LOGICON 2+2

ASSEMBLER MANUAL

LOGICON INC.
1075 CAMINO DEL RIO, SOUTH
SAN DIEGO, CALIFORNIA

15 December 1970

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

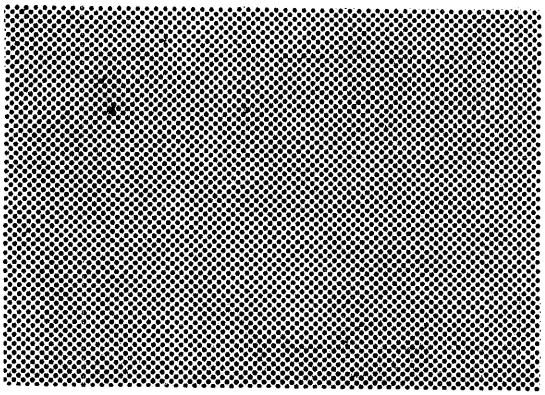
TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

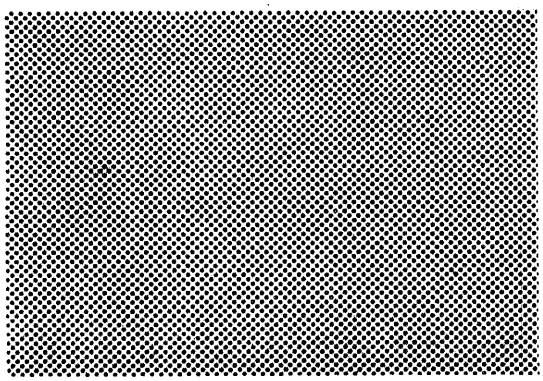


Preface

This Assembler Manual is the basic document for programming on the LOGICON 2+2 System. Essentially, it describes programming-related machine features, the instruction repertoire, and the symbolic machine-language-oriented Macro assembler. This manual is only one of a set of publications for programming the LOGICON 2+2. The user should contact his LOGICON representative for others of the set to obtain all the pertinent and necessary programming information.

This document is addressed to programmers experienced with coding in the environment of a computer installation. It assumes knowledge and experience in the use of address modification with indirection, and other features normally encountered in a computer with a very flexible instruction repertoire -- under control of a master monitor program. It is also assumed that the programmer is familiar with the 2's complement number system as used in a sign-number machine. Note that some of the examples given use SDS-940 instructions. These were taken from the SDS-940 NARP manual as it was felt they would be helpful to the programmer unfamiliar with these features.

It is intended that this manual be updated and added to frequently in order to properly reflect errors, changes, and the required additions leading to the final LOGICON 2+2 system.



I . . .

Number System

GENERAL

The binary system of notation is used throughout the LOGICON 2+2 system.

In the "arithmetic" case of addition, subtraction, and comparison, operands and results are considered as binary numbers in 2's complement form. Subtraction, for example, is carried out internally by adding the 2's complement of the subtrahend.

The assumed location of the binary point has significance only for multiplication and division. For integer arithmetic, the binary point may be assumed to the right of the least-significant bit position (i. e. , to the right of bit position 15); and for fractional arithmetic, the position of the binary point may be assumed to the left of the most-significant position (i. e. , between bit positions 0 and 1).

REPRESENTATION OF INFORMATION

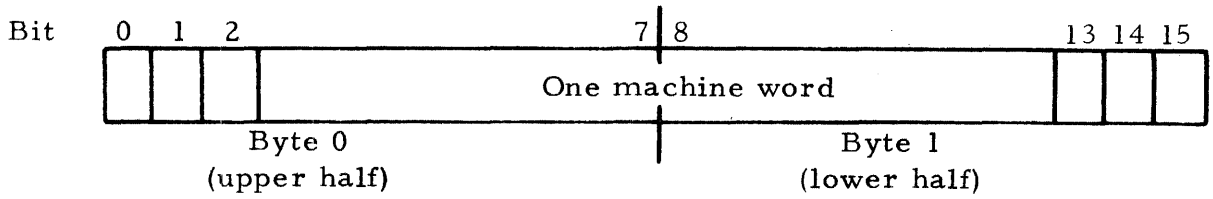
The processor is fundamentally organized to deal with 16-bit grouping of information. Special features are also included for ease of manipulating bits, bytes, and multiple words as groups.

These bit groupings are used by the hardware and software to represent a variety of forms of information.

MACHINE WORD

The machine word consists of 16 bits. The numbering of bit position, character positions, words, etc. increases in the direction of conventional reading from the most- to least-significant.

Data transfers between processor and memory are bit, byte, and word oriented as illustrated below.



ALPHANUMERIC DATA

Alphanumeric data are represented by 8-bit bytes. One machine word contains 2 bytes or characters. The character set used is standard ASCII. Note, however, that for teletype use and access, the 64 character subset indicated in Appendix A is all that is allowed within the assembler.

ONE WORD BINARY INTEGERS

For the "algebraic" group of instructions, results are regarded as signed binary numbers, the leftmost bit being used as a sign bit (a 0 being plus and 1 minus). When the sign is positive all the bits represent the absolute value of the number; and when the sign is negative, they represent the 2's complement of the absolute value of the number. Overflow occurs when the magnitude of a number does not fit within a given word or register. That is, if the carryout of the sign position does not agree with the resultant sign (bit position), overflow has occurred. There are no conditions for underflow. A signed integer ranges from -2^{15} through $2^{15} - 1$.

For the "logical" group of instructions, results are regarded as unsigned, positive binary numbers in the range of 0 through $2^{16} - 1$.

THREE WORD BINARY INTEGERS

The three word integers are sign magnitude, the left most bit of the first word is sign followed by 47 bits of magnitude. The range of three-word extended integers is from $-(2^{47} - 1)$ through $2^{47} - 1 = 140, 737, 488, 355, 327$. Overflow occurs when the magnitude of a number does not fit within the 47 bits.

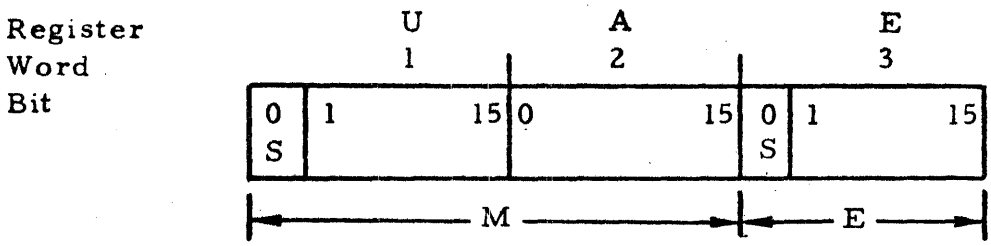
THREE WORD BINARY FLOATING-POINT NUMBERS

The instruction set contains instructions for binary floating-point arithmetic with numbers of two-word precision. The lower word

represents the integral exponent E in 2's complement form, and the upper two words (32 bits) represent the fractional mantissa M in sign magnitude form. The notation for a floating-point number N is:

$$N = M \times 2^{E}$$

The three word format is shown below. S represents the sign bit.

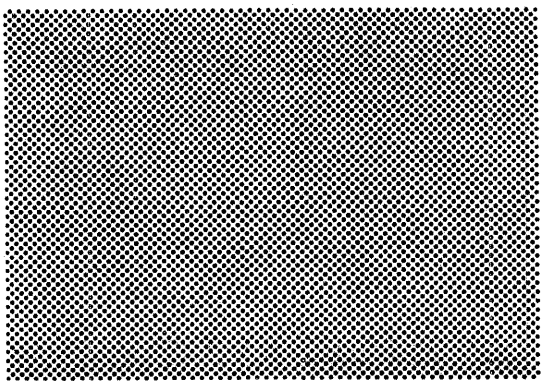


Any number with an absolute value in the range of $.353 \times 10^{-9864}$ through $.708 \times 10^{9864}$ can be represented to more than nine significant decimal digits. For normalized floating-point numbers, the binary point is placed at the left of the most significant bit of the mantissa. Numbers are normalized by shifting the mantissa (and adjusting the exponent) until no leading zeros are present in the mantissa.

To maintain accuracy, the lowest possible exponent (-32768) together with a zero mantissa has been defined as the machine representation of the number zero.

FOUR WORD BINARY FLOATING POINT NUMBERS

These numbers are similar to the 3 word floating point with the exception of one more word of precision. This permits a number with an absolute value in the range of $.353 \times 10^{-9864}$ through $.708 \times 10^{9864}$ to be represented to more than 14 significant decimal digits.



II...

Instructions

GENERAL

Machine instructions are comprised of 10 different format types. In addition, some of these formats may be subdivided into one or more sub-formats. Functionally, the formats are divided as follows:

1. Basic instructions.
2. Miscellaneous instructions.
3. System Calls.
4. Multi-register and register bit instructions.
5. Memory bit instructions.
6. Two-word general instructions.
7. Register operation instructions.
8. Single register shift instructions.
9. Double register shift instructions.
10. Immediate data instructions.

FORMATS

The specific formats (1 thru 10) are shown in Table 2-1, Instruction Formats.

TABLE 2-1. INSTRUCTION FORMATS

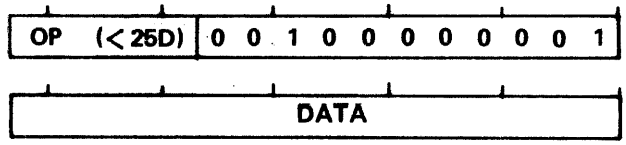
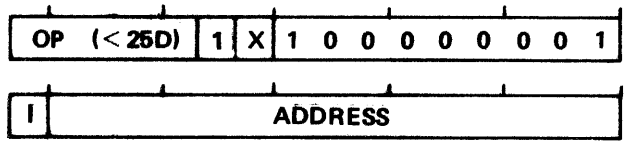
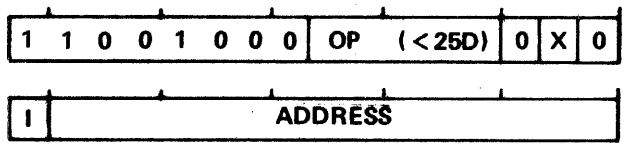
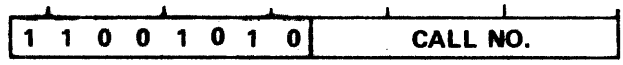
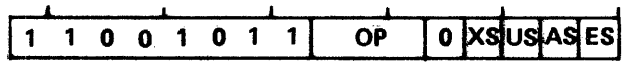
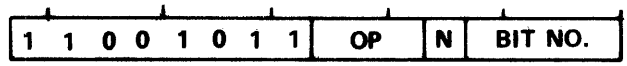
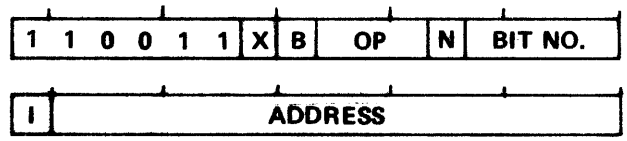
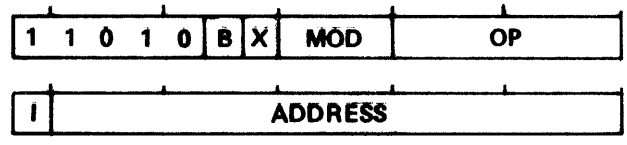
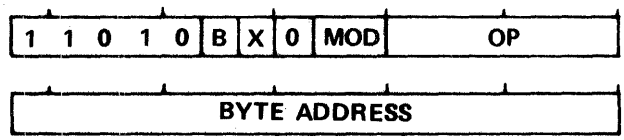
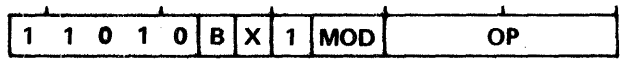
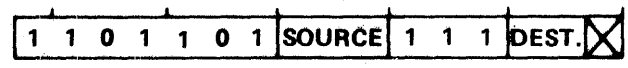
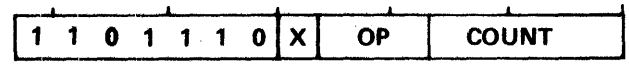
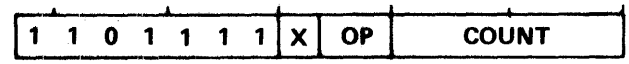
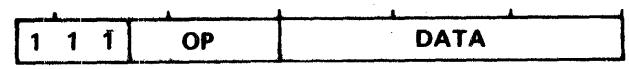
NUMBER	TYPE	FORMAT
1A	BASIC:	
1B	IMMEDIATE FORM OF BASIC	
1C	TWO-WORD FORM OF BASIC, WITHOUT B:	
1D	TWO-WORD FORM OF BASIC, WITH B:	
2	MISCELLANEOUS:	
2B	MISCELLANEOUS:	
3	SYSTEM CALL:	
4A	MULTI-REG. INST :	
4B	REG. BIT INST.:	
5	MEMORY BIT INSTRUCTION:	
6A	NORMAL TWO-WORD:	

TABLE 2-1. INSTRUCTION FORMATS (Cont)

NUMBER	TYPE	FORMAT
6B	TWO-WORD, DIRECT BYTE ADDRESS:	
6C	TWO-WORD, INDIRECT BYTE ADDRESS	 
6D	MULTIPLE LOAD AND STORE:	 
6E	"SPECIFIED MAP" INSTRUCTIONS:	 
6F	"OTHER MEMORY" INSTRUCTIONS:	 
6G	TWO-WORD IMMEDIATE:	 
7A	OPERATE (EXC. RNEG):	
7B	RNEG:	
8	ONE-REG. SHIFT:	
9	TWO-REG. SHIFT:	
10	IMMEDIATE	

The symbols used in Table 2-1 refer to fields of the different instruction formats, where:

OP	=	the operation code, can occur as a 2- to 8-bit field.
I	=	indirect bit, indicating indirect addressing.
X	=	indexing according to contents of X register.
R	=	relative addressing with respect to the P counter and the DISP field (i.e., P + DISP or P +D).
DISP	=	displacement address (D).
DATA	=	9-bit (Sign Extended) or 16-bit (full word) data field.
ADDRESS	=	15-bit word address (< 32768). 16 bits in case of 6B.
CALL NO.	=	specified System Call number.
X_S	=	X-register select.
U_S	=	U-register select.
A_S	=	A-register select.
E_S	=	E-register select.
BIT NO.	=	bit number referenced in register bit instruction.
N	=	bit number; indexing by low order 4 bits of X-register.
B	=	relative addressing with respect to B-register.
BYTEADDRESS	=	16-bit byte address (< 65536).
M	=	specifies user or system map.
Source	=	code indicates source register selected (i.e., X, U, A, etc.)
Destination	=	code indicates destination register selected (i.e., X, U, A, etc.)
Count	=	shift count. (Left shift >0. Right shift 0.)

ABBREVIATIONS AND SYMBOLS

The following abbreviations and symbols are used for description of the machine operation. All registers are 16 bits long, although quantities contained within some of the registers may be less than 16 bits long.

A = Accumulator register

The primary accumulator in the machine. Arithmetic, logical and shift operations are performed directly on this register. It may also be linked with U to form a 32-bit accumulator. A is the low order half of this two-word accumulator.

U = Upper Accumulator register

Some arithmetic, logical, and shift operations are performed directly on the register. In other cases, it is linked with the A register to form a 32-bit accumulator. In these cases, U is the high order half of the two-word accumulator.

E = Exponent register

Contains the exponent in floating point operations. The exponent is expressed as a 2's complement number. This register can be loaded from memory or other registers. It has very limited arithmetic and logical capabilities.

X = Index register

Indexing may be 15-bit word addresses, 16-bit byte addresses, or 16-bit 2's complement displacements. Arithmetic operations on this register do not affect the overflow or carryout status indicators.

P = Program Counter register

This register generally contains the address of the next instruction to be executed. In forming relative addresses in basic instructions it contains the address of the current instruction. The register is 16 bits long but the addresses it contains are all 15-bit quantities.

B = Base of stack register

This register contains the 15-bit address of the base of the stack as seen by the main program or subroutine currently running. Attempts to "pop" the stack beyond this address will result in a stack underflow trap. If the high order bit is set, erroneous results and tests on B may result.

T = Top of stack register

This register contains the 15-bit address of the next word to be pushed into the stack. This address should not be less than the address contained in B nor greater than that contained in L as the result of a stack operation. Checks are made before the stack operation. Note that checks are made only in the appropriate direction (i.e., check for overflow on a "push", and underflow on a "pop").

L = Limit of stack space register

This register contains the 15-bit address of the first word beyond the stack (i.e., the address of the first word the stack is not allowed to occupy). An attempt to "push" the stack beyond this address results in a stack overflow trap.

S = Status register

Bits in this register describe the current status of the machine. Bit positions within the word are defined in Figure 2-1, Status Word Contents.

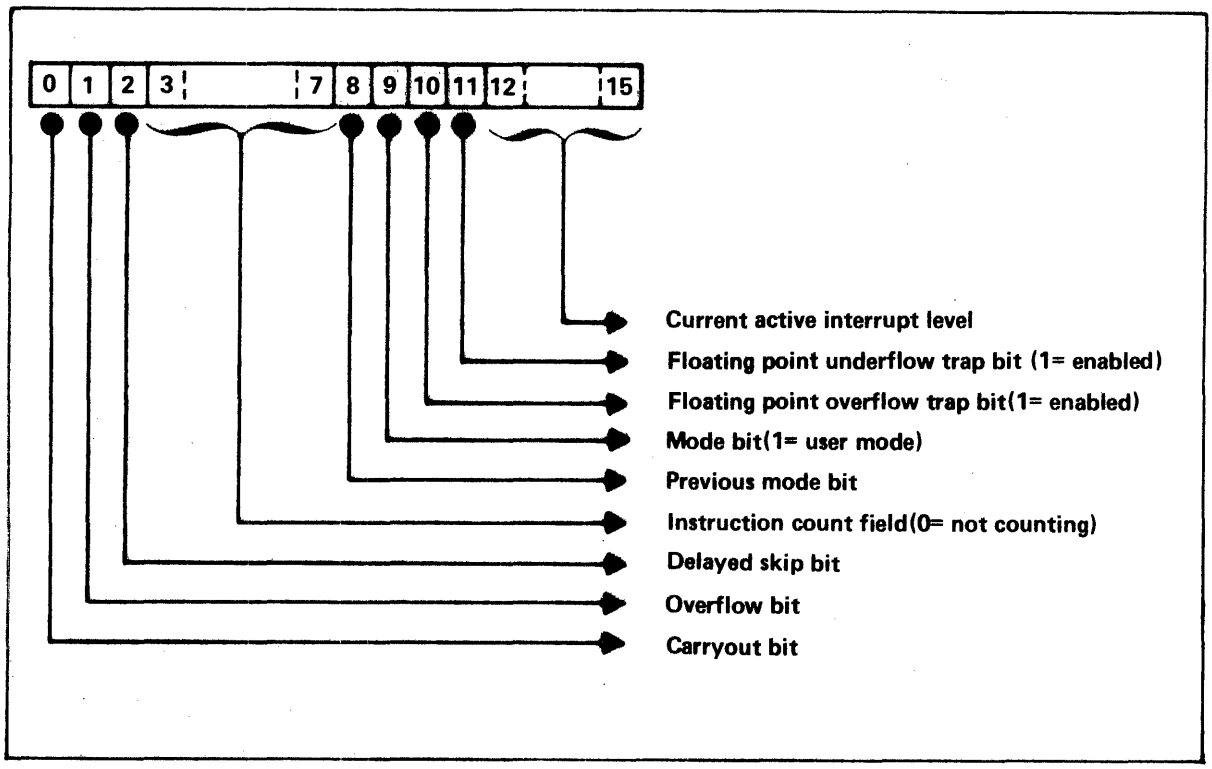


Figure 2-1. Status Word Contents.

Mode - The current mode of the machine.

0 = system mode

1 = user mode.

Previous mode - Defined only if the mode is system.

0 = system mode

1 = user mode.

CO = Carryout

Describes the result of the previous arithmetic instruction.

0 = no carryout occurred

1 = carryout occurred

This bit can be set to either state as a result of an arithmetic operation.

OF = Overflow

Describes the result of all arithmetic and shift instructions on U and A (except register-register instructions) executed since the indicator was last reset.

0 = no overflow has occurred

1 = at least one overflow has occurred.

Floating point Overflow Trap

Indicates whether floating point overflow will result in a trap, or be ignored.

0 = ignored

1 = trap.

Floating point Underflow Trap

Indicates whether floating point underflow will result in a trap, or be ignored.

0 = ignored

1 = trap.

Notation

y	= effective address of instruction
U, A, E, X, B, T, L	= respective registers
(Y)	= contents of effective address Y
Y_B	= effective address of byte Y
A_i	= bit position i of A register
→	= replaces
Ⓧ	= AND
⊕	= OR
⊖	= Exclusive OR
[LC]	= Logical compare
[AC]	= Arithmetic compare

ADDRESS INTERPRETATION

Instructions are translated in a number of ways. The manner of interpretation is governed by the type of instruction. In general, there are the following types of addresses:

- An instruction address, which is the address used for fetching instructions.
- A tentative address, which is the address used for fetching an indirect word.
- An effective address, which is the final address produced by the address modification process. It is the address used for obtaining an operand, for storing a result, or for other special operations during which memory is accessed using the effective address.

ASSEMBLY LANGUAGE PROGRAMMING

The normal operating mode of the LOGICON 2+2 assembler in processing input subprograms is relocatable; that is, each subprogram is handled individually and is assigned memory locations nominally beginning with zero and extending to the upper limit required for that subprogram. Since a job stream can contain many such subprograms, it is apparent that they cannot all be loaded into memory area starting with location zero; they must be loaded into different areas.

Furthermore, they must be movable (relocatable) among the areas. Then for relocatable subprograms, the LOGICON 2+2 assembler provides:

- Delineators identifying each subprogram.
- Symbol linking information.
- Length of each subprogram.
- Relocation control bits for each assembled word.

Label Field

In machine instructions, certain pseudo-operations, and macros, this location may contain a symbol (1 to 6 characters) or may be left blank if no reference is made to the instruction.

Operation Field

The operation field may contain from one to six characters. The group of characters must be a LOGICON 2+2 operation code or pseudo-operation, or a programmer defined macro operation code. Anything else appearing in the operation field is illegal and results in an error message.

An asterisk (*), appearing immediately following the operation code, is a special modifier indicating indirect addressing.

Variable Field

The variable field contains zero, or more subfields separated by the programmer through the use of commas placed between subfields. The number and type of subfields vary depending upon the content of the operation field, machine instruction, pseudo-op, or macro operation.

The subfields within the variable field of machine instructions consist of the address and address modifiers. The address may be any legitimate expression. This is usually the first subfield of the variable field and is separated from modifiers by a comma. Through address modification as directed by the modifier, a program address is defined.

The subfields used with pseudo-operations vary considerably; they are described individually under each pseudo-operation. Subfields used with macro operations are substitutable arguments which, in themselves, may be pseudo-operations, or other macro operations. All of these types of subfields are presented in the discussion on macro operations.

The end of the variable fields is designated by the first blank, semicolon, carriage return, or end-of-file encountered in the variable field.

Any null subfield is interpreted to be zero.

Comments Field

The comments field exists solely for the convenience of the programmer; it plays no part in the assembly process. Programmer comments normally follow the variable field and are separated from that field by at least one blank column or by a semicolon.

A comment may be introduced in several ways:

- An asterisk (*) in column 1.
- A semicolon (;) in any column position.
- By detecting the end of the variable field.

Field Separation

Fields are separated by one or more blanks.

Character Set

All the characters listed in Appendix A have meaning except for '? '. The following classification of character set is useful.

letter:	A-Z
octal digit:	0-7
digit:	0-9
alphanumeric character:	letter or digit
terminator:	, ; blank CR (denotes carriage return)
operator:	! # % & * + - / <=> @ ↑
delimiter:	" $ ' ( ) [ ] . ←

Symbols

Any string of alphanumeric characters beginning with an alphabetic character is a symbol, but only the first six characters distinguish the symbol (thus, Q12345 is the same symbol as Q123456).

Types of Symbols

Symbols are classified as the following types:

- Absolute - if a symbol refers to a specific number.
- Relocatable - if a symbol appears in the label field of an instruction.
- External - if a symbol is considered to be defined external to the subprogram being assembled, and is, furthermore, considered specially by the loader.

Symbol Definitions

Due to the way in which programs are assembled on the Tymshare version of the LOGICON 2+2 assembler, the statement that a symbol or expression is "defined" usually means that it is defined at that instant and not somewhere later in the program. Thus, assuming ALPHA is defined nowhere else, the following

```
BETA    EQU   ALPHA
ALPHA   BSS   3
```

is an error becuase the EQU pseudo-op demands a defined operand and ALPHA is not defined until the next statement. This convention is not strictly adhered to, however, since sometimes the statement XYZ is not defined will mean that XYZ is defined nowhere in the program.

A symbol is defined in one of two ways: by appearing as a label or by being assigned a value with an EQU pseudo-op (or equivalently, by being assigned a value by NARG, NCHR).

- Labels: If a symbol appears in the label field of an instruction (or in the label field of some pseudo-ops) then it is defined with the current value of the current location counter. If the symbol is already defined, either as a label or as an equated symbol, the re-defined error message is typed and the old definition is completely replaced by the new one.
- Equated symbols: These symbols are usually defined by EQU or SET, getting the value of the expression in the operand field of the pseudo-op. This expression must be defined. If the symbol has been previously defined as a label, then the "redefined" error message is typed and the old definition is completely replaced by the new one; if the symbol has already been defined as an equated symbol, then no error message is given, and the old value is replaced by

the new one. Thus, an equated symbol can be defined over and over again, getting a new value each time.

Note that both the SET and EQU pseudo-ops are processed by the EQU pseudo-op when using the Tymshare version of the LOGICON 2+2 assembler. In the final version, both pseudo-ops will be implemented. SET should be used when "redefinition" is desired. The reader should choose the pseudo-op appropriately.

A defined symbol is always local, and may also be external. If a symbol in routine A is to be referred to from routine B, it must be declared external in routine A. This is done in one of the following ways:

Declared external by $: If a label or equated symbol is preceded by a $ when it is defined, then it is declared external.

```
$LABEL1     LDA     ALPHA
LABEL 2     STA     BETA        LABEL2 IS LOCAL ONLY
$GAMMA      EQU     DELTA
```

Declared external by the ENTRY pseudo-op: The symbol in the label field is declared external; it may have already been declared external or may even have a $ preceding it.

If a given symbol is referred to in a program, but is not defined when the END directive is encountered, then it is assumed that this symbol is defined as external in some other package. Whether this is the case cannot be determined until the various packages have been loaded. Such symbols are called "undefined symbols" or "external symbol references." It is possible to perform arithmetic upon them (e.g., LDA UNDEF+1); an expression in post-fix Polish form will be transmitted to the loader.

Expressions

Loosely speaking, an expression is a sequence of constants and symbols connected by operators. Examples:

```
100-2*ABC/(ALPHA+BETA)
GAMMA
F>=Q
```

The value of an expression is obtained by applying the operators to the values of the constants and symbols, evaluating from left to right except when this order is interrupted by the precedence of the operators or by parenthesis "(,)"; the result is interpreted as a 16-bit signed integer. The following table describes the various operators and lists

their precedences (the higher the precedence, the tighter the operator binds its operands):

Operator	Precedence	Comment	
↑	6	exponentiation; exponent must be ≥0.	
*	5	multiplication	
/	5	integer division	
+(u)	4	unary plus	
-(u)	4	negation (arithmetic)	
+	4	addition	
-	4	subtraction	
<	3	less than	
<=	3	less than or equal to	result of operation is 0 if relation is false, otherwise 1
=	3	equal to	
#	3	not equal to	
>=	3	greater than or equal to	
>	3	greater than	
@ (u)	2	logical not	
&	1	logical and	logical operation applied to all 16 bits
!	0	logical or	
%	0	logical exclusive or	

If an expression contains an illegal or undefined symbol, then the entire expression is undefined.

Symbolic Operation Coding and Modifiers

There are several symbolic syntactical elements. They are defined as follows:

$	A label preceded by a dollar sign is declared external.
Label	The label is defined with the current value of the location counter.
Opcode	The opcode must be an instruction, pseudo-operation, or macro operation already defined.
*	If an asterisk follows immediately after the opcode then the indirect bit of the instruction is set.
Operand	The operand can be an expression which may or may not be defined.

Modifiers — The modifiers affect address translation. There are several such characters, and their effect or use is governed by the particular operation code appearing in the operation code field. The symbols allowed are the following:

* — The asterisk is a multi-functional element. It is used as the multiplication operator, designates a comment line, and also indicates indirect addressing. In addition to the above interpretations, it may be used to refer to the location of the instruction in which it appears; for example,

```
A10 JMP *+2
```

is equivalent to

```
A10 JMP A10+2
```

and represents a jump to the second location following the jump instruction. Note, however, that because of the complexity of implementing the LOGICON 2+2 assembler on Tymshare, this element may appear only as the first character of an expression. Thus,

```
A10 JMP *+2
```

is legal, but

```
A10 JMP 2+*
```

is not.

P — the displacement field (D) is computed relative to the P counter (P + D).

```
LDA A (effective address is P + D)
```

is equivalent to

```
LDA A, P (effective address is P + D)
```

The presence of the P modifier forces relative to P counter addressing.

B — the displacement field (D) is computed relative to the contents of the B register (B + D) where D = A-P. Thus,

```
LDA A, B
```

results in the effective address, B + D.

X — the contents of the index register is added to the computed address to complete address translation. Thus,

LDA A, XP results in P + D + X

= — signifies an effective address. If the variable following the equals sign is a literal (i.e., constant), the literal itself is generated as the second word of the instruction. If the variable is a label, the address of the label is generated as the second word of the instruction. The = sign only applies to 1B and 6G formats.

E — is used to indicate an extended or two-word expansion for certain instructions. This is necessary, in many instances due to limitations imposed by the 8-bit displacement field. This field allows direct addressing of locations only within the range of the P counter - $128 \leq D \leq 127$.

When it is necessary to access a memory location outside of this range, two word formats are necessary to accommodate full 15-bit word and/or 16-bit byte addresses.

For example,

LDA A, E

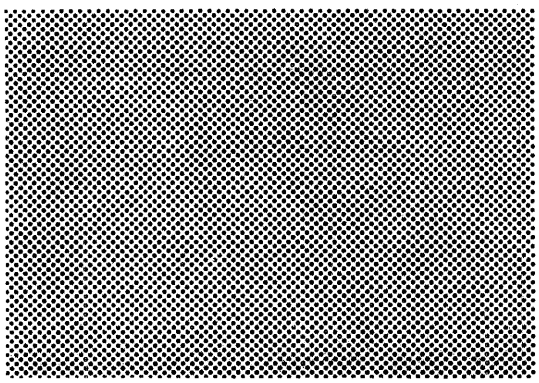
generates (P + 1) as an address -- which means the contents of the second location is used as the effective address for the instruction. The programmer uses the "E" to indicate to the assembler the address A is not in range of the current instruction.

Bit No. — an absolute expression used to select a particular bit involved in a memory- or register-bit type instruction. The bit number is a 4-bit field and may be modified (computed modulo 16).

N — modifier symbol used in memory- and register-bit instruction to indicate that the low order 4 bits of the X register are to be added to the bit number field.

Selected Registers	X, U, A, E

Instructions involving multiple registers or multiple loads and stores, the symbols X, U, A, and E are placed in the variable field to indicate the registers involved in loads, stores, pushes, and pops. The symbols X, U, A, and E represent the X, U, A, and E registers respectively.



III...

Description of Machine Instructions

GENERAL

In describing each instruction, five items may appear underlined preceding the instruction summary. The items are:

Mnemonic Name OpCode (Mod) Format

LOADS AND STORES

LDX Load X 00 1A, B, C, D

(y) → X

Load the contents of memory into the X register.
Modifiers: P, X, E, B, *, =
Refer to Table 3-1 for the allowable address modifiers (and legal combinations) for basic instructions under formats 1A, B, C, D.

1A formats. In general, these are used most frequently to load and store variables where the "address" is within the range of the displacement field, D. That is,

$$-128 \leq \text{address} \leq 127$$

is within the range of the current instruction.

1B formats. These are used, generally, to access constants. It is the closest thing to a literal in the 2+2 assembler. Literal pools, themselves, do not exist.

1C formats. This is the extended or two word form of the basic instruction and is used to access address outside the range of the displacement field.

TABLE 3-1. ADDRESS MODIFIERS FOR BASIC INSTRUCTION FORMATS 1A, B, C, D

where,

* = Indirect Addressing
X = Indexing
P = Relative to P counter
B = Relative to B register
E = Extended address format (2 word form)

and,

- All other combinations of modifiers are illegal
- The modifiers may be specified in any order
- "A" may not be a literal but is the address of the word to be loaded

Format	Example	Address Code	Modifier Codes			Restrictions
			I	X	R	
1A	LDA A	P+D	0	0	1	D = A-P; -128 ≤ D ≤ 127
1A	LDA* A	(P+D)	1	0	1	D = A-P; -128 ≤ D ≤ 127
1A	LDA A,X	D+X or	0	1	0	if -128 ≤ A ≤ 127
		P+D+X	0	1	1	if A < -128 or A > 127
1A	LDA A,XP	P+D+X	0	1	1	D = A-P; -128 ≤ D ≤ 127
1A	LDA* A,X	(P+D)+X	1	1	1	D = A-P; -128 ≤ D ≤ 127
1A	LDA A,B	B+D	0	0	0	D = A-P; -128 ≤ D ≤ 127
1A	LDA* A,B	(B+D)	1	0	0	D = A-P; -128 ≤ D ≤ 127
1A	LDA* A,BX	(B+D)+X	1	1	0	D = A-P; -128 ≤ D ≤ 127
1B	LDA =A	P+1				
1C	LDA A,E	(P+1)				
1C	LDA* A,E	((P+1))				
1C	LDA A,EX	(P+1)+X				
1C	LDA* A,XE	((P+1))+X				
1D	LDA A,BE	B + (P + 1)				
1D	LDA* A,EB	(B + (P + 1))				
1D	LDA A,BXE	B + (P + 1) + X				
1D	LDA* A,EXB	(B + (P + 1)) + X				

1D format. The format is used to force address translation relative to the base of stack or B register. This is needed only if the displacement added to the contents of B is not in the -128 to +127 range, or if both B and X are to be added to the displacement.

LDXEA Load X with Effective Address 04(0) 6A

Y → X

Load the effective address of memory into the X register.

Modifiers: B, X, *

LDXI Load X, Immediate 00 10

LIT9 → X

The 9-bit literal contained in bit positions 7-15 of the instruction is loaded into the X register. The sign of the literal is extended through X_{0-6}.

Modifiers: None.

STX Store X 01 1A, C, D

(X) → y

Store the contents of the X register in memory location y.

Modifiers: P, X, B, E, *

XXM Exchange X and Memory 05(0) 6A

(y) → X; (X) → y

The contents of the X register and memory location y are exchanged.

Modifiers: B, X, *

LDU Load U 02 1A, B, C, D

(y) → U

Load the contents of memory into the U register.

Modifiers: P, X, E, B, *, =

LDUI LoadU, Immediate 01 10

LIT9 → U

The 9-bit literal contained in bit positioning 7-15 of the instruction is loaded into the U register. The sign of the literal is extended through U_{0-6}.

Modifiers: None.

STU Store U 03 1A, C, D

(U) → y

Store the contents of the U register in memory location y.

Modifiers: P, X, B, E, *

LDA Load A 04 1A, B, C, D

(y) → A

Load the contents of memory into the A register.

Modifiers: P, X, E, B, *, =

LDAEA Load A with Effective Address 04(2) 6A

y → A

Load the effective address of memory into the A register.

Modifiers: B, X, *

LDAI Load A, Immediate 02 10

LIT9 → A

The 9-bit literal contained in bit positions 7-15 of the instruction is loaded into the A register. The sign of the literal is extended through A_{0-6}.

Modifiers: None.

STA Store A 05 1A, C, D

(A) → y

Store the contents of the A register in memory location y.

Modifiers: P, X, B, E, *

XAM Exchange A and Memory 05(22) 6A

(y) → A; (A) → y

The contents of the A register and memory location y are exchanged.

Modifiers: B, X, *

LDE Load E 06 1A, B, C, D

(y) → E

Load the contents of memory into the E register.

Modifiers: P, X, B, E, *, =

LDEI Load E, Immediate 03 10

LIT9 → E

The 9-bit literal contained in bit positions 7-15 of the instruction is loaded into the E register. The sign of the literal is extended through E_{0-6}.

Modifiers: None.

STE Store E 07 1A, C, D

(E) → y

Store the contents of the E register in memory location y.

Modifiers: P, X, B, E, *

LDM Load Multiple 01, 41 6D

$(y, \ldots, y+n,\ 0 \leq n \leq 3) \rightarrow X, U, A,$ and/or E

The selected registers, X, U, A, and/or E, are loaded from the contents of memory locations y, y+1, . . . , y+n, where n is determined by the number of registers selected. The variable field of the instruction has three subfields: the selected registers, the memory address, and modifiers, if any. For example:

```
LDM  EU, A, X
```

The contents of A + X is loaded into the U register, and A + X + 1 is loaded into the E register. Registers are always loaded in the order X, U, A, and/or E, no matter how the order is specified in the symbolic instruction.

Modifiers: B, X, *

STM Store Multiple 02, 42 6D

(X, U, A, and/or E) $\rightarrow y, \ldots, y+n,\ 0 \leq n \leq 3$

The contents of the X, U, A, and/or E registers are stored in memory locations y, y+1, . . . , y+n, where n is determined by the number of registers selected.

The variable field for this instruction contains three subfields: the selected registers, the memory address, and modifiers; if any. For example:

```
STM   AX,A,B
```

The contents of the X and A registers are stored in memory locations B+A and B+A+1, respectively. Registers are always stored in the order X, U, A, and/or E, no matter how the order is specified in the symbolic instruction.

Modifiers: B, X, *

PUSHM Push Multiple 0 4A

(X, U, A, and/or E) $\rightarrow$ (T), . . . , (T) + n; $0 \leq n \leq 3$
(T) + n + 1 $\rightarrow$ T

The contents of the selected registers, X, U, A, and/or E are stored in consecutive locations defined by the contents of the top of stack pointer, T. T is then incremented by n + 1 so that the pointer is set to the next available word in the stack. Registers are pushed into the stack in the order X, U, A, and/or E, no matter how the order is specified in the symbolic instruction.

Stack overflow trap if (T) > (L).

Modifiers: None.

POPM Pop Multiple 1 4A

((T) - 1), . . . , ((T) - n - 1) → E, A, U, and/or X;
$0 \leq n \leq 3$; (T) - n - 1 → T

The selected registers, E, A, U, and/or X are loaded from the memory location specified by the top of stack pointer, T.

T is then decremented by n + 1 to reflect the next available word in the stack. Registers are popped from the stack into registers in the order E, A, U, and/or X, regardless of the order specified in the symbolic instruction.

Stack underflow trap if (T) < (B).

Modifiers: None.

PUSHN Push Null 06(0) 6A, G

(T) + (y) → T

The contents of the top of stack pointer, T, is incremented by the contents of the memory location y. There are two forms to the instruction:

6A format – Normal two word form

PUSHN A

The contents of A are added to the T register.

Modifiers: B, X, *

6G format – Immediate or literal form

PUSHN =A

The address or value A is added to the T register.

Modifiers: None.

Stack overflow trap if (T) > (L). Stack underflow trap if (T) < (B).

Modifiers: None.

"Specified map" instructions will be set to select the user map, which is the map in which the address is valid.

MSKM Mask Mode Bit 56(0) 2

The complement of the previous mode bit in the status register is logically "anded" with bit 0 of the X register, and the result left in bit 0 of the X register. That is, if the previous mode bit is 1, bit 0 of the X register is cleared. This is useful for passing addresses back from a system call to a calling routine: if the calling routine is system code, then bit 0 (map select bit) is left alone. If the calling routine is user code, then bit 0 (which is no longer map select in user code) is cleared.

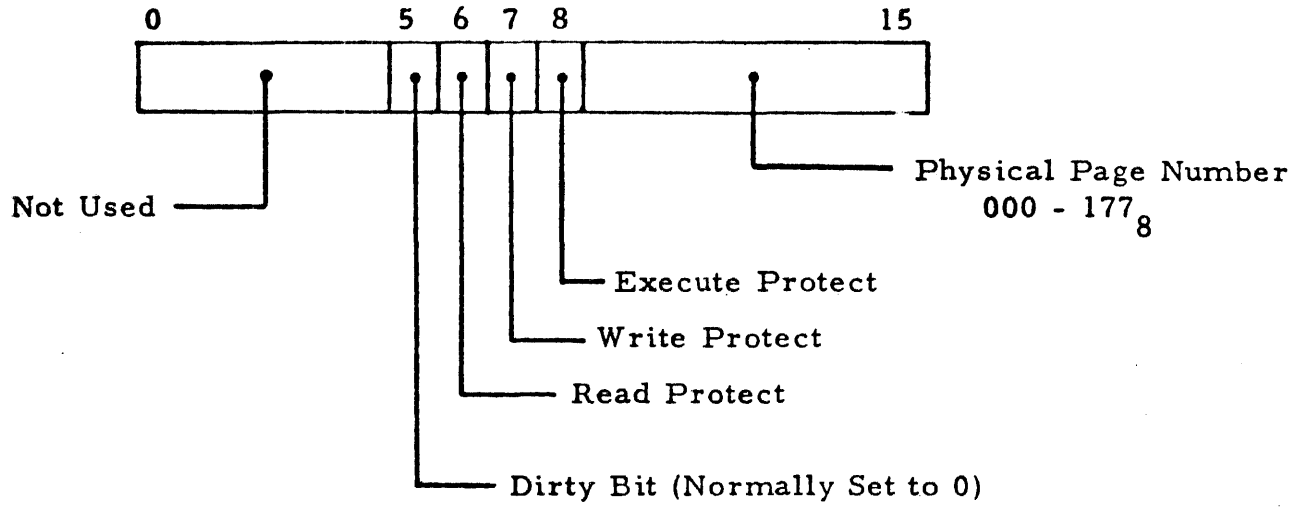
INPUT OUTPUT

LDAC Load A from Console Switches 57 2

The 16 data switches on the programmer's console are interrogated and their state placed in the A register.

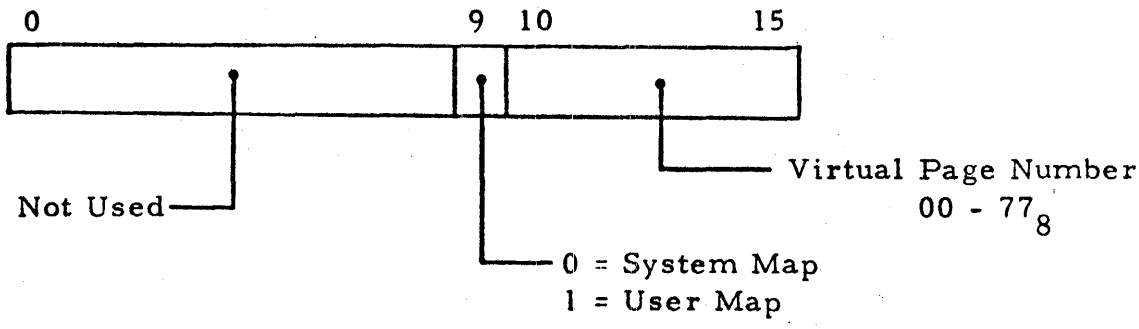
LDMAP Load Map 60 2

A number of consecutive map entries are set from consecutive core locations: the starting map page number is in A (00-77, system map pages 00 to 77; 100-177, user map pages 00 to 77), the starting core location is in X, and the number of map cells to be loaded is in U. The format of the core locations to be transferred to the map is as follows:



LLDB Locate Leading Dirty Bit 70 2

The map entries are inspected, beginning at the page number in X, for a "dirty" bit that is set. If one is found, the next instruction will be skipped and X will contain the page number of the page containing the dirty bit. If none is found, the next instruction will be executed with no skip. The format of the X register when a dirty bit is found is as follows:



SIM SET Interrupt Mask 75 6A, 6G

The operand is logically ANDed with a constant of 137777B (all 1's except for the system stack overflow interrupt mask bit) and placed in the software interrupt mask register. The firmware interrupt mask register is then loaded from the software mask down to, but not including, the bit number specified in bits 12-15 of the current status register. The system stack overflow interrupt is generated by firmware rather than by an external signal, so it is always enabled regardless of the contents of the mask registers.

DOUT Direct Output 75 2

The contents of the X register are placed on the I/O address lines. The contents of the A register are placed on the I/O data lines, and an I/O cycle is initiated. See descriptions of the I/O system for the address codes used to access the various I/O devices.

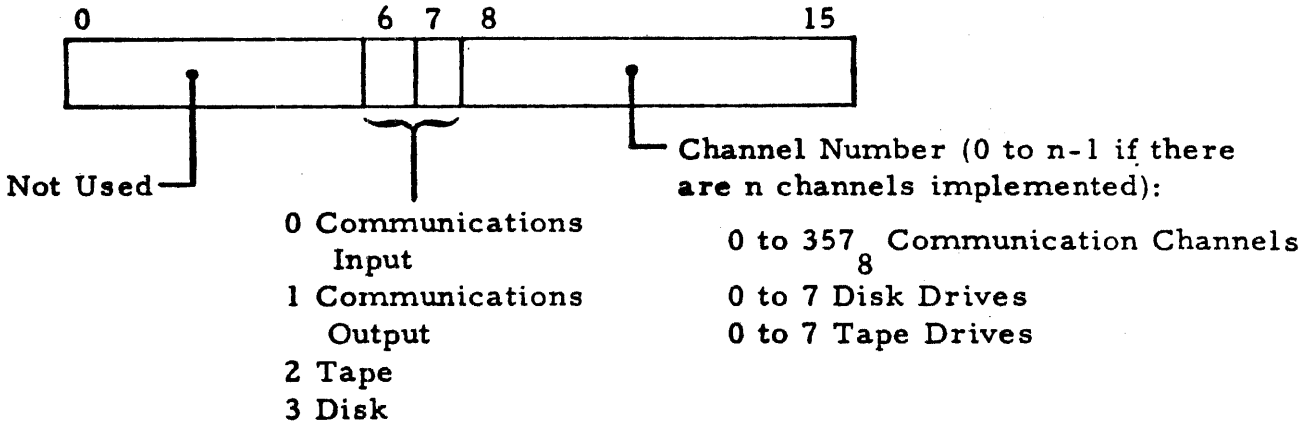
DIN Direct Input 74 2

The contents of the X register are placed on the I/O address lines. The I/O data lines are sampled after an appropriate delay and the data sampled is placed in the A register. See descriptions of the I/O system for the address codes used to access the various I/O devices.

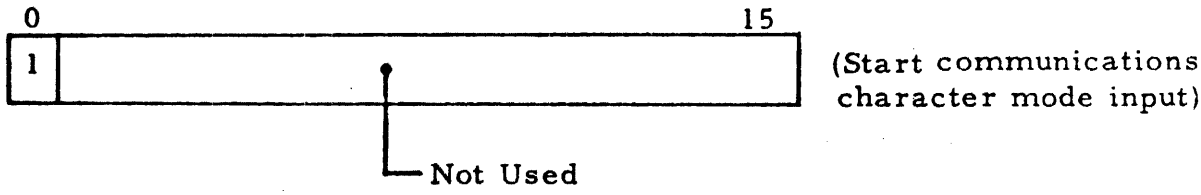
IOC Input/Output Control 71 2

This instruction is used in the Disk, Tape, and Communications I/O subsystems. A channel code is given in X and a function code in A, to control input or output to a tape unit, disk drive or communications channel. The register formats are:

X Register (Channel Code)



A Register (Function Code)



SIL Set Interrupt Lockout 72(0) 2

The firmware interrupt mask is set to zero, locking out all interrupts except system stack overflow. The software mask is unchanged.

RIL Release Interrupt Lockout 73(0) 2

The firmware interrupt mask is loaded from the software mask down to, but not including, the bit number specified in bits 12-15 of the current status register.

SRTRN System Return 2 4A

This instruction is used to return from system calls. It resets the status, program location counter, and stack pointers to the states they had when the system call was entered. It also restores any of the registers X, U, A, and E that are not used for passing parameters. If a return to user mode occurs, then the stack pointers are shifted back to the user stack. Symbolically,

(B)-7 → T
((T)) → S
((T)+1) → X if X flagged in instruction
((T)+2) → U if U flagged in instruction
((T)+3) → A if A flagged in instruction
((T)+4) → E if E flagged in instruction
((T)+5) → P
((T)+6) → B
If (B)>(T), stack underflow trap

If mode is now user (after S is restored), then:

((T)-3) → B
((T)-1) → L
((T)-2) → T

The instruction count mechanism may be activated by setting ((B)-7) properly before executing SRTRN.

IRTRN Interrupt Return 64 2

Return from an interrupt routine, restoring registers to the state they had when the interrupt became active:

(B)-7 → T

((T)) → S

((T)+1) → X

((T)+2) → U

((T)+3) → A

((T)+4) → E

((T)+5) → P

((T)+6) → B

If (B)>(T), stack underflow trap

If mode is now user (after S is restored), then:

((T)-3) → B

((T)-1) → L

((T)-2) → T

The firmware interrupt mask is loaded from the software interrupt mask down to but not including the bit number specified in bits 12-15 of the restored status register. This enables all interrupts of higher priority (lower number) than the one to which the return is made.

HLT Halt 77 2

The processor enters the halt mode, lights the HALT status light on the control panel, and stops executing instructions. The programmers control panel is enabled while the processor is in the halt mode.

CHARACTER INSTRUCTIONS

LDC Load Character 64(0) 6B, C

$$(y_B) \rightarrow A_{8-15};\quad 0 \rightarrow A_{0-7}$$

Load the contents of byte location y_B into bit positions 8-15 of the A register. Bit positions 0-7 of A are set to zero.

Modifiers: B, X, *

STC Store Character 64(1) 6B, C

$$(A_{8-15}) \rightarrow y_B$$

Store bit positions 8-15 of the A register into byte location y_B. The A register is unchanged.

Modifiers: B, X, *

CPRS Compare Strings 052 2

Two byte strings in memory are compared. The byte addresses of the first character of each string must initially be contained in the X and A registers. The number of characters to be compared must be contained in the U register.

A simple ASCII comparison is performed, character by character. Hence, "G" is > "F", and "5" is > "4".

If the string designated in the A register > string in the X register, the next sequential instruction is executed.

If the string designated in the A register = string in the X register, the next sequential instruction is skipped, and execution continues with the following instruction.

If the string designated in the A register < string in the X register, the next two sequential instructions are skipped, and execution continues with the following instruction.

If an equal compare is made, the contents of the X and A registers point one character beyond the last character compared. If an unequal compare is made, the contents of the X and A registers point to the characters found to be unequal.

The CPRS instruction is interruptable and may be restarted.

Modifiers: None.

GFC Get First Character 65(0) 6A

$((y)_B) \rightarrow A_{8-15}$; $0 \rightarrow A_{0-7}$

Memory location y contains a byte address used to access a string. The instruction loads the contents of the specified byte address into bit positions 8-15 of the A register. Bit positions 0-7 of the A register are set to zero.

The byte address referred to above is interpreted as a string pointer. A string is thought of as being defined by two string pointers: a left pointer (LP), and a right pointer (RP). For purposes of utilizing the character instructions, these pointers are thought of as occurring in pairs, left and right, respectively. The pointers are described in more detail in the subsequent discussion of the GFCT instruction. The reader is referred to this section for further explanation.

The GFC instruction simply loads one byte of a string into the A register. No modification of the string pointers occurs. Therefore, repeated execution of a GFC instruction results in repeatedly loading the same byte.

Modifiers: B, X, *

GFCT Get First Character with Test 65(1) 6A

String is tested for null; If not null, $((y)_B) \rightarrow A_{8-15}$; $0 \rightarrow A_{0-7}$

Assume that the memory word pair BA and BA + 1 are memory locations containing byte addresses for two string pointers – the left pointer and right pointer, respectively,

BA	LP
+1	RP

Both the left and right pointers (LP and RP) are 16 bit byte addresses. The left pointer indicates the first byte of the string. The right pointer is set at the last byte of the string plus one.

The length of a designated string is always defined as RP-LP. A string is defined as null if LP ≥ RP. That is, if the left pointer has caught up with or passed beyond the right pointer. All "get" and "insert" character instructions access and modify strings via the left and right string pointers.

The instruction GFCT executes in the following manner. First the string pointers indicated at memory locations y and y + 1 are tested for a null string. If the left pointer is greater than or equal to the right pointer (LP ≥ RP), the string is null, and execution continues with the next sequential instruction. The contents of the A register are unchanged.

If the string is not null, the contents of the byte address specified in memory location y are loaded into bit positions 8-15 of the A register. Bit positions 0-7 of A are set to zero. The next sequential instruction is skipped and execution continues with the following instruction.

Modifiers: B, X, *

GCI Get Character and Increment 65(2) 6A

$((y)_B) \rightarrow A_{8-15}$; $0 \rightarrow A_{0-7}$; $(y) + 1 \rightarrow y$

Memory location y contains a byte address used to access a string. The instruction loads the contents of the specified byte address into bit positions 8-15 of the A register. Bit positions 0-7 of A are set to zero, and the byte address is incremented by one.

Modifiers: B, X, *

GCIT Get Character and Increment with Test 65(3) 6A

String is tested for null; If not null, $((y)_B) \rightarrow A_{8-15}$; $0 \rightarrow A_{0-7}$;

$(y) + 1 \rightarrow y$

The string pointers indicated at memory locations y and y + 1 are tested for a null string. If the left pointer is greater than or equal to the right pointer (LP ≥ RP), the string is null and execution resumes at the next sequential instruction. The contents of A are unchanged, and the left pointer is not incremented.

If the string is not null, the contents of the byte address specified in memory location y are loaded into bit positions 8-15 of the A register.

Bit position 0-7 of A are set to zero, and the byte address left pointer is incremented by one. The next sequential instruction is skipped, and execution resumes with the following instruction.

Modifiers: B, X, *

IFC Insert First Character 65(4) 6A

$(A_{8-15}) \rightarrow (y)_B$

The contents of bit positions 8-15 of the A register replace the contents of the byte address referred to by the contents of memory location y of the instruction.

The byte in the A register is placed in the byte address defined by the left pointer of the string. This instruction may develop a null string since no test concerning the right pointer is made.

Modifiers: B, X, *

IFCT Insert First Character with Test 65(5) 6A

String is tested for null; If not null, $(A_{8-15}) \rightarrow (y)_B$

The string pointers indicated at memory location y and y + 1 are tested for a null string. If the left pointer is greater than or equal to the right pointer ($LP \geq RP$), the string is null and execution resumes at the next sequential instruction. The byte specified by the left pointer is unchanged.

If the string is not null, the contents of bit positions 8-15 of the A register replace the contents of the byte address referred to by the contents of memory location y of the instruction. The next sequential instruction is skipped and execution resumes with the following instruction.

Modifiers: B, X, *

ICI Insert Character and Increment 65(6) 6A

$(A_{8-15}) \rightarrow (y)_B; \ (y) + 1 \rightarrow y$

The contents of bit positions 8-15 of the A register replace the contents of the byte address referred to by the contents of memory location y of the instruction. The byte address (left pointer) is incremented by one.

This instruction may develop a null string since no test concerning the right pointer is made.

Modifiers: B, X, *

ICIT Insert Character and Increment, with Test 65(7) 6A

String is tested for null; if not null, $(A_{8-15}) \rightarrow (y)_B$; $(y) + 1 \rightarrow y$

The string pointers indicated at memory locations y and y + 1 are tested for a null string. If the left pointer is greater than or equal to the right pointer (LP ≥ RP), the string is null and execution continues with the next sequential instruction. The byte specified by the left pointer is unchanged, and the left pointer is not incremented.

If the string is not null, the contents of bit positions 8-15 of the A register replace the contents of the byte address referred to by the contents of memory location y of the instruction. The byte address (left pointer) is incremented by one. The next sequential instruction is skipped and execution continues with the following instruction.

Modifiers: B, X, *

PRIVILEGED INSTRUCTIONS

The machine operates in either system mode or user mode. The system mode is the basic operating mode of the computer. In this mode, all legal operations are permissible. It is assumed that there is a resident monitor that controls and supports the operation of all other programs.

The user mode is the normal problem-solving mode of the computer. In this mode, certain privileged instructions are prohibited. Privileged instructions are those relating to input/output and to changes in the basic control state of the computer. Any attempt by a program to execute a privileged instruction while the computer is in the user mode results in a trap that returns control to the resident monitor. This unconditionally aborts execution of the instruction and may result in aborting the job or program.

A user program cannot directly change the computer mode from user to system. However, the user program can gain direct access to certain privileged program operations by means of the System Call instructions. The operations available through System Calls are established by the resident monitor.

LDAOM Load A from Other Memory 74(0) 6F

The addressed cell in "other memory" is loaded into the A register. If executed in the AP, the addressed cell in CP memory will be obtained. If executed in the CP, the (unmapped) addressed cell in AP memory will be obtained. No protection violation is possible in either case. One level of indirect addressing through own, not other, memory is allowed.

STAOM Store A in Other Memory 74(2) 6F

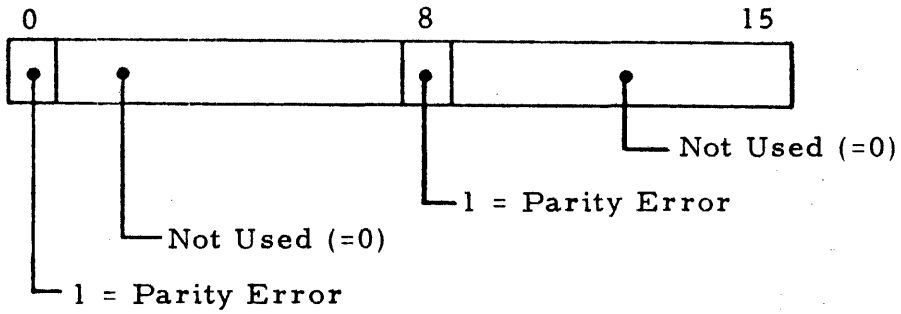
The contents of the A register are stored in the addressed cell in "other memory." If executed in the AP, the contents of A are stored in CP memory. If executed in the CP, the contents of A are stored in AP memory (unmapped). No protection violation is possible. One level of indirect addressing through own, not other, memory is allowed.

TSLOM Test and Set Lock in Other Memory 74(3) 6F

The contents of the addressed cell in other memory are set to 0; if the previous contents of bit 15 of the addressed cell in other memory were 1, skip. Otherwise take a normal return. This instruction is used to interlock critical areas of code between processors. Note that the address is unmapped and that no protection violation is possible. One level of indirect addressing through own, not other, memory is allowed.

LDAOMF Load A From Other Memory With Force 74(1) 6F

The addressed cell in other memory (as in LDAOM) is loaded into the A register. No parity trap is permitted. The contents of the memory status register at the completion of the memory reference are loaded into the U register with the format



One level of indirect addressing through own, not other, memory is allowed.

LDASM Load A through Specified Map 73(0) 6E

The addressed cell is loaded into the A register, using bit 0 of the final address as a "map select" bit. Bit #0=0 means use system map, bit #0=1 means use user map. (AP only)

STASM Store A through Specified Map 73(2) 6E

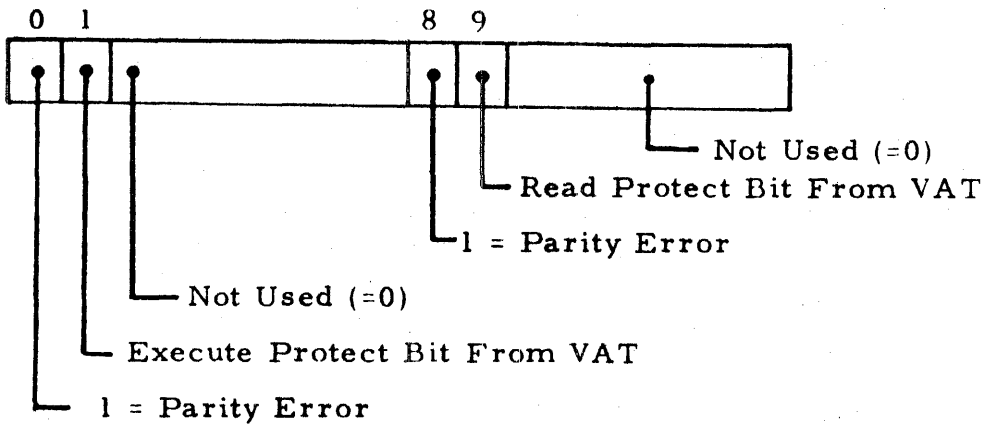
The contents of the A register are stored in the addressed cell, using the specified map as in LDASM. (AP only)

LDXSM Load X through Specified Map 73(3) 6E

The addressed cell is loaded into the X register using the specified map as in LDASM. Then the "specified map" bit is logically "ored" with bit 0 of the X register. This instruction is used for referencing addresses: the address obtained will have a "specified map" bit appended that specifies the map used to read the address. Thus the address will be interpreted through the map through which it was addressed. (AP only)

LDASMF Load A through Specified Map with Force 73(1) 6E

The contents of the addressed cell in AP memory are loaded into the A register using the specified map as in LDASM. No parity or protection traps are allowed. The contents of the memory status register after memory reference are loaded into the U register with the format



MRGM Merge Mode Bits 55(0) 2

The "previous mode" bit of the computer's status register is logically "ored" with bit 0 of the X register, and the result left in bit 0 of the X register. This is useful for passing parameter addresses from one system call to another: if the parameter address came from user code, then the high order bit (map select bit in

POPN Pop Null 06(1) 6A, G

(T) - (y) → T

The contents of the top of stack pointer, T, is decremented by the contents of memory location y. There are two forms to the instruction:

6A format – Normal two word form

POPN A

The contents of A are subtracted from the T register.

Modifiers: B, X, *

6G format – Immediate or literal form

POPN = A

The address or value A is subtracted from the T register.

Modifiers: None.

Stack overflow trap if (T) (L).
Stack underflow trap if (T) (B).

LDB Load B 07 6A, G

(y) → B

The contents of memory are loaded into the base of stack pointer B. There are two forms to the instruction:

6A format – Normal two word form

LDB A

The contents of A are loaded into the B register.

Modifiers: B, X, *

6G format – Immediate or literal form

LDB = A

The address or value A is loaded into the B register.

Modifiers: None.

STB Store B 10(0) 6A

(B) → y

Store the contents of the B register in memory location y.

Modifiers: B, X, *

LDSP Load Stack Pointers 11(0) 6A

(y, y + 1, y + 2) → B, T, L

Load the contents of memory locations y, y + 1, and y + 2 into the stack pointers B, T, and L, respectively.

Stack overflow trap if (T) > (L)
Stack underflow trap if (T) < (B)

Modifiers: B, X, *

LDBTL Load B, T, and L - 11(1) 6A

This instruction is the same as LDSP except that no stack overflow or underflow checks are made.

STSP Store Stack Pointers 10(1) 6A

(B, T, L) → y, y + 1, y + 2

Store the contents of the B, T, and L registers in memory locations y, y + 1, and y + 2.

Modifiers: B, X, *

STZ Store Zeros 12(n) 6A

$0 \rightarrow y, \ldots, y + n - 1; \; 1 \leq n \leq 8$

Words of zeros are placed in memory locations y, . . . , y + n - 1.

The variable field for this instruction contains three subfields: the number n (absolute expression), the beginning memory address, and modifiers, if any.

Modifiers: B, X, *

LSABM Load Sign of A from Bit in Memory 1 5

$(y_i) \rightarrow A_o$; (A_{1-15}) unchanged

where i is a designated bit number.

The sign position of the A register is loaded from the bit position of memory location y, designated by the bit number in the variable field of the instruction.

The variable field of the instruction has three subfields: the bit number (absolute expression), the memory address, and modifiers, if any.

The rules for modifiers X, and N are the same as those defined for the SETBM instruction.

Modifiers: X, B, N, *

SSABM Store Sign of A in Bit in Memory 2 5

$(A_o) \rightarrow y_i$

where i is a designated bit number.

The sign position of the A register is stored in the bit position of memory location y, designated by the bit number in the variable field of the instruction.

The variable field of the instruction contains three subfields: the bit number (absolute expression), the memory address, and modifiers, if any.

The rules for modifiers X, and N are the same as those defined for the SETBM instruction.

Modifiers: X, B, N, *

MOVE Move Word String 003 2

Move (U) words from (X) to (A)

N words, specified in the U register, are moved from a source memory location, specified in the X register, to a destination memory location, specified in the A register. The instruction may be interrupted and restarted without affecting its execution.

Modifiers: None.

CLX Clear X 00 10

This instruction is the same as LDXI 0

CLU Clear U 01 10

This instruction is the same as LDUI 0

CLA Clear A 02 10

This instruction is the same as LDAI 0

CLE Clear E 03 10

This instruction is the same as LDEI 0

LDF Load Floating Point Registers 41(3) 60

This instruction is the same as LDM UAE.

STF Store Floating Point Registers 42(3)

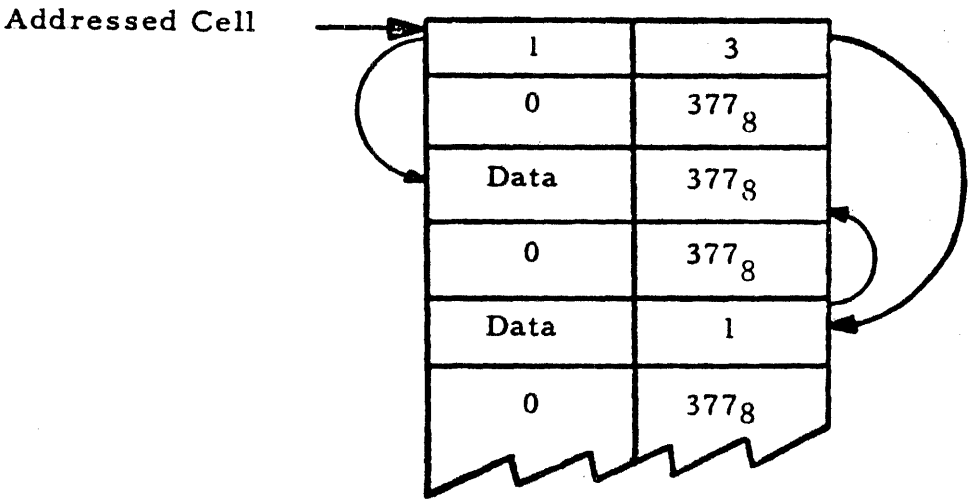
This instruction is the same as STM UAE.

LDD Load Double 41(3) 6D

This instruction is the same as LDM UA.

LINK Link Item Item Into FIFO List 67(0) 6A

The LINK and DLINK instructions address a first in, first out (FIFO) queue with the following structure:



The addressed cell contains start and end pointers for the elements of the queue. The cell following the addressed cell is queue entry number 0. There are a maximum of 377_8 entries in the queue (0 to 376_8). A pointer of 377_8 is used to mean "no pointer". Thus if the queue is empty, the addressed cell will contain 377_8 in each byte. The example shows a queue of two entries, number 3 and number 1.

The LINK instruction operates as follows:

The contents of the X register, $0 \leq (X) \leq 376_8$, are the entry number to be added to the queue. This entry must not already be linked into the queue, (i.e., must have a forward pointer of 377, and must not be pointed to by the queue end pointer). If this test fails then a no skip return is given. Otherwise a new queue entry is added to the end, with the data given in the low order 8 bits of A.

DLINK Remove Item from FIFO List 67(1) 6A

The number of the first item in the queue is placed in X, the data is placed in the lower byte of A, the item is removed from the queue, and a skip return is given. If there are no items in the queue a no skip return is given.

INTER-REGISTER INSTRUCTIONS

RCPY Register Copy 0 7A

(S) → D

where S may be X, U, A, E, B, T, L, or 1
and D may be X, U, A, E, B, T, or L.

The contents of the source register S are loaded into the destination register D. For example,

```
RCPY   1E
```

places the constant 1 in the E register.

Modifiers: None.

RNEG Register Negate — 7B

(S) → D

where S may be X, U, A, E, B, T, L, or 1
and D may only be X, U, A, or E.

The contents of the source register S are negated and the result is loaded into the destination register D. When the source and destination registers are the same the argument need only be specified once. Hence,

```
RNEG   UU
```

is equivalent to RNEG U

Modifiers: None.

RXCH Register Exchange 005-012 2

(S) → D; (D) → S

where S may be X, U, A, or E
and D may be X, U, A, or E.

The contents of the specified registers are exchanged.

For example,

```
RXCH AE
```

exchanges the contents of the A and E registers.

Modifiers: None.

XSA Extend Sign of A 014 2

$(A_0) \rightarrow U$

The sign of A, bit position 0, is extended through the U register. This instruction is very useful in preparing a single word argument for a double word instruction - as in a fixed point divide, etc.

Modifiers: None.

RDS Read Status 015 2

(Status) → A

where (Status) = machine status.

The contents of the (Status) register are placed into the A register. Bit positions and functions are described in Table 2, Status Word Contents.

Modifiers: None.

FIXED-POINT ARITHMETIC

ADX Add to X 12 1A, B, C, D

(X) + (y) → X

The contents of memory location y are added to the contents of the X register.

Modifiers: P, X, E, B, *, =

ADXI Add to X, Immediate 04 10

(X) + LIT0 → X

The 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is added to the contents of the X register.

Modifiers: None.

ADXIS Add to X, Immediate and Skip 05 10

Skip if X = 0; if not set (X) + LIT9 → X

If X is zero, the next sequential instruction is skipped and the following instruction is executed. If X is not zero, the literal in bit positions 7-15 is added to the X register as in the ADXI instruction.

Modifiers: None.

SBX Subtract from X 16 1A, B, C, D

(X) - (y) → X

The contents of memory location y are subtracted from the contents of the X register.

Modifiers: P, X, E, B, *, =

RSBX Reverse Subtract X 15(1) 6A, G

(y) - (X) → X

The contents of the X register are subtracted from the memory location y.

Modifiers: P, X, E, B, *, =

MPX Multiply X 13(0) 6A, G

(X) * (y) → X

The contents of the X register and memory location y are multiplied. The result is placed in the X register.

Modifiers: B, X, *, =

ADU ADD to U 14(0) 6A, G

(U) + (y) → U

The contents of memory location y are added to the contents of the U register. Overflow (OF) may be set. Carryout (CO) is set or reset.

Modifiers: B, X, *, =

ADUI Add to U, Immediate 06 10

(U) + LIT9 → U

The 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is added to the U register. Overflow (OF) may be set. Carryout (CO) is set or reset.

Modifiers: None.

SBU Subtract from U 14(1) 6A, G

(U) - (y) → U

The contents of memory location Y are subtracted from the U register. Overflow (OF) may be set. Carryout (CO) is set or reset.

Modifiers: B, X, *, =

ADA Add to A 10 1A, B, C, D

(A) + (y) → A

The contents of memory location y are added to the A register. Overflow (OF) may be set. Carryout (CO) is set or reset.

Modifiers: P, X, E, B, *, =

ADAI Add to A, Immediate 07 10

(A) + LIT9 → A

The 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is added to the A register. Overflow (OF) may be set. Carryout (CO) is set or reset.

Modifiers: None.

SBA Subtract from A 14 1A, B, C, D

(A) - (y) → A

The contents of memory location y are subtracted from the A register. Overflow (OF) may be set. Carryout (CO) is set or reset.

Modifiers: P, X, E, B, *, =

RSBA Reverse Subtract A 14(2) 6A, G

(y) - (A) → A

The contents of the A register are subtracted from memory location y. The result is placed in the A register. Overflow (OF) may be set. Carryout (CO) is set or reset.

Modifiers: B, X, *, =

MPA Multiply A 13(1) 6A, G

(A) * (y) → U, A

The contents of the A register and memory location y are multiplied. The two word product is placed in the extended accumulator U, A. If

the product does not fit in one register, overflow (OF) is set. That is, if either $(A_0) = 0$ and (U) = 0, or if $(A_0) = 1$, and (U) = 177777.

Modifiers: B, X, *, =

DVUA Divide U and A 16(0) 6A, G

(U, A) / (y) → A; Remainder → U

The contents of the extended accumulator U, A are divided by the contents of memory location y. The quotient is placed in the A register. The remainder is placed in the U register. Overflow (OF) may be set.

Modifiers: B, X, *, =

DVA Divide A 16(1) 6A, G

(A) / (y) → A; Remainder → U

The contents of the A register are divided by the contents of memory location y. The quotient is placed in the A register. The remainder is placed in the U register. Overflow (OF) may be set.

Modifiers: B, X, *, =

RDVA Reverse Divide A 16(3) GA, G

(y) / (A) → A; Remainder → U

The contents of memory location y are divided by the contents of the A register. The quotient is placed in the A register. The remainder is placed in the U register. Overflow (OF) may be set.

Modifiers: B, X, *, =

RADD Register Add 1 7A

(D) + (S) → D

where S may be X, U, A, E, B, T, L, or 1
and D may be X, U, A, E, B, T, or L.

The contents of the source register S are added to the contents of the destination register D. If the source and destination registers are the same, the argument need only be specified once. Hence,

RADD XX

is equivalent to RADD X

Modifiers: None.

RSUB Register Subtract 2 7A

(D) - (S) → D

where S = X, U, A, E, B, T, L, or 1

and D = X, U, A, E, B, T, or L.

The contents of the source register S are subtracted from the contents of the destination register D. If the source and destination registers are the same, the argument need only be specified once. Hence,

RSUB TT

is equivalent to RSUB T

Modifiers: None.

ADDM Add to Memory 17(0) 6A

(y) + (A) → y

The contents of the A register are added to the contents of memory location y. Overflow may be set. Carryout is set or reset. The contents of A are not changed.

Modifiers: B, X, *

SUBM Subtract from Memory 17(1) 6A

(y) - (A) → y

The contents of the A register are subtracted from the contents of memory location y. Overflow may be set. Carryout is set or reset. The contents of A are not changed.

Modifiers: B, X, *

MINC Memory Increment, Skip 20(SC) 6A

(y) + 1 → y; Skip on Condition

The contents of memory location y are incremented by one. The contents of memory location y are compared to zero. If the specified condition is met, the next sequential instruction is skipped and the following instruction is executed.

The variable field of the instruction may have three subfields. They are: The skip condition, the address, and modifiers, if any. For example,

MINC GE, A

The contents of A are increased by one. If the result is "greater than or equal to" (GE) zero, the next sequential instruction is skipped.

Refer to Table 3-2 for the mnemonics and meaning for all skip condition instructions.

TABLE 3-2. CONDITIONS FOR ALL SKIP/JUMP INSTRUCTIONS

Condition	Meaning	Skip/Jump Condition bits 7, 8, 9 of instruction
N	Never Skip (Jump)	000
GT	Greater than	001
EQ	Equal	010
GE	Greater than or Equal to	011
LT	Less than	100
NE	Not Equal	101
LE	Less than or Equal to	110
U	Unconditional Skip (Jump)	111

MDEC Memory Decrement, Skip 21(SC) 6A

(y) - 1 → y; Skip on Condition

The contents of memory location y are decremented by one. If the specified condition is met, the next sequential instruction is skipped and the following instruction executed.

The variable field of instruction may have three subfields. They are: the skip condition, the address, and modifiers, if any. For example,

MDEC EQ, A

The contents of A are decremented by one. If the result is zero, the next sequential instruction is skipped.

Modifiers: B, X, *

TAD Triple Add 26(0) 6A

(U,A,E) + (y, y + 1, y + 2) → U,A,E

The contents of the memory locations y, y + 1, and y + 2 are added to the contents of U,A,E. If the sign magnitude add results in an overflow, the results are right shifted one and the overflow bit is set.

Modifiers: B, X, *

NTAD Negate Triple Add 26(3) 6A

-(U,A,E) - (y, y +1, y + 2) → U,A,E

The contents of U,A,E and (y, y + 1, y + 2) are negated. After negated both the results are added and placed in U, A, E. If overflow occurs, the results are right shifted one and the overflow bit is set.

Modifiers: B, X, *

TSB Triple Subtract 26(1) 6A

(U,A,E) - (y, y +1, y + 2) → U,A,E

The contents of the memory locations (y, y + 1, y + 2) are subtracted from the contents of U,A,E. If overflow occurs, the results are right shifted one and the overflow bit is set.

Modifiers: B, X, *

RTSB Reverse Triple Subtract 26(2) 6A

Y, y + 1, y + 2) - (U,A,E) → U,A,E

The contents of U,A,E are subtracted from (y, y + 1, y + 2) and the results placed in U,A,E. If an overflow occurs, the results are right shifted one and the overflow bit is set.

Modifiers: B, X, *

TMP Triple Multiply 24(1) 6A

(U,A,E) * (y, y + 1, y + 2) → U,A,E

This is a 3 word (sign magnitude) integer multiply. If an overflow occurs, the overflow bit will be set.

Modifiers: B, X, *

TMPF Triple Multiply Fractional 24(3) 6A

(U,A,E) (y, y + 1, y + 2) → U,A,E

This instruction was implemented for use within a 4 word floating point multiply routine. This instruction ignores both input signs and uses the sign bit of the result to return an extra bit of significance. For example:

$$\overline{010}0 * \overline{010}0 = \overline{010}0$$
$$\overline{110}0 * \overline{010}0 = \overline{010}0$$
$$\overline{110}0 * \overline{110}0 = \overline{010}0$$
$$\overline{01111\ldots1} * \overline{01111\ldots1} = 1111\ldots$$

This instruction does not affect overflow or carryout bits.

TDV Triple Divide 25(1) 6A

(U,A,E) / (y, y +1, y + 2) → U,A,E

This is a 3 word (sign magnitude) integer divide. If any number is divided by zero, overflow will occur and the overflow bit will be set.

Modifiers: B, X, *

TDVF Triple Divide Fractional 25(3) 6A

(U,A,E) / (y, y + 1, y + 2) → U,A,E

This instruction was implemented for use within a 4 word floating point divide routine. This instruction ignores both input signs and uses the sign bit of the result to return the most significant bit of the results. For example:

$$\overline{010}0/\overline{010}0 = \overline{10}0$$
$$\overline{110}0/\overline{010}0 = \overline{10}0$$
$$\overline{110}0/\overline{110}0 = \overline{10}0$$
$$0111\ldots1/\overline{010}0 = 1111\ldots1$$
$$\overline{010}0/01111\ldots1 = 01000\ldots$$

This instruction does not affect overflow or carryout bits.

ADAS Add to A, Stack 020 2

((T) - 1) + (A) → A; (T) - 1 → T

The most current item in the stack (pointed to by the top-of-stack pointer T) is added to the contents of the A register. The entry is popped from the stack when the contents of the T register are decremented by 1.

Overflow may be set. Carryout is set or reset. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

SBAS Subtract from A, Stack 021 2

(A) - ((T) - 1) → A; (T) - 1 → T

The most current item in the stack (pointed to by the top-of-stack pointer T) is subtracted from the contents of the A register. The entry is popped from the stack when the contents of the T register are decremented by one.

Overflow may be set. Carryout is set. Carryout is set or rest. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

RSBAS Reverse Subtract from A, Stack 022 2

((T) - 1) - (A) → A; (T) - 1 → T

The contents of the A register are subtracted from the most current item in stack (pointed to by the top-of-stack pointer T). The result is placed in the A register. The entry is popped from the stack when the contents of the T register are decremented by one.

Overflow may be set. Carryout is set or reset. Stack underflow occurs if (T) < (B).

Modifiers: None

MPAS Multiply A, Stack 023 2

(A) * ((T) - 1) → U, A; (T) - 1 → T

The most current item in the stack is multiplied by the contents of the A register. The double word product is placed into the extended accumulator U, A. The entry is popped from the stack when the contents of the T register are decremented by one.

Stack underflow trap occurs if (T) < (B). Overflow is set if either $(A_0) = 0$ and (U) ≠ 0, or, if $(A_0) = 1$ and (U) ≠ 177777. That is, if the product does not fit into one register.

Modifier: None.

DVAS Divide A, Stack 024 2

(A)/((T)-1) → A; (T) - → T; Remainder → U

The contents of the A register are divided by the most current item in the stack. The quotient is placed in the A register, and the remainder is placed in the U register. The entry in the stack is popped when the contents of the T register are decremented by one.

Overflow may be set. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

RDVAS Reverse Divide A, Stack 025 2

((T) -1)/(A) → A; (T) - 1 → T; Remainder → U

The most current item in the stack is divided by the contents of the A register. The quotient is placed in the A register, and the remainder is placed in the U register. The entry in the stack is popped when the contents of the T register are decremented by one.

Overflow may be set. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

ADUS Add to U, Stack 25(0) 2

((T) - 1) + (U) → A; (T) - 1 → T

The most current item in the stack (pointed to by the top-of-stack pointer T) is added to the contents of the U register. The entry is popped from the stack when the contents of the T register are decremented by 1.

Overflow may be set. Carryout is set or reset. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

SBUA Subtract from A, Stack 021 2

(U) - ((T) - 1) → U; (T) - 1 → T

The most current item in the stack (pointed to by the top-of-stack pointer T) is subtracted from the contents of the U register. The entry is popped from the stack when the contents of the T register are decremented by one.

Overflow may be set. Carryout is set. Carryout is set or reset. Stack underflow trap occurs if (T) < (B).

DVUAS Divide UA, Stack 27(0) 2

(U, A) / ((T) - 1) → A Remainder → U (T) - 1 → T

The contents of the U, A registers are divided by the most current item in the stack. The quotient is placed in the A register, and the remainder

is placed in the U register. The entry in the stack is popped when the contents of the T register are decremented by one.

Overflow may be set. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

ADXS Add to X, Stack 020 2

((T) - 1 + (X) → X; (T) - 1 → T

The most current item in the stack (pointed to by the top-of-stack pointer T) is added to the contents of the X register. The entry is popped from the stack when the contents of the T register are decremented by 1.

Overflow may be set. Carryout is set or reset. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

SBXS Subtract from X, Stack 021 2

(X) - ((T) - 1 → A; (T) - 1 → T

The most current item in the stack (pointed to by the top-of-stack pointer T) is subtracted from the contents of the X register. The entry is popped from the stack when the contents of the T register are decremented by one.

Overflow may be set. Carryout is set. Carryout is set or reset. Stack underflow trap occurs if (T) < (B).

Modifiers: None.

RSBXS Reverse Subtract from X, Stack 022 2

((T) - 1) - (X) → X; (T) - 1 → T

The contents of the X register are subtracted from the most current item in stack (pointed to by the top-of-stack pointer T). The result is placed in the X register. The entry is popped from the stack when the contents of the T register are decremented by one.

Overflow may be set. Carryout is set or reset. Stack underflow occurs if (T) < (B).

Modifiers: None.

MPXS Multiply X, Stack 023 2 C

(X) * ((T) - 1) → X; (T) - 1 → T

The most current item in the stack is multiplied by the contents of the X register. The product is placed into the X register. The entry is popped from the stack when the contents of the T register are decremented by one.

Stack underflow trap occurs if (T) < (B).

Modifier: None.

TADS Triple Add Stack 33(0) 2

(U,A,E) + ((T) - 3, (T) - 2, (T) - 1) → U,A,E; (T) - 3 → T

The three most current words in the stack (pointed to by the top of stack pointer T) are added to the three-word accumulator U,A,E. The value is automatically popped from the stack when the T register is decremented by three. If the sign magnitude add results in an overflow, the results are right shifted one and the overflow bit is set.

An overflow or underflow trap may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

NTADS Negative Triple Add Stack 33(3) 2

-(U,A,E) - ((T - 3, (T) - 2, (T) - 1) → U,A,E; (T) - 3 → T

U,A,E and its 3 most current words in the stack are negated, then added together and results placed in U,A,E. The value is automatically popped from the stack when the T register is decremented by three.

If overflow occurs, the results are right shifted one and the overflow bit is set.

Modifiers: None.

TSBS Triple Subtract, Stack 33(1) 2

(U,A,E) - ((T) - 3, (T) - 2, (T) - 1 → U,A,E; (T) - 3 → T

The three most current words in the stack (pointed to by the top-of-stack pointer T) is subtracted from the three-word accumulator U,A,E. The value is automatically popped from the stack when the T register is decremented by three. If overflow occurs, the results are right shifted one and the overflow bit is set.

An overflow or underflow trap may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

RTSBS Reverse Triple Subtract, Stack 33(2) 2

((T) - 3, (T) - 2, (T) - 1) - (U,A,E) → U,A,E; (T) - 3 → T

The quantity in the three-word accumulator U,A,E is subtracted from the three most current words in the stack. The result is placed in U,A,E. The item is automatically popped from the stack when the T register is decremented by three. If an overflow occurs, the results are right shifted one and the overflow bit is set.

An overflow or underflow may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

TMPS Triple Multiply, Stack 31(1) 2

(U,A,E) * ((T) - 3, (T) - 2, (T) -1) → U,A,E; (T) - 3 → T

The 3 word (sign magnitude) integer in the accumulator U,A,E, is multiplied by the three most current words in the stack. The product is placed in U,A,E. Three words are automatically popped from the stack when the T register is decremented by three. If an overflow occurs, the overflow bit will be set.

A stack overflow or underflow may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

TMPFS Triple Multiple Fractional Stack 31(3) 2

(U,A,E) * ((T) - 3, (T) - 2, (T) - 1) → U,A,E; (T) - 3 → T

The quantity in the three-word accumulator U,A,E, is multiplied by the three most current words in the stack. The product is placed in U,A,E. The 3 words are automatically popped from the stack when the T register is decremented by three.

The instruction was implemented for use within a 4 word floating point multiply routine. This instruction ignores both input signs and uses the sign bit of the result to return an extra bit of significance.

$$010\overline{0} * 010\overline{0} = 010\overline{0}$$
$$110\overline{0} * 010\overline{0} = 010\overline{0}$$
$$110\overline{0} * 110\overline{0} = 010\overline{0}$$
$$01111\ldots 1 * 01111\ldots 1 = 1111\ldots$$

This instruction does not affect overflow or carryout bits.

A stock overflow or underflow may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

TDVS Triple Divide, Stack 32(1) 2

(U,A,E) / ((T) - 3, (T) - 2, (T) - 1) → U,A,E; (T) - 3 → T

The 3 word (sign magnitude) integer in the accumulator U,A,E, is divided by the three most current words in the stack. The quotient is placed in U,A,E. The item is automatically popped from the stack when the T register is decremented by three. If any number is divided by zero, overflow will occur and the overflow bit will be set.

A stack overflow or underflow trap may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

TDVFS Triple Divide Fractional, Stack 32(3) 2

(U,A,E) / ((T) - 3, (T) - 2, (T) - 1) → U,A,E; (T) - 3 → T

The quantity in the three-word accumulator U,A,E, is divided by the three most current words in the stack. The quotient is placed in U,A,E. Three words are automatically popped from the stack when the T register is decremented by three.

This instruction was implemented for use within a 4 word floating point divide routine. This instruction ignores both input signs and uses the sign bit of the result to return the most significant bit of the results. For example:

$$\dot{0}10\overline{\quad}0/010\overline{\quad}0 = 10\overline{\quad}0$$
$$110\overline{\quad}0/010\overline{\quad}0 = 10\overline{\quad}0$$
$$110\overline{\quad}0/110\overline{\quad}0 = 10\overline{\quad}0$$
$$01111\ldots 1/010\overline{\quad}0 = 1111\ldots 1$$
$$010\overline{\quad}0/01111\ldots 1 = 01000\ldots$$

This instruction does not affect overflow or carryout bits.

A stack overflow or underflow trap may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

TNEG Triple Negate 53 2

If [(U) ⊕ (A) ⊕ (E)] 0 Then (Uo) → Uo

This instruction changes the sign bit of the U register if U, A, and E are not all equal to zero. If U = 0, A = 0, and E = 0 the instruction is a NOP.

Modifiers: None

FLOATING POINT ARITHMETIC

FAD Floating Add 23(0) 6A

(U, A, E) + (y, y + 1, y + 2) → U, A, E

The floating-point quantity contained in memory locations y, y + 1, and y + 2, is added to the contents of the three-word floating-point accumulator U, A, and E. The result is normalized.

A floating-point overflow or underflow trap may occur.

Modifiers: B, X, *

NFAD Negated Floating Add 23(3) 6A

-(U, A, E) - (y, y + 1, y + 2) → U, A, E

U, A, E and the floating-point quantity contained in memory locations y, y + 1, y + 2, are negated. After negating the two quantities are add and the results placed in U, A, E.

A floating-point overflow or underflow trap may occur.

Modifiers: B, X, *

FSB Floating Subtract 23(1) 6A

(U, A, E) - (y, y + 1, y + 2) → U, A, E

The floating-point quantity contained in memory locations y, y + 1, y + 2, is subtracted from the contents of the three-word accumulator U, A, E. The result is normalized.

A floating-point overflow or underflow trap may occur.

Modifiers: B, X, *

RFSB Reverse Floating Subtract 23(2) 6A

(y, y + 1, y + 2) - (U, A, E) → U, A, E

The floating-point quantity contained in the 3 word accumulator U, A, E, is subtracted from the contents of memory locations y, y + 1, y + 2. The result is normalized and placed into the three-word accumulator U, A, E.

A floating-point overflow or underflow trap may occur.

Modifiers: B, X, *

FMP Floating Multiply 24(0) 6A

(U, A, E) * (y, y + 1, y + 2) → U, A, E

The floating-point quantity contained in the three-word accumulator, U, A, E, is multiplied by the contents of memory location y, y + 1, y + 2. The result is normalized.

A floating-point overflow or underflow trap may occur.

Modifiers: B, X, *

FDV Floating Divide 25(0) 6A

(U, A, E) / (y, y + 1, y + 2) → U, A, E

The floating-point quantity contained in the three-word accumulator U, A, E, is divided by the contents of the memory locations y, y + 1, y + 2. The result is normalized and placed in U, A, E.

A floating-point overflow or underflow or underflow trap may occur.

Modifiers: B, X, *

RFDV Reverse Floating Divide 25(2) 6A

(y, y + 1, y + 2) / (U, A, E) → U, A, E

The contents of memory locations y, y + 1, y + 2 are divided by the floating-point quantity contained in the three-word accumulator U, A, E. The result is normalized and placed in U, A, E.

A floating-point overflow or underflow trap may occur.

Modifiers: B, X, *

FADS Floating Add, Stack 30(0) 2

(U, A, E) + ((T) - 3, (T) - 2, (T) - 1) → U, A, E; (T) - 3 → T

The most current floating-point quantity in the stack (pointed to by the top of stack pointer T) is added to the three-word accumulator U, A, E.

The value is automatically popped from the stack when the T register is decremented by three.

A floating-point overflow or underflow trap may occur. Stack underflow trap occurs if initially (T)-3 < (B).

Modifiers: None.

NFADS Negated Floating Add, Stack 30(3) 2

$$-(U, A, E) - ((T) - 3, (T) - 2, (T) - 1) \rightarrow U, A, E; (T) - 3 \rightarrow T$$

U, A, E and the 3 most current words in the stack are negated, then added together and results placed in U, A, E. The value is automatically popped from the stack when the T register is decremented by three.

A floating-point overflow or underflow trap may occur. Stack underflow trap occurs if initially (T)-3 < (B).

Modifiers: None.

FSBS Floating Subtract, Stack 30(1) 2

$$(U, A, E) - ((T) - 3, (T) - 2, (T) - 1) \rightarrow U, A, E; (T) - 3 \rightarrow T$$

The most current floating-point quantity in the stack (pointed to by the top-of-stack pointer T) is subtracted from the three-word accumulator U, A, E. The value is automatically popped from the stack when the T register is decremented by three.

A floating-point overflow or underflow trap may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

RFSBS Reverse Floating Subtract, Stack 30(2) 2

$$((T) - 3, (T) - 2, (T) - 1) - (U, A, E) \rightarrow U, A, E; (T) - 3 \rightarrow T$$

The floating-point quantity in the three-word accumulator U, A, E is subtracted from the most current floating-point item in the stack. The result is placed in U, A, E. The item is automatically popped from the stack when the T register is decremented by three.

A floating-point overflow or underflow may occur. Stack underflow trap occurs in initially (T) - 3 < (B).

Modifiers: None.

FMPS Floating Multiply, Stack 31(0) 2

(U, A, E) * ((T) - 3, (T) - 2, (T) - 1) → U, A, E; (T) - 3 → T

The floating-point quantity in the three-word accumulator U, A, E, is multiplied by the most current floating point item in the stack. The product is placed in U, A, E. The item is automatically popped from the stack when the T register is decremented by three.

A floating-point overflow or underflow may occur. Stack underflow trap occurs if initially (T) - 3 < (B).

Modifiers: None.

FDVS Floating Divide, Stack 32(0) 2

(U, A, E) / ((T) - 3, (T) - 2, (T) - 1) → U, A, E; (T)-3 → T

The floating-point quantity in the three-word accumulator U, A, E, is divided by the most current floating-point item in the stack. The quotient is placed in the U, A, E. The item is automatically popped from the stack when the T register is decremented by three.

A floating-point overflow or underflow trap may occur. Stack underflow trap occurs if initially (T)-3 < (B).

Modifiers: None.

RFDVS Reverse Floating Divide, Stack 32(2) 2

((T) - 3, (T) - 2, (T) - 1) / (U, A, E) → U, A, E; (T) -3 → T

The most current floating-point item in the stack is divided by the contents of the three-word accumulator U, A, E. The quotient is placed in U. A, E. The item is automatically popped from the stack when the T register is decremented by three.

A floating-point overflow or underflow trap may occur. Stack underflow trap occurs if initially (T)-3 < (B).

Modifiers: None.

FIX Fix a Floating-point Number 041 2

The floating-point quantity contained in the three-word accumulator U, A, E is converted to a fixed-point integer and placed in the A register.

Overflow may be set.

Modifiers: None.

FLOAT Float an integer 042 2

The fixed-point integer in the A register is converted to a floating-point quantity and placed in the three-word accumulator U, A, E.

Modifiers: None

NORM Floating Normalize 043 2

The instruction normalizes the floating-point quantity contained in the three-word accumulator U, A, E.

Modifiers: None.

FNEG Floating Negate 54(0) 2

$$\text{If } [(U) \oplus (A)] \neq 0 \text{ then } (\overline{U_o}) \rightarrow U_o$$

If U and A are not both zero, the sign bit of U will be changed. If U = 0 and A = 0 the instruction is a NOP.

LOGICAL INSTRUCTIONS

In Boolean operations, the operators $\otimes$, $\oplus$, and $\ominus$ have the definitions shown in Table 3-3.

TABLE 3-3 DEFINITIONS OF BOOLEAN OPERATIONS

Operator	Meaning	Definition
$\otimes$	AND; intersection	$0 \otimes 0 = 0$ $0 \otimes 1 = 0$ $1 \otimes 0 = 0$ $1 \otimes 1 = 1$
$\oplus$	OR, inclusive, union	$0 \oplus 0 = 0$ $0 \oplus 1 = 1$ $1 \oplus 0 = 1$ $1 \oplus 1 = 1$
$\ominus$	EXCLUSIVE OR, symmetric difference	$0 \ominus 0 = 0$ $0 \ominus 1 = 1$ $1 \ominus 0 = 1$ $1 \ominus 1 = 0$

ANX AND with X 27(0) 6A, G

(X) ⊗ (y) → X

The contents of memory location y are ANDed with the contents of the X register.

Modifiers: B, X, *, =

ANU AND with U 27(1) 6A, G

(U) ⊗ (y) → U

The contents of memory location y are ANDed with the contents of the U register.

Modifiers: B, X, *, =

ANUI AND with U, Immediate 10 10

(U) ⊗ LIT9 → U

The 9-bit literal contained in bit position 7-15 of the instruction (with sign extended) is ANDed with the contents of the U register.

Modifiers: None.

ANUA And with U and place results in A 27(3) 6A, 6G

(U) ⊗ (y) → A

The contents of memory location y are ANDed with the contents of the U registers (U is left unchanged) and the results are placed in the A registers.

ANA AND with A 20 1A, B, C, D

(A) ⊗ (y) → A

The contents of memory location y are ANDed with the contents of the A register.

Modifiers: P, X, B, E, *, =

ANAI AND with A, Immediate 11 10

(A) ⊗ LIT9 → A

The 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is ANDed with the contents of the A register.

Modifiers: None.

ORA OR with A 22 1A, B, C, D

(A) $\oplus$ (y) → A

The contents of memory location y are ORed with the contents of the A register.

Modifiers: P, X, B, E, *, =

ORAI OR with A, Immediate 12 10

(A) $\oplus$ LIT9 → A

The 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is ORed with the contents of the A register.

Modifiers: None.

XRA EXCLUSIVE OR with A 24 1A, B, C, D

(A) $\ominus$ (y) → A

The contents of memory location y are EXCLUSIVE ORed with the contents of the A register.

Modifiers: P, X, B, E, *, =

XRAI EXCLUSIVE OR with A, Immediate 13 10

(A) $\ominus$ LIT9 → A

The 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is EXCLUSIVE ORed with the contents of the A register.

Modifiers: None.

RAND Register AND 3 7A

(S) $\otimes$ (D) → D

where S may be X, U, A, E, B, T, L, or 1
and D may be X, U, A, E, B, T, or L

The contents of the source register S are ANDed with the contents of the destination register D. For example.

```
RAND EX
```

the E register is ANDed with the X register. The result is placed in the X registers.

Modifiers: None.

SETBA Set Bit in A 3 4B

$1 \rightarrow A_i$

where i is a designated bit number.

The bit number in the A register, designated in the variable field of the instruction, is set to a one. The bit number specified must be an absolute expression. There are two forms of the instruction.

```
SETBA    3
```

sets bit position 3 of the A register to a one; and,

```
SETBA    3,N
```

where N is a modifier indicates the bit number (in the example, 3), is modified by the X register with the result, truncated to four bits, used as the effective bit number.

Modifiers: N

CLRBA Clear Bit in A 4 4B

$0 \rightarrow A_i$

where i is a designated bit number.

The bit position in the A register, designated by the bit number in the variable field of the instruction, is set to zero. The bit number specified must be an absolute expression. There are two forms of the instruction.

```
CLRBA    13
```

sets bit position 13 of the A register to zero; and,

```
CLRBA    13,N
```

where N specifies a modifier indicating the bit number is modified by the X register with the result truncated to four bits, used as the effective bit number.

Modifiers: N

CMPBA Complement Bit in A 5 4B

$$-(A_i) \quad A_i$$

where i is a designated bit number.

The bit position in the A register, designated by the bit number in the variable field of the instruction, is complemented. The bit number specified must be an absolute expression.

Modifiers: N

SETBM Set Bit in Memory 3 5

$$1 \rightarrow y_i$$

where i is a designated bit number.

The bit position of memory location y, designated by the bit number in the variable field of the instruction, is set to a one.

The variable field of the instruction contains three subfields: the bit number (absolute expression), the memory address, and modifiers, if any.

If the instruction is modified by X and not N, the memory address is modified by all 16 bits of the X register.

If the instruction is modified by N and not X, the bit number is modified by the low order four bits of the X register (modulo 16).

If the instruction is modified by both X and N, the memory address is modified by the high order 12 bits of the X register (bit positions 0-11); and the bit number is modified by the low order four bits of the X register (bit positions 12-15). If there is carryout after modifying the bit number (> 15), the carry is added to the memory address calculation for y.

Modifiers: X, B, N, *

CLRBM Clear Bit in Memory 4 5

$$0 \rightarrow y_i$$

where i is a designated bit number.

The bit position of memory location y, designated by the bit number in the variable field of the instruction, is set to zero.

The variable field of the instruction contains 3 subfields: the bit number (absolute expression), the memory address, and modifiers, if any.

If the instruction is modified by X and not N, the memory address is modified by all 16 bits of the X register.

If the instruction is modified by N and not X, the bit number is modified by the low order 4 bits of the X register (modulo 16).

If the instruction is modified by both X and N, the memory address is modified by the high order 12 bits of the X register (bit positions 0-11); and the bit number is modified by the low order four bits of the X register (bit positions 12-15). If there is a carryout after modifying the bit number (1 > 15), the carry is added to the memory address calculation for y.

Modifiers: X, B, N, *

CMPBM Complement Bit in Memory 5 5

$$-(y_i) \rightarrow y_i$$

where i is a designated bit number.

The bit position of memory location y, designated by the bit number in the variable field of the instruction, is complemented.

The variable field of the instruction contains three subfields: the bit number (absolute expression), the memory address, and modifiers, if any.

The rules for modifiers X, and N are the same as those defined for the SETBM instruction.

Modifiers: X, B, N, *

ANAS AND with A, Stack 22(1) 2

$$(A) \otimes ((T) - 1) \rightarrow A; \quad (T) - 1 \rightarrow T$$

The most current item in the stack (pointed to by the top-of-stack pointer T) is ANDed with the contents of the A register. The entry is

automatically popped from the stack when the contents of the T register are decremented by one.

Stack underflow trap occurs if (T) < (B).

Modifiers: None.

ORAS OR with A, Stack 22(2) 2

$$(A) \oplus ((T) - 1) \rightarrow A; \quad (T) - 1) \rightarrow T$$

The most current item in the stack (pointed to by the top-of-stack pointer T) is ORed with the contents of the A register. The entry is automatically popped from the stack when the contents of the T register are decremented by one.

Stack underflow trap occurs if (T) < (B).

Modifiers: None.

XRAS EXCLUSIVE OR with A, Stack 22(3) 2

$$(A) \ominus ((T) - 1) \rightarrow A; \quad (T) - 1 \rightarrow T$$

The most current item in the stack (pointed to by the top-of-stack pointer T) is EXCLUSIVE ORed with the contents of the A register. The entry is automatically popped from the stack when the contents of the T register are decremented by one.

A stack underflow trap occurs if (T) < (B).

Modifiers: None.

ANXS AND with X, Stack 22(0) 2

$$(X) \otimes ((T) - 1) \rightarrow X; \quad (T) - 1 \rightarrow T$$

The most current item in the stack (pointed to by the top-of-stack pointer T) is ANDed with the contents of the X register. The entry is automatically popped from the stack when the contents of the T register are decremented by one.

Stack underflow trap occurs if (T) < (B).

Modifiers: None.

SHIFT INSTRUCTIONS

A note concerning modification of shift counts by indexing. On all shifts, indexing is performed modulo 2^5 (or 2^6 for double register shifts). The sign of the result is bit position 11 (or 10). That is, the 5- or 6-bit shift count is added to the contents of the X register. The result is treated modulo 2^5 (or 2^6).

LLX/LRX Logical Left/Right Shift X 0 8

The contents of the X register are shifted left or right C (C = count) places, with zeros filling vacated bit positions. Bits shifted past bit position 0 (left), or bit position 15 (right) are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0, then left shift. If count < 0, then right shift.

Modifiers: X

ALU/ARU Arithmetic Left/Right Shift U 1 8

The contents of the U register are shifted left or right C (C = count) places. If the shift is to the left, zeros are filled into vacated bit positions on the right. If the shift is to right, the contents of bit position 0 are filled into vacated positions on the left. Bits shifted past bit positions 0 (left) or bit position 15 (right) are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0, then left shift. If count < 0, then right shift.

If the sign bit changes during a left shift, overflow is set.

Modifiers: X

LLU/LRU Logical Left/Right Shift U 2 8

The contents of the U register are shifted left or right C (C = count) places, with zeros filling vacated bit positions. Bits shifted past bit position 0 (left), or bit position 15 (right) are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0, then left shift. If count < 0, then right shift.

Modifiers: X

RLU/RRU Rotate Left/Right U 3 8

The contents of the U register are circular shifted left or right C places. For a rotate left, bits shifted past bit position 0 are placed into bit position 15. For a rotate right, bits shifted past bit position 15 are placed into bit position 0.

No bits are lost. The direction of rotation is determined by the count, C, after indexing. If count > 0, then rotate left. If count < 0, then rotate right.

Modifiers: X

ALA/ARA Arithmetic Left/Right Shift A 4 8

The contents of the A register are shifted left/right C places. If the shift is to the left, zeros are filled into vacated bit positions on the right. If the shift is to the right the contents of bit position 0 are filled into vacated positions on the left. Bits shifted past bit position 0 (left), or bit position 15 (right) are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0, then left shift. If count < 0, then right shift.

If the sign bit changes during a left shift, overflow is set.

Modifiers: X

LLA/LRA Logical Left/Right Shift A 5 8

The contents of the A register are shifted left or right C places, with zeros filling vacated bit positions. Bits shifted past bit position 0 (left), or bit position 15 (right) are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0, then left shift. If count < 0, then right shift.

Modifiers: X

RLA/RRA Rotate Left/Right A 6 8 —

The contents of the A register are circular-shifted left or right C places. For a rotate left, bits shifted past bit position 15 are placed into bit position 0.

No bits are lost. The direction of rotation is determined by the count, C, after indexing. If count > 0, then rotate left. If count < 0, then rotate right.

Modifiers: X

LLUAE/LRUAE Logical Left/Right Shift UAE 0 9

The contents of the three registers U, A, E are shifted left or right C places, with zeros filling vacated bit positions. For a left shift, bits shifted past bit position 0 of E are placed into bit position 15 of the A register; bits shifted past bit position 0 of A are placed in bit position 15 of U, bit shifted past bit position 0 of U are lost. For a right shift, bits shifted past bit position 15 of E are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0, then left shift. If count < 0, then right shift.

Modifiers: X

ALUA/ARUA Arithmetic Left/Shift U, A 1 9

The contents of the double U, A registers are shifted left or right C places. If the shift is to the left, bits shifted past bit position 0 of A are placed into bit position 15 of the U register; bits shifted past bit position 0 of U are lost. If the shift is to the right, bits shifted past bit position 15 of U are placed into bit position 0 of the A register; bits shifted past bit position 0 are filled into vacated positions on the left; bits shifted past bit position 15 of A are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0, then left shift. If count < 0, then right shift.

If the sign bit of U changes during a left shift, overflow is set.

Modifiers: X

LLUA/LRUA Logical Left/Right Shift U, A 2 9

The contents of the double U, A registers are shifted left or right C places, with zeros filling vacated bit positions. For a left shift, bits shifted past bit position of 0 of A are placed into bit position 15 of the U register; bits shifted past bit position 0 of U are lost. For a right shift, bits shifted past bit position 15 of U are placed into bit position 0 of the A register; bits shifted past bit position of A are lost.

The direction of shift is determined by the count, C, after indexing. If count > 0 then left shift. If count < 0, then right shift.

Modifiers: X

RLUA/RRUA Rotate Left/Right U, A 3 9

The contents of the double U, A registers are circular shifted left or right C places. For a rotate left; bits shifted past bit position 0 of U are placed into bit position 15 of the A register; bits shifted past bit position 0 of A are placed into bit position 15 of the U register. For a rotate right; bits shifted past bit position 15 of A are placed into bit position 0 of the U register; bits shifted past bit position 15 of U are placed into bit position 0 of the A register.

No bits are lost. The direction of rotation is determined by the count, C, after indexing. If count > 0, then rotate left. If count < 0, then rotate right.

Modifiers: X

LLO Locate Leading One 044 2

The contents of the A register are searched and shifted to locate a leading one bit.

If the contents of A are zero, the next sequential instruction is executed.

If the contents of A are non-zero, A is shifted left until a one bit is shifted into bit position zero. Bits are zero filled from the right of A. The X register is incremented by the number of shifts that have occurred. Bit position 0 of the A register is set to zero. The next sequential instruction is skipped and execution continues with the following instruction.

Modifiers: None.

COMPARES AND TESTS

SKXEI Skip if X Equal, Immediate 14 10

Skip if (X) = LIT9

If the 9-bit literal contained in bit position 7-15 of the instruction (with sign extended) is equal to the contents of the X register, the next

sequential instruction is skipped. Otherwise, the next instruction is executed.

Modifiers: None.

SKXNI Skip if X Not Equal, Immediate 15 10

Skip if (X) ≠ LIT9

If the 9-bit literal contained in bit positions 7-14 of the instruction (with sign extended) is not equal to the contents of the X register, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

Modifiers: None.

SKAE Skip if A Equal to Memory 26 1A, B, C, D

Skip if (A) = (y)

If the contents of the A register are equal to the contents of memory location y, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

Modifiers: P, X, B, E, *, =

SKAN Skip if A Not Equal to Memory 30 1A, B, C, D

Skip if (A) ≠ (y)

If the contents of the A register are not equal to the contents of memory location y, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

Modifiers: P, X, B, E, *, =

SKAEI Skip if A Equal, Immediate 16 10

Skip if (A) = LIT9

If the 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is equal to the contents of the A register, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

Modifiers: None.

SKANI Skip if A Not Equal, Immediate 17 10

Skip if (A) ≠ LIT9

If the 9-bit literal contained in bit positions 7-15 of the instruction (with sign extended) is not equal to the contents of the A register, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

ACX Arithmetic Compare X 30(SC) 6A, G

(X) [AC] (y); Skip on Condition

The contents of the X register are algebraically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise the next instruction is executed.

The variable field of the instruction may have three subfields: the skip condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

Modifiers: B, X, *, =

ACU Arithmetic Compare U 31(SC) 6A, G

(U) [AC] (y); Skip on Condition

The contents of the U register are algebraically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

Modifiers: B, X, *, =

ACA Arithmetic Compare A 32(SC) 6A, G

(A) [AC] (y); Skip on Condition

The contents of the A register are algebraically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped Otherwise, the next instruction is executed.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

Modifiers: B, X, *, =

ACE Arithmetic Compare E 33(SC) 6A, G

(E) [AC] (y); Skip on Condition

The contents of the E register are algebraically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meanings of all skip conditions.

Modifiers: B, X, *, =

FCP Floating Compare 22(SC) 6A

UAE [AC] (y, y + 1, y + 2) Skip on Condition

The normalized floating point number in UAE is algebraically compared to the normalized floating point number in memory location y, y + 1, y + 2. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

Modifiers: B, X, *, =

FCPS Floating Compare, Stack 36(SC) 2B

U.A.E [AC] (T) - 3, (T) - 2, (T) -1 Skip on Condition

The normalized floating point number in UAE is algebraically compared to the normalized floating point in the Stack. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

The value of the T register is unchanged in either case.

Modifiers: B, X, *, =

LCX Logical Compare X 34(SC) 6A, G

(X) [LC] (y); Skip on Condition

The contents of the X register are logically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed. Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Modifiers: B, X, *, =

LCU Logical Compare U 35(SC) 6A, G

(U) [LC] (y); Skip on Condition

The contents of the U register are logically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed. Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Modifiers: B, X, *, =

LCA Logical Compare A 36(SC) 6A, G

(A) [LC] (y); Skip on Condition

The contents of the A register are logically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped.

Otherwise, the next instruction is executed. Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

The variable field may have three subfields: the skip condition, the address and modifiers, if any.

Modifiers: B, X, *, =

LCE Logical Compare E 37(SC) 6A, G

(E) [LC] (y); Skip on Condition

The contents of the E register are logically compared to the contents of memory location y. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed. Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Modifiers: B, X, *, =

MSK Memory Skip 40(SC) 6A, G

(y) [AC] 0; Skip on Condition

The contents of memory location y are algebraically compared to zero. If the specified skip condition is met, the next sequential instruction is skipped. Otherwise, the next instruction is executed. Refer to Table 3-2 for the mnemonics and meaning of all skip conditions.

The variable field may have three subfields: the skip condition, the address, and modifiers, if any.

Modifiers: B, X, *, =

SKZA Skip if Zero in A 6 4B

Skip if $(A_i) = 0$

where i is a designated bit number.

Bit A_i of the A register is tested. If the bit is a zero, the next sequential instruction is skipped If the bit in A is a one, the next instruction is executed. There are two forms to the instruction.

```
SKZA  5
```

checks bit position 5 of the A register for a zero. If the bit is zero, the skip is executed. Otherwise, no skip. And,

```
SKZA  5,N
```

where N is a modifier indicates the bit number (in the example, 5) is modified by the low order 4 bits of the X register (bit positions 12-15).

Modifiers: N

SKOA Skip if $(A_i) = 1$ 7 4B

where i is a designated bit number.

Bit A_i of the A register is tested. If the bit is a one, the next sequential instruction is skipped. If the bit is a zero, the next instruction is executed. There are two forms to the instruction.

```
SKOA  14
```

skips the next sequential instruction if A_{14} is a one. And,

```
SKOA  0,N
```

modifies the bit number, 0, by the low order 4 bits of the X register to form the effective bit number, i.

Modifiers: N

SKZM Skip if Zero in Memory, y_i 6 5

Skip if $(y_i) = 0$

where i is a designated bit number.

The bit position of memory location y, designated by the bit number in the variable field, is tested. If the bit is zero, the next sequential instruction is skipped. If the bit is one, the next instruction is executed.

The variable field of the instruction contains three subfields: the bit number (absolute expression), the memory address, and modifiers, if any.

If the instruction is modified by X and not N, the memory address is modified by all 16 bits of the X register.

If the instruction is modified by N and not X, the bit number is modified by the low order 4 bits of the X register (modulo 16).

If the instruction is modified by both X and N, the memory address is modified by the high order 12 bits of the X register (bit positions 0-11); and the bit number is modified by the low order 4 bits of the X register (bit positions 12-15). If there is any carryout after modifying the bit number (15), the carry is added to the memory address calculations for y.

Modifiers: X, B, N, *

SKOM Skip if One in Memory, y_i 7 5

Skip if $(y_i) = 1$

where i is a designated bit number.

The bit position of memory location y, designated by the bit number in the variable field, is tested. If the bit is one, the next sequential instruction is skipped. If the bit is zero, the next instruction is executed.

The variable field may contain 3 subfields: the bit number (absolute expression), the memory address, and modifiers, if any.

The rules for modifiers X, and N are the same as those defined for the SKZM instruction.

Modifiers: X, B, N, *

SKNOF Skip if No Overflow 45(0) 2

Skip if (OF) = 0; 0 → OF

The overflow indicator is tested. If overflow is not set ((OF) = 0), then the next sequential instruction is skipped. If overflow is set ((OF) = 1), then the very next instruction is executed. In both cases, the overflow indicator is reset (0 OF)

Modifiers: None

SKNCO Skip if No Carryout 46(0) 2

Skip if (CO) = 0; 0 → CO

The carryout indicator is tested. If carryout is not set ((CO) = 0), then the next sequential instruction is skipped. If carryout is set ((CC) = 1), then the very next instruction is executed. In both cases, the carryout indicator is reset (0 → CO).

Modifiers: None

TSL Test and Set Lock 43(0) 6A

Skip if (y) 15 = 1 0 → y

Bit position 15 of memory location y is tested. If the bit is a one, the next sequential instruction is skipped. If bit position 15 of y is zero, the next instruction is executed. In both cases, memory location y is set to zero.

Modifiers: B, X, *

DSK Delayed Skip 47(0) 2

After the next instruction has been executed, the instruction logically following it will be skipped.

JUMPS

JMP Jump Unconditionally 11 1A, C, D

$y \rightarrow P$

The next instruction executed is determined by the effective address specified in the variable field of the instruction.

Modifiers: P, X, B, E, *

JZE Jump if A Zero 13 1A, C, D

If (A) = 0 then $y \rightarrow P$

If the contents of the A register are zero, control is transferred to the instruction specified by the variable field. Otherwise, the next instruction in sequence is executed.

Modifiers: P, X, B, E, *

JNZ Jump if A Non Zero 15 1A, C, D

If $(A) \neq 0$ then $y \rightarrow P$

If the contents of the A register are non-zero, control is transferred to the instruction specified by the variable field. Otherwise, the next sequential instruction is executed.

Modifiers: P, X, B, E, *

JPL Jump if A Plus 17 1A, C, D

If $(A) \geq 0$ then $y \rightarrow P$

If the contents of the A register are greater than or equal to zero, control is transferred to the instruction specified by the variable field. Otherwise, the next sequential instruction is executed.

Modifiers: P, X, B, E, *

JMI Jump if A Minus 21 1A, C, D

If (A) < 0 then y → P

If the contents of the A register are less than zero, control is transferred to the instruction specified by the variable field. Otherwise, the next sequential instruction is executed.

Modifiers: P, X, B, E, *

XJP X Jump 44(JC) 6A

If (X) condition met, then y → P

The contents of the X register are algebraically compared to zero. If the specified jump condition is met, control is transferred to the instruction specified by the address portion of the variable field. Otherwise, the next sequential instruction is executed.

The variable field of the instruction contains 3 subfields: the jump condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all jump conditions.

Modifiers: B, X, *

UJP U Jump 45(JC) 6A —

If (U) condition is met, then y → P

The contents of the U register are algebraically compared to zero. If the specified jump condition is met, control is transferred to the instruction specified by the address portion of the variable field. Otherwise, the next sequential instruction is executed.

The variable field of the instruction contains 3 subfields: the jump condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all jump conditions.

Modifiers: B, X, *

AJP A Jump 46(JC) 6A –

If (A) condition met, then y → P

The contents of the A register are algebraically compared to zero. If the specified jump condition is met, control is transferred to the instruction specified by the address portion of the variable field. Otherwise, the next sequential instruction is executed.

The variable field of the instruction contains 3 subfields: the jump conditions, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all jump conditions.

Modifiers: B, X, *

EJP E Jump 47(JC) 6A –

If (E) condition met, then y → P

The contents of the E register are algebraically compared to zero. If the specified jump condition is met control is transferred to the instruction specified by the address portion of the variable field. Otherwise, the next sequential instruction is executed.

The variable field of the instruction contains 3 subfields: the jump condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all jump conditions.

Modifiers: B, X, *

TJP Triple Jump, UA 50(JC) 6A –

If (U), A, E) condition met, then y → P

The contents of the triple U, A, E registers are algebraically compared to zero. If the specified condition is met, control is transferred to the instruction specified by the address portion of the variable field. Otherwise, the next sequential instruction is executed.

The variable field of instruction contains 3 subfields: the jump condition, the address, and modifiers, if any.

Refer to Table 3-2 for the mnemonics and meaning of all jump conditions.

Modifiers: B, X, *

IJXN Jump if Non Zero, Increment X 25 1A, C, D

If (X) ≠ 0 then y → P; (X) + 1 → X

If X is non-zero, control is transferred to the instruction specified by the variable field. Otherwise, the contents of the X register are incremented by one and the next sequential instruction is executed.

Modifiers: P, X, B, E, *

DJXN Jump if Non Zero, Decrement X 27 1A, C, D

If (X) ≠ 0 then y → P; (X) - 1 → X

If X is non-zero, then control is transferred to the instruction specified by the variable field. Otherwise, the contents of the X register are decremented by one and the next sequential instruction is executed.

Modifiers: P, X, B, E, *

IJMP Indirect Jump 55(0) 6A

(y) → P

The address of the next instruction to be executed is determined by the effective address specified in the variable field of the instruction.

Modifiers: B, X, *

SUBROUTINE AND SYSTEM LINKAGE

JSPX Jump, Store P in X 23 1A, C, D

(P) + 1 → X; y → P

The contents of the P register plus one, are stored in the X register. Control is transferred to the instruction specified by the variable field.

Modifiers: P, X, B, E, *

JSPM Jump, Store P in Memory 56(7) 6A

$$(P) + 1 \rightarrow (y); \quad y + 1 \rightarrow P$$

The contents of the P register plus one, are stored in the location specified by memory location y. Control is transferred to memory location y + 1.

Modifiers: B, X, *

CALL Subroutine Call Linkage 63(7) 6A

$$(P) + 1 \rightarrow (T); (B) \rightarrow (T) + 1; (T) + 2 \rightarrow B \rightarrow T; \quad y \rightarrow P$$

The contents of the P register plus one, is stored in the location specified by the top of the stack pointer T. The contents of the base of stack B register is stored in the location specified by the T register plus one. The address, specified by the contents of the T register plus two, is stored in both the B and T registers. Control is transferred to the instruction specified by the variable field.

In this way, the return location from the subroutine, and the original values of the B and T registers are preserved. Various subroutine stack pointers are automatically protected from routine to routine.

A stack overflow trap occurs if $(T) \geq (L)$.

Modifiers: B, X, *

RTRN Subroutine Return Linkage 050 2

$$(B) - 2 \rightarrow T; \quad ((T) + 1) \rightarrow B; \quad ((T)) \rightarrow P$$

The contents of the base of stack B register minus two, is stored in the T register. This restores the top of stack T to the value at the time of the subroutine CALL. The contents of the value contained in the top of stack pointer T plus one, is stored in the B register. This restores the value of the B register to the value at the time of the subroutine CALL.

Control is transferred to the memory location contained in the T register. This effectively returns control to the instruction following the subroutine CALL with the values of the B and T registers restored.

A stack underflow track occurs if $(T) < (B)$.

Modifiers: None.

SCALL System Call – 3 –

System Calls are those calls concerned with defining, determining, and manipulating the system environment of a process.

There are limited number of System Calls simulated by Tymshare. These calls are primarily used to:

- open and close files
- I/O to and from files, TTY
- manipulate edited lines

Most of the calls will have counterparts in the final version of the LOGICON 2 + 2 assembler. They are identified in Table 3-4 with mnemonics. Some calls are unique to the Tymshare version, and, as such, no mnemonics are defined for them.

There are two allowable forms to the System Call instruction. The general form is:

(label) SCALL (System Call no.)

Note that System Call numbers are octal. All existing (simulated) calls can be made via this form. Those calls having a unique mnemonic may be accessed by the name as:

(label) (System Call Name)

For example, the Get Edited Line System Call, GEL, may be written

(label) SCALL 204B

or, (label) GEL

Table 3-4 shows all of the simulated System Calls, characteristics, and any anomalies. For a detailed description of each of the System Calls, the user is referred to the LOGICON 2+2 MONITOR SPECIFICATION.

Modifiers: None.

IJSPX Indirect Jump, Store P in X 55(1) 6A

$(P) + 1 \rightarrow X;\ (y) \rightarrow P$

The contents of the P register plus one, is stored in the X register. Control is transferred to the address specified by the variable field.

Modifiers: B, X, *

IJSPM Indirect Jump, Store P in Memory 55(2) 6A

$(P) + 1 \rightarrow ((y));\quad (y) + 1 \rightarrow P$

The contents of the P register plus one, is stored at the address specified by memory location y. Control is transferred to the location following the address specified by the variable field.

Modifiers: B, X, *

ICALL Indirect Subroutine Call Linkage 55(3) 6A

$(P) + 1 \rightarrow (T);\ (B) \rightarrow (T) + 1;\ (T) + 2 \rightarrow B \rightarrow T;\ (y) \rightarrow P$

The contents of the P register plus one, is stored in the location specified by the top of the stack pointer T. The contents of the base of stack B register is stored in the location specified by the T register plus one. The address, specified by the contents of the T register plus two, is stored in both the B and T registers. Control is transferred to the address specified by the variable field.

In this way, the return location from the subroutine, and the original values of the B and T registers are preserved. Various subroutine stack pointers are automatically protected from routine to routine.

A stack overflow trap occurs if $(T) \geq (L)$.

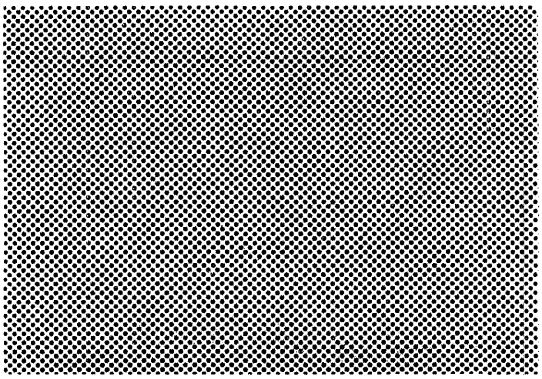
Modifiers: B, X, *

TABLE 3-4. SIMULATED SYSTEM CALLS

Name		Number	Input	Output	Function
TCI	Terminal Character Input	200_8	-----	$(A)_{8-15}$ = character	Input one character from TTY
TCO	Terminal Character Output	201_8	$(A)_{8-15}$ = character	-----	Output one character to TTY
SLEM	Set Line Editing Mode	202_8	Same as 2 + 2 spec except (A) and (U) ignored	-----	Same as 2 +2 spec
LLEM	Leave Line Editing Mode	203_8	-----	-----	Same as 2 +2 spec
GEL	Get Edited Line	204_8	Same as 2 + 2 spec	Same as 2 + 2 spec	Same as 2 +2 spec
BEL	Build Edited Line	205_8	Same as 2 + 2 spec	Same as 2 + 2 spec	Same as 2 +2 spec
GSEL	Get Status of Edited Line	206_8	Same as 2 + 2 spec	Same as 2 + 2 spec	Same as 2 +2 spec
--	Open File for Input	207_8	(U, A) = string pointers	(A) = file no. (X) = file size (U) = file type	Open specified file for input. Skip return if O.K.

TABLE 3-4. SIMULATED SYSTEM CALLS (Cont)

Name		Number	Input	Output	Function
--	Open File for Output	210_8	(U,A) = string pointers. (X) = file type	(A) = file no.	Open specified file for output. Skip return if O.K.
--	I/O character to or from file	211_8	(U) = file no. (A) = character if file open for output	(A) = character if file open for input	Input or Output one character from or to file
--	Close File	212_8	(U) = file no.	-----	Close specified file.
--	Simulated Drum Call	213_8	(A) = drum block number (max. 200_{10}). (U) = core block number (unmapped and always in CPU regardless of from which processor the call is executed). (X) = 0 read from drum. (X) ≠ 0 write on drum.	-----	Reads or writes 512 words from/ to the simulated drum. Skip return if O.K. Illegal instruction trap if ;L LSIM command not executed before SCALL 213B.



IV...

Pseudo Operations

GENERAL

Pseudo-operations are defined as such because of their similarity to machine operations in an object program. They work indirectly on a problem by performing machine conditioning functions, such as memory allocating, and by directing the preparations of machine coding as in the case of macros. A pseudo-operation may generate several, one, or no words in the object program.

The initial set of implemented pseudo-operations by functional group, principal use, and name are given in Table 4-1, Initial Set of Pseudo-Operations.

CONTROL PSEUDO-OPERATIONS

RADIX 8 Interpret integers as octal

The radix for integers is set to eight so that all following integers are interpreted as octal.

RADIX 10 Interpret integers as decimal

The radix for integers is set to ten so that all following integers are interpreted as decimal. When an assembly begins, the radix is initialized to ten, so that RADIX 10 need never be used unless the RADIX 8 pseudo-op has been used.

END End of Assembly

When the pseudo-op is encountered, the assembly terminates.

The END pseudo-op must not appear in a macro, nor following an unterminated MACRO, LMACRO, IF, or RPT pseudo-op.

TABLE 4-1. INITIAL SET OF OFFERED PSEUDO-OPERATIONS

Pseudo-op Function	Principal Use	Name
Control	Select and control the operations of assembler.	RADIX 8 RADIX 10 END
Program Linking	Generate linking information from subprogram to subprogram.	ENTRY BENTRY
Storage Allocation	Control use of memory.	BSS BES
Symbol Defining	Define assembler source program symbols.	EQU SET
Data Generating	Produce data words for the assembly program.	PAR BPAR DATA STR STRC
Conditional	Conditional assembly of variable numbers of input words.	IF ELSE ELSF ENDF RPT ENDR
Macros	Generation of argument symbols and repeated substitution of arguments within macros.	MACRO ENDM LMACRO NARG NCHR

PROGRAM LINKING PSEUDO-OPERATIONS

ENTRY Define a symbol as external

The pseudo-op is used to declare a symbol as external when it is to be referred to from another subprogram.

The format is:

(symbol) ENTRY (no operand)

The symbol in the label field is declared external.

BENTRY Define a byte symbol as external

This pseudo-op is temporarily defined exactly as the ENTRY pseudo-op, and is interpreted as such by the Tymshare version of the LOGICON 2+2 assembler. In the final assembler, the pseudo-op will be utilized to define byte external symbols.

STORAGE ALLOCATION PSEUDO-OPERATIONS

BSS Block starting symbol

The BSS pseudo-op is most commonly used to reserve a block of memory for use as data or working storage. Its format is as follows:

(symbol) BSS (expression)

If there is a symbol in the label field it is defined as the value of the current location counter. The counter is then incremented by the value of the expression in the variable field, which must be predefined and absolute ($8K \leq e < 8K$).

When the counter is incremented (or decremented, if the value of the expression is negative) the cells skipped over are not initialized or modified in any way.

BES Block ending symbol

The BES pseudo-op is commonly used to reserve a block of cells for use as working storage as follows:

(symbol) BES (expression)

The current counter is incremented by the value of the expression in the variable field, which must be predefined and absolute ($8K \leq e < 8K$). If there is a symbol in the name field it is then defined as the new value of the counter. When the counter is incremented (or decremented), the cells skipped over are not initialized nor modified in any way.

Note that the only difference between the BSS and BES pseudo-ops is that BSS defines the name first and then increments the counter, while BES increments first and then defines the name. Thus, NAME BSS N reserves a block of N cells and assigns NAME to the first cell, and NAME BES N reserves a block of N cells and assigns NAME to the last cell plus one.

SYMBOL DEFINING PSEUDO-OPERATIONS

EQU — Equate a symbol to a value

The form of the EQU pseudo-op is:

(symbol) EQU (expression)

The symbol is defined with the value of the expression.

If the symbol is already defined, its value is redefined. The expression in the variable field must be defined. If the symbol has been declared external before, the EQU pseudo-op preserves the information. The redefinition capability just described is available only for the Tymshare version of the LOGICON 2+2 assembler. When redefinition is desired, the SET pseudo-op would be used as described below.

SET — Symbol redefinition

The SET pseudo-op defines or redefines a symbol as being equal to an expression. Its form is:

(symbol) SET (expression)

The symbol in the name field is given the same value as the expression contained in the variable field. If the symbol has already been defined by a previous SET or by the increment list of a previous RPT pseudo-op, it is redefined. The expression in the variable field must be predefined. A symbol defined by a SET pseudo-op (or by the increment list of a RPT pseudo-op) must be so defined at least once before it is referenced.

DATA GENERATING PSEUDO-OPERATIONS

PAR Parameter

The PAR pseudo-op generates a data word containing a 15-bit word address, with an optional indirect addressing flag. The format is as follows:

(symbol) PAR (expression)

or,

(symbol) PAR* (expression)

One word is generated, whose low-order 15 bits contain the value of the expression in the first subfield. The expression must be a word address. If an asterisk appears after the pseudo-operation code the high-order bit of the word will be set to one, indicating indirect addressing; if not, the high-order bit will be zero.

If there is a symbol in the label field it is defined as the location of the word generated.

BPAR Byte parameter

This pseudo-op is temporarily defined exactly as the DATA pseudo-op and is interpreted as such by the Tymshare version of the LOGICON 2+2 assembler. In the final assembler, the pseudo-op will be utilized to define byte data parameters.

DATA Generate data

The DATA pseudo-op is used to generate one or more 16 bit data words. Its format is as follows:

(symbol) DATA (expression$_1$), (e_2), (e_3), . . .

One data word, containing the value of the corresponding expression, is generated for each subfield in the variable field. There is no restriction on expression type.

If there is a symbol in the label field it is defined as the location of the first word generated.

STR Generate string

The STR pseudo-op is used to convert a string of characters to a string of ASCII 8-bit bytes. The format is:

(symbol) STR (string of characters enclosed in unique delimiters)

Each character in the string between the delimiters is converted to an 8-bit byte, and pairs of bytes are packed into successive 16-bit words. If the total number of characters is odd, the last word is filled out with a null (zero) byte. Thus, for an N-character string the total number of words generated is (N+1)/2.

If there is a symbol in the label field, it is defined as the location of the first word generated. Note that the symbol is defined as a word address, not a byte address.

STRC Generate string with counts

The STRC pseudo-op is used to convert a string of characters to a string of ASCII 8-bit bytes headed by a one word count. The count word contains the byte or character count of the string. The format is as follows:

(symbol) STRC (string of characters enclosed in unique delimiters)

One word is generated containing the byte count. Each character in the string is converted to an 8-bit byte and pairs of bytes are packed into successive 16-bit words. If the total number of characters in the string is odd, the last word is filled out with a null (zero) byte. Thus, for an N-character string the total number of words generated is (N+3)/2.

If there is a symbol in the label field it is defined as the location of the first word generated. Note that the symbol is defined as a word address, not a byte address.

CONDITIONAL PSEUDO-OPERATIONS

A note concerning conditional expressions and true-false values. The Tymshare version of the LOGICON 2+2 assembler defines the value of true as >0, and false as ≤ 0. A potential problem develops in going to

the final LOGICON 2+2 assembler where these definitions will be changed. The final version of the assembler will define true as <0, and false as ≥ 0.

This means that, in using the Tymshare version of the assembler, expressions that evaluate to true or false cause no problem in the final assembler. However, if expressions are used that evaluate to a numeric value, then definite problems are generated when going from one assembler to the other.

IF, ELSF, ELSE, AND ENDF If statements

It is frequently desirable to permit the assembler either to assemble or to skip blocks of statements, depending on the value of an expression at assembly time. This is primarily what is meant by conditional assembly. Conditional assemblies are done by using either an if statement or a repeat statement.

The format of an if statement is

```
IF  expression
<if body>
ENDF
```

The if body is any block of statements, in particular, it may contain directives of the form

```
ELSF  expression
```

and

```
ELSE
```

If the operand of IF is true, then the block of code up to the matching ENDF (or ELSF or ELSE) is processed; otherwise, it is skipped. The values for true and false are:

```
true :  value of expression >0
false:  value of expression ≤0
```

Examples:

```
IF      1>0
LDA     ALPHA
STA     BETA
ENDF
```

processed

```
IF      0
LDA     GAMMA
STA     DELTA
ENDF
```

skipped

Often there are more than two alternatives, so the ELSF directive is used in the if body. When ELSF is encountered while skipping a block of statements, its operand is evaluated (just as for IF) to decide whether to process the block following the ELSF.

Examples:

```
IF      0>1
LDA     ALPHA       skipped
ELSF    1>0
LDA     BETA        processed
ENDF
```

```
IF      0>1
LDA     ALPHA       skipped
ELSF    0>1
LDA     BETA        skipped
ENDF
```

```
IF      1>0
LDA     ALPHA       processed
ELSF    1>0
LDA     BETA        skipped
ENDF
```

```
IF      0>1
LDA     ALPHA       skipped
ELSF    1>0
LDA     BETA        processed
ELSF    1>0
LDA     GAMMA       skipped
ENDF
```

From the last two examples above it should be clear that either no blocks are processed or precisely one is; thus, as soon as one block is processed, all following blocks are skipped regardless of whether the ELSF expressions are true.

An ELSE directive is equivalent to an ELSF directive with a true expression.

Example:

```
IF      0>1
LDA     ALPHA           skipped
ELSE
LDA     BETA            processed
ENDF
```

As a more general example, consider the following:

```
IF  e1
<body 1>
ELSF  e2
<body 2>
ELSF  e3
<body 3>
ELSE
<body 4>
ENDF
```

There are four possibilities:

1. e1>0 : process body 1, skip the other three.
2. e1≤0, e2>0 : process body 2, skip the other three.
3. e1≤0, e2≤0, e3 0: process body 3, skip the other three.
4. e1≤0, e2≤0, e3≤0: process body 4, skip the other three.

The bodies between the IF, ELSF, ELSE, and ENDF directives may contain arbitrary statements, in particular they may contain other if statements. This nesting of if statements may go to any level.

When evaluating the expression in the operand field of IF or ELSF, all undefined symbols are treated as if they were defined with value -1. These expressions must be absolute.

RPT and ENDR Repeat Statements

A repeat statement is a means of processing the same text many times. The format is

```
(symbol)   RPT   expression [, increment list]
<repeat body>
ENDR
```

The value of the RPT operand (which must be defined and absolute) determines how many times the repeat body will be processed, while the increment list (described below) is a mechanism to allow the values of various symbols to be changed each time the repeat body is processed.

Example:

```
ABC     RPT     4
        DATA    0
        ENDR
```

This is equivalent to

```
ABC     DATA    0
        DATA    0
        DATA    0
        DATA    0
```

An increment list has the form (s=e1 [,e2] ...)s=e1 [,e2]) where s stands for a symbol and e1 and e2 denote expressions, (which must be absolute; undefined symbols are treated as if they were defined with the value -1). Before the repeat body is processed for the first time, each symbol in the list is given the value of its associated e1. Thereafter, each symbol is incremented by the value of its associated e2 just before the repeat body is processed. If e2 is missing, the value 1 is assumed. There is no limit on the number of elements that may appear in an increment list.

Example:

```
RPT  3, (I = 4) (J = 0, -1)
DATA I
DATA J * I + 1
ENDR
```

This results in code equivalent to the following:

```
DATA   4
DATA   0 * 4 + 1 = 1
DATA   5
DATA   -1 * 5 + 1 = -4
DATA   6
DATA   -2 * 6 + 1 = -11
```

There is another format for RPT:

```
(symbol)   RPT   (s=e1 [,e2] ,e3) [increment list]
```

In this case, the number of times the repeat body is processed is determined by the construct (s=e1 [,e2] ,e3). This is the same as an increment list except that it includes a third expression (which must be absolute; all undefined symbols are treated as if they were defined with the value -1), namely a bound on the value of the symbol. As soon as the bound is passed, processing of the repeat body stops. In the example above, the same effect could have been achieved by writing the head of the repeat statement as

```
RPT   (J = 0, -1, -2) (I = 4)
```

or as

```
RPT   (I = 4, 6) (J = 0, - 1)
```

Note that the bound does not have to be positive or greater than the initial value of the symbol being incremented; the algorithm for determining when the bound has been passed is given below.

Figure 4-1 illustrates precisely the actions of the RPT repeat options:

```
RPT   expression [, increment list]
```

The contents of a repeat body may contain any NARP code, in particular it may contain other repeat statements; the nesting of repeat statements may go to any level.

INTRODUCTION TO MACROS

On the simplest level a macro name may be thought of as an abbreviation or shorthand notation for one or more assembly language statements. In this respect it is like an opcode in that an opcode is the name

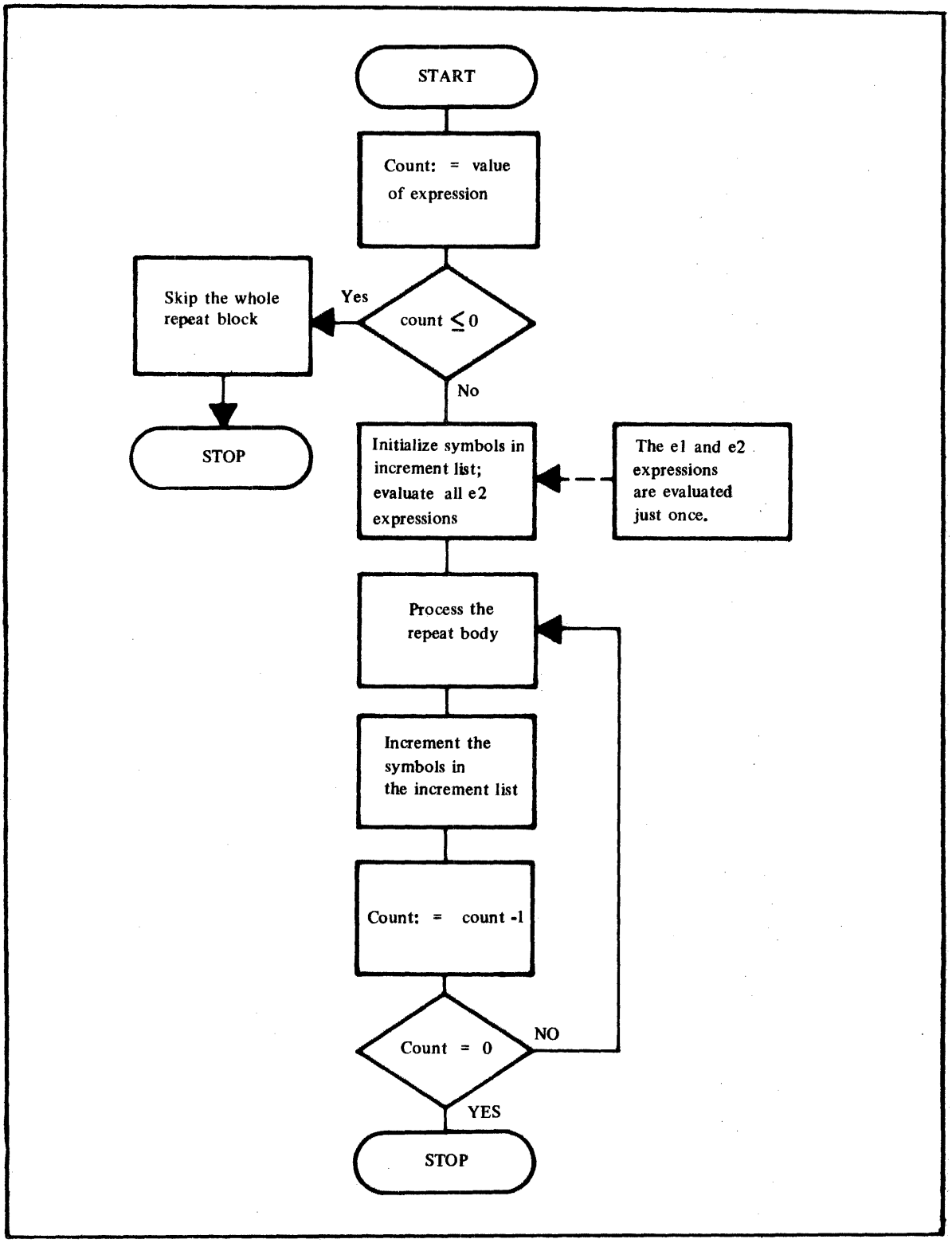


Figure 4-1. RPT Repeat Options, Flowchart

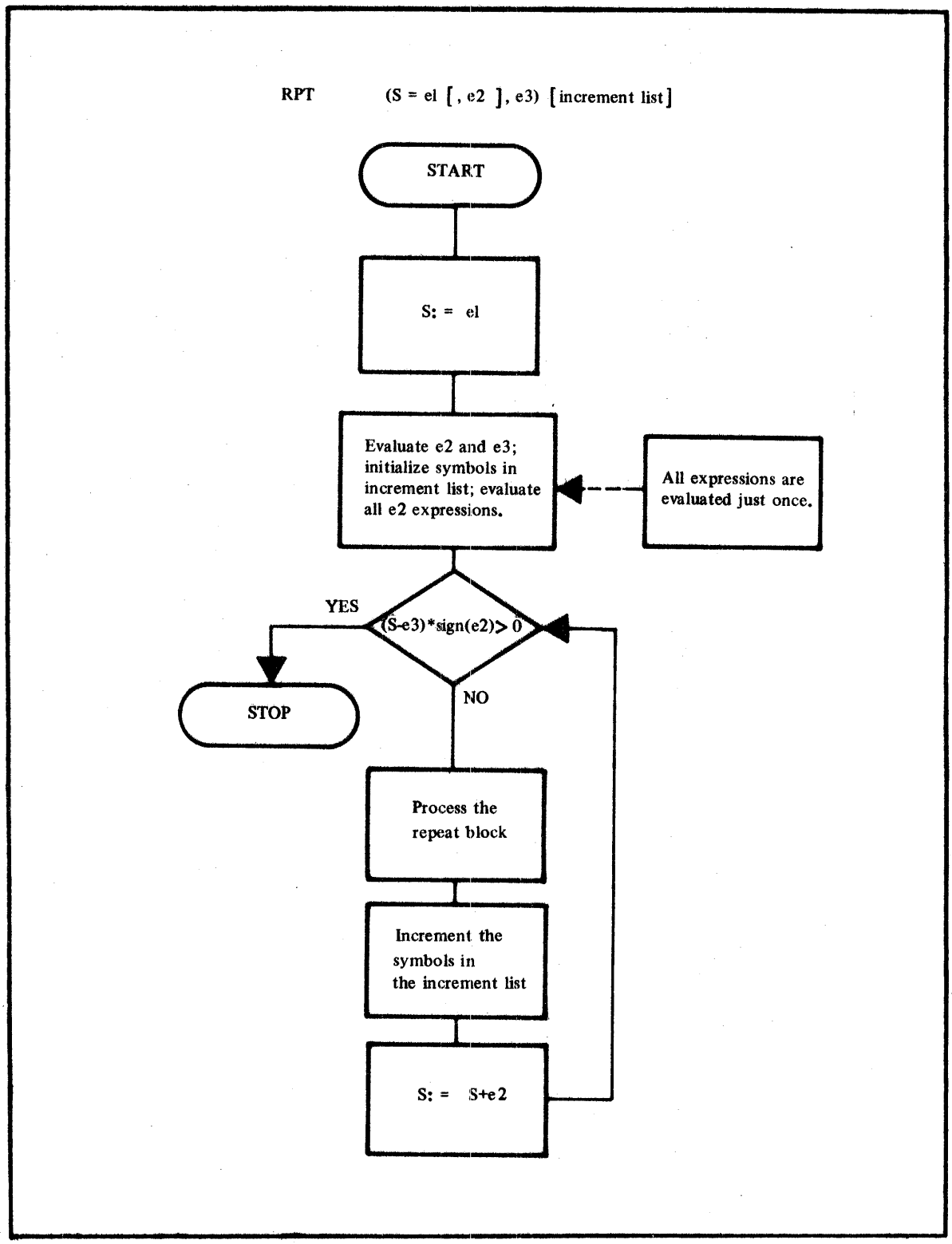


Figure 4-1. RPT Repeat Options, Flowchart (Continued)

of a machine command and a macro name is the name of a sequence of assembly language statements.

The "940" has an instruction for skipping if the contents of a specified location are negative, but there is no instruction for skipping if the accumulator is negative. The instruction SKA (skip if memory and the accumulator do not compare ones) will serve when used with a cell whose contents mask off all but the sign bit. The meaning of SKA when used with such an operand is "skip if A is positive." Thus a programmer writes

```
SKA     =4B7
BRU     NEGCAS          NEGATIVE CASE
```

However, it is more than likely the case that the programmer wants to skip if the accumulator is negative. Then he must write

```
SKA     =4B7
BRU     *+2
BRU     POSCAS          POSITIVE CASE
```

Both of these situations are awkward in terms of assembly language programming.

But we have in effect just developed simple conventions for doing the operations SKAP and SKAN (skip if accumulator positive or negative). Define these operations as macros:

```
SKAP    MACRO
        SKA             =4B7
        ENDM
SKAN    MACRO
        SKA             =4B7
        BRU             *+2
        ENDM
```

Now, more in keeping with the operations he had in mind, the programmer may write

```
A22     SKAN
        BRU             POSCAS
```

The advantages of being able to use SKAP and SKAN should be apparent. The amount of code written in the course of a program is reduced; this in itself tends to reduce errors. A greater advantage is that SKAP and SKAN are more indicative of the action that the programmer had in mind, so that programs written in this way tend to be easier to read. Note, incidentally, that a label may be used in conjunction with a macro. Labels used in this way are usually treated like labels on instructions; they are assigned the current value of the location counter. This will be discussed in more detail later.

Before discussing more complicated uses of macros, some additional vocabulary should be established. A macro is an arbitrary sequence of assembly language statements together with a symbolic name. During assembly, the macro is stored in an area of memory called the string storage. Macros are created (or, as is more frequently said, defined) by giving a name and the associated sequence of statements. The name and the beginning of the sequence of statements are designated by the MACRO pseudo-op:

```
name        MACRO
              •
              •
            ENDM
```

The end of the sequence of statements is indicated by the ENDM pseudo-op.

Refer to Figure 4-2. When the assembler encounters a MACRO pseudo-op, switch B is thrown to position 1 so that the macro is simply copied into the string storage; note that the assembler does no normal processing but simply copies the source language. When the ENDM terminating the macro definition is encountered, switch B is put back to position 0 and the assembler goes on processing as usual.

It is possible that within a macro definition other definitions may be embedded. The macro defining machinery counts the occurrences of the MACRO pseudo-op and matches them against the occurrences of ENDM. Thus switch B is actually placed back in position 0 only when the ENDM matching the first MACRO is encountered. Therefore,

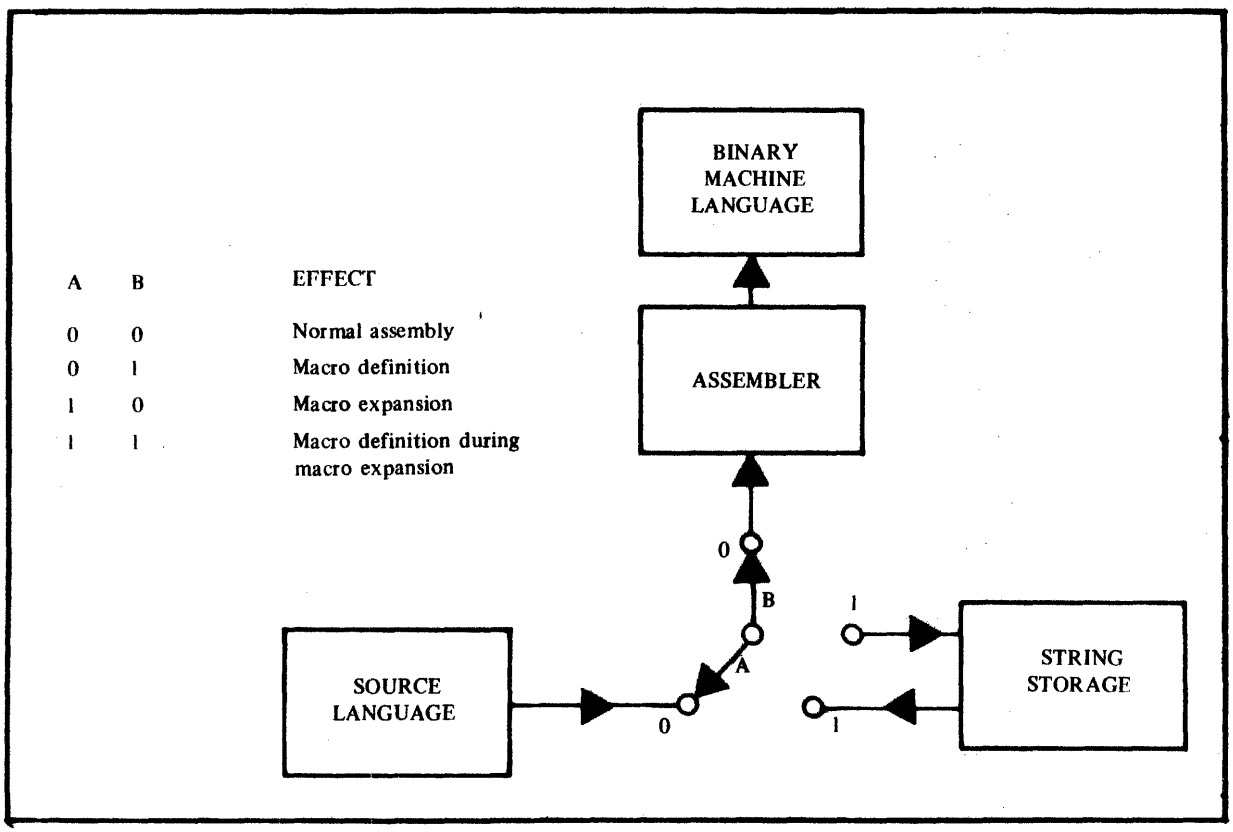


Figure 4-2. Information Flow During Macro Processing

MACRO and ENDM are opening and closing brackets around a segment of source language. Structures like the following are possible:

```
name 1      MACRO
name 2      MACRO
name 3      MACRO
            ENDM
name 4      MACRO
            ENDM
            ENDM
name 5      MACRO
            ENDM
            ENDM
```

The utility of this structure will not be discussed here. Use of this feature of imbedded definitions should in fact be kept to a minimum since the implementation of this assembler is such that it uses large amounts of string storage in this case. What is important, however, is an understanding of when the various macros are defined. In particular, when name 1 is being defined, name 2, 3, etc., are not defined; they are merely copied into string storage. Name 2, for example, will not be defined until name 1 is expanded.

The use of a macro name in the opcode field of a statement is referred to as a call. The assembler, upon encountering a macro call, moves switch A to position 1 (see Figure 4-2). Input to the assembler from the original source file temporarily stops and comes instead from the string storage. During this period the macro is said to be undergoing expansion. It is clear that a macro must be defined before it is called.

An expanding macro may include other macro calls, and these, in turn, may call still others. In fact, macros may even call themselves; this is called recursion. Examples of the recursive use of macros are given later. When a new macro expansion begins within a macro expansion, information about the progress of the current expansion is saved. Successive macro calls cause similar information to be saved. At the end of each expansion the information about each previous expansion is restored. When the final expansion terminates, switch A is placed back in position 0, and input is again taken from the source file.

Now let us carry our example one step further. One might argue that the action of skipping is itself awkward. It might be preferable to write macros BRAP and BRAN (branch to specified location if contents of accumulator are positive or negative). How is one to do this? The location to which the branch should go is not known when the macro is defined, in fact, different locations will be used from call to call. The macro processor, therefore, must enable the programmer to provide some of the information for the macro expansion at call time. This is done by permitting dummy arguments in macro definitions to be replaced by arguments (i.e., arbitrary substrings) supplied at call time. Each dummy argument is referred to in the macro definition by a subscripted symbol. This symbol or dummy name is given in the operand field of the MACRO pseudo-op.

Let us define the macro BRAP:

```
BRAP    MACRO   LABEL
        SKAN
        BRU     LABEL(1)
        ENDM
```

When called by the statement 'BRAP POSCAS', macro will expand to

```
        SKA     =4B7
        BRU     *+2
        BRU     POSCAS
```

Note that BRAP was defined in terms of another macro, SKAN. Also note that, as defined, BRAP was intended to take only one argument; other macros may use more than one argument.

The macro CBE (compare and branch if equal) takes two arguments. The first argument is the location of a cell to be compared for equality with the accumulator; the second is a branch location in case of equality. The definition is

```
CBE     MACRO   D
        SKE     D(1)
        BRU     *+2
        BRU     D(2)
        ENDM
```

When CBE is called by the statement

```
CBE     =21B, EQLOC
```

the statements generated will be

```
        SKE     =21B
        BRU     *+2
        BRU     EQLOC
```

Note that in the macro call, the arguments are separated by commas.

The following sections describe macro definitions and calls in more detail.

MACRO, LMACRO, and ENDM Macro definition

The form of a macro definition is:

$$\text{name} \quad \left\{ \begin{matrix} \text{MACRO} \\ \text{or} \\ \text{LMACRO} \end{matrix} \right\} \quad [\text{dummy} [, \text{generated}, \text{expression}]$$

where name, generated, and dummy are all symbols, and expression is an expression.

LMACRO is completely equivalent to MACRO except that if name is defined as a macro with MACRO the construct

label name arguments

will automatically cause label to be defined as the current value of the location counter, whereas if name were defined with LMACRO this automatic definition of label would not occur.

Some details of the definition

If generated appears, it should not be the same symbol as dummy, and neither of them should be "MACRO", "LMACRO", or "ENDM".

If name is already defined as an opcode, the old definition is completely replaced by the new.

If the MACRO (or LMACRO) directive has no operand, then name is defined as an opcode that takes no operand. Otherwise, name becomes an opcode that may or may not take an operand.

Whole-line comments (lines beginning with *) in the macro body are not saved in string storage as part of the macro definition, but comments following instructions are. Thus, it behooves the programmer to avoid the latter, as they eat string storage.

When a macro body is placed in string storage, superfluous blanks are removed. Thus, any contiguous string of blanks is compressed to one blank with the following exceptions:

1. Blanks enclosed in single quotes (') are not compressed.
2. Blanks enclosed in double quotes (") are not compressed.
3. Blanks enclosed in parentheses are not compressed. In this use, the nesting of parentheses is taken into account, but a parenthesis between single or double quotes is not considered as part of the nesting structure.

In most cases the programmer need not worry about these conventions, although there are times when he may get pinched. For example, if

```
ASC   %A222B%
```

appears in a macro definition, it will be expanded as

```
ASC   %A2B%
```

To avoid such problems use

```
ASC   'A222B'
```

Dummy arguments

The dummy argument specified as an operand of the MACRO pseudo-op may appear anywhere in the macro body, followed by a subscript. At call time the subscript is evaluated and its value is used to select the appropriate argument supplied in the call. Before describing the various kinds of dummy arguments a few conventions are needed:

1. In the following, "argument" will refer to the character string as given in the macro call after possible enclosing parentheses have been removed.
2. The number of arguments supplied by the call is n ($n \geq 0$).
3. The number of characters in argument ei is n(ei).
4. The structure ei for i an integer stands for an expression. (However, its value stands for some argument usually, so ei will be used somewhat ambiguously to stand for an expression or the value of an expression.) The first argument in a call is numbered 1.
5. The dummy argument is assumed to be "D".

With the preceding in mind, we consider the three forms of dummy arguments:

1. D(e1)

   This expands to argument e1 (which may be the null string), where $0 \leq e1 \leq n$. (If e1 = 0 then D(e1) expands to the label field of the macro call)

Special notation: D() = D(1)

2. D(e1, e2)

   If e1 > e2 then this expands to the null string (range of values of e1 and e2 is arbitrary), otherwise, this expands to argument e1 through e2, where $0 \le e1 \le e2 \le n$, with each argument enclosed in parentheses and a comma inserted between each argument. For example, D(3, 3) = (D(3)).

   Special notation: D(,) = D(1, n)
   D(, e1) = D(1, e1)
   D(e1,) = D(e1, n)

3. D(e1$e2, e3)

   In all cases, $0 \le e1 \le n$ must be true. If e2 > e3 then this expands to the null string (range of values of e2 and e3 is arbitrary), otherwise, it expands to characters e2 through e3 of argument e1, counting the first character of an argument as character 1. If either e2 or e3 lies outside the argument, then the nearest boundary is chosen. To be more precise, before using e2 and e3 to select the piece of argument e1 that is desired, the following transformation is made:

   e2:= max (1, e2); e3:= max (1, e3);
   e2:= min (n(e1), e2); e3:= min (n(e1), e3);

   If argument e1 is the null string, then the dummy argument expands to the null string regardless of the values of e2 and e3.

   Special notations:

   D(e1$,) = D(e1$1, n(e1)) = D(e1)
   D(e1$, e2) = D(e1$1, e2)
   D(e1$e2,) = D(e1$e2, n(e1))
   D(e1$e2) = D(e1$e2, e2)
   D(e1$) = D(e1$1) = D(e1$1, 1)

In any of the forms mentioned above, e1 may be missing; if so, 1 is assumed, e.g., D($) = D(1$1, 1).

A general rule that will help in remembering what the special notations mean is the following: "Whenever an expression is missing from a form, the value 1 is assumed unless the expression is missing from a

place where an upper bound is expected (as in D(3,) or D(3$2,), in which case the largest 'reasonable' value is assumed.

In any of the preceding three cases, if an expression which designates an argument is out of range, then an error message is typed and argument 0 is taken.

Generated symbols

A macro should not, of course, have in its definition an instruction having a label. Successive calls of the macro would produce a multiply-defined symbol. Sometimes, however, it is convenient to put a label on an instruction within a macro. There are at least two ways of doing this. The first involves transmitting the label as a macro argument when it is called. This is most reasonable in many cases; it is in fact often desirable so that the programmer can control the label being defined and can refer to it elsewhere in the program.

However, situations do arise in which the label is used purely for reasons local to the macro and will not be referred to elsewhere. In cases like this it is desirable to allow for the automatic creation of labels so that the programmer is freed from worrying about this task. This may be done by means of the generated symbol.

A generated symbol name may be declared when a macro is defined, specifying the name and the maximum number of generated symbols which will be encountered during an expansion. These two items follow the dummy symbol name given in the MACRO pseudo-op if the programmer wishes to use generated symbols in a macro. For example,

```
MUMBLE    MACRO       D,G,4
          <macro body>
          ENDM
```

might contain references to G(1), G(2), G(3), and G(4), these being individual generated symbols.

With regard to generated symbols the macro expansion machinery operates in the following fashion: A generated symbol base value for each macro is initialized to zero at the beginning of assembly. As each generated symbol is encountered, the expression constituting its subscript is evaluated. This value is added to the base value, and the sum is produced as a string of digits concatenated to the generated symbol

name; the first digit is always 0 to reduce the likelihood of the generated symbol being identical to a normal symbol defined elsewhere by the programmer. Thus, the first time MUMBLE is called, G(2) will be expanded as GØ2, G(4) as GØ4, etc.

At the end of a macro expansion the generated symbol base value is incremented by the amount designated by the expression following the generated symbol name in the MACRO directive. This is 4 in the case of MUMBLE. Thus, the second call of MUMBLE will produce in place of G(2), GØ6, the third call will produce GØ 1Ø, etc. It should be clear that the generated symbol name should be kept as short as possible.

The expression in the macro head (call it m) must have a value in the range [1, 1023] . A generated symbol subscript must have a value in the range [1, m] .

Concatenation

Occasionally, it is desirable to have a dummy argument follow immediately after an alphanumeric character, for example, to have D(1) follow just after ALPHA. But then the assembler would not recognize the dummy because it would see ALPHAD(1). To get around this problem the concatenation symbol '. &' is introduced. Its sole purpose is to separate a dummy argument (or conceivably a generated symbol) from a preceding alphanumeric character during macro definition. Thus, the example becomes ALPHA. &D(1). The concatenation symbol is not stored in string storage so it does not appear during expansion.

The concatenation symbol may appear anywhere in a macro definition, but it is only necessary in the case described above. If one macro is defined within another, any concatenation symbols within the inner macro will not be removed during the definition of the enclosing macro.

Conversion of a value to a digit string

As an adjunct to the automatic generation of symbols (or for any other purposes for which it may be suited) a capability is provided in the assembler's macro expansion machinery for conversion of the value of an expression at call time to a string of decimal digits. The construct

($expression)

will be replaced by a string of digits equal to the value of the expression. For example, if X=5 then

```
AB($2*X+1)
```

will be transformed into

```
AB11
```

If the value of the expression is zero then the digit string is '0'; if it is negative then the digit string is preceded by a minus sign.

This conversion scheme can also be used inside repeat blocks; for example

```
            RPT     (I=1,10)
TEMP($I)    BSS     1
            ENDR
```

creates 10 cells labeled TEMP1 through TEMP10.

A note on subscripts

The expressions used as subscripts for dummy arguments and generated symbols, as well as the expressions used in the conversion to a digit string must be absolute. Any undefined symbols appearing in these expressions are treated as if they were defined with the value -1. These expressions may themselves contain dummy arguments, generated symbols, and ($...), so constructs like (R4+D(I*D(3))) are possible.

NARG and NCHR Number of arguments and number of characters

Macros are more useful if the number of arguments supplied at call time is not fixed. The precise meaning of a macro (and indeed, the result of its expansion) may depend on the number or arrangements of its arguments. In order to permit this, the macro undergoing expansion must be able to determine at call time the number of argument supplied. The NARG directive makes this possible.

NARG functions like EQU except that no expression is used with it. Its form is

```
(symbol)    NARG
```

The function of the directive is to equate the value of the symbol to the number of arguments supplied to the macro currently undergoing expansion. The symbol can then be used by itself or in expressions for any purpose. NARG may appear in any macro, even one which has no dummy argument (and thus never has any arguments at call time); it is an error for NARG to appear outside a macro.

It is also useful to be able to determine at call time the number of characters in an argument. NCHR functions by equating the symbol in its label field to the number of characters in its operand field. Its form is

(symbol) NCHR [character string]

where "character string" has exactly the same form as an argument supplied for a macro call, i. e., if it involves blanks, commas, or semicolons it should be enclosed in parentheses. NCHR can appear anywhere, both inside and outside macros, but it is most useful in macros for determining the length of arguments.

Examples:

```
A   NCHR   ABCDEF    A:=6
B   NCHR   (,,XYZ,,)  B:=7
C   NCHR   D(I)       C:=?
```

Macro calls

The format of a macro call is:

(symbol) macroname [argstring]

Such a call causes the macro whose name appears in the opcode field to be expanded, with the dummy argument in the macro body replaced by the actual arguments of the argstring.

The label field is always transmitted as argument 0, so that D(e1), where e1 has value 0, is always legal inside a macro. An occurrence of D(e1), where e1=0, will be replaced by the label field. If the label field is empty, then D(e1) expands to the null string. At most seven characters will be transmitted this way: the first six characters of the symbol in the label field, preceded by '$' if the label field begins with '$'.

If the user wishes to transmit an argument to a macro in the label field of the macro call, but does not wish to have the symbol in this field defined, he should define the macro with LMACRO rather than MACRO. As an example:

```
NT      LMACRO    D
        RPT       D(1)
        DATA      D(2)
        ENDR
D(0)    DATA      -D(1)
        ENDM
```

when called by:

```
        DTE     NT          4,4B7
```

expands as:

```
        DATA      4B7
        DATA      4B7
        DATA      4B7
        DATA      4B7
DTE     DATA      -4
```

Notice that this would have caused a doubly-defined symbol error had MACRO been used rather than LMACRO.

A macro call may or may not have an arg string. If an arg string is present, it may contain any number of arguments, in fact, more than are referred to by the macro.

Before describing an arg string, the following should be noted: blanks, commas, semi-colons, and parentheses that are enclosed in single or double quotes are treated exactly like ordinary characters enclosed in quotes; they do not serve as terminators, separators, delimiters, or the like. In effect, when the argument collector is collecting arguments for a macro call, the occurrence of a quote causes it to stop looking for special characters except for a matching quote (and, of course, carriage return, which is an absolute terminator). A single quote enclosed in double quotes is not a special character and vice versa. Thus, when a blank, comma, semi-colon, or parenthesis is referred to in the following, it is understood that it is <u>not</u> enclosed in quotes.

An arg string for a macro call has the following format:

<arg>, <arg>,..., <arg> <terminator>

where a terminator is a blank, semi-colon, or carriage return. There are three forms of arg:

1. <arg> may be the null string.

2. If the first character of <arg> is not a left parenthesis then <arg> is a string of characters not containing blank, comma, semi-colon, or carriage return (remember that blanks, commas, and semi-colons may appear in <arg> if they are enclosed in quotes).

3. If the first character of <arg> is a left parenthesis the <arg> does not terminate until a blank, comma, or semi-colon is encountered after the right parenthesis which matches the initial left parenthesis ("matches" means that all left and right parentheses in the argument are noted and paired off with each other so that a nested parenthesis structure is possible). Of course, a carriage return at any point immediately terminates <arg>. Again, remember that blanks blanks, commas, semi-colons, and parentheses enclosed in quotes are ignored when <arg> is being delimited. The initial left parenthesis and its matching right parenthesis (which need not be the last character in <arg>) are removed before <arg> is transmitted to the macro. Examples:

```
AMAC      (,,;,,),,'HOUSE,,ROGER',(AB'')''
  D(1)  =  ,,;,,
  D(2)  =  null string
  D(3)  =  'HOUSE,, ROGER'
  D(4)  =  AB'')''
```

Example of conditional assembly and macros

The following macro, MOVE, takes any number of pairs of arguments; the first argument of each pair is moved to the second, but an argument may itself be a pair of arguments, which may themselves be pairs of arguments, etc. MOVE extracts pairs of argument structures and transmits them to a second macro MOVE1.

```
MOVE    MACRO   D
X       NARG
        RPT     (Y=1,2,X)
        MOVE1   D(Y),D(Y+1)
        ENDR
        ENDM
```

The main work is done in MOVE1 which calls itself recursively until it comes up with a single pair of arguments.

```
MOVE1   MACRO   D,G,2
G(1)    NARG
G(2)    EQU     ∅
        IF      G(1)=2
        LDA     D(1)
        STA     D(2)
        ELSE
        RPT     G(1)/2,(G(2)=G(2)+1)
        MOVE1   D(G(2)),D(G(2)+G(1)/2)
        ENDR
        ENDF
        ENDM
```

When MOVE is called by

```
        MOVE    A,B
```

the code generated is

```
        LDA     A
        STA     B
```

When called by

```
        MOVE    A,B,C,D,
```

the code generated is

```
        LDA     A
        STA     B
        LDA     C
        STA     D
```

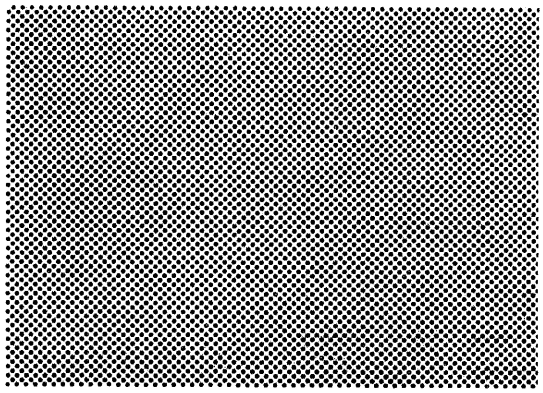
And when called by

```
MOVE    ((A, B), (C, D)), ((E, F), (G, H))
```

the code generated is

```
LDA     A
STA     E
LDA     B
STA     F
LDA     C
STA     G
LDA     D
STA     H
```

It is instructive to trace the example by hand to see how the recursive calls of MOVE1 work. This is an exercise left to the reader.



V...

Assembler Operating Instructions

INSTRUCTIONS

1. Calling the Assembler.

   When the EXEC asks for a new command by typing "-", type the sequence: "(P) ITRAN". (See any example)

2. Input File.

   The assembler will ask for the name of the input file by typing: "INPUT: ".
   The operator should respond by typing the name of the input file. (See any example)

3. Output File.

   The assembler will ask for the name of the output file by typing: "OUTPUT: ".
   The operator should respond by typing the name of the output file. (See Example 1)
   The assembler will respond with "NEW FILE" (See Example 1) or "OLD FILE" (See Example 2). If the response is correct the operator should verify it by typing a carriage return. (See any example)

4. Second Input File.

   When the input translation phase of the assembly is complete, the assembler will space two lines and ask for another input file by typing "INPUT: ". If the operator does not have another input file, he should respond by typing a carriage return. (See Example I). If the operator has another input file to be appended to the first file and assembled with it, he should respond by typing the file name. (See Example 5) This step will be repeated until the operator runs out of input files and types a carriage

return in response to the "INPUT: " message. The operator can cause the request for a second (or n^{th}) input file to be skipped by typing a line feed following the name of the first (or $n-1^{st}$) input file name. (See Example 3).

5. Pass 2 of the Assembly.

   The second phase of the assembly process is performed by the 940 NARP assembler running on commands retrieved from a commands file generated by the input translator. As it runs, three lines such as the following will be printed.

```
SOURCE FILE:   OBJECT FILE:   NEW FILETEXT FILE:
1 SEC    0 ERR      33 (27) WRD   (S:6,o:52,L:0,M:0,U:0)
 ?
```

   (See any example)
   The first line is produced by NARP asking for information that is supplied from the commands file, and can be ignored. The second line is a summary printed by NARP. The time given is for the NARP pass only. The total time taken by the assembly is approximately twice the time given. The number of errors given is the number detected by NARP. The word count is the number of words of output generated. The meaning of the remainder of the summary is unknown. The question mark on the third line is printed as the commands file terminates. It has no meaning, but there appears to be no way to get rid of it.

6. Errors During Input Translation Pass

   If any errors are detected during the input translation phase (hopefully nearly all errors will be detected in this phase), two lines will be printed for each. The first line gives the symbolic address of the offending statement and an error message. The second line is the offending statement exactly as it was received.

   When the operator indicates that he has no more input files, ITRAN will print a message of the following form: "n ERRORS - CONTINUE? (Y OR N) ". The operator should respond with "N" if he wishes to terminate the assembly process at this point and correct his errors, or with "Y" if he wishes to ignore the errors and continue with the assembly. Continuation will frequently result in a second error message being produced by NARP. In general that message will not be meaningful. (The "Y" or "N" typed in response

to the CONTINUE question must be followed by a carriage return.)
(See Examples 6 and 7)

If the operator types "N" in response to the CONTINUE question, a question mark will print on the next line. This is caused by the commands file terminating and should be ignored. (See Example 6)

7. Intermediate Files

The assembler uses two intermediate files, /ASSEMCOMM/ and /ASSEMINTERNAL/. If the operator has any files by either of these names, they will be destroyed. If the assembly is terminated under unusual circumstances, such as with an ESCAPE, one or both of these files may be left in the operators directory.

EXAMPLES

Example 1

```
-(P)ITRAN
INPUT: TEST
OUTPUT: BTEST
NEW FILE

INPUT:
SOURCE FILE:      OBJECT FILE:      NEW FILETEXT FILE:

1 SEC      0 ERR      33 (27) WRD      (S:6, 0:52, L:0, M:0, U:0)

 ?
```

Example 2

```
-(P)ITRAN
INPUT: TEST 1
 ?
INPUT: TEST
OUTPUT: BTEST
 OLD FILE

INPUT:
SOURCE FILE:      OBJECT FILE:      OLD FILETEXT FILE:

1 SEC      0 ERR      33 (27) WRD      (S:6, 0:52, L:0, M:0, U:0)
```

Example 3

```
-(P)ITRAN
INPUT: TEST
OUTPUT: BTEST
 OLD FILE
SOURCE FILE:      OBJECT FILE:      OLD FILETEXT FILE:
1 SEC      0 ERR      33 (27) WRD      (S:6, 0:52, L:0, M:0, U:0)
 ?
```

Example 4

```
-(P)ITRAN
INPUT: TEST
OUTPUT: TEST
FILE NOT BINARY
OUTPUT: TEST 1
 NEW FILEN

OUTPUT: BTEST
 OLD FILE

INPUT:
SOURCE FILE:      OBJECT FILE:      OLD FILETEXT FILE:
1 SEC      0 ERR      33 (27) WRD      (S:6, 0:52, L:0, M:0, U:0)
 ?
```

Example 5

```
-(P)ITRAN
INPUT: TEST
OUTPUT: BTEST
 OLD FILE

INPUT: TEST 1

INPUT:
SOURCE FILE:      OBJECT FILE:      OLD FILETEXT FILE:
1 SEC      0 ERR      61 (49) WRD      (S:6, 0:52, L:0, M:0, U:0)
 ?
```

Example 6

```
-(P)ITRAN
INPUT: TEST 2
OUTPUT: BTEST
 NEW FILE
AB + 6 - ILLEGAL ADDRESS MODIFIER
 LDE AB, BX

INPUT:

1 ERRORS - CONTINUE?  (Y OR N)  N
 ?
```

Example 7

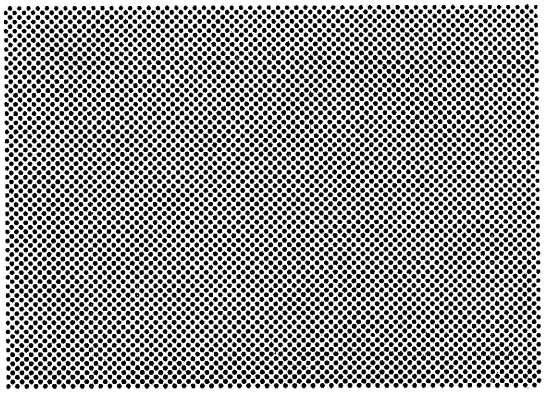
```
-(P)ITRAN
INPUT: TEST 2
OUTPUT: BTEST
 NEW FILE
AB + 6 - ILLEGAL ADDRESS MODIFIER
 LDE AB, BX

INPUT:

1 ERRORS - CONTINUE?  (Y OR N)  Y
SOURCE FILE:    OBJECT FILE:   NEW FILETEXT FILE:

1 SEC       0 ERR       34 (28) WRD       (S:6, 0:52, L:0, M:0, U:0)

 ?
```



VI...

LSIM Loading, Simulating, and Debugging

GENERAL

LSIM is a loader, simulator and debugger for the LOGICON 2+2 machine which runs on Tymshare. It has facilities for loading relocatable NARP output, linking external symbols and expressions, typing out and altering the contents of registers and memory cells, and inserting breakpoints. References in commands and type out of symbols may be numeric to any radix between 2 and 10 or symbolic.

Symbols

A symbol is any string of alphanumeric characters beginning with an alphabetic character. Symbols may not contain more than six characters. There are two special symbols that are recognized only as the first operand in an expression: "." and "$". Their meaning will be explained below. Symbols are introduced to LSIM in two different ways: they may be read in from assembly output or they may be entered via a command. If an undefined symbol is read when inputting a command, LSIM will type the error message (U).

Constants

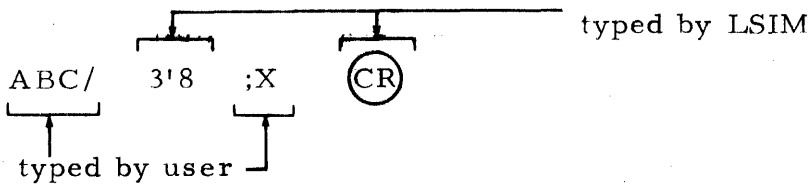
A constant is any string of numeric characters. The value of a constant is computed to the current radix which is initially 8 and may be set to any value between 2 and 10 by a command. When constants are typed out they are also converted through the current radix.

Expressions

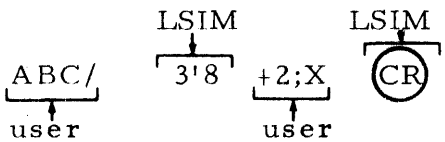
Expressions are composed of symbols, constants, and the operators + and -. Evaluation is strictly left to right and parentheses are not allowed.

Open Registers or Memory Cells

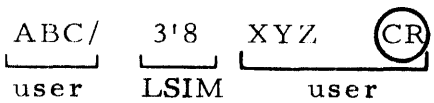
The / command causes a memory cell to be opened. This means that the contents of the memory cell is typed out as an unsigned integer (followed by a quote and the current radix evaluated in decimal to avoid confusion) followed by 3 spaces. LSIM then waits for the user to type something. If he types a symbol or constant followed by a (CR), (LF), or ↑ the value of the symbol or constant typed is entered in the open memory cell. If he types any other command or an expression followed by a command the contents of the open cell is taken as the first operand in the following expression whose value becomes the argument for the following command. Consider the following example:



The user opens cell ABC whose contents is 3_8. He then types the ;X command that sets the current radix. The 3_8 is taken as the argument to the ;X command causing the current radix to be set to 3. A further example:



This sequence will cause the 3 to be used as the first operand of the following expression which is then used as the argument to the command changing the current radix to 5. In both of the above examples the contents of cell ABC is not altered. The following sequence will enter the value of the symbol XYZ in cell ABC.



The ;R commands do for the central registers exactly what the / command does for memory cells (see list of commands below).

COMMANDS

Symbols:

n	=	unsigned integer.
e	=	expressions consisting of symbols, integers evaluated to the current radix, and the operators + and -.
s	=	symbol or integer.
.	=	address of last memory cell opened.
$	=	address +1 of last cell loaded.
e;X		sets current radix to value of e.
;D		sets current radix to 10 (decimal).
e=		types values of e as an unsigned integer.
e		types value of e as a symbol+displacement; if the value of e is less than the smallest symbol loaded "0+e" is typed.
e/		opens memory location e.
↑		equivalent to typing . -1/
(1f)		equivalent to typing . +1/
;RA		opens the A-register just as if it were a memory cell.
;RB		opens the B-register just as if it were a memory cell.
;RX		opens the X-register just as if it were a memory cell.
;RE		opens the E-register just as if it were a memory cell.
;RL		opens the L-register just as if it were a memory cell.
;RU		opens the U-register just as if it were a memory cell.
;RP		opens the P-register just as if it were a memory cell.
;RT		opens the T-register just as if it were a memory cell.
;RW		opens switch register just as if it were a memory cell.
;I		opens the status register just as if it were a memory cell.
e;F		if the value of e is between 0 and 77B then same as ;RA for mapping register e; if the value of e is between 100B and 177B then system mapping register e-100B is opened; e is computed modulo 200B.
;U		lists all undefined symbols.

e!	sets breakpoint 0 at location e.
e_1, e_2!	sets breakpoint e_1 at location e_2; e_1 is computed modulo 4.
e;!	clears breakpoint e.
!	clears all breakpoints.
e;G	begins execution at location e.
;P	begins execution at the current location.
e:S	executes e instructions in trace mode; the trace prints L (P) X U A before execution of (P).
e;T	types 3 spaces and waits for user to type a file name from which program and/or symbols are to be loaded; file name is terminated by a (cr); LSIM then types last location loaded +1; e specifies the starting address of the load. Symbolic addresses of any range errors are typed in the form R: address.
;T	same as e;T except that loading is started following the last load; if there was no previous load then loading is started at 1000B.
e;Y	same as e;T except that local symbols are not loaded.
;Y	same as ;T except that local symbols are not loaded.
;C	prints elapsed Logicon 2+2 cpu time in microseconds.
e;C	resets cpu time counter to 0.00; the value of e is ignored except that if e contains any symbols they must be defined.
;J	prints the symbolic address of the last jump instruction.
e;1	sets lower bound for memory search to the value of e.
e;2	sets upper bound for memory search to the value of e.
e;M	sets mask for memory search to the value of e.
e;W	for every location between upper and lower bound, the value of the mask is anded with the contents of the cell and compared with the value of e; if the match is successful, the symbolic address of the cell is typed.

e_1, e_2;K types 3 spaces and waits for the name of an output file. if the name is valid, LSIM writes out a binary core image and the contents on the registers of the given file. If e_1 is absent it is assumed to be zero. e_1 and e_2 specify the range of memory location to be written out.

;K types 3 spaces and waits for the name of a binary input file previously written by the e_1, e_2;K command. If the name is valid, the data from the file is loaded into memory and the central registers. LSIM then types the starting and ending addresses of the load; these should match the values of e_1 and e_2 in the command when the file was written out. A ;P command following this one should restart the program in the same state that it was dumped, except for open files.

e;H types the value of e in hexadecimal. e must be $2^{20}-1$.

;L types 3 spaces and waits for the name of a binary file (new or old) to be used as the simulated drum for SCALL 213B.

e;L closes the current simulated-drum file (does not delete, etc.)

e;N punches an absolute binary paper tape in Logicon 2+2 bootstrap format. The range punched is between the upper and lower bound set by ;1 and ;2 commands. The expression supplied to ;N is the transfer address punched at the end of the tape. After typing e;N turn on the punch and type CR to start the punching operation.

;0 prints all defined symbols and their values in the current radix.

;! types the symbolic addresses of all existing breakpoints.

e;P resumes execution at the current value of P and continues until e breakpoints have been encountered, then breaks, e.g., 2;P will stop at the second breakpoint encountered. If e is absent it is assumed to be 1.

e'' types the value of e in ASCII characters; control characters are preceded by & and 000B (null character) prints as &@.

e <s> defines the symbol as the value of e. s must be previously undefined; if a program already loaded contains a reference to s the newly defined value of s will be inserted into the reference.

e;$ sets $ to the value of e.

$e_1, e_2;Z$ zeros memory between e_1 and e_2; if e_1, is absent it is assumed to be zero.

ERROR MESSAGES

LSIM responds with a ? if it cannot execute or fails to recognize a command. The following error messages are typed while a program is executing if the given error condition occurs.

LSIM Error Messages

LOGICON 2+2 MEM TRAP	an attempt was made to access a real core address greater than 77777B.
READ ERROR	I/O error was detected during reading of the input file for loading.
EARLY EOF	an EOF was encountered on the input file during loading before loading was completed.
ILLEGAL CONTROL WORD	bad control word on the file; usually means a binary file not produced by NARP was being loaded.
SYMBOL TABLE OVERFLOW	LOGICON 2+2 uses four symbol tables, one for external symbols, one for undefined symbols, one for defined symbols, and one for external expressions; this message indicates that one of the tables overflowed; the current limit is 100 undefined, and 300 defined symbols; the external symbol table is reset at the start of each load, the other 3 tables are cumulatively filled during the total load.
ILLEGAL MEMORY ACCESS	usually indicates an attempt to reference a cell containing a reference to an undefined symbol; this message can also occur when referencing uncleared memory that was

	not loaded; the message also gives the address of the instruction attempting to make the memory reference.
2 + 2 READ VIOLATION	LOGICON 2+2 mapping unit protection violation.
2 + 2 WRITE VIOLATION	LOGICON 2+2 mapping unit protection violation.
2 + 2 EXECUTE VIOLATION	LOGICON 2+2 mapping unit protection violation.
HALT AT	a 144477B instruction was executed.
ILLEGAL INSTRUCTION	an attempt was made to execute a privileged instruction in user mode.
UNIMPLEMENTED INSTRUCTION	an attempt was made to execute an unrecognized extended op-code; this includes LOGICON 2+2 instructions not implemented in the simulator such as I/O instructions.
INSTR. PANIC AT	in the course of simulation the simulator executed an illegal 940 instruction; this is usually caused by illegal parameters in a CALL; could also be a bug in the simulator.
MEMORY PANIC AT	in the course of simulation the simulator caused a 940 system memory panic; this message should be caused only by a bug in the simulator.

USING LSIM ON TYMSHARE

Calling LSIM

LSIM is on the system as a public GO file under user name P. Thus, it can be called by typing GO (P) LSIM (in the exec) or just (P) LSIM. When LSIM is ready to accept commands it will type a (LF) .

Programming Considerations

LSIM simulates a full 32K LOGICON 2+2 machine, but it has a working set of only 8K on the 940. The simulator demand pages to get additional memory. Thus, programs larger than 6K will execute rather slowly.

Escapes

LSIM execution can be interrupted by depressing the escape button. This will cause a bell and (LF) to be typed when the interrupt is recognized. Two successive escapes will cause an exit to the exec. The CONTINUE command in Tymshare will usually restart LSIM in usable condition.

MULTIPLE PROCESSING

Since the LOGICON 2+2 System will require two processors, the AP and CP, a feature has been added to LSIM to allow operation of both of these processors. LSIM now has two modes: CP and AP. Several commands have been added for determining the current mode and changing it.

Modes

LSIM is initialized in AP mode and will remain in that mode unless a ;A or ;B instruction is executed to change it. When in AP mode, LSIM memory access is through the simulated map and a maximum hardware address of 32K is allowed. In CP mode unmapped access to 8K of memory separate from the AP memory is allowed. LSIM maintains a separate set of registers for each processor and the commands in Paragraph 6. 1 apply to the registers or memory of the processor whose mode is currently in effect. Except for the special symbols . and $, there is no distinction between symbols in the two modes. The only other things unique to each mode are the breakpoints: LSIM keeps a separate set of these for each mode.

Instructions executed will affect registers and memory of the processor whose current mode is in effect.

Changing Modes. The current mode will change when a ;A command is executed or when the instruction interval count reaches -1. This count is set by the e;B command and is reset to its original value after it reaches -1. It is decremented whenever an instruction is executed in either mode.

Mode Commands

e;A	sets the current mode to CPU if e = 0 or CIOP if e ≠ 0. To avoid confusion of listings LSIM will print the new mode in effect following this command.
;B	locks the current mode so that another e;B or e;A command is required before any commands or instructions can effect the other processor being simulated.
e;B	sets the instruction interval count to the value of e.
;E	prints the current mode.

Programming Considerations

The memory for both the CPU and CIOP are demand paged from the same 8K working set in the 940. This will make processing of programs larger than 8K total for both processors very slow.

It takes about 150 sec. of 940 CPU time to change modes, thus using an instruction interval of 0 will result in 25 to 50 percent slower execution. By increasing this interval to 100 or more the slowdown can be reduced to .5 percent or less.

APPENDIX A

LOGICON 2+2 CHARACTER SET

ASCII CODE	A/N CHARS:	ASCII CODE	SPECIAL CHARS:	
101	A	012		LINE FEED
102	B	015		CARRIAGE RETURN
103	C			
104	D	040		SPACE
105	E	041	!	EXCLAMATION POINT
106	F	042	"	DOUBLE QUOTE
107	G	043	#	NUMBER SIGN
110	H	044	$	DOLLAR SIGN
111	I	045	%	PERCENT SIGN
112	J	046	&	AMPERSAND
113	K	047	'	APOSTROPHE OR SINGLE QUOTE
114	L	050	(	LEFT PARENTHESIS
115	M	051	)	RIGHT PARENTHESIS
116	N	052	*	ASTERISK
117	O	053	+	PLUS SIGN
120	P	054	,	COMMA
121	Q	055	-	MINUS SIGN OR HYPHEN
122	R	056	.	PERIOD OR DECIMAL POINT
123	S	057	/	SLASH
124	T	072	:	COLON
125	U	073	;	SEMICOLON
126	V	074	<	LESS THAN
127	W	075	=	EQUALS SIGN
130	X	076	>	GREATER THAN
131	Y	077	?	QUESTION MARK
132	Z	100	@	AT SIGN
		133	[	LEFT BRACKET
060	0	134	\	BACKWARDS SLASH
061	1	135	]	RIGHT BRACKET
062	2	136	↑	UP ARROW
063	3	137	←	LEFT ARROW
064	4			
065	5			
066	6	376		END-OF-FILE
067	7			
070	8			
071	9			

APPENDIX B

LOGICON 2+2 MNEMONICS IN ALPHABETICAL ORDER

MNEM	FUNCTION	PAGE NO.	MNEM	FUNCTION	PAGE NO.
ACA	Arithmetic Compare A	3-57	FCP	Floating Compare	3-57
ACE	Arithmetic Compare E	3-57	FCPS	Floating Compare, String	3-58
ACU	Arithmetic Compare U	3-56	FDV	Floating Divide	3-41
ACX	Arithmetic Compare X	3-56	FDVS	Floating Divide, Stack	3-43
ADA	Add to A	3-28	FIX	Fix Floating Point Number	3-43
ADAI	Add to A, Immediate	3-28	FLOAT	Float Integer	3-44
ADAS	Add to A, Stack	3-33	FMP	Floating Multiply	3-41
ADDM	Add to Memory	3-30	FMPS	Floating Multiply, Stack	3-43
ADE	Add to E	NA	FNEG	Floating Negate	3-44
ADU	Add to U	3-27	FSB	Floating Subtract	3-40
ADUI	Add to U, Immediate	3-27	FSBS	Floating Subtract, Stack	3-42
ADUS	Add to U, Stack	3-35			
ADX	Add to X	3-26	GCI	Get Character and Increment	3-15
ADXI	Add to X, Immediate	3-26	GCIT	Get Character and Increment with Test	3-15
ADXIS	Add to X, Immediate and Skip	3-27	GFC	Get First Character	3-14
ADXS	Add to X, Stack	3-36	GFCT	Get First Character with Test	3-14
AJP	A Jump	3-65			
ALA	(ARA) Arithmetic Left/Right Shift A	3-52	HLT	Halt*	3-12
ALU	(ARU) Arithmetic Left/Right Shift U	3-51			
ALUA	(ARUA) Arithmetic Left/Right Shift U, A	3-53	ICALL	Indirect Call	3-69
ANA	AND with A	3-45	ICI	Insert Character and Increment	3-16
ANAI	AND with A, Immediate	3-45	ICIT	Insert Character and Increment with Test	3-17
ANAS	AND with A, Stack	3-49	IFC	Insert First Character	3-16
ANU	AND with U	3-45	IFCT	Insert First Character with Test	3-16
ANUA	AND U with Memory to A	3-45	IJMP	Indirect Jump	3-66
ANUI	AND with U, Immediate	3-45	IJSPM	Indirect Jump, Store P in Memory	3-69
ANX	AND with X	3-45	IJSPX	Indirect Jump, Store P in X	3-69
ANXS	AND with X, Stack	3-50	IJXN	Increment and Jump if x ≠ 0	3-66
			IOC	Input/Output Control*	3-10
CALL	Subroutine CALL Linkage	3-67	IRTRN	Interrupt Return*	3-12
CLA	Clear A	3-23			
CLE	Clear E	3-23	JMI	Jump if A Minus	3-64
CLRBA	Clear Bit in A	3-47	JMP	Jump Unconditionally	3-63
CLRBM	Clear Bit in Memory	3-48	JNZ	Jump if A Non Zero	3-63
CLU	Clear U	3-23	JPL	Jump if A Plus	3-63
CLX	Clear X	3-23	JSPM	Jump, Store P in Memory	3-67
CMPBA	Complement Bit in A	3-48	JSPX	Jump, Store P in X	3-66
CMPBM	Complement Bit in Memory	3-48	JZE	Jump if A Zero	3-63
CPRS	Compare Strings	3-13			
			LCA	Logical Compare A	3-59
DIN	Direct Input*	3-9	LCE	Logical Compare E	3-59
DJXN	Decrement and Jump if x ≠ 0	3-66	LCU	Logical Compare U	3-58
DLINK	Remove Item from List	3-25	LCX	Logical Compare X	3-58
DOUT	Direct Output*	3-9	LDA	Load A	3-4
DSK	Skip after Next Instruction	3-62	LDAC	Load A from Console Switches*	3-8
DVA	Divide A	3-29	LDAEA	Load A with Effective Address	3-4
DVAS	Divide A, Stack	3-34	LDAI	Load A, Immediate	3-4
DVUA	Divide U and A	3-29	LDAOM	Load A from Other Memory*	3-18
DVUAS	Divide U and A, Stack	3-35	LDAOMF	Load A from Other Memory with Force*	3-18
			LDASM	Load A through Specified Map*	3-19
EJP	E Jump	3-65	LDASMF	Load A through Specified Map with Force*	3-19
			LDB	Load B	3-20
FAD	Floating Add	3-40	LDBTL	Load B, T, and L	3-21
FADS	Floating Add, Stack	3-41			

N/A - Not Applicable
*Privileged Instruction

MNEM	FUNCTION	PAGE NO.	MNEM	FUNCTION	PAGE NO.
LDC	Load Character	3-13	RFDVS	Reverse Floating Divide, Stack	3-43
LDE	Load E	3-5	RFSB	Reverse Floating Subtract	3-41
LDEI	Load E, Immediate	3-5	RFSBS	Reverse Floating Subtract, Stack	3-42
LDF	Load Floating	3-23	RIL	Release Interrupt Lockout*	3-11
LDM	Load Multiple	3-6	RLA	(RRA) Rotate Left/Right A	3-52
LDMAP	Load Map*	3-8	RLU	(RRU) Rotate Left/Right U	3-52
LDSP	Load Stack Pointers	3-21	RLUA	(RRUA) Rotate Left/Right U, A	3-54
LDU	Load U	3-3	RNEG	Register Negate	3-25
LDUI	Load U, Immediate	3-4	RSBA	Reverse Subtract A	3-28
LDX	Load X	3-1	RSBAS	Reverse Subtract A, Stack	3-34
LDXEA	Load X with Effective Address	3-3	RSBX	Reverse Subtract X	3-27
LDXI	Load X, Immediate	3-3	RSBXS	Reverse Subtract X, Stack	3-36
LDXSM	Load X through Specified Map*	3-19	RSUB	Register Subtract	3-30
LINK	Link Item into List	3-24	RTRN	Subroutine Return Linkage	3-67
LLA	(LRA) Logical Left/Right Shift A	3-52	RTSB	Reverse Triple Subtract	3-32
LLDB	Locate Leading Dirty Bit*	3-9	RTSBS	Reverse Triple Subtract, Stack	3-38
LLO	Locate Leading One	3-54	RXCH	Register Exchange	3-25
LLU	(LRU) Logical Left/Right Shift U	3-51			
LLUA	(LRUA) Logical Left/Right Shift U, A	3-53	SBA	Subtract from A	3-28
LLX	(LRX) Logical Left/Right Shift X	3-51	SBAS	Subtract from A, Stack	3-33
LLUAE	(LRUAE) Logical Left/Right Shift U, A, E	3-53	SBU	Subtract from U	3-28
LSABM	Load Sign of A from Bit in Memory	3-22	SBUS	Subtract U, Stack	NA
			SBX	Subtract from X	3-27
MDEC	Memory Decrement, Skip	3-31	SBXS	Subtract X, Stack	3-36
MINC	Memory Increment, Skip	3-30	SCALL	System Call	3-68
MOVE	Move Word String	3-23	SETBA	Set Bit in A	3-47
MPA	Multiply A	3-28	SETBM	Set Bit in Memory	3-48
MPAS	Multiply A, Stack	3-34	SIL	Set Interrupt Lockout*	3-10
MPX	Multiply X	3-27	SIM	Set Interrupt Mask*	3-9
MPXS	Multiply X, Stack	3-36	SKAE	Skip if A Equal	3-55
MRGM	Merge Mode Bits*	3-19	SKAEI	Skip if A Equal, Immediate	3-55
MSK	Memory Skip	3-59	SKAN	Skip if A Not Equal	3-55
MSKM	Mask Mode Bit*	3-8	SKANI	Skip if A Not Equal, Immediate	3-56
			SKNCO	Skip if No Carryout	3-62
NFAD	Negative Floating Add	3-40	SKNOF	Skip if No Overflow	3-62
NFADS	Negative Floating Add, Stack	3-41	SKOA	Skip if One in A_n	3-60
NORM	Normalize Floating Point Number	3-44	SKOM	Skip if One in Memory, Y_n	3-61
NTAD	Negative Triple Add	3-32	SKXEI	Skip if X Equal, Immediate	3-54
NTADS	Negative Triple Add, Stack	3-37	SKXNI	Skip if X Not Equal, Immediate	3-55
			SKZA	Skip if Zero in A_n	3-60
ORA	OR with A	3-46	SKZM	Skip if Zero in Memory, Y_n	3-61
ORAI	OR with A, Immediate	3-46	SRTRN	System Return*	3-11
ORAS	OR with A, Stack	3-50	SSABM	Store Sign of A in Bit in Memory	3-22
			STA	Store A	3-5
POPM	Pop Multiple	3-7	STAOM	Store A in Other Memory*	3-18
POPN	Pop Null	3-20	STASM	Store A through Specified Map*	3-19
PUSHM	Push Multiple	3-6	STB	Store B	3-21
PUSHN	Push Null	3-7	STC	Store Character	3-13
			STE	Store E	3-5
RADD	Register Add	3-29	STF	Store Floating	3-24
RAND	Register AND	3-46	STM	Store Multiple	3-6
RCPY	Register Copy	3-25	STSP	Store Stack Pointers	3-21
RDS	Read Status	3-26	STU	Store U	3-4
RDVA	Reverse Divide A	3-29	STX	Store X	3-3
RDVAS	Reverse Divide A, Stack	3-35	STZ	Store Zeros	3-22
RFDV	Reverse Floating Divide	3-41	SUBM	Subtract Memory	3-30

N/A - Not Applicable
*Privileged Instruction

MNEM	FUNCTION	PAGE NO.	MNEM	FUNCTION	PAGE NO.
TAD	**Triple Add**	3-31	TSL	Test and Set Lock	3-62
TADS	**Triple Add, Stack**	3-37	TSLOM	Test and Set Lock in Other Memory*	3-18
TDV	**Triple Divide**	3-33			
TDVF	**Triple Divide, Fractional**	3-33	UJP	U Jump	3-64
TDVFS	**Triple Divide, Fractional, Stack**	3-39			
TDVS	**Triple Divide, Stack**	3-39	XAM	Exchange A and Memory	3-5
TMP	**Triple Multiply**	3-32	XJP	X Jump	3-64
TMPF	**Triple Multiply, Fractional**	3-33	XRA	EXCLUSIVE OR with A	3-46
TMPFS	**Triple Multiply, Fractional, Stack**	3-38	XRAI	EXCLUSIVE OR with A, Immediate	3-46
TMPS	**Triple Multiple, Stack**	3-38	XRAS	EXCLUSIVE OR with A, Stack	3-50
TNEG	**Triple Negate**	3-40	XSA	Extend Sign of A	3-26
TSB	**Triple Subtract**	3-32	XXM	Exchange X and Memory	3-3
TSBS	**Triple Subtract, Stack**	3-37			

***Privileged Instruction**

APPENDIX C

LOGICON 2+2 MNEMONICS BY FORMAT

FORMAT 1					
OpCode	Format	Instruction	OpCode	Format	Instruction
00	1A, B, C, D	LDX	15	1A, C, D	JNZ
01	1A, C, D	STX	16	1A, B, C, D	SBX
02	1A, B, C, D	LDU	17	1A, C, D	JPL
03	1A, C, D	STU	20	1A, B, C, D	ANA
04	1A, B, C, D	LDA	21	1A, C, D	JMI
05	1A, C, D	STA	22	1A, B, C, D	ORA
06	1A, B, C, D	LDE	23	1A, C, D	JSPX
07	1A, C, D	STE	24	1A, B, C, D	XRA
10	1A, B, C, D	ADA	25	1A, C, D	IJXN
11	1A, C, D	JMP	26	1A, B, C, D	SKAE
12	1A, B, C, D	ADX	27	1A, C, D	DJXN
13	1A, C, D	JZE	30	1A, B, C, D	SKAN
14	1A, B, C, D	SBA			

FORMAT 2			
OpCode(Mod)	Instruction	OpCode(Mod)	Instruction
03(0)	MOVE	23(0)	ADAS
05(0)	RXCH XU	24(0)	SBAS
06(0)	RXCH XA	25(0)	ADUS
07(0)	RXCH XE	25(1)	SBUS
10(0)	RXCH UA	25(2)	RSBAS
11(0)	RXCH UE	26(0)	MPXS
12(0)	RXCH AE	26(1)	MPAS
14(0)	XSA	27(0)	DVUAS
15(0)	RDS	27(1)	DVAS
20(0)	ADXS	27(3)	RDVAS
20(1)	RSBXS	30(0)	FADS
21(0)	SBXS	30(1)	FSBS
22(0)	ANXS	30(2)	RFSBS
22(1)	ANAS	30(3)	NFADS
22(2)	ORAS	31(0)	FMPS
22(3)	XRAS	31(1)	TMPS

FORMAT 2 (Continued)			
OpCode(Mod)	Instruction	OpCode(Mod)	Instruction
31(3)	TMPFS	47(0)	DSK
32(0)	FDVS	50(0)	RTRN
32(1)	TDVS	52(0)	CPRS
32(2)	RFDVS	54(0)	FNEG
32(3)	TDVFS	55(0)	MRGM*
33(0)	TADS	56(0)	MSKM*
33(1)	TSBS	57(0)	LDAC*
33(2)	RTSBS	60(0)	LDMAP*
33(3)	NTADS	64(0)	IRTRN*
36(0)	FCPS	67(0)	RROM
40(0)	DNEG	70(0)	LLDB*
41(0)	FIX	71(0)	IOC*
42(0)	FLOAT	72(0)	SIL*
43(0)	NORM	73(0)	RIL*
44(0)	LLI(LLO)	74(0)	DIN*
45(0)	SKNOF	75(0)	DOUT*
46(0)	SKNCO	76(0)	FCPS
		77(0)	HLT*

*Privileged Instruction

FORMAT 3		FORMAT 4		FORMAT 5	
SCALL	System Call	OpCode	Instruction	OpCode	Instruction
		0	PUSHM	1	LSABM
		1	POPM	2	SSABM
		2	SRTRN*	3	SETBM
		3	SETBA	4	CLRBM
		4	CLRBA	5	CMPBM
		5	CMPBA	6	SKOM
		6	SKOA	7	SKIM
		7	SKIA		

* = Privileged Instruction

FORMAT 6					
OpCode	Format	Instruction	OpCode	Format	Instruction
01, 41	6D	LDM	25(3)	6A	TDVF
02, 42	6D	STM	26(0)	6A	TAD
04(0)	6A	LDXEA	26(1)	6A	TSB
04(2)	6A	LDAEA	26(2)	6A	RTSB
05(0)	6A	XXM	26(3)	6A	NTAD
05(2)	6A	XAM	27(0)	6A, G	ANX
06(0)	6A, G	PUSHN	27(1)	6A, G	ANU
06(1)	6A, G	POPN	27(2)	6A, G	ADE
07(0)	6A, G	LDB	27(3)	6A, G	ANUA
10(0)	6A	STB	30(SC)	6A, G	ACX
10(1)	6A	STSP	31(SC)	6A, G	ACU
11(0)	6A	LDSP	32(SC)	6A, G	ACA
11(1)	6A	LDBTL	33(SC)	6A, G	ACE
12(N)	6A	STZ	34(SC)	6A, G	LCX
13(0)	6A, G	MPX	35(SC)	6A, G	LCU
13(1)	6A, G	MPA	36(SC)	6A, G	LCA
14(0)	6A, G	ADU	37(SC)	6A, G	LCE
14(1)	6A, G	SBU	40(SC)	6A, G	MSK
14(2)	6A, G	RSBA	41	6D	LDM
15(0)	6A, G	RSBX	42	6D	STM
16(0)	6A, G	DVUA	43(0)	6A	TSL
16(1)	6A, G	DVA	44(JC)	6A	XJP
16(3)	6A, G	RDVA	45(JC)	6A	UJP
17(0)	6A	ADDM	46(JC)	6A	AJP
17(1)	6A	SUBM	47(JC)	6A	EJP
20(SC)	6A	MINC	50(JC)	6A	TJP
21(SC)	6A	MDEC	55(0)	6A	IJMP
22(SC)	6A	FCP	55(1)	6A	IJSPX
23(0)	6A	FAD	55(2)	6A	IJSPM
23(1)	6A	FSB	55(3)	6A	ICALL
23(2)	6A	RFSB	56(7)	6A	JSPM
24(0)	6A	FMP	63(7)	6A	CALL
25(0)	6A	FDV	64(0)	6B, C	LDC
25(1)	6A	TDV	64(1)	6B, C	STC
25(2)	6A	RFDV	65(0)	6A	GFC

SC = JC = Skip/Jump Conditions

FORMAT 6 (Continued)					
OpCode	Format	Instruction	OpCode	Format	Instruction
65(1)	6A	GFCT	73(0)	6E	LDASM*
65(2)	6A	GCI	73(1)	6E	LDASMF*
65(3)	6A	GCIT	73(2)	6E	STASM*
65(4)	6A	IFC	73(3)	6E	LDXSM*
65(5)	6A	IFCT	74(0)	6F	LDAOM*
65(6)	6A	ICI	74(1)	6F	LDAOMF*
65(7)	6A	ICIT	74(2)	6F	STAOM*
67(0)	6A	LINK	74(3)	6F	TSLOM*
67(1)	6A	DLINK	75(0)	6A, G	SIM*

* = Privileged Instruction

FORMAT 7		
OpCode	Format	Instruction
0	7A	RCPY
1	7A	RADD
2	7A	RSUB
3	7A	RAND
-	7B	RNEG

FORMAT 8	
OpCode	Instruction
0	LLX/LRX
1	ALU/ARU
2	LLU/LRU
3	RLU/RRU
4	ALA/ARA
5	LLA/LRA
6	RLA/RRA

FORMAT 9	
OpCode	Instruction
0	LLUAE/LRUAE
1	ALUA/ARUA
2	LLUA/LRUA
3	RLUA/RRUA

FORMAT 10	
OpCode	Instruction
00	LDXI
00	CLX
01	LDUI
01	CLU
02	LDAI
02	CLA
03	LDEI
03	CLE
04	ADXI
05	ADXIS
06	ADUI
07	ADAI
10	ANUI
11	ANAI
12	ORAI
13	XRAI
14	SKXEI
15	SKXNI
16	SKAEI
17	SKANI

APPENDIX D

LOGICON 2+2 MNEMONICS ARRANGED BY REGISTER

INSTRUCTIONS INVOLVING A-REGISTER									
LOADS	CLA	**LDA**	**LDAEA**	**LDAI**	**LDC**	**LDM**	**POPM**	**RDS**	LDAC LDAOM LDAOMF LDASM LDASMF
STORES		STA	STC	STM	PUSHM	STAOM	STASM		
COMPARES		ACA	LCA	CPRS					
JUMP/SKIPS		AJP JMI	TJP SKAE	SKOA SKAEI	SKZA SKAN	JZE SKANI	JNZ	JPL	
EXCHANGES		RXCH	XAM						
INTER-REGISTER		RADD	RSUB	RNEG	RCPY	RAND			
FIXED-POINT		ADA SBA MPA DVA	ADAI SBAS MPAS DVAS	ADAS SUBM	ADDM DVUA	DVUAS RSBA RDVA	 RSBAS RDVAS		
FLOATING POINT AND THREE WORD ARITHMETIC		FAD FSB FMP FDV FIX	FADS FSBS FMPS FDVS FLOAT	FCP FCPS NFAD NFADS NORM	FNEG RFSB RFSBS RFDV RFDVS	NTAD NTADS RTSB RTSBS TAD	TADS TDV TDVF TDVFS TDVS TMP	TMPF TMPFS TMPS TNEG TSB TSBS	
LOGICAL		ANA XRA	ANAI XRAI	ANAS XRAS	ANUA CMPBA	ORA LCA	ORAI	ORAS	
SHIFTS		ALA LLUA	ARA LRUA	ALUA RLA	ARUA RRA	LLA RLUA	LRA RRUA	LLUAE LLO	LRUAE
MOVE		MOVE							
BIT/BYTE		GCI IFC LSABM	GCIT IFCT SSABM	GFC LDC SETBA	GFCT STC CLRBA	ICI XSA CMPBA	ICIT		

INSTRUCTIONS INVOLVING X-REGISTER									
LOADS		LDX	LDXEA	LDXI	LDM	POPM	CLX		
STORES		STX	STM	PUSHM					
COMPARES		ACX	LCX	CPRS					
JUMPS/SKIPS		XJP	SKXEI	SKXNI	IJXN	DJXN	JSPX	ADXIS	IJSPX
EXCHANGES		RXCH	XXM						
INTER-REGISTER		RADD	RSUB	RNEG	RAND	RCPY			
FIXED-POINT		ADX	ADXI	ADXIS	SBX	SBXS	MPX	ADXS MPXS	RSBX RSBXS
LOGICAL		ANX	ANXS						
SHIFTS		LLX	LRX						
MOVE		**MOVE**							

INSTRUCTIONS INVOLVING U-REGISTER							
LOADS	LDU	LDUI	LDM	POPM	CLU		
STORES	STU	STM	PUSHM				
COMPARES	ACU	LCU	CPRS				
JUMPS/SKIPS	UJP	TJP					
EXCHANGES	RXCH						
INTER-REGISTER	RADD	RSUB	RNEG	RAND	RCPY		
FIXED-POINT	ADU ADUS	ADUI DVA	DVAS	SBU DVUA	SBUS RDVA	MPA RDVAS	MPAS
FLOATING POINT AND THREE WORD ARITHMETIC	FAD FSB FMP FDV FIX	FADS FSBS FMPS FDVS FLOAT	FCP FCPS NFAD NFADS NORM	FNEG RFSB RFSBS RFDV RFDVS	NTAD NTADS RTSB RTSBS TAD	TADS TDV TDVF TDVFS TDVS TMP	TMPF TMPFS TMPS TNEG TSB TSBS
LOGICAL	ANU	ANUI	ANUA				
SHIFTS	ALU LLUA RLUA	ARU LRUA RRUA	ALUA LLUAE	ARUA LRUAE	LLU RLU	LRU RRU	
MOVE	MOVE						
BIT	XSA						

INSTRUCTIONS INVOLVING E-REGISTER							
LOADS	LDE	LDEI	LDM	POPM	CLE		
STORES	STE	STM	PUSHM				
COMPARES	ACE	LCE					
JUMPS	EJP	TJP					
EXCHANGE	RXCH						
INTER-REGISTER	RADD	RSUB	RNEG	RAND	RCPY		
FIXED-POINT	ADE						
FLOATING POINT AND THREE WORD ARITHMETIC	FAD FSB FMP FDV FIX	FADS FSBS FMPS FDVS FLOAT	FCP FCPS NFAD NFADS NORM	FNEG RFSB RFSBS RFDV RFDVS	NTAD NTADS RTSB RTSBS TAD	TADS TDV TDVF TDVFS TDVS TMP	TMPF TMPFS TMPS TNEG TSB TSBS
SHIFTS	LLUAE	LRUAE					

INSTRUCTIONS INVOLVING B-REGISTER					
LOADS	LDB	LDSP	LDBTL		
STORES	STB	STSP			
JUMPS	CALL	RTRN	ICALL		
INTER-REGISTER	RADD	RSUB	RNEG	RCPY	RAND

INSTRUCTIONS INVOLVING T-REGISTER						
LOADS	LDSP	POPM	POPN	LDBTL		
STORES	STSP	PUSHM	PUSHN			
JUMPS	CALL	RTRN	ICALL			
INTER-REGISTER	RADD	RSUB	RNEG	RCPY	RAND	
FIXED-POINT	ADAS	SBAS	RSBAS	MPAS	DVAS	RDVAS
FLOATING POINT	FADS	FSBS	RFSBS	FMPS	FDVS	RFDVS
LOGICAL	ANAS	ORAS	XRAS			

INSTRUCTIONS INVOLVING L-REGISTER					
LOADS	LDSP	LDBTL			
STORES	STSP				
INTER-REGISTER	RADD	RSUB	RNEG	RCPY	RAND

APPENDIX E

LOGICON 2+2 MNEMONICS ARRANGED BY FUNCTION

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		LOAD AND STORE			
00	1A, B, C, D	LDX	(Y) → X	3-1	
04(0)	6A	LDXEA	Y → X	3-3	
00	10	LDXI	LIT9 → X	3-3	
01	1A, C, D	STX	(X) → Y	3-3	
05(0)	6A	XXM	(Y) → X; (X) → Y	3-3	
02	1A, B, C, D	LDU	(Y) → U	3-3	
01	10	LDUI	LIT9 → U	3-4	
03	1A, C, D	STU	(U) → Y	3-4	
04	1A, B, C, D	LDA	(Y) → A	3-4	
04(2)	6A	LDAEA	Y → A	3-4	
02	10	LDAI	LIT9 → A	3-4	
05	1A, C, D	STA	(A) → Y	3-5	
05(2)	6A	XAM	(Y) → A; (A) → Y	3-5	
06	1A, B, C, D	LDE	(Y) → E	3-5	
03	10	LDEI	LIT9 → E	3-5	
07	1A, C, D	STE	(E) → Y	3-5	
01, 41	6D	LDM	(Y...Y+N, 0 ≤ N ≤ 3) → X, U, A AND/OR E	3-6	
02, 42	6D	STM	(X, U, A AND/OR E) → Y...Y+N, 0 ≤ N ≤ 3	3-6	
0	4A	PUSHM	(X, U, A AND/OR E) → (T)...(T)+N; (T)+N+1 → T	3-6	1
1	4A	POPM	((T)-1)...((T)-N-1) → E, A, U AND/OR X; (T)-N-1 → T	3-7	2
06(0)	6A, 6G	PUSHN	(T)+(Y) → T	3-7	1, 2
06(1)	6A, 6G	POPN	(T)-(Y) → T	3-20	1, 2
07(0)	6A, 6G	LDB	(Y) → B	3-20	
10(0)	6A	STB	(B) → Y	3-21	
11(0)	6A	LDSP	(Y, Y+1, Y+2) → B, T, L	3-21	1, 2

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		LOAD AND STORE (CONTINUED)			
11(1)	6A	LDBTL	(Y, Y+1, Y+2) → B, T, L	3-21	
10(1)	6A	STSP	(B, T, L) → Y, Y+1, Y+2	3-21	
12(N)	6A	STZ	0 → Y...Y+N, 0 ≤ N ≤ 7	3-22	
1	5	LSABM	$(Y_N) \leftarrow A_0, A_1$-15 UNCHANGED	3-22	11
2	5	SSABM	$(A_0) \rightarrow Y_N$	3-22	11
03(0)	2	MOVE	MOVE WORDS: (X) = SOURCE ADDRESS (A) = DESTINATION ADDRESS (U) = NO. OF WORDS	3-23	14
00	10	CLX	0 → X	3-23	
01	10	CLU	0 → U	3-23	
02	10	CLA	0 → A	3-23	
03	10	CLE	0 → E	3-23	
41(3)	6D	LDF	(Y, Y+1, Y+2) → U, A, E	3-23	
42(3)	6D	STF	(U, A, E) → Y, Y+1, Y+2	3-24	
01	6D	LDD	(Y, Y+1) → U, A	3-24	
02	6D	STD	(U, A) → Y, Y+1	NA	
67(0)	6A	LINK	If $(Y)_{0-7}$ = X, normal return If $(Y+1+X)_{8-15} \neq 255$, normal return If $(Y)_{0-7} = 255$, X → $(Y)_{0-7}$ and $(Y)_{8-15}$ $(A)_{8-15} \rightarrow (Y+1+X)_{0-7}$ skip return	3-24	23

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		LOAD AND STORE (CONTINUED)			
67(1)	6A	DLINK	If $(Y)_{0-7} \neq 255$, $X \rightarrow ((Y)_{0-7} + Y + 1)_{8-15}$ $X \rightarrow (Y)_{0-7}$ $(A)_{8-15} (Y+1+X)_{0-7}$ skip return If $(Y)_{8-15} = 255$, normal return If $(Y)_{8-15} \neq 255$, $(Y)_{8-15} \rightarrow X$ $((Y)_{8-15}+Y+1)_{0-7} \rightarrow A$ $((Y)_{8-15}+Y+1)_{8-15} \rightarrow (Y)_{8-15}$ $255 \rightarrow (X+1+Y)$ skip return	3-25	23
		INTERREGISTER			
0	7A	RCPY	(S) → D	3-25	3,4
-	7B	RNEG	-(S) → D	3-25	3,5
05(0)	2	RXCH XU	(X) → U;(U) → X	3-25	
06(0)	2	RXCH XA	(X) → A;(A) → X	3-25	
07(0)	2	RXCH XE	(X) → E;(E) → X	3-25	
10(0)	2	RXCH UA	(U) → A;(A) → U	3-25	
11(0)	2	RXCH UE	(U) → E;(E) → U	3-25	
12(0)	2	RXCH AE	(A) → E;(E) → A	3-25	
14(0)	2	XSA	$(A_0) \rightarrow U$	3-26	
15(0)	2	RDS	(STAT) → A	3-26	
		FIXED-POINT ARITHMETIC			
12	1A, B, C, D	ADX	(X)+(Y) → X	3-26	
04	10	ADXI	(X)+LIT9 → X	3-26	
05	10	ADXIS	(X)=LIT9 → X, SKIP IF SIGN CHANGED OR IF RESULT = 0	3-27	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
	FIXED-POINT ARITHMETIC (CONTINUED)				
16	1A,B,C,D	SBX	(X)-(Y) → X	3-27	
15(1)	6A, 6G	RSBX	(Y)-(X) → X	3-27	
13(0)	6A, 6G	MPX	(X)*(Y) → X	3-27	
14(0)	6A, 6G	ADU	(U)+(Y) → U	3-27	6
06	10	ADUI	(U)+LIT9 → U	3-27	6
14(1)	6A, 6G	SBU	(U)_(Y) → U	3-28	6
10	1A,B,C,D	ADA	(A)+(Y) → A	3-28	6
07	10	ADAI	(A)+LIT9 → A	3-28	6
14	1A,B,C,D	SBA	(A)-(Y) → A	3-28	6
14(2)	6A, 6G	RSBA	(Y)-(A) → A	3-28	6
13(1)	6A, 6G	MPA	(A)*(Y) → (U,A)	3-28	7
16(0)	6A, 6G	DVUA	(U,A)/(Y) → A, REMAINDER → U	3-29	8
16(1)	6A, 6G	DVA	(A)/(Y) A, REMAINDER → U	3-29	8
16(3)	6A, 6G	RDVA	(Y)/(A)→A, REMAINDER → U	3-29	8
27(2)	6A, 6G	ADE	(E)+(Y) → E	NA	
1	7A	RADD	(D)+(S) → D	3-29	3,4
2	7A	RSUB	(D)-(S) → D	3-30	3,4
17(0)	6A	ADDM	(Y)+(A) → Y	3-30	6
17(1)	6A	SUBM	(Y)-(A) → Y	3-30	6
20(SC)	6A	MINC	(Y)+1 → Y, SKIP ON CONDITION	3-30	
21(SC)	6A	MDEC	(Y)-1 → Y, SKIP ON CONDITION	3-31	
26(0)	6A	TAD	(U,A,R)+(Y,Y+1, Y+2) → U,A,E	3-31	22
26(3)	6A	NTAD	(U,A,E)-(Y,Y+1, Y+2) → U,A,E	3-32	22
26(1)	6A	TSB	(U,A,E)-(Y,Y+1, Y+2) → U,A,E	3-32	22
26(2)	6A	RTSB	(Y,Y+1,Y+2) -(UAE)→ U,A,E	3-32	22
24(1)	6A	TMP	(U,A,E)*(Y,Y+1, Y+2) → U,A,E	3-32	22
24(3)	6A	TMPF	(U,A,E)*(Y,Y+1, Y+2) → U,A,E	3-33	22
25(1)	6A	TDV	(U,A,E)/(Y,Y+1, Y+2) → U,A,E	3-33	22

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		FIXED-POINT ARITHMETIC (CONTINUED)			
25(3)	6A	TDVF	(U,A,E)/(Y,Y+1, Y+2) → U,A,E	3-33	22
23(0)	2	ADAS	((T)-1)+(A) A, (T) - 1 → T	3-33	2,6
24(0)	2	SBAS	(A)-((T)-1) → A, (T)-1 → T	3-33	2,6
25(2)	2	RSBAS	((T)-1)-(A) →A, (T)-1 →T	3-34	2,6
26(1)	2	MPAS	(A)*((T)-1) → U, A;(T)-1→ T	3-34	2,7
27(1)	2	DVAS	(A)/((T)-1) → A, REMAINDER → U,(T)-1 → T	3-34	2,8
27(3)	2	RDVAS	((T)-1)/(A) → A, REMAINDER → U,(T)-1 → T	3-35	2,8
25(0)	2	ADUS	((T)-1)+(U)→ U, (T)-1 → T →	3-35	2,6
25(1)	2	SBUS	(U)-((T)-1) → U, -1 → T	NA	2,6
27(0)	2	DVUAS	(U,A)/((T)-1) → A, REMAINDER → U, (T)-1 → T	3-35	2,8
20(0)	2	ADXS	((T)-1)+)X) →X, (T)-1 → T	3-36	2,6
21(0)	2	SBXS	(X)-((T)-1), →X, (T)-1 → T	3-36	2,6
20(1)	2	RSBXS	((T)-1)-(X) →X, (T)-1 → T	3-36	2,6
26(0)	2	MPXS	(X)*((T)-1) → X, (T)-1 → T	3-36	2
33(0)	2	TADS	(U,A,E)+((T)-3, (T)-2,(T)-1) → U,A,E,(T)-3 → T	3-37	22
33(3)	2	NTADS	(U,A,E)-((T)-3, (T)-2,(T)-1) → U,A,E (T)-3 → T	3-37	22

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		FIXED-POINT ARITHMETIC (CONTINUED)			
33(1)	2	TSBS	(U,A,E)-((T)-3, (T)-2,(T)-1) → U,A,E (T)-3 → T	3-37	22
33(2)	2	RTSBS	((T)-3,(T)-2,(T) -1)-(U,A,E) → U,A,E (T)-3 → T	3-38	22
31(1)	2	TMPS	(U,A,E)*((T)-3, (T)-2(T)-1 →U, A,E (T)-3 →T	3-38	22
31(3)	2	TMPFS	(U,A,E)*((T)-3, (T)-2(T)-1 → U, A,E (T)-3 → T	3-38	22
32(1)	2	TDVS	(U,A,E)/((T)-3, (T)-2,(T)-1 → UAE (T)-3 → (T)	3-39	22
32(3)	2	TDVFS	(U,A,E)/((T)-3, (T)-2,(T)-1 → UAE (T)-3 → T	3-39	22
53	3	TNEG	IF(U⊕A⊕E)=0 then ($\overline{U_0}$ → U_0	3-40	22
		FLOATING-POINT ARITHMETIC			
23(0)	6A	FAD	(U,A,E)"+"(Y,Y +1,Y+2) → U,A,E	3-40	9,10,22
23(3)	6A	NFAD	(U,A,E)"-"(Y,Y +1,Y+2) → U,A,E	3-40	9,10,22
23(1)	6A	FSB	(U,A,E)"-"(Y,Y +1,Y+2) → U,A,E	3-40	9,10,22
23(2)	6A	RFSB	(Y,Y+1,Y+2)"-" (U,A,E)→ U,A,E	3-41	9,10,22
24(0)	6A	FMP	(U,A,E)"*"(Y,Y +1,Y+2) → U,A,E	3-41	9,10,22
25(0)	6A	FDV	(U,A,E)"/"(Y,Y +1,Y+2) → U,A,E	3-41	9,10,22

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
	FLOATING-POINT ARITHMETIC (CONTINUED)				
25(2)	6A	RFDV	(Y,Y+1,Y+2)"/" (U,A,E) →U,A,E	3-41	9,10,22
30(0)	2	FADS	(U,A,E)"="((T) -3, (T)-2,(T)-1) → U,A,E;(T)-3 → T	3-41	2,9,10, 22
30(3)	2	NFADS	(U,A,E)"-"((T) -3,(T)-2,(T)-1) → UAE	3-42	9,10,22
30(1)	2	FSBS	(U,A,E)"-"((T) -3,(T)-2,(T)-1 → U,A,E;(T)-3 → (T)	3-42	2,9,10, 22
30(2)	2	RFSBS	((T)-3,(T)-2,(T) -1)"-"(U,A,E) → U,A,E;(T)-3 → T	3-42	2,9,10, 22
31(0)	2	FMPS	(U,A,E)"*"((T) -3,(T)-2,(T)-1 → U,A,E; (T) -3 → T	3-43	2,9,10, 22
32(0)	2	FDVS	(U,A,E)"/"((T) -3(T)-2,(T)-1) → U,A,E;(T)-3 → T	3-43	2,9,10, 22
32(2)	2	RFDVS	((T)-3,(T)-2,(T) -1)"/"(U,A,E) → U,A,E;(T)-3 → T	3-43	2,9,10, 22
41(0)	2	FIX	CONVERT FLOATING POINT NUMBER IN (U,A,E) TO INTEGER IN (A).	3-43	8,22
42(0)	2	FLOAT	CONVERT INTEGER IN (A) TO FLOATING POINT NUMBER IN (U,A,E).	3-44	22

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
	FLOATING-POINT ARITHMETIC (CONTINUED)				
43(0)	2	NORM	FLOATING NORMALIZE	3-44	9,22
54(0)	2	FNEG	0"-"(U,A,E) → (U,A,E) ("+" = FLOATING ADD "-" = FLOATING SUBTRACT "*" = FLOATING MULTIPLY "/" = FLOATING DIVIDE)	3-44	22
		LOGICAL			
27(0)	6A, 6G	ANX	(X)ⓧ(Y) → X	3-45	
27(1)	6A, 6G	ANU	(U)ⓧ(Y) → U	3-45	
10	10	ANUI	(U)ⓧLIT9 → U	3-45	
27(3)	6A, 6G	ANUA	(U)ⓧ(Y) → A	3-45	
20	1A,B,C,D	ANA	(A)ⓧ(Y) → A	3-45	
11	10	ANAI	(A)ⓧLIT9 → A	3-45	
22	1A,B,C,D	ORA	(A)⊕ (Y) → A	3-46	
12	10	ORAI	(A)⊕LIT9 → A	3-46	
24	1A,B,C,D	XRA	(A)⊖(Y) → A	3-46	
13	10	XRAI	(A)⊖LIT9 → A	3-46	
3	7A	RAND	(S)ⓧ(D) → D	3-46	3,4
3	4B	SETBA	$1 \rightarrow A_n$	3-47	
4	4B	CLRBA	$0 \rightarrow A_n$	3-47	
5	4B	CMPBA	$(A_n) \rightarrow A_n$	3-48	
3	5	SETBM	$1 \rightarrow Y_n$	3-48	11
4	5	CLRBM	$0 \rightarrow Y_n$	3-48	11
5	5	SMPBM	$(Y_n) \rightarrow Y_n$	3-49	11
22(1)	2	ANAS	(A)ⓧ((T)-1) → A,(T)-1 → T	3-49	2
22(2)	2	ORAS	(A)⊕((T)-1) → A,(T)-1 →T	3-50	2

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		LOGICAL (CONTINUED)			
22(3)	2	XRAS	(A)⊖((T)-1) → A, (T)-1 → T	3-50	2
22(0)	2	ANXS	(X)⊗((T)-1 → X, (T)-1 → T	3-50	2
		SHIFTS			
0	8	LLX/LRX	LOCIGAL LEFT/ RIGHT X	3-51	
1	8	ALU/ARU	ARITHMETIC LEFT/ RIGHT U	3-51	12
2	8	LLU/LRU	LOGICAL LEFT/ RIGHT U	3-51	
3	8	RLU/RRU	ROTATE LEFT/ RIGHT U	3-52	
4	8	ALA/ARA	ARITHMETIC LEFT/RIGHT A	3-52	12
5	8	LLA/LRA	LOGICAL LEFT/ RIGHT A	3-52	
6	8	RLA/RRA	ROTATE LEFT/ RIGHT A	3-52	
0	9	LLUAE/ LRUAE	LOGICAL LEFT/ RIGHT U,A,E	3-53	
1	9	ALUA/ ARUA	ARITHMETIC LEFT/RIGHT UA	3-53	12
2	9	LLUA/ LRUA	LOGICAL LEFT/ RIGHT U,A	3-53	
3	9	RLUA/ RRUA	ROTATE LEFT/ RIGHT U,A	3-54	
44(0)	2	LLO	LOCATE LEADING ONE (DIRECTION OF SHIFT IS DETERMINED BY SHIFT COUNT, AFTER INDEXING IF ANY: COUNT > 0 → LEFT SHIFT,	3-54	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		SHIFTS (CONTINUED)			
			COUNT < 0 → RIGHT SHIFT)		
		COMPARES AND TESTS			
14	10	SKXEI	SKIP IF (X) = LIT9	3-54	
15	10	SKXNI	SKIP IF (X) ≠ LIT9	3-55	
26	1A, B, C, D	SKAE	SKIP IF (A) = (Y)	3-55	
30	1A, B, C, D	SKAN	SKIP IF (A) ≠ (Y)	3-55	
16	10	SKAEI	SKIP IF (A) = LIT9	3-55	
17	10	SKANI	SKIP IF (A) ≠ LIT9	3-56	
30(SC)	6A, 6G	ACX	(X) [AC] (Y), SKIP ON CONDITION	3-56	
31(SC)	6A, 6G	ACU	(U) [AC] (Y), S. O. C.	3-56	
32(SC)	6A, 6G	ACA	(A) [AC] (Y), S. O. C.	3-57	
33(SC)	6A, 6G	ACE	(E) [AC] (Y), S. O. C.	3-57	
22(SC)	6A	FCP	U, A, E [AC] (Y, Y+1, Y+2) S. O. C.	3-57	22
36(SC)	2B	FCPS	U, A, E [AC] ((T)-3, (T)-2, (T)-1) S. O. C. T IS UNCHANGED	3-58	22
34(SC)	6A, 6G	LCX	(X) [LC] (Y), S. O. C.	3-58	
35(SC)	6A, 6G	LCU	(U) [LC] (Y), S. O. C.	3-58	
36(SC)	6A, 6G	LCA	(A) [LC] (Y), S. O. C.	3-59	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
			COMPARES AND TESTS (CONTINUED)		
37(SC)	6A, 6G	LCE	(E) [LC] (Y), S. O. C.	3-59	
40(SC)	6A, 6G	MSK	(Y) [AC] 0, SKIP ON CONDITION	3-59	
6	4B	SKZA	SKIP IF $(A_N) = 0$	3-60	
7	4B	SKOA	SKIP IF $(A_N) = 1$	3-60	
6	5	SKZM	SKIP IF $(Y_N) = 0$	3-61	
7	5	SKOM	SKIP IF $(Y_N) = 1$	3-61	
45(0)	2	SKNOF	SKIP IF (OF) = 0; 0 → OF	3-62	
46(0)	2	SKNCO	SKIP IF (CO) = 0; 0 → CO	3-62	
43(0)	6A	TSL	SKIP IF $(Y_{15}) = 1$; 0 → Y	3-62	
47(0)	2	DSK	SKIP AFTER NEXT INSTRUCTION [AC] MEANS ALGEBRAIC COMPARE [LC] MEANS LOGICAL COMPARE	3-62	
			JUMPS		
11	1A, C, D	JMP	Y → P	3-63	
13	1A, C, D	JZE	Y → P IF (A) = 0	3-63	
15	1A, C, D	JNZ	Y → P IF (A) ≠ 0	3-63	
17	1A, C, D	JPL	Y → P IF (A) ≥ 0	3-63	
21	1A, C, D	JMI	Y → P IF (A) < 0	3-64	
44(JC)	6A	XJP	Y → P IF (X) CONDITION MET	3-64	
45(JC)	6A	UJP	Y → P IF (U) CONDITION MET	3-64	
46(JC)	6A	AJP	Y → P IF (A) CONDITION MET	3-65	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		JUMPS (CONTINUED)			
47(JC)	6A	EJP	Y → P IF (E) CONDITION MET	3-65	
50(JC)	6A	TJP	Y → P IF (U,A,E) CONDITION MET	3-65	
25	1A, C, D	IJXN	Y → P IF (X) ≠ 0; (X)+1 → X	3-66	
27	1A, C, D	DJXN	Y → P IF (X) ≠ 0; (X) -1 → X	3-66	
55(0)	6A	IJMP	(Y) → P	3-66	
		SUBROUTINE AND SYSTEM LINKAGE			
23	1A, C, D	JSPX	(P)+1 → X, Y → P	3-66	
56(7)	6A	JSPM	(P)+1 → (Y), Y+1 → P	3-67	
63(7)	6A	CALL	(P)+1 → (T), (B) → (T)+1, (T)+2 → B → T, Y → P	3-67	
50(0)	2	RTRN	(B)-2 → T, ((T)+1) → B, ((T)) → P	3-67	
	3	SCALL	SYSTEM CALL If user mode: (B, T, L) → ((BASE)+0, 1, 2)) (BASE) → B, (BASE)+3 → T (LIMIT) → L If (T)+7 > L (address compare stack overflow (SXUAEPB) → ((T)+0, 1--6) (T)+7 → B, T STATUS → S SCBASE + Call # → P STATUS is old status with MODE →	3-68	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
	SUBROUTINE AND SYSTEM LINKAGE (CONTINUED)				
	3	SCALL	PMODE 0→FPOF, FPUF, MODE, CO, OF		
55(1)	6A	IJSPX	(P)+1 →X, (Y)→ P	3-69	
55(2)	6A	IJSPM	(P)+1 →((Y)), (Y)+1 → P	3-69	
55(3)	6A	ICALL	(P)+1 → (T), (B) → (T)+1 (T)+2 →B →T, (Y) →P	3-69	
64(0)	6B, 6C	LDC	0 → A_{0-7}, (Y_B) → A8-15	3-13	
64(1)	6B, 6C	STC	(A_{8-15}) → Y_B	3-13	
52(0)	2	CPRS	COMPARE STRINGS: CPRS EXPECTS THE BYTE ADDRESSES OF THE FIRST CHARACTER OF EACH STRING IN (X) AND (A), AND THE NUMBER OF CHARACTERS TO BE COMPARED IN (U). IF STRING (A) > STRING (X), NEXT INST. IF STRING (A) = STRING (X), SKIP 1. IF STRING (A) < STRING (X), SKIP 2. ON A ">" EXIT (NEXT INST. OR SKIP 2), (X) AND (A) POINT TO THE CHARACTERS THAT WERE FOUND TO BE UNEQUAL ON AN	3-13	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
	SUBROUTINE AND SYSTEM LINKAGE (CONTINUED)				
52(0)	2	CPRS	" = " EXIT (SKIP 1), (X) and (A) POINT ONE BEYOND THE LAST CHARACTERS COMPARED.	3-13	
65(0)	6A	GFC	$0 \rightarrow A_{0-7}$, $((Y)_B) \rightarrow A_{8-15}$	3-14	
65(1)	6A	GFCT	GFC W/TEST	3-14	
65(2)	6A	GCI	$0 \rightarrow A_{0-7}$, $((Y)_B) \rightarrow A_{8-15}$ $(Y)+1 \rightarrow Y$	3-15	
65(3)	6A	GCIT	GCI W/TEST	3-15	
65(4)	6A	IFC	$(A_{8-15}) \rightarrow (Y)_B$	3-16	
65(5)	6A	IFCT	IFC W/TEST	3-16	
65(6)	6A	ICI	$(A_{8-15}) \rightarrow (Y)_B$, $(Y)+1 \rightarrow Y$	3-16	
65(7)	6A	ICIT	ICI W/TEST	3-17	
		OTHER MEMORY			
74(0)	6F	LDAOM	$(Y) \rightarrow A$	3-18	
74(2)	6F	STAOM	$(A) \rightarrow Y$	3-18	
74(3)	6F	TSLOM	Skip if $(Y)_{15} = 1$, $0 \rightarrow Y$	3-18	
74(1)	6F	LDAOMF	$(Y) \rightarrow A$, Memory Status $\rightarrow U$	3-18	
		SPECIFIED MAP			
73(0)	6E	LDASM	$(Y) \rightarrow A$	3-19	
73(2)	6E	STASM	(A)- (Y)	3-19	
73(3)	6E	LDXSM	$(Y) \rightarrow X$ If $Y_0 = 1$, set X_0 to 1	3-19	
73(1)	6E	LDASMF	$(Y) \rightarrow A$, Memory Status $\rightarrow U$	3-19	
55(0)	2	MRGM	$(X)_0$ V (Previous Mode from Status Register) $\rightarrow X_0$	3-19	

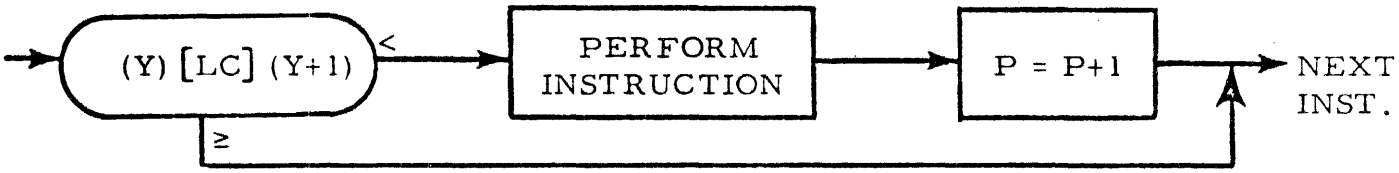
OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		SPECIFIED MAP (CONTINUED)			
56(0)	2	MSKM	$(X)_0 \Lambda$ (Previous Mode from Status Register) $\rightarrow X_0$	3-8	
		INPUT OUTPUT			
57(0)	2	LDAC	(Console Switches) $\rightarrow$ A	3-8	
60(0)	2	LDMAP	LOAD MAP (U) = Number of Words (X) = Starting core address (A) = Starting map page	3-8	
70(0)	2	LLDB	Locate leading dirty bit. Beginning with the map cell specified in X look for the next dirty bit set. If one is found, set X to the address of the map cell containing it and skip the next instruction. If none is found, do not skip	3-9	
75(0)	6A, 6G	SIM	(Y) $\rightarrow$ Interrupt Enable Mask	3-9	
75(0)	2	DOUT	(X) $\rightarrow$ I/O Address (A) $\rightarrow$ I/O Data	3-9	
74(0)	2	DIN	(X)- I/O Address I/O Data $\rightarrow$ A	3-9	
71(0)	2	IOC	I/O Control X = Channel Number, Unit Type	3-10	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		INPUT OUTPUT (CONTINUED)			
71(0)	2	IOC	A = Channel Control Cell Contents (See I/O writeup)	3-10	
72(0)	2	SIL	Set Interrupt Lockout	3-10	
73(0)	2	RIL	Release Interrupt Lockout	3-11	
		CALL AND RETURN			
2	4A	SRTRN	(B)-7 → T;((T)) S; ((T)+1 X if $INST_{12}$ = 1; ((T)+2 →U if $INST_{13}$ = 1; ((T)+3) → A if $INST_{14}$ = 1; ((T)+4; → E if $INST_{15}$ = 1; ((T)+5) →P; ((T)+6) → B Stack underflow trap if (B) > (T) If mode now = 1, ((T)-3)--- ((T)-1) →B, T, L	3-11	
64(0)	2	IRTRN	(B)-7 →T ((T)) → S ((T)+1) →X ((T)+2) →U ((T)+3) →A ((T)+4) →E ((T)+5) →P ((T)+6) →B IF (B) >(T), stack underflow trap If mode now user, then ((T)-3), ((T)-2), ((T)-1) B, T, L	3-12	

OP CODE (MOD)	FORMAT	MNEMONIC	FUNCTION	PAGE	NOTES
		CALL AND RETURN (CONTINUED)			
64(0)	2	IRTRN	Reset interrupt enabled mask for all levels up to the one returned to.	3-12	
77(0)	2	HLT	Halt Processor; Enable Panel	3-12	

NOTES

1. STACK OVERFLOW TRAP IF (T) ≥ (L).
2. STACK UNDERFLOW TRAP IF (T) < (B).
3. S :: = X|U|A|E|B|T|L| + 1
4. D :: = X|U|A|E|B|T|L
5. D :: = X|U|A|E
6. SETS OR RESETS CO, MAY SET OF.
7. SETS OF IF $(A_0) = 0$ AND (U) ≠ 0, OR IF $(A_0) = 1$ AND (U) ≠ 177777 (i.e., IF PRODUCT DOES NOT FIT INTO ONE REGISTER).
8. MAY SET OF.
9. FLOATING POINT UNDERFLOW TRAP MAY OCCUR.
10. FLOATING POINT OVERFLOW TRAP MAY OCCUR.
11. SPECIAL ADDRESSING MODES ALLOW BIT INDEXING.
12. ARITHMETIC LEFT SETS OF IF SIGN BIT CHANGES DURING SHIFT.
13. TESTS ON CHARACTER INSTRUCTIONS:



14. MAY BE INTERRUPTED AND RESTARTED.
15. Y IS A 16-BIT ADDRESS IN THE OTHER MEMORY (UNMAPPED).
16. BIT 0 OF THE FINAL ADDRESS SPECIFIES THE MAP.
    0 → SYSTEM MAP, 1 → USER MAP
    The final address is formed by ORing the "previous mode" bit with bit 0 of the effective address. The effective address includes (B) + (X) if specified.
17. THE ADDRESS IN X IS MAPPED THROUGH THE SYSTEM MAP.

18. BIT 9 OF THE MAP ADDRESS SPECIFIES THE MAP.

    0 ⇒ SYSTEM MAP, 1 ⇒ USER MAP

19. BASE, LIMIT, and SCBASE are dedicated core locations. STATUS is the S register with mode moved to previous mode, and floating point underflow trap enable, floating point overflow trap enable, mode, carryout, and overflow bits and instruction counter set to zero.

20. Memory reference performed even if parity or protect violation occurs, memory status returned for software analysis.

21. Privileged Instruction.

22. Implemented in AP only.

23. Implemented in CP only.

COMMENTS

1. The "current active interrupt level" in the status register will contain all ones when no interrupt is being processed.

2. The "software" interrupt enabled mask is set by the SIM instruction. The "firmware" interrupt enabled mask is set by the firmware when an interrupt is taken or an IRTRN is processed. On an interrupt, the "current active interrupt level", n, in the new status register contents is set to the bit number of the interrupt being accepted. Bits n to 15 in the "firmware" interrupt mask are set to zero. On an IRTRN, the "firmware" interrupt mask is set from the "software" interrupt mask with bits n to 15 set to zero; n is the "current active interrupt level" in the status word being restored.

3. User stack overflow should cause a trap.

4. System stack overflow should stop and print a message.

5. Memory violations should cause a trap. Like a system call with the "specified map" form of the address at which the violation occurred in X.