G. Bell

CONFIDENTIAL

PDP-X Technical Memorandum # 34

TITLE: PDP-X Model I Extended Op Simulator Initial Specifications

AUTHOR (S): C. McComas

INDEX KEYS: Software Specification
EOP SIM

DISTRIBUTION KEY: B, C

OBSOLETE: None

REVISION: None

DATE: November 17, 1967

PROGRAM SPECIFICATION

PDP-X MODEL I EXTENDED OP SIMULATOR

C. McComas
11-7-67

0.1 Overall Description

The purpose of this program is to allow PDP-X Model I users to program with extended op (EOP) instructions. The program will reside in core during user program run time. The PDP-X Model I is designed so that if an EOP instruction is encountered, the machine will execute a trap in which the instruction itself, its effective address (EFA) and the current state of the program counter (PC) are saved in general registers R1Ø, R11,12(8). It will then branch to the address specified by the contents of general register R13(8). This address should be the starting address of the Extended Op Simulator. When control is thus transferred to the Extended Op Simulator, it will execute the trapped EOP instruction, giving results and setting up condition code indicators (CCØ,1,2) exactly the same as if the instruction were executed by Model II hardware. Control will then return to the user program in such a way that it will appear that the instruction was executed by hardware rather than software. In other words, use of the Extended Op Simulator should, except for the desired results of the EOP instruction, be invisible to the user program. No AC's or condition codes (CC's), except those indicated by the particular EOP instruction, will be modified in any way. The Extended Op Simulator is completely re-entrant so that its use may be interrupted by an EOP trap at a higher priority level.

The complete list of Model II EOP instructions, all of which are simulated, is provided below:

Arithmetic Group:

SUB	Subtract the effective word (EFW) from contents of the specified AC (R).
CMP	Algebraically compare EFW and c (R).
LCMP	Logically compare EFW and c (R), i.e., compare them as 16 bit positive integers.
MUL	Algebraically multiply EFW by c (R), giving a single or double precision result.
LMUL	Logically multiply as above.
DIV	Algebraically divide EFW into the double or single precision c (R), giving quotient and remainder.
LDIV	Logically divide as above.

Test Group:

TSTN	Test selected bits.
TSTZ	Test and zero selected bits
TSTO	Test selected bits and set them to 1. This is an effective inclusive OR.
TSTC	Test and complement selected bits. This is an effective exclusive OR.

Push Group:

PUC	Increment push down pointer and counter but leave push down list unchanged.
PUSH	Store EFW on push down list.
PUB	Push and branch.
PUL	Push, branch and link.
POC	Reverse of PUC.
POP	Store last word on push down list in EFA.
POB	Return from PUB.
POL	Return from PUL.

Miscellaneous:

LDC	Load a character from the indicated 8 bit byte.
STC	Store right half of AC in indicated 8 bit byte.
SHFT	Shift AC left or right a specified number of bits, using one of 4 techniques: (1) arithmetic shift, (2) logical shift, (3) rotate without CCØ, (4) rotate with CCØ.

1. General Specification

1.1 Machine Requirements

The program is intended for use on the basic Model I PDP-X. No special peripheral equipment is needed.

1.2 Machine Options

Not Applicable .

1.3 System Requirements

None required.

1.4 Resident Programs

The Extended Op Simulator does not run as an independent program. It operates only when an EOP instruction is encountered in another PDP-X Model I program. Therefore, besides the Extended Op Simulator, there must be some other program, say a user program, simultaneously resident in core and running.

2. Design Specifications

2.1 Design Goals

Since the Extended Op Simulator must reside in core with one or more user programs, it should occupy a minimum amount of core space. It is hoped that 1ØØØ (octal) registers will be sufficient.

Because the Extended Op Simulator may be needed by real-time application programs for such operations as multiply, divide, or shift, and because it is intended to simulate single instructions in such programs, it is of prime importance that the action of the simulator be as fast as possible.

Finally, the use of the Extended Op Simulator must be capable of interruption at any point in case it is called from a higher priority level. All routines in it must be completely reentrant.

2.2 Input

No physical input is required or possible. The program does however, expect internal software input in the form of a trapped EOP instruction. The EOP trap information is stored by the PDP-X Model I in general registers R1Ø-13(8) as follows:

R1Ø Updated PC from user program.
R11 EOP instruction word.
R12 EOP effective address.
R13 Entry point to EOP Simulator.

2.3 Output

There is no physical output from the Extended Op Simulator. The only output in any form from it is the expected result of the trapped EOP instruction stored in the AC or address indicated by the instruction, and the correct setting of the condition code bits in the Program Status Word.

2.4 Organization

2.4.1 Operational Organization

Not Applicable.

2.4.2 Internal Organization

The general structure of the Extended Op Simulator is outlined in Chart 1. Step 1 of that chart is executed by the routine titled SAV, steps 2-4 by the routine titled EOP, steps 6-1Ø by the routine titled RET. These routines are flow-charted in Charts 2-4.

SAV: The first action which the Extended Op Simulator must take is to save, in a protected place, the current state of the processor. This means all the current general registers RØ-7, except R1, must be saved. R1 need not be saved because it contains the PC, which was saved in R13 when the EOP trap occurred and replaced by the entry address to the Extended Op Simulator. A duplicate copy of the RG bits (bits 1Ø-12 of the PSW) is saved so that any change made to them while the user's PSW is on the stack can be ignored. The others must be saved because the simulator uses RØ and R2-R7. These AC's must be saved in such a way that if use of the EOP Simulator at this priority level is interrupted by a higher priority level, the AC's at this level will not be overwritten by storage of the new level AC's.

Protected storage of the AC's is accomplished by using separate stacks for separate priority levels. These stacks also will contain space for certain variable storage needed by the various Simulator routines. These stacks will be approximately 1Ø words each (dependent upon amount of storage needed by subroutines) in length, and there must be one stack for each priority level available on the user machine.

The actual stack selection and protection must be carried out in a single instruction, for otherwise an interruption will cause unpredictable results. In addition, it must be such that when use of the stack is completed, the selector-protector can be reset in a single instruction.

An AC stack will look like this:

AØ	PSW (from RØ)
A1	Effective AC (AØ + R)
A2	R2
A3	R3
A4	R4
A5	R5
A6	R6
A7	R7
A1Ø	Variable Storage
A11	Variable Storage

The implementation of the stack selection must be such that any number of priority levels from 2 to 8 may be allowed for . Extra space should not be wasted for those who have only two priority levels. Expansion for more priority levels should involve no major change.

The SAV routine also clears the current (not user) PSW bits Ø & 1 (the trap enable bits) to prevent an error trap from occurring until the TRAP routine is entered at the end of the action by the extended OP Simulator.

EOP: This routine will determine the AC for the instruction, if any is required, and save a pointer to it on the stack. It will then determine which particular EOP instruction has been trapped and dispatch to the proper subroutine to execute the instruction. Dispatching will be done in a straightforward way using a simple dispatch table and index register R2. The EOP routine will also set the correct error bits in case there is a memory reference error on an attempt to reference R1 (the PC).

RET: This routine will restore all AC's which have been saved on the stack, including any modifications in RØ (the Program Status Word) except that the RG bits will be restored as they were originally, and one or two possible AC's. It must also restore the stack selector in the "SAV" routine to what it was before this trap occurred. This must be done in such a way that an interruption in any part of this routine will cause no trouble. Finally, there must be a branch back to the user program which caused the EOP trap. Return will be to the next step in the user program

after the EOP which trapped. It is accomplished by using the trapped PC in R1Ø.

Before returning to the user program, this routine will check to see if an error trap should occur. (This is done by examining the actual PSW error bits and the user PSW error enable bits). If a trap condition exists, RET branches to the user error trap handler address (contained in R17), instead of to the normal return in the user's program.

Charts 5-24 are flow charts for the routines which execute the 22 individual EOP instructions which could be trapped. Additional explanations of these routines follow.

Charts 27-37 are flow charts for the special subroutines used by the EOP instruction routines. Additional explanations of these subroutines can also be found below.

SUB: The result of C(R) + (-EFW) must be stored in the AC specified by the SUB instruction. When the operation C(R) + (-EFW) is executed, the condition codes (CC) will be set in the current PSW as they should be set for the SUB instruction. Specifically, CCØ = 1 if there was carry out of bit Ø, * CC1 = 1 if the result was negative, * and CC 2 = 1 if the result was not zero. * These 3 CC bits must be loaded into the CC bits of the user program PSW now resident in the stack.

This routine also OR's the arithmetic error bit into the user PSW in case the result is such there should be an error trap.

CMP: To determine whether or not C (R) is algebraically greater than or equal or less than the EFW, the operation C (R) - EFW suffices. Condition code bit 1 will be set if the result is negative, indicating that C (R) is less than EFW, and CC2 will be set if the result is non-zero, indicating that C (R) $\neq$ EFW. CCØ must be ignored. Note that the actual difference is not saved.

LCMP: Here C (R) and EFW are treated as 16 bit positive integers. If C $(R)_Ø$ and $EFW_Ø$ are alike, C (R) - EFW will set the CC's as required.
Example 1: C (R) = ØØØØ44, EFW = ØØØØ54
C (R) - EFW = 1Ø meaning C (R)<EFW

Example 2: C (R) = 177776, EFW = 177774
C (R) - EFW = (-2) - (-4) = + 2 meaning C (R)> EFW

Example 3: C (R) = 177772, EFW = 177774
C (R) - EFW = (-6) - (-4) = -2 meaning C (R)<EFW

* (Cleared otherwise) .

If C (R)$_\emptyset$ and EFW$_\emptyset$ are unlike, then $\emptyset$ + EFW will set the CC bits as they should be, because $\emptyset$ + EFW$\geq$0 means EFW$_\emptyset$ = $\emptyset$ (while C (R)$_\emptyset$ = 1) so that C (R)$>$EFW, and $\emptyset$ + EFW$<$0 means EFW$_\emptyset$ = 1 (while C (R)$_\emptyset$ = $\emptyset$) so that C (R)$<$EFW. Note that this test will not clear CC2 if EFW = $\emptyset$, so if C (R)$_\emptyset$ = EFW$_\emptyset$ and EFW = $\emptyset$, the bits must be set for C (R)$>$EFW.

LMUL: This routine calls the 16 bit logical multiply subroutine to do most of its work. The CC bits will not be affected by a multiply instruction. If the addressed AC (R) is even, the 16 bit multiplier is assumed to be in R + 1, and the product will be stored with high order in R and low order in R + 1. If R is odd, the multiplier is assumed to be in R, and only the low order of the product is saved (in R). The high order of the product is ignored in this case. The actual work of multiplication is done by the subroutine LMULT. See the explanation of that routine for the multiply algorithm.

MUL: The function of this routine is exactly the same as the LMUL routine, except that to do the actual multiplying it calls the signed integer multiplication subroutine(IMULT) instead of just the logical multiplication subroutine. See the explanation of that routine for the multiply and sign compare techniques.

LDIV: This routine assumes R is even. If R is odd, the preceding even register (R-1) is cleared. Operation then continues as if R-1 is the designated AC instead of R. If R is even, C (R) and C (R + 1) are treated as the high and low order, respectively, of a 32 bit positive dividend which is divided by the EFW, treating the EFW as a 16 bit positive integer. The 16 bit quotient is placed in R + 1 and the remainder in R. If the relative magnitudes of the divisior & dividend are such that the quotient is larger than 16 bits, an arithmetic error condition is set up and both arguments are left unchanged.

The actual division is carried out by the LDIVD subroutine. See the explanation of that subroutine for the divide algorithm. This routine leaves a 32 bit quotient which must be tested to see if it is too large for R + 1. If the divisior is $\emptyset$, LDIV will set up the error condition bit and branch to RET without carrying out any division.

The CC bits are left unchanged.

DIV: The function of this routine is exactly the same as the LDIV routine, except that to do the actual dividing it calls the signed integer divide subroutine (IDIVD) instead of the logical divide subroutine.

LDC: This routine loads the byte indicated by the EFW into the right half of R. The left half of R is cleared.

STC: This routine stores the right half of R into the byte indicated by the EFW. C (R) and the other byte of the receiver word remain unchanged.

SHFT: This routine shifts the given AC left or right a given number of bit positions. There are actually 4 modes of shifting as described below:

(1) Arithmetic shift; in which the sign bit remains unchanged. On a left shift, Ø's enter the low order bits; on a right shift, bit Ø is copied into bit 1. If the sign bit is ever changed in the process of a left shift, the arithmetic error bit is set.

(2) Rotate combined with CCØ.

(3) Simple rotate, in which bits leaving one end enter at the other.

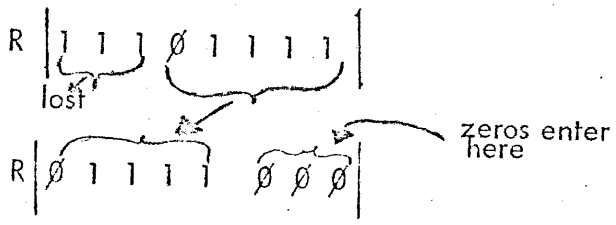
(4) Logical shift, in which bits leaving one end are lost and zeros enter the other end.

Examples of the 4 modes of shifting are shown in Diagram 3. Outlines of the basic shift routines are shown in Diagram 4.

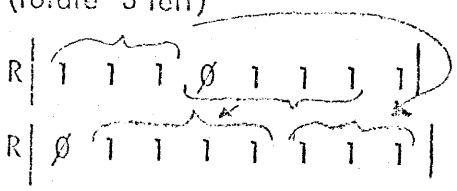
Diagram 3

SHFT examples (using 8 bits only)

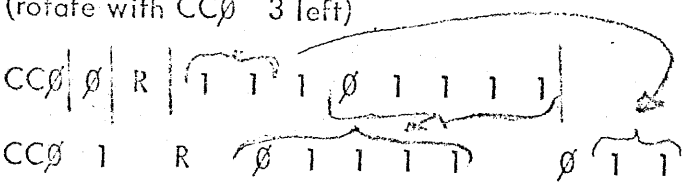
SHFT R, [1 4 Ø 3]
(logical shift 3 left)



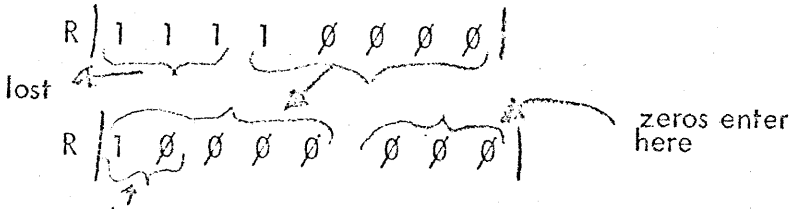
SHFT R, [1 Ø Ø 3]
(rotate 3 left)



SHFT R, [Ø 4 Ø 3]
(rotate with CCØ 3 left)

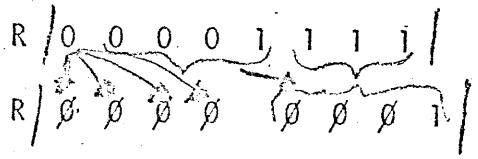


SHFT R, [Ø Ø Ø 3]
(arithmetic shift 3 left)



arithmetic error bit of PSW set when bit Ø & 1 become unequal

SHFT R, [Ø 2 Ø 3]
(arithmetic shift 3 right)



the sign bit remains in place, but also is copied into bit 1

Diagram 4

SHFT Routines:

SHFTL-L	clear CCØ, RL, inc count
SHFTL-R	clear CCØ, RR, inc count
SHFTR-L	clear CCØ, RL, inc count
SHFTR-R	clear CCØ, RR, inc count
SHFTC-L	RL, inc count, fix user CCØ
SHFTC-R	RR, inc count, fix user CCØ
SHFTA-L	clear CCØ, RL, set a.e. bit. if CCØ ≠ bit Ø, inc count, set arithmetic error bit if sign bit changes
SHFTA-R	save sign bit, clear CCØ, RR, add sign bit, inc count

Test Group: (TSTN, TSTZ, TSTO, TSTC)

All 4 of these routines call a common subroutine (TEST) which ANDs together C (R) and the EFW and then sets the user PSW cc bits 1 and 2 accordingly. This is to be done at the beginning of each routine.

TSTN does nothing else.

TSTZ (after calling TEST) then complements the mask (EFW) and ANDs this together with C (R) and stores the result in R. This zeros all selected bits.

TSTO (after calling TEST) also calls the inclusive OR routine for C (R) and EFW. The result goes to R.

TSTC is the same as TSTO except it calls the exclusive OR routine.

See the examples in Diagram 5.

Diagram 5 TEST example (using 8 bits only)

R = 1 Ø 1 Ø 1 1 Ø Ø
M = 1 1 Ø Ø Ø 1 1 Ø

TSTN: R = 1 Ø 1 Ø 1 1 Ø Ø CC 1 = 1, CC 2 = 1

TSTZ: R = Ø Ø 1 Ø 1 Ø Ø Ø CC 1 = 1, CC 2 = 1

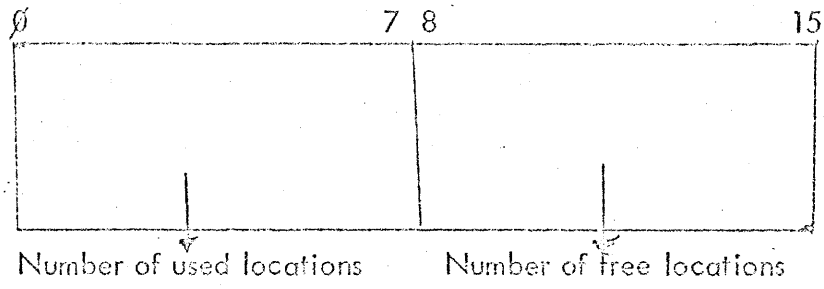
TSTO: R = 1 1 1 Ø 1 1 1 Ø CC 1 = 1, CC 2 = 1

TSTC: R = Ø 1 1 Ø 1 Ø 1 Ø CC 1 = 1, CC 2 = 1

Push Group:

Before using a PUSH-group instruction a user program must set up a push down list (PDL), within its own core space by doing three things: (1) allot a number (N) of words for the list, (2) store the first address of the push down list allotment in R14 (8), and (3) store the number N in R15 (8). All PUSH and POP-type instructions modify these two registers. The actual arrangment of the push down list counter in R15 is as shown below:

PDL Counter



PUC: This routine will increment the PDL pointer, increment PDL counter left half, and decrement the PDL counters right half. It will actually make use of a subroutine PU to do this work, because all other PUSH-type instructions must do the same things to the pointer & counter.

POC: This routine will do the exact reverse of the work of PUC. It will call the subroutine PO to do all its work.

PUSH: This routine will store the EFW on the PDL at the address specified by the current PDL pointer. It will then call the PU subroutine to modify the pointer & counter.

POP: This routine will first call the PO routine to modify the PDL pointer and counter, and then it will remove the last word stored on the list and place it in the EFA.

PUB: This routine stores the PC in the next location on the PDL, then calls the PU subroutine, then loads the EFA into R1Ø (the saved user PC).

POB: This calls PO to modify the PDL pointer & counter, then loads the sum of the last word stored on the PDL + the EFW into R1Ø.

PUL: This routine stores the (stacked) R2 on the PDL, places the updated PC (R1Ø) into R2, calls PU, places the updated PC on the PDL, calls PU again, then places the EFA into R1Ø.

POL: Calls PO to modify the PDL pointer and counter, loads the sum of the last word stored on the PDL and EFW into R1Ø, calls PO again, then loads the new last word on the PDL into R2.

Special Subroutines:

CC:	set CCØ, 1, 2
CC12:	set CC1, 2
INOR:	inclusive OR
EXOR:	exclusive OR
IABS:	convert 2 Args. to pos. & compare signs
LMULT:	logical multiply
IMULT:	signed multiply
LDIVD:	logical multiply
IDIVD:	signed divide
AETR:	set arithmetic error bit
TEST:	general TEST for selected bits
PU:	push-type pointer-counter modification
PO:	pop-type pointer-counter modification

CC: The cc bits Ø, 1, 2 of the traped user program PSW must be cleared and replaced with the current cc bits.

CC12: Similar to above, except leave CCØ of user PSW unchanged.

INOR: Here the logic is AVB = (Not A/\B) + A.

EXOR: Here the logic is AVB = —(A/\B) *2 + A + B.

IABS: This subroutine is needed by the signed multiply and signed divide routines. Its function is two-fold. First it must convert both arguments to positive (i.e. get their absolute values). Secondly it must set an indicator as to whether or not the signs of the two arguments were alike. This latter function is carried out as shown in the following diagram.

	Case 1	Case 2	Case 3	Case 4
Start	SW = ∅	SW = ∅	SW = ∅	SW = ∅
Argument A	Pos leave SW alone	Pos leave SW alone	Neg set SW = 1	Neg set SW = 1
Argument B	Pos leave SW alone	Neg reverse status of SW	Pos leave SW alone	Neg reverse status of SW
Final Result	SW = ∅	SW = 1	SW = 1	SW = ∅

LMULT: This subroutine carries out the actual multiplication of one 16 bit word times another. It assumes these numbers are positive 16 bit integers. It will be used by the LMUL EOP routine as a subroutine, and by the MUL EOP routine as a sub-sub-routine.

The example shown by Diagram 1 demonstrates fully how the routine is to work. The basic multiply loop is used a maximum of 16 times. The algrorithim for the procedure is:

$$\text{Product} = X_{15}\,(2^0)\,M + X_{14}\,(2^1)\,M + \ldots + X_0\,(2^{15})\,M$$

where X_i = ith bit of the multiplier
and M = the multiplicand.

Note that the result left by this routine is always 32 bits.

Diagram 1

LMULT (example using 6 bits only)

Stage	Product	Multiplicand	Multiplier	CC
0	0 0 0 0 0 0 0 0 0 0 0 0 + 1 1 0 0 1	0 0 0 0 0 0 0 1 1 0 0 1	1 0 1 0 1 1	0
1	0 0 0 0 0 0 0 1 1 0 0 1 + 1 1 0 0 1 0	0 0 0 0 0 0 1 1 0 0 1 0	0 1 0 1 0 1	1
2	0 0 0 0 0 1 0 0 1 0 1 1	0 0 0 0 0 1 1 0 0 1 0 0	0 0 1 0 1 0	1
3	0 0 0 0 0 1 0 0 1 0 1 1 + 1 1 0 0 1 0 0 0	0 0 0 0 1 1 0 0 1 0 0 0	0 0 0 1 0 1	0
4	0 0 0 1 0 0 0 1 0 0 1 1	0 0 0 1 1 0 0 1 0 0 0 0	0 0 0 0 1 0	1
5	0 0 0 1 0 0 0 1 0 0 1 1 + 1 1 0 0 1 0 0 0 0 0	0 0 1 1 0 0 1 0 0 0 0 0	0 0 0 0 0 1	0
6	0 1 0 0 0 0 1 1 0 0 1 1	0 1 1 0 0 1 0 0 0 0 0 0	0 0 0 0 0 0	1

$31_{(8)} \times 53_{(8)} = 2063_{(8)}$

IMULT: This subroutine carries out a signed integer multiply, leaving a double precision product, by calling sucessively IABS and LMULT. If the signs of the arguments are unlike, the product is negated.

LDIVD: This subroutine carries out the division of a 32 bit positive integer by a 16 bit positive integer. The algorithm of the routine is exactly the same as that used by common long division. The dividend is successively tested from left to right until a group of bits is found which is greater than the divisior. At this point the divisor is subtracted from these bits and the bit of the quotient corresponding to the last used bit of the dividend is set. Then the unused bits of the dividend are placed to the right of the remainder to form a new dividend, and the process is repeated until all bits of the dividend have been used. Exactly 32 passes through the basic divide loop are required. An example is shown below and further illustrated in Diagram 2.

```
         10010
101    |1011101
      - 101
       -------
         01101
         -101
      ---------
            11
```

The actual setting of the correct bits of the quotient is easily accomplished by using the same register for the dividend and the quotient. Then as the dividend is rotated left bit by bit into the remainder storage registers (see diagram 2) for comparison with the divisor, the low order bits of the dividend register will successively become the high order and decreasing bits of the quotient. When the divisor does divide into a protion of the dividend, this is accounted for by setting the current low bit of the dividend/quotient register.

The remainder must be double precision because the dividend is, and the divisor must be treated as double precision to accomodate subtraction from the remainder and also because the negative of a 16 bit positive divisor (which is possible with the LMUL instruction) may require 17 bits.

Diagram 2

LDIVD (example using 7 bits only)

Stage	Remainder	Dividend & Quotient	
Ø	ØØØØØØØ	1Ø111Ø1	Clear rem., load dividend
1	ØØØØØØ1 -1Ø1	Ø111Ø1Ø	Rotate Add neg. divisor; reject if result neg.
2	ØØØØ1Ø -1Ø1	111Ø1ØØ	Rotate Try divisor again & reject neg. resu
3	ØØØ1Ø1 -1Ø1	11Ø1ØØØ	Rotate Try divisor; it fits, so increment quotient
	ØØØØØØØ	11Ø1ØØ1	
4	ØØØØØ1 -1Ø1	1Ø1ØØ1Ø	Rotate Try divisor & reject neg. result
5	ØØØØ11 -1Ø1	Ø1ØØ1ØØ	Rotate Try divisor & reject neg. result
6	ØØØ11Ø -1Ø1	1ØØ1ØØØ	Rotate Try divisor; it fits so increment quotient
7	ØØØØ11 -1Ø1	ØØ1ØØ1Ø	Rotate Try divisor & reject neg. result
Final	ØØØØ11	ØØ1ØØ1Ø	

$$5\overline{)135_{(8)}} \quad \frac{22_{(8)}\ R\ 3}{}$$

IDIVD: This subroutine carries out a signed integer divide, leaving a signed quotient and a signed remainder. It does this by calling IABS and LDIVD. If the signs of the divisor and dividend are unlike it negates the quotient. The remainder agrees with the dividend in sign.

AETR: This subroutine merely sets bit 2 of the user PSW in the stack as an arithmetic error condition indicator. This is done by calling the inclusive OR routine instead of merely adding ØØØØØ2 to the PSW, in order to prevent unusual circumstances.

TEST: This is the generalized TST subroutine. It ANDs together the EFW and C (R), then sets the user PSW cc bits 1, 2 accordingly.

PU: This is the general push-type instr. subroutine to modify the PDL pointer & counter as described for the PUC instruction. This routine sets the push down error bit if the PDL counter right half = Ø when the routine is called.

PO: This is the general POP-type inst. subroutine to modify the PDL pointer & counter as described for the POC instruction. This routine sets the push down error bit if the PDL counter left half = Ø when the routine is called.

3. Operating Procedure

3.1 Loading Procedure

Not yet available *

3.2 Switch Settings

None required or available.

3.3 Start-up Procedure

None required. The program operates only when an EOP is trapped in a running user program.

3.4 Command Language

Not applicable.

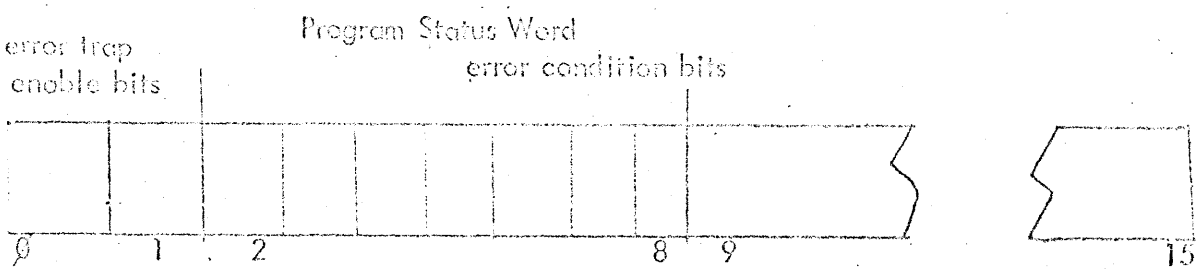
3.5 Operation

Not applicable.

* The loading tape will also store the entry address of the program in R11 of all priority levels of the machine.

3.6 Error Recovery

The PDP-X is designed so that certain types of programming and program operation errors may cause error traps. Such error conditions are indicated by setting bits in the PSW. The first 9 bits of the PSW are allocated for this purpose.



Error trap enable bits:

bit 0: enables arithmetic error trap if set
bit 1: enables all other error traps if set

Error condition bits:

bit 2: arithmetic error
bit 3: push down list error
bit 4: non-extended memory error
bit 5: addressing exception error
bit 6: I/O error
bit 7: priveledged instruction error
bit 8: read only violation error

If one of the error condition bits is set upon execution of an instruction, and if the corresponding error enable bit is set, an error trap will occur in the following manner:

The address of the instruction which caused the error will be loaded into $R16_{(8)}$ and a branch to the address contained in $R17_{(8)}$ will occur. Presumably the user will have stored the address of his error trap handler in R17.

The Extended OP Simulator must simulate error traps if the trapped EOP instruction is such that an error condition exists. The following types of errors must be dealt with: (1) Arithmetic error: If the result of a SUB instruction, or the quotient of a DIV or LDIV instruction is too large for its intended AC, the arithmetic error condition bit (bit 2) is set. On an arithmetic left shift, if the sign bit gets changed at any point of the shift, the arithmetic error condition bit is set.

(2) Push down list error: If the right half of the PDL counter is $\emptyset$ when a PUSH-type instruction is encountered, or if the left half of the PDL counter is $\emptyset$ when a POP-type instruction is encountered, bit 3 of the PSW is set.

(3) Non-extended memory error: If the EFA is too large for the memory of the user's machine, bit 4 is set.

(4) Addressing exception error: If a reference to R1 (the PC) is made, bit 5 is set.

If any attempt is made to write the Register Group bits of the PSW, it is ignored.

Before returning to the user program, the "RET" routine compares the error condition bits with the error enable bits. If a trap condition exists, the trap will be simulated and a branch to the user's error trap handler will occur, instead of a return to his running program.

4. Internal Environment

4.1 Trade-offs

There are shorter routines which could have been used for multiplication and division. Namely, multiplication can be done by merely adding the multiplicand to the AC a number of times equal to the numerical value of the multiplier, and division can be done similarly by subtraction. These routines though shorter, in terms of instructions required, are a great deal more time consuming than the routines which were chosen.

No attempt has been made to carry out the operation in the manner that will be used by the Model II hardware. The program will execute these operations in the most efficient way they can be done by software, and will produce the same results, in all respects, as are produced by Model II hardware.

4.2 Software Interfaces

Not applicable.

4.3 Conventions

Subroutine calling sequences are all of the form:

```
BAL     SUBR
ARG 1   ADDRESS
ARG 2   ADDRESS
  .
  .
  .
```

ARGN ADDRESS
RETURN

4.4 Language

The program will be written in PDP-X assembly language, using Basic OPS only.

5. External Environment

5.1 Execution Speed

Not yet available .

5.2 Use

With the Extended Op Simulator PDP-X Model I users may employ EOP instructions in their programs. When and EOP instruction in the user's program is encountered, the Model I will store its (updated) PC in general register R1Ø, the EOP instruction word in R11, the EFA in R12, and then branch to the address contained in R12. This address will be the starting address of the Extended Op Simulator. The Extended Op Simulator will save the user program AC's, execute the EOP instruction, then restore the user's AC's, update them with the results of the EOP instruction, and finally branch back into the user program at the address contained in R1Ø, i. e., the instruction following the EOP instruction.

5.3 Interface

This program is designed to be used by running PDP-X Model I system or applications program. Its function is to allow these programs to employ EOP instructions normally available only on Model II. It must reside in core simultaneously with the program that will call it. When EOP instructions are encountered, the machine will automatically branch to the Extended Op Simulator. After execution of the EOP, the Extended Op Simulator will return control to the user program making everything appear that the program is running on a Model II machine.

5.4 Examples of Use

Suppose a user program contains the following sequence of steps:

```
15Ø6/   LDA 4,[X]        ; (long form) assume X = 5
151Ø/   SUB 4, [35]      ; (long form)
1512/   BM  ADDR
```

When the CMP instruction is encountered, the EOP trap registers are loaded as follows:

```
R1Ø / ØØ1512      ;UPDATED PC
R11 / 15Ø511      ; OP = 6, R = 4, X = 1, D1 = 112
R12 / ØØ1511      ; EFA
R13 / ØØ7ØØØ      ; ST. ADDR. of Ex. Op. Sim. (which
                  ; was loaded earlier by the
                  ; Ex. Op. Sim. LOADING PROCEDURE)
```

Then 7ØØØ is loaded from R13 into the PC, and the operation of the Extended Op Simulator begins.

The Extended Op Simulator will store the contents of the general register on the stack as follows:

```
; STACK A (assume priority Ø)
71ØØ (AØ) / Ø          ; Assume this was the PSW
                       ; when the trap occured
71Ø1 (A1) / 71Ø4       ; pointer to effective AC (R4)
71Ø2 (A2) / C (R2)
71Ø3 (A3) / C (R3)
71Ø4 (A4) / C (R4) = 5
etc.
```

The Simulator will then carry out the operation 5-35 = -3Ø, with CCØ = Ø, (no carry), CC1 = 1 (negative result), and CC2 = 1 (non-zero result). Then - 3Ø, is stored at 71Ø4 (A4 = the effective AC for the SUB) and the condition codes are ORed into 71ØØ giving C (AØ) = ØØØØØ3. Finally the Extended Op Simulator loads contents of all the general registers on the stack into the actual general register and executes a B @ 1Ø back to the instruction at 1512 in the user's program.

6. Documentation

6.1 Major Aspects

(1) Function of the programs.
(2) Loading procedure.
(3) Full explanation of all EOP instructions (if these are not provided with standard Model I manuals).

6.2 Checkout

(1) Debugging separate portions of the program, using PDP-X Simulator on a PDP-7 or 9.
(2) Test and debug the entire program running as a unit on a PDP-7 or 9 with the PDP-X Simulator.
(3) Test and exercise the entire program on a real PDP-X when such is available.

6.3 Marketing

This relatively small program will allow PDP-X Model I users to employ the powerful Extended Op instructions usually available only on the larger Model II PDP-X. It will save them from writing and loading separate subroutines to carry out such operations as multiply, divide, inclusive OR, exclusive OR, and use of a push down list. By using a relatively small portion of the PDP-X core, it in effect converts the Model I into a Model II (though not quite as fast).

CHART 1

Extended Op Simulator

Macro Flow Chart

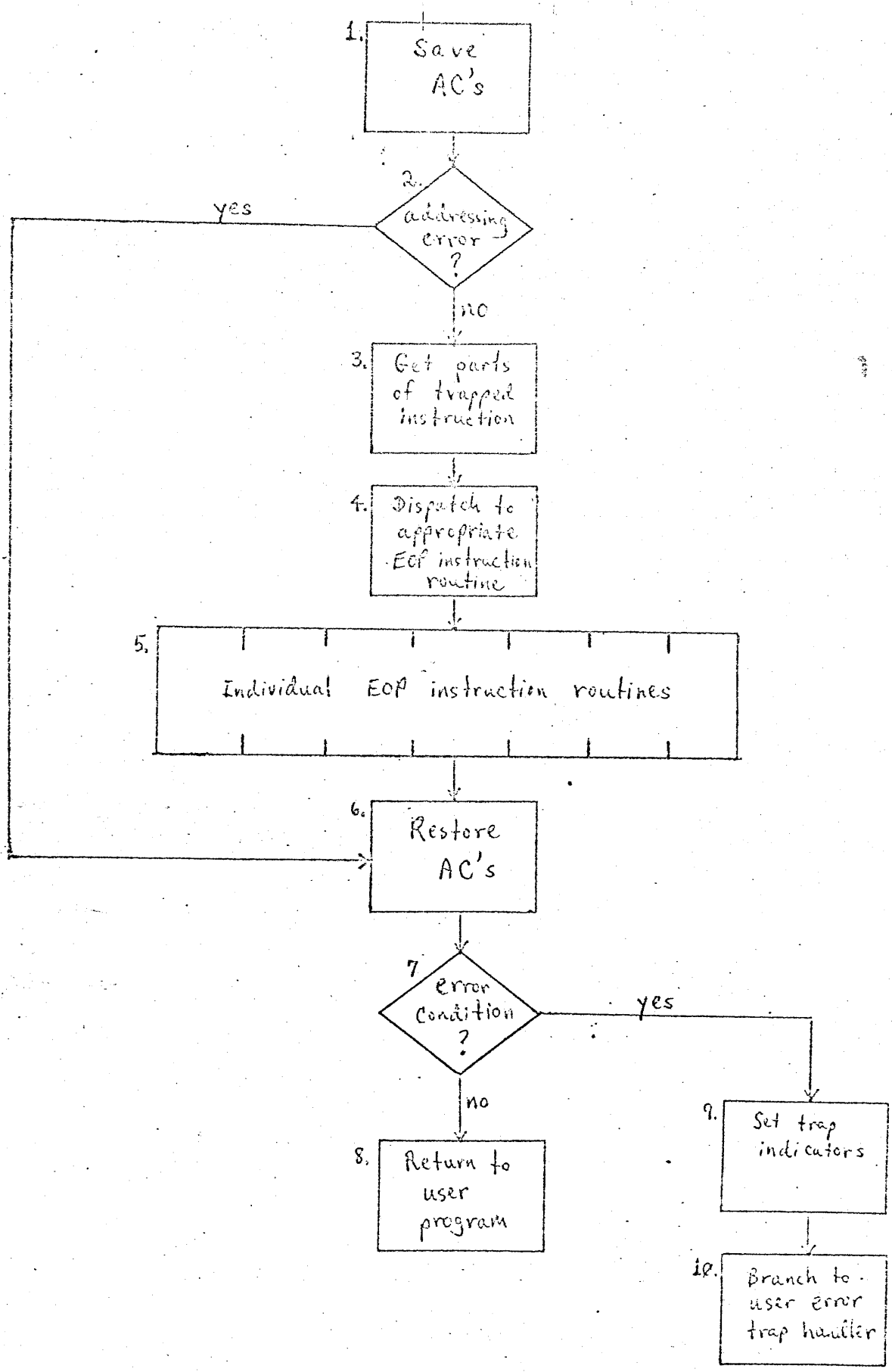


CHART 2

SAV

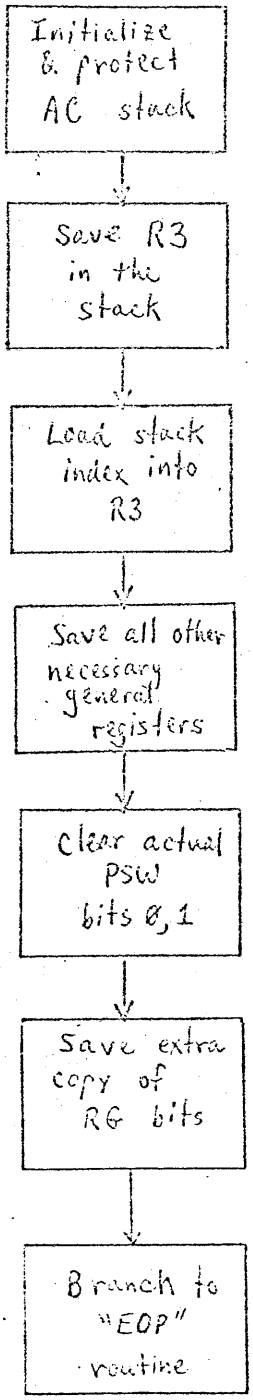

Initialize & protect AC stack
Save R3 in the stack
Load stack index into R3
Save all other necessary general registers
Clear actual PSW bits 0,1
Save extra copy of RG bits
Branch to "EOP" routine

CHART 3

ECP

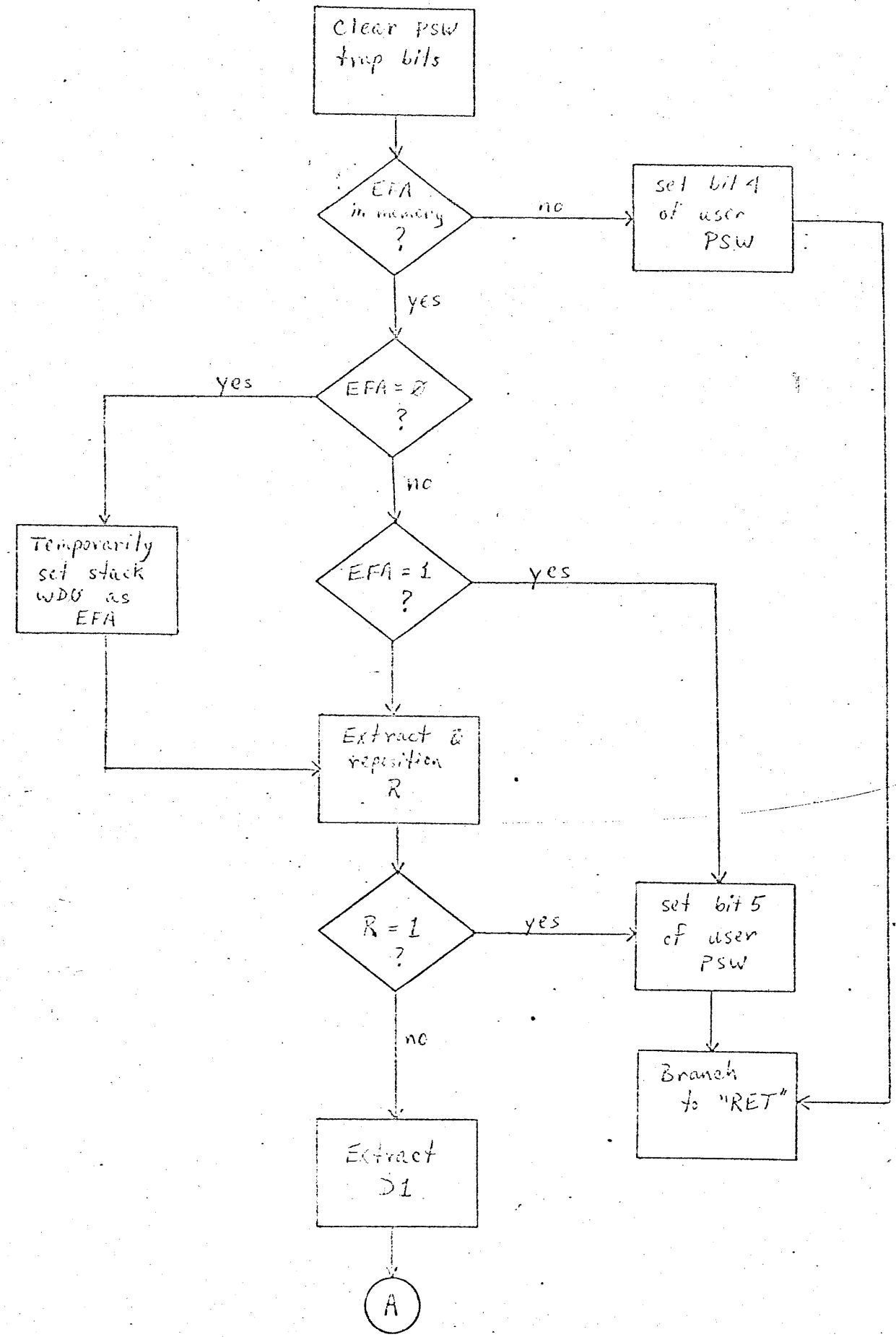

Clear PSW trap bits
EFA in memory ?
no
set bit 4 of user PSW
yes
EFA = 0 ?
yes
Temporarily set stack WDW as EFA
no
EFA = 1 ?
yes
Extract & reposition R
R = 1 ?
yes
set bit 5 of user PSW
Branch to "RET"
no
Extract D1
A

CHART 3
(continued)

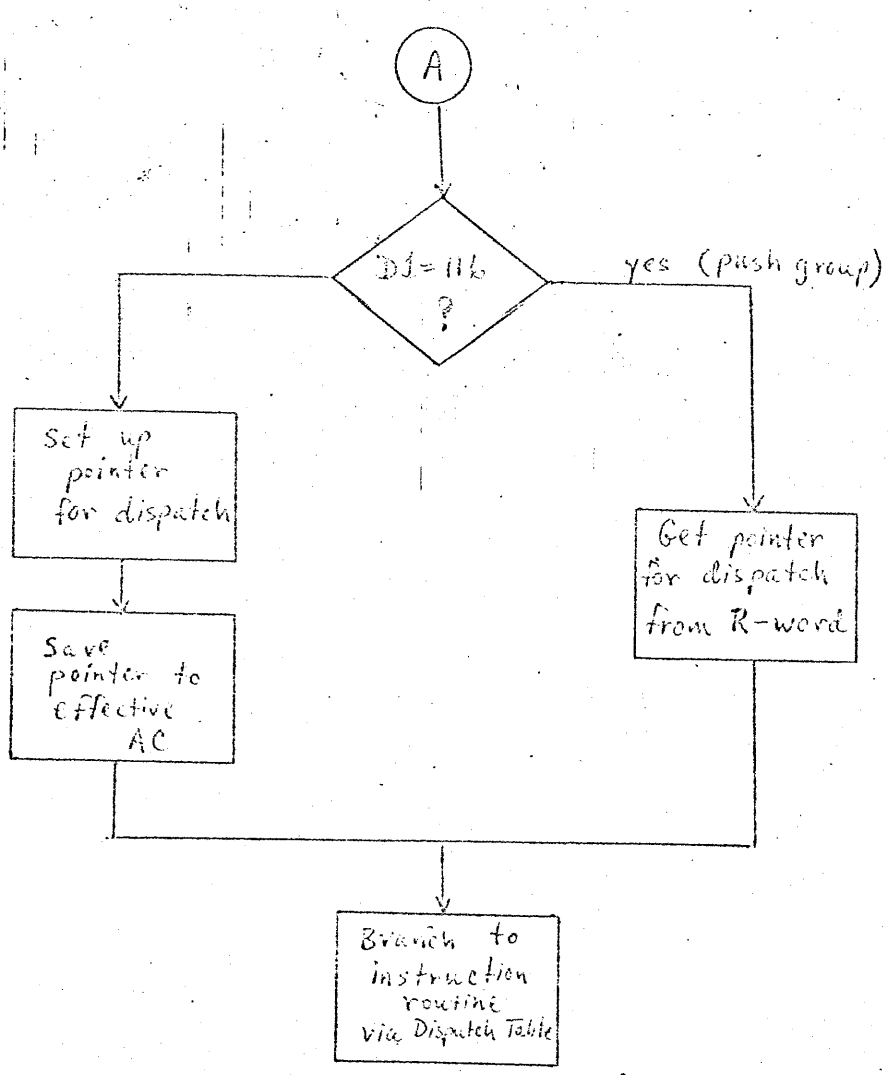


CHART 4
RET

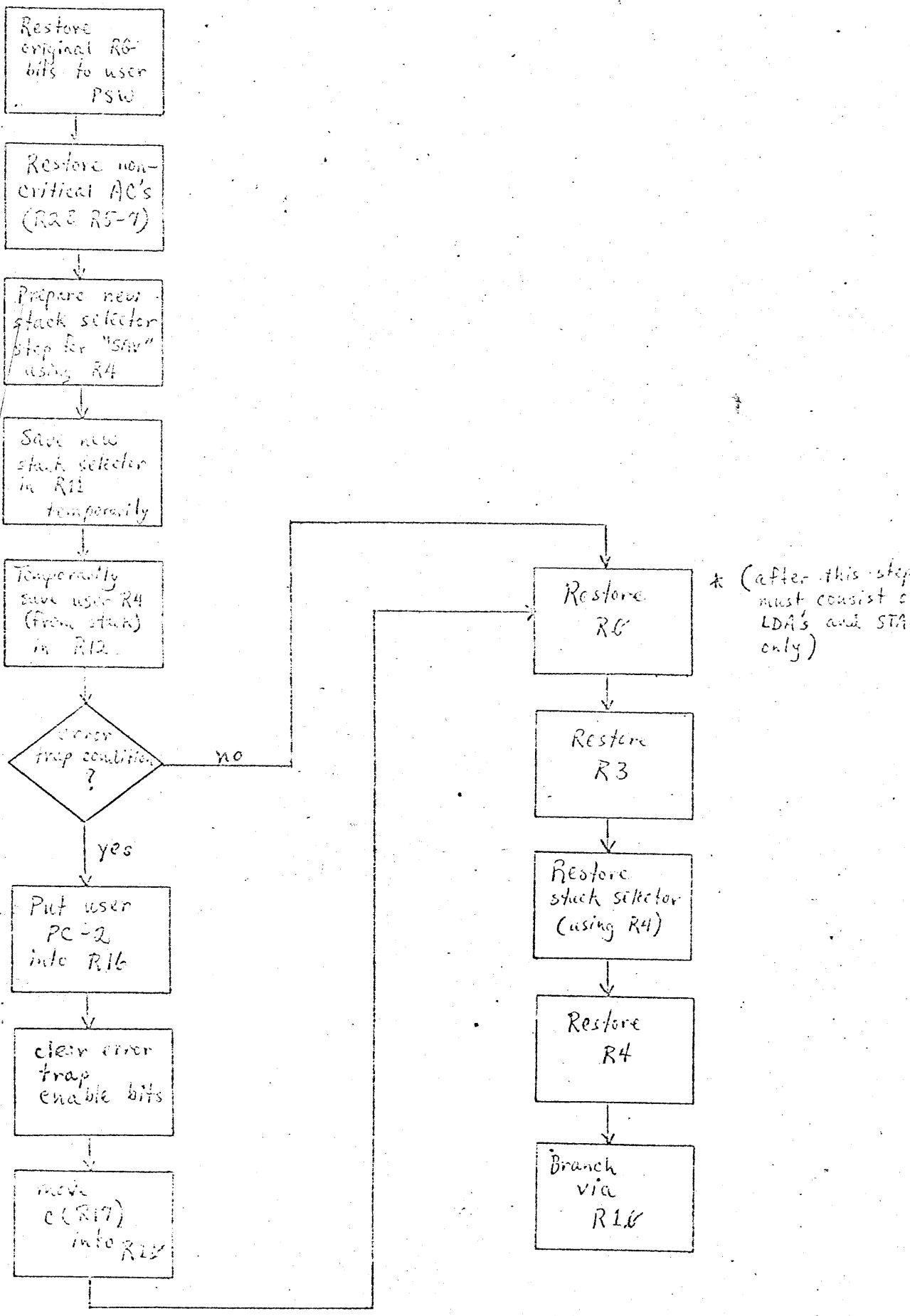

Restore original R0 bits to user PSW
Restore non-critical AC's (R2 & R5-7)
Prepare new stack selector step for "SAV" using R4
Save new stack selector in R12 temporarily
Temporarily save user R4 (from stack) in R12
Stack trap condition ?
no
yes
Put user PC-2 into R16
clear error trap enable bits
move C(R17) into R15
Restore R0
Restore R3
Restore stack selector (using R4)
Restore R4
Branch via R16
* (after this step must consist o LDA's and STA only)

CHART 5

SUB

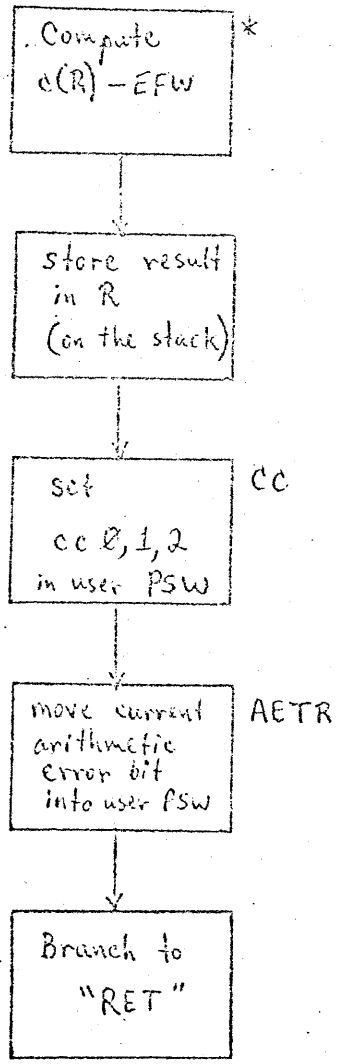
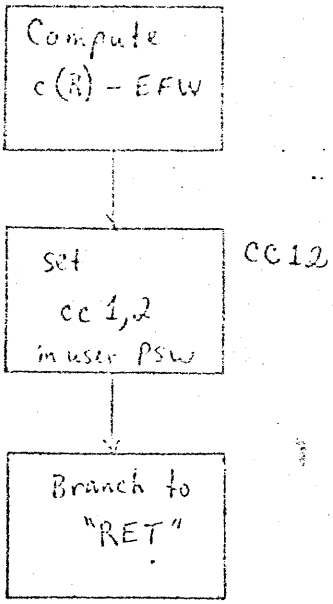


CHART 6

CMP



* R = effective AC for the trapped instruction.

CHART 7

LCMP

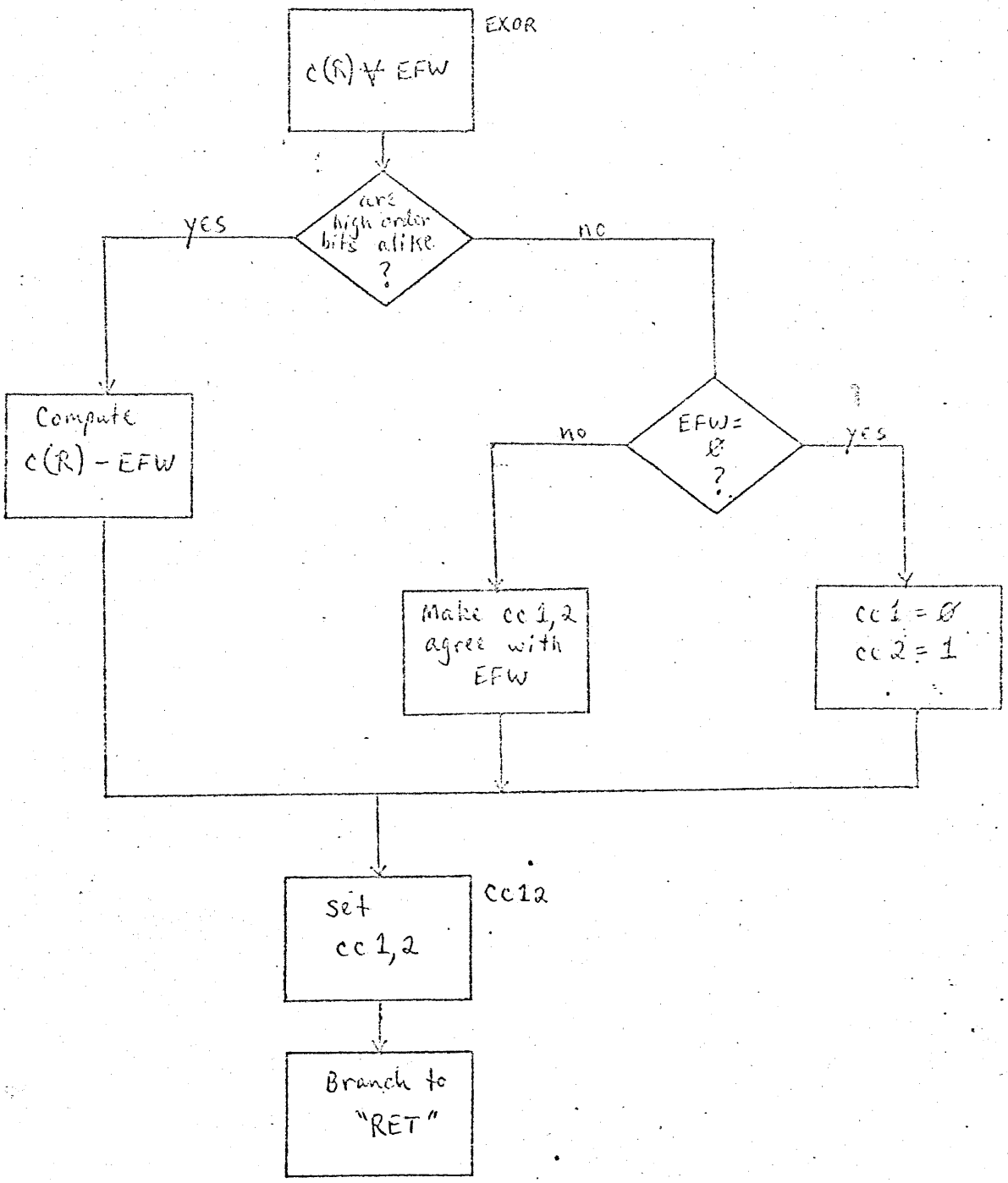

EXOR
c(R) ∀ EFW
are high order bits alike ?
yes
no
Compute c(R) - EFW
EFW = ∅ ?
no
yes
Make cc 1,2 agree with EFW
cc 1 = ∅
cc 2 = 1
set cc 1,2
cc12
Branch to "RET"

CHART 8

LMUL & MUL

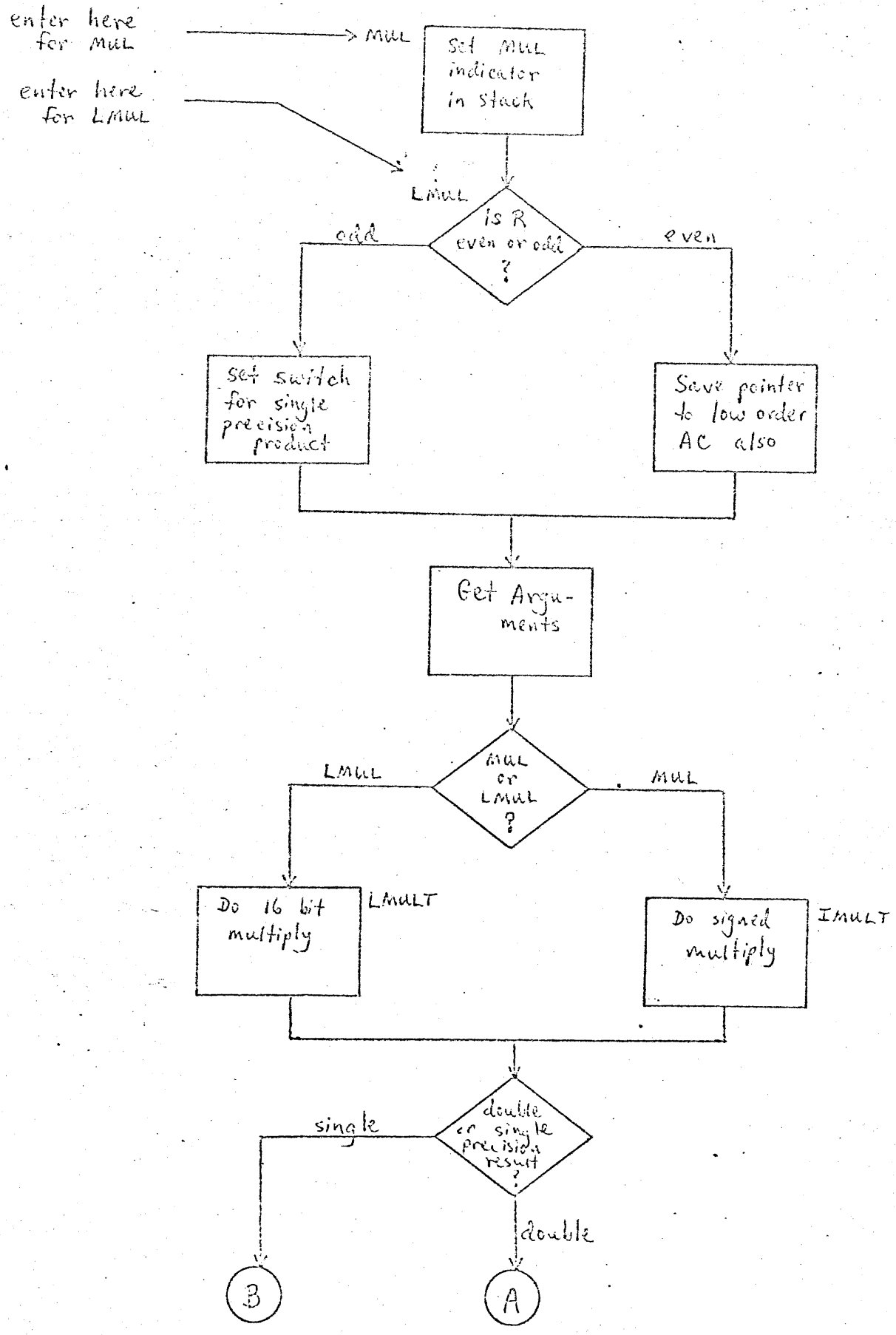


CHART 8
(continued)

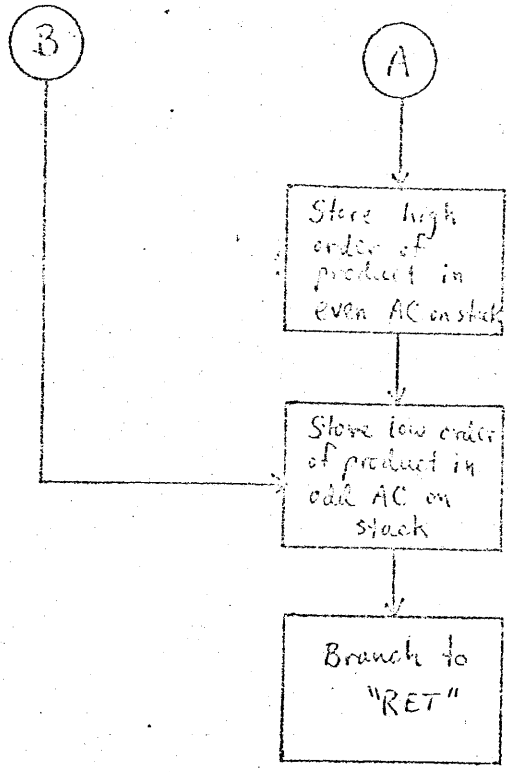


CHART 9

LDIV & DIV

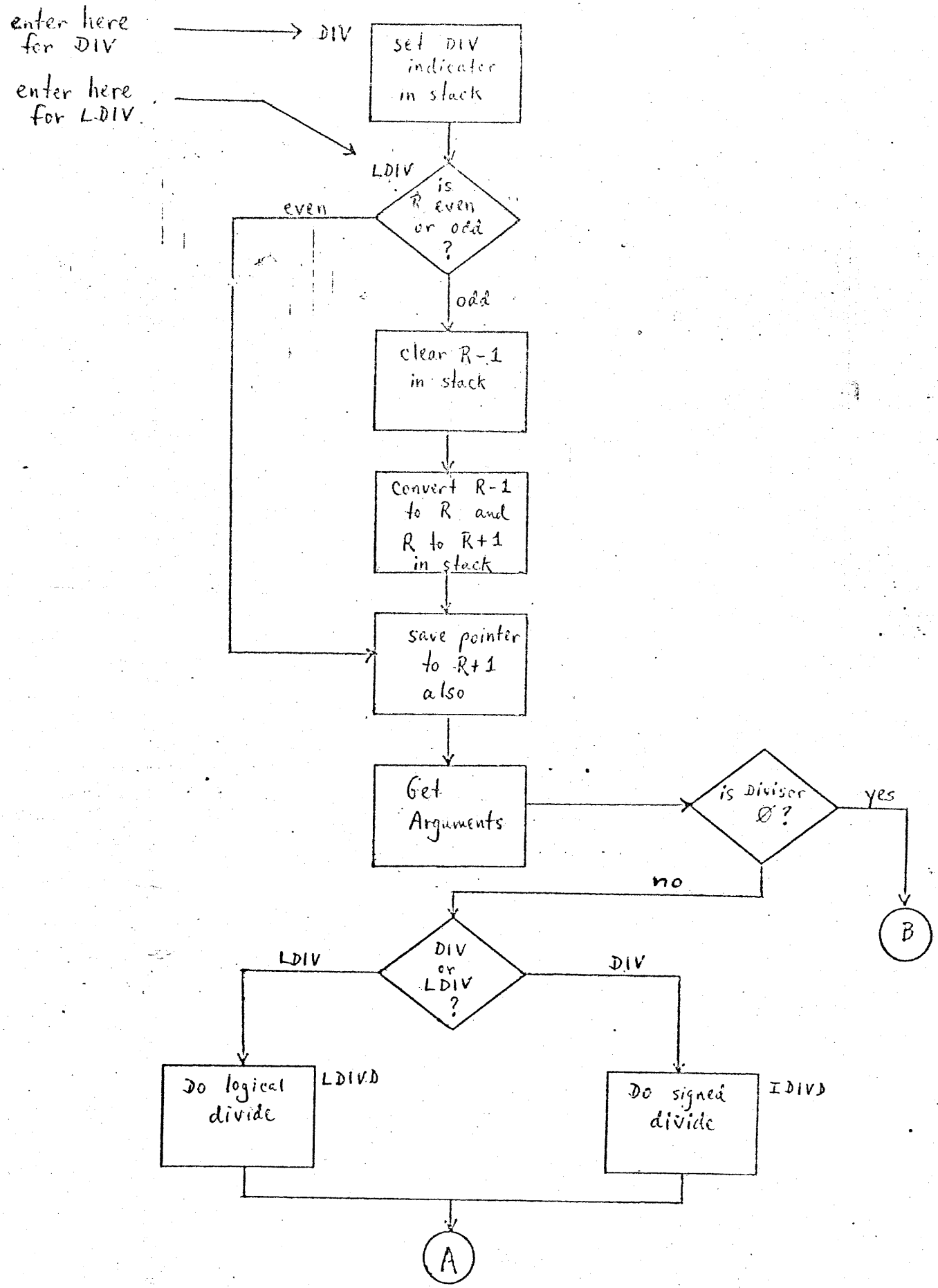


CHART 9
(continued)

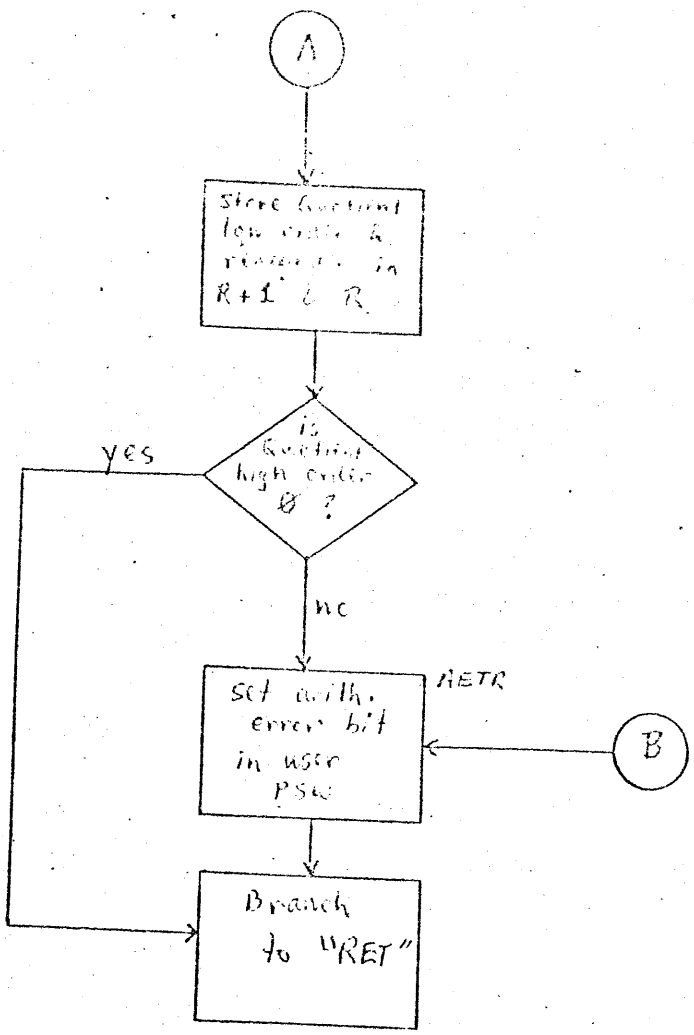


CHART 10

LDC

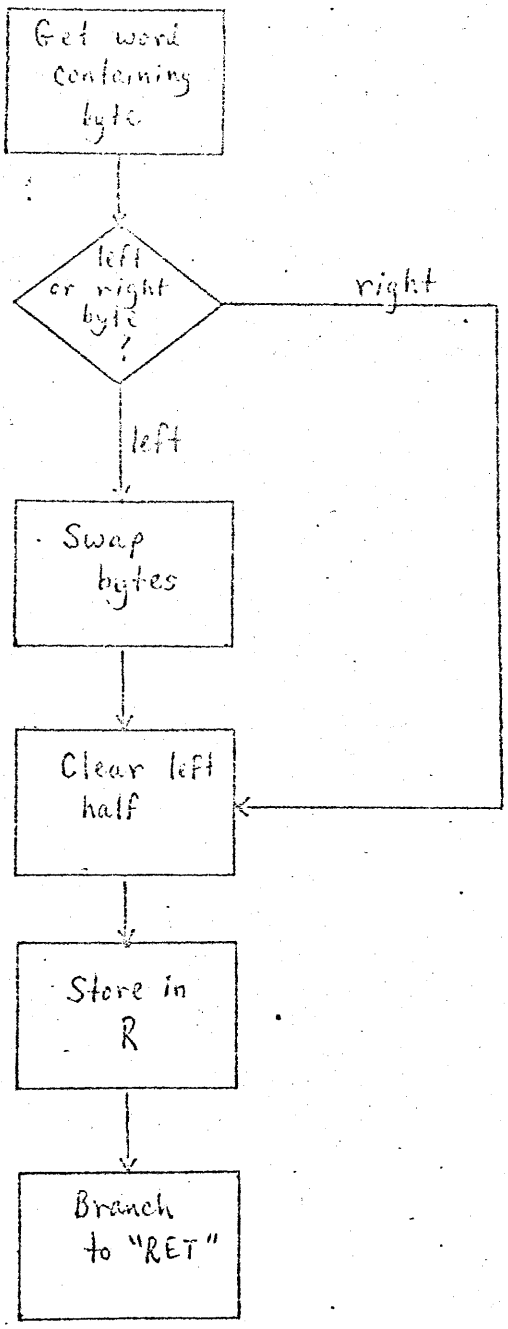

Get word containing byte
left or right byte ?
right
left
Swap bytes
Clear left half
Store in R
Branch to "RET"

CHART 11

STC

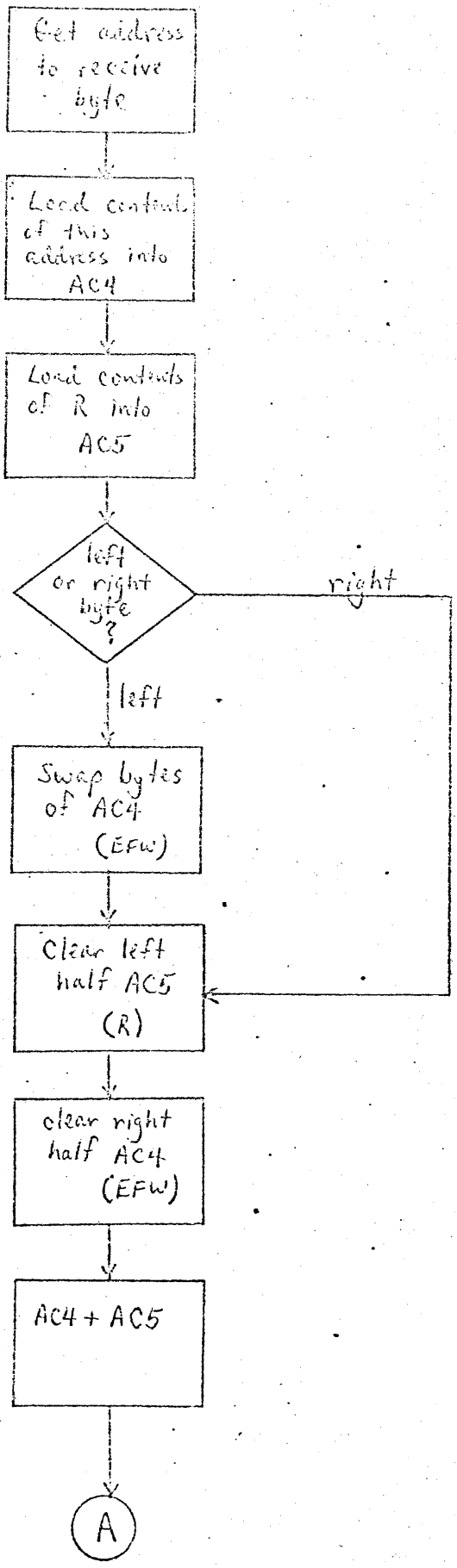

Get address to receive byte
Load content of this address into AC4
Load contents of R into AC5
left or right byte ?
right
left
Swap bytes of AC4 (EFW)
Clear left half AC5 (R)
clear right half AC4 (EFW)
AC4 + AC5
A

CHART 11
(continued)

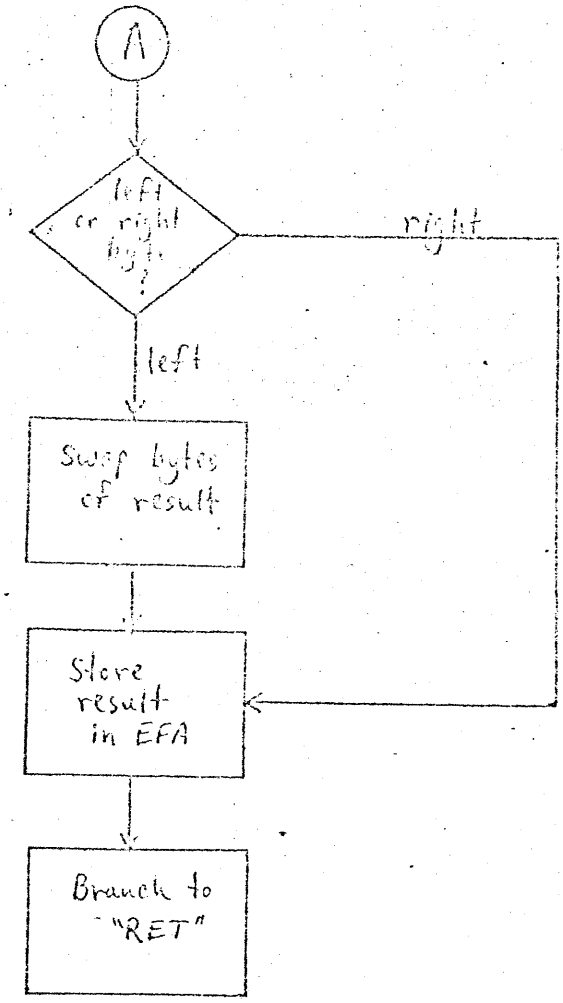


CHART 12

SHFT

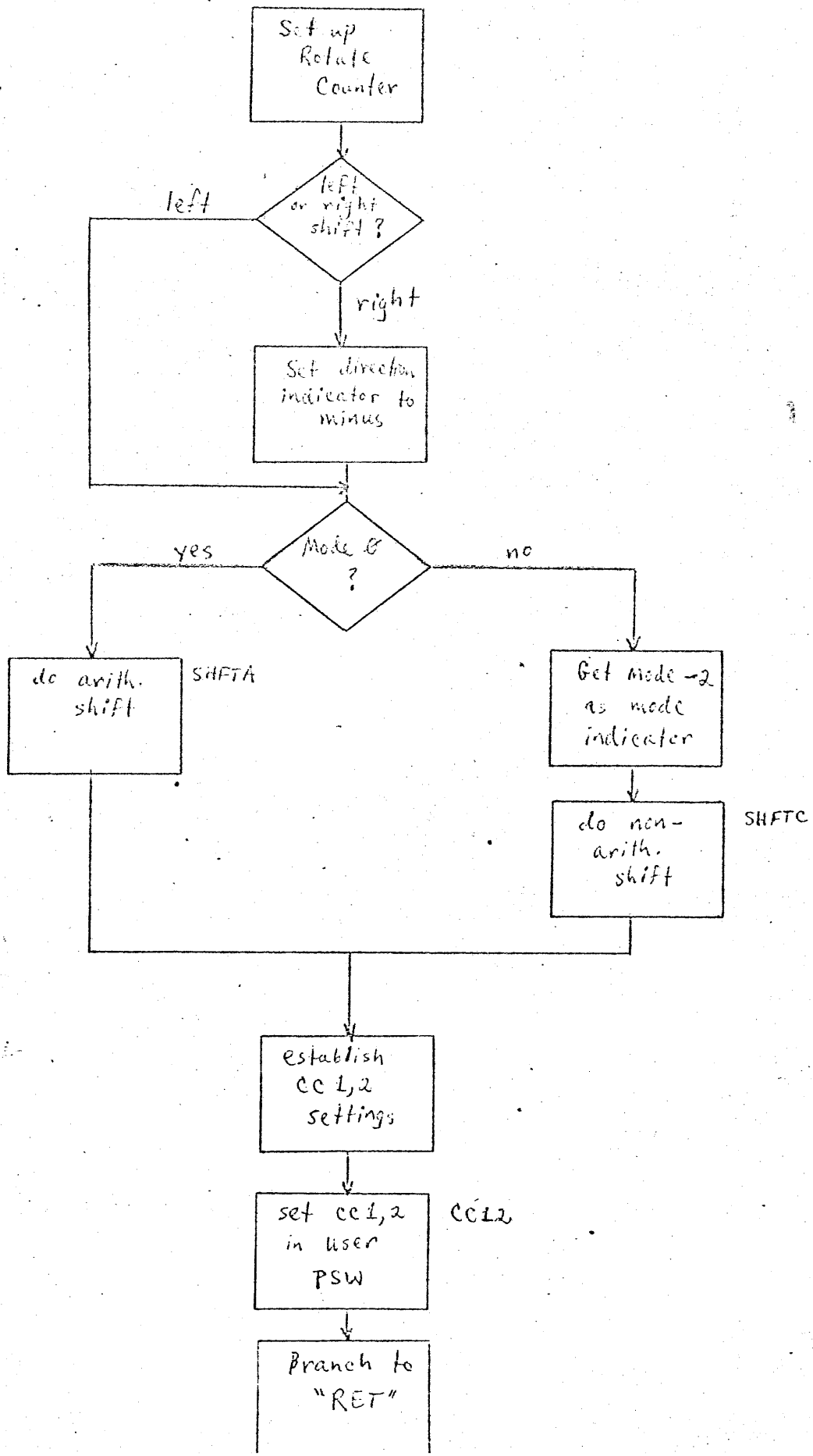


CHART 12A

SHFTA

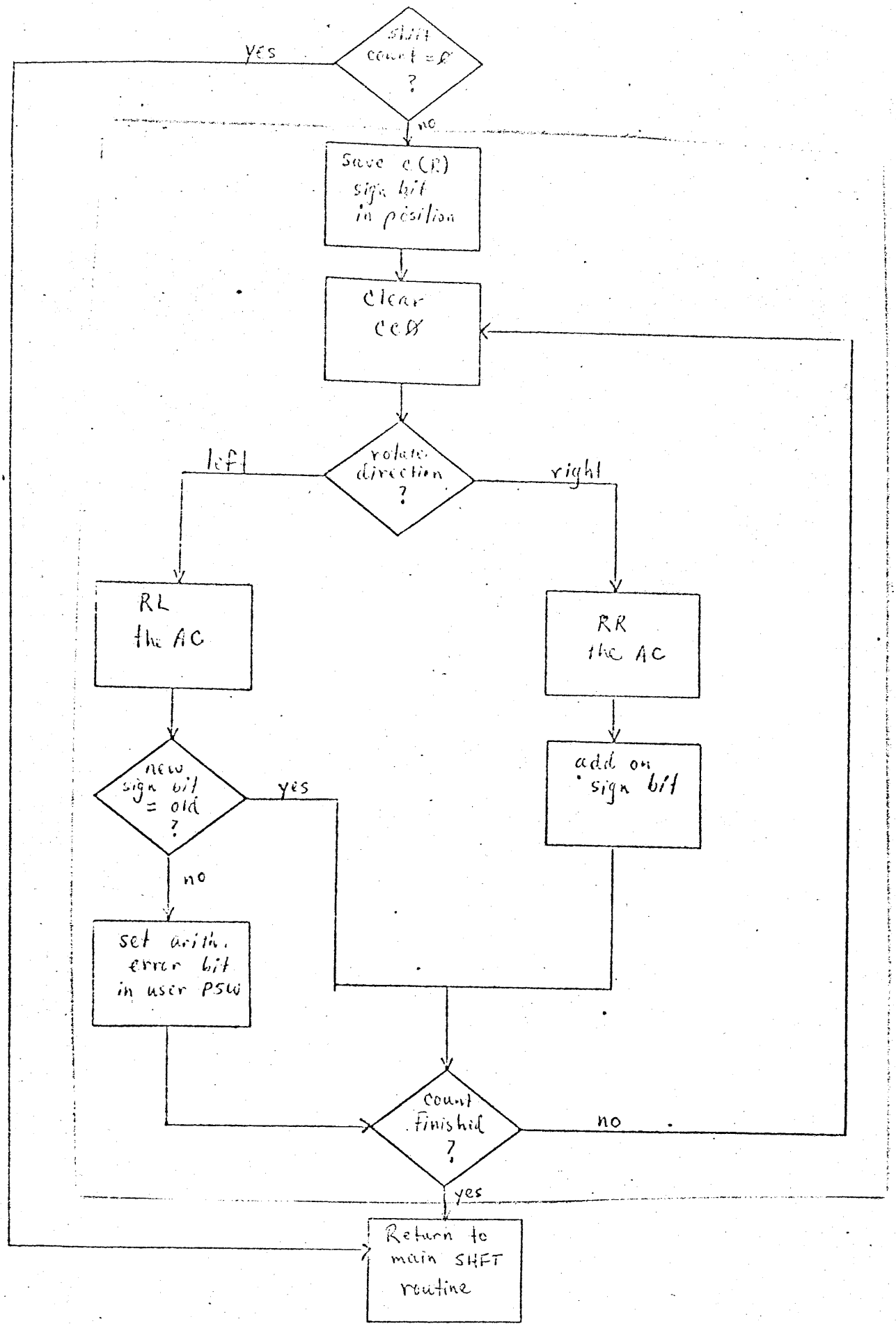
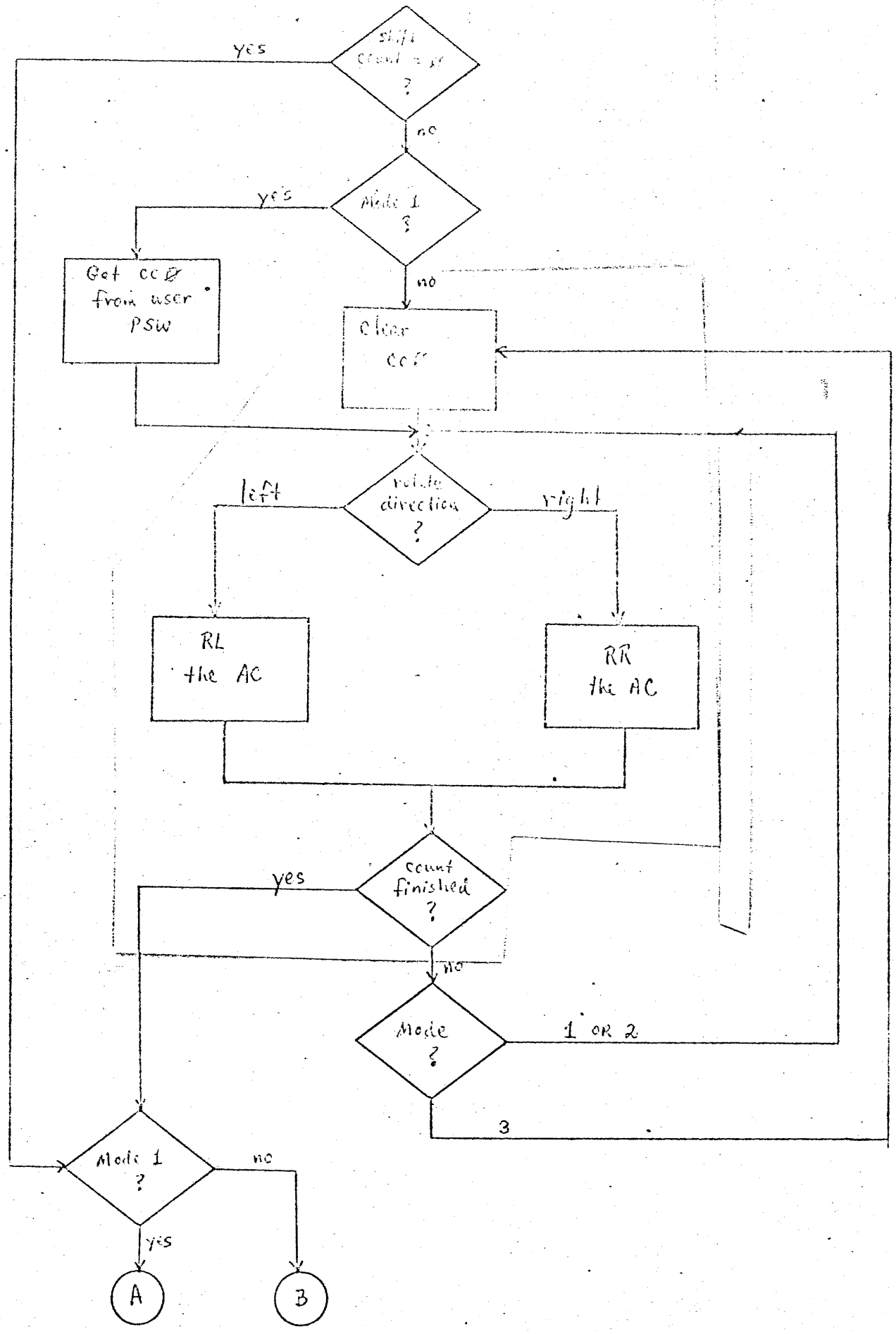


CHART 12 B

SHFTC



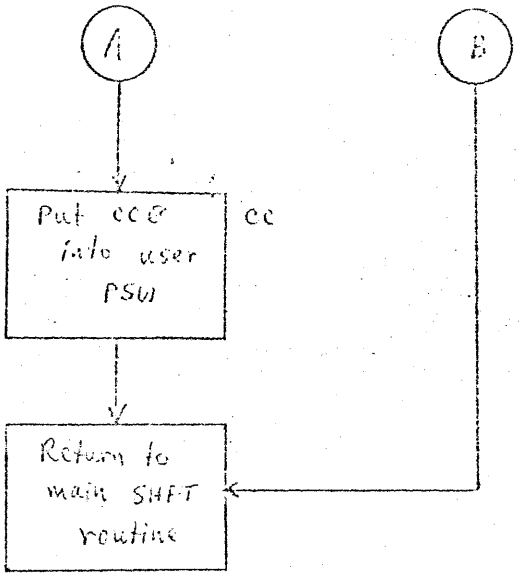

A
B
Put cc & into user PSW
cc
Return to main SHIFT routine

CHART 13

TSTN

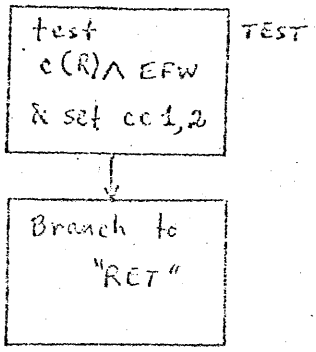


CHART 14

TSTE

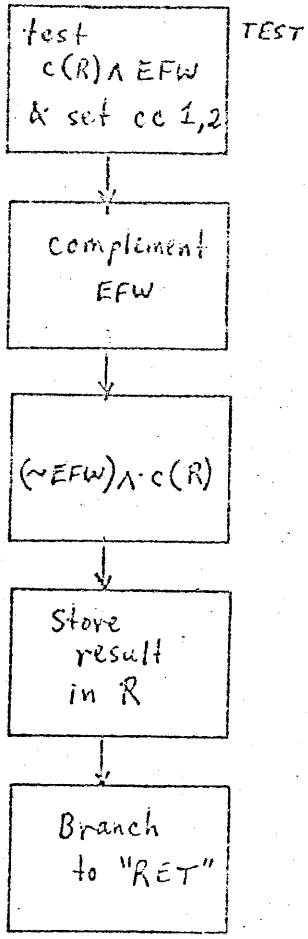


CHART 15

TSTO

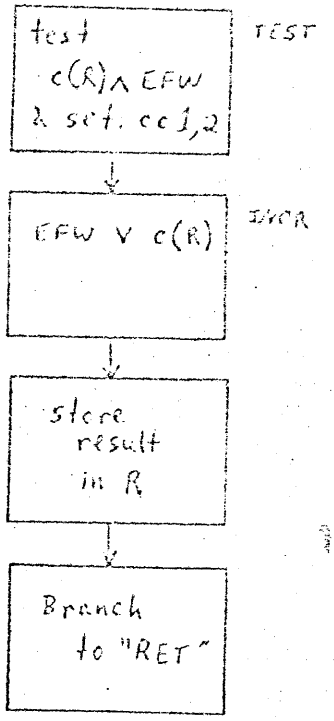


CHART 16

TSTC

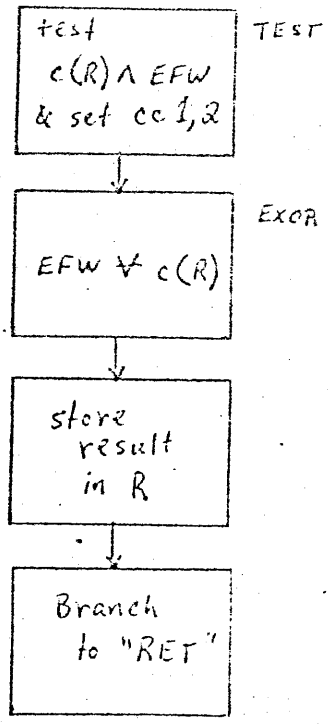


CHART 17

PUC

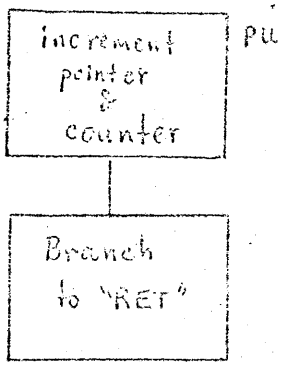


CHART 18

POC

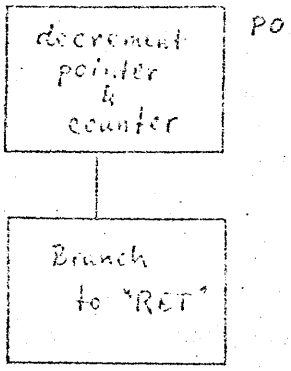


CHART 19

PUSH

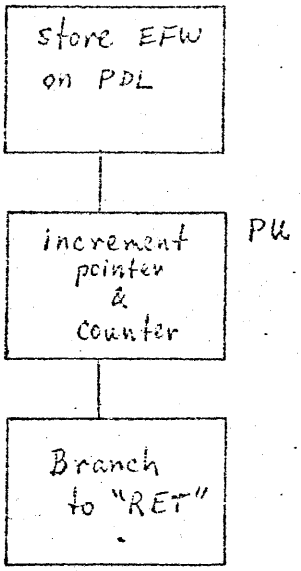


CHART 20

POP

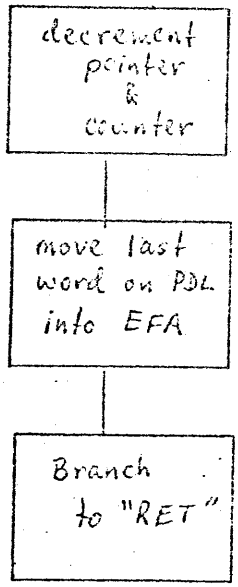


CHART 21

PUB

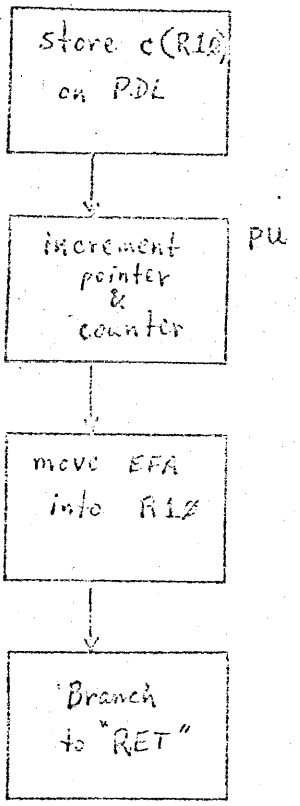


CHART 22

POB

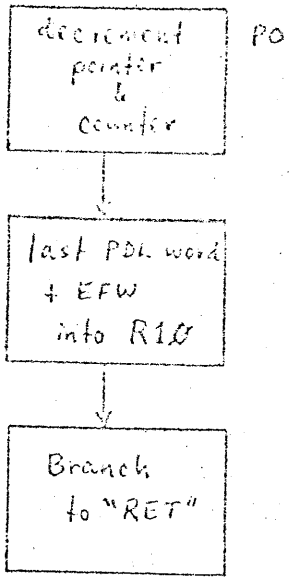


CHART 23

PUL

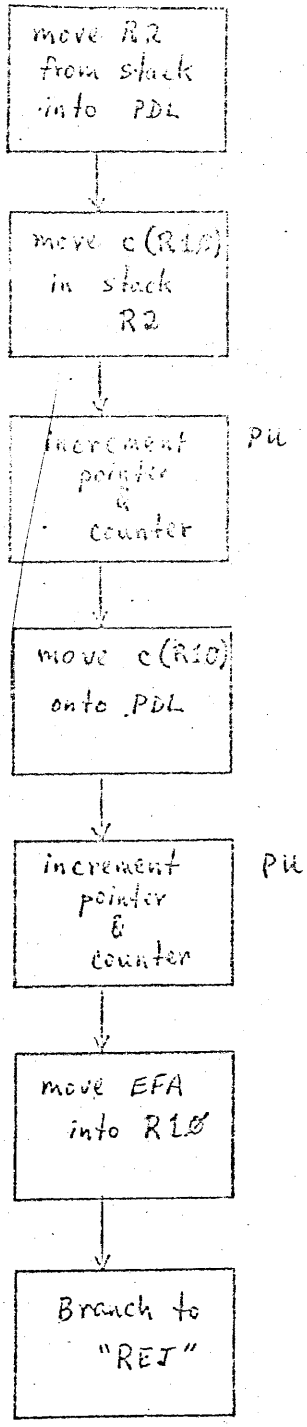


CHART 24

POL

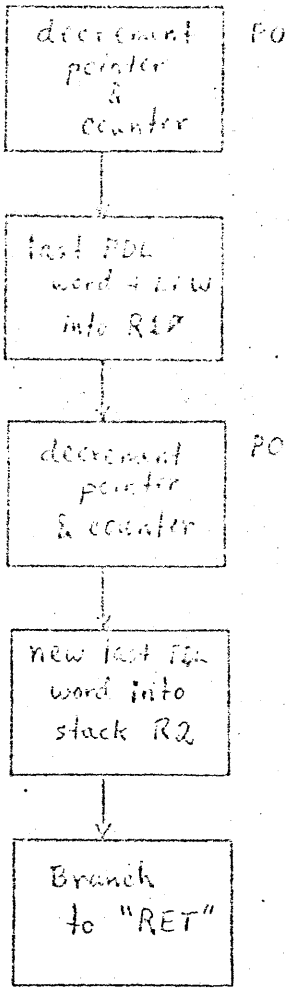


CHART 25

CC

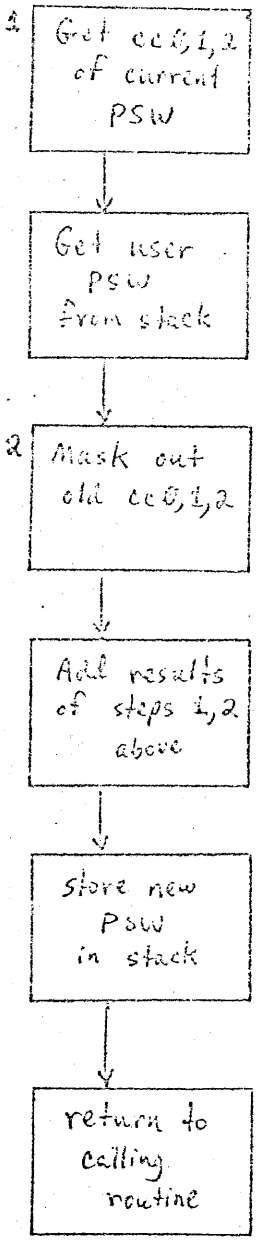


CHART 26

CC12

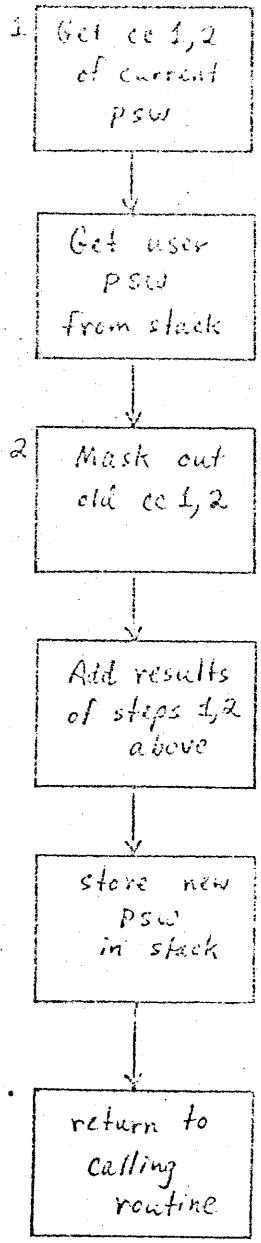


CHART 27

INOR

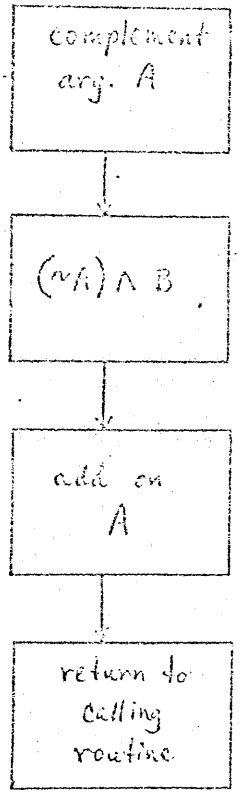


CHART 28

EXOR

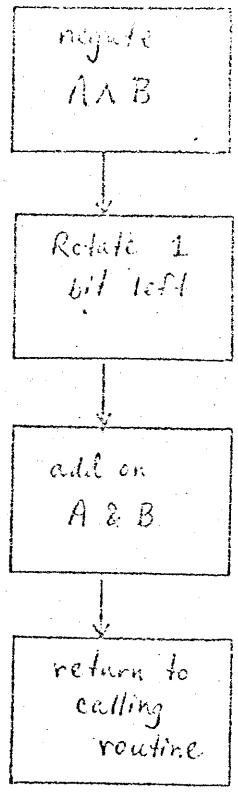


CHART 29

IAOS

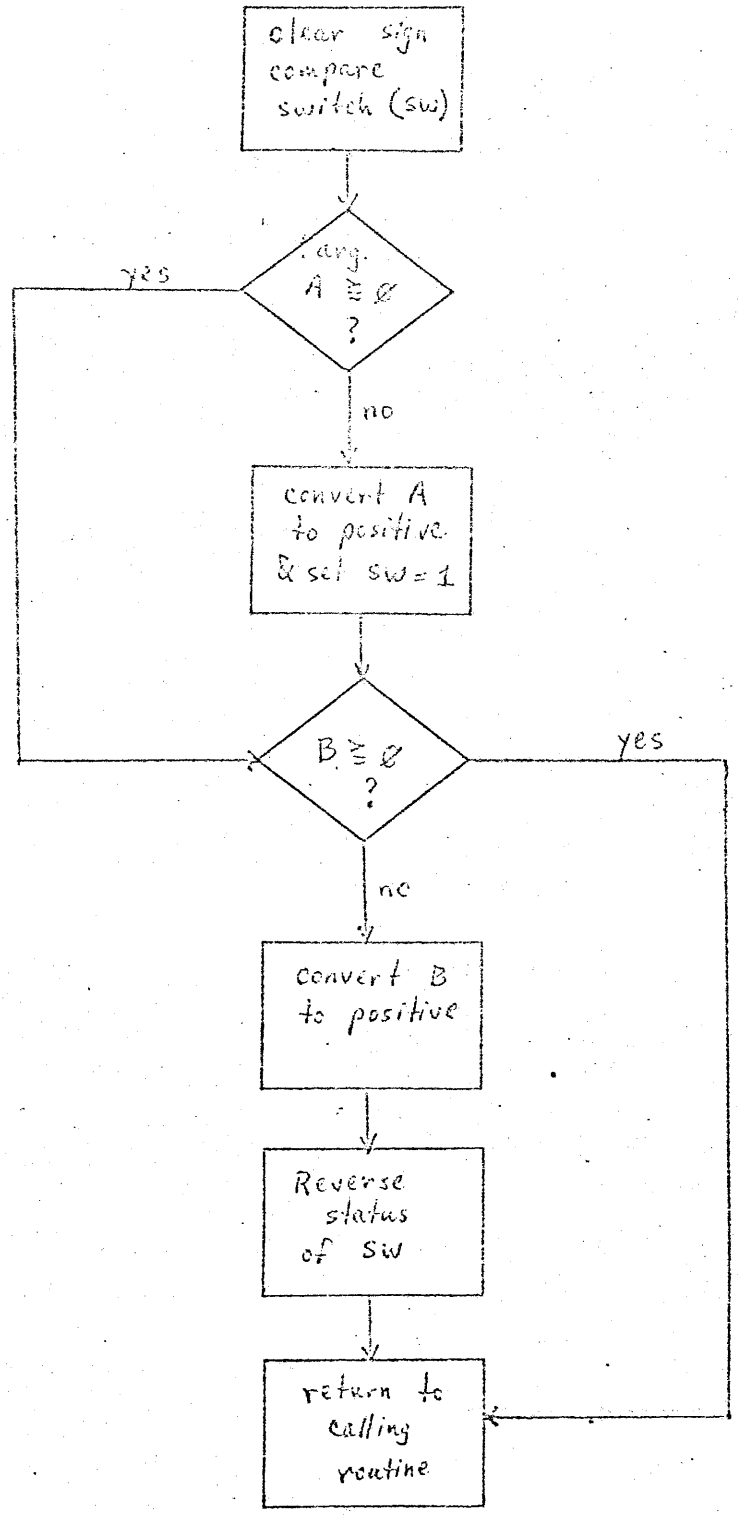


CHART 30

LMULT

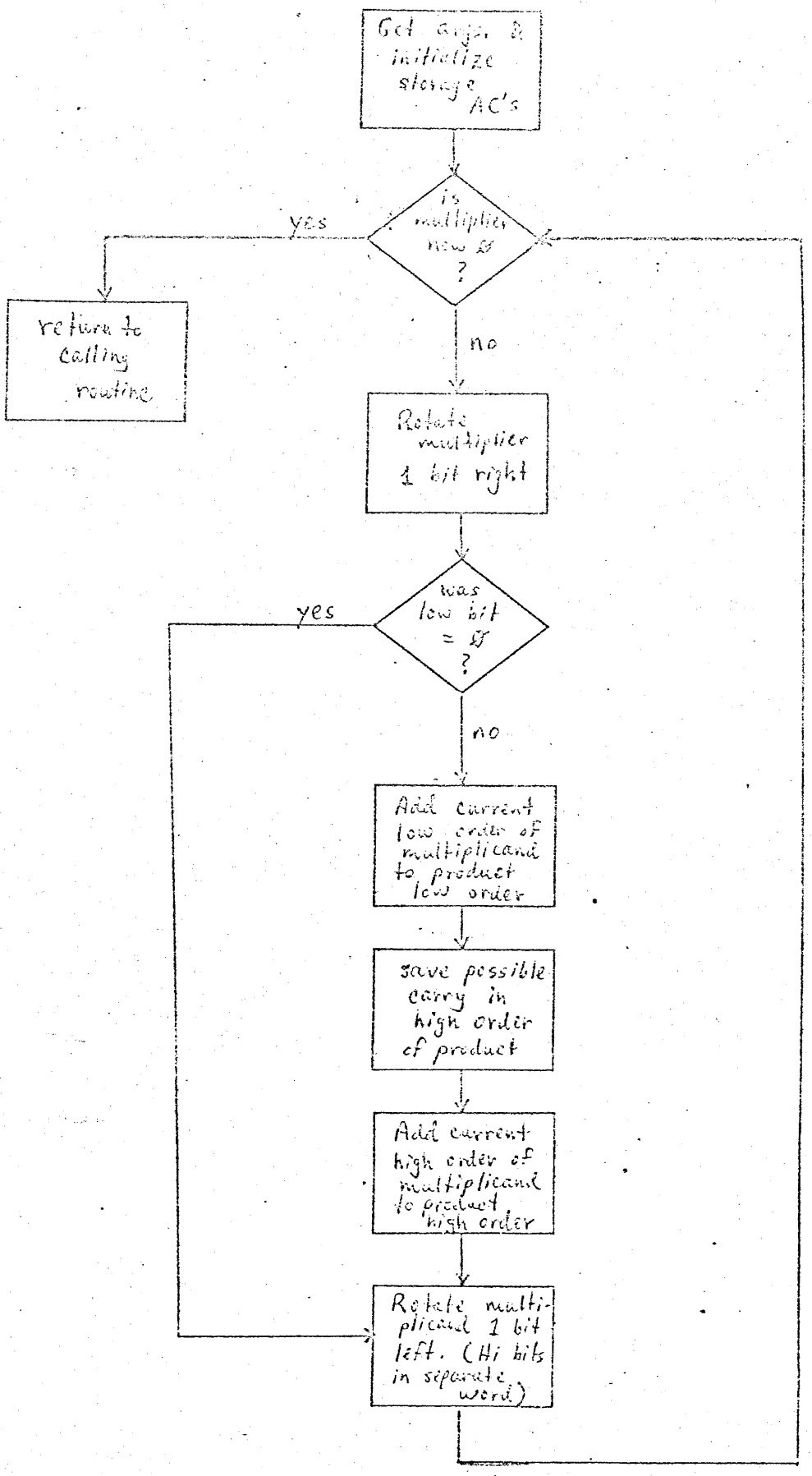


CHART 31

IMULT

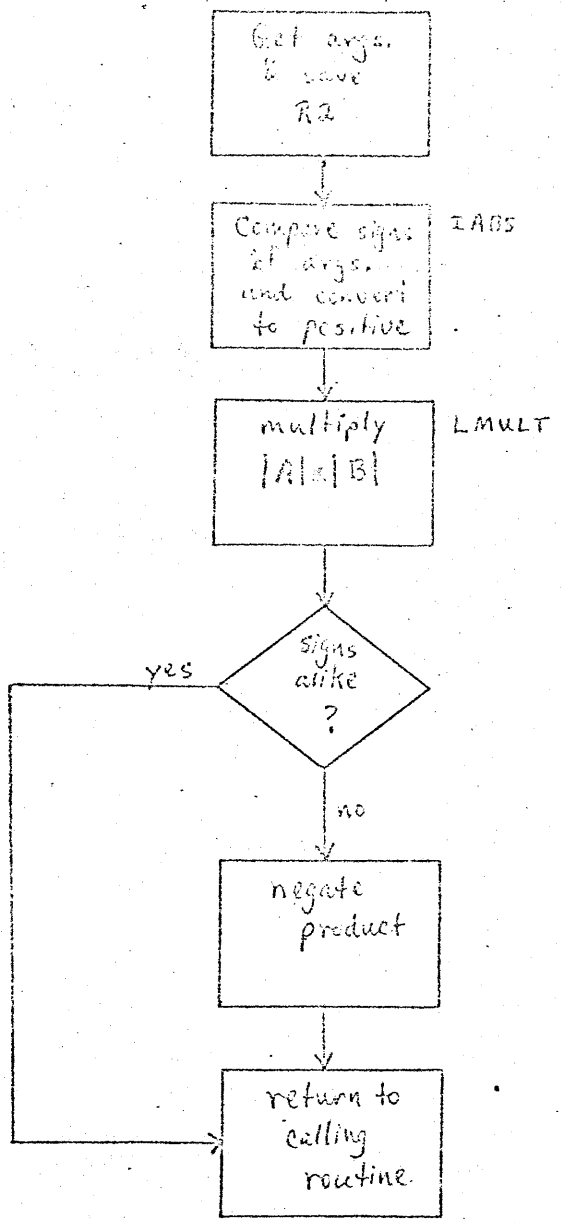


CHART 32

LDIVD

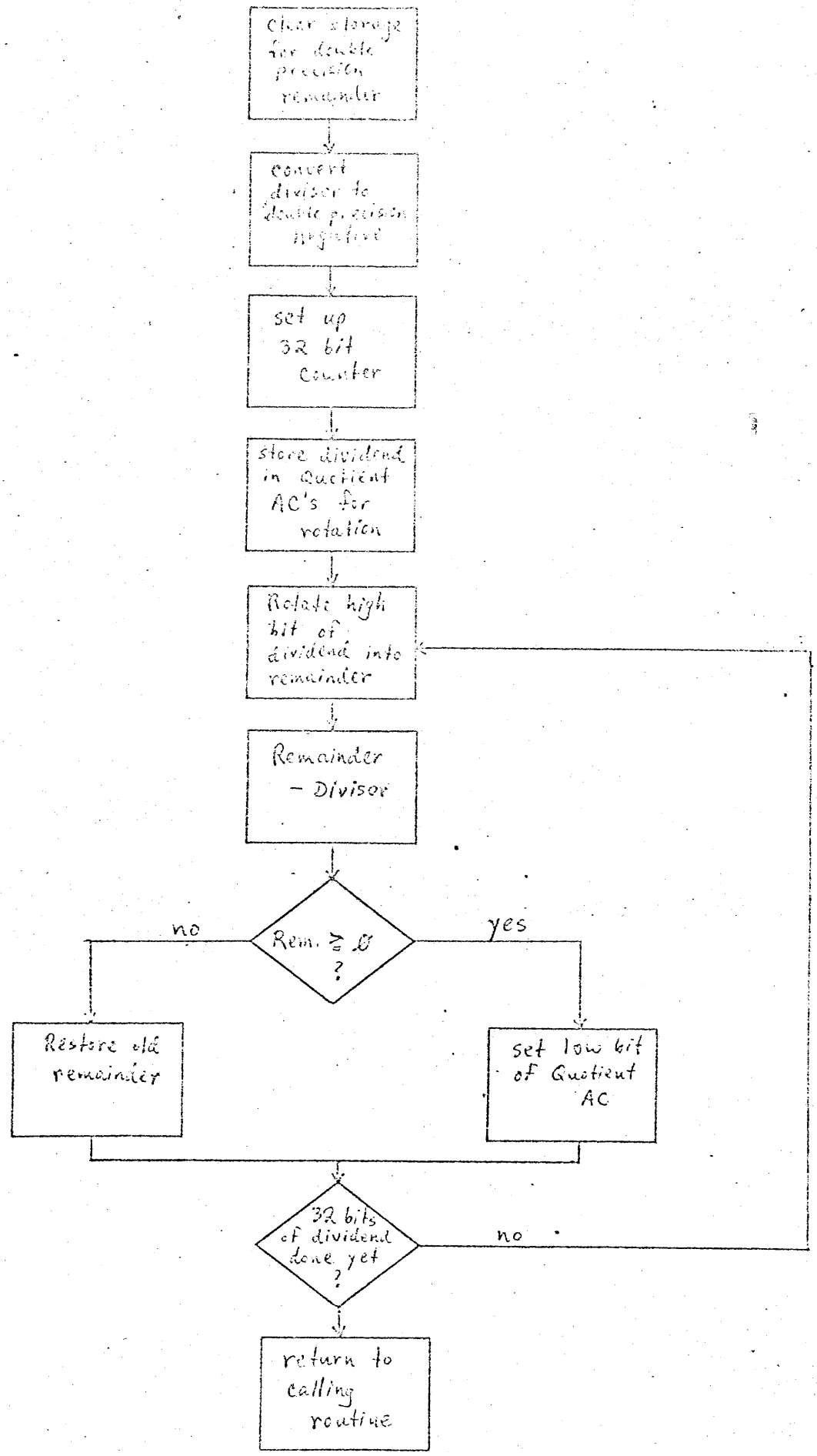


CHART 33

IDIVD

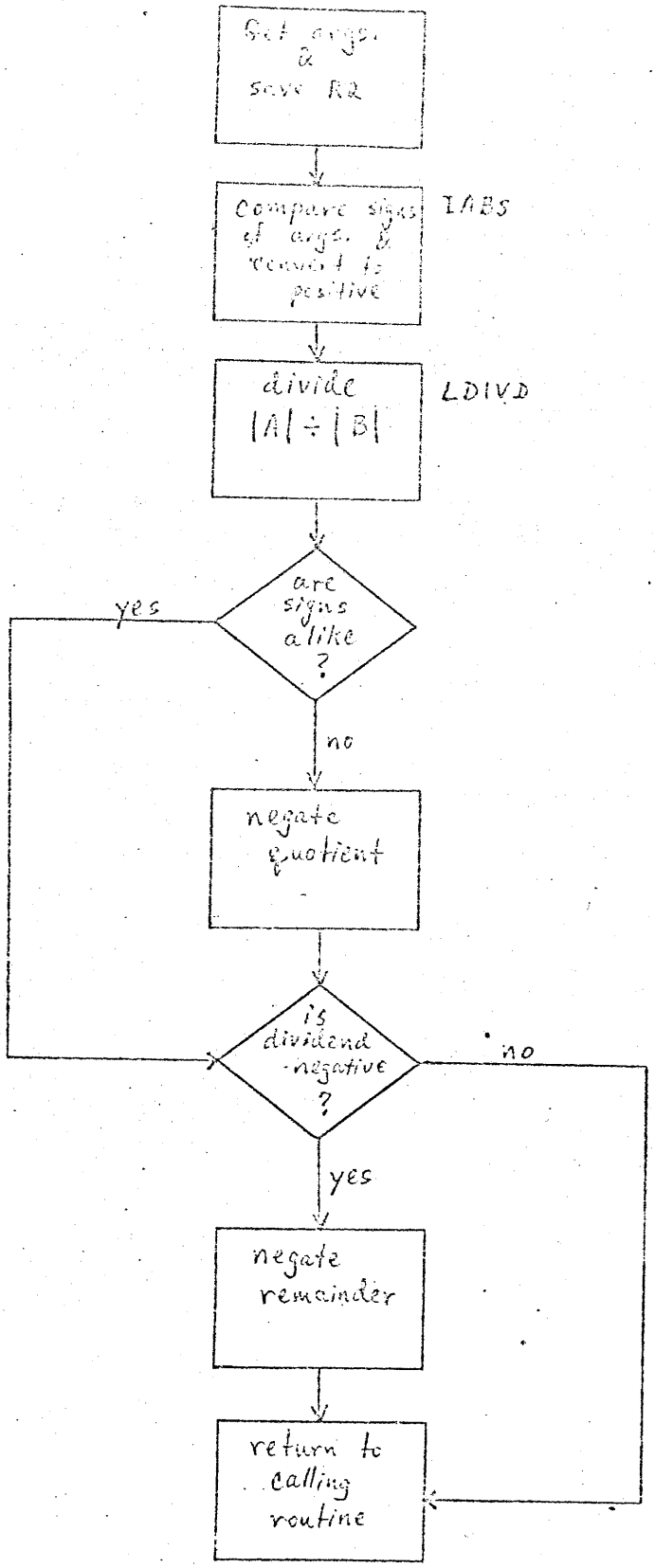


CHART 34

AETR

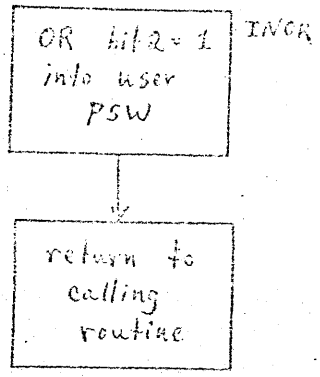


CHART 35

TEST

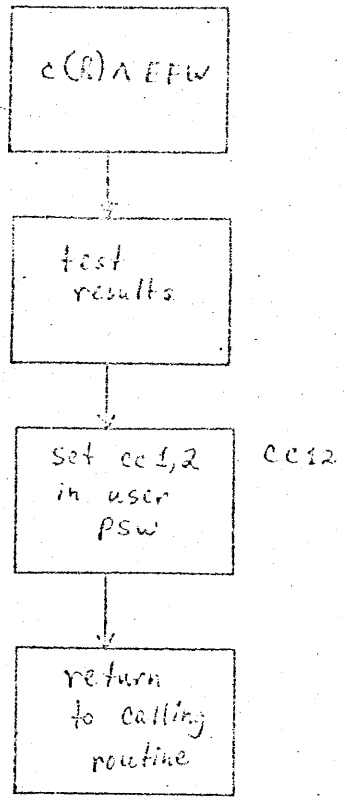


CHART 36

PW

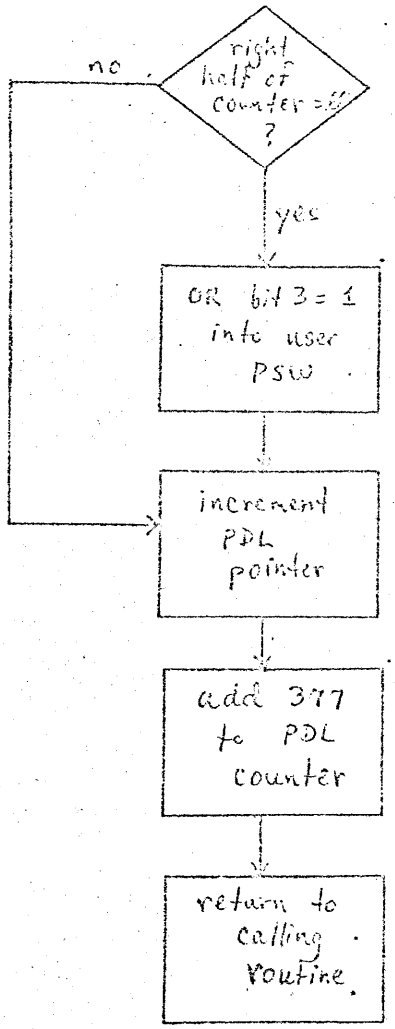


CHART 37

PO