TX - 0 INSTRUCTION SET

CHARACTERISTICS OF THE TX-O

The TX-O is a one of a kind general purpose machine that was produced by Lincoln Laboratories in the late fifties as a test bed for the then new transistor logical circuitry and core memory.

Its essential features are given below: -

1. Store Size and Type:
   8192, 18 bit words, with an access time of six microseconds (magnetic ferrite cores)

2. Active Registers:

   1. 18 bit accumulator (called AC)
   2. 14 bit signed index register
   3. 18 bit in-out register (called live register)
   4. 18 bit memory buffer register

3. IO Devices

   a. An on line electric typewriter (flexowriter) which puts a 6-bit character code into the live register when a key is depressed; and types or punches 6 bit character data from the accumulator under program control. 10 character /sec rate.

   b. An on-line paper tape reader (300 character /sec) which transfers the coded contents of punched paper tape into the AC.

   c. An on-line magnetic tape unit which reads and writes on a 6 channel magnetic tape into and from the live register, under program control.

   d. A cathode ray tube display for converting the contents of the AC into the position of an intensified spot.

   e. A "Light Pen" photocell pickup for picking up the occurence of displayed points and effecting the AC.

   f. Lines for connection of special devices.

   g. Toggle switch registers (TAC, TBR, etc.)

   h. Neon display lights showing AC, MBR, IR, etc.

4. Internal Representation of Numbers and Arithmetic System: - 18 bit, ones complement.

5. Addressing: - Direct, indexed and operate class. (No indirect or immediate addressing)

6. Operation Set: - Most arithmetic and logical ops, with the exception of multiply, divide, subtract. Extremely large range of operate class commands. Mostly commonly used command: -

Indexed Version	Direct		Action
ldx	lda	-	Load AC
stx	sto	-	Store AC contents
adx	add	-	add to AC
llx	llr	-	load live register
slx	slr	-	store live register
	ldx	-	load index register
	sxa	-	store index
	aux	-	add to index
	trz	-	jump on zero AC
	trn	-	" " negative AC
	tix	-	" " index contents
	tra	-	unconditional jump
	tsx	-	jump and set index
	com	-	change AC sign
	cla	-	set AC to +0
	shr	-	shift right (divide by 2)
	cyr	-	cyle AC right
	cyl	-	circle AC left
	ana	-	logical "and" of AC + LR AC
	ora	-	logical "OR" of AC + LR AC
	hlt	-	stop computer

In addition, the principal IØ ops are listed below

dis - display AC contents interpreted as a
pen - if light pen observes a point, set AC bit 0 to 1.
tac - transfer test AC (toggle switches) to AC
prt - print a character from the AC on the flexo
ril - read a character from the PETR into the AC

Obviously, considering the number of active registers and the possible data transfers between them, there are a very large number of operate and addressable operations that are both desirable and possible. A large number of these do exist.

II. ADDRESSABLE COMMANDS

(A) STORE CLASS

MNEMONIC	OCTAL VALUE	OPERATION	SYMBOLIC DESCRIPTION AND TIMING
STO y	000000 + y	Store	$C(AC) \rightarrow C(y)$
Place the contents of AC in register y. The previous contents of y are destroyed. Contents of AC remain unchanged.			2 cycles
STX y	020000 + y	Store AC, Indexed	$C(AC) \rightarrow C(y+C(XR))$ 2 cycles
SXA y	040000 + y	Store Index in Address	$C(XR)_{5-17} \rightarrow C(y)_{5-17}$
Store the digits of the index register in the address portion of register y. The sign of XR is ignored. The contents of XR are unchanged. Bits 0 through 4 of register y are unchanged.			3 cycles
ADO y	060000 + y	Add One	$C(y) + 1 \rightarrow C(AC) \rightarrow C(y)$
Add One to the contents of memory register y and leave the results in the accumulator and register y.			3 cycles
SLR y	100000 + y	Store Live Register	$C(LR) \rightarrow C(y)$
The contents of LR are placed in register y. The previous contents of y are destroyed. Contents of LR are unchanged.			2 cycles
SLX y	120000 + y	Store LR Indexed	$C(LR) \rightarrow C(y+C(XR))$ 2 cycles
STZ y	a40000 + y	Store Zero Clear register y to +0.	$0 \rightarrow C(y)$ 2 cycles

II. ADDRESSABLE COMMANDS

(B) ADD CLASS

MNEMONIC	OCTAL VALUE	OPERATION	SYMBOLIC DESCRIPTION
ADD y	200000 + y	Add the contents of register y to AC. Contents of y are unchanged.	$C(y) + C(AC) \rightarrow C(AC)$ 2 cycles
ADX y	220000 + y	Add, indexed.	$C(y+C(XR)) + C(AC) \rightarrow C(AC)$ 2 cycles
LDX y	240000 + y	Load Index. Load the index register from bit 0 and bits 5 through 17 of register y. The contents of y are unchanged.	$C(y)_{5-17} \rightarrow C(XR)_{5-17}$ $C(y)_0 \longrightarrow C(XR)_4$ 2 cycles
AUX y	260000 + y	Augment Index. The contents of memory register y are added to XR. The fourteen bit number added consists of bit 0 and bits 5 through 17 of register y. Addition is ones' complement on 14 bit numbers.	$C(y)_{0,5-17} + C(XR) \rightarrow C(XR)$ 2 cycles
LLR y	300000 + y	Load Live Register. The contents of register y replace the previous contents of LR. Contents of y are unchanged. Previous contents of LR are destroyed.	$C(y) \rightarrow C(LR)$ 2 cycles
LLX y	320000 + y	Load LR, Indexed	$C(y+C(XR)) \rightarrow C(LR)$ 2 cycles
LDA y	340000 + y	Load Accumulator. The contents of register y replace the previous contents of the AC. Contents of y are unchanged. Previous contents of the AC are destroyed.	$C(y) \rightarrow C(AC)$ 2 cycles
LAX y	360000 + y	Load Accumulator, Indexed	$C(y+C(XR)) \rightarrow C(AC)$ 2 cycles

II. ADDRESSABLE COMMANDS

(C) TRANSFER CLASS

MNEMONIC	OCTAL VALUE	OPERATION	CONDITIONS AND SYMBOLIC DESCRIPTION	
			$y \rightarrow C(PC)$ Timing: 1 cycle	$C(PC)+1 \rightarrow C(PC)$ Timing: 2 cycles
TRN y	400000 + y	Transfer on Negative AC	$C(AC)_0 = 1$	$C(AC)_0 = 0$
If the AC bit 0 is a one, take next instruction from register y. Otherwise, take next instruction in sequence.				
TZE y	420000 + y	Transfer on Zero	$C(AC) = +0$ or $C(AC) = -0$	$C(AC) \neq +0$, and $C(AC) \neq -0$
If the contents of the accumulator are either plus zero or minus zero, the next instruction is taken from register y. If the accumulator contents are not plus or minus zero, the next instruction in sequence will be executed.				
TSX y	440000 + y	Transfer and Set Index	Always; $C(PC) \rightarrow C(XR)$ 5-17 $0 \rightarrow C(XR)_4$	Never
The next instruction is taken from register y and the address of the register following the TSX instruction is placed in the index register.				
TIX y	460000	Transfer and Index	$C(XR) \neq +0$ and $C(XR) \neq -0$ If $C(XR)_4 = 1$, $C(XR)+1 \rightarrow C(XR)$; If $C(XR)_4 = 0$, $-[-C(XR)+1] \rightarrow C(XR)$	$C(XR) = +0$ or $C(XR) = -0$
If the index register contains plus or minus zero, perform the next instruction in sequence without changing the contents of the index register. If the index register contains a non-zero positive number, its contents are reduced by one and the next instruction is taken from register y. If the index register contains a non-zero				

II. ADDRESSABLE COMMANDS

(C) TRANSFER CLASS

MNEMONIC	OCTAL VALUE	OPERATION	CONDITIONS AND SYMBOLIC DESCRIPTION $y \rightarrow C(PC)$ Timing: 1 cycle	 $C(PC)+1 \rightarrow C(PC)$ Timing: 2 cycles
negative number, its contents are increased by one and the next instruction is taken from register y. A zero result will have the same sign as the initial contents of the index register.				
TRA y	500000 + y	Transfer	Always	Never
The next instruction is taken from register y.				
TRX y	520000 + y	Transfer, indexed.	Always $y+C(XR) \rightarrow C(PC)$	Never
TLV y	540000 + y	Transfer on external Level	External level = 0 volts	External level = -3 volts
This instruction provides a means of testing an external condition, provided said condition can provide a 0 or -3 volt level at the in-out panel connection labeled TLV.				

III. OPERATE CLASS COMMANDS

(C) MICRO-ORDERS

MNEMONIC	ACTION	SYMBOLIC DESCRIPTION
CLA	CLear AC	$0 \rightarrow C(AC)$
AMB	transfer AC contents to MBR	$C(AC) \rightarrow C(MBR)$
XMB	transfer XR contents to MBR	C(XR) bits 5-17 $\rightarrow$ C(MBR) bits 5-17. C(XR) bit 4 $\rightarrow$ C(MBR) bits 0-4.
MBL	transfer MBR contents to LR	$C(MBR) \rightarrow C(LR)$
LMB	transfer LR contents to MBR. Note: LMB and MBL, if used simultaneously, interchange C(LR) and C(MBR).	$C(LR) \rightarrow C(MBR)$
MBX	transfer MBR contents to XR	$C(MBR)_{5-17} \rightarrow C(XR)_{5-17}$ $C(MBR)_0 \rightarrow C(XR)_4$
CYR	CYcle AC contents Right one binary position. (AC bit 17 goes to AC bit 0)	$C(AC)_i \rightarrow C(AC)_j$ $i = 0, 1, \ldots, 17$ $j = (i+1) \bmod 18$
SHR	SHift AC contents Right one binary position [AC bit 0 is unchanged, bit 17 is lost)	$C(AC)_i \rightarrow C(AC)_{i+1}$, $i = 0, 1, 2, \ldots 16$
ANB	ANd (logical product) LR MBR.	$C(LR) \wedge C(MBR) \rightarrow (MBR)$
ORB	OR (logical sum) LR contents into MBR.	$C(LR) \vee C(MBR) \rightarrow C(MBR)$
COM	COMplement AC	$\overline{C(AC)} \oplus C(AC)$

III. OPERATE CLASS COMMANDS

(C) MICRO-ORDERS

MNEMONIC	ACTION	SYMBOLIC DESCRIPTION
PAD	Partial ADd MBR to AC (for each MBR one, complement the corresponding AC bit.	C(MBR) C(AC) C(AC)
CRY	A CarRY digit is a ONE if in the next least significant digit, either AC = 0 and MBR = 1, or AC = 1 and carry digit = 1. The carry digits so determined are partial added to the AC by CRY. PAD and CRY used together give a full one's complement addition of C(MBR) to C(AC).	$\mathrm{CRY}[C(AC), C(MBR)] = C(AC) \oplus C \rightarrow AC.$ $C_i = [C(MBR)_j \wedge \overline{C(AC)}_j] \vee [C_j \wedge C(AC)_j]$ $i = 0, 1, \ldots, 17$ $j = (i+1) \bmod 18.$ $\mathrm{CRY}\ [C(AC) \oplus C(MBR), C(MBR)] = C(AC) + C(MBR)$

III. OPERATE CLASS COMMANDS

(D) IN-OUT GROUP COMMANDS WHICH CAN BE USED WITH MICRO-ORDERS SPECIFIED BY FITS 9-17.

OCTAL CODE	MNEMONIC	ACTION	CYCLE AND TIME PULSE
631000	CLL	CLear Left 9 bits of AC	0.6
632000	CLR	CLear Right 9 bits of AC	0.6
607000	SPF	Set Program Flag register from MBR	1.6
606000	RPF	Read Program Flag register into MBR. (inclusive or)	1.2
602000	TBR	transfer TBR contents to MBR (inclusive or)	1.1
601000	TAC	transfer TAC contents to AC (inclusive or)	1.2
603000	PEN	set AC bit 0 from light PEN FF, and AC bit 1 from light gun FF. (FF's contain one if pen or gun saw displayed point). Then clear both light pen and light gun FF's.	1.1
620000	CPY	CoPY synchronizes transmission of information between in-out equipment and computer.	**
621000	R1L	Read ONE Line of tape from PETR into AC bits 0, 3, 6, 9, 12, 15, with CYR before read (inclusive or)	IOS
623000	R3L	Read THREE Lines of tape from PETR AC bits 0, 3, 6, 9, 12, 15, with CYR before each read (inclusive or)	IOS
622000	DIS	DISplay a point on scope (AC bits 0-8 specify X coordinate, AC bits 9-17 specify Y coordinate). The coordinate (0,0) is usually at the lower left hand corner of the scope. A console switch is available to relocate (0,0) to the center.	IOS

OCTAL CODE	MNEMONIC	ACTION	CYCLE AND TIME PULSE
626000	P6H	Punch one SIX-bit line of Flexo tape (without seventh hole) from AC bit 2, 5, 8, 11, 14, 17. NOTE: Lines without seventh hole are ignored by PETR.	IOS
627000	P7H	same as P6H, but with SEVENTH hole	IOS
610000 through 617000	EXO through EX7	operate user's EXTernal equipment Causes signals to appear at corresponding terminals on the in-out panel.	IOS
600000	NOP	Perform No in-out group OPeration	
630000	HLT	HaLT the computer and sound chime	1.8
624000	PRT	PRinT one six bit flexo character from AC bits 2, 5, 8, 11, 14, 17.	IOS

TX-0 UTILITY SOFTWARE

Over the years, many different assembly programs which convert from symbolic machine language to binary code; and "debugging"programs for on-line program correction, have been written for TX-0. The currently used assembly program is called "MIDAS" and the debugging program "DOCTOR".

MIDAS uses programs as input, which are punched on paper tape by flexowriter, while DOCTOR uses the program assembled by MIDAS, and an associated "symbol table" on magnetic or punched tape. The on-line flexowriter is used as an output device by MIDAS (to note programmer errors) and as an input device by DOCTOR (to change programs and data). The symbol table (A symbol to address directory) allows the discourse to be in terms of previously assigned names, rather than binary or octal.

MIDAS:-

As has been mentioned previously, the object of a machine language assembly program is threefold: - first to recode mnemonic operation codes into their binary equivalent; secondly, to convert numeric values from decimal notation, to the appropriate binary value; and finally, to assign specific binary locations to each data and instruction operand, which can thus be referred to an arbitrary symbol in lieu of an equivalent binary address.

Since programs to be assembled by MIDAS are prepared by punching paper tape on a flexowriter, the character set to be used in MIDAS is somewhat constrained. The flexowriter character set includes upper and lower case alpha-numeric symbols; a small number of punctuation symbols " , - . / * | " etc.; and some formatting codes: color shift, space, tab and carriage return.

Let us now consider their usage in constructing normal operation and data coding:

1. Tabs and carriage returns are used interchangeably, and if a number of them are used in sequence, all but the first are ignored. They are used to delimit instructions. The assembler assigns sequential locations to instructions and data beginning at location octal 20, unless a numeric assignment is made by the programmer.

2. Strings of lower case alpha-numeric characters (delimited by punctuation or spacing and not exclusively numeric) are used to denote and refer to symbolic locations. Only the first six characters in a string are used. Some specific 3 letter strings can only be used for mnemonic op codes. If a string contains a single capital letter, it becomes a "variable", and the location assignment can be automatic.

3. MIDAS has two numeric modes: - octal and decimal. A string of numeric characters is re-encoded into binary from whichever mode was last declared (Normal mode is octal).

4. MIDAS will interpret an expression made up of only symbols, numeric constants, and the arithmetic operators +, -, and X (times) to compute a single equivalent address:

Thus:

```
add    a - b + 1
```

would be interpreted as an add command, whose address part was the address of a minus the address of b, plus 1.

5. Special characters: - The character . period; , comma; | vertical bar, and (right paren, have special significance in formatting an instruction. An instruction is generally made up of a location field; an operation field and an operand field; and a comment field, separated by the appropriate delimiters.

a) A location field can be one of three forms for a specific numeric location. It is the numeric field, followed by a vertical bar, and a tab or carriage return: -

thus: -

```
100 | (tab)
```

b) For a symbolic address, it is the symbol, followed by a comma , and a tab.

c) If no tag is needed for the operand instruction or data, the field may be omitted.

All operands referred to by name in the program must somehow have a defined location. The one exception is "variables".

An operation field may be one of the three letter TX-0 op-codes; one of the MIDAS "pseudo-operations" or a "Macro-operation", previously defined by the programmer, (more about this later) or, in the case of numeric data, it may be omitted. The operation field is terminated by either a space, or (for operate class commands with no operands) carriage return.

The operand field may be numeric data; a symbolic expression made from one or more symbolic addresses, or a "literal". The special character "." (period) if it appears in a symbolic expression has the value of the location of the instruction in which it appears.

Example:

```
a, add b
   tze . -1
```

A literal operand field is delimited on the left by a left paren "(", and is interpreted like an <u>immediate</u> address, i.e., as the <u>data</u> to be operated upon, rather than the address of the data. (This is clearly a programmer convenience as the assembler must set aside an operand location for the data and refer to it in the instruction). The right paren may be used or elided.

For example: -

```
a,   lda  (+1)
```

is broken down by the assembler into: -

```
a,   lda  u
   . . . .
u,   +1
```

A comment field, when not omitted, is delimited on the left by a vertical | . Comments may be any sequence of characters, and are terminated by a carriage return.

Comment fields, and color shifts are ignored by MIDAS.

Pseudo-Operations

In addition to the normal TX-0 operation codes, MIDAS recognizes some other pseudo-operations for internal and bookkeeping reasons. Such pseudo-ops are inserted in the program in the normal manner. The user has the option, also of defining his own pseudo-ops. A list of the most useful of these follow:

Octal: -	declares that all subsequent numbers are to be interpreted as base 8 (until a "decimal" op)
decimal: -	declares all subsequent numbers to be interpreted as base 10. (until next "octal" op)
start: -	Must be used as the last op of a program. Must specify symbolic start address, and be followed by one or more carriage returns.
constants:	Normally used at the end of program prior to start - Must be used if any literals are employed in the program. (Informs the assembler of where constants are to be stored).
variables:	Similar to constants, but for "variable", i.e., operands with one or more capital letters.
character:	Declares the operand character to be stored as its flexo code, suitable for printing out.
define: -	Must be used to delimit macro instruction definitions
terminate	
dimension:	Reserve space for operand arrays: Space to be reserved is given as a literal, and operands are separated by commas: - Example: dimension array (100), index (200)
title block:	The title block - although not strictly a pseudo-operation must be present at the start of a program tape. It is the first string of characters on the tape, up to the first carriage return. It appears on the on-line flexo during assembly, and is punched legibly on the punched paper tape result of the assembly.

An example of a TX-0 program, written in MIDAS follows: -

EXAMPLE OF PROGRAM FOR THE TX-0 WRITTEN IN THE MIDAS LANGUAGE

Integer multiplication program.

This program computes the signed product of integers a and b, stopping with the product in the accumulator. If the product exceeds 2^{17}-1 in absolute value, the program stops at register "overflow".

```
beg,    stz  Tt           |temporary register to count whether none, one or both
        lda  a            |factors are negative
        trn  .+2
        tra  .+4
        com               |if factor is negative, complement factor and index tt.
        sto  a
        ado  tt
        lda  b
        trn  .+2
        tra  .+4
        com
        sto  b
        ado  tt
        lda  tt           |if tt holds 0 or 2, product is positive, otherwise negative
        cyr
        trn  .+3
        llr  (opr
        tra  .+2
        llr  (com
        slr  setagn       |set instruction "com" or "opr" in register setagn".
        ldx  (16.         |set index to one less than number of binary digits to be
        stz  Prod         |considered; set product initially to zero
        llr  b
 loop,  lda  a            |start multiplication loop
        cyr               |cycle multiplier right, least significant bit to sign pos.
        sto  a
        trn  .+2          |is next bit a 1?
        tra  .+5
        lac               |add multiplicand to partial product
        add prod
        sto prod
```

```
          trn overflow      overflow if result negative
          lac               cycle multiplicand left

          cyl
          alr
          tix loop          finished?
          lda
 setsgn,  0                 set sign of product
          hlt
overflow,     hlt           overflow stop
a         21                arbitrary factor value
b         10
constants
variables
start
```

Macro-Instructions

Often certain instruction sequences appear several times throughout a program in almost identical form. In such cases the sequence that is being repeated may be defined as a macro-instruction and given a specific name which may be subsequently used in place of the sequence. In place of the characters of the sequence that are expected to vary between occurrences, we may define dummy arguments in the macro-instructions definition, these dummy arguments being replaced by specific arguments on each occurrence of the macro-instructions.

Example: In the sequence:

```
lda  a
add  b
sto  c
lda  d
add  e
sto  f
```

the sequence:

```
lda  x
add  y
sto  z
```

is the model for the repetition, with x, y, and z taking on the specific characters a, b, and c in the first case and d, c, and f in the second case. A macro-instruction "augment" may be defined as follows:

```
define  augment x,y,z
        lda  x
        add  y
        sto  z
        terminate
```

The pseudo-instructions define and terminate separate the macro-instruction definition from the rest of the program. define defines the first legal symbol following it as the macro name and the dummy arguments follow, as required, separated by commas and terminated by a tab or carriage return. Next follows the body of the macro definition, ending just before the pseudo-instructions terminate.

The macro-instructions may be referred to by means of a macro call, consisting of the macro name, followed by the list of arguments, if any, separated by commas, and terminated with a tab or carriage return.

Example:

```
augment a,b,c
augment d,e,f
```

When preceded somewhere in the program by the above macro-definition, the above two macro-instruction references are equivalent to the six instructions listed previously.

Macro definitions may contain other macro definitions or macro calls. For further information refer to TX-0 Memo M5001-39, Nov. 26, 1962.

C. Source Programs

A source program for MIDAS consists of one or more flexo tapes, each with a title, a body, and a start pseudo-instruction. The first string of characters, terminated by a carriage return, is interpreted as the title. The start pseudo-instruction denotes the end of the source program tape and must be preceded and followed by a carriage return. If a location is specified after the start instruction and before the carriage return, pressing RESTART on the computer console will cause the execution of the program starting in that location. An argument of the form start addr, where addr is some specified location, will cause automatic execution of the program starting at location addr as soon as the binary or object tape is read in by the machine.

D. Preparation of Source Programs in the MIDAS Assembly Language

Programs for conversion into binary or machine language are prepared on off-line Flexowriters operated independently of the computer. Typing on the Flexowriter produces a punched paper tape version of the program for entry into the computer, as well as a printed version for reference purposes. The following advice on program preparation is based on the experience of many users.

1. Leave at least six inches of blank tape before starting to punch a program. This is accomplished by depressing the Tape Feed Switch as well as leaving the Punch On Switch depressed. Mark tape title and your name on leading end with a white pencil.

2. Make sure that the "seventh hole" switch is depressed when punching a program on the Flexowriter.

3. Make corrections on your tape by turning the knob under the tape punch until the punched line to be corrected is on top of the punch pins. Press the "delete" switch to delete that row. If a sequence of characters is to be deleted, press the "delete" and "tape feed" switches simultaneously.

4. If correction is to be made on the printed copy independently of the tape version, make sure the "punch on" key is first lifted.

5. To edit a tape, reproduce the portions of the program tape requiring changes, making insertions and deletions as necessary. Insert the start of the tape to be reproduced in the reader mechanism, being careful that the tape is properly placed with respect to the tape guides. Reading and reproducing (if"punch on" switch is down) is started when the "start read" switch is depressed and continues until the "stop read" notch is depressed or a stop code is read from the tape. Stop codes are ignored by MIDAS. To reproduce large sections of tape rapidly, turn on the "reproduce, no print" switch.

6. Type programs, using one carriage return between instructions except where distinct logical divisions exist, where it is a good idea to use two or three carriage returns. Tabulate your copy so that only address tags and pseudo-instructions are typed along left-hand margin, instructions are typed starting at the first tabulation stop of the carriage and comments one or two tab stops after the end of the instruction.

7. Color shift is ignored by the computer and may be used for differentiation purposes by the programmer.

8. It is bad practice to reference addresses by the notation .$\pm$n where n exceeds roughly six. A common error is not correcting n when an instruction is inserted into or removed from the range across which the reference is made. Address tag assignments are safer.

9. For long programs leave frequent blocks of blank tape, say at the end of every typed page. Tape sections may be easily spliced if such blocks containing no information exist. Splice tape by overlapping the tape ends with feed holes aligned as they would look once they are joined, cut the ends with scissors diagnonally across the width of the tape. To join the two edges, place them side by side - not overlapping - and cover with scotch tape, then trim off the scotch tape extending beyond the width of the paper tape.

III. NOTES ON COMPUTER OPERATION, PROGRAM ASSEMBLY USING THE MIDAS ASSEMBLY PROGRAM AND PROGRAM DEBUGGING USING THE DOCTOR SYMBOLIC DEBUGGING PROGRAM

A. Program Assembly

The primary input facility to the computer is the photoelectric tape reader (PETR) Machine language programs may be read into the computer by placing their front end in the PETR (7th hole toward the operator) and pressing the READ IN button on the console.

For normal use the system tape consisting of DOCTOR and MIDAS are stored in a special section of magnetic tape and may be read into the computer using the CALL program. To start an assembly, set the toggle switch TBR on the console to trn 20 (400020) and read in the CALL tape. The computer will rewind the magnetic tape reel and stop with the load point light (beginning of tape mark) lit. This tape has two load point marks, one near the physical beginning of the tape and one closing off the area on which the system tapes are stored from programmer use. Manual momentar operation on the REVERSE key will cause the tape to rewind beyond the interior load point and pressing the RESTART button will cause the MIDAS program to be read from the magnetic tape into the computer. The machine, when halted with +0 showing as contents of the AC, is ready for assembly of programs written in the MIDAS language.

Place the off-line prepared tape in the PETR and press RESTART (not READ IN). MIDAS will commence Pass 1 or the first stage in processing the symbolic program and write intermediate information on the mag. tape. Errors, if any are printed out on the on-line flexowriter, see Page ___ for the error codes used. Pressing TEST will normally continue assembly, ignoring the found errors unless the error is of a type preventing continuation of the assembly. The machine halts when it reaches the end of the tape and pressing RESTART causes the commencement of Pass 2 of the assembly Assembly is completed when the computer halts without typing anything on the on-line Flexowriter.

The contents of the symbol table, as generated by MIDAS, may be preserved for subsequent use by the DOCTOR program by reading in the program MIDAS MAG. TAPE SYMBOL PUNCH. The same symbol table may be printed out on the on-line Flexowriter using the program MIDAS ALPHA SYMBOL PRINT.

Page missing from original document

upper case T causes the symbol table, if previously generated with MIDAS MAG. TAPE SYMBOL PUNCH, to be read in and will allow the use of the programmer's symbols in debugging the program. The computer will type the upper limit in memory for DOCTOR and the symbol table and only locations above this (lower than this value) should be used by the programmer.

A complete list of the facilities provided by DOCTOR, as well as the action of typed in characters is attached. Experience is the only reliable teacher in the use of the program.

One of DOCTOR's most useful features is the "breakpoint". When debugging a program, it is occasionally desirable to allow control to flow up to a certain instruction, reaching which the programmer would like to examine the contents of the AC, LR and relevant memory locations in his program. To facilitate this, DOCTOR will insert into the user's program a transfer instruction into itself, which will cause the contents of the AC, LR and XR to be saved and printed out. It is then possible to examine arbitrary program locations, make any necessary changes, move the breakpoint if desired, and continue the program, restoring all indicators and executing the instruction which was originally replaced by the breakpoint transfer.

The following list describes the action of typed in characters.

<u>CHARACTER</u>	<u>ACTION</u>
space	separation character meaning arithmetic plus
+	separation character meaning arithmetic plus
-	separation character meaning arithmetic minus
|	register examination character; preceded by an address, causes the addressed register to be opened, and the location sequence to be reset to this address. Immediately following a register printout, it will cause the register addressed therein to be opened. Opening a register causes the contents to be typed out as an instruction or constant, according to the current mode and makes the contents available for modifcation.
(	same as , but forces printout as octal constant for this examinatio
/	same as , but forces printout as instruction for this examination.
carriage return	if a register is open for examination and any expression has been typed immediately prior to the carriage return, the value of that expression is stored in the open register. Otherwise, no change is made.
backspace	has the same effect as carriage return, but then opens the next sequential register. This sequence is not altered by additional , (, or / characters typed after a register has been opened.
tape feed	same as backspace, but opens the previous register.
tab	same as carriage return, but opens the register addressed by the contents of the last opened register (after modification, if any). Tab alters the sequence of locations.
=	types out the last quantity as an octal integer
I	types out the last quantity as an instruction.
3	types out the last quantity as flexo code, in the order right, middle, left.
.	as a single symbol, has the value of the current location. Following a string of digits, means decimal (integer).
A	has the value of the location in which the preserved <u>accumulator</u> is stored

<u>CHARACTER</u>	<u>ACTION</u>
L	has the value of the location in which the preserved live register is stored. Register L immediately follows register A in DOCTOR.
X	has the value of the location in which the preserved lindex register is stored. Register X immediately follows register L in DOCTOR.
F	has the value of the register containing the lowest location being used by DOCTOR for symbols. Its contents will change from time to time, as symbols are defined. Register F immediately follows register X in DOCTOR.
M	has the value of the register which contains the <u>mask</u> used in searches (see below). Register M+1 contains the lower limit of all searches, and M+2 contains the upper limit. Register M immediately follows register F in DOCTOR.
Q	has the value of the last quantity typed by DOCTOR or you.
1	causes the last three characters typed in to be taken as their flexo code value. This applies only to letters or numerals.
6	print integers in octal.
o	print integers in decimal.
.	when preceded by a legal symbol, causes that symbol to be defined as the current location
=	when preceded by a legal symbol, causes that symbol to be defined as the address part of the last quantity typed by DOCTOR or you.
K	deletes all but initial symbols by setting the contents of F back to its initial value. Any redefinitions of initial symbols are not affected.
:	sets the symbol definition value to the expression typed by either DOCTOR or the operator beforehand. (See below).
)	causes the legal symbol typed immediately preceding the) to be defined as the current symbol value, as set by : or ,
S	sets the mode in which DOCTOR types out words to <u>symbolic</u>.
C	sets the mode in which DOCTOR types out words to octal <u>constants</u>
R	sets the mode in which DOCTOR types out locations to <u>relative</u> (symbolic)
O	sets the mode in which DOCTOR types out locations to <u>octal</u>.

CHARACTER	ACTION
W	causes DOCTOR to search memory between the limits specified in M+1 and M+2 for words equal to the expression preceding the W. Only bits masked 1 in register M are compared. All occurrences are typed out with their locations. Typing W alone is an error. DOCTOR will not search itself.
N	same as W, but finds all words not equal to the expression typed preceding the N
E	causes DOCTOR to search memory for all words whose address is equal to that of the expression preceding the E.
delete	deletes all typed input since last DOCTOR printout, unless the operator has typed an intervening carriage return
case shifts	inform DOCTOR of the case in which the operator is typing, are otherwise ignored.
B	conditions DOCTOR to insert a breakpoint at the location specified before the B. If no such location was specified, DOCTOR removes the previous breakpoint. A breakpoint is actually inserted only when a G, P, or U is executed (see below). DOCTOR will remove the instruction at the break location, and will save it for future restoration. The instruction at the break-location is only executed after the proceed is given.
P	after the break trap occurs, causes DOCTOR to proceed with the user's program. The proceed will cause the instruction which was at the break-location to be executed and control to return to the user's program at the point at which it was interrupted, after all registers and indicators have been restored. If the breakpoint was moved after a trap, control will still return to the instruction trapped by the last breakpoint.
U	execute the preceding expression as an instruction. The breakpoint, if any, and all registers and indicators will be set up and saved.
G	go to the location specified before the G. All indicators and registers will be restored, and the breakpoint, if any will be inserted. Typing G alone is an error.
Y	read a binary tape in standard binary block format. The tape is read into storage between the limits specified in M+1 and M+2. If a checksum error is encountered, the program will stop. It is then possible to move the tape back one block, and press Restart to continue reading, if desired.
T	read MIDAS symbol table, and merge it with the existing symbol table. Definitions on tape take precedence over definitions in storage. The new contents of register F are typed out upon completing reading the symbol section of the tape. Checksum errors are handled as in Y. A number preceding T is taken as relocation to be applied to relocatable symbols.

CHARACTER	ACTION
V	verify: reads a binary tape in binary block format and compares it against memory between locations specified by M+1 and M+2. No change is made to memory. Discrepancies are typed out as: location/ memory tape checksum errors are handled as in Y.
H	puts DOCTOR into the title punch listen loop. Characters typed in are punched out in readable format on paper tape. The terminating characters are tab, carriage return, or backspace, which do the following. tab: sets DOCTOR to punch read-in mode data blocks. car. ret.: punches a standard input routine and sets DOCTOR to punch standard checksummed data blocks. backspace: sets DOCTOR to punch standard checksummed data blocks, but punches no input routine. (a trn 17756 will be punched instead).
-	when a register is open, make the modification, if any, and punch a one word block containing that register, in format specified by H (see above).
fa:laD	punches data blocks from fa through la in format specified by H(above). fa and la are any sumbolic expressions.
J	punch a start (jump) block to the address specified to denote end of binary tape.
Z	zero all memory between register 0 and the lowest register used by DOCTOR (contents of register F).
fa:laZ	zero memory between fa and la except that part, if any, occupied by DOCTOR.

HINTS AND KINKS

Breakpoints are extremely useful for investigating misbehavior of long programs. Do not try to break at program-modified instructions, or TSX's followed by program parameters to be picked up by subroutines.

If the operator types an undefined symbol, DOCTOR will respond with a U. All typed input up to that point is deleted automatically.

If and when attempting to type out a word as flexo code, the typewriter should hang, pressing Start Read will clear it.

When trying to determine the best symbol to fit a given value, and given two equally good symbols, DOCTOR WILL pick the one last defined for its printout.

There are two ways to print a block of registers. Either set the mask to zero, set up M+1 and M+2 to enclose the area to be printed and search for any word; or, if irrelevant parts of memory happen to contain zero, merely do an N-search for zero. If you change the mask or search limits, it is well to set them back to their usual values when you are through.

<u>APPENDIX III MIDAS ERROR CODES</u>

An error listing has the following format:

Column 1:	A three letter code describing the type of error. A number following is the depth of macro calls.
2:	The octal location in the object program. The symbolic <u>r</u> means relocation.
3:	The symbolic location, in terms of the last address tag seen.
4:	The last pseudo- or macro-instruction name seen.
5:	The offending symbol, if a symbol was in error.

APPENDIX IV. SUMMARY OF DOCTOR CONTROL CHARACTERS

A	accumulator storage
B	insert breakpoint
C	print words as constants
D	punch data blocks
E	address search
F	lowest location in Doctor
G	go to
H	enter title punch (header) mode
I	equals as instruction
J	punch start block
K	kill defined symbols
L	live register storage
M	mask register
N	not word search
O	print addresses in octal
P	proceed
Q	last quantity
R	print locations in symbolic (relative)
S	print words in symbolic
T	read symbol table
U	execute as instruction
V	verify tape against memory
W	word search
X	index register storage
Y	read binary tape
Z	zero memory
0-9	numerals and symbol constituents
a-z	symbol constituents
1	take as flexo code
3	print as flexo code
6	print integers in octal
o	print integers in decimal
:	set first argument value
	examine register
/	examine register, print in symbolic
(	examine register, print in octal
)	define symbol
=	equals as octal constant
.	current location or take as decimal
,	define symbol as current location
_	define symbol as address typed
-	minus
+	plus
-	upper case minus - punch this register
tape feed	modify and open previous register
delete	delete
tab	modify and open addressed register
bk sp	modify and open next register
car ret	modify and close register
uc, lc	set case
space	plus
all other	ignored, but respond with X

The error codes are:

us- :	In general, undefined symbol. Undefined symbols are evaluated as 0. The third letter tells where it was found.
w:	In a storage word or argument of pseudo-instruction word.
m:	In a storage word generated by a macro call.
d:	In the size of a dimension array.
p:	In a parameter assignment.
c:	In a constant.
s:	In the argument of start.
e:	In the argument of entry.
r:	In the count of a repeat.
t:	In an address tag of more than one syllable. This will frequently be the result of an undefined macro instruction
i:	In an argument of 0if or 1if.
ich	Illegal character. The bad character is ignored.
ilf	Illegal format. Some character or characters were used in an improper manner. Characters are ignored to next tab or carriage return.
ile	Illegal entry. Argument of entry is improper and will be ignored.
ilx	Illegal exit. Argument of exit is improper and will be ignored.
ir- :	Illegal relocation. The relocation is taken as). The third letter identifies where it was found, and will be the same as listed under undefined symbols (above).
mnd:	Macro name disagrees. The argument of terminate disagrees with the name of the macro being defined. First name is used.
mdt:	Multiply defined tag. Original definition retained.
mdx:	Multiply defined exit. An argument of exit is previously defined with a conflicting value. Original definition retained.
mdv:	Multiply defined variable. A symbol containing an upper case letter is previously defined as other than a variable. Original definition retained.

mdd:	Multiply defined dimension. An array name in a dimension statement has a conflicting definition. Original definition retained.
ipa:	Improper parameter assignment. The expression to the left of an equal sign is improper. The assignment is ignored.
ace:	Storage capacity exceeded. Assembly cannot continue.
tmc:	Too many constants: the pseudo-instruction constants used more than 10. times in one program.
tmp:	Too many parameters: the storage reserved for macro instruction arguments has been exceeded.
tme:	Too many entries. Maximum number of arguments of an entry pseudo-instruction is 37 octal.
tmv:	Too many variables. The pseudo-instruction variables has been used more than 8 times in one program. Assembly cannot continue.
cld:	Constants location disagrees. The pseudo-instruction constants has appeared on Pass 2 in a different location from that found on Pass 1, meaning all the constants syllables have been assigned the wrong value. Assembly cannot continue.
vld:	Variables location disagrees. The pseudo-instruction variables has appeared on Pass 2 in a different location from that found on Pass 1. The condition is ignored.
iae:	Internal assembler error. MIDAS has found that it has made a mistake in assembling the program. Deliver the error message and a copy and listing of the source program to a member of the TX-0 staff so that the trouble may be found. Assembly cannot continue. The octal location given is the location in MIDAS where the error was found.

OPERATE CLASS INSTRUCTIONS RECOGNIZED BY MIDAS

MNEMONIC	OCTAL VALUE	OPERATION
opr	600000	No operation.
xro	600001	Clear XR to +0.
lro	600200	Clear LR to +0.
cla	700000	Clear entire AC to +0.
com	600040	Complement the AC.
clc	700040	Clear and complement: set AC to -0.
shr	600400	Shift accumulator right one place, bit 0 remains unchanged.
cyr	600600	Cycle AC right one place.
cll	631000	Clear left half of AC to zero.
clr	632000	Clear right half of AC.
cyl	640030	Cycle AC left one place.
amz	640040	Add minus zero to AC.
cal	700200	Clear AC and LR to +0.
alr	640200	Place accumulator contents in live register.
alo	640220	ALR, then set AC to +0.
alc	640260	ALR, then set AC to -0.
all	640230	ALR, then cycle left once.
lac	700022	Place LR in AC.
lad	600032	Add LR to AC.
laz	700072	Add LR to minus zero in AC.
lpd	600022	Logical <u>exclusive or</u> of AC is placed in AC (partial add)
cry	600012	Carry the contents of AC according to bits of LR. Results of this operation is same as if contents of LR were added to <u>exclusive</u> or of AC and LR. PAD followed by CRY is equivalent to LAD.
lcc	700062	Place complement of LR in AC
lcd	600072	Contents of LR minus those of AC are placed in AC.
lal	700012	Place LR in AC cycled left once.
lar	700622	Place LR in AC cycled right once.
ana	740027	Logical <u>and</u> of AC and LR is placed in AC
anl	640207	Logical <u>and</u> of AC and LR is placed in LR
anc	740207	ANL, then clear AC.

MNEMONIC	OCTAL VALUE	OPERATION
ora	740025	Logical <u>or</u> of AC and LR is placed in AC.
orl	640205	Logical or of AC and LR is placed in LR.
oro	740205	ORL, then clear AC.
ial	740222	Interchange AC and LR.
iad	640232	Interchange and add: AC contents are placed in the LR and the previous contents of the LR are added to AC.
cax	700001	Clear AC and XR to +0.
axr	640001	Place AC contents in XR.
axo	640021	AXR, then set AC to +0.
axc	640061	AXR, then set AC to -0.
alx	640031	AXR, then cycle AC left once.
arx	640601	AXR, then cycle AC right once.
xac	700120	Place index register in accumulator.
xad	600130	Add index register to accumulator.
xcc	700160	Place complement of XR in accumulator.
xcd	600170	Contents of XR minus those of AC are placed in AC.
xal	700110	XAC, then cycle AC left once.
lxr	600003	Place LR in XR.
xlr	600300	Place XR in LR.
ixl	600303	Interchange XR and LR.
rax	640203	Place LR in XR, then place AC in LR.
rxa	700322	Place LR in AC, then place XR in LR.
hlt	630000	Stops computer.
cpf	607000	The program flag register is cleared.
spf	647000	Place AC in program flag register.
rpf	706020	The program flag register is placed in AC.
tac	701000	Contents of test accumulator are placed in AC.
tbr	702020	Contents of test buffer register are placed in AC.
dis	622000	Display point on CRT corresponding to contents of AC.
dso	662020	DIS, then clear AC.
pen	603000	Contents of light pen and light cannon flip-flops replace contents of AC bits 0 and 1. The flip-flops are cleared.
typ	625000	Read one character from on-line flexowriter into LR bits 12 through 17.

MNEMONIC	OCTAL VALUE	OPERATION
prt	624000	Print one on-line flexo character from bits 2, 5, etc. of AC.
pnt	624600	PRT, then cycle AC right once to set up another character.
pno	664020	PRT, then clear AC.
pnc	664060	PRT, then clear AC to -0.
p6h	626600	Punch one line of paper tape; 6 holes from bits 2, 5, etc. of AC then cycle right once.
p6o	666020	p6h then clear AC.
p6s	726000	Clear AC and punch a line of blank tape.
p6b	766020	Punch a line of blank tape but save AC.
p7h	627600	Same as p6h, but punch 7th hole.
p7o	667020	p7h then clear AC.
rlc	721000	Read one line paper tape into AC bits 0, 3, etc.
rlr	721600	rlc, then cycle AC right once.
r3c	723000	Read three line of paper tape.
rew	604010	Rewind tape unit.
wrs	604014	Select tape unit for writing a record.
rds	604004	Select tape unit for reading a record.
bsr	604000	Backspace tape unit by one record.
rtb	604004	Read tape binary. (odd parity)
wtb	604014	Write tape binary.(odd parity)
rtd	604024	Read tape decimal. (even parity)
wtd	604034	Write tape decimal. (even parity)
cpy	620000	Transmits information between the live register and selected input-output unit.

1.7 MACHINE ORGANIZATION

Let us now consider the general organization of a general purpose machine so that we can draw some conclusions with respect to the hardware aspects of constructing it. Figure 1.71 is a schematic diagram of a machine.

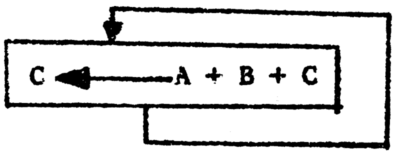
The store in this diagram must hold a large number of fixed length operand words. Typically, 2^{15} words of 2^5 bits each, (i.e., on the order of 10^6 bits). It holds the most numerous single component, and is one of the most critical subsystems with respect to speed, thus the cost per bit, power level per bit, and access time per bit are major factors in the overall system.

These factors, as we shall see, force us to organize the storage subsystem into a monolithic passive file from which the contents of but a single location (specified by the contents of a single memory address register) are gated into a single "memory buffer register". Following each such access, the store must be allowed a recovery time before it can again be accessed.

The "registers" of this system must be capable of holding fixed-length operand words; changing their state rapidly; and of driving logical "gates" directly These characteristics again constrain the realization of registers, as does the cost per bit of such "active" storage.

Before proceeding further, let us consider, in general terms, how we wish our system to operate. Specifically, let us consider the detailed execution of a program sequence in the machine of Figure 1.71.

```
U   LØAD    A
    ADD     B
    STORE   C
    JUMP    U + 1
A,   + 4
B,   + 1
C,   + 0
```



with U = 1, and thus A, B, and C are at locations 5, 6, and 7 respectively.

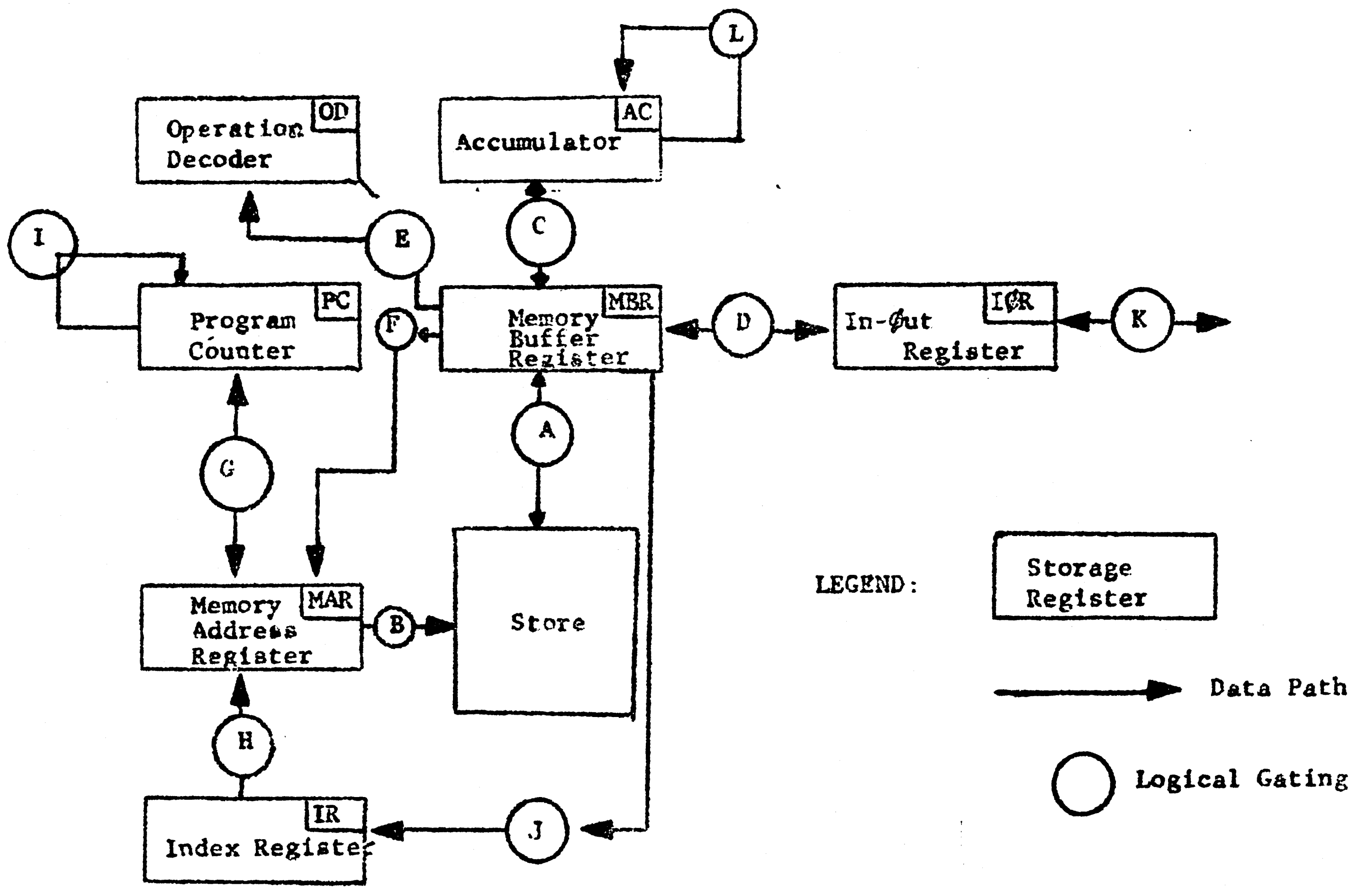
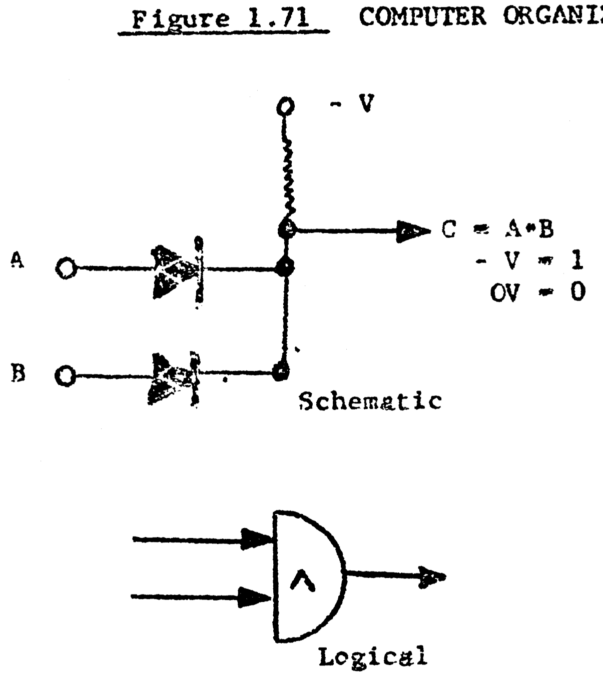


Figure 1.71 COMPUTER ORGANIZATION



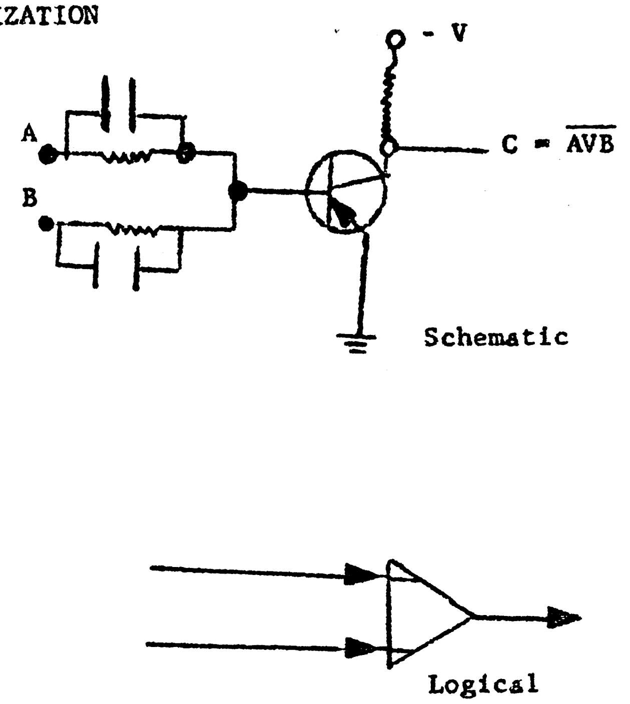
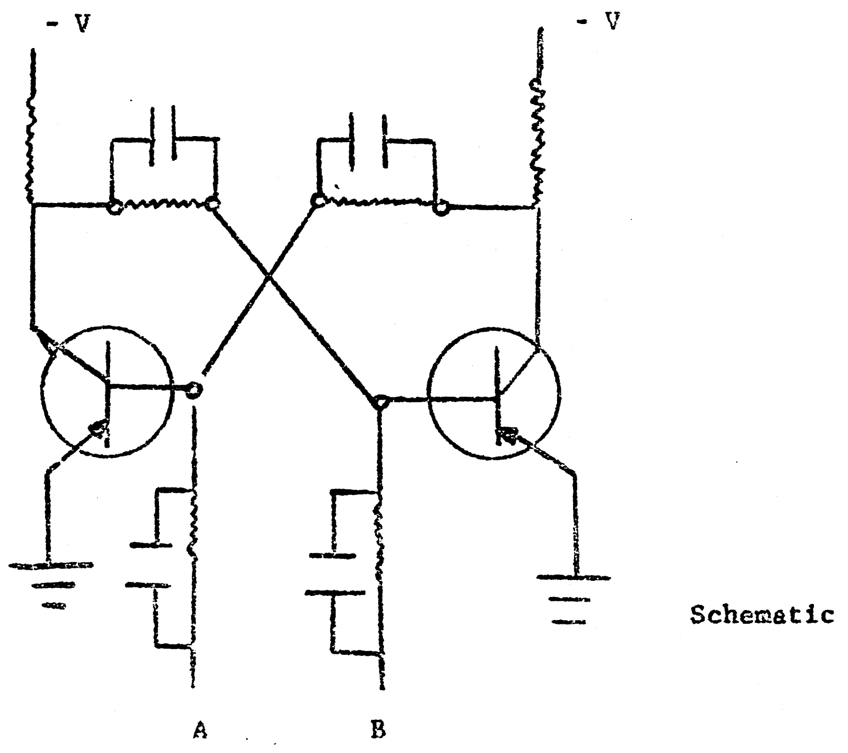
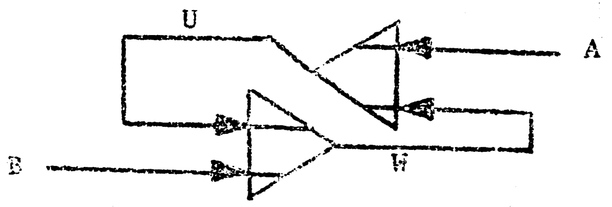


Figure 1.72a DIODE "AND" GATE

Figure 1.72b TRANSISTOR "NOR" GATE (NOT - OR)

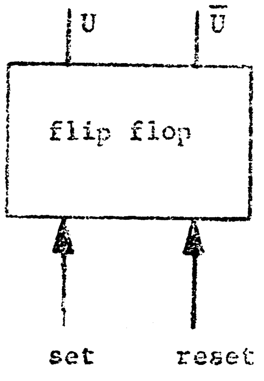
Figure 1.72c
FLIP-FLOP STORE





$U = \overline{A + W}$

$V = \overline{B + U}$



Solutions: - W = 1 U = 0

W = 0 U = 1

To be inserted in Section 1.7.

Figure 1.73

substep: -		ACTION	COMMENT
	1.	"1" $\longrightarrow$ C(PC)	Program is set to start at 1.
	2.	C(PC) $\longrightarrow$ C(MAR)	Memory is set to retrieve C(1)
(Memory Access) op	3.	$\underset{\text{MAR}}{\text{C(STORE)}}$ $\longrightarrow$ C(MBR)	Contents of Location 1 goes into MBR.
	4.	$\underset{\text{op}}{\text{C(MBR)}}$ $\longrightarrow$ C(OD)	op code bits (LOAD) go into Decoder.
		$\underset{\text{Address}}{\text{C(MBR)}}$ $\longrightarrow$ C(MAR)	Address bits go into MAR (location 5)
	6.	C(PC) + 1 $\longrightarrow$ C(PC)	Program counter incremented to +2.
(Memory Access) Data	7.	$\underset{\text{MAR}}{\text{C(STORE)}}$ $\longrightarrow$ (MBR)	Contents of Location 5 go into MBR
	8.	C(MBR) $\longrightarrow$ C(AC)	Plus 4 goes into the AC from MBR
	9.	C(PC) $\longrightarrow$ C(MAR)	2 $\longrightarrow$ MAC
(Memory Access) op	10.	$\underset{\text{MAR}}{\text{C(STORE)}}$ $\longrightarrow$ C(MBR)	Next instruction is fetched.

	11.	C(MBR) $\longrightarrow$ C(OD)
		C(MBR) $\longrightarrow$ C(MAR)
			"ADD" op to decoder
			6 to MAR

	12.	C(PC) + 1 $\longrightarrow$ C(PC)	3 $\longrightarrow$ PC
(Memory Access) Data	13.	$\underset{\text{MAR}}{\text{C(STORE)}}$ $\longrightarrow$ C(MBR)	+1 $\longrightarrow$ MBR
	14.	C(MBR) + C(AC) $\longrightarrow$ C(AC)	Add is executed and +5 is result (1st time)
	16.	C(PC) $\longrightarrow$ C(MAR)	3 $\longrightarrow$ MAR
(Memory Access) op	17.	$\underset{\text{MAR}}{\text{C(STORE)}}$ $\longrightarrow$ C(MBR)	C(3) $\longrightarrow$ MBR

	18.	C(MBR) $\longrightarrow$ C(OD)
		C(MBR) $\longrightarrow$ C(MAR)
			"Store" op to decoder
			> to MAR

	19.	C(PC) + 1 $\longrightarrow$ C(PC)	4 $\longrightarrow$ PC
(Memory Access) data	20.	C(AC) $\longrightarrow$ C(STORE)	+5 $\longrightarrow$ C(7)
	21.	C(PC) $\longrightarrow$ C(MAR)	4 $\longrightarrow$ PC
(Memory Access) op	22.	$\underset{\text{MAR}}{\text{C(STORE)}}$ $\longrightarrow$ C(MBR)	

	23	C(MBR) $\longrightarrow$ C(OD)
		C(MBR) $\longrightarrow$ C(MAR)
			"JMP" op to decode.
			2 $\longrightarrow$ MAR

* Memory Access (op) After 24, repeat from step 10.	24.	C(MAR) $\longrightarrow$ C(PC)	2 $\longrightarrow$ PC

Although not exercised in this somewhat useless program, the index register (If called for by an operation code) would be used prior to memory accesses for data (such as between step 12 and 13 for an indexed ADD).

Its action would be: -

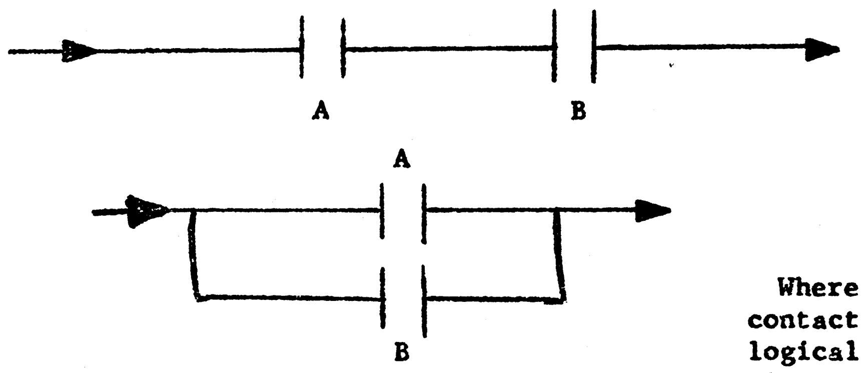
$$C(IR) + C(MAR) \longrightarrow C(MAR)$$

From the brief sequential description, we are in a much better position to consider the form of the logical gating connections in figure 1.71. Most of these connections involve the transfer of data, unchanged, from one register to another, others involve adding before transfer, and some (such as to Store and I∅) are highly dependant upon the actual hardware involved.

Let us now return to the logical calculus in order to use it as a design tool for these functions.

The link between the logical calculus, and the design of switching, or logical circuits was first pointed out by Claude Shannon, in a famous Master's thesis. Shannon noted that in a contact network, there are only <u>two</u> states of interest at each switch, or for the network as a whole: - continuity or non-continuity, (short or open circuit). If the logical value "1" were used for continuity, then a series connection behaved as a logical "AND"; and a parallel connection as a logical "OR".

Thus:



$f = A \cdot B$

$f = A+B$

Where A and B are contact networks having logical value "1" if closed.

In high speed electronic switching networks, diodes and transistors have largely replaced relay contacts, however, the same tools of analysis and synthesis are applicable.

For an electronic switching network, the two states of interest may be two distinct voltage levels. For the circuits of figure 1.72 such a choice might be -V for a logical "one" and zero for a logical zero. The "AND" circuit of 1.72a can best be understood in terms of the properties of a diode, which looks like a short circuit for current flow in the direction of the arrow symbol, and an open circuit for current flow in the reverse direction. Thus, if a number of diodes with the indicated polarity are connected to a minus voltage through a load resistor, and <u>any</u> of the diodes are connected to an input voltage of zero, then the output voltage will also be zero. If <u>all</u> the diodes are connected to logical ones (-V inputs) no current will flow through any of the diodes, and the output will be -V, i.e., a logical one. With inverted diode and source voltage polarities, 1.72a becomes an "OR" circuit.

Unfortunately, diode logical circuits have no inherent gain, and ~~thus~~ logical circuits capable of supplying power are needed.

One such circuit, which appears to be extremely versatile, is the transistor "N gate of figure 1.72b. If the base connection (horizontal lead in the symbol) is hel to zero volts, no current flows through the transistor to the supply, and thus the output voltage is -V. If negative current flows from the base; it is amplified many fold, and the output voltage drops to zero. Since each input has the same effect, the output of the NØR gate is the "NOT" of the "OR" combination of inputs; thus, a single input NØR gate is a logical inverter.

Two "NOR" gates may be cross-connected to form a "flip-flop", the most common form of active register storage.

From the definition of a NØR gate, it can be seen that if the output of one gate (in the flip-flop connection) is one, the other must be zero. If a short "one" pulse is fed to both inputs, the flip flop will "toggle" to the opposite state. Similarly, if a one is fed to one input and a zero to the other, the flip fl will assume the corresponding state. If both inputs are zero, it will hold the state it is in.

The schematic box of 1.72c will be used to represent a flip-flop. It should be noted that NØR gates are not perfect elements. They have a non-trivial cost $(< \$10^1)$; finite time to switch, and thus an upper bound on switching rate, (Mainly due to rise storage, and fall time of the transistor) on the order of 10^7 - 10^8 /sec, and since power is drawn, they can only drive a small number (called fan-out) of similar elements (about 10).

Section 1.31 INSTRUCTION OPERANDS

An instruction operand for our general machine must clearly specify two pieces of information: -

a) What is to be done - the operation (such as add, store, jump, etc.)

b) Where to find the necessary data.

With respect to a), we merely must decide on a unique binary code for each different operation that is to be performed. Usually the number of different operations that we wish to perform is small compared to the possible combinations that can be coded into N op-code bits. For hardware reasons, however 2^N may be $\gg$ number of ops. The op-code bits are usually coded at the far left (most significant bits) of instruction operands.

With respect to b), our choices are usually much wider, depending upon our mode of addressing. These modes include: -

1. No address (operate class)
2. Immediate Address
3. Direct Address
4. Indexed Address
5. Indirect Address

1. No Address: In this case, the operation to be performed involves only the contents of active registers (for example - change the sign of the accumulator, shift the accumulator right one place, etc.), and thus, the entire operand word can be treated as an operation code.

2. Immediate Address: This relatively rare form of addressing uses the contents of the rest of the operand word itself, usually coded as a positive binary number, as the data to be operated upon and, thus, can obviously save some memory accesses. The range of such data operands is clearly less than in a normal data operand. For example, if our word length is M bits, and the op code uses N bits, we can only accomodate data operands between zero and $2^{(M-N)}$.

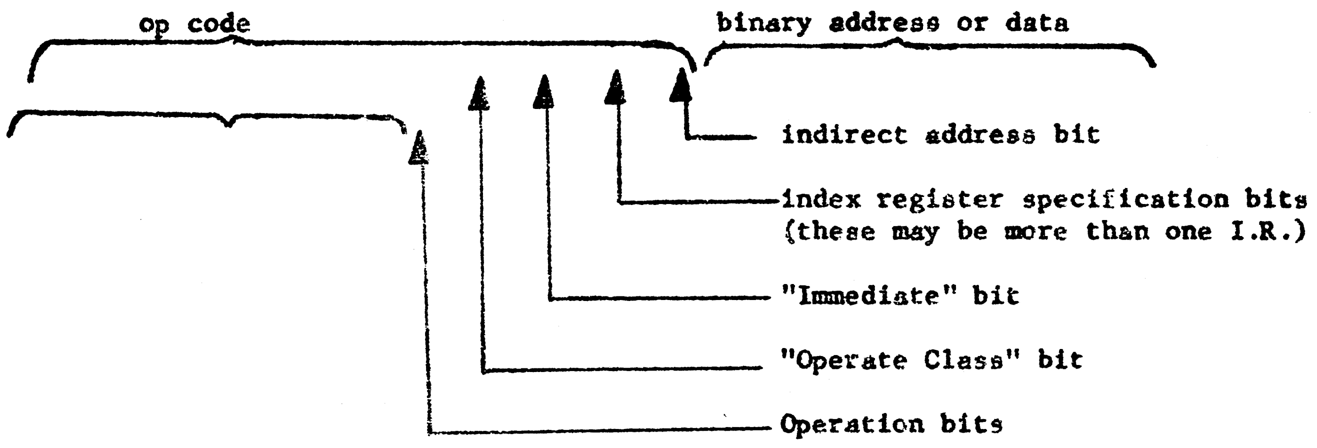
3. Direct Address: This is the predominant mode of addressing, and in this case, the M-N address bits are interpreted as a binary number, giving the memory address at which the operand can be found. Thus, as a directly addressed instruction, "add A" is interpreted as: add to the accumulator the data operand which is located in store address "A". (Note that "Add" would have a unique code, and "A" represents the binary decoding of the address bits.)

4. Indexed Address: In this mode of addressing, the contents of an "index register" are first added (or subtracted) from the address bits of instruction operand before the instruction is executed. In this way, it is possible to modify the address of an instruction (for vector or array operations, for example) without having to change the data in the instruction itself.

Thus, "add indexed A" would be executed as "Add the contents of (A + contents of index)".

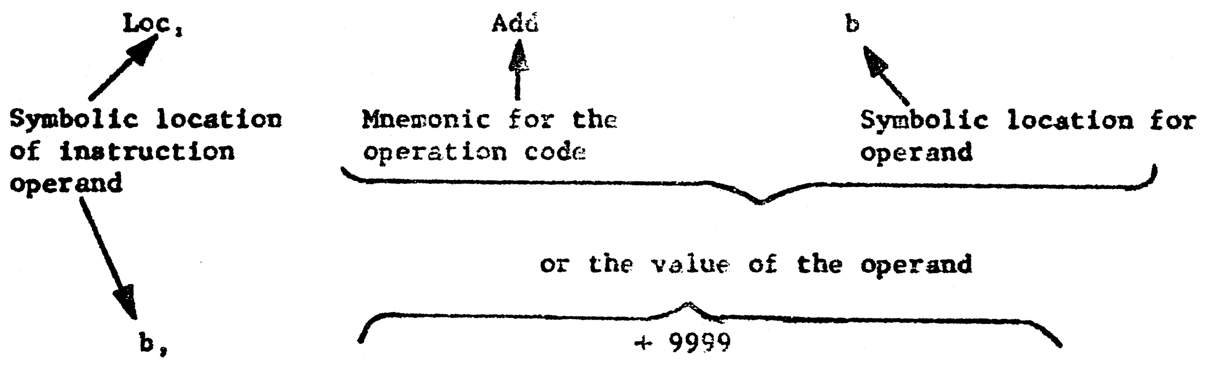
5. Indirect Address: In this addressing mode, the address bits are interpreted as the address of the address of the data. Thus, "add indirect A" would be executed as: add the contents of (the address specified in the address part of A). This would obviously require an extra memory access to acquire A before treating it as an operand address.

One common form of coding such instruction operands is to set aside specific bits for each mode of addressing inside the op code itself



Section 1.32 MACHINE CODES

When describing program steps directly, we will usually use symbols for operations, and addresses; as well as decimal numbers, as pppoaed to the equivalent binary code. Such a procedure has the benefit of eliminating some strictly clerical (but error prone) transliterations that can be better done by machine. The usual format for such a so-called machine language program step is:


Loc,
Add
b
Symbolic location of instruction operand
Mnemonic for the operation code
Symbolic location for operand
or the value of the operand
b,
+ 9999

Machine language programs are usually written, one line per sequential step; transcribed onto some form of machine-readable document (such as punched cards, or punched tape); and then translated into the actual binary coded operations and addresses, as well as binary data by "assembly programs".

The operation repetoir and hardware configuration of the object machines and the punctuation and format conventions of associated machine languages are sufficiently diverse so that it is more profitable to consider each such "language" as a special case when specific machines are considered.