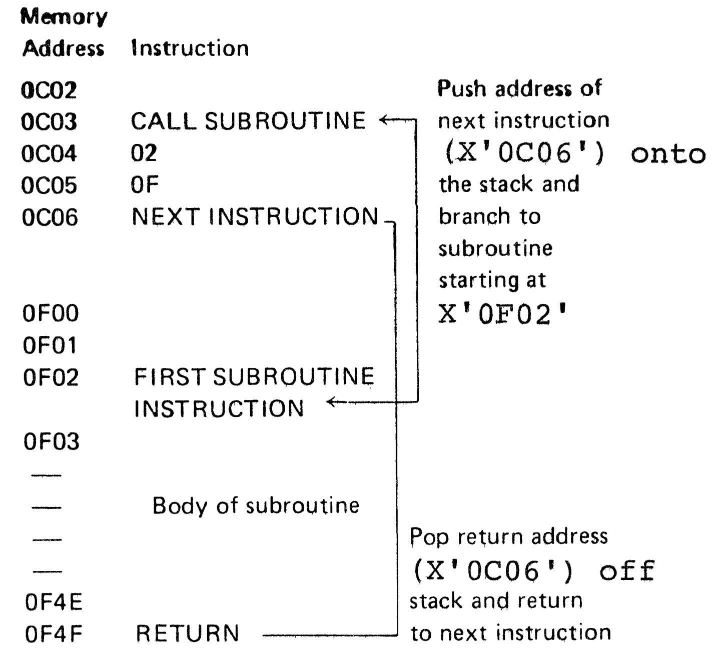
A frequently accessed routine such as the above is called a subroutine, and the 9000 series provides instructions that call and return from subroutines.

When a subroutine is executed, the sequence of events may be depicted as follows:



The arrows indicate the execution sequence.

When the "Call" instruction is executed, the address of the "next" instruction (that is, the address held in the Program Counter), is pushed onto the stack, and the subroutine is executed. The last executed instruction of a subroutine will usually be a "Return" instruction, which pops an address off the stack into the Program Counter, and thus causes program execution to continue at the "Next" instruction as illustrated below:



Memory Address	Instruction
0C02	
0C03	CALL SUBROUTINE
0C04	02
0C05	0F
0C06	NEXT INSTRUCTION
0F00	
0F01	
0F02	FIRST SUBROUTINE INSTRUCTION
0F03	
—	
—	Body of subroutine
—	
—	
0F4E	
0F4F	RETURN

Push address of next instruction (X'0C06') onto the stack and branch to subroutine starting at X'0F02'

Pop return address (X'0C06') off stack and return to next instruction

9003: Subroutines may be nested up to any depth, limited only by the amount of memory available for the stack. For example, the first subroutine could itself call some other subroutine, and so on. An examination of the sequence of stack pushes and pops will show that the return path will always be identical to the call path, even if the same subroutine is called at more than one level.

9002: Subroutines may be nested up to a depth of seven levels.

2.4 CONDITION BITS

Four condition (or status) bits are provided by the 9000 series to reflect the results of data operations. In addition, the 9003 also provides an Auxiliary Carry bit. All but one of these bits (the Auxiliary Carry bit) may be tested by program instructions which affect subsequent program execution. The descriptions of individual instructions in Section 5 specify which condition bits are affected by the execution of the instruction, and whether the execution of the instruction is dependent in any way on prior status of condition bits.

In the following discussion of condition bits, "setting" a bit causes its value to be 1, while "resetting" a bit causes its value to be 0.

2.4.1 Carry Bit

The Carry bit is set and reset by certain data operations, and its status can be directly tested by a program. The operations which affect the Carry bit are addition, subtraction, rotate, and logical operations. For example, addition of two one-byte numbers can produce a carry out of the high-order bit:

```
Bit No.  7  6  5  4  3  2  1  0
_______________________________
    AE=  1  0  1  0  1  1  1  0
  + 74= ┌0  1  1  1  0  1  0  0
   122  └►carry out=1, sets Carry bit=1
```

An additional operation that results in a carry out of the high-order bit will set the Carry bit; an addition operation that could have resulted in a carry out but did not will reset the Carry bit.

2.4.2 Auxiliary Carry Bit

The Auxiliary Carry bit indicates carry out of bit 3. It is used in decimal operations. The following addition will reset the Carry bit and set the Auxiliary Carry bit:

```
Bit No.  7  6  5  4  3  2  1  0
_______________________________
    2E=  0  0  1  0  1  1  1  0
  + 74=  0  1  1  1  0  1  0  0
     A2  1  0  1  0  0  0  1  0
         │           │
         └──►Carry=0 └──►Auxiliary Carry=1
```

The Auxiliary Carry bit will be affected by all addition, subtraction, increment, decrement, and compare instructions.

2.4.3 Sign Bit

It is possible to treat a byte of data as having the numerical range -128_{10} to $+127_{10}$. In this case, by convention, the 7 bit will always represent the sign of the number; that is, if the 7 bit is 1, the number is in the range -128_{10} to -1. If bit 7 is 0, the number is in the range 0 to $+127_{10}$.

At the conclusion of certain instructions (as specified in the instruction description sections of Section 5), the Sign bit will be set to the condition of the most significant bit of the answer (bit 7).

2.4.4 Zero Bit

This condition bit is set if the result generated by the execution of certain instructions is zero. The Zero bit is reset if the result is not zero.

A result that has a carry but a zero answer byte, as illustrated below, will also set the Zero bit:

```
Bit No.  7  6  5  4  3  2  1  0
         1  0  1  0  0  1  1  1
       + 0  1  0  1  1  0  0  1
     1]  0  0  0  0  0  0  0  0
Carry out    Zero answer
of bit 7.    Zero bit set to 1.
```

2.4.5 Parity Bit

Byte "parity" is checked after certain operations. The number of 1 bits in a byte are counted, and if the total is odd, "odd" parity is flagged; if the total is even, "even" parity is flagged.

The Parity bit is set to 1 for even parity, and is reset to 0 for odd parity.

SECTION 3

FORMAT OF THE ASSEMBLY LANGUAGE STATEMENT

Assembly language instructions must adhere to a fixed set of rules, as described in this section. An instruction has four separate and distinct parts of fields.

Field 1 is the LABEL field. It is a name used to reference the instruction's address.

Field 2 is the CODE field. It specifies the operation that is to be performed.

Field 3 is the OPERAND field. It provides any address or data information needed by the CODE field.

Field 4 is the COMMENT field. It is present for the programmer's convenience and is ignored by the assembler. The programmer uses comment fields to describe the operation and thus make the program more readable.

The assembler uses free fields; that is, any number of blanks may separate fields.

Before describing each field in detail, here are some general examples:

Label	Code	Operand	Comment
HERE	LBI	RC,0	Load register C with zero
THERE	DC	X'3A'	Create a one-byte data constant
LOOP	AR	RA,RE	Add register E to register A
	ROL	RA	Rotate register A to left

NOTE

These examples and the ones which follow are intended to illustrate how the various fields appear in complete assembly language statements. It is not necessary at this point to understand the operations that the statements perform.

3.1 LABEL FIELD

This is an optional field, which, if present, may be from 1 to 6 characters long. The first character of the label must be a letter of the alphabet.

The register names (RA, RB, RC, RD, RE, RH, and RL) are specially defined within the assembler and may not be used as labels. In addition, the names of Zentec's Basic System Subroutines should not be used in the label field. These subroutines are described in Appendix B; their names are summarized in Table 3-1.

Here are some examples of valid label fields:

```
LABELS
F14F
Q
```

Here are some invalid label fields:

```
123  begins with a decimal number
ADD2 is one of the Basic System Subroutines
RA   is a register name
```

The label INSTRUCTION has more than six characters; only the first six will be recognized. That is, the assembler will read this label as INSTRU.

Since labels serve as instruction addresses, they cannot be duplicated. For example, the sequence:

```
HERE      B         THERE
          ---
          ---
THERE     LR        RC,RD
          ---
          ---
THERE     CALL      SUB
```

is ambiguous; the assembler cannot determine which address is to be referenced by the B (Branch) instruction.

One instruction may have more than one label, however. The following sequence is valid:

```
LOOP1     EQU       *         First label
LOOP2     LR        RC,RD     Second label
          ---
          B         LOOP1
          ---
          B         LOOP2
```

Each B instruction will cause program control to be transferred to the same LR instruction.

TABLE 3-1

ZENTEC SYSTEM SUBROUTINE NAMES

NAME	SECTION
ABTAB	B.1.1
ADD2	B.3.1
ATAB	B.1.2
BLANK	B.2.1
BTAB	B.1.3
CDOWN	B.1.4
CLEAR	B.2.2
CLEFT	B.1.5
CMESSA	B.2.3
COM1	B.3.2
COMPER	B.3.3
CONV	B.3.4
CRIGHT	B.1.6
CUP	B.1.7
DELBYT	B.2.4
DELFD	B.2.5
DLINE	B.2.6
DPAGE	B.4.1
DREAD	7.2
DSCROL	B.4.2
DWRITE	7.2
EOL	B.2.7
EOS	B.2.8
HOME	B.1.8
ILINE	B.2.9
INSERT	B.2.10
LDCURS	B.3.5
LDFAS	B.3.6
LDSAS	B.3.7
LDTAS	B.3.8
LMOVE	B.3.9
NEWFRM	B.2.11
RECON	B.3.10
RETURN	B.1.9
RMOVE	B.3.11
SMOVE	B.3.12
STFAS	B.3.13
STSAS	B.3.14
STTAS	B.3.15
SUBREG	B.3.16
SUBT2	B.3.17
TAB	B.1.10
UPAGE	B.4.3
USCROL	B.4.4

3.2 CODE FIELD

This field contains a code which identifies the machine operation (add, subtract, etc.) to be performed; hence the term operation code, or op code. The instructions described in Section 5 are each identified by a mnemonic label which must appear in the code field. For example, since the branch instruction is identified by the letter "B", this letter must appear in the code field to identify the instruction as "Branch".

There must be at least one space following the code field. Thus,

```
     HERE    B         THERE
```

is legal, but

```
     HERE    BTHERE
```

is illegal.

3.3 OPERAND FIELD

This field contains information used in conjunction with the code field to precisely define the operation to be performed by the instruction. Depending upon the code field, the operand field may consist of one or more items, where items are separated by a comma.

Legal operands are as follows:

1. A register code. The codes RA, RB, RC, RD, RE, RH, and RL specify registers A, B, C, D, E, H, and L, respectively, as a source or destination for the operation.

   Example:

Label	Code	Operand	Comment
HERE	LR	RA,RC	Load C into A

   specifies that the contents of register C (the source register) is to be loaded into register A (the destination register).

2. A hexadecimal, decimal, or ASCII constant. Hexadecimal constants can be from one to four digits. Each hexadecimal constant must be enclosed with an X and single quotes.

   Example:

Label	Code	Operand	Comment
HERE	LBI	RC,X'3F'	Load register C with hex. 3F
	DC	X'1000',X'2F',X'3566'	

Decimal constants can be from one to five decimal digits, not to exceed 32676 maximum or -32768 minimum. Decimal constants are written without any operators.

Example:

```
Label     Code     Operand      Comment

HERE      LBI      RC,63        Load register C with 63
          DC       66,5,128,32000
```

An ASCII constant is one or more ASCII characters enclosed in single quotes. Appendix D contains a list of legal ASCII characters and their hexadecimal representations.

Example:

```
Label     Code     Operand      Comment

CHAR      LBI      RC,'*'       Load register C with
*                               eight-bit ASCII
*                               representation of an
*                               asterisk
```

3. Labels that appear in the label field of another instruction.

   Example:

```
Label     Code     Operand      Comment

HERE      B        THERE        Jump to instruction at
*                               THERE

          ---

          ---

THERE     LBI      RC,X'3F'
```

4. The current program counter. This is specified as the character '*' and is equal to the address of the current instruction.

   Example:

```
Label     Code     Operand

GO        B        *+6
```

   This instruction causes program control to be transferred to the address six bytes beyond the location of the B instruction.

5. An expression. An expression is a symbol, a constant, or a series of such items separated by the arithmetic operators + (plus) or - (minus). The following instructions illustrate the use of expressions:

```
Code    Operand

B       LOOP+4
B       TABLE+X'12'
B       STOP-GO+2
LBI     RA,-FROG
LBI     RA,'A'+1
```

An expression is absolute if its value is absolute. Similarly, an absolute expression does not change as a function of the physical location of the program in memory. The value of a relocatable expression does change when the location of the program changes. The relocatable value changes by the difference in byte locations between the originally assigned area of storage.

An expression, when evaluated, produces a value which is considered absolute or relocatable according to the rules outlined in Table 3-2.

TABLE 3-2

ABSOLUTE AND RELOCATABLE EXPRESSION RULES

	A+B	A-B
A is absolute, B is absolute	Absolute	Absolute
A is absolute, B is relocatable	Relocatable	Invalid
A is relocatable, B is absolute	Relocatable	Relocatable
A is relocatable, B is relocatable	Absolute	Absolute

3.4 COMMENT FIELD

The only rule governing this field is that it must be separated from the operand field by at least one space.

In addition, a comment field may appear alone on a line by coding an asterisk into column 1. This is useful for general program comments, such as

```
*  Begin loop here
```

or for comment field continuations, such as

```
Label    Code    Operand    Comment
CHAR     LBI     RC,'*'     Load register C with eight-bit
*                           ASCII representation of an
*                           asterisk
```

3.5 ASSEMBLER LISTING FIELDS

On the assembly listing there are five fields of information associated with each source statement. These fields appear on the listing to tbe left of each assembly language statement.

Field 1 is the ERROR flag. It is an error diagnostic associated with the source statement. The meaning of the error flag is as follows:

ERROR CODE	MEANING
1	Error in the operand field
2	Multiply defined symbol
4	Undefined symbol
8	Op-code undefined

Either above or combination of above values.

Field 2 is the line number associated with the source statement to be used for future editing references.

Field 3 is the memory location (relative or absolute) of the object code generated by the source statement.

Field 4 is the ADDRESS TYPE. All addresses (or data) encountered which are relocatable or external will be so indicated by "R" or "E", respectively.

Field 5 is the listing of object code generated by the assembly language statement.

SECTION 4

ASSEMBLER DIRECTIVES

There are two types of assembler directives: pseudo-instructions and declaration instructions.

Pseudo-instructions provide the assembler with various types of information pertaining to the program about to be assembled, and how the results of the assembly should be printed. This print is called an assembly listing.

Pseudo-instructions (Section 4.1) are written in a source program, but unlike the instructions in Section 5, pseudo-instructions generate no object code.

Declaration instructions (Section 4.2) are used to generate data constants and addresses that are output by the assembler as part of the object code.

4.1 PSEUDO-INSTRUCTIONS

Pseudo-instructions provide the assembler with information that will be used when it generates object code.

4.1.1 EJT Eject A Page

Assembler Format: EJT

This pseudo-instruction is used to separate assembler listings for easy reading. When EJT is encountered in a source program, the printer attached to the 9003 advances to the top of the next page.

NOTE

The EJT instruction cannot be labelled.

4.1.2 END End of Program

Assembler Format: END

The END pseudo-instruction signifies to the assembler that the physical end of the program has been reached, and that generation of the object program and (possibly) listing of the source program should now begin.

One, and only one, END instruction must appear in every assembly, and it must be the (physically) last statement of the assembly.

END may be labelled.

4.1.3 ENTY Identify Entry Point

Assembler Format: ENTY symbol$_1$,symbol$_2$,etc.

The ENTY pseudo-instruction identifies symbols in this program that may be used by other programs. This permits programs that are assembled separately to communicate with each other. Only those symbols identified as entry symbols (symbol$_1$,symbol$_2$,etc.) are available to other separately-assembled programs. All ENTY statements must precede any symbols they reference in the program.

NOTE

> ENTY instructions cannot be labelled.

Example:

```
          ENTY    IN1,IN2
           .
           .
           .
     IN1  EQU     *
           .
           .
           .
     IN2  LR      RA,RC
           .
           .
           .
          END
```

4.1.4 EQU Equate

Assembler Format: name EQU exp

The symbol "name" is assigned the value "exp" by the assembler, where "exp" is any defined expression. Whenever the symbol "name" is encountered subsequently in the assembly, this value will be used.

Each EQU instruction <u>must</u> be labelled.

NOTE

> A symbol may appear in the label field of only one EQU pseudo-instruction. That is, an EQU symbol cannot be re-defined.

Example:

```
     LABEL     EQU    TAG
     HERE      EQU    *
     LENGTH    EQU    BOTTOM-TOP
     START     EQU    X'2000'
```

Subsequently, for instance, the instruction

```
          CALL  LABEL
```

would actually cause the same effect as

```
          CALL  TAG
```

4.1.5 EXTN Identify External Symbol

Assembler Format: EXTN symbol$_1$,symbol$_2$...,symbol$_n$

The EXTN pseudo-instruction identifies symbols in another program that are referenced by this program. This permits programs that are assembled separately to communicate with each other. Only those symbols identified as ENTY symbols (see Section 4.1.3) in another program can be identified as externally-defined in this program. All EXTN statements must precede any references to the external symbols within the program.

Example:

The sequence

```
          EXTN    XOUT1,XOUT2
            :
          CALL    XOUT1
            :
          LHI    RH,XOUT2
            :
          END
```

would allow the program to use the XOUT1 and XOUT2 symbols that are defined in an external program.

Any symbols declared external have the following restrictions:

1. EXTN symbols must not be combined in arithmetic expressions. So

   ```
   LHI  RH,XOUT1+3
   ```

   is illegal.

2. EXTN symbols must only be used in a two-byte address field; i.e.,

```
LHI     RH,XOUT1
B       XOUT1
CALL    XOUT1
```

3. EXTN symbols may not be used with assembler pseudo-instructions such as EQU, END, etc.

EXTN and ENTY instructions are particularly valuable for sharing subroutines. Rather than having to assemble the main program and its subroutines at the same time to establish correct communication, the EXTN/ENTY capability permits the main program to be assembled and then loaded with the previously-assembled subroutines. The symbols identified by EXTN or ENTY statements are then linked at load time by the Zentec Linking Loader. Such modularity will enhance both the structure of program design and the productivity of the programmer.

Consider the following two hypothetical programs:

```
Label     Operation     Operand
*  Main program
*
          EXTN          XOUT1,XOUT2
START     LHI           RH,XOUT1
          LBI           RC,X'FF'
           :
          STB           RA
          CALL          XOUT2
          B             START
          END

*  Subroutine XOUT2
*
          ENTY          XOUT1,XOUT2
XOUT2     EQU           *
          LB            RA
           :
          RET
 XOUT1    DS            2
          END
```

The symbols XOUT1 and XOUT2 are used by the main program, but their values are not known at assembly time. Since they are defined as EXTN, the symbols XOUT1 and XOUT2 and the location at which they are referenced in the main program are output to the object file together with the rest of the assembled program.

In a similar fashion, when the subroutine XOUT2 is assembled, the symbols XOUT1 and XOUT2 are output along with the subroutine.

As the main program and subroutines are loaded, the loader accumulates a table of references to symbols and their values. This information is used by the loader to link the main program and subroutine by replacing every reference to XOUT1 and XOUT2 by the values passed on from the subroutine by the ENTY instruction.

4.1.6 ORG Origin

Assembler Format: ORG exp

The assembler's location counter is set to the value of "exp", which must be a valid 16-bit memory address. The next instruction or data byte(s) assembled will be assembled at address exp, exp+1, etc.

If no ORG instruction appears before the first instruction or data byte in the program, assembly will begin at relative location 0. The ORG statement must precede any EXTN and ENTY op-codes which define variables to be modified by the ORG instruction.

Example 1:

```
  Hex
Address    Label    Code    Operand

                    ORG     X'1000'
 1000               LR      RA,RC
 1001               AI      RA,2
 1003               B       NEXT
           HERE     ORG     X'1050'
 1050      NEXT     XR      RA,RA
```

The first ORG pseudo-instruction informs the assembler that the object program will begin at memory address X'1000'. The second ORG tells the assembler to set its location counter to X'1050' and continue assembling machine instructions or data bytes from that point. The label HERE refers to memory location X'1050', since this is the address immediately following the jump instruction. Note that the portion of memory from X'1006' to X'104F' is still included in the object program, but does not contain assembled data. In particular, the programmer should not assume that these locations will contain zero, or any other value.

Example 2:

The ORG pseudo-instruction can perform a function equivalent to the DS (Define Storage) instruction in Section 4.2.2. The following two sections of code are exactly equivalent:

```
           LR   RA,RC    |          LR   RA,RC
           B    NEXT     |          B    NEXT
           DS   12       |          ORG  *+12
      NEXT XR   RA,RA    |     NEXT XR   RA,RA
```

4.1.7 SPC Space

Assembler Format: SPC number

This pseudo-instruction causes the printer to space "number" lines. If the number of lines to be spaced exceeds the number of lines remaining on the page, this instruction has the same effect as EJT (see Section 4.1.11).

4.2 DECLARATION INSTRUCTIONS

A declaration instruction differs from a pseudo-instruction (Section 4.1) in that a declaration instruction actually creates object code.

Declaration instructions reserve memory locations, either with specified contents (DC instruction) or without specified contents (DS instruction).

4.2.1 DC Define Constant

The format of the DC instruction is

DC constant$_1$,constant$_2$...,constant$_n$

where "constant" is either a hexadecimal, decimal, or ASCII constant or a label that is specified following the format in Section 3.3. The constant may be absolute, relocatable or external.

Each DC instruction will reserve the number of memory locations required to store its constants and fill those locations with the bit patterns that represent the constants.

Hexadecimal constants can be from one to four digits. One- or two-digit hexadecimal constants will be stored in one memory location, whereas three- or four-digit hexadecimal constants will be stored in two memory locations. Any two-digit constant (address) will be stored in reverse order, except for the ASCII strings. Example 1:

```
        CONS  DC  X'1F32', X'FB'
```

will store X'32' into memory location CONS, X'1F' into CONS+1 and X'FB' into CONS+2.

ASCII constants can be any length and will use one memory location for each character.

Example 2:

```
        MESG1  DC  'LOAD THE TAPE'
```

will use thirteen memory locations, storing the appropriate ASCII code (see Appendix D) into each location.

A label represents a storage address that is translated into a constant. It is a relocatable, external or absolute constant as determined by the combinations of symbols and constants in the expression.

Example 3:

```
          CONL  DC  XOUT1
                DC  XIN+2
                DC  XIN1-XIN2
```

will cause the address constant stored to be relocatable or absolute as determined by the rules given in Table 3-2 of Section 3.3.

The following example shows how a single DC instruction can be used to define different types of data. Each operand is separated from the next by a comma.

Example 4:

```
     TABLE  DC  X'OFDE',X'FF'
            DC  'START OF PROG', 598
            DC  XIN, XOUT
```

4.2.2 DS Define Storage

The format of the DS instruction is

```
     DS  exp
```

where "exp" is an expression that is specified following the format in Section 3.3. This expression must be a constant (hexadecimal or decimal) or must reduce to a constant when evaluated. The value of the expression determines the number of memory locations that will be reserved. Although the DS instruction reserves memory locations, it does not alter the current contents of those locations. The programmer should not assume that these locations contain zero, or any other value.

Example:

```
     DS  80          Reserves 80 memory locations
     DS  TOP-BOTTOM  Reserves (TOP-BOTTOM) memory locations
```

4.2.3 DB Define Byte

The format of the DB instruction is:

DB constant$_1$, constant$_2$, ..., constant$_n$

where "constant" is either a hexadecimal, decimal, or ASCII constant or a label that is specified following the format in Section 3.3.

Each DB instruction will reserve only one byte of memory location for each constant and fill that location with the least significant byte that represents the constant.

Example:

```
    CONS  DB  TAG, END-START, X'FB'
```

where TAB = X'1F8B' and END-START = '01FC'. The DB will store X'8B' into CONS, X'FC' into CONS+1 and X'FB' into CONS+2. Only absolute values can be stored

SECTION 5

THE 9000 INSTRUCTION SET

Assembly language instructions can be classified by groups, and when learning assembly language, it is advisable to study individual instructions in a logical sequence.

For simple reference purposes, descriptions of instructions are easier to find if instructions are documented in alphabetic order of the mnemonic.

This section groups instructions by group, then instructions are described in alphabetic order, starting at Section 5.2. Instructions printed in *italics* can be used only with the 9003.

The hexadecimal codes shown for each instruction are for the 9003.

The entire instruction set for both the 9002 and the 9003 is summarized in alphabetic and in hexadecimal order in Appendix A.

5.1 INSTRUCTIONS GROUPS

The 9000 instruction set can be divided into the following groups:

Group	Section
Carry Bit	5.2
Single Register	5.3
NOP	5.4
Data Transfer	5.5
Register or Memory To Accumulator	5.6
Rotate Accumulator	5.7
Register Pair	5.8
Direct Addressing	5.9
Branch	5.10
Call Subroutine	5.11
Return From Subroutine	5.12
Interrupt Enable/Disable	5.13
Input/Output	5.14

Within the groups, instructions can be further classified by their individual attributes. For example, within the Data Transfer Group there are Store and Load instructions. Within these instruction types there are three different Store instructions and four different Load instructions. However, they all have one thing in common: they all transfer data back and forth between memory and the working registers, hence their group name, Data Transfer.

5.2 CARRY BIT INSTRUCTIONS

This section describes the instructions which operate directly upon the Carry bit.

5.2.1 COMC Complement Carry

Assembler Format: COMC

0	0	1	1	1	1	1	1

If the Carry bit=0, it is set to 1. If the Carry bit=1, it is reset to 0.

Operation: $\mathit{Carry} \leftarrow \overline{\mathit{Carry}}$

Condition bits affected: Carry

5.2.2 SETC Set Carry

Assembler Format: SETC

0	0	1	1	0	1	1	1

The Carry bit is set to one.

Operation: $\mathit{Carry} \leftarrow 1$

Condition bits affected: Carry

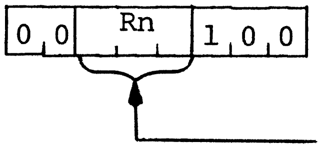
5.3 SINGLE REGISTER INSTRUCTIONS

This section describes instructions which operate on a single register or memory location. If a memory reference is specified, the memory byte addressed by the H and L registers is operated upon. The H register holds the most significant 8 bits of the address while the L register holds the least significant 8 bits of the address.

5.3.1 BUMP Bump Register or Memory

Assembler Format: BUMP Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)

or BUMP M (M is optional)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
110 for memory ref. M
111 for register A

The specified register *or memory byte* is incremented by one.

Operation: Rn ← Rn+1

Condition bits affected: Zero, Sign, Parity, *Auxiliary Carry*

Example:

If register C contains X'99', the instruction

```
BUMP   RC
```

will cause register C to contain X'9A'.

If register C contains X'FF', the instruction

```
BUMP   RC
```

will cause register C to contain X'00' and the Zero bit to set.

5.3.2 COM Complement Accumulator

Assembler Format: COM RA

0	0	1	0	1	1	1	1

Each bit of the contents of the Accumulator is complemented (producing the one's complement).

Operation: $RA \leftarrow \overline{RA}$

Condition bits affected: None

Example:

> *If the Accumulator contains X'51', the instruction*
>
> *COM RA*
>
> *will cause the Accumulator to contain X'AE'.*

5.3.3 DAA Decimal Adjust Accumulator

Assembler Format: DAA RA

0	0	1	0	0	1	1	1

The eight-bit hexadecimal number in the Accumulator is adjusted to form two four-bit binary-coded-decimal digits by the following two-step process:

1. *If the least significant four bits of the Accumulator represents a number greater than 9, or if the Auxiliary Carry bit is equal to one, the Accumulator is incremented by six. Otherwise, no incrementing occurs.*

2. *If the most significant four bits of the Accumulator now represent a number greater than 9, or if the normal Carry bit is equal to one, the most significant four bits of the Accumulator are incremented by six. Otherwise, no incrementing occurs.*

If a carry out of the least significant four bits occurs during step 1, the Auxiliary Carry bit is set; otherwise, it is reset. Likewise, if a carry out of the most significant four bits occurs during step 2, the normal Carry bit is set; otherwise, it is unaffected.

NOTE

> The instruction is used when adding decimal numbers. It is the only instruction whose operation is affected by the Auxiliary Carry bit.

Operation: If $(A_0\text{-}A_3)>9$ or (Aux. Carry)=1, $(A) \leftarrow (A)+6$

Then if $(A_4\text{-}A_7)>9$ or (Carry)=1, $(A)=(A)+6\cdot 2^4$

Condition bits affected: Zero, Sign, Parity, Carry, Auxiliary Carry

Example:

Suppose the Accumulator contains X'9B', and both Carry bits=0. The DAA instruction will operate as follows:

1. Since bits 0-3 are greater than 9, add 6 to the Accumulator. This addition will generate a carry out of the lower four bits, setting the Auxiliary Carry bit.

```
Accumulator = 1001 1011 = X'9B'
+6          =      0110
              ---------
              1010 0001 = X'A1'
                   \
                    -->Auxiliary Carry=1
```

2. Since bits 4-7 now are greater than 9, add 6 to these bits. This addition will generate a carry out of the upper four bits, setting the Carry bit.

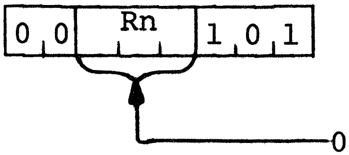
```
Accumulator = 1010 0001 = X'A1'
+6          = 0110
              ---------
          ┌-1] 0000 0001
          |
          └->Carry=1
```

Thus, the Accumulator will now contain 1, and both Carry bits will be=1.

5.3.4 DEC Decrement Register or Memory

Assembler Format: DEC Rn (where Rn = RA, RB, RC, RD, RE, RH,
or RL)

or DEC M (M is optional)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
110 for memory ref. M
111 for register A

The specified register *or Memory byte* is decremented by one.

Operation: Rn ← Rn-1

Condition bits affected: Zero, Sign, Parity, *Auxiliary Carry*

Example:

If register C contains X'99', the instruction

```
DEC  RC
```

will cause register C to contain X'98'.

5.4 NOP INSTRUCTION

This instruction causes no operation.

5.4.1 NOP No Operation

Assembler Format: NOP

0	0	0	0	0	0	0	0

No operation occurs. Operation proceeds with the next sequential instruction.

Operation: No operation

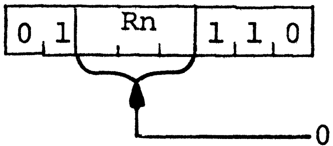
Condition bits affected: None

5.5 DATA TRANSFER INSTRUCTIONS

This section describes instructions that transfer data between registers or between memory and a register.

5.5.1 LB Load Byte

Assembler Format: LB Rn (where Rn = RA, RB, RC, RD, RE, RH,
or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The contents of the memory location addressed by registers H and L replace the contents of the specified register.

Operation: Rn ← M

Condition bits affected: None

Example:

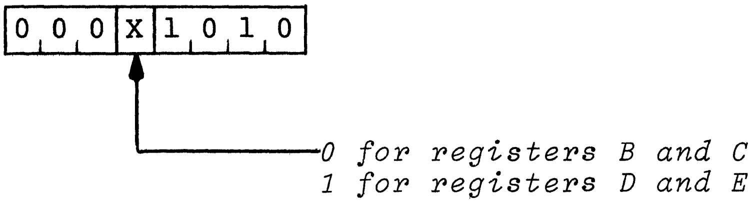
If register H contains X'13' and register L contains X'8B', the instruction

```
LB   RC
```

will load register C with the contents of memory location X'138B'.

5.5.2 LBA Load Byte to Accumulator

Assembler Format: LBA RB or LBA RD



The contents of the memory location addressed by register pair B and C, or by register pair D and E, replace the contents of the Accumulator.

Operation: RA ← M

Condition bits affected: None

Example:

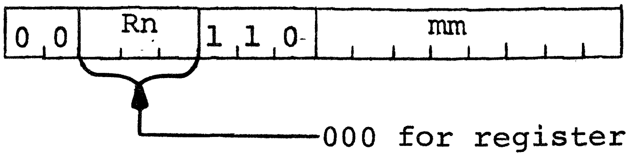
If register B contains X'21' and register C contains X'03', the instruction

```
LBA  RB
```

will load the Accumulator with the contents of memory location X'2103'.

5.5.3 LBI Load Byte Immediate

Assembler Format: LBI Rn,mm (where Rn = RA, RB, RC, RD, RE, RH,
or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The byte of immediate data replaces the contents of the specified register.

Operation: Rn ← mm

Condition bits affected: None

Example:

The instruction

```
LBI  RC,X'12'
```

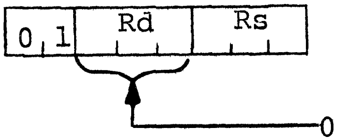
will load X'12' into register C, whereas the instruction

```
LBI  RC,12
```

will load X'0C' (i.e., 12_{10}) into register C.

5.5.4 LR Load Register

Assembler Format: LR Rd,Rs (where Rd,Rs = RA, RB, RC, RD, RE, RH, or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

One byte of data is moved from the register specified by Rs (the source register) to the register specified by Rd (the destination register). The data replaces the contents of the destination register; the source remains unchanged.

Operation: Rd ← Rs

Condition bits affected: None

Examples:

> If register A contains X'9A' and register B contains X'0C', the instruction
>
> ```
> LR   RB,RA
> ```
>
> will cause registers A and B to both contain X'9A'. Instructions of the type
>
> ```
> LR   RB,RB
> ```
>
> can be used as no-op instructions.

5.5.5 STA Store Accumulator

Assembler Format: STA RB or STA RD

0	0	0	X	0	0	1	0

0 for registers B and C
1 for registers D and E

The contents of the Accumulator are stored in the memory location addressed by register pair B and C, or by register pair D and E.

Operation: M ← RA

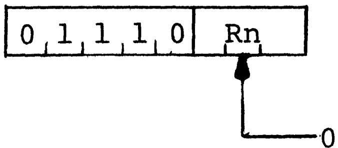
Condition bits affected: None

Example:

> *If register B contains X'21' and register C contains X'03', the instruction*
>
> *STA RB*
>
> *will store the contents of the Accumulator at memory location X'2103'.*

5.5.6 STB Store Byte

Assembler Format: STB Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The contents of the specified register is stored into the memory location addressed by registers H and L.

Operation: M ← Rn

Condition bits affected: None

Example:

> If register H contains X'13', register L contains X'8B', and register C contains X'1C', the instruction
>
> STB RC
>
> will store X'1C' into memory location X'138B'.

5.5.7 STBI Store Byte Immediate

Assembler Format: STBI mm

0	0	1	1	0	1	1	0	mm

The byte of immediate data is stored into the memory location addressed by registers H and L.

Operation: M ← mm

Condition bits affected: None

Example:

> If register H contains X'23' and register L contains X'8B', the instruction
>
> ```
> STBI   X'1C'
> ```
>
> will store X'1C' into memory location X'238B'.

5.6 REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS

This instruction group alters the contents of register A, the Accumulator.

5.6.1 A Add

Assembler Format: A RA

1	0	0	0	0	1	1	0

The contents of the memory location addressed by registers H and L is added to register A using two's complement arithmetic.

Operation: RA ← RA+M

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

If register H contains X'13' and register L contains X'8B', memory location X'138B' will be addressed. So, if memory location X'138B' contains X'8A' and register A contains X'0C', the instruction

A RA

will cause the following addition to be performed:

```
Register A = 0000 1100 = X'0C'
    Memory = 1000 1010 = X'8A' = -X'76'
    Result = 1001 0110 = X'96' = -X'6A'
```

To summarize the results:

```
Register A = X'96'
Carry      = 0
Sign       = 1
Zero       = 0
Parity     = 1
Aux. Carry = 1
```

5.6.2 AC Add Carry

Assembler Format: AC RA

1	0	0	0	1	1	1	0

The contents of the memory location addressed by registers H and L, plus the Carry, is added to register A using two's complement arithmetic.

Operation: RA ← RA+M+Carry

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

If register H contains X'13' and register L contains X'8B', memory location X'138B' will be addressed. So, if memory location X'138B' contains X'3D', register A contains X'42, and the Carry bit=0, the instruction

```
AC  RA
```

will perform the addition as follows:

```
 X'3D' = 0011 1101
 X'42' = 0100 0010
 Carry = _________0
Result = 0111 1111 = X'7F'
```

The results are:

```
Register A = X'7F'
Carry      = 0
Sign       = 0
Zero       = 0
Parity     = 0
Aux. Carry = 0
```

If the Carry bit had been one at the beginning of the example, the following would have occurred:

```
 X'3D' = 0011 1101
 X'42' = 0100 0010
 Carry = _________1
Result = 1000 0000 = X'80'
```

```
Register A = X'80'
Carry      = 0
Sign       = 1
Zero       = 0
Parity     = 0
Aux. Carry = 1
```

5.6.3 ACI Add Carry Immediate

Assembler Format: ACI RA,mm

1	1	0	0	1	1	1	0	mm

The byte of immediate data is added to the contents of register A plus the contents of the Carry bit.

Operation: RA ← RA+mm+Carry

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

If Carry=1 and register A contains X'42', the instruction

```
ACI   RA,X'3D'
```

will cause the following addition to occur:

```
 X'3D' = 0011 1101
 X'42' = 0100 0010
 Carry = _________1
Result = 1000 0000 = X'80'
```

The results are:

```
Register A = X'80'
Carry      = 0
Sign       = 1
Zero       = 0
Parity     = 0
Aux. Carry = 1
```

5.6.4 AI Add Immediate

Assembler Format: AI RA,mm

1	1	0	0	0	1	1	0	mm

The byte of immediate data is added to the contents of register A using two's complement arithmetic.

Operation: RA ← RA+mm

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

Label	Code	Operand
AD1	LBI	RA,20
AD2	AI	RA,66
AD3	AI	RA,-66

The instruction at AD1 loads register A with X'14'. The instruction at AD2 performs the following addition:

```
  X'14' = 0001 0100
  X'42' = 0100 0010
         ----------
 Result = 0101 0110 = X'56'
```

The parity bit is set; all other bits are reset.

The instruction at AD3 restores the original contents of register A. The Carry, Auxiliary Carry and Parity bits are set. The Zero and Sign bits are reset.

5.6.5 AND Logical AND

Assembler Format: AND RA

1	0	1	0	0	1	1	0

The byte in the memory location addressed by registers H and L is logically ANDed, bit by bit, with register A. The Carry bit is reset to zero.

The logical AND of two bits produces 1 if and only if both bits equal 1.

Operation: $RA \leftarrow RA \wedge M$, Carry $\leftarrow 0$

Condition bits affected: Carry, Zero, Sign, Parity

Example:

Since any bit ANDed with a zero produces a zero and any bit ANDed with a one remains unchanged, the AND function is often used to zero groups of bits.

If register H contains X'13' and register L contains X'8B', memory location X'138B' will be addressed. So, if location X'138B' contains X'0F' and register A contains X'FC', the instruction

```
AND  RA
```

will act as follows:

```
Register A = 1111 1100 = X'FC'
    Memory = 0000 1111 = X'0F'

    Result = 0000 1100 = X'0C' in register A
```

This particular example guarantees that the high-order four bits of the accumulator are zero, and the low-order four bits are unchanged.

5.6.6 ANDI AND Immediate

Assembler Format: ANDI RA,mm

1	1	1	0	0	1	1	0	mm

The byte of immediate data is logically ANDed with the contents of register A. The Carry bit is reset to zero.

Operation: RA ← RA ∧ mm, Carry ← 0

Condition bits affected: Carry, Zero, Sign, Parity

Example:

Consider the instruction sequence

```
LR     RA,RC
ANDI   RA,X'0F'
```

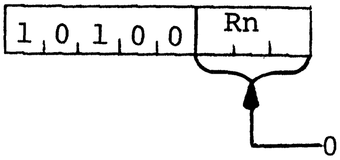
The contents of the C register are moved to register A. The ANDI instruction then zeroes the high-order four bits, leaving the low-order four bits unchanged. The Zero bit will be set if and only if the low-order four bits were originally zero.

If the C register contained X'3A', the ANDI would perform the following:

```
Register A = 0011 1010 = X'3A'
        mm = 0000 1111 = X'0F'
    Result = 0000 1010 = X'0A' in register A
```

5.6.7 ANDR AND Register

Assembler Format: ANDR RA,Rn (where Rn = RA, RB, RC, RD, RE, RH,
or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The specified register is logically ANDed, bit by bit, with the contents of register A. The Carry bit is reset to zero.

The logical AND function of two bits is 1 if and only if both the bits equal 1.

Operation: $RA \leftarrow RA \wedge Rn$, $Carry \leftarrow 0$

Condition bits affected: Carry, Zero, Sign, Parity

Example:

Since any bit ANDed with a zero produces a zero and any bit ANDed with a one remains unchanged, the AND function is often used to zero groups of bits.

Assuming that register A contains X'3A' and the C register contains X'0F', the instruction

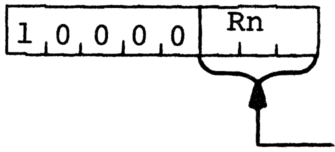
```
ANDR  RA,RC
```

will act as follows:

```
Register A = 0011 1010 = X'3A'
Register C = 0000 1111 = X'0F'
    Result = 0000 1010 = X'0A' in register A
```

5.6.8 AR Add Register

Assembler Format: AR RA,Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The specified register is added to the contents of register A using two's complement arithmetic.

Operation: RA ← RA+Rn

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example 1:

Assume that the D register contains X'2E' and register A contains X'6C'. Then the instruction

```
AR  RA,RD
```

will perform the addition as follows:

```
 X'2E' = 0010 1110
 X'6C' = 0110 1100
Result = 1001 1010 = X'9A' in register A
```

The Zero and Carry bits are reset; the Parity and Sign bits are set. *Since there is a carry out of bit A_3, the Auxiliary Carry bit is set.*

Example 2:

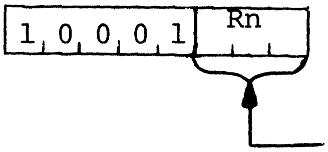
The instruction

```
AR  RA,RA
```

will double the contents of register A.

5.6.9 ARC Add Register Carry

Assembler Format: ARC RA,Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The contents of the specified register, plus the Carry bit, is added to register A using two's complement arithmetic.

Operation: RA ← RA+Rn+Carry

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

Assume that register C contains X'3D', register A contains X'42', and the Carry bit=0. The instruction

```
ARC  RA,RC
```

will perform the addition as follows:

```
 X'3D' = 0011 1101
 X'42' = 0100 0010
 Carry = _________0
Result = 0111 1111 = X'7F'
```

The results can be summarized as follows:

```
Register A = X'7F'
Carry      = 0
Sign       = 0
Zero       = 0
Parity     = 0
Aux. Carry = 0
```

5.6.10 C Compare

Assembler Format: C RA

1	0	1	1	1	1	1	0

The contents of the memory location addressed by, registers H and L is compared with the contents of register A. The comparison is performed by internally subtracting the contents of memory from register A (leaving both unchanged) and setting the condition bits according to the result. The Zero bit is set if the quantities are equal, and reset if they are unequal. Since a subtract operation is performed, the Carry bit will be set if there is no carry out of bit 7, indicating that the contents of memory are greater than the contents of register A, and reset otherwise.

NOTE

> If the two quantities to be compared differ in sign, the sense of the Carry bit is reversed.

Operation: (RA-M)

Condition bits affected: Carry, Zero, Sign, Parity, *Auxiliary Carry*

Example 1:

Assume that register A contains the number X'0A' and memory contains the number X'05'. Then the instruction

C RA

performs the following internal subtraction:

```
    X'0A' = 0000 1010
 -(X'05') = 1111 1011
         ┌─1] 0000 0101 = Result
         └──►carry=1, causing the Carry bit to be reset
```

Register A still contains X'0A' and memory still contains X'05'; however, the Carry bit is reset and the Zero bit reset, indicating that memory is less than register A.

Example 2:

If register A had contained the number X'02', the internal subtraction would have produced the following:

```
     X'02'  = 0000 0010
  -(X'05')  = 1111 1011
          ┌──0} 1111 1101 = Result
          │
          └──► carry=0, Carry bit=1
```

The Zero bit would be reset and the Carry bit set, indicating memory greater than A.

5.6.11 CI Compare Immediate

Assembler Format: CI RA,mm

1	1	1	1	1	1	1	0	mm

The byte of immediate data is compared to the contents of register A.

The comparison is performed by internally subtracting the data from register A using two's complement arithmetic, leaving register A unchanged but setting the condition bits by the result.

The Zero bit is set if the quantities are equal, and reset if they are unequal.

Since a subtract operation is performed, the Carry bit will be set if there is not carry out of bit 7, indicating the immediate data is greater than the contents of register A, and reset otherwise.

NOTE

> If the two quantities to be compared differ in sign, the sense of the Carry bit is reversed.

Operation: (RA-mm)

Condition bits affected: Carry, Zero, Sign, Parity, *Auxiliary Carry*

Example:

Consider the instruction sequence

```
LBI  RA,X'4A'
CI   RA,X'40'
```

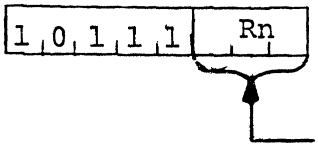
The CI instruction performs the following internal subtraction:

```
   X'4A' = 0100 1010
-(X'40') = 1100 0000
           ---------
    ----1] 0000 1010
   |
    ----->carry out=1, causing the Carry bit to be reset
```

Register A still contains X'4A', but the Zero bit is reset indicating that the quantities were unequal, and the Carry bit is reset indicating mm is less than the register A.

5.6.12 CR Compare Registers

Assembler Format: CR RA,Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)



000	for register B
001	for register C
010	for register D
011	for register E
100	for register H
101	for register L
111	for register A

The contents of the specified register is compared with the contents of register A. The comparison is performed by internally subtracting the contents of register n from register A (leaving both unchanged) and setting the condition bits according to the result. The Zero bit is set if the quantities are equal, and reset if they are unequal. Since a subtract operation is performed, the Carry bit will be set if there is no carry out of bit 7, indicating that the contents of register n are greater than the contents of register A, and reset otherwise.

NOTE

> If the two quantities to be compared differ in sign, the sense of the Carry bit is reversed.

Operation: (RA-Rn)

Condition bits affected: Carry, Zero, Sign, Parity, *Auxiliary Carry*

Example 1:

Assume that register A contains the number X'0A' and RC contains the number X'05'. Then the instruction

```
CR  RA,RC
```

performs the following internal subtraction:

```
   X'0A' = 0000 1010
-(X'05') = 1111 1011
           ---------
 ┌─────1] 0000 0101 = Result
 │
 └──►carry=1, causing the Carry bit to be reset
```

Register A still contains X'0A' and register C still contains X'05'; however, the Carry bit is reset and the Zero bit reset, indicating that memory is less than A.

Example 2:

If register A had contained the number X'02', the internal subtraction would have produced the following:

```
      X'02' = 0000 0010
   -(X'05') = 1111 1011
          ┌──0] 1111 1101 = Result
          │
          └──►carry=0, Carry bit=1
```

The Zero bit would be reset and the Carry bit set, indicating memory greater than A.

5.6.13 O Logical OR

Assembler Format: O RA

1 0 1 1 0 1 1 0

The contents of the memory location addressed by registers H and L is logically ORed, bit by bit, with the contents of register A. The Carry bit is reset to zero.

The logical OR function of two bits equals zero if and only if both bits equal zero.

Operation: RA ← RA V M, Carry ← 0

Condition bits affected: Carry, Zero, Sign, Parity

Example:

Since any bit ORed with a one produces a one, and any bit ORed with a zero remains unchanged, the O function is often used to set groups of bits to one.

If register H contains X'13' and register L contains X'8B', memory location X'138B' contains X'0F' and register A contains X'33', the instruction

```
O   RA
```

acts as follows:

```
Register A = 0011 0011 = X'33'
    Memory = 0000 1111 = X'0F'
    Result = 0011 1111 = X'3F' in register A
```

This particular example guarantees that the low-order four bits of register A are one, and the high-order four bits are unchanged.

5.6.14 OI OR Immediate

Assembler Format: OI RA,mm

1	1	1	1	0	1	1	0	mm

The byte of immediate data is logically ORed with the contents of register A.

The result is stored in register A. The Carry bit is reset to zero, while the Zero, Sign, and Parity bits are set according to the result.

Operation: RA ← RA V mm, Carry ← 0

Condition bits affected: Carry, Zero, Sign, Parity

Example:

Consider the instruction sequence

```
LR  RA,RC
OI  RA,X'0F'
```

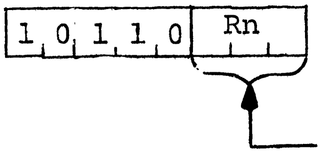
If the C register contained X'B5', the OI would perform the following:

```
Register A = 1011 0101 = X'B5'
        mm = 0000 1111 = X'0F'
    Result = 1011 1111 = X'BF' in register A.
```

Thus the contents of the C register are moved to register A. The OI instruction then sets the low-order four bits to one, leaving the high-order four bits unchanged.

5.6.15 OR OR Registers

Assembler Format: OR RA,Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The specified register is logically ORed, bit by bit, with the contents of the accumulator. The Carry bit is reset to zero.

The logical OR function of two bits equals zero if and only if both the bits equal zero.

Operation: RA ← RA V Rn

Condition bits affected: Carry, Zero, Sign, Parity

Example:

> Since any bit ORed with a one produces a one, and any bit ORed with a zero remains unchanged, the OR function is often used to set groups of bits to one.
>
> Assuming that register C contains X'0F' and register A contains X'33', the instruction:
>
> OR RA,RC
>
> acts as follows:
>
> Register A = 0011 0011 = X'33'
> Register C = 0000 1111 = X'0F'
> Result = 0011 1111 = X'3F' in register A
>
> This particular example guarantees that the low-order four bits of register A are one, and the high-order four bits are unchanged.

5.6.16 S Subtract

Assembler Format: S RA

1	0	0	1	0	1	1	0

The contents of the memory location addressed by registers H and L is subtracted from register A using two's complement arithmetic.

If there is no carry out of the high-order bit position, indicating that a borrow occurred, the Carry bit is set; otherwise it is reset. (Note that this differs from an add operation, which resets the carry if no overflow occurs.)

Operation: RA ← RA-M

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

If register H contains X'13' and register L contains X'8B', memory location X'138B' contains X'8A' and register A contains X'0C', the instruction

S RA

will cause X'8A' to be two's complemented (=X'76') and added to X'0C'. That is,

$$\begin{array}{rcl} \text{Register A} = & 0000\ 1100 & = \text{X'0C'} \\ \text{-Memory} = \text{-(X'8A')} = & \underline{0111\ 0110} & = \text{X'76'} \\ & 1000\ 0010 & = \text{X'82'} \end{array}$$

This operation also resets the Carry bit, indicating that the result is positive!

5.6.17 SC Subtract Carry

Assembler Format: SC RA

1	0	0	1	1	1	1	0

Carry is added to the contents of the memory location addressed by registers H and L. This sum is then subtracted from register A using two's complement arithmetic.

The SC instruction is useful when performing subtractions. It adjusts the result of subtracting two bytes when a previous subtraction has produced a negative result (a borrow).

Operation: RA ← RA-(M+Carry)

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

If register H contains X'13' and register L contains X'8B', memory location X'138B' will be addressed. So, if memory location X'138B' contains X'02', register A contains X'04', and Carry=1, the instruction

SC RA

will act as follows:

X'02' + Carry = X'03'

Two's Complement of X'03' = 1111 1101

Adding this to register A produces:

```
Register A = 0000 0100 = X'04'
             1111 1101
             ---------
   ┌──────1] 0000 0001 = X'01'
   │
   └──────►carry out=1, causing the Carry bit to be reset
```

The final value in register A is X'01', causing the Zero bit to be reset. The Carry bit is reset since this is a subtract operation and there was a carry out of the high-order bit position. *The Auxiliary Carry bit is set since there was a carry out of bit* A_3. The Parity and the Sign bits are reset.

5.6.18 SCI Subtract Carry Immediate

Assembler Format: SCI RA,mm

1	1	0	1	1	1	1	0	mm

The Carry bit is internally added to the byte of immediate data. This sum is then subtracted from register A using two's complement arithmetic.

This instruction and the SC and SRC instructions are most useful when performing multibyte subtractions.

Since this is a subtraction operation, the Carry bit is set if there is no carry out of the high-order position, and reset if a carry out occurs.

Operation: RA ← RA-(mm+Carry)

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

Consider the instruction sequence

```
XR   RA,RA
SCI  RA,1
```

The XR instruction will zero register A. If the Carry bit is zero, the SCI instruction will then perform the following operation:

mm + Carry = X'01'

-X'01' = 1111 1111

Adding this to register A produces:

```
Register A = 0000 0000
             1111 1111
             ---------
        ┌────1111 1111 = X'FF' = -X'01'
        │
        └───►Carry out=0, setting the Carry bit
```

The Carry bit is set, indicating a borrow. The Zero and Auxiliary Carry bits are reset, while the Sign and Parity bits are set.

5.6.19 SI Subtract Immediate

Assembler Format: SI RA,mm

1	1	0	1	0	1	1	0	mm

The byte of immediate data is subtracted from the contents of register A using two's complement arithmetic.

Since this is a subtraction operation, the Carry bit is set, indicating a borrow, if there is no carry out of the high-order bit position, and reset if there is a carry out.

Operation: RA ← RA-mm

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

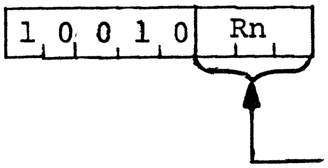
Example:

This instruction can be used as the register A equivalent of the DEC (Decrement Register) instruction by coding

```
SI   RA,1
```

5.6.20 SR Subtract Register

Assembler Format: SR RA,Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The specified register is subtracted from register A using two's complement arithmetic.

If there is no carry out of the high-order bit position, indicating that a borrow occurred, the Carry bit is set; otherwise it is reset. (Note that this differs from an add operation, which resets the carry if no overflow occurs).

Operation: RA ← RA-Rn

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

Assume that register A contains X'3E'. Then the instruction:

```
SR  RA,RA
```

will subtract the accumulator from itself producing a result of zero as follows:

```
      X'3E' = 0011 1110
  +(-X'3E') = 1100 0001 negate and add one to produce
                      1 two's complement
            ___________
   Carry → 1] 0000 0000 = Result = 0
```

Since there was a carry out of the high-order bit position, and this is a subtraction operation, the Carry bit will be reset.

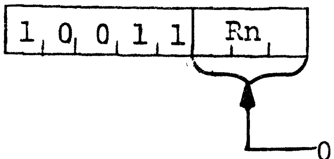
Since there was a carry out of bit A_3, the Auxiliary Carry bit will be set.

The Parity and Zero bits will also be set, and the Sign bit will be reset.

Thus the SR RA,RA instruction can be used to reset the Carry bit (and clear register A).

5.6.21 SRC Subtract Register Carry

Assembler Format: SRC RA,Rn (where Rn = RA, RB, RC, RD, RE, RH, or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
111 for register A

The Carry bit is internally added to the contents of the specified register. This value is then subtracted from register A using two's complement arithmetic.

This instruction is most useful when performing subtractions. It adjusts the result of subtracting two bytes when a previous subtraction has produced a negative result (a borrow).

Operation: RA ← RA-(Rn+Carry)

Condition bits affected: Carry, Sign, Zero, Parity, *Auxiliary Carry*

Example:

> Assume that register L contains X'02', register A contains X'04', and the Carry bit=1. Then the instruction
>
> ```
> SRC   RA,RL
> ```
>
> will act as follows:
>
> X'02' + Carry = X'03'
>
> Two's Complement of X'03' = 1111 1101
>
> Adding this to register A produces:
>
> ```
> Register A = 0000 0100 = X'04'
>              1111 1101 = -X'03'
>            1 0000 0001 = X'01'
>
>        carry out=1, causing the Carry bit to be reset
> ```
>
> The final value in register A is X'01', causing the Zero bit to be reset. The Carry bit is reset since this is a subtract operation and there was a carry out of the high-order bit position. The Auxiliary Carry bit is set since there was a carry out of bit A_3. The Parity and the Sign bits are reset.

5.6.22 X Exclusive OR

Assembler Format: X RA

1	0	1	0	1	1	1	0

The contents of the memory location addressed by registers H and L is Exclusive-ORed, bit by bit, with the contents of register A. The Carry bit is reset to zero.

The Exclusive-OR of the two bits produces 1 if and only if the values of the bits are different.

Operation: $RA \leftarrow RA \veebar M$, $Carry \leftarrow 0$

Condition bits affected: Carry, Zero, Sign, Parity

Example:

> Any bit Exclusive-ORed with a one is complemented. Therefore, if register H contains X'13' and register L contains X'8B', memory location X'138B' will be addressed. So, if memory location X'138B' contains X'FF' and register A contains X'0C', the instruction
>
> ```
> X   RA
> ```
>
> will *one's* complement register A (X'F3'). Further, the sequence
>
> ```
> X   RA
> AI  RA,1
> ```
>
> will *two's* complement register A (X'F4').

5.6.23 XI Exclusive OR Immediate

Assembler Format: XI RA,mm

1	1	1	0	1	1	1	0	mm

The byte of immediate data is Exclusive-ORed with the contents of register A. The Carry bit is set to zero.

Operation: $RA \leftarrow RA \veebar mm$, $Carry \leftarrow 0$

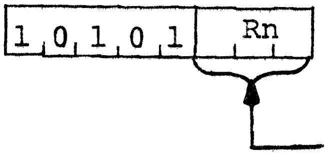
Condition bits affected: Carry, Zero, Sign, Parity

Example:

> Since any bit Exclusive-ORed with a one is complemented and any bit Exclusive-ORed with a zero is unchanged, this instruction can be used to complement specific bits of register A. For example, the instruction
>
> ```
> XI   RA,X'81'
> ```
>
> will complement bits 0 and 7 of register A, leaving bits 1-6 unchanged.

5.6.24 XR Exclusive OR Register

Assembler Format: XR RA,Rn (where Rn = RA, RB, RC, RD, RE, RH,
or RL)



000 for register B
001 for register C
010 for register D
011 for register E
100 for register H
101 for register L
110 for register M
111 for register A

The specified register is Exclusive-ORed, bit by bit, with the contents of register A. The Carry bit is reset to zero.

The Exclusive-OR function of two bits equals 1 if and only if the values are different.

Operation: RA ← RA $\veebar$ Rn, Carry ← 0

Condition bits affected: Carry, Zero, Sign, Parity

Example:

Since any bits Exclusive-ORed with itself produces zero, the instruction

```
XR  RA,RA
```

will clear register A.

5.7 ROTATE ACCUMULATOR INSTRUCTIONS

This section describes the instructions which rotate the contents of the Accumulator. No memory locations or other registers are affected.

5.7.1 RLC Rotate Left Carry

Assembler Format: RLC RA

0	0	0	1	0	1	1	1

The contents of register A are rotated one bit position to the left. The high-order bit of the register replaces the Carry bit, while the Carry bit replaces the low-order bit of the register.

Operation: Carry ← RA_7,RA_{1-7} ← RA_{0-6} and RA_0 ← Carry

Condition bits affected: Carry

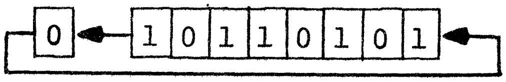
Example:

Assume that register A contains X'B5'. Then the instruction

RLC RA

acts as follows.

Before RLC is executed: Carry Register A



After RLC is executed:

1	0 1 1 0 1 0 1 0
Carry=1	RA=X'6A'

5.7.2 ROL Rotate Left

Assembler Format: ROL RA

0	0	0	0	0	1	1	1

The Carry bit is set equal to the high-order bit of register A. The contents of register A are rotated one bit position to the left, with the high-order bit being transferred to the low-order bit position of the register.

Operation: Carry ← RA_7,RA_{1-7} ← RA_{0-6} and RA_0 ← RA_7

Condition bits affected: Carry

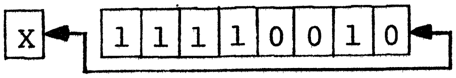
Example:

Assume that register A contains X'F2'. Then the instruction

ROL RA

acts as follows.

Before ROL is executed: Carry Register A



After ROL is executed:

1		1	1	1	0	0	1	0	1

Carry=1 RA=X'E5'

5.7.3 ROR Rotate Right

Assembler Format: ROR RA

0	0	0	0	1	1	1	1

The Carry bit is set equal to the low-order bit of register A. The contents of register A are rotated one bit position to the right, with the low-order bit being transferred to the high-order bit position of the register.

Operation: Carry ← RA_0, RA_{0-6} ← RA_{1-7} and RA_7 ← RA_0

Condition bits affected: Carry

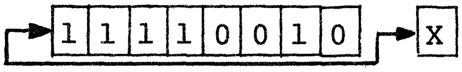
Example:

Assume that register A contains X'F2'. Then the instruction

ROR RA

acts as follows.

Before ROR is executed: Register A Carry



1	1	1	1	0	0	1	0	X

After ROR is executed:

0	1	1	1	1	0	0	1	0

RA=X'79' Carry=0

5.7.4 RRC Rotate Right Carry

Assembler Format: RRC RA

0	0	0	1	1	1	1	1

The contents of register A are rotated one bit position to the right.

The low-order bit of the register replaces the Carry bit, while the Carry bit replaces the high-order bit of the register.

Operation: Carry $\leftarrow RA_0, RA_{0-6} \leftarrow RA_{1-7}$ and $RA_7 \leftarrow RA_0$

Condition bits affected: Carry

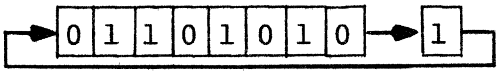
Example:

Assume that register A contains X'6A'. Then the instruction

RRC RA

acts as follows.

Before RRC is executed: Register A Carry



After RRC is executed:

1	0	1	1	0	1	0	1		0

RA=X'B5' Carry=0

5.8 REGISTER PAIR INSTRUCTIONS

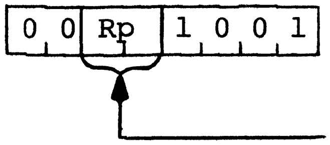
This section describes instructions that operate on pairs of registers.

In addition to these instructions, several of the Basic System Subroutines in Appendix B can be used to perform some useful register pair operations. Specifically:

- ADD2 (Section B.3.1) adds register C to register pair D and E
- COMPER (Section B.3.3) compares register pair B and C with register pair D and E
- SUBREG (Section B.3.16) subtracts register pair B and C from register pair D and E
- SUBT2 (Section B.3.17) subtracts register C from register pair D and E.

5.8.1 DAD Double Add

Assembler Format: DAD Rp (where Rp = RB, RD, RH, or SP)



00 for registers B and C
01 for registers D and E
10 for registers H and L
11 for register SP

The 16-bit number in the specified register pair is added to the 16-bit number held in the H and L registers using two's complement arithmetic. The result replaces the contents of the H and L registers.

Operation: H,L ← (H,L)+Rp

Condition bits affected: Carry

Example 1:

Assume that register B contains X'33', register C contains X'9F', register H contains X'A1', and register L contains X'7B'. Then the instruction

```
DAD  RB
```

performs the following addition:

```
      Registers B and C = X'339F'
    + Registers H and L = X'A17B'
 New contents of H and L = X'D51A'
```

Register H now contains X'D5' and register L now contains X'1A'. Since no carry out was produced, the Carry bit is reset=0.

Example 2:

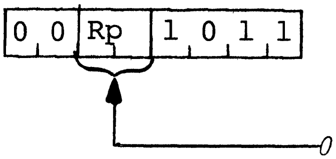
The instruction

```
DAD  RH
```

will double the 16-bit number in the H and L registers (which is equivalent to shifting the 16 bits one position to the left).

5.8.2 DECP Decrement Register Pair

Assembler Format: DECP Rp (where Rp = RB, RD, RH, or SP)



00 for registers B and C
01 for registers D and E
10 for registers H and L
11 for register SP

The 16-bit number held in the specified register pair is decremented by one.

NOTE

In the Zentec 9002, the H and L register pair can be decremented by one by calling a basic system subroutine labelled DECHL.

Operation: Rp ← Rp-1

Condition bits affected: None

Example:

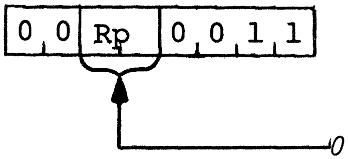
If register H contains X'98' and register L contains X'00', the instruction

DECP RH

will cause register H to contain X'97' and register L to contain X'FF'.

5.8.3 INCP Increment Register Pair

Assembler Format: INCP Rp (where Rp = RB, RD, RH, or SP)



00 for registers B and C
01 for registers D and E
10 for registers H and L
11 for register SP

The 16-bit number held in the specified register pair is incremented by one.

NOTE

In the Zentec 9002, the H and L register pair can be incremented by one by calling a basic system subroutine labelled BUMPHL.

Operation: Rp ← Rp+1

Condition bits affected: None

Example:

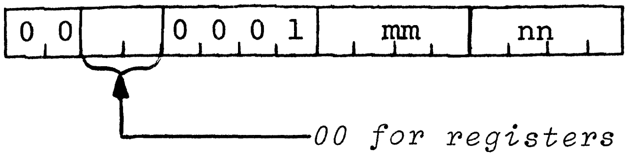
If registers D and E contain X'38' and X'FF' respectively, the instruction

INCP RD

will cause register D to contain X'39' and register E to contain X'00'.

5.8.4 LHI Load Half-Word Immediate

Assembler Format: LHI Rp, nn,mm *(where Rp = RB, RD, RH, or SP)*



00 for registers B and C
01 for registers D and E
10 for registers H and L
11 for register SP

The third byte of the instruction (nn) is loaded into the first register of the specified pair, while the second byte of the instruction is loaded into the second register of the specified pair. If SP is specified as the register pair, the second byte of the instruction replaces the least significant 8 bits of the stack pointer, while the third byte of the instruction replaces the most significant 8 bits of the stack pointer.

Operation: Rp ← mmnn

Condition bits affected: None

NOTE

> *The immediate data for this instruction is a 16-bit quantity. All other immediate instructions require an 8-bit data value.*

Example:

Assume that instruction label STRT refers to memory location X'2103'. Then the following instructions will each load the H register with X'21' and the L register with X'03':

```
LHI  RH,X'2103'
LHI  RH,STRT
```

9002 LHI Macro

In order for the 9002 user to effectively use relocatable addressing, a predefined macro is available which allows the addressing of a relocatable, external or absolute location.

Assembler Format: LHI Rp,mmnn (where Rp=RA,RB,RC,RD, or RH)

The operation is similar to that of the LHI instruction for the 9003. The first register of the pair is loaded with the most significant byte (mm); the second register is loaded with the least significant byte (nn).

Operation: Rp ← mmnn

Condition bits affected: None

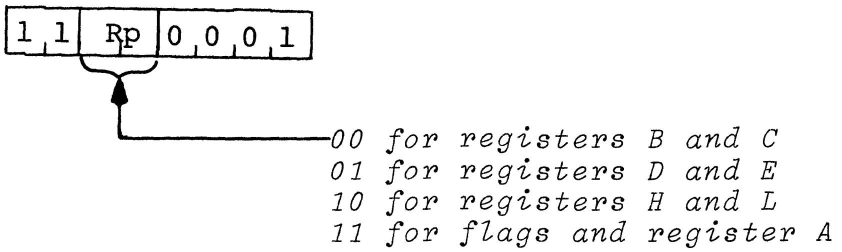
Example:

Assume that LABEL refers to location X'1F8B'. The following will then load the C registers with X'1F' and the D register with X'8B':

```
LHI  RC,LABEL
```

5.8.5 POP Pop Data Off Stack

Assembler Format: POP Rp (where Rp = RB, RD, RH, or PSW)



The contents of the specified register pair are restored from two bytes of memory indicated by the stack pointer SP. The byte of data at the memory address indicated by the stack pointer is loaded into the second register of the register pair; the byte of data at the address one greater than the address indicated by the stack pointer is loaded into the first register of the pair. If register pair PSW is specified, the byte of data indicated by the contents of the stack pointer plus one is used to restore the values of the five condition bits (Carry, Zero, Sign, Parity, and Auxiliary Carry) using the format described in the last section.

In any case, after the data has been restored, the stack pointer is incremented by two.

Operation: (RP1) ← (SP+1), (RP2) ← SP, SP ← (SP+2)

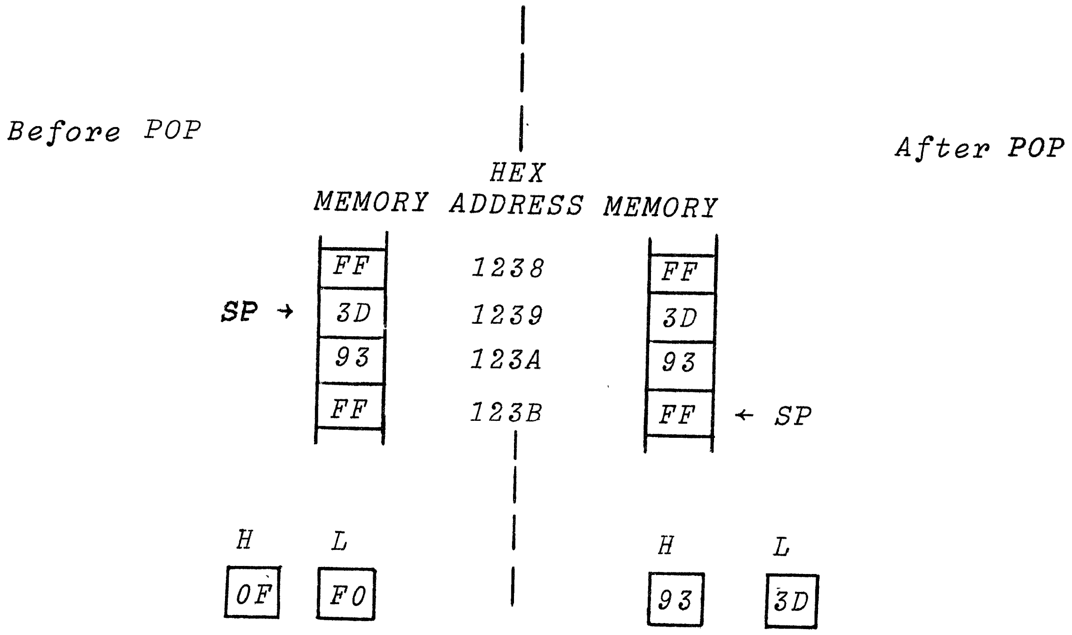
Condition bits affected: If register pair PSW is specified, Carry, Sign, Zero, Parity, and Auxiliary Carry may be changed. Otherwise, none are affected.

Example 1:

Assume that memory locations X'1239' and X'123A' contain X'3D' and X'93', respectively, and that the stack pointer contains X'1239'. Then the instruction

```
POP  RH
```

loads register L with the value X'3D' from location X'1239', loads register H with the value X'93' from location X'123A', and increments the stack pointer by two, leaving it equal to X'123B'.

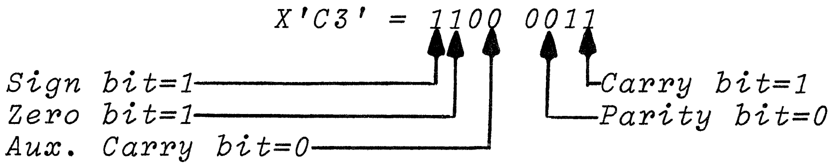


Example 2:

Assume that memory locations X'2C00' and X'2C01' contain X'C3' and X'FF', respectively, and that the stack pointer contains X'2C00'. Then the instruction

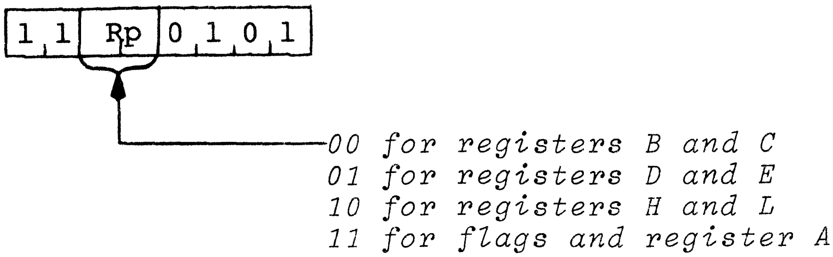
```
POP  PSW
```

will load the Accumulator with X'FF' and set the condition bit as follows:



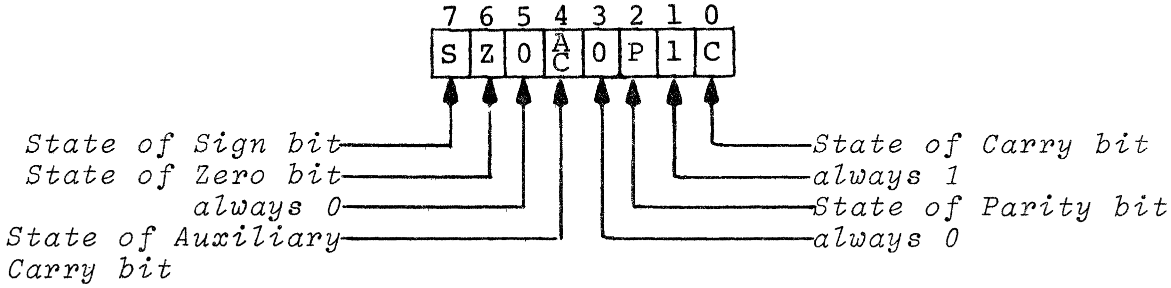
5.8.6 PUSH Push Data Onto Stack

Assembler Format: PUSH Rp (where Rp = RB, RD, RH, or PSW)



The contents of the specified register pair are saved in two bytes of memory indicated by the stack pointer SP.

The contents of the first register are saved at the memory address one less than the address indicated by the stack pointer; the contents of the second register are saved at the address two less than the address indicated by the stack pointer. If register pair PSW is specified, the first byte of information saved holds the contents of the A register; the second byte holds the settings of the five condition bits, i.e., Carry, Zero, Sign, Parity, and Auxiliary Carry. The format of this byte is:



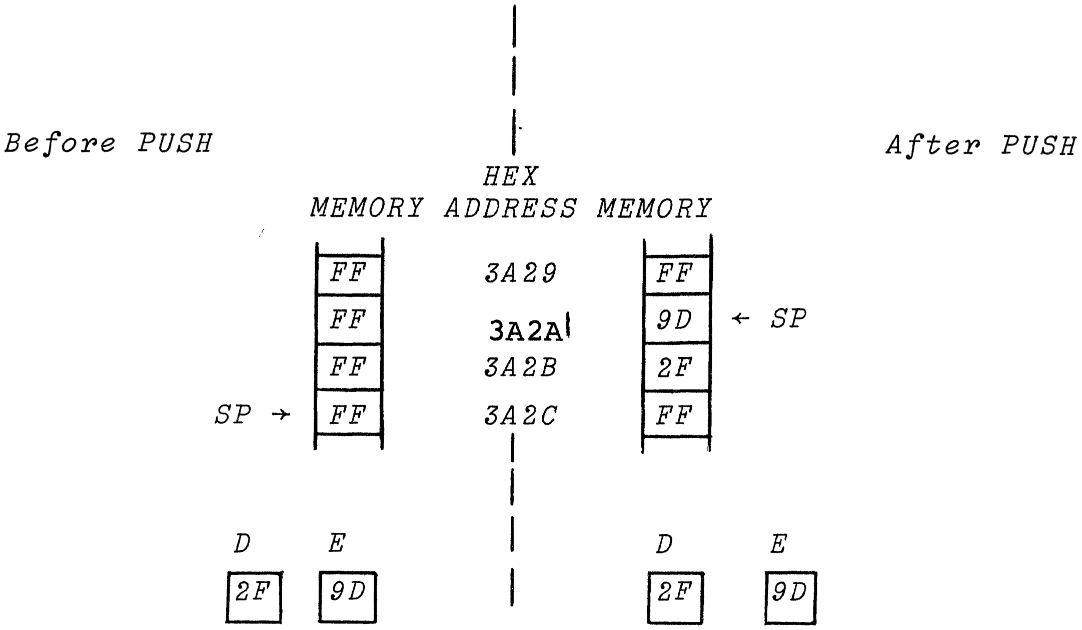
In any case, after the data has been saved, the stack pointer is decremented by two.

Operation: (SP-1) ← (RP1),(SP-2) ← (RP2), SP ← (SP-2)

Condition bits affected: None

Example 1:

> *Assume that register D contains X'2F', register E contains X'9D', and the stack pointer contains X'3A2C'. Then the instruction*
>
> *PUSH RD*
>
> *stores the D register at memory address X'3A2B', stores the E register at memory address X'3A2A', and then decrements the stack pointer by two, leaving the stack pointer equal to X'3A2A'.*



Example 2:

Assume that the Accumulator contains X'1F', the stack pointer contains X'302A', the Carry, Zero and Parity bits all equal 1, and the Sign and Auxiliary Carry bits all equal 0. Then the instruction

PUSH PSW

stores the Accumulator (X'1F') at location X'3029', stores the value X'47', corresponding to the flag settings, at location X'3028', and decrements the stack pointer to the value X'3028'.

5.8.7 SPHL Load SP From H and L

Assembler Format: SPHL

1	1	1	1	1	0	0	1

The 16 bits of data held in the H and L registers replace the contents of the stack pointer SP. The contents of the H and L registers are unchanged.

Operation: SP ← H,L

Condition bits affected: None

Example:

> *If registers H and L contain X'20' and X'6C', respectively, the instruction SPHL will load the stack pointer with the value X'206C'.*

5.8.8 XCHG Exchange Registers

Assembler Format: *XCHG*

1	1	1	0	1	0	1	1

The 16 bits of data held in the H and L registers are exchanged with the 16 bits of data held in the D and E registers.

Operation: *D,E ↔ H,L*

Condition bits affected: *None*

Example:

> *If register H contains X'00', register L contains X'FF', register D contains X'33', and register E contains X'55', the instruction XCHG will perform the following operation:*

Before XCHG			*After XCHG*	
D	*E*		*D*	*E*
33	*55*		*00*	*FF*
H	*L*		*H*	*L*
00	*FF*		*33*	*55*

5.8.9 XTHL Exchange Stack

Assembler Format: *XTHL*

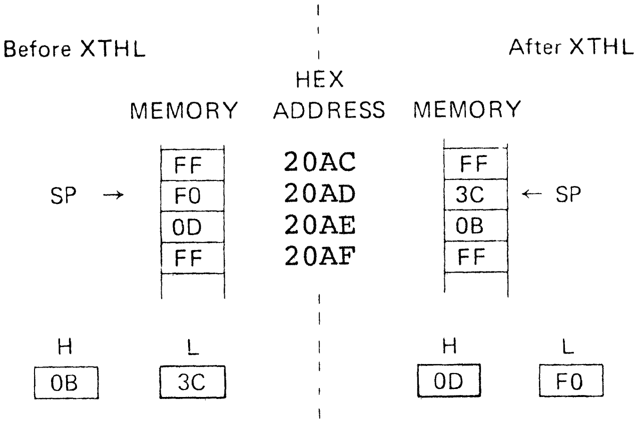
1	1	1	0	0	0	1	1

The contents of the L register are exchanged with the contents of the memory byte whose address is held in the stack pointer SP. The contents of the H register are exchanged with the contents of the memory byte whose address is one greater than that held in the stack pointer.

Operation: *L ↔ (SP), H ↔ (SP+1)*

Condition bits affected: *None*

Example: *If register H contains X'0B', register L contains X'3C', register SP contains X'20AD', and memory locations X'20AD' and X'20AE' contain X'F0' and X'0D', respectively, the XTHL instruction will perform the following operation:*



5.9 DIRECT ADDRESSING INSTRUCTIONS

This section describes instructions which reference memory by a two-byte address that is part of the instruction itself. Instructions in this class occupy three bytes, as follows:

code	low addr	hi addr

where "low addr' is the least-significant byte of a memory address and "hi addr" is the most-significant byte of a memory address. The address may be absolute, relocatable, or external.

5.9.1 LAD Load Accumulator Direct

Assembler Format: LAD addr

0 0 1 1 1 0 1 0	low addr	hi addr

The byte at the memory address formed by concatenating "hi addr" with "low addr" replaces the contents of the Accumulator.

Operation: RA ← M

Condition bits affected: None

Example:

The instruction

```
LAD  X'211C'
```

will replace the Accumulator contents with the contents of memory location X'211C'.

5.9.2 LHLD Load H And L Direct

Assembler Format: LHLD addr

0 0 1 0 1 0 1 0	low addr	hi addr

The byte at the memory address formed by concatenating "hi addr" with "low addr" replaces the contents of register L. The byte at the next higher memory address replaces the contents of register H.

Operation: RL ← M and RH ← M+1

Condition bits affected: None

Example:

If memory locations X'2113' and X'2114' contain X'EF' and X'22', respectively, the instruction

LHLD X'2113'

will load register L with X'EF' and register H with X'22'.

5.9.3 SHLD Store H And L Direct

Assembler Format: SHLD addr

0 0 1 0 0 0 1 0	low addr	hi addr

The contents of register L are stored at the memory address formed by concatenating "hi addr" with "low addr". The contents of register H are stored at the next higher memory address.

Operation: M ← RL and M+1 ← RH

Example:

If the H and L registers contain X'21' and X'BC', respectively, the instruction

SHLD X'24CE'

will store X'BC' into location X'24CE' and X'21' into location X'24CF'.

5.9.4 STD Store Accumulator Direct

Assembler Format: STD addr

0 0 1 1 0 0 1 0	low addr	hi addr

The contents of the Accumulator replace the byte at the memory address formed by concatenating "hi addr" with "low addr".

Operations: M ← RA

Condition bits affected: None

Example:

The instruction

STD X'241C'

will store the contents of the Accumulator at memory address X'241C'.

5.10 BRANCH INSTRUCTIONS

This instruction group causes the normal execution sequence of instructions to be altered. With the exception of PCHL (Section 5.10.21), instructions in this class occupy three bytes, as follows:

code	low addr	hi addr

where low addr = least-significant byte of a memory address
      hi addr = most-significant byte of a memory address

The address may be absolute, relocatable or external.

5.10.1 B Absolute Branch

Assembler Format: B label

1 1 0 0 0 0 1 1	low addr	hi addr

Program execution branches unconditionally to the memory address of "label".

Operation: Prog. Counter ← label

Condition bits affected: None

Example 1:

The instruction

```
B  START
```

will cause program control to transfer to the instruction that has the label START.

Example 2:

The instruction

```
B  X'0113'
```

will cause program control to transfer to the instruction located at memory location X'0113'.

5.10.2 BE Branch On Equal

Assembler Format: BE label

1 1 0 0 1 0 1 0	low addr	hi addr

Program execution will branch to "label" if the preceding compare operation (C, CI or CR) was successful; that is, if the Zero bit is set.

Operation: Prog. Counter ← label if Zero=1

Condition bits affected: None

Example:

The instruction sequence

```
              CI   RA,3    Does A equal 3?
              BE   EQUAL   Yes, go to EQUAL
              LR   RB,RA   No, load A into B
              B    CONT    Go to CONT
       EQUAL  LR   RC,RA   Load A into C
       CONT   LR   RH,RD   Load D into H
```

checks if register A contains 3. If it does, it is loaded into register C; if not, it is loaded into register B.

5.10.3 BFC Branch False Carry

Assembler Format: BFC label

1 1 0 1 0 0 1 0	low addr	hi addr

Program execution will branch to "label" if Carry=0

Operation: Prog. Counter ← label if Carry=0

Condition bits affected: None

5.10.4 BFP Branch False Parity

Assembler Format: BFP label

1 1 1 0 0 0 1 0	low addr	hi addr

Program execution will branch to "label" if Accumulator parity is odd; that is, if the Parity bit is reset.

Operation: Prog. Counter ← label if Parity=0

Condition bits affected: None

5.10.5 BFS Branch False Sign

Assembler Format: BFS label

1 1 1 1 0 0 1 0	low addr	hi addr

Program execution will branch to "label" if the Accumulator contains a non-zero positive number; that is, if Sign=0 and Zero=0.

Operation: Prog. Counter ← label if Sign=0 and Zero=0

Condition bits affected: None

Example:

The instruction sequence

```
        SI   RA,X'42'  Subtract X'42' from A
        BFS  MORE      Branch on A GT X'42'
        SR   RA,RA     Zero A if LT or EQ X'42'
  MORE  AR   RA,RB     Add B to A
```

checks to see if the value in register A is greater than X'42'. If it is, this difference is added to register B; if not, register A is zeroed and register B is, in effect, loaded into register A.

5.10.6 BFZ Branch False Zero

Assembler Format: BFZ label

1 1 0 0 0 0 1 0	low addr	hi addr

Program execution will branch to "label" if the Accumulator is not zero or a compare operation (C, CI or CR) was not equal.

Operation: Prog. Counter ← label if Zero=0

Condition bits affected: None

5.10.7 BH Branch On High Or Equal

Assembler Format: BH label

1 1 0 1 0 0 1 0	low addr	hi addr

Program execution will branch to "label" if the preceding compare operation (C, CI or CR) detected that the Accumulator was equal to or higher than the comparand.

Operation: Prog. Counter ← label if Carry=0

Condition bits affected: None

Example:

The instruction sequence

```
          CI    RA,6    A GT or EQ to 6?
          BHE   HIGH    Yes, branch to HIGH
          SR    RA,RA   No, clear A
          B     ADDB    Go add B
    HIGH  SI    RA,6    Sub 6 from A
    ADDB  AR    RA,RB   Add A and B
```

checks to see if the Accumulator is greater than or equal to 6. If it is, the difference is added to register B; if not, the Accumulator is cleared before adding.

5.10.8 BL Branch On Low

Assembler Format: BL label

1 1 0 1 1 0 1 0	low addr	hi addr

Program execution will branch to "label" if the preceding compare operation (C, CI or CR) detected that the Accumulator was less than the comparand.

Operation: Prog. Counter ← label if Carry=1

Condition bits affected: None

5.10.9 BM Branch On Minus

Assembler Format: BM label

1 1 1 1 1 0 1 0	low addr	hi addr

Program execution will branch to "label" if the Accumulator contains a negative number; that is, if Sign=1.

Operation: Prog. Counter ← label if Sign=1

Condition bits affected: None

5.10.10 BNE Branch On Not Equal

Assembler Format: BNE label

1	1	0	0	0	0	1	0	low addr	hi addr

Program execution will branch to "label" if the preceding compare operation (C, CI or CR) was unsuccessful; that is, if the Zero bit is reset.

Operation: Prog. Counter ← label if Zero=0

Condition bits affected: None

5.10.11 BNM Branch On Not Minus

Assembler Format: BNM label

1	1	1	1	0	0	1	0	low addr	hi addr

Program execution will branch to "label" if the Accumulator contains a positive number or zero.

Operation: Prog. Counter ← label if Sign=0

Condition bits affected: None

5.10.12 BNP Branch On Not Plus

Assembler Format: BNP label

1	1	1	1	1	0	1	0	low addr	hi addr

Program execution will branch to "label" if the Accumulator contains a negative number.

Operation: Prog. Counter ← label if Sign=1

Condition bits affected: None

5.10.13 BNZ Branch On Not Zero

Assembler Format: BNZ label

1	1	0	0	0	0	1	0	low addr	hi addr

Program execution will branch to "label" if the Accumulator is not zero.

Operation: Prog. Counter ← label if Zero=0

Condition bits affected: None

5.10.14 BP Branch On Plus

Assembler Format: BP label

1	1	1	1	0	0	1	0	low addr	hi addr

Program execution will branch to "label" if the Accumulator is positive or zero.

Operation: Prog. Counter ← label if Sign=0

Condition bits affected: None

5.10.15 BTC Branch True Carry

Assembler Format: BTC label

1	1	0	1	1	0	1	0	low addr	hi addr

Program execution will branch to "label" if the Carry bit is on.

Operation: Prog. Counter ← label if Carry=1

Condition bits affected: None

5.10.16 BTP Branch True Parity

Assembler Format: BTP label

1 1 1 0 1 0 1 0	low addr	hi addr

Program execution will branch to "label" if Accumulator parity is even; that is, if the Parity bit is set.

Operation: Prog. Counter ← label if Parity=1

Condition bits affected: None

5.10.17 BTS Branch True Sign

Assembler Format: BTS label

1 1 1 1 1 0 1 0	low addr	hi addr

Program execution will branch to "label" if the Accumulator sign is negative.

Operation: Prog. Counter ← label if Sign=1

Condition bits affected: None

5.10.18 BTZ Branch True Zero

Assembler Format: BTZ label

1 1 0 0 1 0 1 0	low addr	hi addr

Program execution will branch to "label" if the Accumulator contains zero or a compare operation (C, CI or CR) showed equality.

Operation: Prog. Counter ← label if Zero=1

Condition bits affected: None

5.10.19 BZ Branch On Zero

Assembler Format: BZ label

1 1 0 0 1 0 1 0	low addr	hi addr

Program execution will branch to "label" if the Accumulator is zero.

Operation: Prog. Counter ← label if Zero=1

Condition bits affected: None

5.10.20 PCHL Load Program Counter

Assembler Format: PCHL

1 1 1 0 1 0 0 1

The contents of the H register replace the most significant 8 bits of the program counter, and the contents of the L register replace the least significant 8 bits of the program counter. This causes program execution to continue at the address contained in the H and L registers.

Operation: PC ← H,L

Condition bits affected: None

Example:

> *If the H register contains X'21' and the L register contains X'3E', the instruction:*
>
> *PCHL*
>
> *will cause program execution to continue with the instruction at memory address X'213E'.*

5.11 CALL SUBROUTINE INSTRUCTIONS

This section describes the instructions that call subroutines. These instructions operate like the Branch instructions, causing a transfer of program control, but they also push a return address onto the stack for use by the Return instructions (Section 5.11).

Instructions in this class occupy three bytes, as follows:

code	low addr	hi addr

where low addr = least-significant byte of a memory address
hi addr = most-significant byte of a memory address

5.11.1 CALL Absolute Call

Assembler Format: CALL sub

1 1 0 0 1 1 0 1	low addr	hi addr

A call operation is unconditionally performed to subroutine "sub".

Operation: (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

Example:

The instruction

```
CALL SUBR
```

transfers control to subroutine SUBR.

5.11.2 CE Call On Equal

Assembler Format: CE sub

1 1 0 0 1 1 0 0	low addr	hi addr

Call "sub" if the preceding compare operation (C, CI or CR) was successful; that is, if the Zero bit is set.

Operation: If Zero=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.3 CFC Call False Carry

Assembler Format: CFC sub

1 1 0 1 0 1 0 0	low addr	hi addr

Call "sub" if Carry=0

Operation: If Carry=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.4 CFP Call False Parity

Assembler Format: CFP sub

1 1 1 0 0 1 0 0	low addr	hi addr

Call "sub" if Accumulator parity is odd; that is, if the Parity bit is reset.

Operation: If Parity=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.5 CFS Call False Sign

Assembler Format: CFS sub

1 1 1 1 0 1 0 0	low addr	hi addr

Call "sub" if the Accumulator contains a non-zero positive number; that is, if Sign=0 and Zero=0.

Operation: If Sign=0 and Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← add

Condition bits affected: None

5.11.6 CFZ Call False Zero

Assembler Format: CFZ sub

1 1 0 0 0 1 0 0	low addr	hi addr

Call "sub" if the Accumulator is not zero or a compare operation (C, CI or CR) was not equal.

Operation: If Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.7 CH Call On High Or Equal

Assembler Format: CH sub

1 1 0 1 0 1 0 0	low addr	hi addr

Call "sub" if the preceding compare operation (C, CI or CR) detected that the Accumulator was equal to or higher than the comparand.

Operation: If Carry=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.8 CL Call On Low

Assembler Format: CL sub

1 1 0 1 1 1 0 0	low addr	hi addr

Call "sub" if the preceding compare operation (C, CI or CR) detected that the Accumulator was less than the comparand.

Operation: If Carry=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.9 CM Call On Minus

Assembler Format: CM sub

1 1 1 1 1 1 0 0	low addr	hi addr

Call "sub" if the Accumulator contains a negative number; that is, if Sign=1.

Operation: If Sign=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.10 CNE Call On Not Equal

Assembler Format: CNE sub

1 1 0 0 0 1 0 0	low addr	hi addr

Call "sub" if the preceding compare operation (C, CI or CR) was unsuccessful; that is, if the Zero bit is reset.

Operation: If Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.11 CNM Call On Not Minus

Assembler Format: CNM sub

1 1 1 1 0 1 0 0	low addr	hi addr

Call "sub" if the Accumulator contains a positive number or zero.

Operation: If Sign=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.12 CNP Call On Not Plus

Assembler Format: CNP sub

1 1 1 1 1 1 0 0	low addr	hi addr

Call "sub" if the Accumulator contains a negative number.

Operation: If Sign=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.13 CNZ Call On Not Zero

Assembler Format: CNZ sub

1 1 0 0 0 1 0 0	low addr	hi addr

Call "sub" if the Accumulator is not zero.

Operation: If Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.14 CP Call On Plus

Assembler Format: CP sub

1 1 1 1 0 1 0 0	low addr	hi addr

Call "sub" if the Accumulator is positive or zero.

Operation: If Sign=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.15 CTC Call True Carry

Assembler Format: CTC sub

1	1	0	1	1	1	0	0	low addr	hi addr

Call "sub" if the Carry bit is on.

Operation: If Carry=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.16 CTP Call True Parity

Assembler Format: CTP sub

1	1	1	0	1	1	0	0	low addr	hi addr

Call "sub" if Accumulator parity is even; that is, if the Parity bit is set.

Operation: If Parity=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.17 CTS Call True Sign

Assembler Format: CTS sub

1	1	1	1	1	1	0	0	low addr	hi addr

Call "sub" if the Accumulator sign is negative.

Operation: If Sign=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.18 CTZ Call True Zero

Assembler Format: CTZ sub

1 1 0 0 1 1 0 0	low addr	hi addr

Call "sub" if the Accumulator contains zero or a compare operation (C, CI or CR) showed equality.

Operation: If Zero-1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.11.19 CZ Call On Zero

Assembler Format: CZ sub

1 1 0 0 1 1 0 0	low addr	hi addr

Call "sub" if the Accumulator is zero.

Operation: If Zero=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

Condition bits affected: None

5.12 RETURN FROM SUBROUTINE INSTRUCTIONS

This section describes the instructions used to return from subroutines. These instructions pop the last address saved on the stack into the program counter, causing a transfer of program control to that address.

Instructions in this class occupy one byte.

5.12.1 RET Absolute Return

Assembler Format: RET

1	1	0	0	1	0	0	1

A return operation is unconditionally performed.

Operation: Prog. Counter ← SP

Condition bits affected: None

5.12.2 RE Return On Equal

Assembler Format: RE

1	1	0	0	1	0	0	0

Return if the preceding compare operation (C, CI or CR) was successful; that is, if the Zero bit is set.

Operation: Prog. Counter ← SP if Zero=1

Condition bits affected: None

5.12.3 RFC Return False Carry

Assembler Format: RFC

1	1	0	1	0	0	0	0

Return if Carry=0.

Operation: Prog. Counter ← SP if Carry=0

Condition bits affected: None

5.12.4 RFP Return False Parity

Assembler Format: RFP

1	1	1	0	0	0	0	0

Return if Accumulator parity is odd; that is, if the Parity bit is reset.

Operation: Prog. Counter ← SP if Parity=0

Condition bits affected: None

5.12.5 RFS Return False Sign

Assembler Format: RFS

1	1	1	1	0	0	0	0

Return if the Accumulator contains a non-zero positive number; that is, if Sign=0 and Zero=0.

Operation: Prog. Counter ← SP if Sign=0 and Zero=0

Condition bits affected: None

5.12.6 RFZ Return False Zero

Assembler Format: RFZ

1	1	0	0	0	0	0	0

Return if the Accumulator is not zero or a compare operation (C, CI or CR) was not equal.

Operation: Prog. Counter ← SP if Zero=0

Condition bits affected: None

5.12.7 RH Return On High Or Equal

Assembler Format: RH

1	1	0	1	0	0	0	0

Return if the preceding compare operation (C, CI or CR) detected that the Accumulator was equal to or higher than the comparand.

Operation: Prog. Counter ← SP if Carry=0

Condition bits affected: None

5.12.8 RL Return On Low

Assembler Format: RL

1	1	0	1	1	0	0	0

Return if the preceding compare operation (C, CI or CR) detected that the Accumulator was less than the comparand.

Operation: Prog. Counter ← SP if Carry=1

Condition bits affected: None

5.12.9 RM Return On Minus

Assembler Format: RM

1	1	1	1	1	0	0	0

Return if the Accumulator contains a negative number; that is, if Sign=1.

Operation: Prog. Counter ← SP if Sign=1

Condition bits affected: None

5.12.10 RNE Return On Not Equal

Assembler Format: BNE label

1	1	0	0	0	0	0	0

Return if the preceding compare operation (C, CI or CR) was unsuccessful; that is, if the Zero bit is reset.

Operation: Prog. Counter ← SP if Zero=0

Condition bits affected: None

5.12.11 RNM Return On Not Minus

Assembler Format: RNM

1	1	1	1	0	0	0	0

Return if the Accumulator contains a positive number or zero.

Operation: Prog. Counter ← SP if Sign=0

Condition bits affected: None

5.12.12 RNP Return On Not Plus

Assembler Format: RNP

1	1	1	1	1	0	0	0

Return if the Accumulator contains a negative number.

Operation: Prog. Counter ← SP if Sign=1

Condition bits affected: None

5.12.13 RNZ Return On Not Zero

Assembler Format: RNZ

1	1	0	0	0	0	0	0

Return if the Accumulator is not zero.

Operation: Prog. Counter ← SP if Zero=0

Condition bits affected: None

5.12.14 RP Return On Plus

Assembler Format: RP

1	1	1	1	0	0	0	0

Return if the Accumulator is positive or zero.

Operation: Prog. Counter ← SP if Sign=0

Condition bits affected: None

5.12.15 RTC Return True Carry

Assembler Format: RTP

1	1	0	1	1	0	0	0

Return if the Carry bit is on.

Operation: Prog. Counter ← SP if Carry=1

Condition bits affected: None

5.12.16 RTP Return True Parity

Assembler Format: RTP

1	1	1	0	1	0	0	0

Return if Accumulator parity is even; that is, if the Parity bit is set.

Operation: Prog. Counter ← SP if Parity=1

Condition bits affected: None

5.12.17 RTS Return True Sign

Assembler Format: RTS

1	1	1	1	1	0	0	0

Return if the Accumulator sign is negative.

Operation: Prog. Counter ← SP if Sign=1

Condition bits affected: None

5.12.18 RTZ Return True Zero

Assembler Format: RTZ

1	1	0	0	1	0	0	0

Return if the Accumulator contains zero or a compare operation (C, CI or CR) showed equality.

Operation: Prog. Counter ← SP if Zero=1

Condition bits affected: None

5.12.19 RZ Return On Zero

Assembler Format: RZ

1 1 0 0 1 0 0 0

Return if the Accumulator is zero.

Operation: Prog. Counter ← SP if Zero=1

Condition bits affected: None

5.13 INTERRUPT ENABLE/DISABLE INSTRUCTIONS

This section describes the instructions that enable and disable interrupts.

5.13.1 EI Enable Interrupts

Assembler Format: EI

1	1	1	1	1	0	1	1

This instruction enables the CPU to recognize and respond to interrupts.

Operation: (INTE) ← 1

Condition bits affected: None

5.13.2 DI Disable Interrupts

Assembler Format: DI

1	1	1	1	0	0	1	1

This instruction causes the CPU to ignore all interrupts.

Operation: (INTE) ← 0

Condition bits affected: None

5.14 INPUT/OUTPUT INSTRUCTIONS (9003)

This section describes the instructions that cause a byte of information to be input or output from the 9003. Instructions in this class occupy two bytes as follows:

code	device no.

The device number is a hardware characteristic of the input or output device, not under the programmer's control. Section 7 discusses Input/Output programming.

5.14.1 IN Input

Assembler Format: IN dev

1	1	0	1	1	0	1	1	dev

An eight-bit data byte is read from input device number dev and replaces the contents of the Accumulator.

Operation: RA ← input device

Condition bits affected: None

Example:

The instruction

IN 1

causes one byte to be input from device #1 into the Accumulator.

5.14.2 OUT Output

Assembler Format: OUT dev

1	1	0	1	0	0	1	1	dev

The contents of the Accumulator are sent to output device number dev.

Operation: RA → output device

Condition bits affected: None

Example:

The instruction

```
OUT 10
```

sends the contents of the Accumulator to device number 10, as does

```
OUT X'A'
```

5.15 INPUT/OUTPUT INSTRUCTIONS (9002)

This section describes the instructions that cause a byte of information to be input or output from the 9002. Instructions in this class occupy one byte.

The device number is a hardware characteristic of the input or output device, not under the programmer's control. Section 7 provides additional information on Input/Ouptut programming.

5.15.1 IN Input

Assembler Format: IN dev

0	1	0	0	dev			

The instruction code will be between X'41' and X'4F'.

The contents of the Accumulator and the one byte dev code are concatenated internally by the hardware to address the input device. The Accumulator contents is then replaced by the data byte from the input device.

Operation: RA ← input device

Condition bits affected: None

Example:

The instructions:

```
LBI  RA,X'00'
IN   X'0F'
```

cause one byte to be input from the RS-232 communications.

5.15.2 OUT Output

Assembler Format: OUT dev

0	1	dev					

The instruction code will be between X'51' and X'7F'.

The contents of the Accumulator are sent to output device number dev.

Operation: RA → output device

Condition bits affected: None

Example:

The instruction

```
OUT  X'1D'
```

sends the contents of the Accumulator out to the RS-232 communications.

SECTION 6

SUBROUTINE PROGRAMMING

Frequently, a certain operation must be performed many times in a program. Rather than recoding the instruction sequence each time, it is more efficient to code it once in a subroutine.

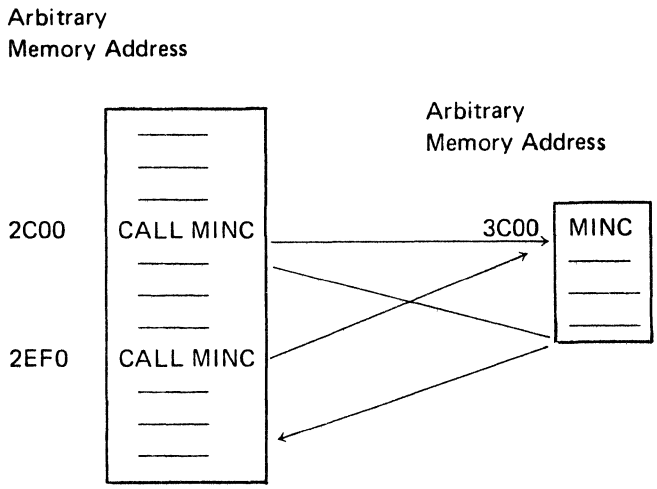
6.1 BASIC OPERATION OF A SUBROUTINE

A subroutine is coded like any other group of assembly language statements, and is referred to by its name, which is the label of the first instruction. The programmer references a subroutine by writing its name in the operand field of a Call instruction. When the Call is executed, the address of the next sequential instruction after the Call is pushed onto the stack, and program execution proceeds with the first instruction of the subroutine. When the subroutine has completed its work, a Return instruction is executed, which causes the top address in the stack to be popped into the program counter, causing program execution to continue with the instruction following the Call. Thus, one copy of a subroutine may be called from many different points in memory, preventing duplication of code.

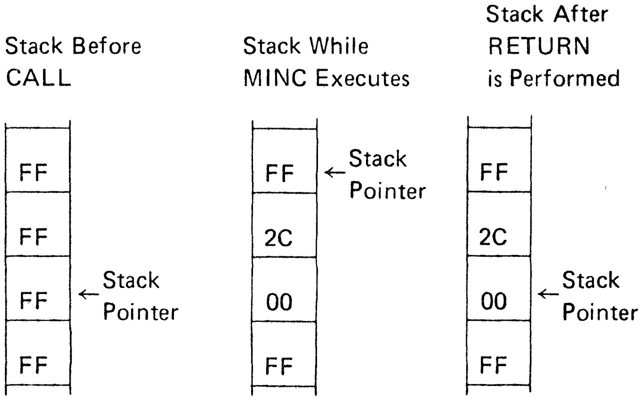
As an example, subroutine MINC increments a 16-bit number held least-significant-byte first in two consecutive memory locations, and then returns to the instruction following the last Call statement executed. The address of the number to be incremented is passed in the H and L registers.

```
Label    Code    Operand    Comment
MINC     BUMP    M          Increment low-order byte
         RNZ                If non-zero, return to calling
*                           routine
         INCP    RH         Address high-order byte
         BUMP    M          Increment high-order byte
         RET                Return unconditionally
```

Assume MINC appears in the following program:



When the first call is executed, address X'2C03' is pushed onto the stack indicated by the stack pointer, and control is transferred to X'3C00'. Execution of either Return statement in MINC will cause the top entry to be popped off the stack into the program counter, causing execution to continue at X'2C03' (since the Call statement is three bytes long).



When the second call is executed, address X'2EF3' is pushed onto the stack, and control is again transferred to MINC. This time, either Return instruction will cause execution to resume at X'2EF3'.

Note that MINC could have called another subroutine during its execution, causing another address to be pushed onto the stack. This can occur as many times as necessary, limited only by the size of memory available for the stack.

Note also that any subroutine could push data onto the stack for temporary storage without affecting the call and return sequences as long as the same amount of data is popped off the stack before executing a Return statement.

6.2 TRANSFERRING DATA TO A SUBROUTINE

A subroutine often requires data to perform its operations. In the simplest case, this data may be transferred in one or more registers. Subroutine MINC in the last section, for example, receives the memory address which it requires in the H and L registers.

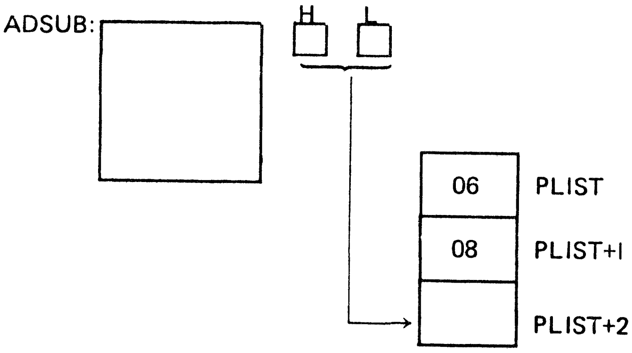
Sometimes it is more convenient and economical to let the subroutine load its own registers. One way to do this is to place a list of the required data (called a parameter list) in some data area of memory, and pass the address of this list to the subroutine in the H and L registers.

For example, the subroutine ADSUB expects the address of a three-byte parameter list in the H and L registers. It adds the first and second bytes of the list, and stores the result in the third byte of the list.

```
Label    Code    Operand    Comment
         LHI     RH,PLIST   Load address of parameter list
*                           into H and L
         CALL    ADSUB      Call the subroutine
RET1      :
          :
PLIST    DC      6          First number to be added
         DC      8          Second number to be added
         DS      1          Result will be stored here
         LHI     RH,LIST2   Load H and L
         CALL    ADSUB      for another call to ADSUB
RET2      :
          :
LIST2    DC      10
         DC      35
         DS      1
          :
          :
*  Subroutine
ADSUB    LB      RA         Get first parameter
         INCP    RH         Increment memory address
         LB      RB         Get second parameter
         AR      RA,RB      Add first to second
         STB     RA         Store result at third
*                           parameter location
         RET                Return unconditionally
```

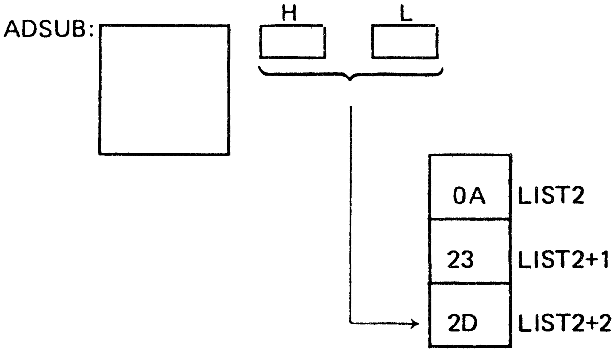
The first time ADSUB is called, it loads the A and B registers from PLIST and PLIST+1, respectively, adds them, and stores the result in PLIST+2. Return is then made to the instruction at RET1.

First call to ADSUB:



The second time ADSUB is called, the H and L registers point to the parameter list LIST 2. The A and B registers are loaded with X'0A' and X'23', respectively, and the sum is stored at LIST2+2. Return is then made to the instruction at RET2.

Second call to ADSUB:



Note that the parameter lists PLIST and LIST2 could appear anywhere in memory without altering the results produced by ADSUB.

6.2.1 External Subroutines

Data can also be transferred to a subroutine by external linkage. This involves the use of the ENTY and EXTN statements that were described in Section 4.1.

Suppose you have a main program and a subroutine used by it. Rather than assembling both at one time, they can be modularized into two separate assemblies. This is advantageous in that if one routine needs to be reassembled, it will not be necessary to include the other program in the assembly.

The example to follow shows a program calling an external subroutine, SUBDP. This subroutine will subtract a double-precision number in VAR2 from a double-precision number in VAR1 and store the result in RESULT.

```
Label     Code    Operand    Comment
          EXTN    SUBDP,VAR1,VAR2,RESULT
* The external subroutine and its variables are linked
* via the EXTN to this program.  Thus one need only
* reassemble the main module and then link up to the
* subroutine
          SHLD    VAR1       Store RH&RL into VAR1
          XCHG               Exchange RH&RL with RD&RE
          SHLD    VAR2       Store RH&RL into VAR2
          CALL    SUBDP      Subtract double-precision
            .
            .
            .
          LHLD    RESULT     Result of subtract into RH&RL
            .
            .
            .
* Subroutine module to perform double-precision subtraction
          ENTY    SUBDP,VAR1,VAR2,RESULT
SUBDP     LHLD    VAR2       RH&RL get VAR2
          XCHG               Exchange RH&RL with RD&RE
          LHLD    VAR1       RH&RL get VAR1
          LR      RA,RL      Subtract RE
          SR      RA,RE      from RL
          LR      RL,RA      and store in RL
          LR      RA,RH      Subtract RD
          SRC     RA,RD      from RH
          LR      RH,RA      and store in RH
          SHLD    RESULT     Save RH&RL in RESULT
          RET                and exit
VAR1      DS      2
VAR2      DS      2
RESULT    DS      2
          END
```

6.3 WRITING INTERRUPT SUBROUTINES

In general, any registers or condition bits changed by an interrupt subroutine must be restored before returning to the interrupted program, or error will occur.

For example, suppose a program is interrupted just prior to the instruction:

```
    BTC  LOC
```

and the carry bit equals 1. If the interrupt subroutine happens to zero the carry bit just before returning to the interrupted program, the branch to LOC which should have occurred will not, causing the interrupted program to produce erroneous results.

Like any other subroutine then, any interrupt subroutine should save at least the condition bits and restore them before performing a Return operation. (The obvious and most convenient way to do this is to save the data in the stack, using PUSH and POP operations.)

Further, the interrupt enable system is automatically disabled whenever an interrupt is acknowledged. Except in special cases, therefore, an interrupt subroutine should include an EI instruction somewhere to permit detection and handling of future interrupts. Any time after an EI is executed, the interrupt subroutine may itself be interrupted. This process may continue to any level, but as long as all pertinent data are saved and restored, correct program execution will continue automatically.

A typical interrupt subroutine, then, could appear as follows:

```
Code    Operand    Comment
----    -------    -------
PUSH    PSW        Save condition bits and Accumulator

EI                 Re-enable interrupts
         .
         .
         .
Perform necessary actions to service the interrupt
         .
         .
         .
POP     PSW        Restore machine status
RET                Return to interrupted program
```

SECTION 7

INPUT/OUTPUT PROGRAMMING

The Zentec 9000 Microcomputer Terminal System communicates with all external devices using two instructions, IN and OUT, which were introduced in Section 5.14. For the 9003 each of these instructions occupy two bytes, where the first byte is the instruction code and the second byte is a device number. The 9002 IN and OUT instructions combine the code and the device number in one byte.

The device number identifies the byte of information (control, data, or status) the 9000 is sending to, or wants to receive from, the external device. As such, most devices will be assigned several device numbers, with each number having a unique meaning to the device.

This section includes I/O programming examples for four common devices: the 9003 keyboard, Disk, RS-232 Tele-Communications and Printer. Tables 7-1 and 7-2 list the device numbers used for the Printer and RS-232. Appendix E lists all of the currently assigned device numbers.

TABLE 7-1

ZENTEC 9003 DEVICE NUMBERS

	INPUT
Device Number	Function
01	*
03	Printer status
05	*
07	*
09	*
0B	*
0D	*
0F	RS-232 Input Character
41	RS-232 Interface Status
43	RS-232 Modem Status
45	RS-232 I.D. No.
47	*
49	*
4B	*
4D	*
4F	*
89	*
8B	*
8D	*
CF	*

	OUTPUT
Device Number	Function
11	*
13	Output to Printer
15	*
17	*
19	*
1B	*
1D	RS-232 Output Character
1F	RS-232 Control Word
21	*
23	*
2B	*
2D	*
2F	*
3B	*
3C	*
3D	*
3E	*
3F	*

*These codes are reserved.

TABLE 7-2

ZENTEC 9002 DEVICE NUMBERS

	INPUT	
Device Number	Accumulator	Function
03	0	Printer Status
0F	0	RS-232 Input Character
01	40	RS-232 Interface Status
03	40	RS-232 Modem Status
05	40	RS-232 I.D. Number

	OUTPUT
Device Number	Function
15	Output to Printer
1D	RS-232 Output Character
1F	RS-232 Control Word

All other device numbers are reserved.

7.1 KEYBOARD AND CRT DISPLAY

This section provides a short keyboard interrogation routine and a description of how to control the special effects on the CRT display.

7.1.1 Keyboard Interrogation Routine

Whenever a key is depressed, the Zentec 9000 keyboard generates an 8-bit character code and loads this code into the keyboard input register in memory (X'1002'). Table 7-3 lists these codes.

The code X'FF' in the keyboard input register has a special meaning and is not generated by any key. This code is a "null" character, which indicates that no keyboard code has been entered in memory. Your program should loop upon sensing X'FF' and should re-store X'FF' after a proper key code has been sensed and processed.

The routine below illustrates one approach by which the program can interrogate this register.

```
Label    Code    Operand       Comment

PICK     LHI     RH,X'1002'    Load keyboard input register
*                              address into H and L
PICKA    LB      RA            Load A from memory
         CI      RA,X'FF'      NULL character present?
         BE      PICKA         Yes, try again
         STBI    M,X'FF'       Store the NULL character
         RET                   Return with character in A
```

7.1.2 CRT Display Special Effects

Memory locations X'1080'-X'17FF' comprise a dedicated portion of RAM memory that stores one byte for every character position on the CRT display. This area is commonly called the video display section of RAM.

Whenever a displayable character or special display effects control code is entered from the keyboard (or another source), the CPU processes that character and writes it into the video display section. From there, it is read out periodically by the video circuitry, transformed into a video signal and displayed on the screen of the CRT display. Thus, the CPU writes into the video display section, as needed to alter the display image, but the video circuitry continuously reads it out.

TABLE 7-3

KEYBOARD CODES

Bit No.

7 6 5 4 3 2 1 0	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011
0000	INSERT LINE	DELETE LINE	SP	0	@	P	`	p	CTRL @	CTRL P		NP Ø
0001	ERASE FIELD	1	!	1	A	Q	a	q	CTRL A	CTRL Q		NP 1
0010	ERASE DISPLAY	2	"	2	B	R	b	r	CTRL B	CTRL R		NP 2
0011	5L	3	#	3	C	S	c	s	CTRL C	CTRL S		NP 3
0100	4	RETURN	S	4	D	T	d	t	CTRL D	CTRL T		NP 4
0101	LINE FEED	MODE	%	5	E	U	e	u	CTRL E	CTRL U		NP 5
0110	DELETE	↑ SCROL	&	6	F	V	f	v	CTRL F	CTRL V		NP 6
0111	ERASE END LINE	↓ SCROL	'	7	G	W	g	w	CTRL G	CTRL W		NP 7
1000	← BACK SPACE	→	(	8	H	X	h	x	CTRL H	CTRL X		NP 8
1001	TAB	BACK TAB	)	9	I	Y	i	y	CTRL I	CTRL Y		NP 9
1010	↓	↑	*	:	J	Z	j	z	CTRL J	CTRL Z		
1011	HOME	ESC	+	;	K	[	k	{	CTRL K	CTRL [		
1100	↑ PAGE	↓ PAGE	,	<	L	\	l	¦	CTRL L	CTRL \		
1101	AUTO TAB	AUTO BACK TAB	-	=	M	]	m	}	CTRL M	CTRL]		
1110	5U	INS / REP	.		N	^	n	~	CTRL N	CTRL ^	NP •	
1111	ERASE END DISP	FORM / EDIT	/	?	O	—	o	DEL	CTRL O	CTRL –		

In the video display section there is space for a total of 1920 bytes of data representing the 80 characters on each of 24 display lines. Any byte stored in the video display section of the RAM is interpreted by the video circuits as either a data character or as a control code. If a byte is interpreted as a data character, it is simply displayed on the screen at the cursor location; if it is a control code, it specifies the special display effect which applies to all following data characters. For example, it can specify that all characters following are to be dimmed, or displayed on a reversed background, etc.

A control character specifies special display effects for all data characters from that location on until the end of the screen, or until another control character is encountered. Figure 7-1 shows the format of the control character.

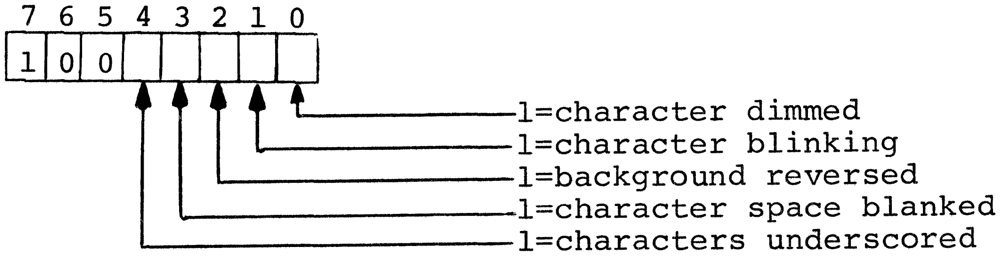


FIGURE 7-1

SPECIAL DISPLAY EFFECTS CONTROL CHARACTER

To turn on one or more of the special effects from a program, simply store the appropriate control character in the associated byte location in RAM. Any of the special display effects can also be combined. All 0 bits in positions 0 thru 4 will cause the display to appear normal.

To initiate a special effect starting at the first character display position, store the control character into memory location X'1004' (Prior Condition Register, see Section 8.1). However, the control character sent to X'1004' should have the three high order bits equal to zero because these bits are reserved.

7.2 DISK

The Zentec 9000 contains two System Support Routines that the program can call to perform disk data transfer operations. Disk read operations are performed by DREAD; disk write operations are performed by DWRITE.

Both routines will fetch the transfer parameters from an eight-byte block of RAM memory. The sequence of this block is shown in Figure 7-2.

LOC.	
N	Error/Status
N+1	Drive Number
N+2	Track Number
N+3	Sector Number
N+4	RAM Location, high-order byte
N+5	RAM Location, low-order byte
N+6	Byte Count, high-order byte
N+7	Byte Count, low-order byte

FIGURE 7-2

DISK TRANSFER PARAMETER LIST

Error/Status indicates the current status of the disk operation. It should be initialized to X'00'. At the completion of the disk I/O routine, Error/Status will contain one of two possible bytes: if bit 5=0, Error/Status should be interpreted as a Status byte; if bit 5=1, Error/Status should be interpreted as an Error byte. The formats are shown in Figure 7-3.

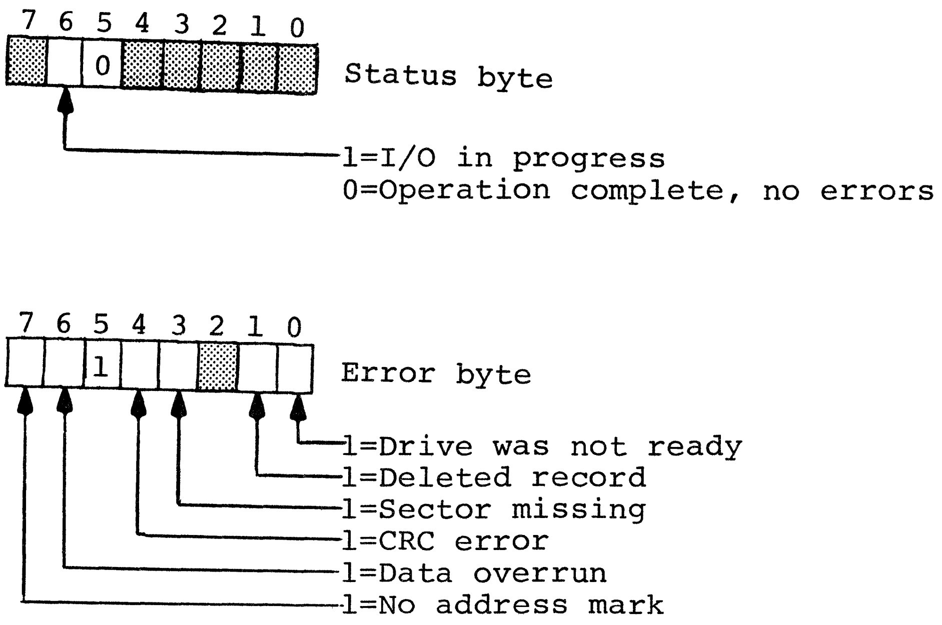
After calling DREAD or DWRITE, the program should interrogate location N to see if the transfer is completed without error (bit 5=0, bit 6=0) or with error (bit 5=1).

Drive Number should be X'00' to select Drive #0 or X'01' to select Drive #1.

Track Number can be assigned a value between X'00' (Track 0) and X'4C' (Track 76).

Sector Number can be assigned a value between X'01' (Sector 1) and X'1A' (Sector 26).

RAM Location is the starting memory address that the disk data will be transferred to (for Read) or from (for Write).



NOTE

Shaded bits are not meaningful to the programmer.

FIGURE 7-3

ERROR/STATUS BYTE

Byte Count is the number of bytes to be transferred. From one byte (X'0001') to 32K bytes (X'7FFF').

Additionally, the DREAD and DWRITE routines will expect to find the starting address of the parameter list (N in Figure 7-2) in registers D and E.

The program below is one way a disk Read operation can be coded. (Note that this same sequence could perform a Write operation if the call was made to DWRITE.)

```
Label    Code    Operand     Comment

         LHI     RD,DLIST    Set up parameter list address
*                            in D and E
         CALL    DREAD       Call Read routine
         LHI     RH,DLIST    Address Error/Status on return
```

```
Label     Code    Operand     Comment

LOOP      LB      RA          Load Error/Status into A
          ANDI    RA,X'60'    Done & no errors?
          BZ      CONT        Yes, branch to continue
          ANDI    RA,X'20'    Error indicated?
          BZ      LOOP        No, try again
          B       ERR         Yes, branch to error routine
           :
CONT      ---
           :
ERR       ---
           :
DLIST     DC      0           Initialize Error/Status
          DC      0           Drive 0
          DC      16          Track 16
          DC      1           Sector 1
          DC      X'22'       Destination is
          DC      X'3F'         location X'223F'
          DC      X'0004'     Read 1K bytes
                                of data
```

7.3 RS-232 COMMUNICATIONS

Programming RS-232 communications requires four device numbers for input and two device numbers for output. Figure 7-4 shows the bit patterns associated with each of these device numbers.

7.3.1 Transmit Routine

The subroutine below, XMIT, will transmit up to 255 characters to a modem. The subroutine assumes the character count has been loaded into register C and the starting memory address of the character buffer is in register pair H and L before XMIT is called.

```
Label     Code     Operand    Comment

XMIT      IN       X'41'      Input interface status
          ANDI     RA,6       Look at XMIT not available or empty
          BNZ      XMIT       Loop if not available
RTS       LBI      RA,3       Load request to send
          OUT      X'1F'      Xmit request to send
MODEM     IN       X'43'      Input modem status
          ANDI     RA,2       Look at clear to send
          BZ       MODEM      Loop if not clear to send
XMITMT    IN       X'41'      Input interface status
          ANDI     RA,2       Look at XMIT not available
          BNZ      XMITMT     Loop if XMIT not available
          LB       RA         Load character into RA
          OUT      X'1D'      Xmit character
          INCP     RH         Address next character
          DEC      RC         Decrement character count
          BNZ      RTS        Branch if not done
          RET                 Return if done
```

Note that XMIT could send more than 255 characters if register pair B and C contained the character count. The only change in the coding would be that a "DECP RB" instruction would replace the existing "DEC RC" instruction.

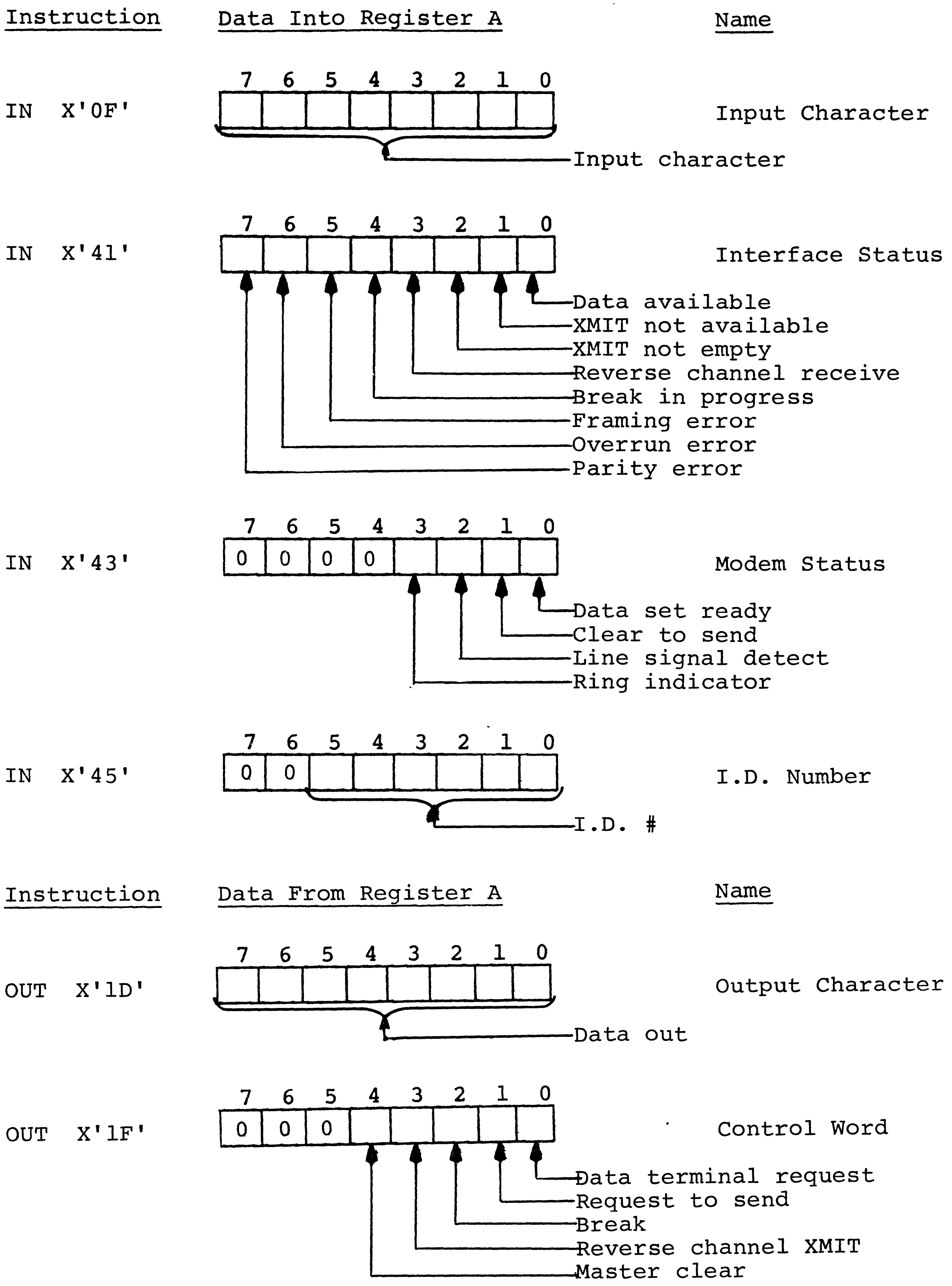


FIGURE 7-4

RS-232 INPUT AND OUTPUT BYTES

7.3.2 Receive Routine

The subroutine below, REC, will receive characters from a data set and store them in memory. The subroutine assumes that the starting memory address of the character buffer is in register pair H and L before REC is called.

```
Label    Code    Operand     Comment

REC      LBI     RA,1        Load data terminal request
         OUT     X'1F'       Send data terminal request
DSR      IN      X'43'       Input modem status
         ANDI    RA,1        See if data set is ready
         BZ      DSR         If not, wait for it
LSD      IN      X'43'       If so, input modem status
         ANDI    RA,4        Line signal detect?
         BZ      LSD         If not, wait for it
DI       IN      X'41'       If so, input interface status
         ANDI    RA,X'01'    Data available?
         BZ      DI          If not, wait for it
         CI      RA,1        If so, check for errors
         BNE     ERRS        Branch on errors
         IN      X'0F'       Input character
         STB     RA          Store character

         INCP    RH          If not, increment buffer address


         .
         .
         .
ERRS     ---                 Error service routine
```

7.4 PRINTER

Programming the printer involves only two instructions: an IN X'03' instruction to check printer status and an OUT X'13' instruction to send the ASCII character byte.

Figure 7-5 shows the format of the printer Status byte. Shaded bits are not meaningful to the programmer.

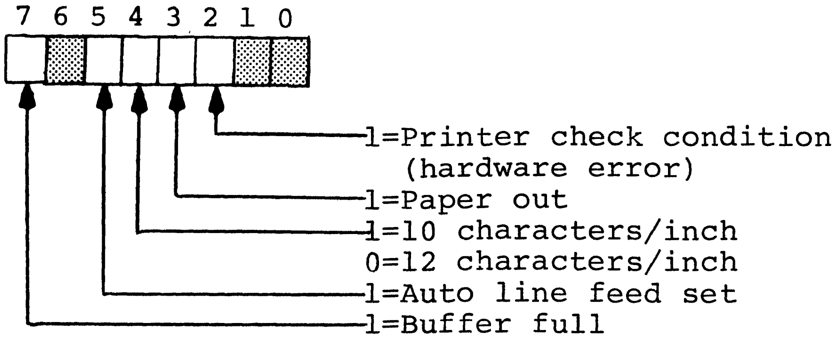


FIGURE 7-5

PRINTER STATUS BYTE

Bits 2 and 3 indicate hardware error conditions that are uncorrectable by the software. Bits 4 and 5 provide printer information to the program. Bit 7 will indicate when the printer is not ready to receive a new character.

Appendix D gives the complete list of ASCII codes.

The instruction sequence below, PRNT, is an example of a subroutine that can be called to print the contents of a buffer in memory. The buffer address is at PLIST and PLIST+1. The character count is at PLIST+2.

```
Label   Code    Operand     Comment

PRNT    LHI     RH,PLIST    Load address of parameter list
*                           into H and L
        LB      RD          Load high-order address into D
        INCP    RH          Increment memory address
        LB      RE          Load low-order address into E
        INCP    RH          Increment memory address
        LB      RC          Load character count into C
PIN     IN      X'03'       Input printer status
        ROL     RA          Rotate buffer full bit to Carry
        BTC     PIN         Loop on buffer full
        ANDI    RA,X'18'    Check or paper out?
        BNZ     HDERR       Yes, branch to hard error routine
```

```
Label     Code      Operand      Comment
POUT      LBA       RD           No, load character into Accumulator
          OUT       X'13'        Send character to printer
          DEC       RC           Decrement character count
          RZ                     Return if count=0
          INCP      RD           Address next character
          B         PIN          Go send next character
           .
           .
           .
           .
PLIST     DC        X'21'        Character buffer is at
          DC        X'3F'        location X'213F'
          DC        25           Print 25 characters
           .
           .
           .
           .
HDERR     --
```

SECTION 8

SYSTEM WORKING REGISTERS

Memory locations X'1000' through X'102F' comprise a dedicated portion of RAM memory that the system uses to communicate with the display, the keyboard, the RS-232 interface, and certain optional interfaces.

Figure 8-1 is a map of this area. The working registers are of two types - software registers and hardware registers. Software registers hold information used by the CPU during data processing, but their contents are not accessible to any other circuits. There are a total of 13 software registers. The software registers derive their name from the fact that they are written into and read out only by the CPU.

Hardware registers are also written into and read out by the CPU, but their main usage is in communication with input/output hardware. Typically, the CPU uses a hardware register to store the results of some data processing routine. These results are read and interpreted by a hardware circuit as an instruction to perform a specific task. Alternatively, a hardware circuit writes data into a register and the CPU fetches the data and processes it. For example, a two-byte register is used by the CPU to actuate the audible tone alarm circuit, and a single byte register is used to receive keyboard data to be processed by the CPU.

8.1 HARDWARE REGISTERS

The hardware registers are as follows:

ADDRESS(ES) DESCRIPTION

X'1000'-X'1001' Cursor Address Registers. There are two cursor address registers that identify the locations of the cursor on the CRT display screen. One register, at location X'1000', identifies the cursor row and the other, at location X'1001', identifies the cursor column.

The values contained in the two registers are derived by the CPU from the contents of the FAS software registers and can range from X'00' to X'19' for the row address register and X'00' to X'4F' for the column address. If the system contains the Page Two Video Display option, row addresses can extend to X'30'; note, however, that row address X'00' corresponds to the 25th line on the screen, which is not normally accessible. Row address

HEX ADDRESS	
1000	Cursor Register, Row Address
1001	Cursor Register, Column Address
1002	Keyboard Input Character Register
1003	Function Register
1004	Prior Condition Register
1005	Page Register
1006	(Reserved)
1007	(Reserved)
1008 1009 100A	Branch Area Address or TAS Software Register
100B	(Reserved)
100C	(Reserved)
100D	(Reserved)
100E	(Reserved)
100F	(Reserved)
1010	Scroll Value
1011	Input Buffer Pointer
1012	Protected Cursor Flag
1013	(Reserved)
1014	Line Space Count
1015	Local Mode Save
1016	Current Mode
1017	Previous Character Hold
1018	Current Character Hold
1019 thru 101E	Keyboard Input Buffer (6 bytes)
101F	(Reserved)
1020 1021	FAS Software Register
1022 1023	SAS Software Register
1024 thru 102D	Open Work Area For CPU (10 bytes)
102E	(Reserved)
102F	(Reserved)

FIGURE 8-1

MAP OF WORKING REGISTERS IN MEMORY

ADDRESS(ES) DESCRIPTION

X'01' corresponds to row 1 of the first video display page and column address X'00' corresponds to the first column on the left of the screen. From these values on, all addresses are contiguous.

X'1002'

Keyboard Input Character Register. The keyboard input character register always receives data from the keyboard and always outputs data to the CPU. A total of 171 different codes are written into the register: 128 standard ASCII alphanumeric characters, punctuation marks, and symbols are defined by the first seven bits, and the eighth bit is used to identify inputs from the numeric pad (11 keys) and 32 codes generated by selected keys while the CTRL key is held depressed (@, A-Z, [, \ ,], ∧, -).

The interface protocol is such that the keyboard is allowed to write any code except X'FF' into the register. After the CPU reads a valid character code out of the register it writes all 1 bits (X'FF') into the register. Thus, when the CPU monitors the register, it interprets X'FF' as the absence of a keyboard character, but any other bit combination is read and processed.

See Section 7 for an example of Keyboard I/O programming.

X'1003'

Function Register. The function register has only one use - to actuate the audible alarm tone. A tone, approximately two seconds in duration, is produced whenever the eighth bit in the function register changes state. That is, whenever the eighth bit changes from 1 to a logic 0 state, or vice versa, the tone alarm circuit is actuated. All other bits in the function register are reserved for program usage, but do not affect the tone alarm circuit.

X'1004'

Prior Condition Register. The prior condition register is used to establish the initial screen polarity and blinking characteristics for the line scans. Special control characters placed within the

ADDRESS(ES) DESCRIPTION

refresh RAM area will vary the screen polarity, tone and blinking characteristics, but it is necessary to establish 'initial' conditions or the first refresh RAM location of each line of the screen would be committed to establishing 'current' screen characteristics. Figure 8-2 shows the format of the prior condition register.

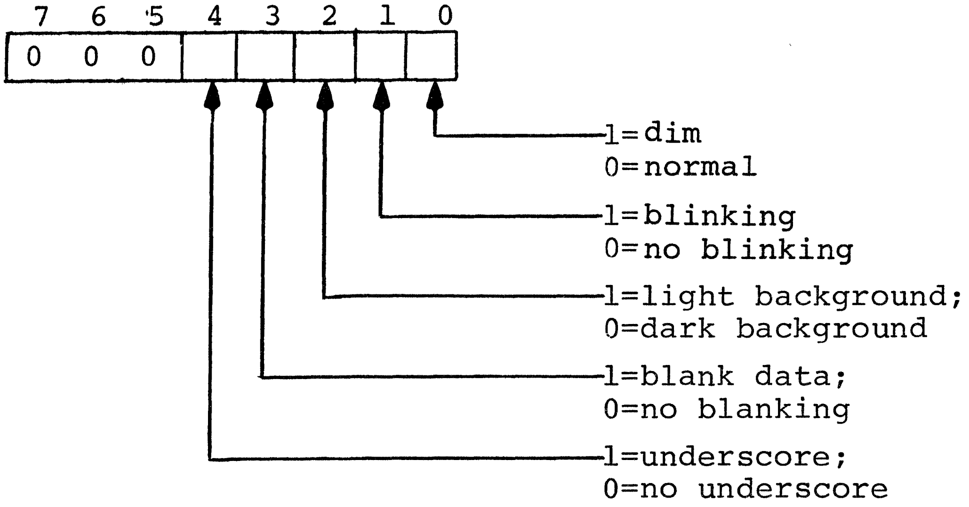


FIGURE 8-2

PRIOR CONDITION REGISTER FORMAT

X'1005'

Page Register. The page register stores the address of that 80-byte segment of the video display section of the RAM which appears as the top line on the CRT screen. Its contents must be X'01' if there is only one page of video data in the RAM, but can be any number between X'01' and X'19' if the Page Two option is installed in the system. If the second page option is in the system, the page register is also used for scrolling. In this case, it can contain any value between X'00' and X'19'. (Even though it is not of any evident practical value, the page register can also be programmed to contain the address X'00', which is

ADDRESS (ES) DESCRIPTION

the 25th line. In that case the control line is displayed both on top and bottom of the screen.)

* The high order bit of the Page Register is used to engage the hardware scroll feature for single page only. If not used, hardware scroll is automatic on two page / 48 line basis.

8.2 SOFTWARE REGISTERS

The software registers are as follows:

ADDRESS(ES) DESCRIPTION

X'1008'-X'100A'

Branch Area or TAS. The Branch Area is a three-byte field used by the system executive to cause branches into the various function routines in the system. Location X'1008' must always contain X'C3', which is a B (Branch) instruction.

The Third Address (TAS) is a two-byte field used as a temporary work area for 16-bit values. Two of the basic routines use TAS. TAS overlays the value portion of the Branch Area (X'1009' and X'100A').

X'1010'

Scroll Value. The scroll value is the amount added to or subtracted from the value in the Page Register to vary the starting line of data on the screen. This value is X'02' unless modified by the user.

X'1011'

Input Buffer Pointer. The system executive supports a six-location buffer (X'1019'-X'101E') to hold keyboard input data. The input buffer pointer holds the next available buffer address.

X'1012'

Protected Cursor Flag. The protected cursor flag is used to allow the cursor to be positioned under a protected character. The flag is tested for zero or non-zero.

X'1014'

Line Space Count. The line space count is used to speed character inserts. This value defines the number of available spaces from the last character, on the line holding the cursor, to the end of the line. This value is used exclusively in 'insert' sub-mode.

X'1015'

Local Mode Save. The local mode save location holds the code representing the mode and sub-mode that was in control prior to entering 'Control' mode. It is used to reestablish the proper mode and sub-mode at return from 'Control' mode.

ADDRESS(ES) DESCRIPTION

X'1016' Current Mode. The current mode code is maintained for program control and list control purposes.

X'1017' Previous Character Hold. The previous character hold maintains the value of the last keyboard character processed. It is used for special double-character sequences.

X'1018' Current Character Hold. The current character hold maintains the value of the keyboard character currently being-processed.

X'1019'-X'101E' Keyboard Input Buffer. The keyboard input buffer is a six-byte buffer used by the system to support keyboard input at its maximum rate, while allowing functions of various speeds to be performed.

X'1020'-X'1021' FAS. The First Address (FAS) is a two-byte field used to hold the 16-bit binary value of the current cursor address. All cursor manipulation programs operate with FAS, and the value is then converted into Row and Column values which are inserted into the cursor hardware registers (see Section 8.1).

X'1022'-X'1023' SAS. The Second Address (SAS) is a two-byte field used as a temporary work area for 16-bit values. A number of the basic routines use SAS.

X'1024'-X'102D' Open Work Area. Locations X'1024' through X'102D' are used by various basic routines as temporary work space.

SECTION 9

HOW TO USE THE ZENTEC ASSEMBLER MODULE (ZAM)

The Zentec Assembler Module (ZAM) consists of several programs. They are:

- Assembler Control Program
- Assembler Edit Program
- Assembler - 9002 Program
- Assembler - 9003 Program
- Loader Program
- Assembler Catalog Program

To use the Zentec Assembler Module, put a ZAM disk and a work disk in your disk unit. The ZAM disk must be put in Drive 0; the work disk must be put in Drive 1.

Place the terminal into the Control mode by pressing the MODE key. Press PAGE ↑, which executes a Disk IPL (Section C.2 of Appendix C) and loads the Assembler Catalog Program.

With the Assembler Catalog Program loaded, press the following sequence of keys:

```
G
E
T
-
E
D
I
T
RETURN
```

This sequence loads the Assembler Control Program. To activate this program, press the MODE key and PROG 2. When the Assembler Control Program is activated, the Terminal will display the message

```
EDITOR READY
```

on the eighth line and the cursor on the twelfth line.

9.1 OPERATION OF THE ASSEMBLER CONTROL PROGRAM

The user can now select one of several operations by entering a pre-assigned command and then pressing the RETURN key (indicated by ® below). The commands are given below.

Command Sequence	Description
EDIT-®	Edit data already on work disk, using Assembler Edit Program (see Section 9.2).
EDIT-name®	Loads 'name' file from ZAM disk onto work disk. If file is not found, work disk is cleared. If file is found, work disk data can be edited using Assembler Edit Program (see Section 9.2).
LOAD-addr,name1,name2...,namen®	Loads programs listed and links them together at starting address 'addr', where 'addr' is a four-digit hexadecimal number standing alone.
LOAD-,name1,name2...,namen®	Loads programs listed and links them together at starting address specified by first module.
SAVE-name®	Creates a file named 'name' on the ZAM disk. File is image of 'NAME' file on work disk.
8ASM-name®	Assembles 9002 code into named data set and stores it on ZAM disk. Prints.
8ASM-name,NP®	Same as above, but without printing.
80ASM-name®	*Assembles 9003 code into named data set and stores it on ZAM disk. Prints.*
80ASM-name,NP®	*Same as above, but without printing.*

The assembler (either 8ASM or 80ASM) will be temporarily halted if any key is typed while it is running. This will, for example, make it possible to adjust the paper in the printer during the printing of the listing. The assembler will restart where it left off when a second key is typed.

9.2 OPERATION OF THE ASSEMBLER EDIT PROGRAM

Either of the EDIT command sequences in Section 9.1 will cause the Assembler Edit Program to be loaded. The following list gives all the Edit Program operations by both keyboard label and ZAM function.

ZAM FUNCTION	FUNCTION	KEYBOARD KEY
REPLACE LINE	Re;aces old line with line of data on screen.	DELETE
DELETE LINE	Deletes line of data.	DELETE LINE
ERASE DISPLAY LINE	Erases data on screen.	ERASE END LINE
COPY LAST LINE	Copies last line of data entered onto screen.	ERASE END DISPLAY
EDIT COMPLETE	Returns control to the Assembler Control Program.	ESC
HOME	Moves cursor to leftmost position on screen.	HOME
LINE 1	Displays line #1 on screen.	F1
SEARCH	Starts a search.	F2
REPLACE	Replaces the data searched by the replace data.	F3
SET LINE GET	Causes next four numeric key strokes to be interpreted as a line number.	F4
PERFORM LINE GET	Loads record line number that was selected by four numeric key strokes after F4. NOTE If selected line number is not higher than last line number, line 1 will be selected.	F5

INSERT	Inserts a new line of data after last entry.	RETURN or
		INSERT LINE

ZAM Function	Function	Keyboard Key
SCROL ↑	Loads next line to screen.	SCROL ↑
TAB	Performs TAB operation.	TAB
←	Moves cursor left one position, except when it is at leftmost position.	←
→	Moves cursor right one position, except when it is at rightmost position.	→
SEARCH AND REPLACE	Performs the F2/F3 functions automatically to the end of data.	9 (Numeric Pad)
DELETE	Successive delete key when held down.	. (Numeric Pad)

A SEARCH field may be defined by pressing the slash key followed by the character to be searched, followed by another slash, Ⓡ .

A REPLACE field may be defined by inserting data to replace the searched data after the second slash in the search definition. The end of the replace field is defined by a slash, Ⓡ .

When using the Editor it is necessary to insert the master diskette containing the software into Disk Drive 0. A scratch diskette should be inserted into Disk Drive 1.

To create a new file:

1. Enter the CATALOG mode and get the EDIT file from the master diskette (i.e., GET-EDIT R).
2. Execute location X'2000', the starting address of EDIT.
3. From the keyboard enter EDIT-NEW R .
4. Wait until a header appears on the screen. Then enter through the keyboard the source statements followed by a carriage return. The edit functions listed above may be used.
5. When the editing process is finished, save the newly created file on the master diskette by pressing the ESCAPE key and entering SAVE-fn where fn is the name of the file to be saved. The file name must not be more than three alphanumeric characters.

To edit an existing file, repeat the above instructions, substituting for #3 EDIT-fn where fn is the existing file name.

APPENDIX A

9000 SERIES INSTRUCTION SET SUMMARY

This appendix is designed to give the manual better potential as a reference tool.

The instruction set in Section 5 is organized to help the reader learn the instructions. However, since the instruction mnemonics are not ordered alphabetically (although they are so ordered within a group), this organization can prove cumbersome for reference work. To solve this problem, Table A-1 is a listing of the instruction set in alphabetic order, Table A-2 lists the 9003 instruction set by hexadecimal code, Table A-3 lists the 9002 instruction set by hexadecimal code and Table A-4 gives an operational summary of the set.

You will note, incidentally, that several of the instructions display the same hex code. Such instructions are equivalent and were assigned several mnemonics merely to make a program easier to understand. For example, the CZ (Call On Zero) and CE (Call On Equal) instructions both have the same hex code: X'CC' for the 9003; X'6A' for the 9002. However, the programmer will feel more at ease in coding CZ if he is looking for a zero Accumulator and CE if he is looking for an "equal" result to a preceding compare operation.

PROGRAM STATUS WORD (PSW) DEFINITIONS:

BIT	PSW
7	sign
6	zero
5	Ø
4	auxillary carry
3	Ø
2	parity
1	1
0	carry

TABLE A-1

ALPHABETIC LISTING OF 9000 SERIES INSTRUCTION SET

INSTRUCTION		MEANING	9003 HEX	9002 HEX	SECTION
A	RA	Add	86	87	5.6.1
AC	RA	Add Carry	8E	8F	5.6.2
ACI	RA,mm	Add Carry Immediate	CE	0C	5.6.3
AI	RA,mm	Add Immediate	C6	04	5.6.4
AND	RA	Logical AND	A6	A7	5.6.5
ANDI	RA,mm	AND Immediate	E6	24	5.6.6
ANDR	RA,Rn	AND Register	A7,A0-A5	A0-A6	5.6.7
AR	RA,Rn	Add Register	87,80-85	80-86	5.6.8
ARC	RA,Rn	Add Register Carry	88-8F	88-8E	5.6.9
B	label	Absolute Branch	C3	44	5.10.1
BE	label	Branch On Equal	CA	68	5.10.2
BFC	label	Branch False Carry	D2	40	5.10.3
BFP	label	Branch False Parity-Odd	E2	58	5.10.4
BFS	label	Branch False Sign	F2	50	5.10.5
BFZ	label	Branch False Zero	C2	40	5.10.6
BH	label	Branch On High Or Equal	D2	40	5.10.7
BL	label	Branch On Low	DA	60	5.10.8
BM	label	Branch On Minus	FA	70	5.10.9
BNE	label	Branch On Not Equal	C2	48	5.10.10
BNM	label	Branch On Not Minus	F2	50	5.10.11
BNP	label	Branch On Not Plus	FA	70	5.10.1
BNZ	label	Branch On Not Zero	C2	48	5.10.13
BP	label	Branch On Plus	F2	50	5.10.14
BTC	label	Branch True Carry	DA	60	5.10.15
BTP	label	Branch True Parity-Even	EA	78	5.10.16
BTS	label	Branch True Sign	FA	70	5.10.17
BTZ	label	Branch True Zero	CA	68	5.10.18
BUMP	M	Bump Memory	04-3C	10-30	5.3.1
BZ	label	Branch On Zero	CA	68	5.10.19
C	RA	Compare	BE	BF	5.6.10
CALL	sub	Absolute Call	CD	46	5.11.1
CE	sub	Call On Equal	CC	6A	5.11.2
CFC	sub	Call False Carry	D4	42	5.11.3
CFP	sub	Call False Parity	E4	5A	5.11.4
CFS	sub	Call False Sign	F4	52	5.11.5
CFZ	sub	Call False Zero	C4	4A	5.11.6
CH	sub	Call On High Or Equal	D4	42	5.11.7
CI	RA,mm	Compare Immediate	FE	3C	5.6.11
CL	sub	Call On Low	DC	62	5.11.8
CM	sub	Call On Minus	FC	72	5.11.9
CNE	sub	Call On Not Equal	C4	4A	5.11.10
CNM	sub	Call On Not Minus	F4	52	5.11.11
CNP	sub	Call On Not Plus	FC	72	5.11.12
CNZ	sub	Call On Not Zero	C4	4A	5.11.13
COM	RA	Complement Accumulator	2F	-	5.3.2
COMC		Complement Carry	3F	-	5.2.1
CP	sub	Call On Plus	F4	52	5.11.14

TABLE A-1 (Continued)

INSTRUCTION		MEANING	9003 HEX	9002 HEX	SECTION
CR	RA,Rn	Compare Registers	BF,B8-BD	B9-BE	5.6.12
CTC	sub	Call True Carry	DC	62	5.11.15
CTP	sub	Call True Parity	EC	7A	5.11.16
CTS	sub	Call True Sign	FC	72	5.11.17
CTZ	sub	Call True Zero	CC	6A	5.11.18
CZ	sub	Call On Zero	CC	6A	5.11.19
DAA	RA	Decimal Adjust Accumulator	27	-	5.3.3
DAD	Rp	Double Add	09,19,29,39	-	5.8.1
DEC	M	Decrement Memory	35	-	5.3.4
DEC	Rn	Decrement Register	05-3D	09-31	5.3.4
DECP	Rp	Decrement Register Pair	0B,1B,2B,3B	-	5.8.2
DI		Disable Interrupts	F3	-	5.13.2
EI		Enable Interrupts	FB	-	5.13.1
IN	dev	Input	DB	41-4F	5.14.1
INCP	Rp	Increment Register Pair	03,13,23,33	-	5.8.3
LAD	addr	Load Accumulator Direct	3A	-	5.9.1
LB	Rn	Load Byte	46-7E	C7-F7	5.5.1
LBA	RB	Load Byte to Accumulator	0A	-	5.5.2
LBA	RD	Load Byte to Accumulator	1A	-	5.5.2
LBI	Rn,mm	Load Byte Immediate	06-3E	06-36	5.5.3
LHLD	addr	Load H and L Direct	2A	-	5.9.2
LHI	Rp,mm	Load Half-Word Immediate	01,11,21,31	(See LBI)	5.8.4
LR	Rd,Rs	Load Register	40-7F	CD-F6	5.5.4
NOP		No Operation	00	C0	5.4.1
O	RA	Logical OR	B6	B7	5.6.13
OI	RA	OR Immediate	F6	34	5.6.14
OR	RA,Rn	OR Registers	B7,B0-B5	B0-B6	5.6.15
OUT	dev	Output	D3	51-7F	5.14.2
PCHL		Load Program Counter	E9	-	5.10.20
POP	Rp	Pop Data Off Stack	C1,D1,E1,F1	-	5.8.5
PUSH	Rp	Push Data Onto Stack	C5,D5,E5,F5	-	5.8.6
RET		Absolute Return	C9	07	5.12.1
RE		Return On Equal	C8	2B	5.12.2
RFC		Return False Carry	D0	03	5.12.3
RFP		Return False Parity	E0	1B	5.12.4
RFS		Return False Sign	F0	13	5.12.5
RFZ		Return False Zero	C0	0B	5.12.6
RHE		Return On High Or Equal	D0	03	5.12.7
RL		Return On Low	D8	23	5.12.8
RLC	RA	Rotate Left Carry	17	12	5.7.1
RM		Return On Minus	F8	33	5.12.9
RNE		Return On Not Equal	C0	0B	5.12.10
RNM		Return On Not Minus	F0	13	5.12.11
RNP		Return On Not Plus	F8	33	5.12.12
RNZ		Return On Not Zero	C0	0B	5.12.13
ROL	RA	Rotate Left	07	02	5.7.2
ROR	RA	Rotate Right	0F	0A	5.7.3
RP		Return On Plus	F0	13	5.12.14
RRC	RA	Rotate Right Carry	1F	1A	5.7.4
RTC		Return True Carry	D8	23	5.12.15

TABLE A-1 (Continued)

INSTRUCTION	MEANING	9003 HEX	9002 HEX	SECTION
RTP	Return True Parity	E8	3B	5.12.16
RTS	Return True Sign	F8	33	5.12.17
RTZ	Return True Zero	C8	2B	5.12.18
RZ	Return On Zero	C8	2B	5.12.19
S RA	Subtract	96	97	5.6.16
SC RA	Subtract Carry	9E	9F	5.6.17
SCI RA,mm	Subtract Carry Immediate	DE	1C	5.6.18
SETC	Set Carry	37	-	5.2.2
SHLD addr	Store H and L Direct	22	-	5.9.3
SI RA,mm	Subtract Immediate	D6	14	5.6.19
SPHL	Load SP From H and L	F9	-	5.8.7
SR RA,Rn	Subtract Register	97,90-95	90-96	5.6.20
SRC RA,Rn	Subtract Register Carry	9F,98-90	98-9E	5.6.21
STA RB	Store Accumulator	02	-	5.5.5
STA RD	Store Accumulator	12	-	5.5.5
STB Rn	Store Byte	71-77	F8-FE	5.5.6
STBI mm	Store Byte Immediate	36	3E	5.5.7
STD addr	Store Accumulator Direct	32	-	5.9.4
X RA	Exclusive OR	AE	AF	5.6.22
XCHG	Exchange Registers	EB	-	5.8.8
XI RA,mm	Exclusive OR Immediate	EE	2C	5.6.23
XR RA,Rn	Exclusive OR Register	AF,AB-AD	A8-AE	5.6.24
XTHL	Exchange Stack	E3	-	5.8.9

NOTES:

1. The exact hex code for some instructions depends upon the specified register or registers. For these instructions a range of hex codes are shown. The hex code for a particular case may be found in Table A-2 (for the 9003) or Table A-3 (for the 9002).

2. Instructions which are not available on the 9002 are indicated by a "-" in the hex code column.

TABLE A-2
HEXADECIMAL CODE FOR 9003 INSTRUCTION SET

HEX	INSTRUCTION	
00	NOP	
01	LHI	RB,mm
02	STA	RB
03	INCP	RB
04	BUMP	RB
05	DEC	RB
06	LBI	RB,mm
07	ROC	RA
09	DAD	RB
0A	LBA	RB
0B	DECP	RB
0C	BUMP	RC
0D	DEC	RC
0E	LBI	RC,mm
11	LHI	RD,mm
12	STA	RD
13	INCP	RD
14	BUMP	RD
15	DEC	RD
16	LBI	RD,mm
17	RLC	RA
19	DAD	RD
1A	LBA	RD
1B	DECP	RD
1C	BUMP	RE
1D	DEC	RE
1E	LBI	RE,mm
1F	R RC	RA
21	LHI	RH,mm
22	SHLD	addr
23	INCP	RH
24	BUMP	RH
25	DEC	RH
26	LBI	RH,mm
27	DAA	RA
29	DAD	RH
2A	LHLD	addr
2B	DECP	RH
2C	BUMP	RL
2D	DEC	RL
2E	LBI	RL,mm
2F	COM	RA
31	LHI	SP,mm
32	STD	addr
33	INCP	SP
34	BUMP	M
35	DEC	M
36	STBI	mm
37	SETC	
39	DAD	SP
3A	LAD	addr
3B	DECP	SP
3C	BUMP	RA
3D	DEC	RA
3E	LBI	RA,mm
3F	COMC	
40	LR	RB,RB
41	LR	RB,RC
42	LR	RB,RD
43	LR	RB,RE
44	LR	RB,RH
45	LR	RB,RL
46	LB	RB
47	LR	RB,RA
48	LR	RC,RB
49	LR	RC,RC
4A	LR	RC,RD
4B	LR	RC,RE
4C	LR	RC,RH
4D	LR	RC,RL
4E	LB	RC
4F	LR	RC,RA
50	LR	RD,RB
51	LR	RD,RC
52	LR	RD,RD
53	LR	RD,RE
54	LR	RD,RH
55	LR	RD,RL
56	LB	RD
57	LR	RD,RA
58	LR	RE,RB
59	LR	RE,RC
5A	LR	RE,RD
5B	LR	RE,RE
5C	LR	RE,RH
5D	LR	RE,R L
5E	LB	RE
5F	LR	RE,RA
60	LR	RH,RB
61	LR	RH,RC
62	LR	RH,RD
63	LR	RH,RE
64	LR	RH,RH
65	LR	RH,RL
66	LB	RH
67	LR	RH,RA
68	LR	RL,RB
69	LR	RL,RC
6A	LR	RL,RD
6B	LR	RL,RE
6C	LR	RL,RH
6D	LR	RL,RL
6E	LB	RL
6F	LR	RL,RA
70	STB	RB
71	STB	RC
72	STB	RD
73	STB	RE
74	STB	RH
75	STB	RL
77	STB	RA
78	LR	RA,RB
79	LR	RA,RC
7A	LR	RA,RD
7B	LR	RA,RE
7C	LR	RA,RH
7D	LR	RA,RL
7E	LB	RA
7F	LR	RA,RA
80	AR	RA,RB
81	AR	RA,RC
82	AR	RA,RD
83	AR	RA,RE
84	AR	RA,RH
85	AR	RA,RL
86	A	RA
87	AR	RA,RA
88	ARC	RA,RB
89	ARC	RA,RC
8A	ARC	RA,RD
8B	ARC	RA,RE
8C	ARC	RA,RH
8D	ARC	RA,RL
8E	AC	RA
8F	ARC	RA,RA
90	SR	RA,RB
91	SR	RA,RC
92	SR	RA,RD
93	SR	RA,RE
94	SR	RA,RH
95	SR	RA,RL
96	S	RA
97	SR	RA,RA
98	SRC	RA,RB
99	SRC	RA,RC
9A	SRC	RA,RD
9B	SRC	RA,RE
9C	SRC	RA,RH
0F	ROR	RA

TABLE A-2 (Continued)

HEX	INSTRUCTION
9D	SRC RA,RL
9E	SC RA
9F	SRC RA,RA
A0	AND RA,RB
A1	AND RA,RC
A2	AND RA,RD
A3	AND RA,RE
A4	AND RA,RH
A5	AND RA,RL
A6	AND RA
A7	AND RA,RA
A8	XR RA,RB
A9	XR RA,RC
AA	XR RA,RD
AB	XR RA,RE
AC	XR RA,RH
AD	XR RA,RL
AE	X RA
AF	XR RA,RA
B0	OR RA,RB
B1	OR RA,RC
B2	OR RA,RD
B3	OR RA,RE
B4	OR RA,RH
B5	OR RA,RL
B6	O RA
B7	OR RA,RA
B8	CR RA,RB
B9	CR RA,RC
BA	CR RA,RD
BB	CR RA,RE
BC	CR RA,RH
BD	CR RA,RL
BE	C RA
BF	CR RA,RA
C0	RFZ
C0	RNE
C1	POP RB
C2	BFZ label
C2	BNE label
C2	BNZ label
C3	B label
C4	CFZ sub
C4	CNE sub
C4	CNZ sub
C5	PUSH RB
C6	AI RA,mm
C8	RE
C8	RTZ
C8	RZ
C9	RET
CA	BE label
CA	BTC label
CA	BZ label
CC	CE sub
CC	CTZ sub
CC	CZ sub
CD	CALL sub
CE	ACI RA,mm
D0	RFC
DO	RH
D1	POP RD
D2	BFC label
D2	BH label
D3	OUT dev
D4	CFC sub
D4	CH sub
D5	PUSH RD
D6	SI RA,mm
D8	RL
D8	RTC
DA	BL label
DA	BTC label
DB	IN dev
DC	CL sub
DC	CTC sub
DE	SCI RA,mm
DF	
E0	RFP
E1	POP RH
E2	BFP label
E3	XTHL
E4	CFP sub
E5	PUSH RH
E6	ANDI RA,mm
E8	RTP
E9	PCHL
EA	BTP label
EB	XCHG
EC	CTP sub
EE	XI RA,mm
F0	RFS
F0	RNM
FO	RP
F1	POP PSW
F2	BFS Label
F2	BNM label
F2	BP label
F3	DI
F4	CFS sub
F4	CNM sub
F4	CP sub
F5	PUSH PSW
F6	OI RA
F8	RM
F8	RNP
F8	RTS
F9	SPHL
FA	BM label
FA	BMP label
FA	BTS label
FB	EI
FC	CM sum
FC	CNP sub
FC	CTS sub
FE	CI RA,mm

TABLE A-3
HEXADECIMAL CODE FOR 9002 INSTRUCTION SET

HEX	INSTRUCTION	
00	(Power Up Restart)	
02	ROL	RA
03	RFC	label
03	RHE	label
04	AI	RA,mm
06	LBI	RA,mm
07	RET	label
08	BUMP	RB
09	DEC	RB
0A	ROR	RA
0B	RFZ	label
0B	RNZ	label
0B	RNE	label
0C	ACI	RA,mm
0E	LBI	RB,mm
0F	RET	label
10	BUMP	RC
11	DEC	RC
12	RLC	RA
13	RFS	label
13	RP	label
13	RNM	label
14	SI	RA,mm
16	LBI	RC,mm
18	BUMP	RD
19	DEC	RD
1A	RRC	RA
1B	RFP	label
1C	SCI	RA,mm
1E	LBI	RD,mm
20	BUMP	RE
21	DEC	RE
23	RTC	label
23	RL	label
24	ANDI	RA,mm
26	LBI	RE,mm
28	BUMP	RH
29	DEC	RH
2B	RTZ	label
2B	RZ	label
2B	RE	label
2C	XI	RA,mm
2E	LBI	RH,mm
30	BUMP	RL
31	DEC	RL
33	RTS	label
33	RM	label
33	RNP	label
34	OI	RA,mm
36	LBI	RL,mm
3B	RTP	label
3C	CI	RA,mm
3E	STBI	M,mm
40	BFC	label
40	BHE	label
41	IN	dev
42	CFC	label
42	CHE	label
43	IN	dev
44	B	label
45	IN	dev
46	CALL	label
47	IN	dev
48	BFZ	label
48	BNZ	label
48	BNE	label
49	IN	dev
4A	CFZ	label
4A	CNZ	label
4A	CNE	label
4B	IN	dev
4C	B	label
4D	IN	dev
4E	CALL	label
4F	IN	dev
50	BFS	label
50	BP	label
50	BNM	label
51	OUT	dev
52	CFS	label
52	CP	label
53	OUT	dev
54	B	label
55	OUT	dev
56	CALL	label
57	OUT	dev
58	BFP	label
59	OUT	dev
5A	CFP	label
5B	OUT	dev
5C	B	label
5D	OUT	dev
5E	CALL	label
5F	OUT	dev
60	BTC	label
60	BL	label
61	OUT	dev
62	CTC	label
62	CL	label
63	OUT	dev
64	B	label
65	OUT	dev
66	CALL	label
67	OUT	dev
68	BTZ	label
68	BE	label
68	BZ	label
69	OUT	dev
6A	CTZ	label
6A	CZ	label
6A	CE	label
6B	OUT	dev
6C	B	label
6D	OUT	dev
6E	CALL	label
6F	OUT	dev
70	BTS	label
70	BM	label
70	BNP	label
71	OUT	dev
72	CTS	label
72	CM	label
72	CNP	label
73	OUT	dev
74	B	label
75	OUT	dev
76	CALL	label
77	OUT	dev
78	BTP	label
79	OUT	dev
7A	CTP	label
7B	OUT	dev
7C	B	label
7D	OUT	dev
7E	CALL	label
7F	OUT	dev
80	AR	RA,RA
81	AR	RA,RB
82	AR	RA,RC
83	AR	RA,RD
84	AR	RA,RE
85	AR	RA,RH
86	AR	RA,RL
87	A	RA

TABLE A-3 (Continued)

HEX	INSTRUCTION	
88	ARC	RA,RA
89	ARC	RA,RB
8A	ARC	RA,RC
8B	ARC	RA,RD
8C	ARC	RA,RE
8D	ARC	RA,RH
8E	ARC	RA,RL
8F	AC	RA
90	SR	RA,RA
91	SR	RA,RB
92	SR	RA,RC
93	SR	RA,RD
94	SR	RA,RE
95	SR	RA,RH
96	SR	RH,RL
97	S	RA
98	SRC	RA,RA
99	SRC	RA,RR
9A	SRC	RA,RC
9B	SRC	RA,RD
9C	SRC	RA,RE
9D	SRC	RA,RH
9E	SRC	RA,RL
9F	SC	RA
A0	ANDR	RA,RA
A1	ANDR	RA,RB
A2	NADR	RA,RC
A3	ANDR	RA,RD
A4	ANDR	RA,RE
A5	ANDR	RA,RH
A6	ANDR	RA,RL
A7	AND	RA
A8	XR	RA,RA
A9	XR	RA,RB
AA	XR	RA,RC
AB	XR	RA,RD
AC	XR	RA,RE
AD	XR	RA,RH
AE	XR	RH,RL
AF	X	RA
B0	OR	RA,RA
B1	OR	RA,RB
B2	OR	RA,RC
B3	OR	RA,RD
B4	OR	RA,RE
B5	OR	RA,RH
B6	OR	RA,RL
B7	O	RA
B8	CR	RA,RA
B9	CR	RA,RB
BA	CR	RA,RC
BB	CR	RA,RD
BC	CR	RA,RE
BD	CR	RA,RH
BE	CR	RA,RL
BF	C	RA
C0	LR	RA,RA
C1	LR	RA,RB
C2	LR	RA,RC
C3	LR	RA,RD
C4	LR	RA,RE
C5	LR	RA,RH
C6	LR	RA,RL
C7	LB	RA
C8	LR	RB,RA
C9	LR	RB,RB
CA	LR	RB,RC
CB	LR	RB,RD
CC	LR	RB,RE
CD	LR	RB,RH
CE	LR	RB,RL
CF	LB	RB
D0	LR	RC,RA
D1	LR	RC,RB
D2	LR	RC,RC
D3	LR	RC,RD
D4	LR	RC,RE
D5	LR	RC,RH
D6	LR	RC,RL
D7	LB	RC
D8	LR	RD,RA
D9	LR	RD,RB
DA	LR	RD,RC
DB	LR	RD,RD
DC	LR	RD,RE
DD	LR	RD,RH
DE	LR	RD,RL
DF	LB	RD
E0	LR	RE,RA
E1	LR	RE,RB
E2	LR	RE,RC
E3	LR	RE,RD
E4	LR	RE,RE
E5	LR	RE,RH
E6	LR	RE,RL
E7	LB	RE
E8	LR	RH,RA
E9	LR	RH,RB
EA	LR	RH,RC
EB	LR	RH,RD
EC	LR	RH,RE
ED	LR	RH,RH
EE	LR	RH,RL
EF	LB	RH
F0	LR	RL,RA
F1	LR	RL,RB
F2	LR	RL,RC
F3	LR	RL,RD
F4	LR	RL,RE
F5	LR	RL,RH
F6	LR	RL,RL
F7	LB	RL
F8	STB	RA
F9	STB	RB
FA	STB	RC
FB	STB	RD
FC	STB	RE
FD	STB	RH
FE	STB	RL

TABLE A-4

OPERATIONAL SUMMARY OF THE 9000 SERIES INSTRUCTION SET

CODE	OPERATION
CARRY BIT INSTRUCTIONS	
COMC	Carry ← $\overline{\text{Carry}}$
SETC	Carry ← 1
SINGLE REGISTER INSTRUCTIONS	
BUMP	Rn ← Rn+1
COM	RA ← $\overline{\text{RA}}$
DAA	If $(A_0-A_3)>9$ or (Aux. Carry)=1, (A) ← (A)+6 Then if $(A_4-A_7)>9$ or (Carry)=1, $(A)=(A)+6\cdot 2^4$
DEC	Rn ← Rn-1
NOP INSTRUCTION	
NOP	No Operation
DATA TRANSFER INSTRUCTIONS	
LB	Rn ← M
LBA	RA ← M
LBI	Rn ← mm
LR	Rd ← Rs
STA	M ← RA
STB	M ← Rn
STBI	M ← mm
REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS	
A	RA ← RA+M
AC	RA ← RA+M+Carry

TABLE A-4 (Continued)

CODE	OPERATION
ACI	RA ← RA+mm+Carry
AI	RA ← RA+mm
AND	RA ← RA Λ M, Carry ← 0
ANDI	RA ← RA Λ mm, Carry ← 0
ANDR	RA ← RA Λ Rn, Carry ← 0
AR	RA ← RA+Rn
ARC	RA ← RA+Rn+Carry
C	(RA-M)
CI	(RA-mm)
CR	(RA-Rn)
O	RA ← RA V M, Carry ← 0
OI	RA ← RA V mm, Carry ← 0
OR	RA ← RA V Rn
S	RA ← RA-M
SC	RA ← RA-(M+Carry)
SCI	RA ← RA-(mm+Carry)
SI	RA ← RA-mm
SR	RA ← RA-Rn
SRC	RA ← RA-(Rn+Carry)
X	RA ← RA ⊻ M, Carry 0
XI	RA ← RA ⊻ mm, Carry 0
XR	RA ← RA ⊻ Rn, Carry 0
ROTATE ACCUMULATOR INSTRUCTIONS	
RLC	Carry ← Ra_{7},RA_{1-7} ← RA_{0-6} and RA_{0} ← Carry
ROL	Carry ← RA_{7},RA_{1-7} ← RA_{0-6} and RA_{0} ← RA_{7}
ROR	Carry ← RA_{0},RA_{0-6} ← RA_{1-7} and RA_{7} ← RA_{0}
RRC	Carry ← RA_{0},RA_{0-6} ← RA_{1-7} and RA_{7} ← RA_{0}
REGISTER PAIR INSTRUCTIONS	
DAD	H,L ← (H,L)+Rp

TABLE A-4 (Continued)

CODE	OPERATION
DECP	Rp ← Rp-1
INCP	Rp ← Rp+1
LHI	Rp ← mmnn
POP	(RP1) ← (SP+1), (RP2) ← SP, SP ← (SP+2)
PUSH	(SP-1) ← (RP1), (SP-2) ← (RP2), SP ← (SP-2)
SPHL	SP ← H,L
XCHG	D,E ↔ H,L
XTHL	L ↔ (SP), H ↔ (SP+1)
DIRECT ADDRESSING INSTRUCTIONS	
LAD	RA ← M
LHLD	RL ← M and RH ← M+1
SHLD	M ← RL and M+1 ← RH
STD	M ← RA
BRANCH INSTRUCTIONS	
B	Prog. Counter ← label
BE	Prog. Counter ← label if Zero=1
BFC	Prog. Counter ← label if Carry=0
BFP	Prog. Counter ← label if Parity=0
BFS	Prog. Counter ← label if Sign=0 and Zero=0
BFZ	Prog. Counter ← label if Zero=0
BHE	Prog. Counter ← label if Carry=0
BL	Prog. Counter ← label if Carry=1
BM	Prog. Counter ← label if Sign=1
BNE	Prog. Counter ← label if Zero=0
BNM	Prog. Counter ← label if Sign=0
BNP	Prog. Counter ← label if Sign=1
BNZ	Prog. Counter ← label if Zero=0
BP	Prog. Counter ← label if Sign=0

TABLE A-4 (Continued)

CODE	OPERATION
BTC	Prog. Counter ← label if Carry=1
BTP	Prog. Counter ← label if Parity=1
BTS	Prog. Counter ← label if Sign=1
BTZ	Prog. Counter ← label if Zero=1
BZ	Prog. Counter ← label if Zero=1
PCHL	PC ← H,L
CALL SUBROUTINE INSTRUCTIONS	
CALL	(Stack) ← PC, SP ← (SP-2), PC ← addr
CE	If Zero=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CFC	If Carry=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CFP	If Parity=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CFS	If Sign=0 and Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CFZ	If Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CHE	If Carry=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CL	If Carry=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CM	If Sign=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CNE	If Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CNM	If Sign=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CNP	If Sign=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CNZ	If Zero=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CP	If Sign=0, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CTC	If Carry=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CTP	If Parity=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CTS	If Sign=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CTZ	If Zero-1, then (Stack) ← PC, SP ← (SP-2), PC ← addr
CZ	If Zero=1, then (Stack) ← PC, SP ← (SP-2), PC ← addr

TABLE A-4 (Continued)

CODE	OPERATION
RETURN FROM SUBROUTINE INSTRUCTIONS	
RET	Prog. Counter ← SP
RE	Prog. Counter ← SP if Zero=1
RFC	Prog. Counter ← SP if Carry=0
RFP	Prog. Counter ← SP if Parity=0
RFS	Prog. Counter ← SP if Sign=0 and Zero=0
RFZ	Prog. Counter ← SP if Zero=0
RHE	Prog. Counter ← SP if Carry=0
RL	Prog. Counter ← SP if Carry=1
RM	Prog. Counter ← SP if Sign=1
RNE	Prog. Counter ← SP if Zero=0
RNM	Prog. Counter ← SP if Sign=0
RNP	Prog. Counter ← SP if Sign=1
RNZ	Prog. Counter ← SP if Zero=0
RP	Prog. Counter ← SP if Sign=0
RTC	Prog. Counter ← SP if Carry=1
RTP	Prog. Counter ← SP if Parity=1
RTS	Prog. Counter ← SP if Sign=1
RTZ	Prog. Counter ← SP if Zero=1
RZ	Prog. Counter ← SP if Zero=1
INTERRUPT ENABLE/DISABLE INSTRUCTIONS	
EI	(INTE) ← 1
DI	(INTE) ← 0
INPUT/OUTPUT INSTRUCTIONS	
IN	RA ← input device
OUT	RA → output device

APPENDIX B

BASIC SYSTEM SUBROUTINES

This appendix contains a description of basic subroutines in the system. They are used by the system itself, but can also be called by users' programs.

B.1 CURSOR MOVE ROUTINES

B.1.1 ABTAB Auto Tab Backward

Moves the cursor to the left past the next preceding set of protected byte(s), and past all the unprotected bytes until it reaches the left-most byte of that unprotected set. If, at initial cursor scan the next preceding byte is not protected, the scan ends at the left-most byte of the current unprotected set.

Registers affected: All

Call sequence: Call ABTAB

Result: All registers indeterminate; hardware and software cursor registers are updated.

B.1.2 ATAB Auto Tab Forward

Moves the cursor to the right to the first unprotected byte beyond the next set of protected byte(s). If the right scan searches beyond the last displayable character, the cursor is set at true HOME.

Registers affected: All

Call sequence: Call ATAB

Result: All registers indeterminate; hardware and software cursor registers are updated.

B.1.3 BTAB Move Cursor To Next Previous Tab Stop

Cursor is moved from current location to next previous tab stop. If the next previous tab stop on this line is a protected byte, the cursor will scan right to the first available non-protected byte. (The cursor cannot ever move left past a protected byte at the tab stop location by using this function.) If there is no next previous tab stop on this line, or no tab stops at all, the cursor will move to the left-most position of the current line. If already at the left-most position, the cursor will move to the last position of the next preceding line and scan left. In no case will the cursor move past true HOME in its backward scan.

Registers affected: All

Call sequence: Call BTAB

Result: All registers indeterminate; hardware and software cursor registers are updated.

B.1.4 CDOWN Move Cursor One Line Down

Moves cursor (on screen) one line down. If cursor is at bottom-most line, it is moved to top-most line. Cursor will not stop at a protected byte, unless location X'1012' (Protected Cursor Flag) is non-zero.

Registers affected: All

Call sequence: Call CDOWN

Result: All registers indeterminate; hardware cursor registers (X'1000'-X'1001') and software cursor register (FAS) are updated. Where double-page option exists, page adjustment is made automatically.

B.1.5 CLEFT Move Cursor One Position To Left

Moves cursor (on screen) one position to left. If cursor is at left-most position, it will be moved to the right-most position of the preceding line. If cursor is at the left-most position of the first line (true HOME), it will not be moved. Unless location X'1012' (Protected Cursor Flag) is non-zero, cursor will not stop at a 'protected' byte location, but will continue going left. If true Home is found to be protected, cursor will search right and stop at first available not protected byte location.

Registers affected: All

Call sequence: Call CLEFT

Result: All registers indeterminate; both actual, terminal cursor registers and binary cursor location (FAS) are updated. Where double-page option exists, page adjust is also made.

B.1.6 CRIGHT Move Cursor One Position To Right

Moves cursor (on screen) one position to right. If cursor is at right-most position, it will be moved to the left-most position of the next line. If it is at the right-most position of the last available line, it will be moved to the left-most position of the first line. Cursor will automatically bypass any 'protected' byte locations, unless location X'1012' (Protected Cursor Flat) is non-zero.

Registers affected: All

Call sequence: Call CRIGHT

Result: All registers indeterminate; both actual, terminal cursor registers and binary cursor location (FAS) are properly updated. Where double-page option exists, page adjustment to cursor position is also updated.

B.1.7 CUP Move Cursor One Line Up

Moves cursor (on screen) one line up. If cursor is already at top-most line, it will not be moved. Cursor will not stop at a protected byte unless location X'1012' (Protected Cursor Flag) is non-zero. Attempt to move cursor to top-line, if it is protected and location X'1012' is zero, will result in cursor moving to right on top line until a non-protected byte location is found.

Registers affected: All

Call sequence: Call CUP

Result: All registers indeterminate; hardware cursor registers (X'1000'-X'1001') and software cursor register (FAS) are updated. Where double-page option exists, adjustment is made automatically.

B.1.8 HOME Move Cursor To HOME Location

Cursor is moved from current location to left-most position, top-line of current screen image. If that position is protected, cursor will scan right until first unprotected byte is found.

Registers affected: All

Call sequence: Call HOME

Result: All registers indeterminate; hardware and software cursor registers updated.

B.1.9 RETURN Move Cursor To First Position of Next Line

Moves cursor from current position to the left-most position of next line. If current cursor position is on last displayable line, cursor is moved to true HOME. Cursor will not stop at protected byte but will scan right to first non-protected location.

Registers affected: All

Call sequence: Call RETURN

Result: All registers indeterminate; hardware and software cursor registers updated. Where necessary, page is adjusted.

B.1.10 TAB Move Cursor To Next Tab Stop

Cursor is moved from current location to next tab stop to right. If there are no more stops on the line, or if no stops exist, the cursor is moved to the left-most position of the following line. In this case it always acts precisely like RETURN.

Registers affected: All

Call sequence: Call TAB

Result: All registers indeterminate; hardware and software cursor registers are updated.

B.2 VIEWABLE SCREEN ROUTINES

B.2.1 BLANK Blank Screen

That portion of the screen ranging from the value (binary address) in FAS up to, but not including, the value in SAS is blanked. Both protected and unprotected areas are blanked. Indeterminate results can be expected if the value in FAS is not less than that in SAS.

Registers affected: All

Call sequence: After establishing FAS and SAS, Call BLANK.

Result: All bytes in the area specified are changed to blanks (X'20'). Register values at completion are indeterminate.

B.2.2 CLEAR Clear All Viewable Memory

Clears all viewable memory to blanks.

Registers affected: All

Call sequence: Call CLEAR

Result: All registers indeterminate; cursor is repositioned to true Home location. Where necessary, first page is set. All cursor registers are updated.

B.2.3 CMESSA Insert Control Label

The control label at the extreme lower right of the visible screen is loaded by this routine. Any new label can be loaded to that area by specifying the address of the right-most byte of an eight-position label in the register pair H and L, then calling this routine.

Registers affected: All

```
Call sequence:  LBI RH, (label address + 7)
                LBI RL, (label address + 7)

                Call CMESSA.
```

Result: Register values at completion are indeterminate; the new label is displayed.

B.2.4 DELBYT Delete Byte

Deletes byte at cursor location, and moves all data to the right, to the end of the line or to the next protected byte to the left one byte. The last byte on the line or the last unprotected byte in the current field is then blanked.

Registers affected: All

Call sequence: Call DELBYT

Result: All registers indeterminate; cursor is not moved.

B.2.5 DELFD Delete Field

From current cursor location, a scan is made to the left until either the start of line is reached or a protected byte is found. Then the scan is made to the right until either the end of line is reached or a protected byte is found. Within the established range, all unprotected bytes are blanked. The cursor is then repositioned at the left-most unprotected byte of the range.

Registers affected: All

Call sequence: Call DELFLD

Result: All registers indeterminate; hardware and software cursor registers are updated.

B.2.6 DLINE Delete Line

Cursor is moved to extreme left position of current line. All lines from the one following the cursor line to the end of displayable memory are moved up one line. Thus all following moved lines overlay their next preceding line. The last line is blanked at completion of the move. If the cursor is on the last line, this line is blanked.

The DLINE routine automatically incorporates Mercury Move if this option is installed.

Registers affected: All

Call sequence: Call DLINE

Result: All registers indeterminate; hardware and software cursor registers are updated.

B.2.7 EOL Erase To End Of Line

Blanks screen from current cursor location to end of line. Protected bytes are not blanked unless the value at X'1016' is X'80' or greater.

Registers affected: All

Call sequence: Call EOL

Result: All registers indeterminate; cursor remains at same location.

B.2.8 EOS Erase To End Of Screen

Blanks screen from current cursor location to last screen displayable position. Protected bytes are not blanked unless the value at X'1016' is X'80' or greater.

Registers affected: All

Call sequence: Call EOS

Result: All registers indeterminate; cursor remains at same position.

B.2.9 ILINE Insert Blank Line

Cursor is moved to extreme left position of current line. All lines up to and including the cursor line are moved down one line. The cursor line is then blanked. The last displayable line is blanked and the operation proceeds normally. If the cursor is on the last line, that line is blanked.

The ILINE routine automatically incorporates Mercury Move if this option is installed.

Registers affected: All

Call sequence: Call ILINE

Result: All registers indeterminate; hardware and software cursor registers are updated.

B.2.10 INSERT Insert Byte

From the current cursor location, all data to the right, to the end of the line or to the next set of protected byte(s), is moved right one position. If the last position on the line was affected, that last position is changed to a blank. The keyed character is then inserted at the current cursor location. The cursor is advanced automatically.

Registers affected: All

Call sequence: Call INSERT

Result: All registers indeterminate; cursor position advanced one position to right.

B.2.11 NEWFRM Clear Viewable Memory Of All Unprotected Data

All viewable memory, with exception of Control line and protected bytes, is cleared to blanks.

Registers affected: All

Call sequence: Call NEWFRM

Result: All registers indeterminate; cursor is repositioned to true Home. Where necessary, first page is set. All cursor registers are updated.

B.3 MICROCOMPUTER ROUTINES

B.3.1 ADD2 Add Register C to Register Pair D and E

Adds value in register C to value in register pair D and E. Results are placed in register pair D and E.

Registers affected: A, D, and E

Call sequence: After loading 16 bit value in registers D and E, and loading add value in register C, then, Call ADD2.

Result: Register A is indeterminate, registers B, C, H, and L are not changed, and register pair D and E contain the new value.

B.3.2

B.3.3 COMPER Compare Register Pair B and C with Register Pair D and E

The value in register pair B and C is compared against the value in register pair D and E. At completion, Register B holds the High, Low or Equal result.

Registers affected: A, B, C, D, and E.

Call sequence: After loading the values to be compared in register pairs B,C and D,E, then, Call COMPER.

Result: Register A is indeterminate; registers C, D, E, H, and L are unchanged. Register B contains:

> X'02' if value in register pair D and E is numerically greater than value in B and C.
>
> X'01' if value in register pair D and E is less than value in B and C.
>
> X'00' if values are equal.

B.3.4 CONV Convert Binary Cursor Value in FAS to Hardware Cursor Value

Takes the 16-bit binary current cursor value from FAS, and converts it to row and column discontinuous binary value of terminal, and stores the value in the cursor address register.

Registers affected: All

Call sequence: Call CONV

Result: Registers A, B, and C are indeterminate; register pair D and E will contain the new cursor address register value, and paired registers H and L will contain the value X'1001'.

B.3.5 LDCURS Load Cursor

The two-byte contents of the cursor address register (row and column) located at X'1000' and X'1001' is loaded into the register pair D and E.

Registers affected: D, E, H, and L

Call sequence: Call LDCURS

Result: At completion, the register pair D and E contain the contents of memory locations X'1000' and X'1001', respectively. Register pair H and L contain the value X'1000'.

B.3.6 LDFAS Load First Address

The two-byte value at address X'1020' is the absolute binary address of the current cursor location. It is known as FAS, or First Address. Since the binary counterpart of the current cursor location is used so often, FAS has special load and store routines. Calling LDFAS will load the H and L registers with the value in FAS. That value will also be loaded in registers D and E.

Registers affected: D, E, H, and L

Call sequence 1: Call LDFAS

Result: The value in FAS (assume X'189C') will be loaded into registers D and E and also registers H and L, Register D will contain X'18', E will contain X'9C', H will contain X'18', and L will contain X'9C'.

A portion of the LDFAS routine can be used to load registers H and L from almost any addressable memory pair (note exception), by loading the address of the value desired into registers H and L, and then performing a call to location LDFAS+4.

```
Call sequence 2:  LBI  RH,XX    (where XX and YY are the high- and
                  LBI  RL,YY    low-order bytes of the desired
                                location, respectively)
                  Call LDFAS+4
```

Result: Same as in basic LDFAS routine, register pair D and E, H and L are loaded with the value located at XX YY.

Exception: Load results will be indeterminate, and almost certainly wrong, if the memory pair addressed crosses a hexadecimal century boundary, i.e., if the memory address to be loaded ends in X'FF'.

B.3.7 LDSAS Load Second Address

The two-byte space at address X'1022' is used as temporary storage by a significant number of the Basic System Subroutines; it is known as Second Address, or SAS. Calling LDSAS will load the register pair H and L with the value in SAS.

Registers affected: D, E, H, and L

Call sequence: Call LDSAS

Result: The value in SAS will be loaded into register pairs D and E, and H and L.

B.3.8 LDTAS Load Third Address

The two-byte space at address X'1009' is used as temporary storage by several Basic System Subroutines; it is known as Third Address, or TAS. Calling LDTAS will load the register pair H and L with the value in TAS.

Registers affected: D, E, H, and L

Call sequence: Call LDTAS

Result: The value in TAS will be loaded into register pairs D and E, and H and L.

B.3.9 LMOVE Move Data Into RAM, High-Order To Low-Order Addresses

Moves up to 256 bytes from any addressable area in memory to any portion of RAM memory. The move is byte by byte, moving the final byte of the 'from' block to the final byte location of the 'to' block first, then decrementing address and byte count and moving each additional byte until all required bytes have been moved. Register pair D and E must be loaded with the starting location of the 'to' block. Register pair H and L must be loaded with the starting location of the 'from' block. Register C is loaded with the value X'01' to X'FF' to move from 1 to 255 bytes. Loading register C with X'00' will move 256 bytes.

This routine uses the Mercury Move option if it is installed.

Registers affected: All

Call sequence: After loading register pair H and L with the 'from' location, D and E with the 'to' location, and register C with move count, then, Call LMOVE.

Result: Registers A and B are indeterminate; register C is X'00', register pair D and E point to the last byte moved, minus one, of the 'to' area, and register pair H and L point to the last byte minus one of the 'from' area.

B.3.10 RECON Generate Binary Cursor value in FAS From Value In Hardware Cursor Register

Takes the Row/Column current Hardware cursor value and converts it to a 16-bit binary value and stores that value at FAS.

Registers affected: All

Call sequence: Call RECON

Result: Registers A, B, and C are indeterminate; register pair D and E contain the new 16-bit binary value representing the current cursor location, and register pair H and L contain the address of FAS+1.

B.3.11 RMOVE Move Data Into RAM, Low-Order To High-Order Addresses

Moves up to 256 bytes from any addressable area in memory to any portion of RAM memory. The move is byte by byte, moving the first byte of the 'from' block to the first byte location of the 'to' block first, then incrementing address and decrementing byte count and moving each additional byte until all required bytes have been moved. Register pair D and E must be loaded with the starting location of the 'to' block. Register pair H and L must be loaded with the starting location of the 'from' block. Register C is loaded with the value X'01' to X'FF' to move from 1 to 255 bytes. Loading register C with X'00' will cause 256 bytes to be moved.

This routine uses the Mercury Move option if it is installed.

Registers affected: All

Call sequence: After loading register pair H and L with the 'from' location, register pair D and E with the 'to' location, and register C with move count, then Call Call LMOVE.

Result: Register A and B are indeterminate; register C is X'00', register pair D and E point to the last byte plus one of the 'to' area, register pair H and L point to the last byte plus one of the 'from' area.

B.3.12 SMOVE Special Move For Data Going To Control Line

The Control Line (bottom line of screen) has a special function associated with the high-order bit of each byte on that line (Addresses X'1030'-X'107F'). The SMOVE routine inserts data on that line without affecting the high-order bits. In all other respects this move is treated as an 'LMOVE' function. Thus, data moved to the control line must be addressed from the right side rather than the left, etc. See LMOVE (Section B.3.9) for additional information.

Registers affected: All

Call sequence: After loading register pair D and E with the 'to' location, register pair H and L with the 'from' location, and register C with move count, then, Call SMOVE.

Result: Registers A and B are indeterminate; register C is X'00', register pair D and E point to the last byte moved, minus one, of the 'to' area, register pair H and L point to the last byte, minus one, of the 'from' area.

B.3.13 STFAS Store First Address

The two-byte value in register pair H and L is stored at FAS (absolute binary address X'1020').

Registers affected: D, E, H, and L

Call sequence 1: Call STFAS

Result: At completion, FAS contains value that was in register pair H and L, register pair D and E also contains value originally in H and L, and register pair H and L contains the address of FAS+1.

A portion of the STFAS routine can be used to store the value in register pair D and E into FAS.

Call sequence 2: Call STFAS+2

Result: Same as basic STFAS result.

B.3.14 STSAS Store Second Address

The two-byte value in register pair H and L is stored at SAS (absolute binary address X'1022').

Registers affected: D, E, H, and L

Call sequence: Call STSAS

Result: At completion, SAS and the register pair D and E will contain the value initially held in register pair H and L; H and L will contain the address of SAS+1.

B.3.15 STTAS Store Third Address

The two-byte value in register pair D and E is stored at TAS (absolute binary address X'1009').

Registers affected: D, E, H, and L

Call sequence: Call STTAS

Result: At completion, TAS and the register pair D and E will contain the value initially held in register pair D and E; register pair H and L will contain the address of TAS+1.

B.3.16 SUBREG Subtract Register Pair B And C From Register Pair D And E

Subtracts 16-bit value in register pair B and C from 16-bit value in register pair D and E and stores 16-bit result in register pair B and C.

Registers affected: A, B, C, D, and E

Call sequence: After establishing registers B, C, D, and E, Call SUBREG.

Result: Registers D and E will remain as they were just prior to entry to this routine; registers B and C will hold the new result value.

B.3.17 SUBT2 Subtract Register C From Register Pair D and E

Subtracts value in register C from value in register pair D and E. Results are placed in register pair D and E.

Registers affected: A, D, and E

Call sequence: After loading 16-bit value in registers D and E, and loading subtract value in register C, then, Call SUBT2.

Result: Register A is indeterminate, registers B, C, H, and L are not changed, and register pair D and E contain the new value.

B.4 TWO-PAGE OPTION ROUTINES

The routines below operate only if the Page Two Video Display option is installed.

B.4.1 DPAGE Display Page One On Screen

Causes Page One to be displayed by setting the hardware Page Register (X'1005') to X'01'.

Registers affected: H, L

Call sequence: Call DPAGE

Result: Register pair H and L contain X'1005'.

B.4.2 DSCROL Scroll Page Data Downward

Screen view 'window' of data is moved upward, but page data appears to move downward. Hardware Page Register value is decreased by value in location X'1010' (scroll value). Page Register value may not be less than X'01'.

Registers affected: A, B, C, H, and L

Call sequence: Call DSCROL

Result: Registers A, B, and C are indeterminate; register pair H and L contain X'1005'.

B.4.3 UPAGE Display Page Two On Screen

Causes Page Two to be displayed by setting the hardware Page Register (X'1005') to X'19'.

Registers affected: H, L

Call sequence: Call UPAGE

Result: Register pair H and L contain X'1005'.

B.4.4 USCROL Scroll Page Data Upward

Screen view 'window' of data is moved downward, but page data appears to move upward. Hardware Page Register value is increased by value in location X'1010' (scroll value). Page Register value may not exceed X'19'.

Registers affected: A, B, C, H, and L

Call sequence: Call USCROL

Result: Registers A, B, and C are indeterminate; register pair H and L contain X'1005'.

SUBROUTINE EQUATE LIST

	8080	8008
CRIGHT	098D	09A1
CLEFT	0996	09AC
CUP	09AA	09BF
CDOWN	099F	09B4
RETURN	0ABC	0AB9
HOME	0AA9	0AA5
TAB	0B1E	0B21
BTAB	0B23	0B26
ATAB	0C12	0C40
ABTAB	0C3F	0C74
CLEAR	0A89	0A8D
NEWFRM	0AD6	0AD8
EOS	0A59	0A5A
EOL	0A6B	0A6F
BLANK	086B	089A
CMESSA	0853	0871
DELBYT	0CA3	0D0E
DELFLD	0CD9	0D4B
INSERT	0D19	0D97
DLINE	0DDF	0E99
ILINE	0DDB	0E95
LDFAS	0800	0800
STFAS	0809	080A
LDSAS	0812	0814
STSAS	0818	081B
LDTAS	083E	0852
STTAS	0844	0859
LDCURS	0820	0824
BUMPHL		0834
DECHL		0838
SUBREG	087B	08AB
CONV	08A1	08DD
RECON	08C7	0905
COMPER	096E	097E
ADD2	0987	0999
SUBT2	0981	0991
RMOVE	08E6	0927
LMOVE	0902	0941
SMOVE	0952	0954
UPAGE	0ACA	0ACA
DPAGE	0AD0	0AD1
USCROL	0AE2	0AE9
DSCROL	0AE7	0AE4
TEST	0EC9	
DISK	0E80	

APPENDIX C

SYSTEM SUPPORT ROUTINES

C.1 CRT TERMINAL SELF-TEST ROUTINE

This routine will write patterns of X'55' and X'AA' into the entire RAM portion of memory and then read them back to verify that those patterns were correctly stored. After performing this operation 16 times, the CRT will display all displayable characters in twelve modes for the operator to visually verify. Pressing any keyboard key will clear the screen and return to Control mode.

Registers affected: All

Keyboard entry sequence: To execute this routine, press MODE, then ESC.

Result: All registers are indeterminate.

C.2 DISK IPL (INITIAL PROGRAM LOAD)

This routine will load the disk catalog into memory locations X'3000'-X'3800' and then branch to warm start.

Registers affected: A, B, C, H, and L

Keyboard entry sequence: To execute this routine, press MODE, then PAGE ↑.

C.3 ZIM (ZENTEC INTERROGATION MODULE) PROGRAM

The ZIM program provides visual access to the entire system memory. The contents of each location in the memory is displayed on the screen in hexadecimal-coded form and various sections of the memory can be moved on or off the screen with the keyboard cursor controls. In addition, contents of any memory location in the RAM segment can be altered from the keyboard when operating under the control of the ZIM program. Consequently, the ZIM program is useful for programing, program debugging, as well as for maintenance purposes.

Installation of the ZIM program requires that the Page Two Video Display option is present in the system.

The ZIM program is entered from the Control mode by pressing MODE, then CLEAR. A segment of the memory contents will be displayed in hexadecimal form in rows across the screen with the row's starting address displayed in the left-hand column. The 25th line will display "CONTROL". A typical display will look like this:

```
0F80   C2 0A 0A 0A 0A 24 0F B1 2E 10 36 24 F8 07 46 00
0F90   08 2E 10 36 01 C4 97 40 9B 0F 19 E0 46 0C 08 16
0FA0   50 46 99 09 46 7A 09 46 59 08 09 0B EB F4 16 50
0FB0   C7 3C 20 48 9F 9F 46 34 08 11 48 B0 0F 07 CA 2E
0FC0   10 36 14 F9 44 C5 0D 00 46 8D 0A 1E 10 26 00 2E
0FD0   0F 36 DE 16 22 46 27 09 46 D3 0B 44 16 00 01 00
0FE0   FF 00 00 01 00 00 44 00 00 00 00 00 00 00 02 19
                                                         CONTROL
```

With the ZIM program executing, the cursor move keys will allow you to index through the memory. (The cursor appears as a reverse video character.) You can also index through memory by keying in a four-digit hexadecimal memory address, most-significant digit first, and pressing the lower-case "l" key.

To alter the contents of a specific RAM memory location, key in the four-digit hexadecimal memory address and press the SPACE BAR. The hexamecimal characters will replace the current contents of that location.

To execute a program in memory, key in the four-digit hexadecimal memory address and press the lower-case "g" key. The 9003 program will branch to the specified memory location and start executing at that location.

To exit the ZIM program and return to the normal operating program, press the RESET key.

APPENDIX D

ASCII TABLE

HEX	CHARACTER
00	NUL
01	SOH
02	STX
03	ETX
04	EOT
05	ENQ
06	ACK
07	BEL
08	BS
09	HT
0A	LF
0B	VT
0C	FF
0D	CR
0E	SO
0F	SI
10	DLE
11	DC1 (X-ON)
12	DC2 (TAPE)
13	DC3 (X-OFF)
15	DC4
15	NAK
16	SYN
17	ETB
18	CAN
19	EM
1A	SUB
1B	ESC
1C	FS
1D	GS
1E	RS
1F	US
20	SP
21	!
22	"
23	#
24	$
25	%
26	&
27	'
28	(
29	)
2A	*

HEX	CHARACTER
2B	+
2C	,
2D	-
2E	.
2F	/
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9
3A	:
3B	;
3C	<
3D	=
3E	>
3F	?
40	@
41	A
42	B
43	C
44	D
45	E
46	F
47	G
48	H
49	I
4A	J
4B	K
4C	L
4D	M
4E	N
4F	O
50	P
51	Q
52	R
53	S
54	T
55	U

HEX	CHARACTER
56	V
57	W
58	X
59	Y
5A	Z
5B	[
5C	\
5D	]
5E	^ (↑)
5F	_ (←)
60	`
61	a
62	b
63	c
64	d
65	e
66	f
67	g
68	h
69	i
6A	j
6B	k
6C	l
6D	m
6E	n
6F	o
70	p
71	q
72	r
73	s
74	t
75	u
76	v
77	w
78	x
79	y
7A	z
7B	{
7C	|
7D	}
7E	~
7F	DEL (RUB OUT)

APPENDIX E

INPUT PORT ASSIGNMENTS

8008		8080		
PORT	REGISTER A	PORT		
41	00	01	Disk Status	
43	00	03	Printer Status	
45	00	05	Mercury Move Status and Paper Tape Reader Status	
47	00	07	Light Pin Input	
49	00	09	I.D. #	Synchronous TCOM Interface #1
4B	00	0B	Data Status	
4D	00	0D	Data Input	
4F	00	0F	Data	Asynchronous TCOM
41	40	41	Status	
43	40	43	Modem Status	
45	40	45	I.D. #	
47	40	47	Tape Drive #1 flags, status, and word count	
49	40	49	Tape Drive #2 flags, status, and word count	
4B	40	4B	Paper Tape Reader Data	
4D	40	4D	Interrupt Drive #	
4F	40	4F	Interface status, Synchronous TCOM interface #1	
41	80	81	Not Assigned	
43	80	83	Not Assigned	
45	80	85	Not Assigned	
47	80	87	Not Assigned	
49	80	89	I.D. #	Synchronous TCOM Interface #2
4B	80	8B	Data Status	
4D	80	8D	Data Input	
4F	80	8F	Not Assigned	
41	C0	C1	Not Assigned	
43	C0	C3	Not Assigned	
45	C0	C5	Not Assigned	
47	C0	C7	Not Assigned	
49	C0	C9	Not Assigned	
4B	C0	CB	Not Assigned	
4D	C0	CD	Not Assigned	
4F	C0	CF	Interface status, Synchronous TCOM interface #2	

APPENDIX E

OUTPUT PORT ASSIGNMENTS

8008	8080	
PORT	PORT	
51	11	Disk Instruction FIFO
53	13	Disk Go, Start Mecury Move , Interrupt Enable
55	15	Printer Output
57	17	Mecury Move Control Data
59	19	Light Pin Control
5B	1B	Special TCOM Control, TCOM Interrupt Enable
5D	1D	Data Out }
5F	1F	Control Out } Asynchronous TCOM
61	21	Tape Drive #1 instructions
63	23	Tape Drive #2 instructions
65	25	Input/Output Control } Synchronous TCOM
67	27	Data Out } Interface #1
69	29	TCOM Data
6B	2B	TCOM Mode Control
6D	2D	TCOM I/O Control
6F	2F	Not Assigned
71	31	Not Assigned
73	33	Not Assigned
75	35	Input/Output Control } Synchronous TCOM
77	37	Data Out } Interface #2
79	39	Not Assigned
7B	3B	TCOM Data
7D	3D	TCOM Mode Control
7F	3F	TCOM I/O Control
-	3C	Least significant byte, 8080 timer
-	3E	Most significant byte, 8080 timer