1-1-1975

The Lehigh University IBM 360 simulator.

Leonard Ira Horey

THE LEHIGH UNIVERSITY IBM 360 SIMULATOR

by

Leonard Ira Horey

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

1975

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

April 3, 1975
(date)

Professor in Charge

Chairman of Department

Acknowledgment

The author would like to express his gratitude to his advisor, Professor Peggy A. Ota. Her comments and suggestions were a valuable aid in preparing this thesis.

Table of Contents

List of Figures

Contents of the Flowchart

Abstract

The Lehigh University IBM 360 Simulator (LUIS) is an interactive program which enables a user to execute an IBM 360 machine language program on a Control Data 6400 Computer System. LUIS simulates all but seven of the instructions in the IBM System/360 standard instruction set.

The simulator provides a simulated ten thousand byte memory whose limits are initially specified by the user. The user can load his IBM 360 machine language program into this memory either by entering the program from the remote terminal or by reading the program from a local file. He may then request that either all or a specific number of instructions in the program be executed. If the simulator encounters a situation which would normally interrupt a real IBM 360, it terminates execution of the user's program and issues an error message explaining the reason for interruption. The user can ask to examine the contents of any portion of the simulated IBM 360 memory and to examine the contents of the simulated IBM 360 general registers and the program status word. He can also modify his machine language program and store the modified program on a file for future use.

I Introduction

1.1 Background

This thesis describes an interactive IBM System/360 simulator written for a Control Data 6400 Computer System. The simulator was written to allow Lehigh University students to familiarize themselves with the IBM System/360 family of machines. These students would otherwise be unable to do so in the Lehigh University computing environment.

In particular, the Department of Electrical Engineering offers a senior level elective in systems programming, EE 315. The text currently being used in the course is Systems Programming by John J. Donovan. Donovan (like many other authors) uses the IBM System/360 for his examples. Lehigh University has only a Control Data 6400 Computer System. It would be beneficial to have a set of programs which would enable Lehigh students, those taking EE 315 and others, to write programs in IBM System/360 Basic Assembly Language and have them assembled and executed on a simulated IBM System/360. A first program would assemble the student's program on the Control Data 6400 Computer System and produce a machine language version of the program. A second program would then take this machine language program and execute it.

This thesis describes the second program, the Lehigh University IBM 360 Simulator (LUIS). LUIS is an interactive program which simulates all but seven of the instructions in the IBM System/360 standard instruction set[1] (Diagnose, Set System Mask, Load PSW, Halt I/O, Supervisor Call, Test Channel, and Test I/O). The program is available through the Lehigh University Computing Center.

The simulated IBM 360 has a ten thousand byte memory (which is stored in the NW array in the simulator) and sixteen general registers (which are stored in the "STATUS" common block). The simulated IBM 360 also has a program status word (PSW) which is stored in the "STATUS" common block. The PSW (Fig. 1.1) contains information needed to execute the user's program. Subroutine FIBM acts as the central processing unit for the simulated IBM 360. (Fig. 1.2)

Since all user input and output must be done through the Control Data 6400 Computer System and the simulator, the input/output instructions Test Channel, Test I/O, and Halt I/O are not used. Instead all input/output is done by using a modified version of the Start I/O instruction. Since input/output is not performed in the usual manner, the system mask, which is concerned with input/output interrupts, is not relevant and thus the Set System Mask instruction is omitted. The user initially specifies the PSW and can change it through the use of

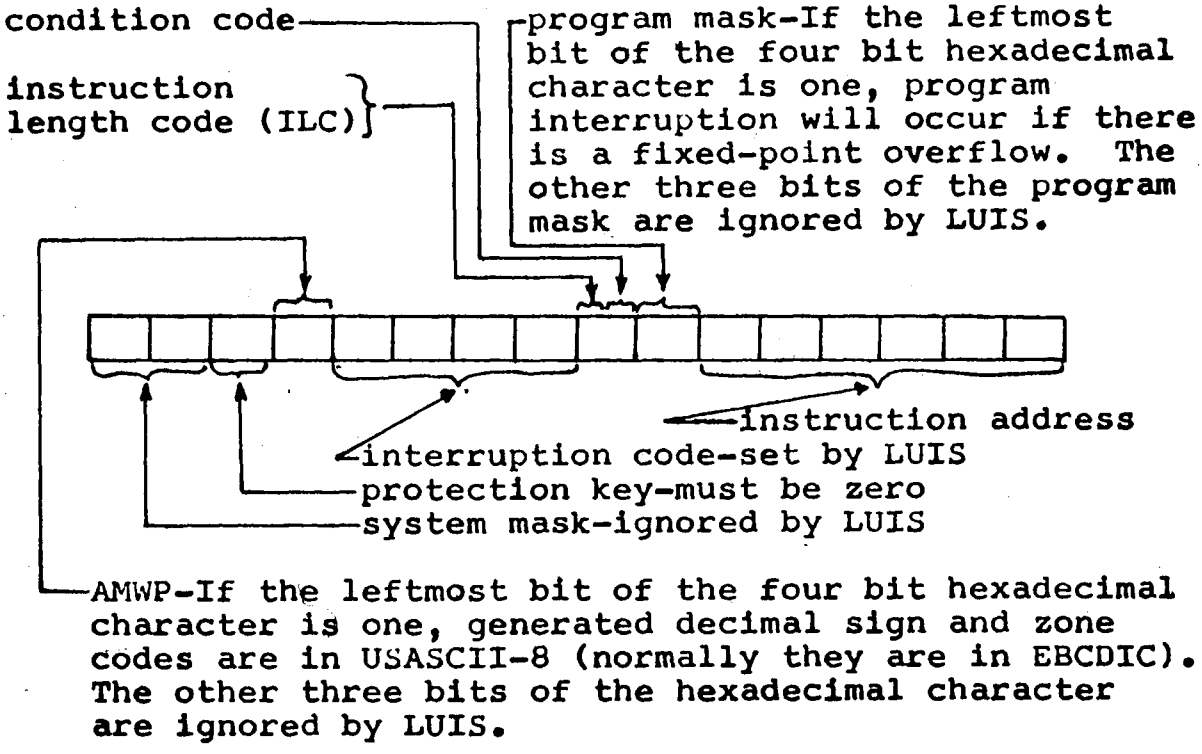


Fig. 1.1 The Program Status Word
(Each block represents one four bit hexadecimal character.)

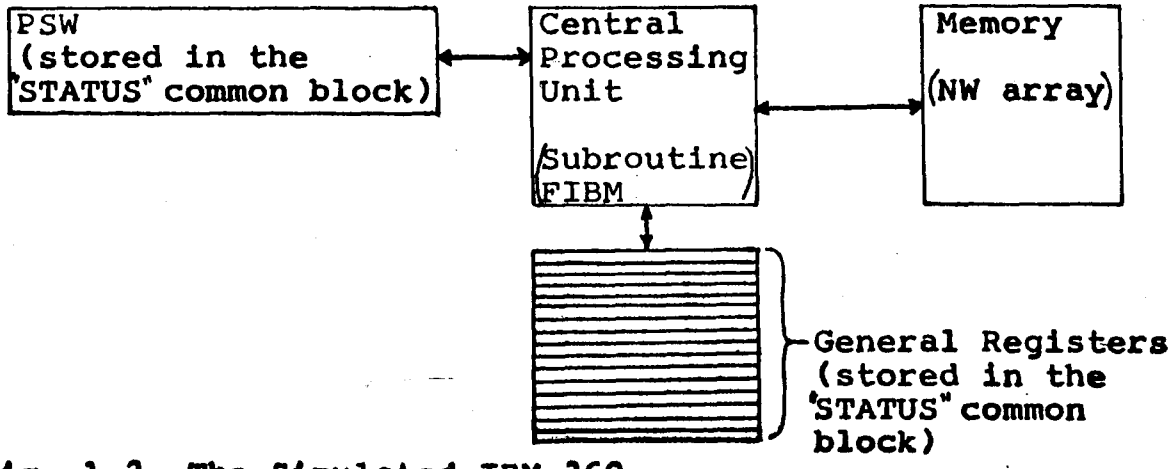


Fig. 1.2 The Simulated IBM 360

the request NEWPSW (see LUIS User's Guide), so the Load PSW instruction is not needed. Obviously, since one is not working with a real IBM System/360, the Diagnose instruction (which is used for testing the system's hardware) has no meaning and is not included. Because all input/output is done through the simulator, and because none of the other privileged operations are included, there is no need for a Supervisor Call and this instruction has also been omitted.

1.2 Contents of the Thesis

The remaining portion of this thesis is divided into five chapters and two appendices. Chapter II is a user's guide to using the Lehigh University IBM 360 Simulator. Chapter III gives a short description of each of the ten routines which comprise the simulator. Chapter IV gives a more detailed description of the subroutine (FIBM) which actually simulates the IBM 360. Chapter V contains the flowchart of LUIS. The last chapter, Chapter VI, describes how additional capabilites can be added to LUIS. Appendix A contains material (reproduced from the manual, IBM System/360 Principles of Operation) which should be of value to those who are unfamiliar with the IBM 360 structure. Appendix B contains a typical output from LUIS.

II User's Guide to LUIS

2.1 Introduction

The Lehigh University IBM 360 Simulator (LUIS) takes a program written in IBM System/360 machine language and executes it on a Control Data 6400 Computer System*. LUIS simulates all but seven of the instructions in the IBM System/360 standard instruction set. (The instructions Diagnose, Set System Mask, Load PSW, Halt I/O, Supervisor Call, Test Channel, and Test I/O are omitted.) This chapter describes the features and use of LUIS. Section 2 discusses a restriction on the addresses used with LUIS. Section 3 describes the commands of LUIS. The last section, Section 4, explains how to use the simulator.

*LUIS operates under INTERCOM. INTERCOM is a subsystem which allows the user to run an interactive program from a remote terminal on the Control Data 6400 Computer System.

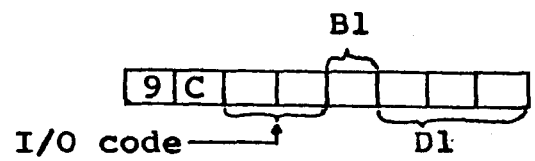
2.2 Restricted Addresses

One important difference between the simulator and the IBM System/360 is that addresses used with LUIS are restricted to a maximum of seventeen significant bits. If the eighteenth bit is a one, the address will be interpreted as a negative number and the program will be terminated if and when that address is actually used for addressing. If more than eighteen significant bits are used, the address will be truncated when it is used for addressing. If the truncated address is within the memory area of the program, the program will continue to execute normally. However, if the resulting address is outside the memory area of the program, the simulator will issue an error message and terminate execution of the program. This restriction on the length of addresses is necessitated by the fact that the A and B registers in the CDC 6400 Computer System are only eighteen bits long. (One should note the fact that if an address field is not actually used for addressing storage such as in a "Load Address" instruction or shifting instructions, then this limitation does not apply.)

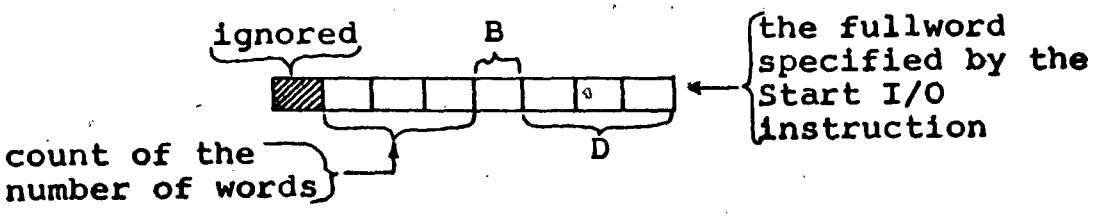
2.3 The Instruction Set

All of the simulated instructions (except Start I/O) function as described in the manual, IBM System/360 Principles of Operation [1]. The floating-point feature instructions, the decimal feature instructions, the protection feature instructions, and the direct control feature instructions are not available. Conditions which would normally produce an interrupt in an IBM 360 cause the simulator to terminate execution of the program and print an error message indicating the reason for the interrupt.

The user's program can perform I/O by using a modified version of the Start I/O instruction.



The I/O code in the instruction indicates the type of I/O that is to be performed by the program. The code is 01 for writing and 00 for reading. (All other codes are invalid.) The address specified by B1 and D1 is a fullword somewhere in the user's (memory) storage area. This fullword contains the count of the number of words which the program will read or write and a base register and displacement which specify the starting address for I/O. The starting address must specify a fullword boundary.



When the I/O code specifies writing, the simulator responds:

WORDS PRINTED FROM MEMORY BY THE USER'S PROGRAM

The simulator then prints the words and their corresponding addresses. When the I/O code specifies reading, the simulator responds:

YOUR PROGRAM WANTS TO READ SOME WORDS.
IF YOU HAVE THE WORDS ON THE FILE PGM, TYPE PGM.
IF YOU ARE GOING TO ENTER THE WORDS, TYPE INPUT.

Words are read in the same manner as described for "INSERT". If an error occurs during reading, the simulator issues an error message and terminates execution of the program. If the reading of the words is completed in a satisfactory manner the simulator responds:

WORDS READ INTO MEMORY BY THE USER'S PROGRAM

The simulator then prints the words which it read and the locations where each was placed.

If no errors were detected while performing I/O, then the execution of the program continues after I/O is completed.

2.4 Use of the Simulator

The simulator is an interactive program. To use the simulator the user first attaches the simulator while in the INTERCOM command mode. If the user has his program on a permanent file, then he must also attach this file (with the local file name of PGM). After the user attaches the simulator, he types:

LUIS.

At this point the user is in the simulator program. The simulator responds:

LEHIGH UNIVERSITY IBM 360 SIMULATOR

ENTER THE SMALLEST ADDRESS IN YOUR PROGRAM AS AN EIGHT DIGIT HEXADECIMAL NUMBER.

The user enters the smallest address in his program. This address determines one of the boundaries of his storage area. The address can be between 00000000 and 0001D8A8 hexadecimal and must specify a fullword boundary. If the address is not acceptable to the simulator, it responds with an error message and repeats its request. If the address is acceptable the simulator responds:

ENTER THE MAXIMUM SIZE OF YOUR PROGRAM (IN BYTES) AS A FOUR DIGIT DECIMAL INTEGER.

The user enters the number of bytes in his storage area. This number, together with the smallest address, determines the boundaries of the user's program. The simulator will prevent the user from exceeding the boundaries of his storage area. If the number which the user enters is

not acceptable to the simulator, it responds with an error message, and repeats its request. If the number is acceptable, the simulator responds:

ENTER THE PSW AS A 16 DIGIT HEXADECIMAL NUMBER.

The user enters his program's PSW. If the PSW is not acceptable, an error message will be issued and the request will be repeated. If the PSW is acceptable the simulator responds:

REQUEST=

At this point the user is in the simulator request mode and can issue any of the following requests:

END
BYE
INSERT
NEWPSW
DUMP
STATUS
S
EXECUTE
REWINDP
STORE
REWINDS

These requests are explained below.

END

The request END enables the user to redefine his storage area and start a new program without leaving the simulator program. The simulator responds:

PROGRAM ENDED BY USER

The simulator then requests the information needed to define the storage area (the smallest address in the

program, the maximum size of the program, and the PSW). One should note that the simulator zeros the program's storage area when the storage area is defined by the user. Thus any bytes which were in the storage area from a previous program are wiped out.

BYE

The request BYE terminates the simulator program and returns the user to the INTERCOM command mode. INTERCOM will respond:

```
EXIT
COMMAND-
```

The request BYE may also be called after the simulator responds:

```
ENTER THE SMALLEST ADDRESS IN YOUR PROGRAM AS AN
EIGHT DIGIT HEXADECIMAL NUMBER.
```

INSERT

The request INSERT tells the simulator that the user wants to insert some bytes into his storage area. The simulator responds:

```
ENTER THE STARTING ADDRESS (OF THE COLLECTION OF
BYTES) AS AN EIGHT DIGIT HEXADECIMAL NUMBER.
```

The address must specify a fullword boundary. If the address which the user enters is not acceptable, the simulator issues an error message and repeats its request. If the address is acceptable the simulator responds:

```
ENTER THE NUMBER OF BYTES WHICH WILL BE INSERTED
AS A FOUR DIGIT DECIMAL INTEGER.
```

If the number is not acceptable the simulator issues an error message and repeats its request. If the number is acceptable the simulator responds:

```
IF YOU HAVE THE BYTES ON THE FILE PGM, TYPE PGM.
IF YOU ARE GOING TO ENTER THE BYTES, TYPE INPUT.
```

If the user types "INPUT" the simulator responds:

```
ENTER THE BYTES.  FOUR BYTES PER LINE.
```

and prints the location where the bytes will be inserted. When the user enters the bytes, the simulator responds by printing the location where the next group of bytes will be inserted. This process continues until the user finishes entering all of the bytes. After the simulator reads all of the bytes, it responds:

```
THE BYTES HAVE BEEN READ
```

and returns the user to the request mode.

If the user types "PGM" the simulator assumes that the bytes are on the local file PGM. If the files does not exist the simulator will respond:

```
PGM DOES NOT EXIST
```

and will return the user to the request mode. The simulator assumes that the file PGM has the bytes packed forty bytes per card image. The simulator reads as many cards as are necessary to satisfy the user's INSERT request. If a second INSERT request causes the simulator to read from PGM a second time, the simulator will begin reading at the next card image. If the simulator reads the EOF because PGM does not contain enough bytes, the

simulator issues an error message and zeros all locations mentioned in the INSERT. After the simulator reads all of the bytes, it responds:

THE BYTES HAVE BEEN READ

and returns the user to the request mode.

NEWPSW

The request NEWPSW enables the user to change his program's PSW. The simulator responds:

ENTER THE PSW AS A 16 DIGIT HEXADECIMAL NUMBER.

After the user enters an acceptable PSW, the simulator returns the user to the request mode.

DUMP

The request DUMP enables the user to dump all or a portion of his memory (storage area). The simulator responds:

TYPE ALL OR PARTIAL

If the user types PARTIAL, the simulator asks the user to supply the starting address of the dump and the length of the dump in bytes. The dump must start on a fullword boundary. The user is not permitted to dump outside his storage area.

Insertions and dumps can only be done with fullword units. Thus all requests are rounded up to the nearest number of fullwords. This rounding process may cause a request to exceed the user's memory area by a fraction of

a word. However, this fact will be ignored by the simulator and will not cause any problem.

After the simulator performs the requested dump, it returns the user to the request mode.

STATUS

The request STATUS causes the simulator to print the PSW, the instruction address, the ILC, the condition code, and the contents of all of the general registers. The simulator then returns the user to the request mode.

S

The request S enables the user to request that a specified number of instructions be executed in his program (starting with the one specified by the instruction address portion of the PSW). The simulator responds:

> ENTER THE NUMBER OF INSTRUCTIONS TO BE EXECUTED AS A FOUR DIGIT DECIMAL INTEGER.

After the user enters the number, the simulator executes the specified number of instructions and then returns the user to the request mode.

It is possible for the simulator to return to the request mode before it finishes executing all of the specified instructions. This happens when the simulator encounters a condition which would produce an interrupt in a real IBM 360. In this case the execution of the program stops, and the simulator prints an error message

indicating the reason for program termination.

EXECUTE

The request EXECUTE enables the user to execute his entire program. EXECUTE tells the simulator to execute 9,999 instructions. Usually this will be much greater than the number of instructions in the user's program. Thus the user's program will end when a condition occurs which would produce an interrupt in a real IBM 360. However, if the user's program is either longer than 9,999 instructions or has an endless loop in it, the simulator will return to the request mode after it executes 9,999 instructions. The user can recognize when this happens because no interrupt message occurs before the simulator returns to the request mode.

REWINDP

The request REWINDP enables the user to rewind the file PGM. The user must do this if he has previously read from PGM, and then wants to start at the beginning of the file. The simulator rewinds the file and returns the user to the request mode. If the file does not exist the simulator will respond:

PGM DOES NOT EXIST

and will return the user to the request mode.

STORE

The request STORE enables the user to dump all or a portion of his memory (storage area) onto a file named STORE. In this manner the user can save the contents of his memory for future use. The simulator responds:

TYPE ALL OR PARTIAL

If the user types PARTIAL, the simulator asks the user to supply the starting address of the dump and the length of the dump in bytes. The request STORE has the same restrictions as the request DUMP (See DUMP). The bytes are written forty bytes per card image on the file STORE. If a second STORE request causes the simulator to write on file STORE a second time, the simulator will begin writing at the next card image. The simulator rewinds file STORE when the user initially executes the simulator program. File STORE is not destroyed when the user terminates the simulator program. Thus the file can be saved by the user and used as the file PGM at some later time.

After the simulator dumps the bytes onto file STORE, it responds:

THE BYTES HAVE BEEN DUMPED ONTO FILE STORE

and returns the user to the request mode.

REWINDS

The request REWINDS enables the user to rewind the file STORE. The user must do this if he has previously written on STORE, and then wants to start writing at the beginning of the file. The simulator rewinds the file and returns the user to the request mode.

III The Routines of LUIS

3.1 Introduction

The Lehigh University IBM 360 Simulator is written in COMPASS [2,3] (assembly language) and FORTRAN. The program consists of a main routine and nine subroutines. The COMPASS subroutines assume that the FORTRAN routines are compiled by using the RUN compiler. Approximately 23.6 CP seconds are needed to compile and assemble the program. 31242_{octal} words of central memory are required by the program and the various system routines which it calls. Naturally, additional central memory is needed for the loader and the loader tables. Section 2 gives a general description of each of the routines of LUIS.

3.2 Description of the Routines

3.2.1 The Main Routine

The main routine is written in FORTRAN. All of the communications between the user and the simulator (except some error messages and program I/O) are handled by this routine. The information needed to define the boundaries of the user's memory area is initially requested by the main routine and all user requests are processed through it. Checks are made to insure that no insertions or dumps are performed outside of the user's memory area. If any invalid requests or otherwise erroneous input are received, an appropriate error message is printed, and the request for the input is repeated.
(See flowchart pp. 36 - 43)

3.2.2 Subroutine BOMB

Subroutine BOMB is written in FORTRAN. This subroutine prints error messages. The calling routine passes one parameter to BOMB. This parameter determines which of twenty error messages is to be printed. BOMB also sets a flag if either of two particular error messages are printed. This flag is passed to the main routine through the "GOOD" common block (Fig. 3.1).
(See flowchart p.44)

Fig. 3.1 The "GOOD" common block

```
PROGRAM STATUS WORD     0000000798000048


INSTRUCTION ADDRESS     00048
CONDITION CODE     1                        ILC     2

                          GENERAL REGISTERS
0  00000000     1  000000A0     2  00000010     3  00000501
4  00000000     5  00000000     6  00000000     7  00000000
8  00000000     9  00000000     A  00000000     B  00000000
C  00000000     D  00000000     E  00000000     F  00000000
```

Fig. 3.2 A typical output requested by the user through the use of the request; STATUS. (Output printed by subroutine DUMP; entry point PREDUMP)

3.2.3 Subroutine DUMP

Subroutine DUMP is written in FORTRAN. The subroutine is actually composed of two separate parts. The first part (DUMP) prints the program status word, the instruction address, the condition code, the ILC, and the contents of each of the sixteen general registers (Fig. 3.2). This information is passed to DUMP through the "STATUS" common block (Fig. 3.3). The second part of the subroutine (entry point PREDUMP) prints the byte addresses and the contents at each address (in hexadecimal) (Fig. 3.4). The subroutine must know the byte address of the first word, the corresponding index in the NW array (where the words are actually stored), and the number of bytes which are to be dumped. This information is passed to DUMP through the "BYTES" common block (Fig. 3.5).
(See flowchart p.44)

3.2.4 Subroutine REED

Subroutine REED is written in FORTRAN. REED allows the user's program to read words into memory. Subroutine FIBM passes three parameters (the first byte address, the corresponding index in the NW array (where the user's program is stored), and the number of bytes which are to be read) to REED. The subroutine stores the parameters it receives in the "BYTES" common block and reads the words from the source indicated by the user. If the reading

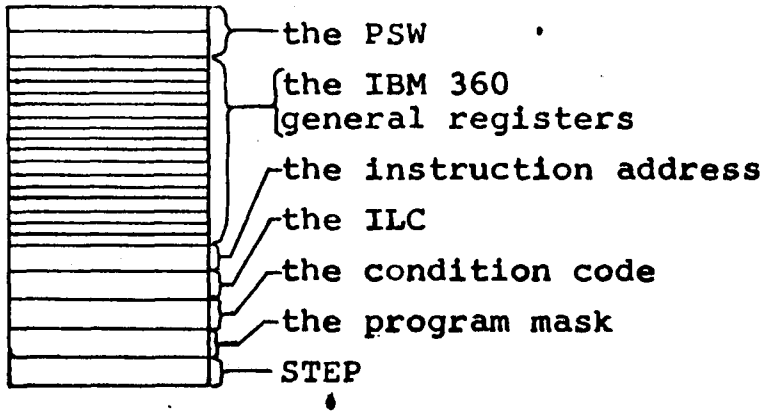


Fig. 3.3 The "STATUS" common block

00000	00020004	00004	00000000	00008	00000000	0000C	00010014
00010	00000000	00014	00000000	00018	00000000	0001C	00000000
00020	9C000000	00024	58100004	00028	58200008	0002C	8B100004
00030	8B200004	00034	50100014	00038	5020001C	0003C	960C0017
00040	960C001F	00044	4F100010	00048	4F200018	0004C	1A124E10
00050	00105830	00054	00148830	00058	00045030	0005C	00149C01
00060	000C1B55	00064	50500010	00068	50500018	0006C	47F00020
00070	00000000						
00074	00000000						

contents at that address

address

Fig. 3.4 A typical output requested by the user through the use of the request; DUMP. (Output printed by subroutine DUMP.)

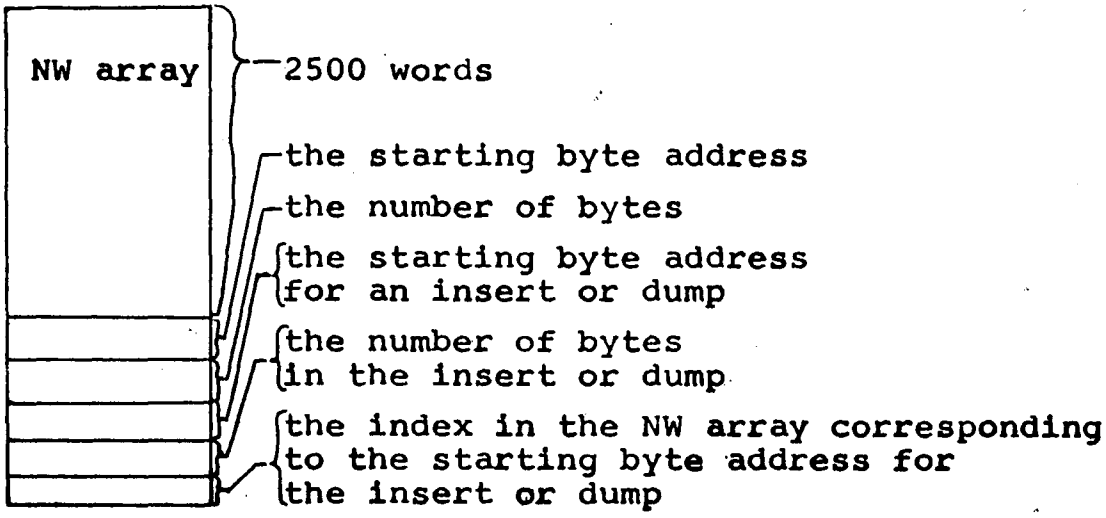


Fig. 3.5 The "BYTES" common block

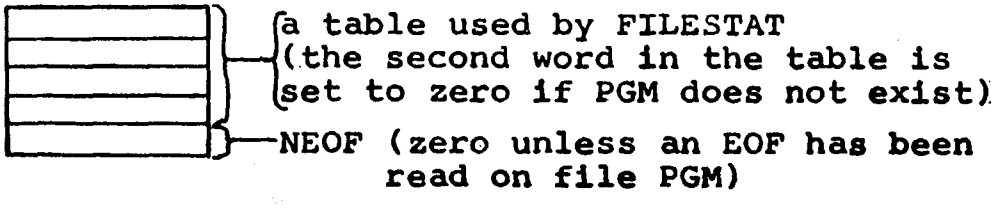


Fig. 3.6 The "IN" common block

process is not completed in a satisfactory manner, a parameter (the number of bytes which are to be read) is set to zero before returning to subroutine FIBM. (See flowchart p.45)

3.2.5 Subroutine RITE

Subroutine RITE is written in FORTRAN. This subroutine allows words to be printed from memory by the user's program. The subroutine is passed three parameters (the first byte address, the corresponding index in the NW array, and the number of bytes which are to be printed) by subroutine FIBM. Subroutine RITE stores the parameters it receives in the "BYTES" common block and then calls the routines needed to print the words in hexadecimal. (See flowchart p.46)

3.2.6 Subroutine EXIST

Subroutine EXIST is written COMPASS. The subroutine builds the necessary table and then calls the system macro [4], FILESTAT, to determine whether the file PGM exists. If PGM does not exist, the second word in the "IN" common block will be set to zero (Fig. 3.6). (See flowchart p.46)

3.2.7 Subroutine DEC

Subroutine DEC is written in COMPASS. This subroutine takes a word which contains four digits in

display code (right justified and zero filled) and converts the word to an integer. The word is passed to DEC as a parameter by the calling routine.
(See flowchart p.46)

3.2.8 Subroutine CONVERT

Subroutine CONVERT is written in COMPASS. Two parameters (the number of characters to be converted and the address of the first word which is to be converted) are passed to subroutine CONVERT by the calling routine. CONVERT assumes that each word contains eight hexadecimal characters in display code (right justified and zero filled). The subroutine replaces the eight characters in each word with their thirty-two bit binary equivalent (right justified and zero filled). (See flowchart p.46)

3.2.9 Subroutine HEX

Subroutine HEX is written in COMPASS. The calling routine passes two parameters (the number of words to be converted and the address of the first word which is to be converted). Hex assumes that each word contains a thirty-two bit binary number (right justified and zero filled). Subroutine HEX replaces each number with its equivalent eight hexadecimal characters. These characters are stored right justified (and zero filled) in display code. (See flowchart p.46)

3.2.10 Subroutine FIBM

Subroutine FIBM is written in COMPASS. This subroutine is the routine which actually simulates the IBM 360. Three parameters are passed to FIBM by the main routine. The parameters passed are the starting address of the user's program, the number of bytes in the user's program, and the starting address of the NW array. The "STATUS" common block contains additional information which is used by FIBM. One of the words in the "STATUS" common block specifies how many instructions are to be executed by FIBM. Subroutine FIBM will continue to execute instructions until the specified number have been executed or until a condition occurs which would produce an interrupt in a real IBM 360.

(See flowchart pp.47 - 103)

IV FIBM

4.1 Introduction

Subroutine FIBM is the heart of the Lehigh University IBM 360 Simulator. FIBM is the routine which actually simulates the IBM 360. Section 2 describes the memory of the simulated IBM 360. Section 3 describes how the PSW and the general registers are stored. Section 4 describes how the user's program is executed by subroutine FIBM. Section 5 discusses an alternative structure for FIBM.

4.2 The Memory of the Simulated IBM 360

The memory of the simulated IBM 360 is a 2500 word array called NW. Each word in the array holds one IBM 360 fullword. The thirty-two bit IBM 360 fullword is stored right justified in the sixty bit CDC word. The leftmost twenty-eight bits of each word are zero. To locate a byte in memory, FIBM first removes the rightmost two bits of the byte address. Then it right shifts the address two places and adds the contents of register B7 to it. Register B7 contains a number which when added to the shifted byte address gives that address' actual location in the CDC 6400. While subroutine FIBM is executing the contents of register B7 remains fixed at that number. In this manner, FIBM is able to locate the word which contains the desired byte. Finally FIBM uses the rightmost two bits which were originally removed from the byte address to determine which of the four bytes in the word is desired. Register B6 contains the smallest byte address which the simulator is allowed to read or write. Register B2 contains one plus the largest byte address which the simulator is allowed to read or write. The contents of these two registers remains fixed while FIBM is executing. FIBM checks to make sure that every byte address falls within the range set by the contents of these two registers.

4.3 The Storage of the PSW and the General Registers

The PSW is stored in the "STATUS" common block. Subroutine FIBM extracts the instruction address, the condition code, the ILC, and the program mask from the PSW and stores them at separate locations in the "STATUS" common block when it is called by the main routine. The locations containing the condition code and the program mask may be examined and their contents may be altered during the execution of the user's program. The contents of the location containing the ILC is altered every time an instruction is fetched by FIBM. While subroutine FIBM is executing, the contents of register X1 contains the updated instruction address. Whenever FIBM terminates execution of the user's program, it stores the contents of register X1 in the location reserved for the instruction address in the "STATUS" common block. Then it puts the current value of the condition code, the ILC, the program mask, and the instruction address in the location reserved for the second half of the PSW in the "STATUS" common block.

Each of the sixteen IBM 360 general registers is stored in the "STATUS" common block. The contents of each register occupies the rightmost thirty-two bits of a sixty bit location.

4.4 The Execution of the User's Program

When FIBM is called by the main routine, the subroutine extracts the instruction address, the condition code, the ILC, and the program mask from the PSW and stores them at separate locations in the "STATUS" common block. FIBM checks the protection key portion of the PSW to make sure that it is zero and sets the contents of registers X1, B2, B6, and B7. Next the subroutine subtracts one from the contents of a location (called STEP) in the "STATUS" common block which specifies the number of instructions which are to be executed. If the resulting contents is not equal to zero, then the subroutine fetches the halfword specified by the contents of register X1 (instruction address). FIBM examines the leftmost two bits to determine whether the halfword is an RR instruction, part of an RX instruction, part of an RS or SI instruction, or part of an SS instruction. Then FIBM branches according to the type of instruction to one of four sections in the subroutine. In these sections the remaining portion of the instruction is fetched if necessary, the contents of register X1 is updated, and all of the information necessary for the execution of the instruction is extracted from the instruction. Any needed byte addresses are generated from the address components contained in the instruction. Finally the op code is examined and a branch is made to the section

of FIBM which actually performs the operation specified by the instruction. Once the operation has been performed, FIBM branches back to the section of the subroutine where one is subtracted from the contents of the location STEP in the "STATUS" common block. The process repeats itself until the contents of that location is zero, or unless a condition occurs which would produce an interrupt in a real IBM 360.

When the contents of location STEP is zero, FIBM terminates execution of the user's program, updates the PSW, and returns to the main routine. When a condition occurs which would produce an interrupt in a real IBM 360, FIBM terminates execution of the user's program, updates the PSW, return jumps to subroutine BOMB (to print the reason for interruption), and returns to the main routine.

4.5 An Alternative Structure for FIBM

One should note that it would be possible to determine the op code of the instruction when the halfword is initially fetched. The subroutine could then branch directly to the section which actually performed the operation specified by the instruction. Each section would fetch the remaining portion of the instruction if necessary, update the contents of register X1, extract all of the needed information from the instruction, and generate any needed byte addresses from the address components contained in the instruction. Undoubtedly this change in the subroutine would tend to decrease somewhat the time required to execute each instruction. However, the change would greatly increase the length of the subroutine and would not produce a significant difference in the time required to execute the user's program.

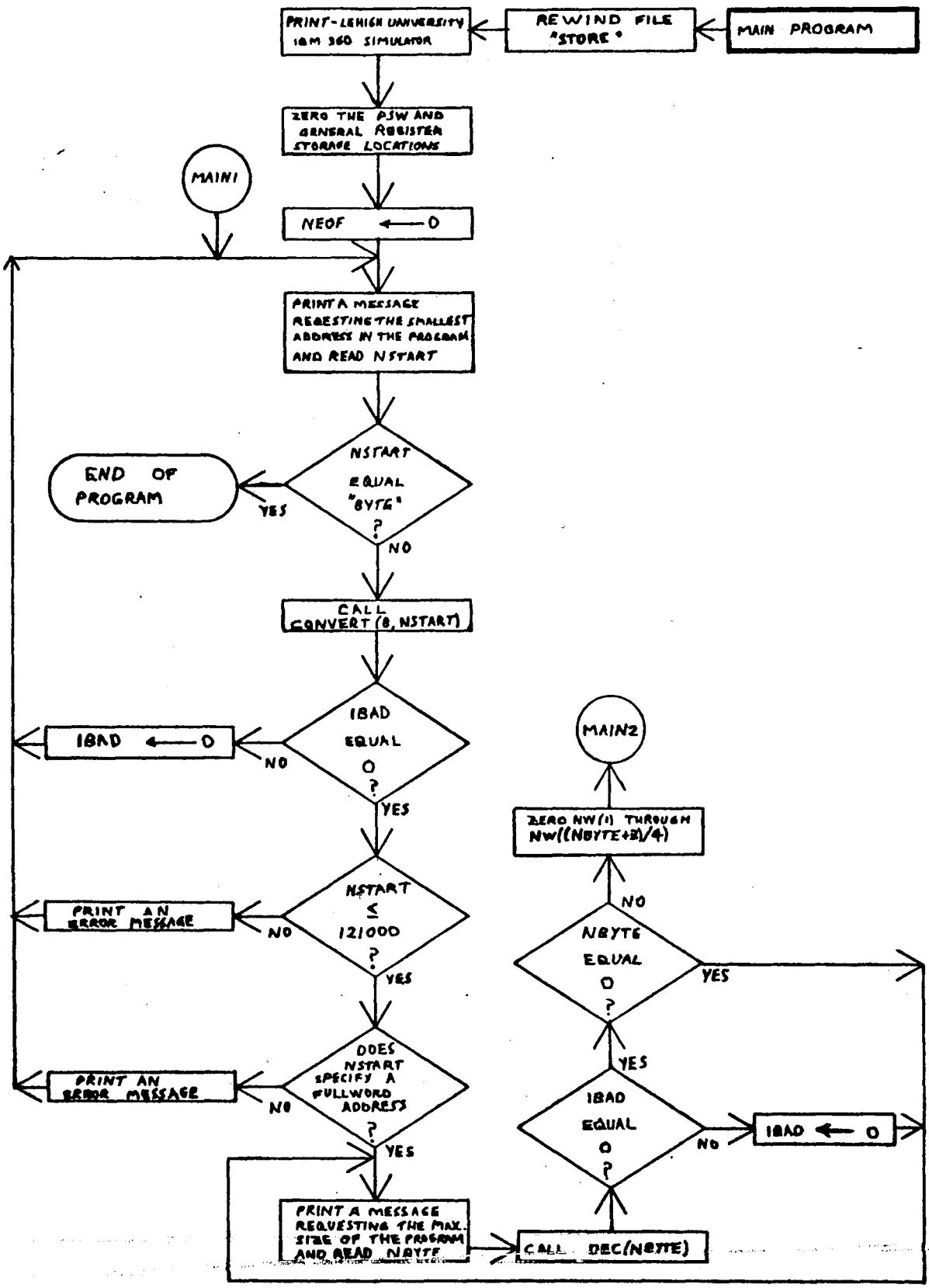
V The Flowchart of LUIS

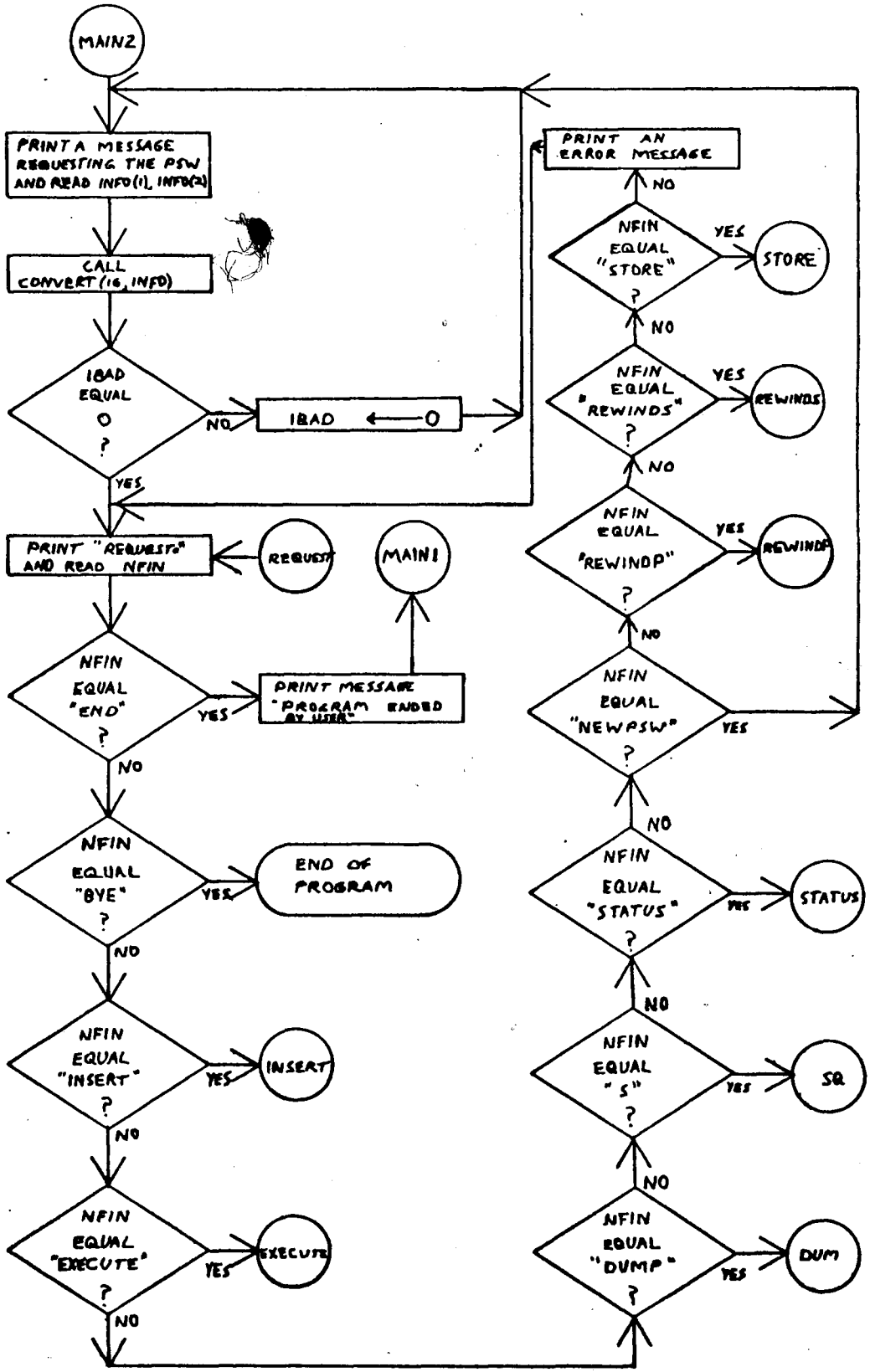
5.1 Notation

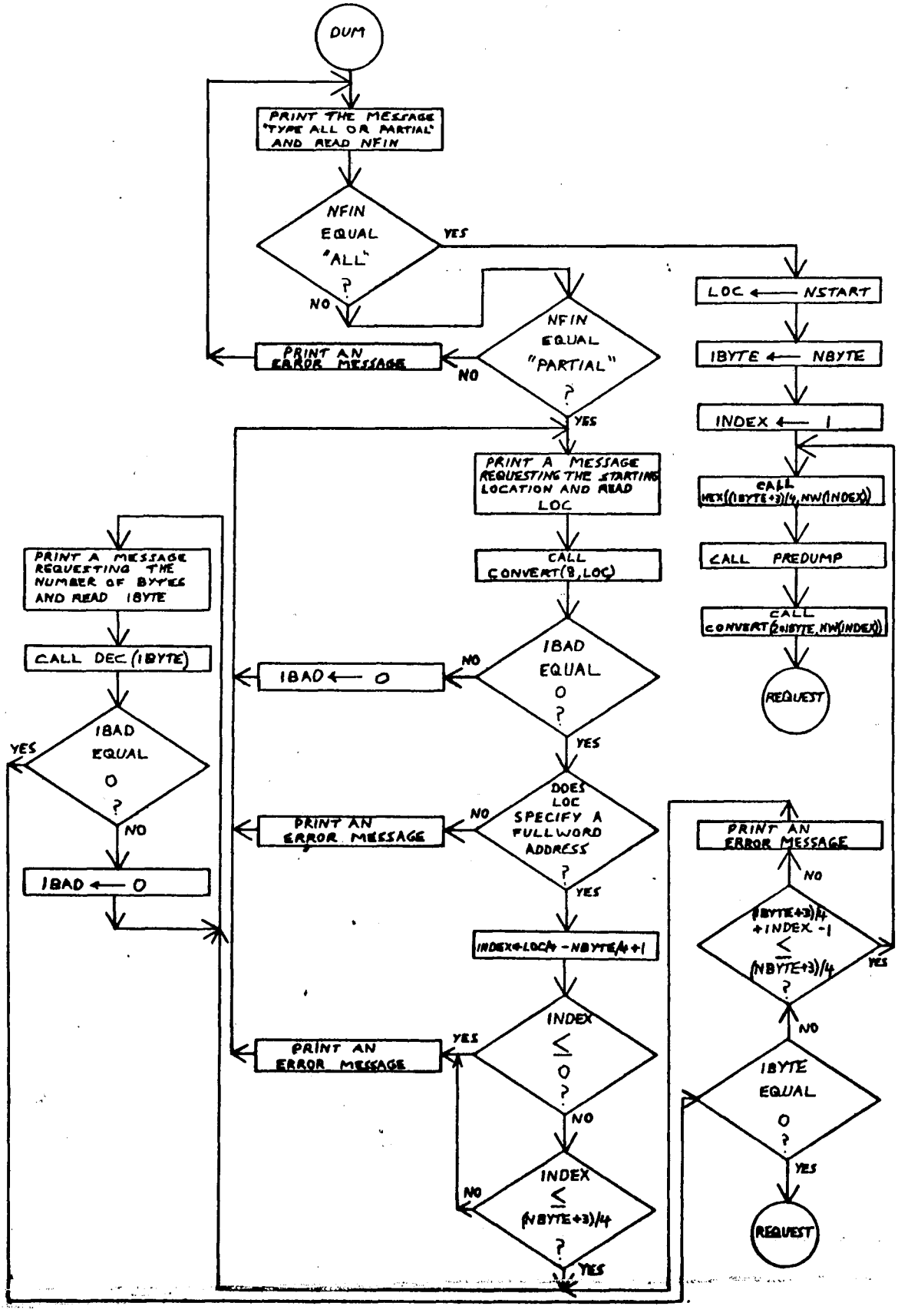
The following symbolic notations are used in the flowchart:

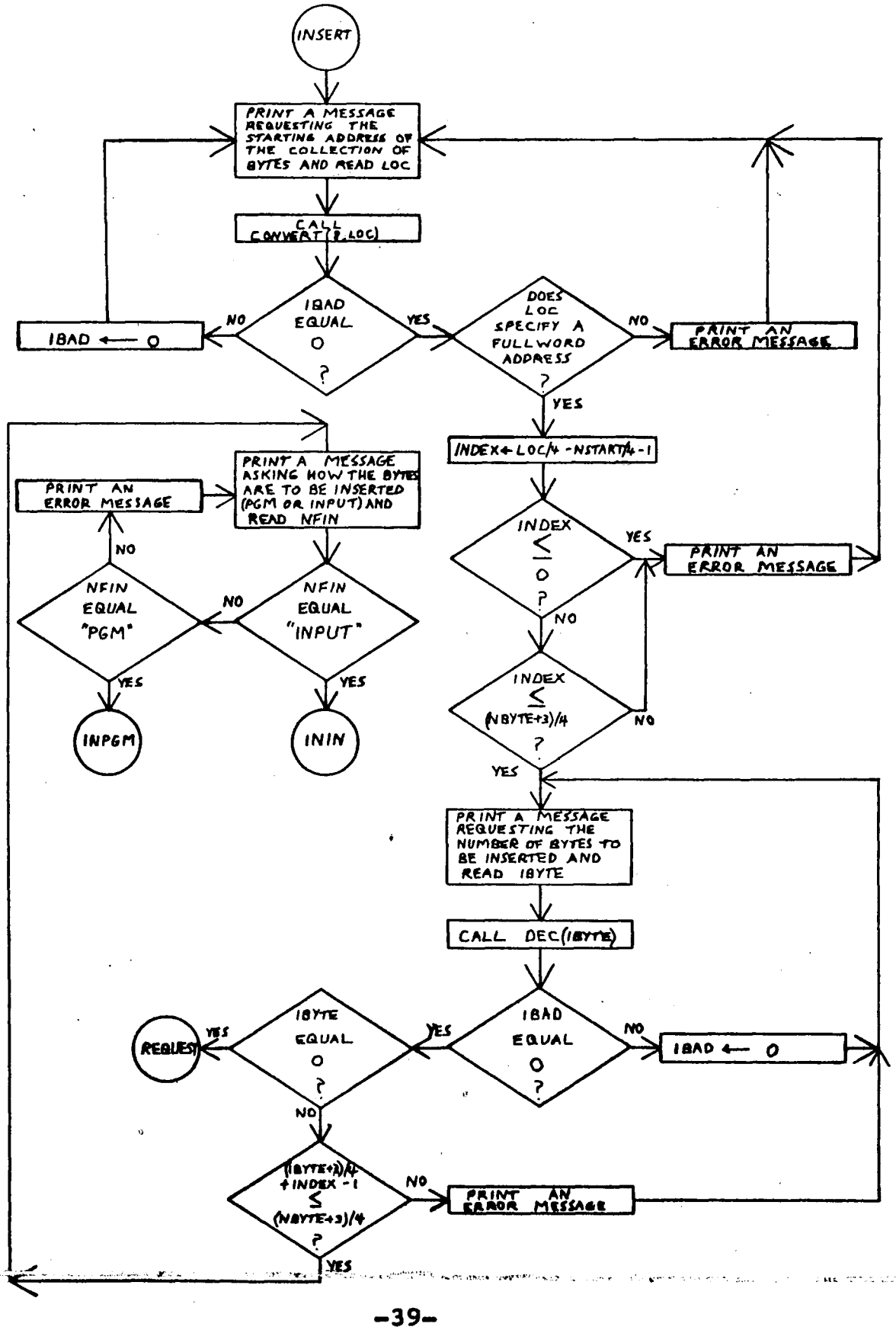
Symbolic Notation	Meaning
name ←— y	set 'name' equal to y
C(Xi) ←— b	set the contents of register Xi equal to b
L(d) ←— C(Xi)	put the contents of register Xi into location d
+	arithmetic sum
•	arithmetic product
-	arithmetic difference
/	arithmetic division
.NOT.	boolean complement
.AND.	boolean AND
.OR.	boolean OR
.XOR.	boolean EXCLUSIVE OR

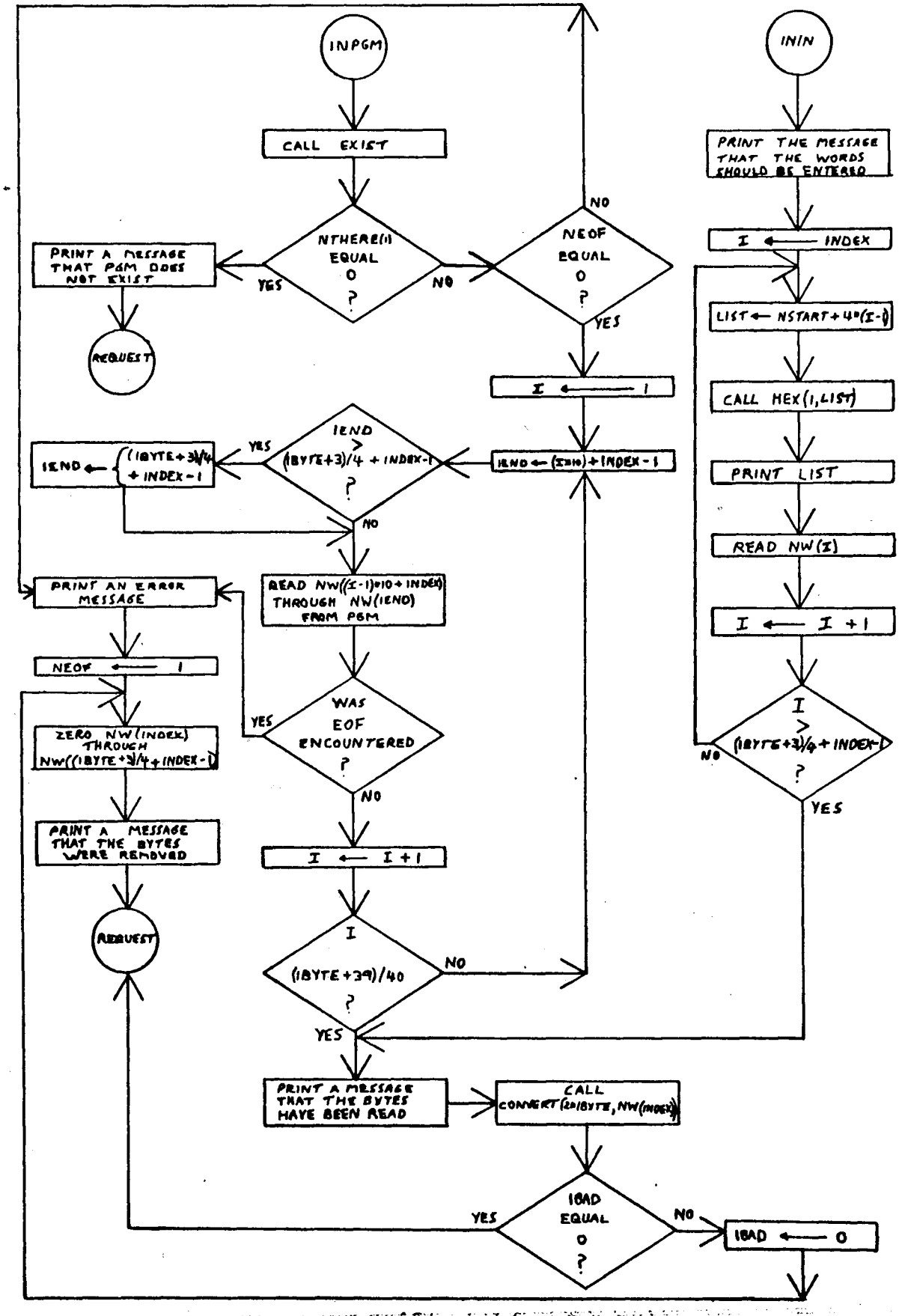
5.2 Flowchart

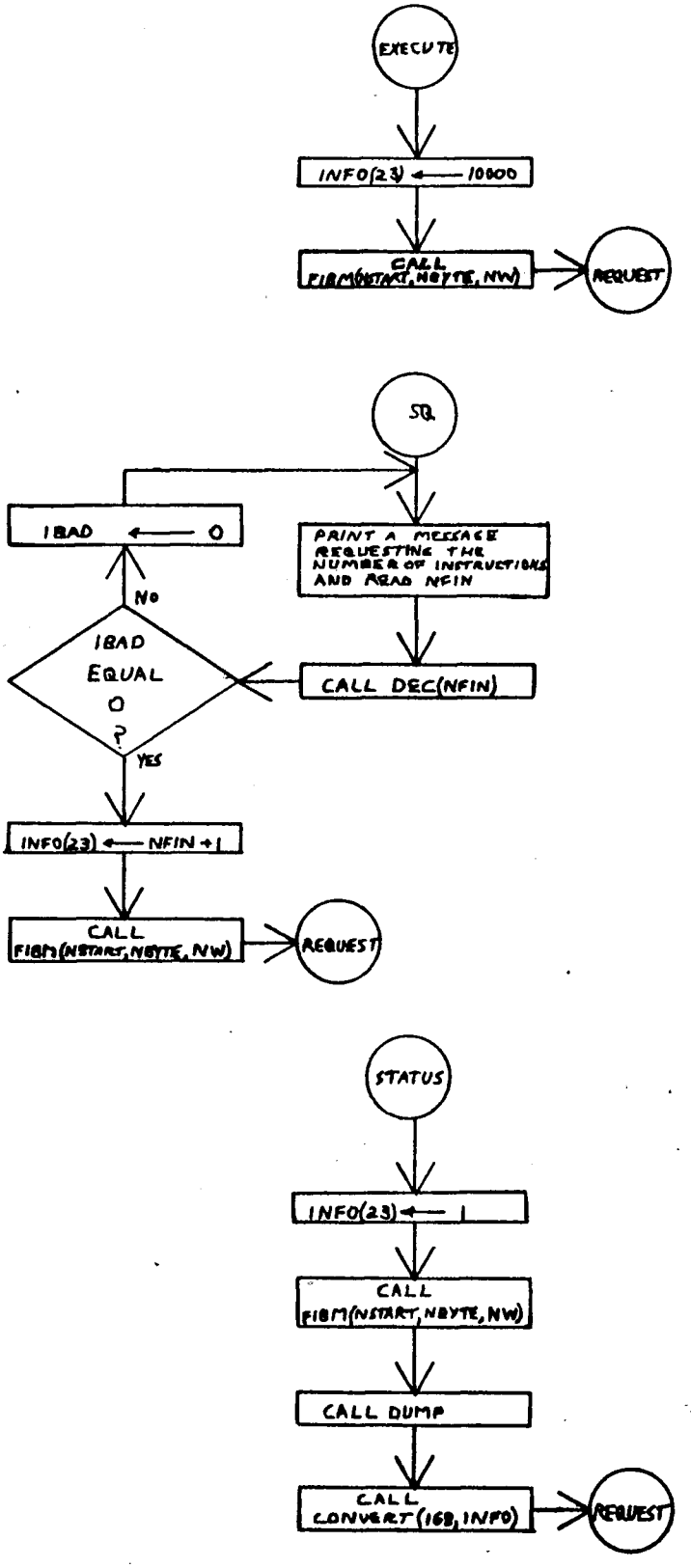


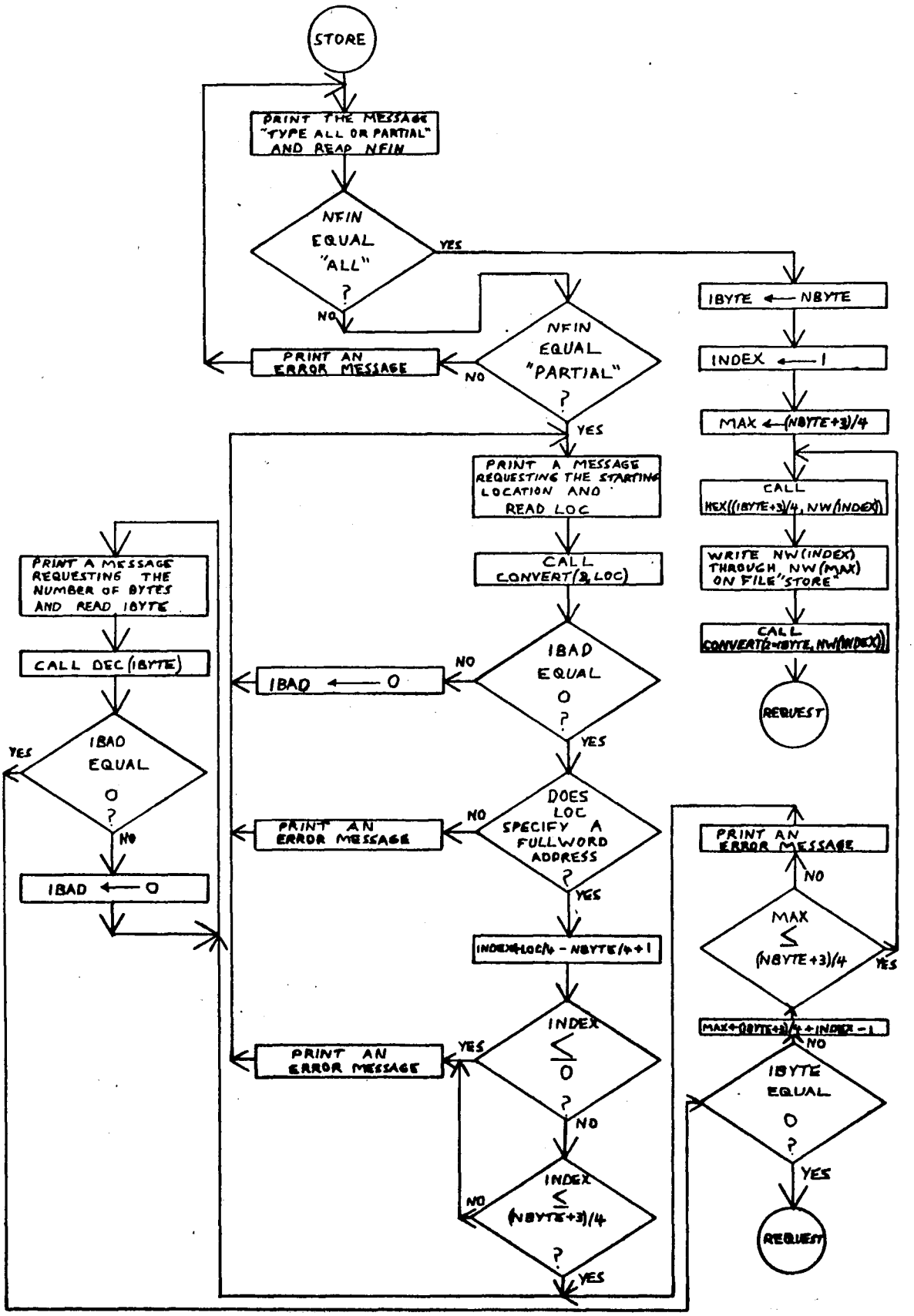

MAIN2
PRINT A MESSAGE REQUESTING THE PSW AND READ INFO(1), INFO(2)
CALL CONVERT(16, INFO)
IBAD EQUAL 0 ?
NO
IBAD ← 0
YES
PRINT "REQUEST" AND READ NFIN
REQUEST
MAIN1
NFIN EQUAL "END" ?
YES
PRINT MESSAGE "PROGRAM ENDED BY USER"
NO
NFIN EQUAL "BYE" ?
YES
END OF PROGRAM
NO
NFIN EQUAL "INSERT" ?
YES
INSERT
NO
NFIN EQUAL "EXECUTE" ?
YES
EXECUTE
NO
PRINT AN ERROR MESSAGE
NO
NFIN EQUAL "STORE" ?
YES
STORE
NO
NFIN EQUAL "REWINDS" ?
YES
REWINDS
NO
NFIN EQUAL "REWINDP" ?
YES
REWINDP
NO
NFIN EQUAL "NEWPSW" ?
YES
NO
NFIN EQUAL "STATUS" ?
YES
STATUS
NO
NFIN EQUAL "S" ?
YES
SQ
NO
NFIN EQUAL "DUMP" ?
YES
DUM
NO

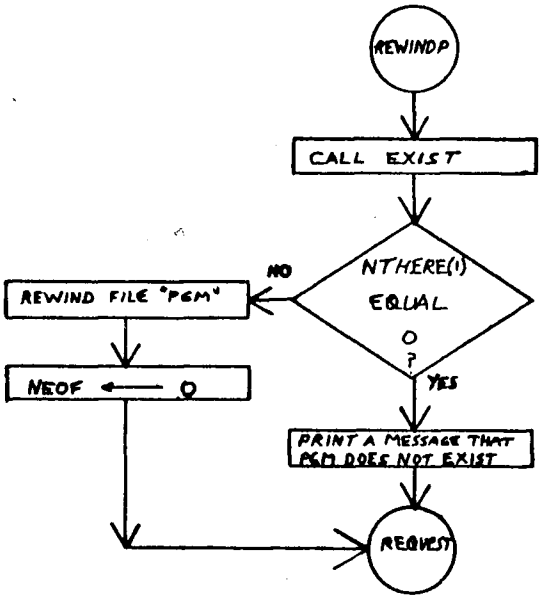

DUM
PRINT THE MESSAGE "TYPE ALL OR PARTIAL" AND READ NFIN
NFIN EQUAL "ALL" ?
YES
NO
NFIN EQUAL "PARTIAL" ?
NO
PRINT AN ERROR MESSAGE
YES
LOC ← NSTART
IBYTE ← NBYTE
INDEX ← 1
CALL HEX((IBYTE+3)/4, NW(INDEX))
CALL PREDUMP
CALL CONVERT(2,IBYTE, NW(INDEX))
REQUEST
PRINT A MESSAGE REQUESTING THE STARTING LOCATION AND READ LOC
CALL CONVERT(8,LOC)
IBAD EQUAL 0 ?
NO
IBAD ← 0
YES
DOES LOC SPECIFY A FULLWORD ADDRESS ?
NO
PRINT AN ERROR MESSAGE
YES
INDEX←LOC/4 - NBYTE/4 +1
INDEX ≤ 0 ?
YES
PRINT AN ERROR MESSAGE
NO
INDEX ≤ (NBYTE+3)/4 ?
NO
YES
PRINT A MESSAGE REQUESTING THE NUMBER OF BYTES AND READ IBYTE
CALL DEC(IBYTE)
IBAD EQUAL 0 ?
YES
NO
IBAD ← 0
PRINT AN ERROR MESSAGE
NO
(IBYTE+3)/4 +INDEX -1 ≤ (NBYTE+3)/4 ?
YES
NO
IBYTE EQUAL 0 ?
YES
REQUEST

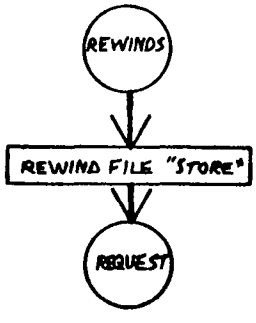

INSERT
PRINT A MESSAGE REQUESTING THE STARTING ADDRESS OF THE COLLECTION OF BYTES AND READ LOC
CALL CONVERT(8,LOC)
IBAD EQUAL 0 ?
NO
YES
IBAD ← 0
DOES LOC SPECIFY A FULLWORD ADDRESS ?
NO
PRINT AN ERROR MESSAGE
YES
INDEX ← LOC/4 - NSTART/4 - 1
INDEX ≤ 0 ?
YES
PRINT AN ERROR MESSAGE
NO
INDEX ≤ (NBYTE+3)/4 ?
NO
YES
PRINT A MESSAGE REQUESTING THE NUMBER OF BYTES TO BE INSERTED AND READ IBYTE
CALL DEC(IBYTE)
IBAD EQUAL 0 ?
YES
NO
IBAD ← 0
IBYTE EQUAL 0 ?
YES
REQUEST
NO
(IBYTE+3)/4 + INDEX - 1 ≤ (NBYTE+3)/4 ?
NO
PRINT AN ERROR MESSAGE
YES
PRINT A MESSAGE ASKING HOW THE BYTES ARE TO BE INSERTED (PGM OR INPUT) AND READ NFIN
NFIN EQUAL "INPUT"
YES
ININ
NO
NFIN EQUAL "PGM"
YES
INPGM
NO
PRINT AN ERROR MESSAGE

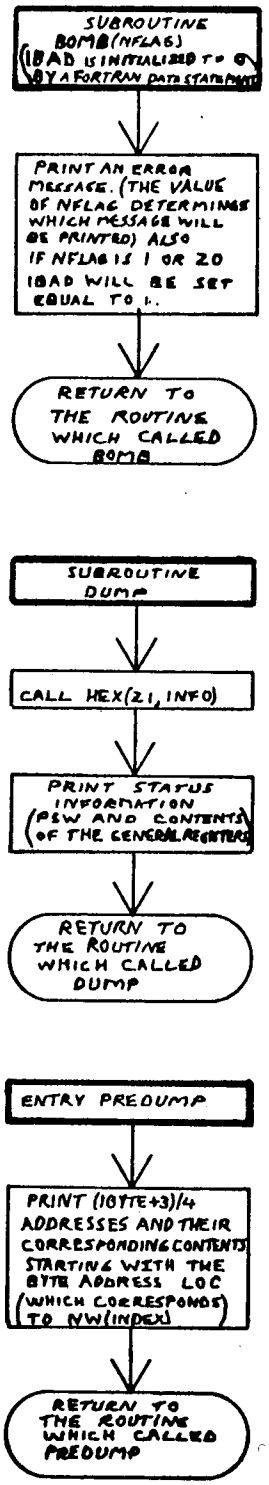

INPGM
CALL EXIST
NTHERE(1) EQUAL 0 ?
YES
NO
PRINT A MESSAGE THAT PGM DOES NOT EXIST
REQUEST
NEOF EQUAL 0 ?
NO
YES
I ← 1
IEND ← (Σ10)+INDEX-1
IEND > (IBYTE+3)/4 + INDEX-1 ?
YES
IEND ← (IBYTE+3)/4 + INDEX-1
NO
READ NW((I-1)*10+INDEX) THROUGH NW(IEND) FROM PGM
WAS EOF ENCOUNTERED ?
YES
NO
I ← I+1
I > (IBYTE+39)/40 ?
NO
YES
PRINT AN ERROR MESSAGE
NEOF ← 1
ZERO NW(INDEX) THROUGH NW((IBYTE+3)/4+INDEX-1)
PRINT A MESSAGE THAT THE BYTES WERE REMOVED
REQUEST
PRINT A MESSAGE THAT THE BYTES HAVE BEEN READ
CALL CONVERT(20,IBYTE,NW(INDEX))
IBAD EQUAL 0 ?
YES
NO
IBAD ← 0
ININ
PRINT THE MESSAGE THAT THE WORDS SHOULD BE ENTERED
I ← INDEX
LIST ← NSTART+4*(I-1)
CALL HEX(1,LIST)
PRINT LIST
READ NW(I)
I ← I+1
I > (IBYTE+3)/4 + INDEX-1 ?
NO
YES

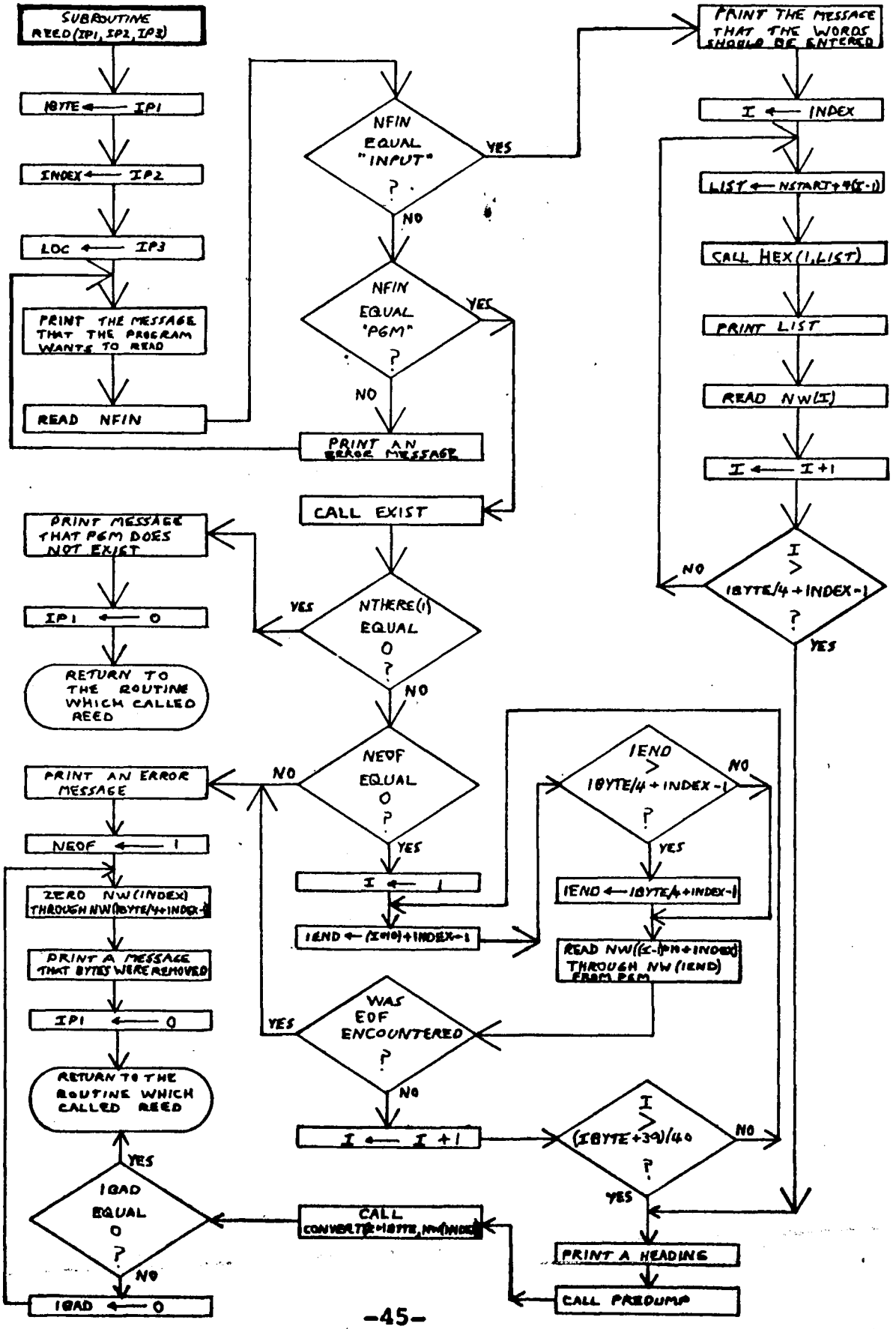

EXECUTE
INFO(23) ← 10000
CALL FIBM(NSTART, NBYTE, NW)
REQUEST
SQ
IBAD ← 0
PRINT A MESSAGE REQUESTING THE NUMBER OF INSTRUCTIONS AND READ NFIN
No
IBAD EQUAL 0 ?
CALL DEC(NFIN)
YES
INFO(23) ← NFIN + 1
CALL FIBM(NSTART, NBYTE, NW)
REQUEST
STATUS
INFO(23) ← 1
CALL FIBM(NSTART, NBYTE, NW)
CALL DUMP
CALL CONVERT(168, INFO)
REQUEST

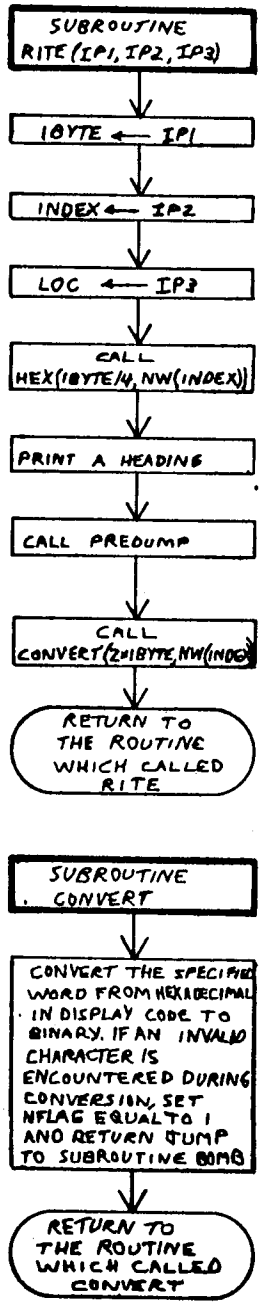

STORE
PRINT THE MESSAGE "TYPE ALL OR PARTIAL" AND READ NFIN
NFIN EQUAL "ALL" ?
YES
NO
NFIN EQUAL "PARTIAL" ?
NO
PRINT AN ERROR MESSAGE
YES
IBYTE ← NBYTE
INDEX ← 1
MAX ← (NBYTE+3)/4
CALL HEX((IBYTE+3)/4, NW(INDEX))
WRITE NW(INDEX) THROUGH NW(MAX) ON FILE "STORE"
CALL CONVERT(2, IBYTE, NW(INDEX))
REQUEST
PRINT A MESSAGE REQUESTING THE STARTING LOCATION AND READ LOC
CALL CONVERT(3, LOC)
IBAD EQUAL 0 ?
NO
IBAD ← 0
YES
DOES LOC SPECIFY A FULLWORD ADDRESS ?
NO
PRINT AN ERROR MESSAGE
YES
INDEX←LOC/4 - NBYTE/4+1
INDEX ≤ 0 ?
YES
PRINT AN ERROR MESSAGE
NO
INDEX ≤ (NBYTE+3)/4 ?
NO
YES
PRINT A MESSAGE REQUESTING THE NUMBER OF BYTES AND READ IBYTE
CALL DEC(IBYTE)
IBAD EQUAL 0 ?
YES
NO
IBAD ← 0
PRINT AN ERROR MESSAGE
NO
MAX ≤ (NBYTE+3)/4
YES
MAX←(IBYTE+3)/4+INDEX-1
IBYTE EQUAL 0 ?
NO
YES
REQUEST


REWINDP
CALL EXIST
NTHERE(1)
EQUAL
0
?
NO
YES
REWIND FILE "PGM"
NEOF ← 0
PRINT A MESSAGE THAT
PGM DOES NOT EXIST
REQUEST

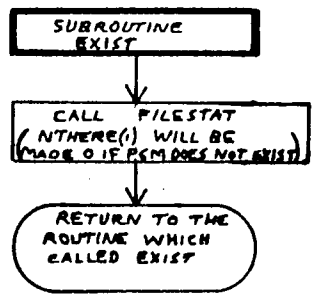

REWINDS
REWIND FILE "STORE"
REQUEST

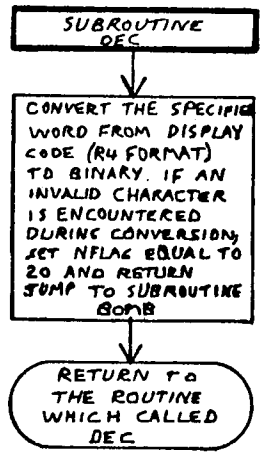

SUBROUTINE BOMB(NFLAG)
(IBAD IS INITIALIZED TO 0 BY A FORTRAN DATA STATEMENT)
PRINT AN ERROR MESSAGE. (THE VALUE OF NFLAG DETERMINES WHICH MESSAGE WILL BE PRINTED) ALSO IF NFLAG IS 1 OR 20 IBAD WILL BE SET EQUAL TO 1.
RETURN TO THE ROUTINE WHICH CALLED BOMB
SUBROUTINE DUMP
CALL HEX(21, INFO)
PRINT STATUS INFORMATION (PSW AND CONTENTS OF THE GENERAL REGISTERS)
RETURN TO THE ROUTINE WHICH CALLED DUMP
ENTRY PREDUMP
PRINT (10TTE+3)/4 ADDRESSES AND THEIR CORRESPONDING CONTENTS STARTING WITH THE BYTE ADDRESS LOC (WHICH CORRESPONDS TO NW(INDEX))
RETURN TO THE ROUTINE WHICH CALLED PREDUMP

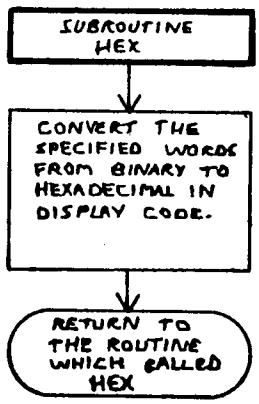

SUBROUTINE REED(IP1, IP2, IP3)
IBYTE ← IP1
INDEX ← IP2
LOC ← IP3
PRINT THE MESSAGE THAT THE PROGRAM WANTS TO READ
READ NFIN
NFIN EQUAL "INPUT" ?
YES
NO
NFIN EQUAL "PGM" ?
YES
NO
PRINT AN ERROR MESSAGE
PRINT THE MESSAGE THAT THE WORDS SHOULD BE ENTERED
I ← INDEX
LIST ← NSTART+9(I-1)
CALL HEX(1,LIST)
PRINT LIST
READ NW(I)
I ← I+1
I > IBYTE/4+INDEX-1 ?
NO
YES
CALL EXIST
NTHERE(1) EQUAL 0 ?
YES
NO
PRINT MESSAGE THAT PGM DOES NOT EXIST
IP1 ← 0
RETURN TO THE ROUTINE WHICH CALLED REED
NEOF EQUAL 0 ?
NO
YES
PRINT AN ERROR MESSAGE
NEOF ← 1
ZERO NW(INDEX) THROUGH NW(IBYTE/4+INDEX-1)
PRINT A MESSAGE THAT BYTES WERE REMOVED
IP1 ← 0
RETURN TO THE ROUTINE WHICH CALLED REED
I ← 1
IEND ← (I*10)+INDEX-1
IEND > IBYTE/4+INDEX-1 ?
NO
YES
IEND ← IBYTE/4+INDEX-1
READ NW((I-1)*10+INDEX) THROUGH NW(IEND) FROM PGM
WAS EOF ENCOUNTERED ?
YES
NO
I ← I+1
I > (IBYTE+39)/40 ?
NO
YES
IGAD EQUAL 0 ?
YES
NO
IGAD ← 0
CALL CONVERT(IP1,IBYTE,NW(INDEX))
PRINT A HEADING
CALL PREDUMP

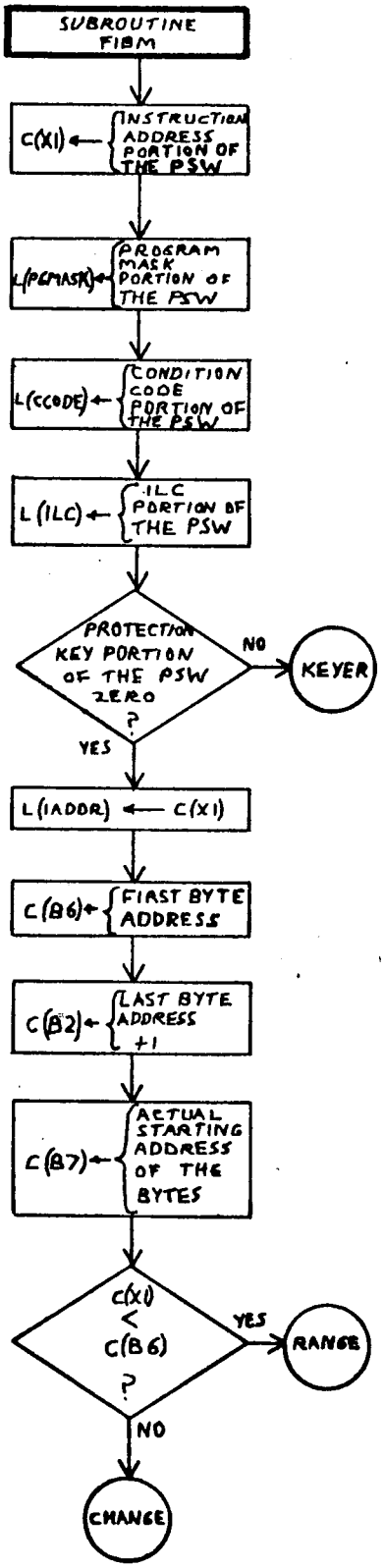

SUBROUTINE RITE(IP1, IP2, IP3)
IBYTE ← IP1
INDEX ← IP2
LOC ← IP3
CALL HEX(IBYTE/4, NW(INDEX))
PRINT A HEADING
CALL PREDUMP
CALL CONVERT(2,IBYTE,NW(INDEX))
RETURN TO THE ROUTINE WHICH CALLED RITE

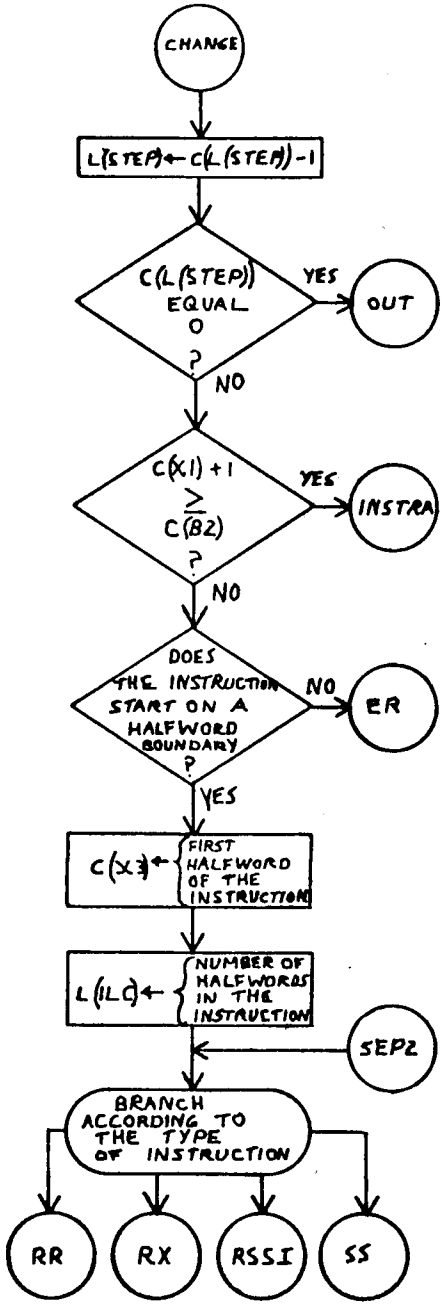
SUBROUTINE CONVERT
CONVERT THE SPECIFIED WORD FROM HEXADECIMAL IN DISPLAY CODE TO BINARY. IF AN INVALID CHARACTER IS ENCOUNTERED DURING CONVERSION, SET NFLAG EQUAL TO 1 AND RETURN JUMP TO SUBROUTINE BOMB
RETURN TO THE ROUTINE WHICH CALLED CONVERT

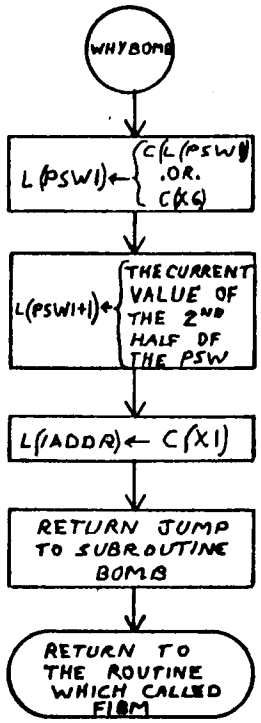

SUBROUTINE EXIST
CALL FILESTAT (NTHERE(I) WILL BE MADE 0 IF PSM DOES NOT EXIST)
RETURN TO THE ROUTINE WHICH CALLED EXIST

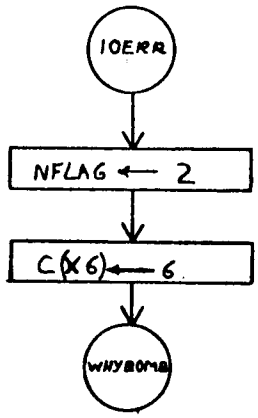

SUBROUTINE DEC
CONVERT THE SPECIFIED WORD FROM DISPLAY CODE (R4 FORMAT) TO BINARY. IF AN INVALID CHARACTER IS ENCOUNTERED DURING CONVERSION, SET NFLAG EQUAL TO 20 AND RETURN JUMP TO SUBROUTINE BOMB
RETURN TO THE ROUTINE WHICH CALLED DEC

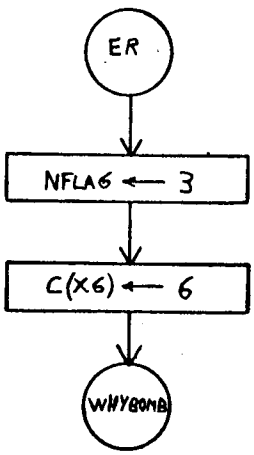

SUBROUTINE HEX
CONVERT THE SPECIFIED WORDS FROM BINARY TO HEXADECIMAL IN DISPLAY CODE.
RETURN TO THE ROUTINE WHICH CALLED HEX

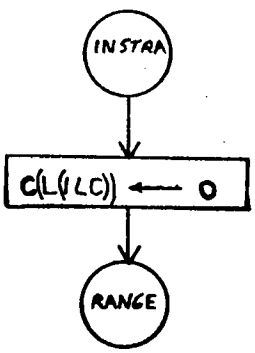

SUBROUTINE FIBM
C(X1) ← INSTRUCTION ADDRESS PORTION OF THE PSW
L(PGMASK) ← PROGRAM MASK PORTION OF THE PSW
L(CCODE) ← CONDITION CODE PORTION OF THE PSW
L(ILC) ← ILC PORTION OF THE PSW
PROTECTION KEY PORTION OF THE PSW ZERO ?
NO
KEYER
YES
L(IADDR) ← C(X1)
C(B6) ← FIRST BYTE ADDRESS
C(B2) ← LAST BYTE ADDRESS +1
C(B7) ← ACTUAL STARTING ADDRESS OF THE BYTES
C(X1) < C(B6) ?
YES
RANGE
NO
CHANGE

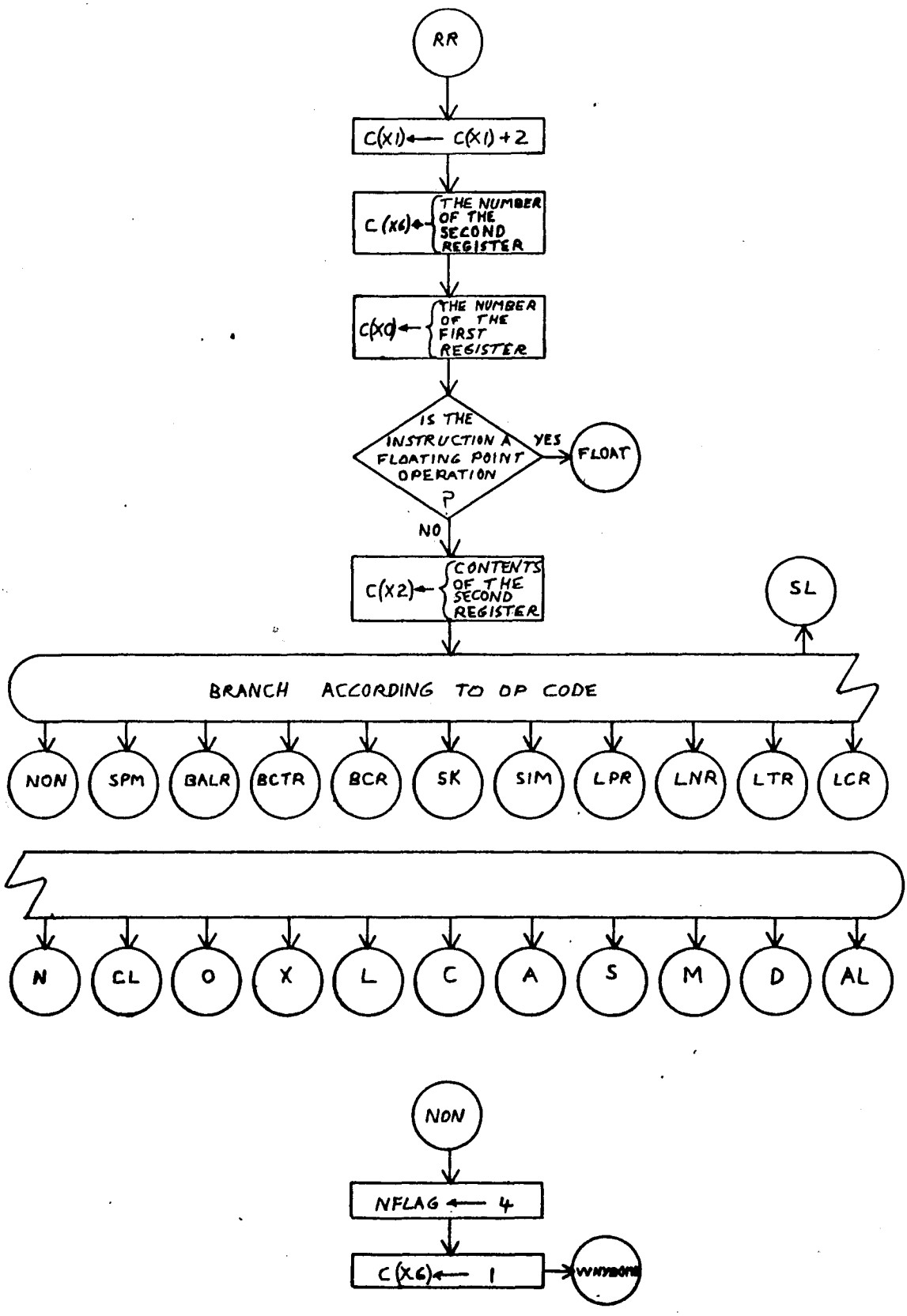

CHANGE
L(STEP)←C(L(STEP))-1
C(L(STEP)) EQUAL 0 ?
YES
OUT
NO
C(X1)+1 ≥ C(B2) ?
YES
INSTRA
NO
DOES THE INSTRUCTION START ON A HALFWORD BOUNDARY ?
NO
ER
YES
C(X3)← FIRST HALFWORD OF THE INSTRUCTION
L(ILC)← NUMBER OF HALFWORDS IN THE INSTRUCTION
SEP2
BRANCH ACCORDING TO THE TYPE OF INSTRUCTION
RR
RX
RSSI
SS

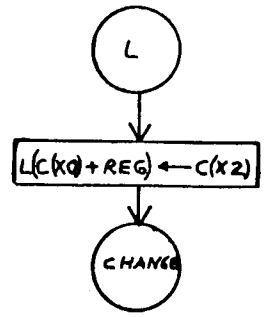

WHYBOMB
L(PSW1)← C(L(PSW1) .OR. C(X4)
L(PSW1+1)← THE CURRENT VALUE OF THE 2ND HALF OF THE PSW
L(IADDR)← C(X1)
RETURN JUMP TO SUBROUTINE BOMB
RETURN TO THE ROUTINE WHICH CALLED FIBM

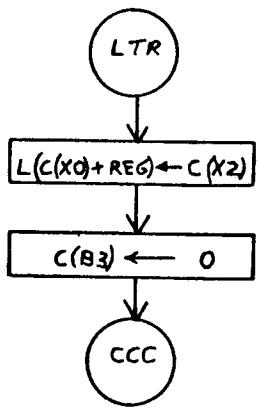

IOERR
NFLAG ← 2
C(X6) ← 6
WHYBOMB

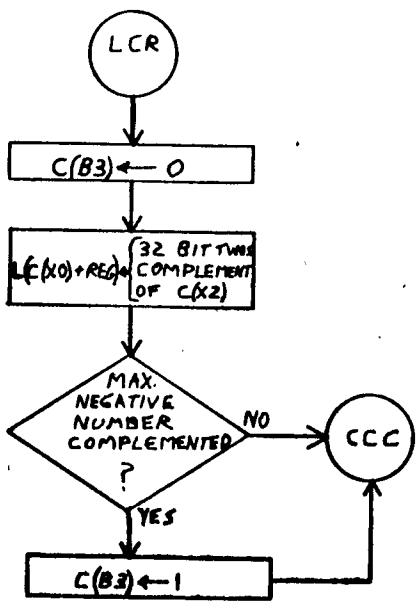

ER
NFLAG ← 3
C(X6) ← 6
WHYBOMB

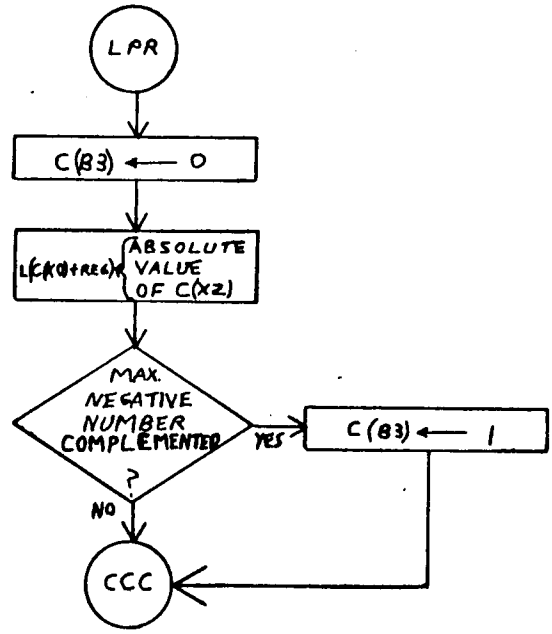

INSTRA
C(L(LC)) ← 0
RANGE

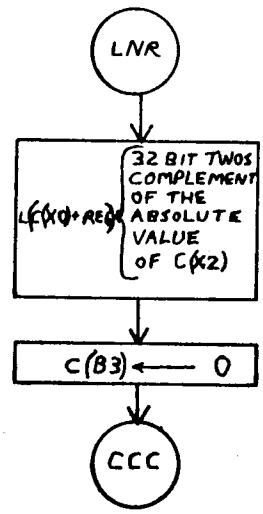

RR
C(X1) ← C(X1) + 2
C(X6) ← THE NUMBER OF THE SECOND REGISTER
C(X0) ← THE NUMBER OF THE FIRST REGISTER
IS THE INSTRUCTION A FLOATING POINT OPERATION ?
YES
FLOAT
NO
C(X2) ← CONTENTS OF THE SECOND REGISTER
SL
BRANCH ACCORDING TO OP CODE
NON
SPM
BALR
BCTR
BCR
SK
SIM
LPR
LNR
LTR
LCR
N
CL
O
X
L
C
A
S
M
D
AL
NON
NFLAG ← 4
C(X6) ← 1

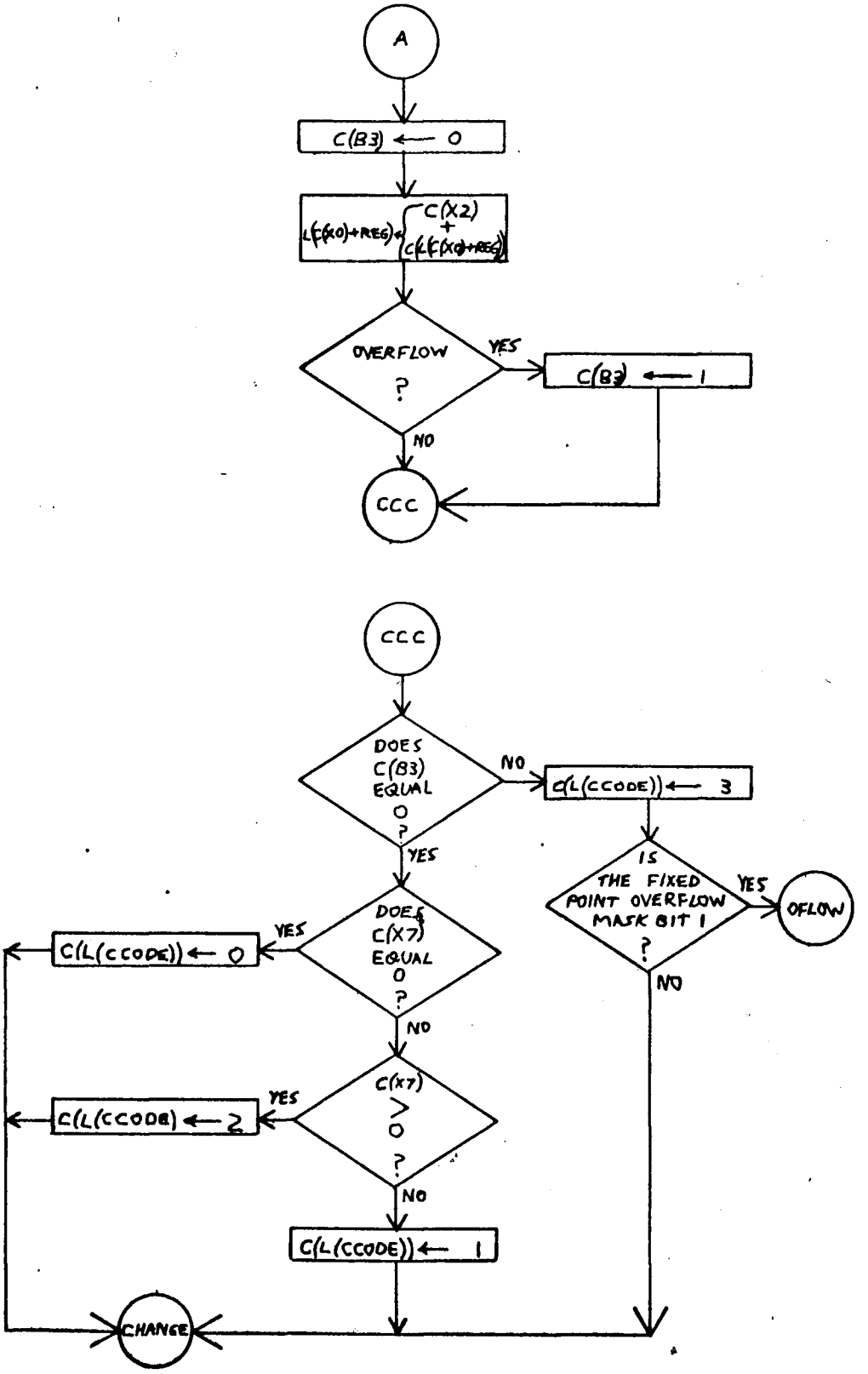

L
L(C(X0)+REG) ← C(X2)
CHANGE

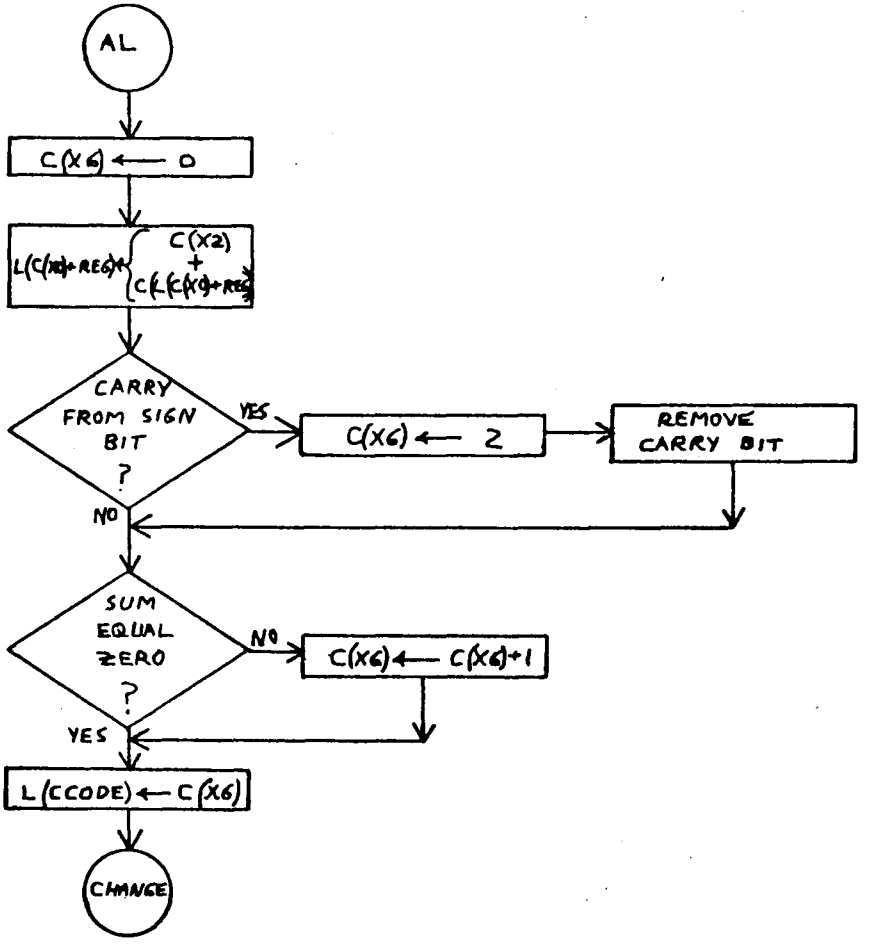

LTR
L(C(X0)+REG) ← C(X2)
C(B3) ← 0
CCC

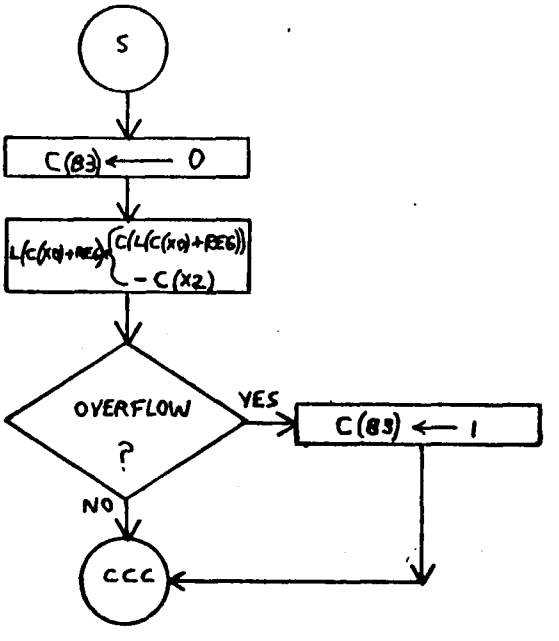

LCR
C(B3) ← 0
L(C(X0)+REG) ← 32 BIT TWOS COMPLEMENT OF C(X2)
MAX. NEGATIVE NUMBER COMPLEMENTED ?
NO
YES
CCC
C(B3) ← 1

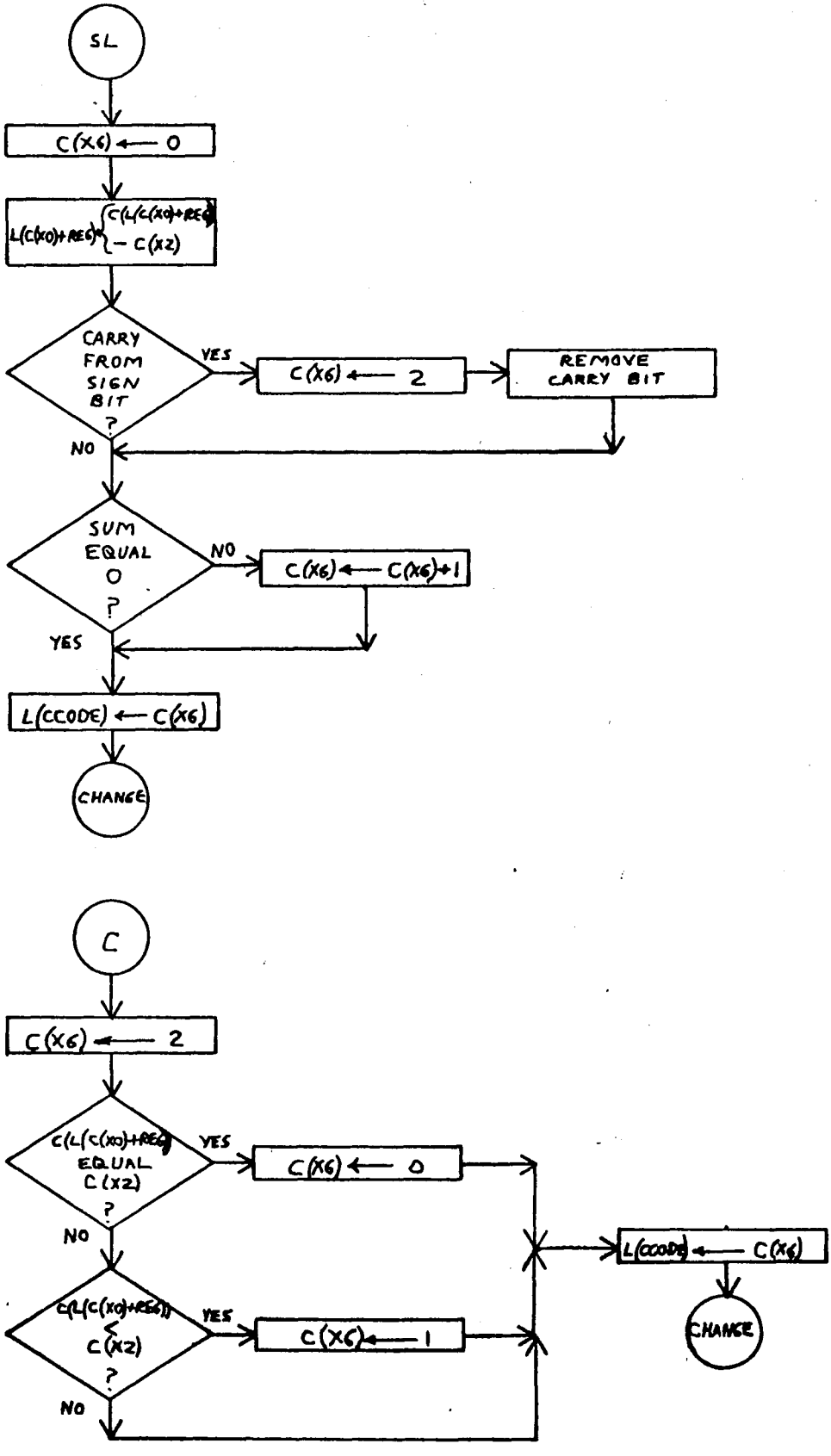

LPR
C(B3) ← 0
LC(X0+REG) ABSOLUTE VALUE OF C(X2)
MAX. NEGATIVE NUMBER COMPLEMENTER ?
YES
C(B3) ← 1
NO
CCC

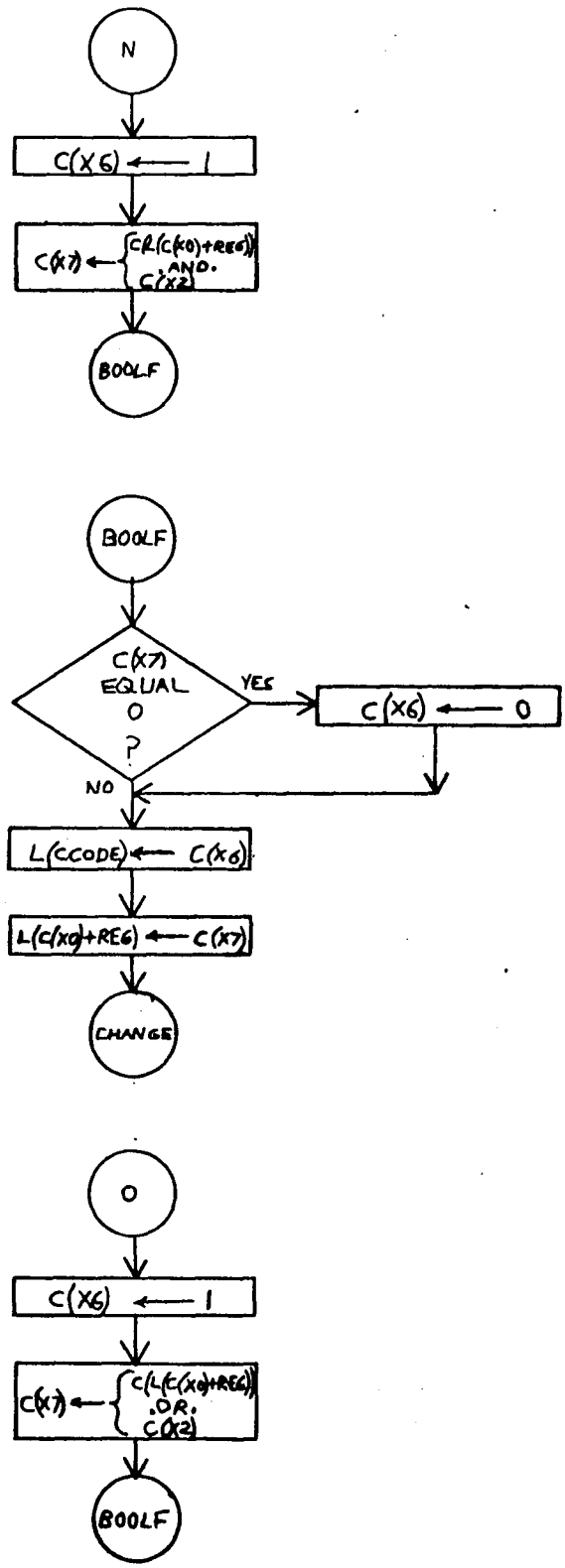

LNR
LC(X0+REG) 32 BIT TWOS COMPLEMENT OF THE ABSOLUTE VALUE OF C(X2)
C(B3) ← 0
CCC

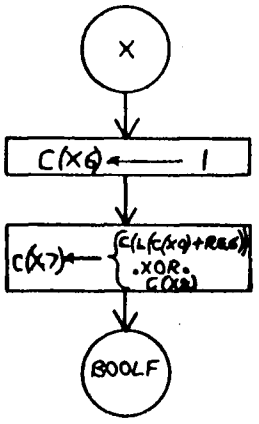

A
C(B3) ← 0
L(C(X0)+REG) ← C(X2) + C(L(C(X0)+REG))
OVERFLOW ?
YES
C(B3) ← 1
NO
CCC
CCC
DOES C(B3) EQUAL 0 ?
NO
C(L(CCODE)) ← 3
YES
IS THE FIXED POINT OVERFLOW MASK BIT 1 ?
YES
OFLOW
NO
DOES C(X7) EQUAL 0 ?
YES
C(L(CCODE)) ← 0
NO
C(X7) > 0 ?
YES
C(L(CCODE)) ← 2
NO
C(L(CCODE)) ← 1
CHANGE

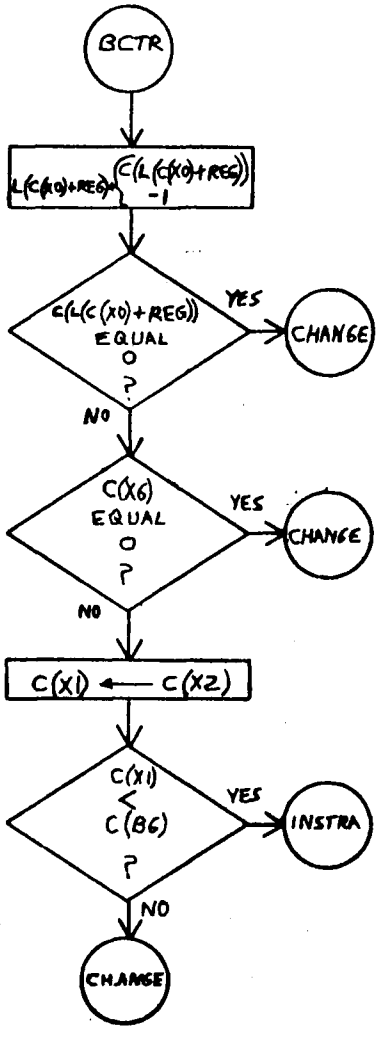

AL
C(X6) ← 0
C(X2)
+
CARRY FROM SIGN BIT ?
YES
C(X6) ← 2
REMOVE CARRY BIT
NO
SUM EQUAL ZERO ?
NO
C(X6) ← C(X6)+1
YES
L(CCODE) ← C(X6)
CHANGE

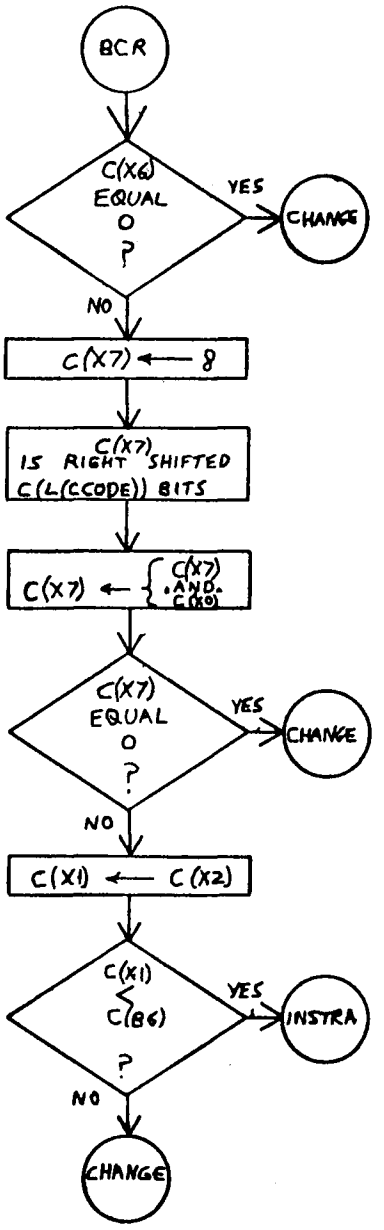

S
C(B3) ← 0
- C(X2)
OVERFLOW ?
YES
C(B3) ← 1
NO
CCC

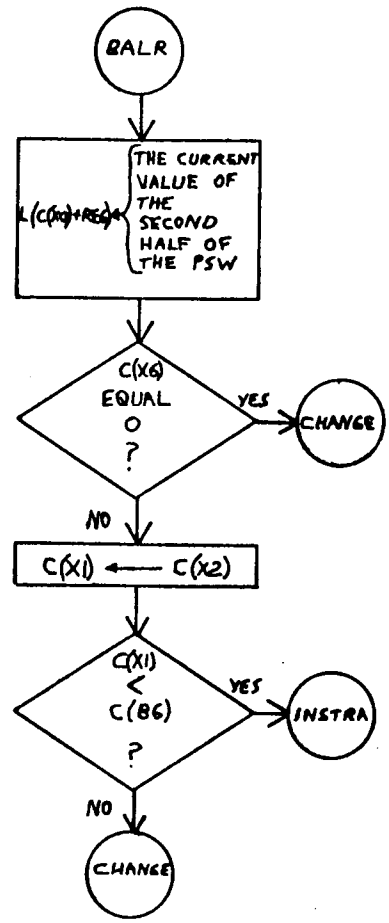

SL
C(X6) ← 0
L(C(X0)+REG) ← C(L(C(X0)+REG)) − C(X2)
CARRY FROM SIGN BIT ?
YES
C(X6) ← 2
REMOVE CARRY BIT
NO
SUM EQUAL 0 ?
NO
C(X6) ← C(X6)+1
YES
L(CCODE) ← C(X6)
CHANGE
C
C(X6) ← 2
C(L(C(X0)+REG)) EQUAL C(X2) ?
YES
C(X6) ← 0
NO
C(L(C(X0)+REG)) < C(X2) ?
YES
C(X6) ← 1
NO
L(CCODE) ← C(X6)
CHANGE

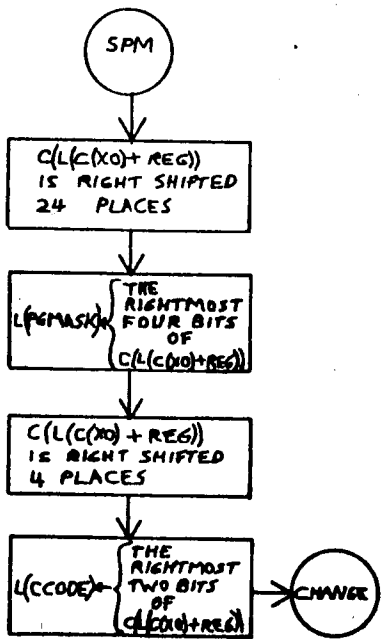

N
C(X6) ← 1
C(X7) ← C(R(C(X0)+REG)) .AND. C(X2)
BOOLF
BOOLF
C(X7) EQUAL 0 ?
YES
C(X6) ← 0
NO
L(CCODE) ← C(X6)
L(C(X0)+REG) ← C(X7)
CHANGE
0
C(X6) ← 1
C(X7) ← C(L(C(X0)+REG)) .OR. C(X2)
BOOLF

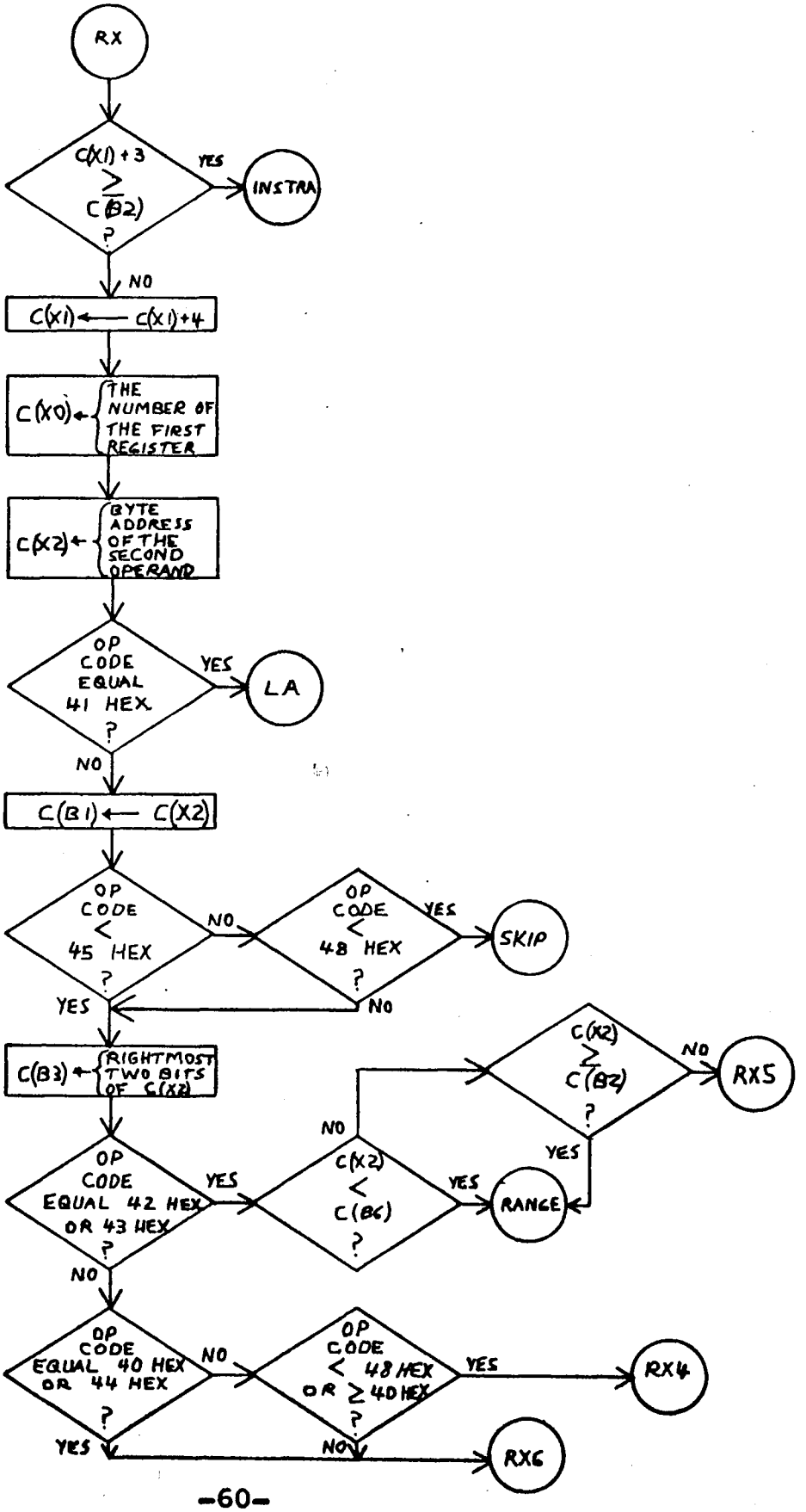

X
C(X6) ← 1
C(X7) ← C(L(C(X0)+REG))
.XOR.
C(X2)
BOOLF

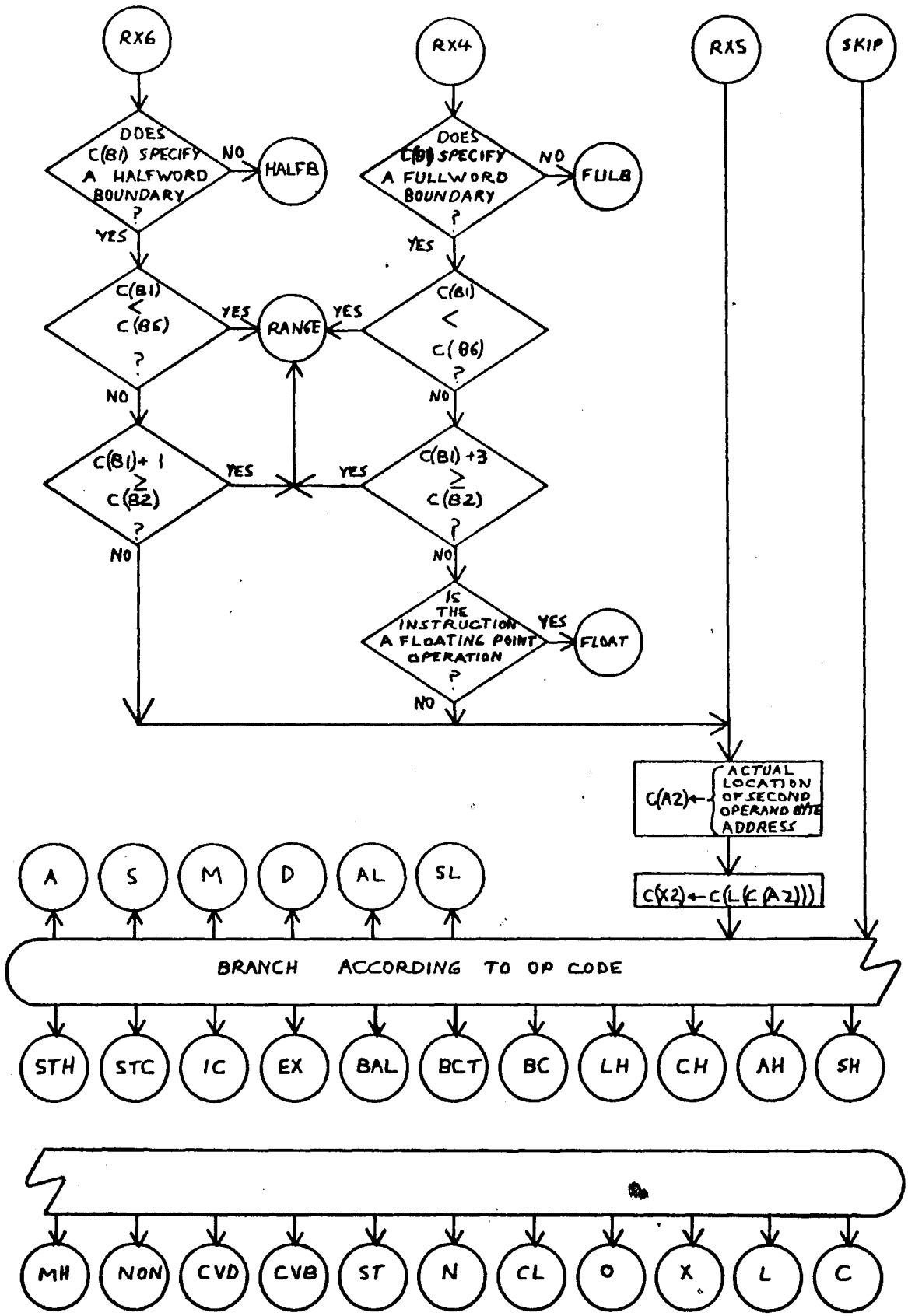

BCTR
L(C(X0)+REG) ← C(L(C(X0)+REG))
-1
C(L(C(X0)+REG)) EQUAL 0 ?
YES
CHANGE
NO
C(X6) EQUAL 0 ?
YES
CHANGE
NO
C(X1) ← C(X2)
C(X1) ≠ C(B6) ?
YES
INSTRA
NO
CHANGE

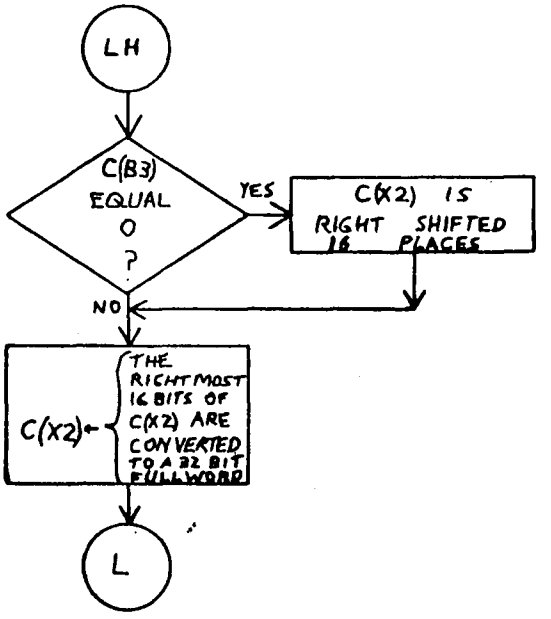

BCR
C(X6) EQUAL 0 ?
YES
CHANGE
NO
C(X7) ← 8
C(X7) IS RIGHT SHIFTED C(L(CCODE)) BITS
C(X7) ← C(X7) .AND. C(X0)
C(X7) EQUAL 0 ?
YES
CHANGE
NO
C(X1) ← C(X2)
C(X1) < C(B6) ?
YES
INSTRA
NO
CHANGE

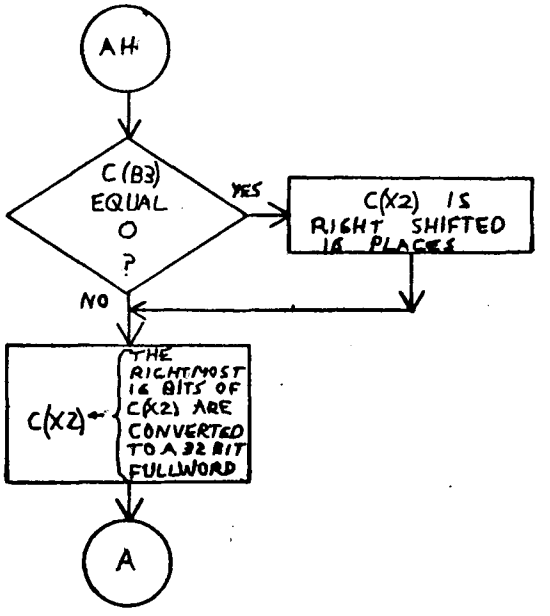

BALR
L(C(X2)+REG)←
THE CURRENT VALUE OF THE SECOND HALF OF THE PSW
C(X6) EQUAL 0 ?
YES
CHANGE
NO
C(X1) ← C(X2)
C(X1) < C(86) ?
YES
INSTRA
NO
CHANGE

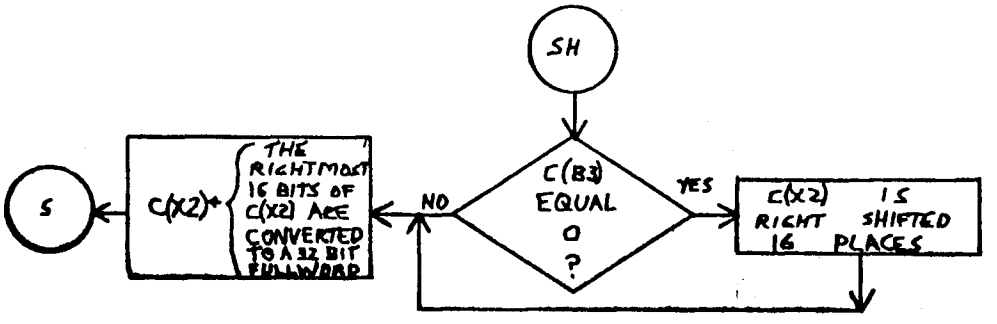

SPM
C(L(C(X0)+ REG)) IS RIGHT SHIFTED 24 PLACES
L(PGMASK)←
THE RIGHTMOST FOUR BITS OF C(L(C(X0)+REG))
C(L(C(X0) + REG)) IS RIGHT SHIFTED 4 PLACES
L(CCODE)←
THE RIGHTMOST TWO BITS OF C(L(C(X0)+REG))
CHANGE

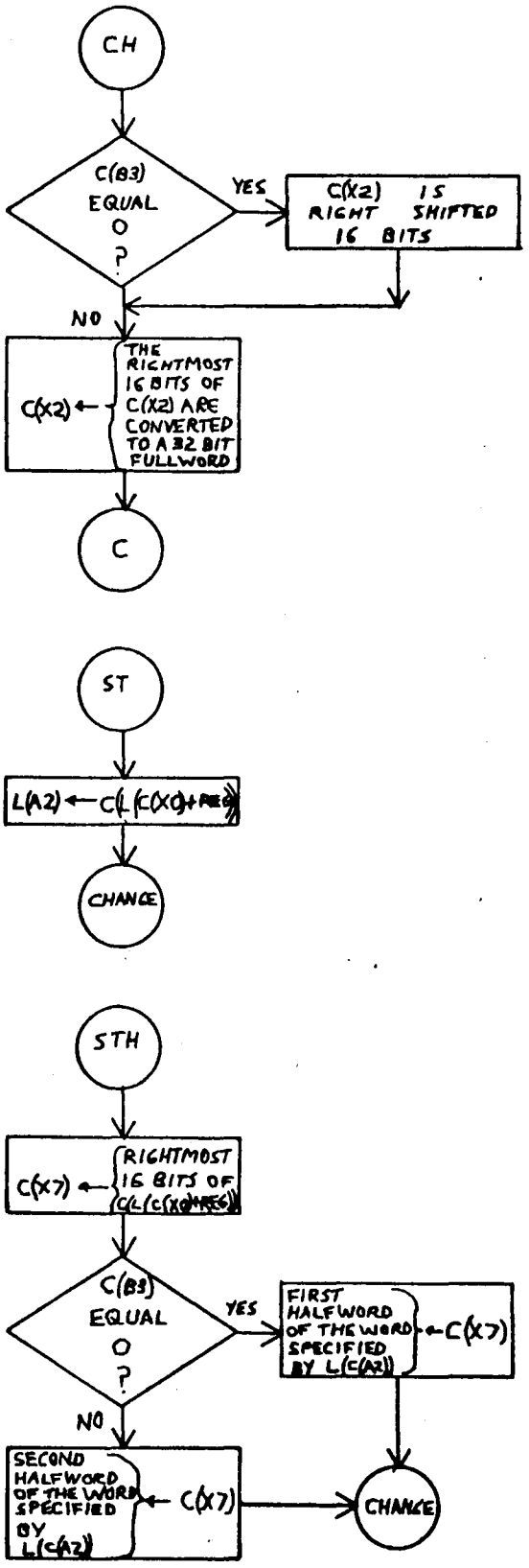

RX
C(X1) + 3 ≥ C(B2) ?
YES
INSTRA
NO
C(X1) ← C(X1)+4
C(X0) ← THE NUMBER OF THE FIRST REGISTER
C(X2) ← BYTE ADDRESS OF THE SECOND OPERAND
OP CODE EQUAL 41 HEX ?
YES
LA
NO
C(B1) ← C(X2)
OP CODE < 45 HEX ?
NO
OP CODE < 48 HEX ?
YES
SKIP
NO
YES
C(B3) ← RIGHTMOST TWO BITS OF C(X2)
OP CODE EQUAL 42 HEX OR 43 HEX ?
YES
C(X2) < C(B6) ?
YES
RANGE
NO
C(X2) ≥ C(B2) ?
NO
RX5
YES
NO
OP CODE EQUAL 40 HEX OR 44 HEX ?
NO
OP CODE < 48 HEX OR ≥ 40 HEX ?
YES
RX4
YES
NO
RX6

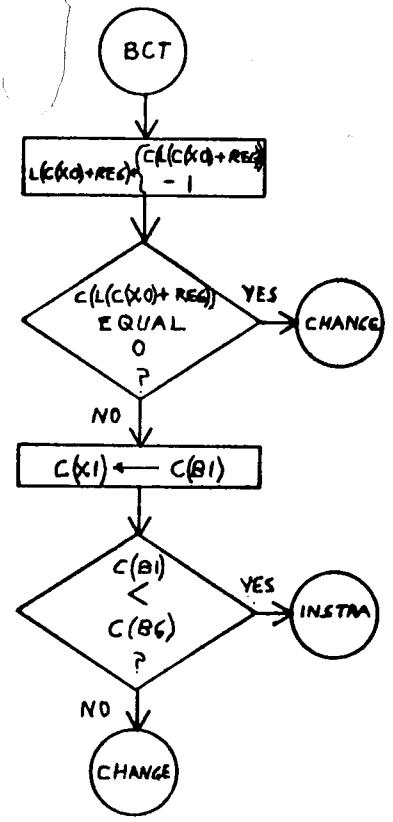

RX6
RX4
RX5
SKIP
DOES C(B1) SPECIFY A HALFWORD BOUNDARY ?
NO
HALFB
YES
DOES C(B1) SPECIFY A FULLWORD BOUNDARY ?
NO
FULLB
YES
C(B1) ≤ C(B6) ?
YES
RANGE
NO
C(B1) < C(B6) ?
YES
NO
C(B1)+1 ≥ C(B2) ?
YES
NO
C(B1)+3 ≥ C(B2) ?
YES
NO
IS THE INSTRUCTION A FLOATING POINT OPERATION ?
YES
FLOAT
NO
C(A2) ← ACTUAL LOCATION OF SECOND OPERAND BYTE ADDRESS
C(X2) ← C(L(C(A2)))
A
S
M
D
AL
SL
BRANCH ACCORDING TO OP CODE
STH
STC
IC
EX
BAL
BCT
BC
LH
CH
AH
SH
MH
NON
CVD
CVB
ST
N
CL
O
X
L
C

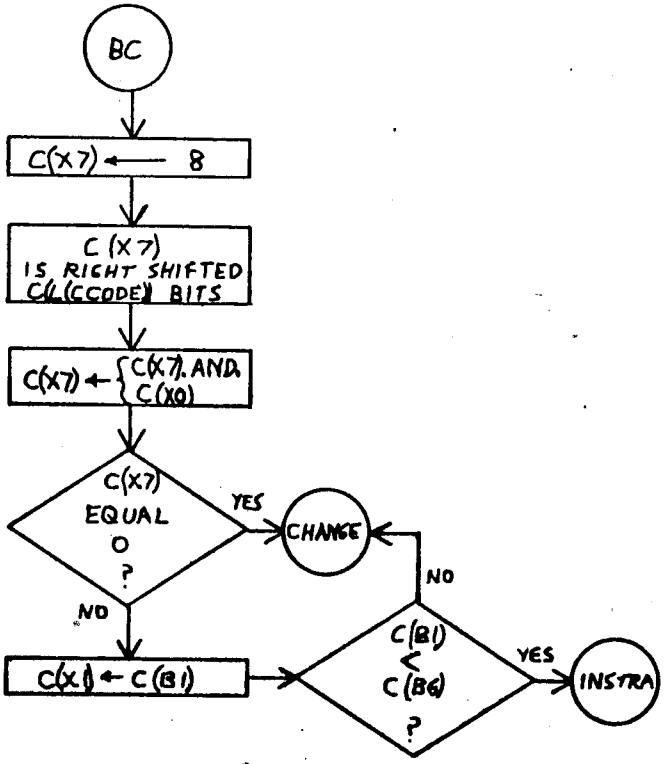

LH
C(B3) EQUAL 0 ?
YES
C(X2) IS RIGHT SHIFTED 16 PLACES
NO
C(X2)←
THE RIGHTMOST 16 BITS OF C(X2) ARE CONVERTED TO A 32 BIT FULLWORD
L

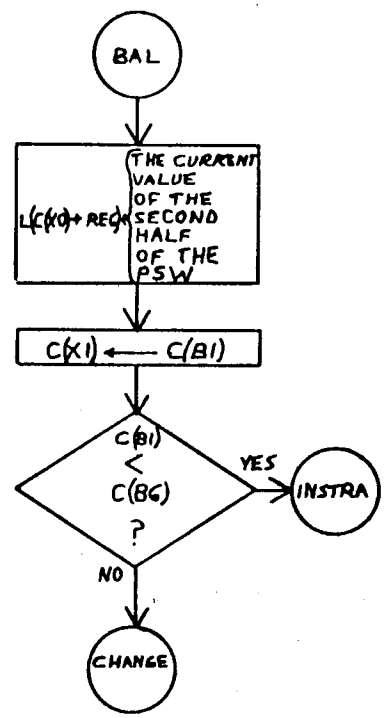

AH
C(B3) EQUAL 0 ?
YES
C(X2) IS RIGHT SHIFTED 16 PLACES
NO
C(X2)←
THE RIGHTMOST 16 BITS OF C(X2) ARE CONVERTED TO A 32 BIT FULLWORD
A

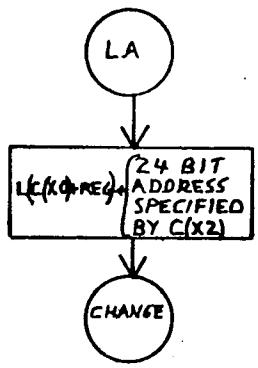

SH
C(B3) EQUAL 0 ?
YES
C(X2) IS RIGHT SHIFTED 16 PLACES
NO
C(X2)←
THE RIGHTMOST 16 BITS OF C(X2) ARE CONVERTED TO A 32 BIT FULLWORD
S

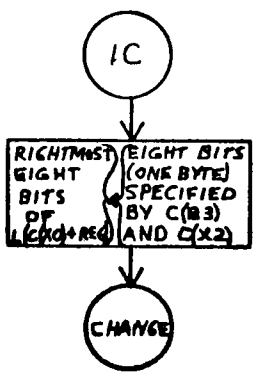

CH
C(B3) EQUAL 0 ?
YES
C(X2) IS RIGHT SHIFTED 16 BITS
NO
C(X2) ←
THE RIGHTMOST 16 BITS OF C(X2) ARE CONVERTED TO A 32 BIT FULLWORD
C
ST
CHANGE
STH
C(X7) ←
RIGHTMOST 16 BITS OF
C(B3) EQUAL 0 ?
YES
FIRST HALFWORD OF THE WORD SPECIFIED BY L(C(A2)) ← C(X7)
NO
SECOND HALFWORD OF THE WORD SPECIFIED BY L(C(A2)) ← C(X7)
CHANGE

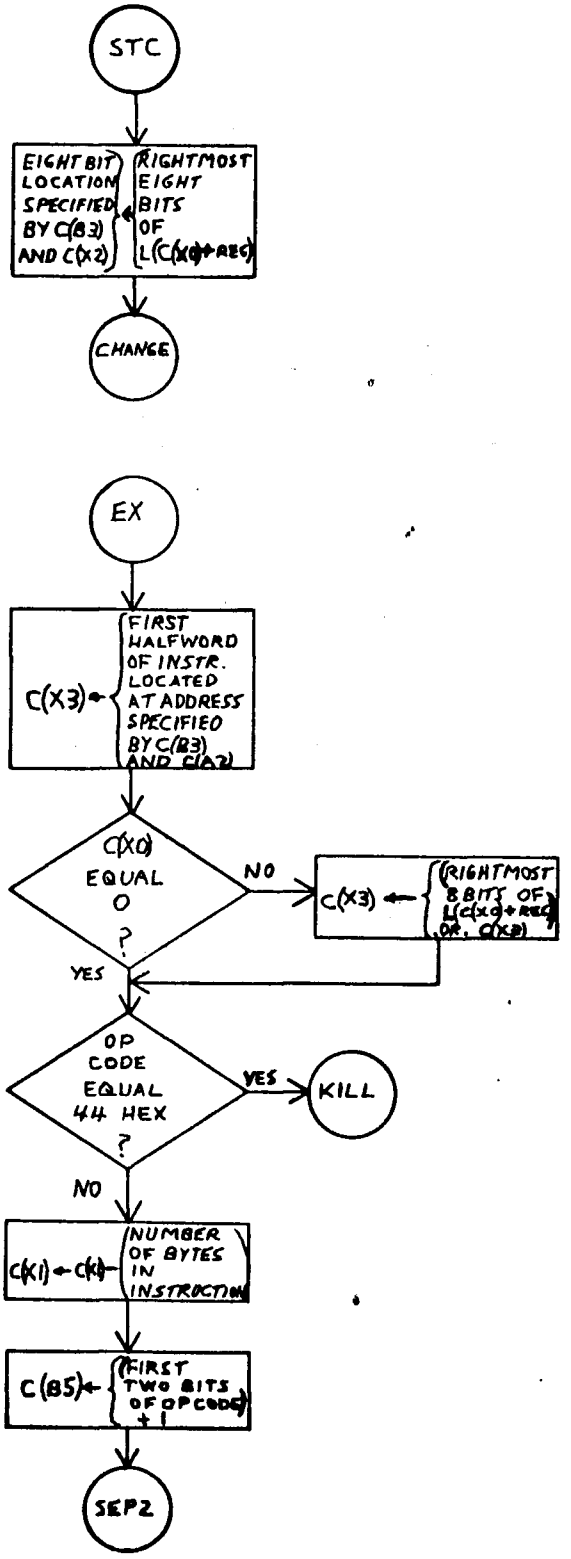

BCT
L(C(X0)+REG)
C(L(C(X0)+REG)) - 1
C(L(C(X0)+REG)) EQUAL 0 ?
YES
CHANGE
NO
C(X1) ← C(B1)
C(B1) < C(B6) ?
YES
INSTRA
NO
CHANGE

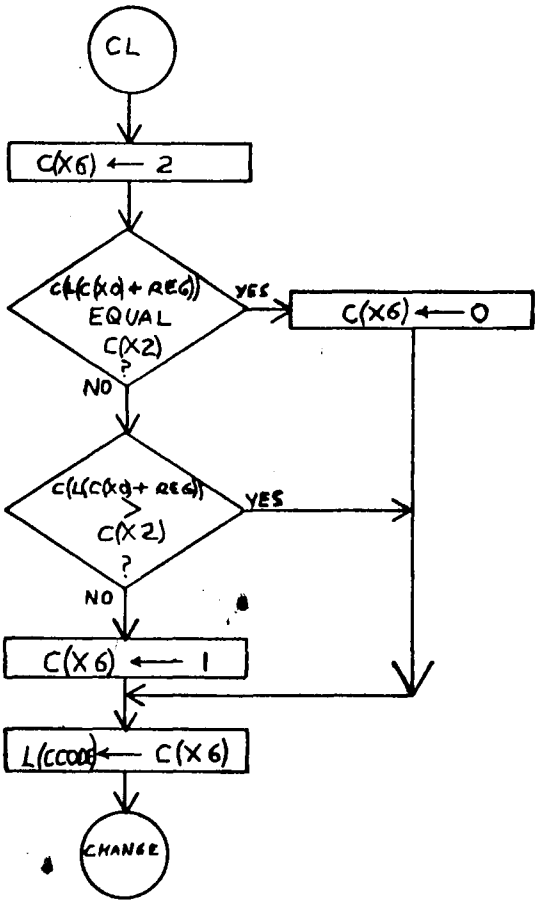

BC
C(X7) ← 8
C(X7) IS RIGHT SHIFTED C(L(CCODE)) BITS
C(X7) ← C(X7).AND. C(X0)
C(X7) EQUAL 0 ?
YES
CHANGE
NO
C(X1) ← C(B1)
C(B1) < C(B6) ?
YES
INSTRA
NO

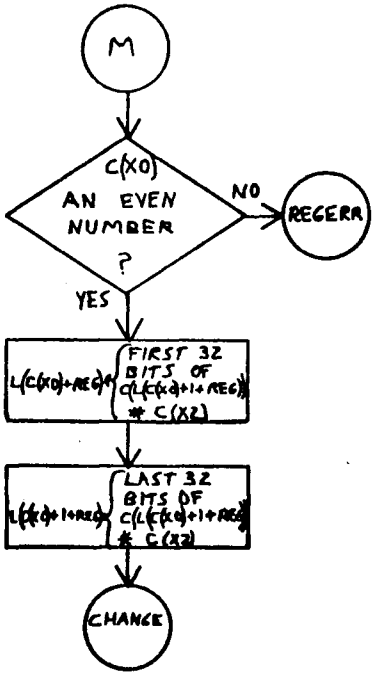

BAL
L(C(X0)+REG) ← THE CURRENT VALUE OF THE SECOND HALF OF THE PSW
C(X1) ← C(B1)
C(B1) < C(B6) ?
YES
INSTRA
NO
CHANGE

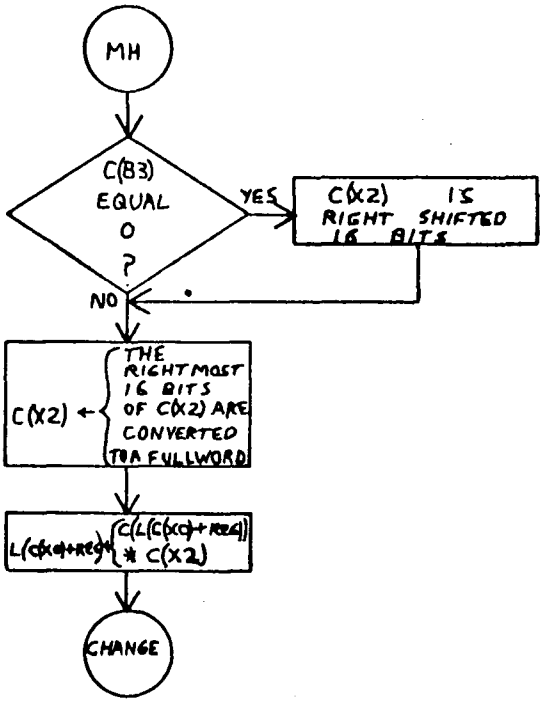

LA
L(C(X0)+REG) ← 24 BIT ADDRESS SPECIFIED BY C(X2)
CHANGE

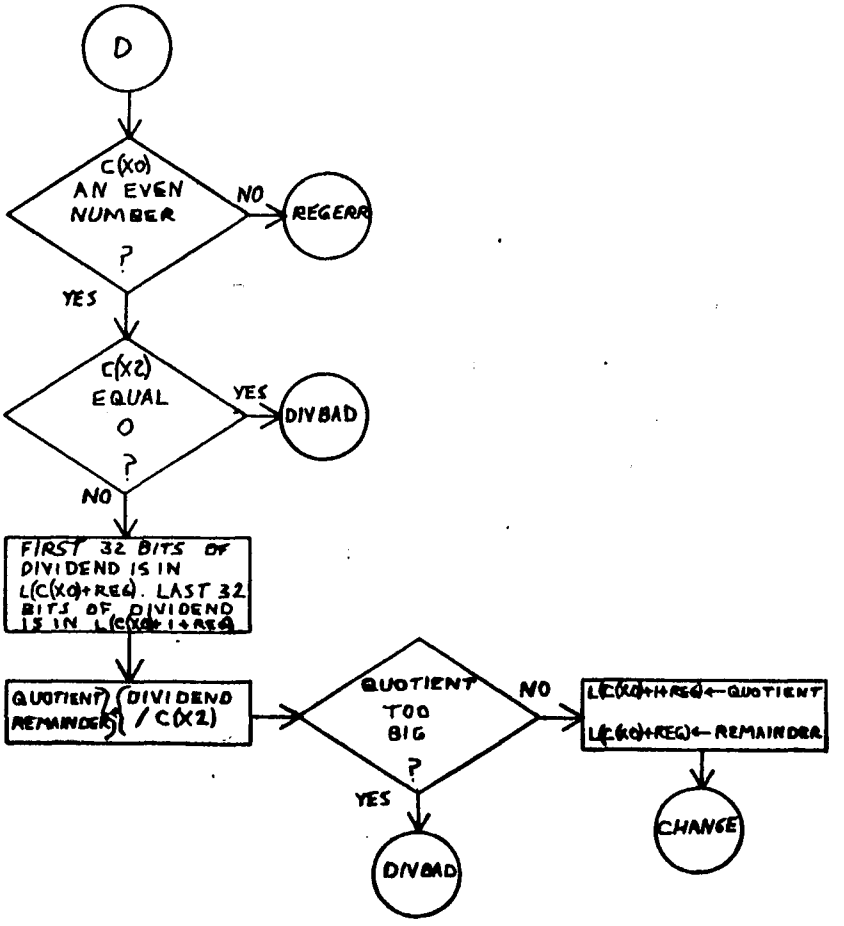

IC
RIGHTMOST EIGHT BITS OF L(C(X0)+REG) ← EIGHT BITS (ONE BYTE) SPECIFIED BY C(B3) AND C(X2)
CHANGE

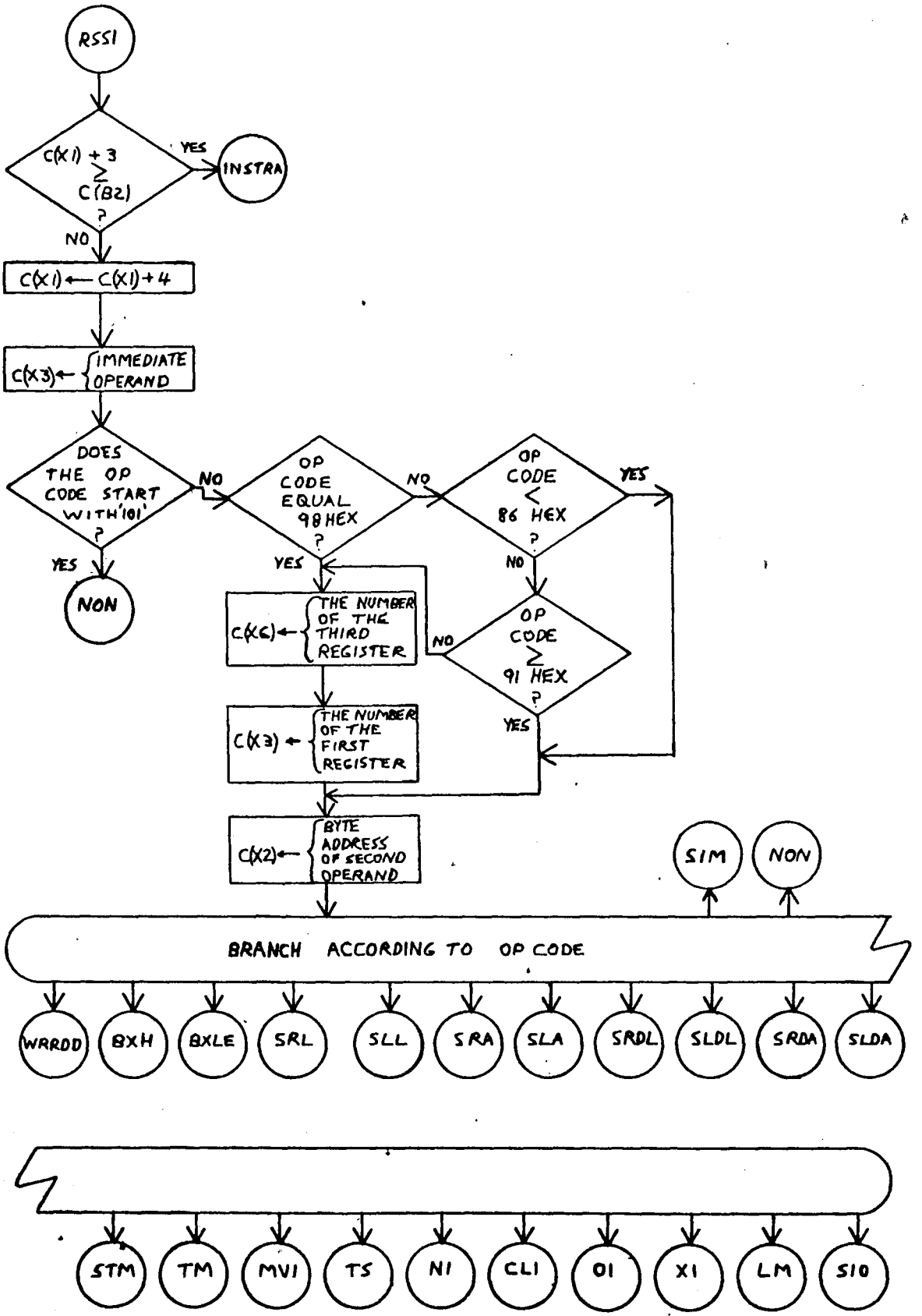

STC
EIGHT BIT LOCATION SPECIFIED BY C(B3) AND C(X2)
RIGHTMOST EIGHT BITS OF L(C(X0)+REG)
CHANGE
EX
C(X3)←
FIRST HALFWORD OF INSTR. LOCATED AT ADDRESS SPECIFIED BY C(B3) AND C(X2)
C(X0) EQUAL 0 ?
NO
C(X3) ←
RIGHTMOST 8 BITS OF L(C(X0)+REG OF C(X3))
YES
OP CODE EQUAL 44 HEX ?
YES
KILL
NO
C(X1) ← C(K)
NUMBER OF BYTES IN INSTRUCTION
C(B5) ←
FIRST TWO BITS OF OP CODE +1
SEP2

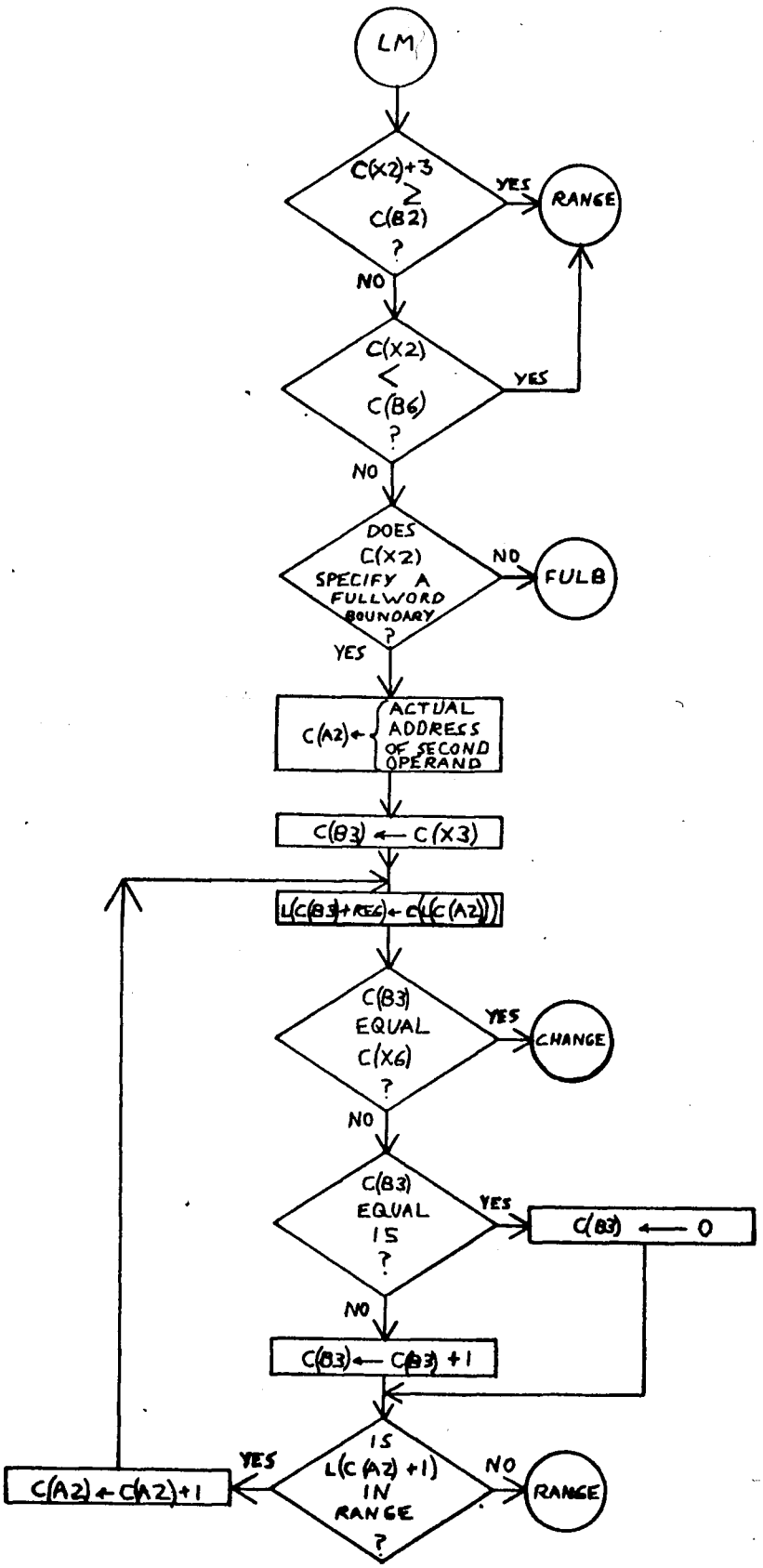

CL
C(X6) ← 2
C(L(C(X0)+REG)) EQUAL C(X2) ?
YES
C(X6) ← 0
NO
C(L(C(X0)+REG)+1) EQUAL C(X2) ?
YES
NO
C(X6) ← 1
L(CCOUNT) ← C(X6)
CHANGE

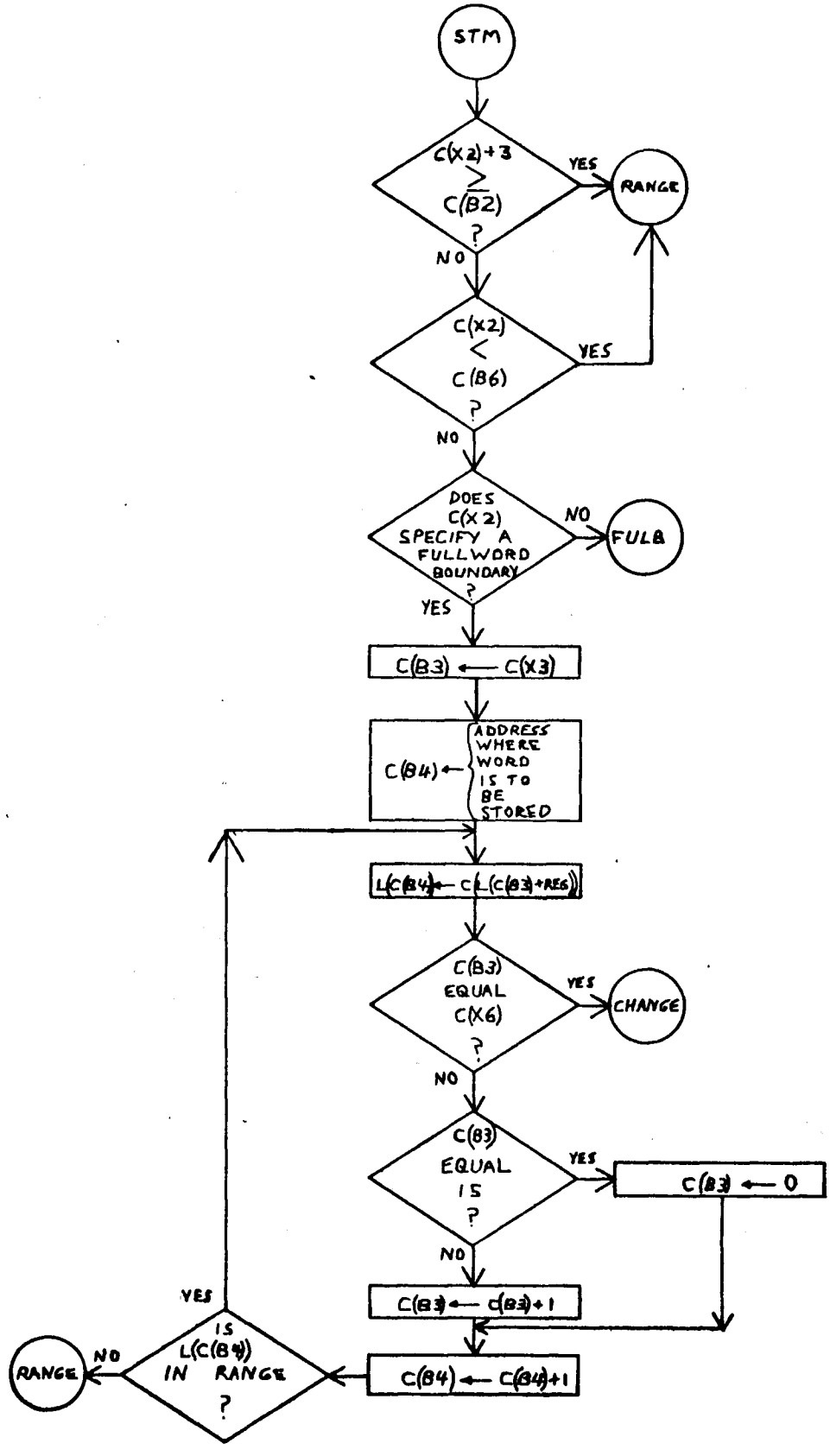

M
C(X0) AN EVEN NUMBER ?
NO
REGERR
YES
L(C(X0)+REG) ← FIRST 32 BITS OF C(L(C(X0)+1+REG)) * C(X2)
L(C(X0)+1+REG) ← LAST 32 BITS OF C(L(C(X0)+1+REG)) * C(X2)
CHANGE

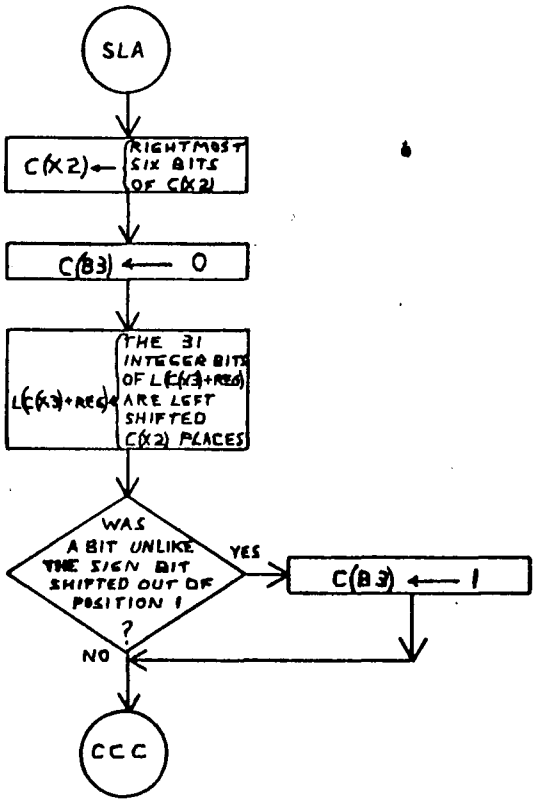

MH
C(B3) EQUAL 0 ?
YES
C(X2) IS RIGHT SHIFTED 16 BITS
NO
C(X2) ←
THE RIGHTMOST 16 BITS OF C(X2) ARE CONVERTED TO A FULLWORD
L(C(X0)+REG)
C(L(C(X0)+REG)) * C(X2)
CHANGE

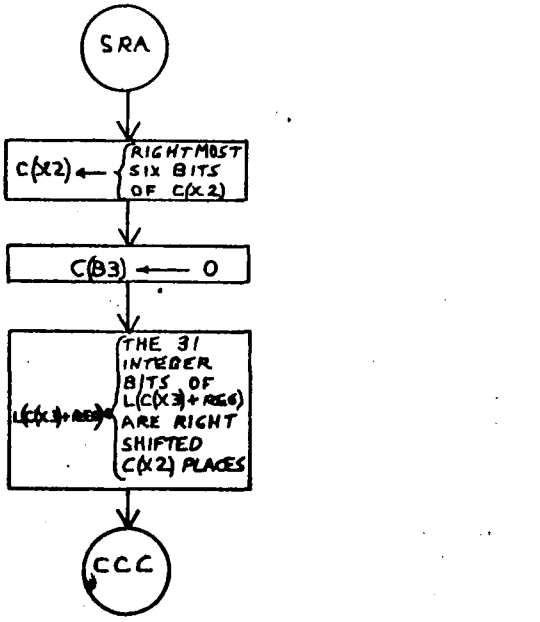

D
C(X0) AN EVEN NUMBER ?
NO
REGERR
YES
C(X2) EQUAL 0 ?
YES
DIVBAD
NO
FIRST 32 BITS OF DIVIDEND IS IN L(C(X0)+REG). LAST 32 BITS OF DIVIDEND IS IN L(C(X0)+1+REG)
QUOTIENT REMAINDER
DIVIDEND / C(X2)
QUOTIENT TOO BIG ?
NO
L(C(X0)+REG) ← QUOTIENT
L(C(X0)+REG) ← REMAINDER
CHANGE
YES
DIVBAD

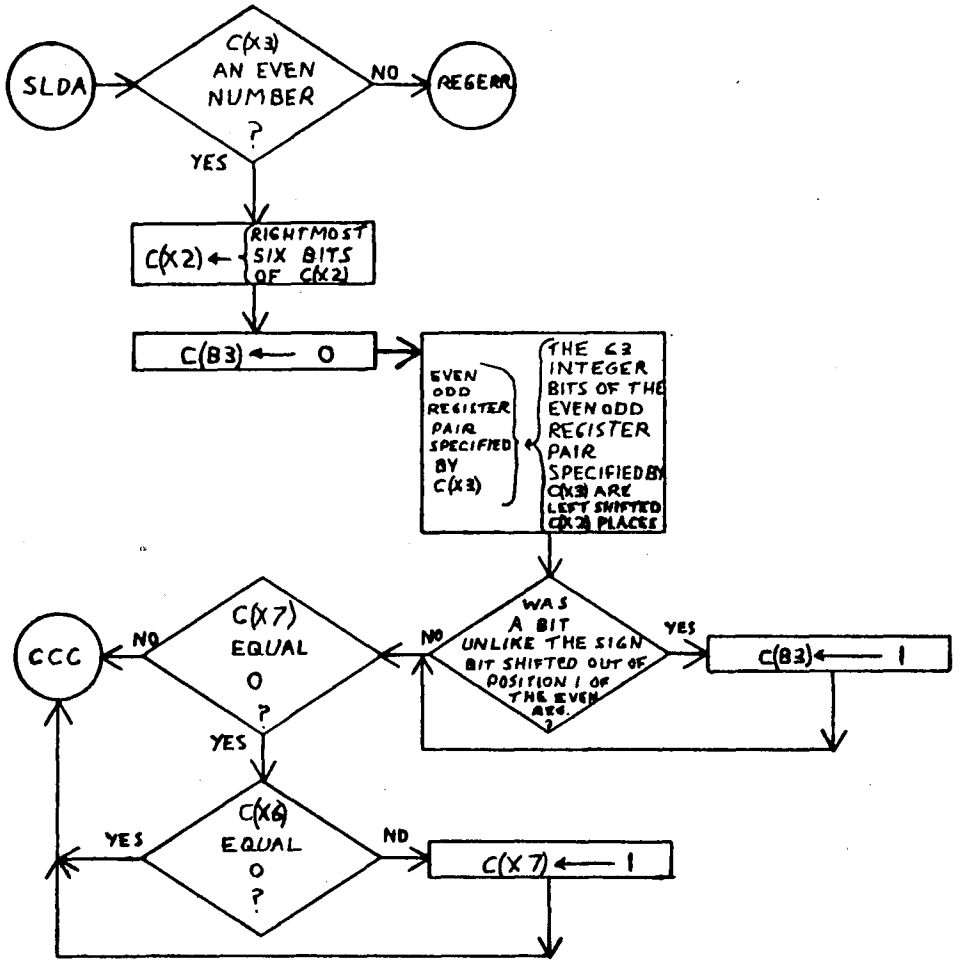

RSSI
C(X1) + 3 ≥ C(B2) ?
YES
INSTRA
NO
C(X1) ← C(X1) + 4
C(X3) ← IMMEDIATE OPERAND
DOES THE OP CODE START WITH '101' ?
NO
YES
NON
OP CODE EQUAL 98 HEX ?
NO
YES
C(XC) ← THE NUMBER OF THE THIRD REGISTER
OP CODE < 86 HEX ?
YES
NO
OP CODE ≥ 91 HEX ?
NO
YES
C(X3) ← THE NUMBER OF THE FIRST REGISTER
C(X2) ← BYTE ADDRESS OF SECOND OPERAND
SIM
NON
BRANCH ACCORDING TO OP CODE
WRRDD
BXH
BXLE
SRL
SLL
SRA
SLA
SRDL
SLDL
SRDA
SLDA
STM
TM
MVI
TS
NI
CLI
OI
XI
LM
SIO

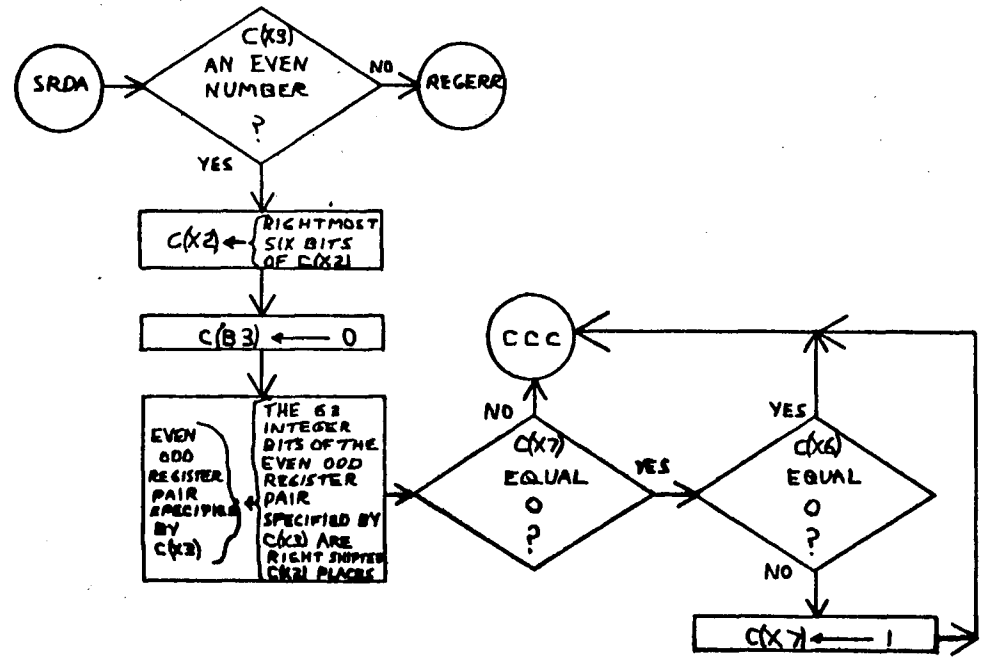

LM
C(X2)+3 ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
DOES C(X2) SPECIFY A FULLWORD BOUNDARY ?
NO
FULB
YES
C(A2) ← ACTUAL ADDRESS OF SECOND OPERAND
C(B3) ← C(X3)
L(C(B3)+REG) ← C(L(C(A2)))
C(B3) EQUAL C(X6) ?
YES
CHANGE
NO
C(B3) EQUAL 15 ?
YES
C(B3) ← 0
NO
C(B3) ← C(B3) +1
IS L(C(A2)+1) IN RANGE ?
YES
C(A2) ← C(A2)+1
NO
RANGE

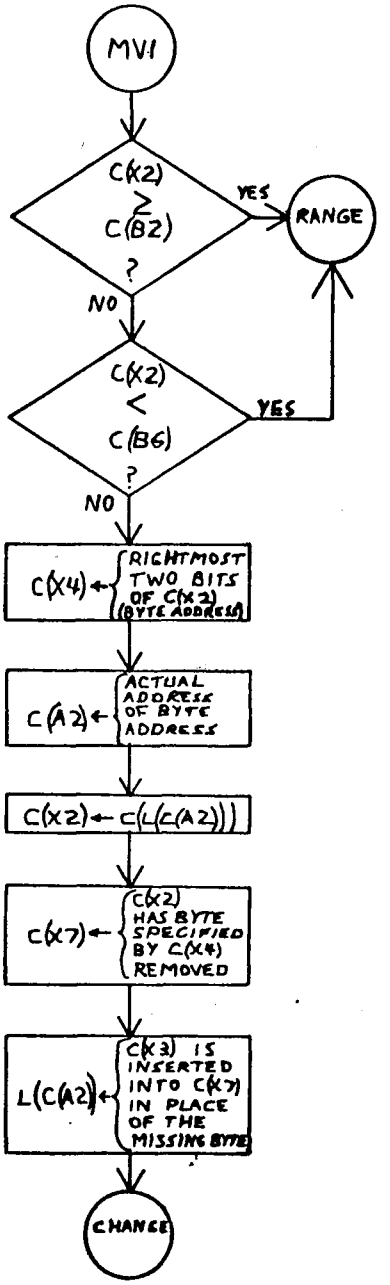

STM
C(X2)+3 ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
DOES C(X2) SPECIFY A FULLWORD BOUNDARY ?
NO
FULB
YES
C(B3) ← C(X3)
C(B4) ← ADDRESS WHERE WORD IS TO BE STORED
L(C(B4)) ← C(C(B3)+REG)
C(B3) EQUAL C(X6) ?
YES
CHANGE
NO
C(B3) EQUAL IS ?
YES
C(B3) ← 0
NO
C(B3) ← C(B3)+1
C(B4) ← C(B4)+1
IS L(C(B4)) IN RANGE ?
YES
NO
RANGE

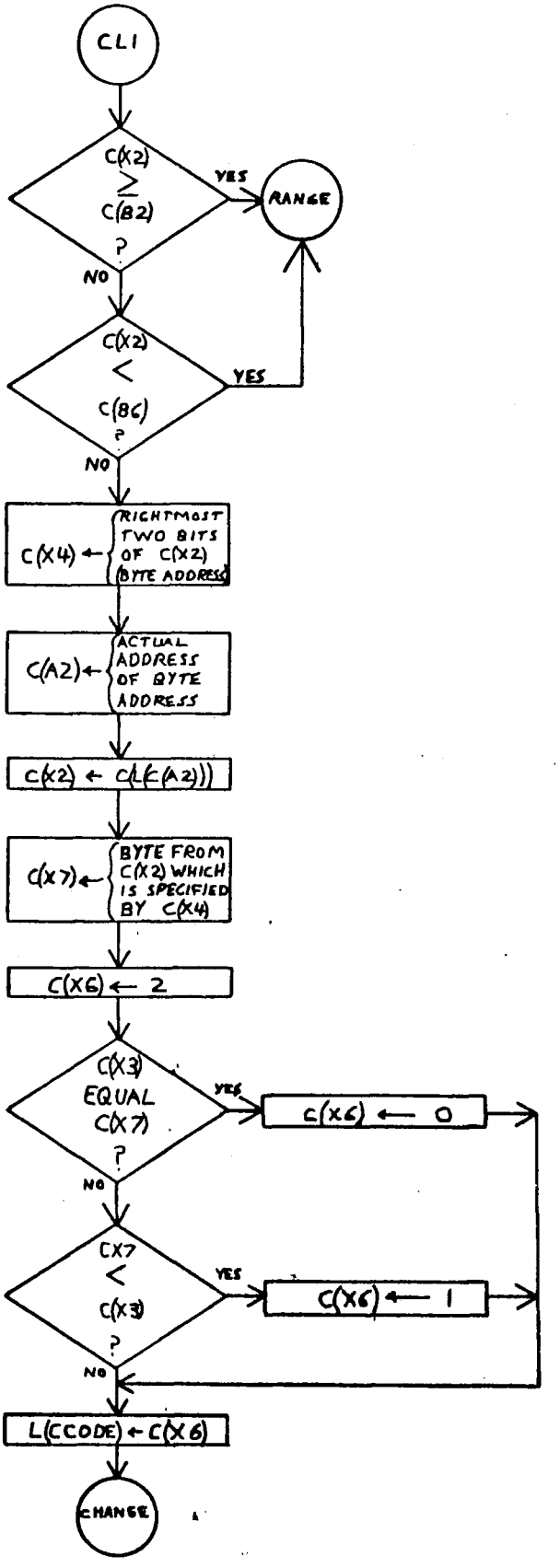

SLA
C(X2) ←
RIGHTMOST SIX BITS OF C(X2)
C(B3) ← 0
L(C(X3)+REG)
THE 31 INTEGER BITS OF L(C(X3)+REG) ARE LEFT SHIFTED C(X2) PLACES
WAS A BIT UNLIKE THE SIGN BIT SHIFTED OUT OF POSITION 1 ?
YES
C(B3) ← 1
NO
CCC

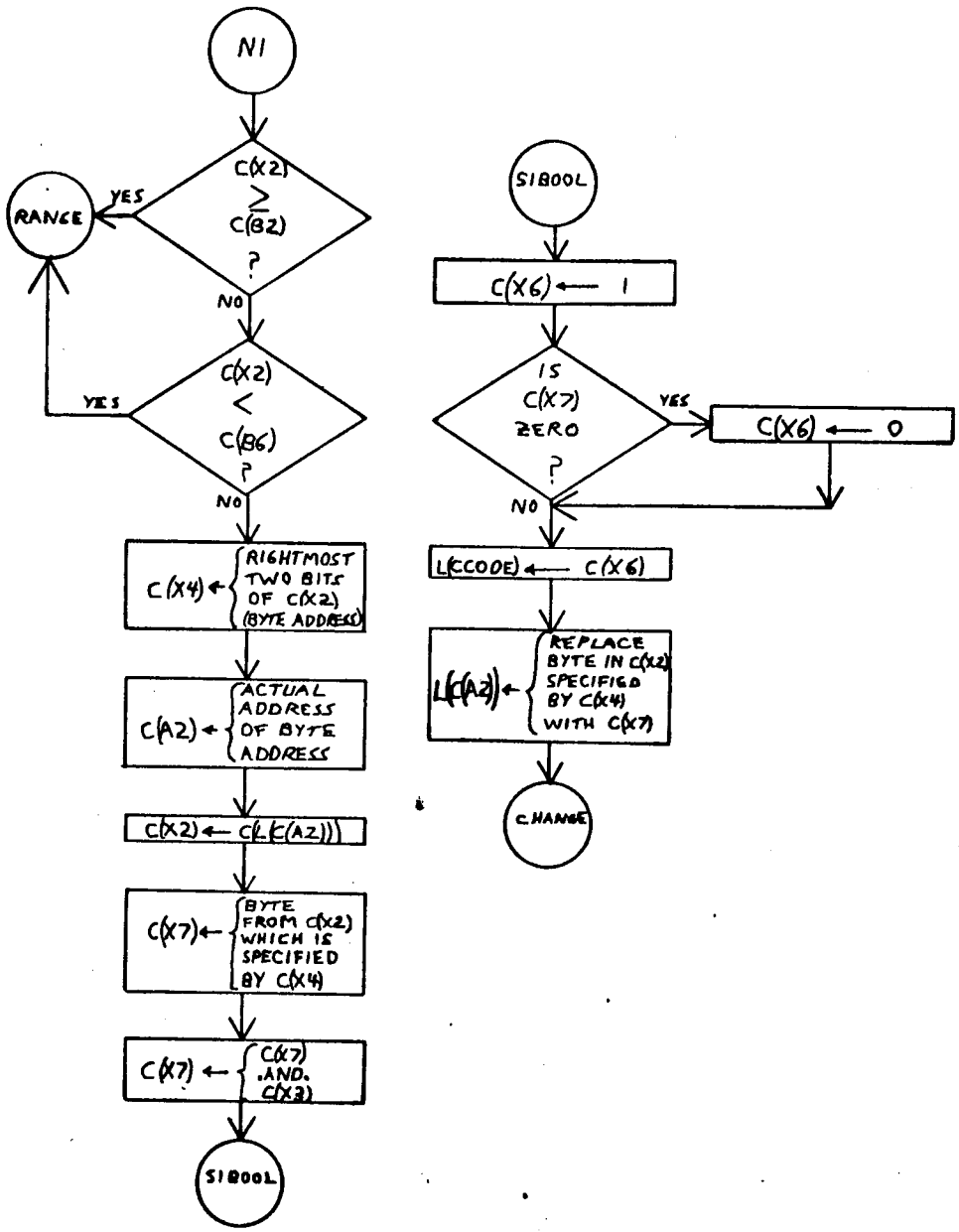

SRA
C(X2) ←
RIGHTMOST SIX BITS OF C(X2)
C(B3) ← 0
L(C(X3)+REG)
THE 31 INTEGER BITS OF L(C(X3)+REG) ARE RIGHT SHIFTED C(X2) PLACES
CCC

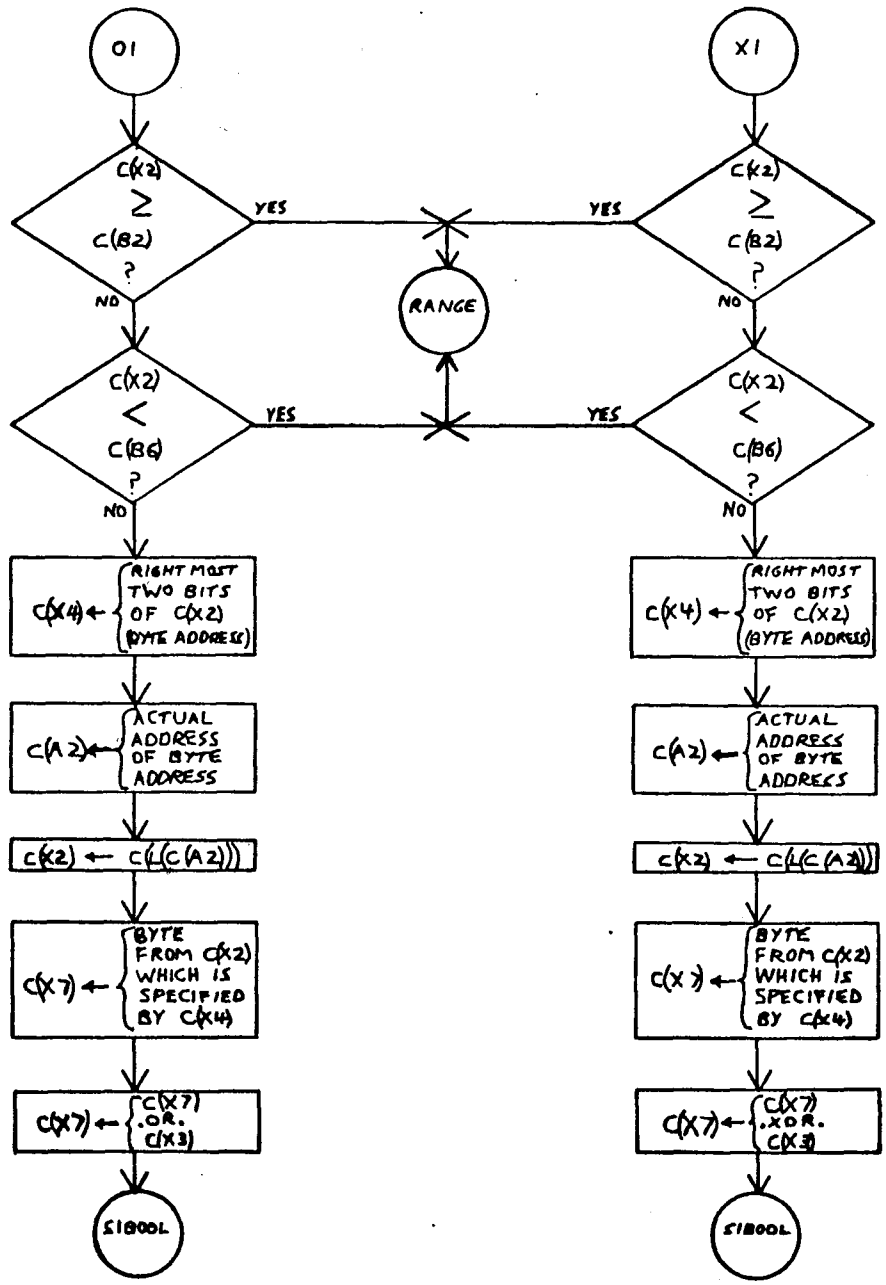

SLDA
C(X3) AN EVEN NUMBER ?
NO
REGERR
YES
C(X2) ← RIGHTMOST SIX BITS OF C(X2)
C(B3) ← 0
EVEN ODD REGISTER PAIR SPECIFIED BY C(X3)
THE 62 INTEGER BITS OF THE EVEN ODD REGISTER PAIR SPECIFIED BY C(X3) ARE LEFT SHIFTED C(X2) PLACES
WAS A BIT UNLIKE THE SIGN BIT SHIFTED OUT OF POSITION 1 OF THE EVEN REG. ?
NO
YES
C(B3) ← 1
C(X7) EQUAL 0 ?
NO
CCC
YES
C(X6) EQUAL 0 ?
YES
NO
C(X7) ← 1

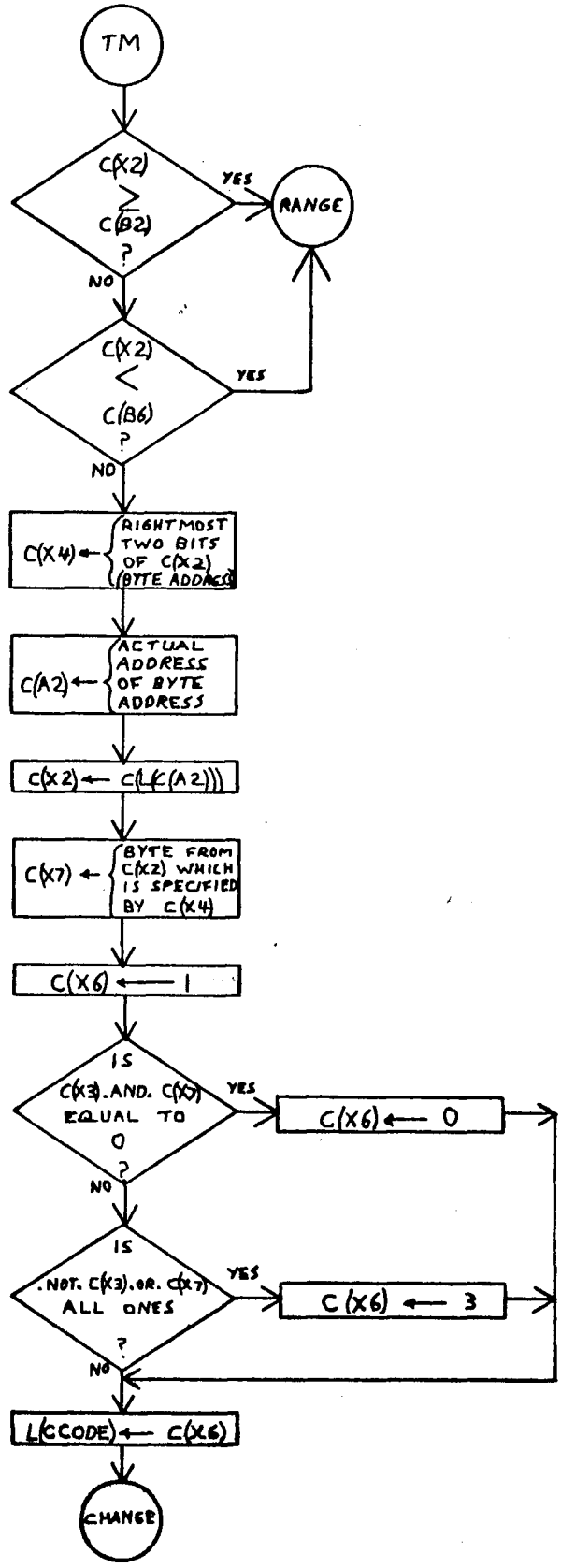

SRDA
C(X3) AN EVEN NUMBER ?
NO
REGERR
YES
C(X2) ← RIGHTMOST SIX BITS OF C(X2)
C(B3) ← 0
EVEN ODD REGISTER PAIR SPECIFIED BY C(X3)
THE 62 INTEGER BITS OF THE EVEN ODD REGISTER PAIR SPECIFIED BY C(X3) ARE RIGHT SHIFTED C(X2) PLACES
CCC
NO
C(X7) EQUAL 0 ?
YES
YES
C(X6) EQUAL 0 ?
NO
C(X7) ← 1

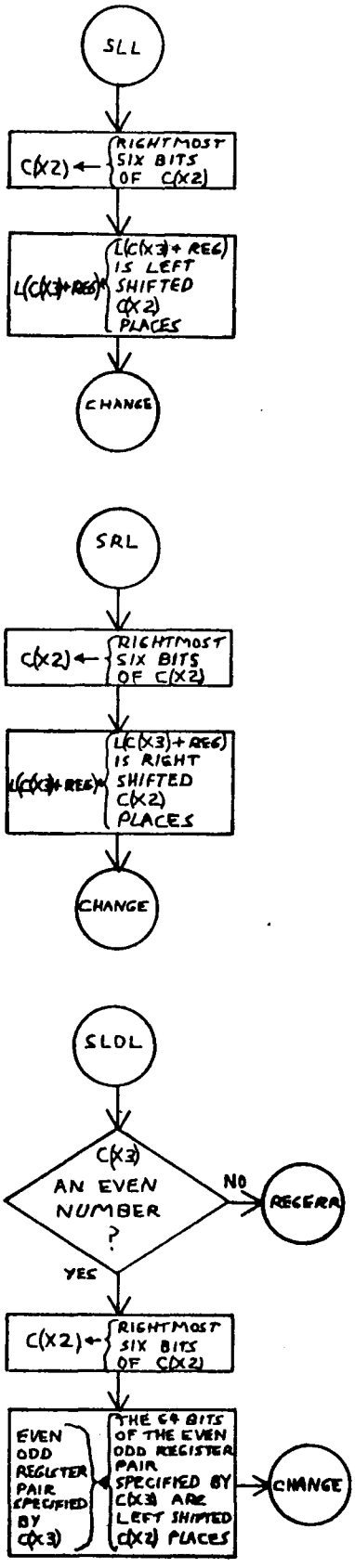

MVI
C(X2) ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
C(X4) ← RIGHTMOST TWO BITS OF C(X2) (BYTE ADDRESS)
C(A2) ← ACTUAL ADDRESS OF BYTE ADDRESS
C(X2) ← C(L(C(A2)))
C(X7) ← C(X2) HAS BYTE SPECIFIED BY C(X4) REMOVED
L(C(A2)) ← C(X3) IS INSERTED INTO C(X7) IN PLACE OF THE MISSING BYTE
CHANGE

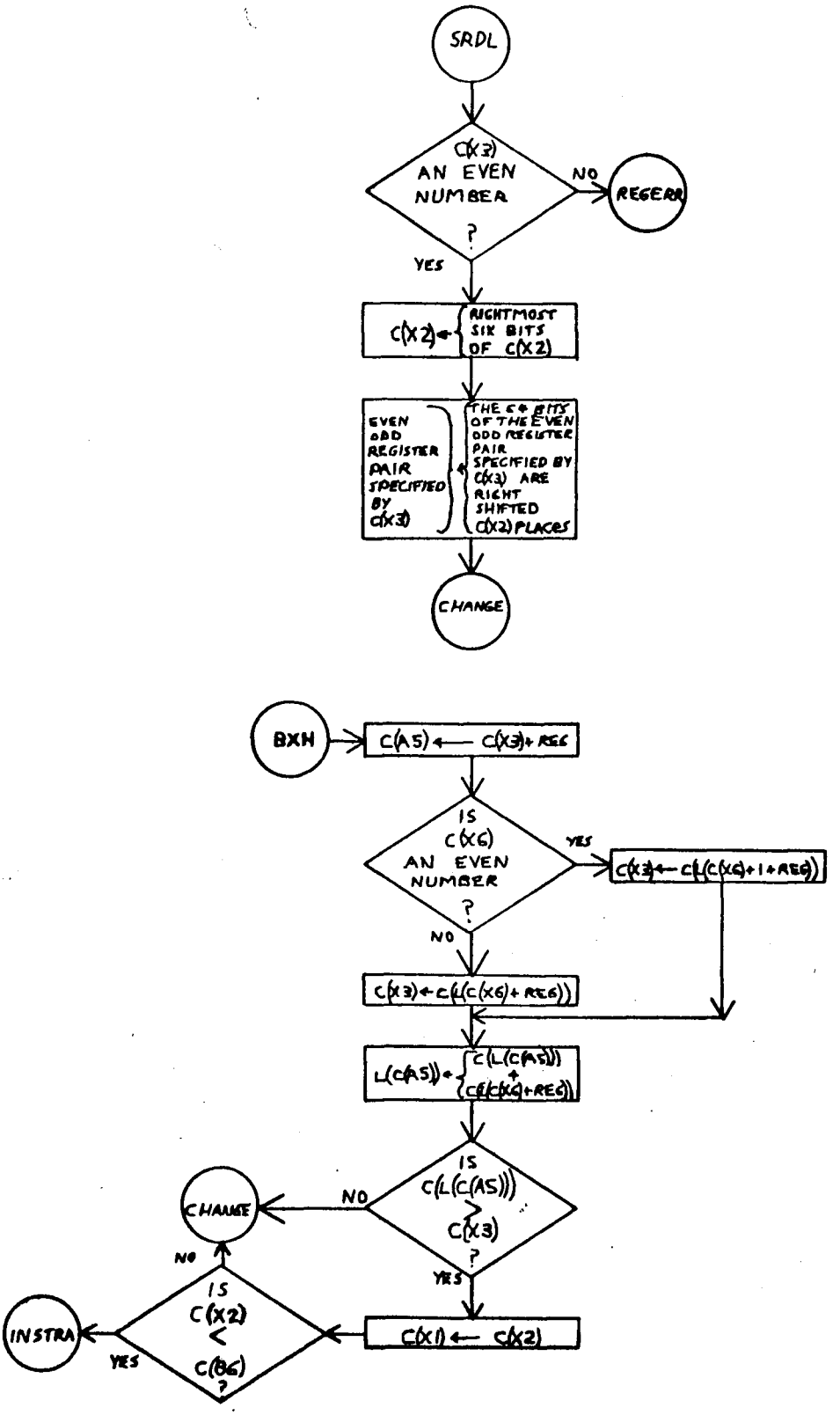

CL1
C(X2) ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
C(X4) ← RIGHTMOST TWO BITS OF C(X2) (BYTE ADDRESS)
C(A2) ← ACTUAL ADDRESS OF BYTE ADDRESS
C(X2) ← C(L(C(A2)))
C(X7) ← BYTE FROM C(X2) WHICH IS SPECIFIED BY C(X4)
C(X6) ← 2
C(X3) EQUAL C(X7) ?
YES
C(X6) ← 0
NO
CX7 < C(X3) ?
YES
C(X6) ← 1
NO
L(CCODE) ← C(X6)
CHANGE

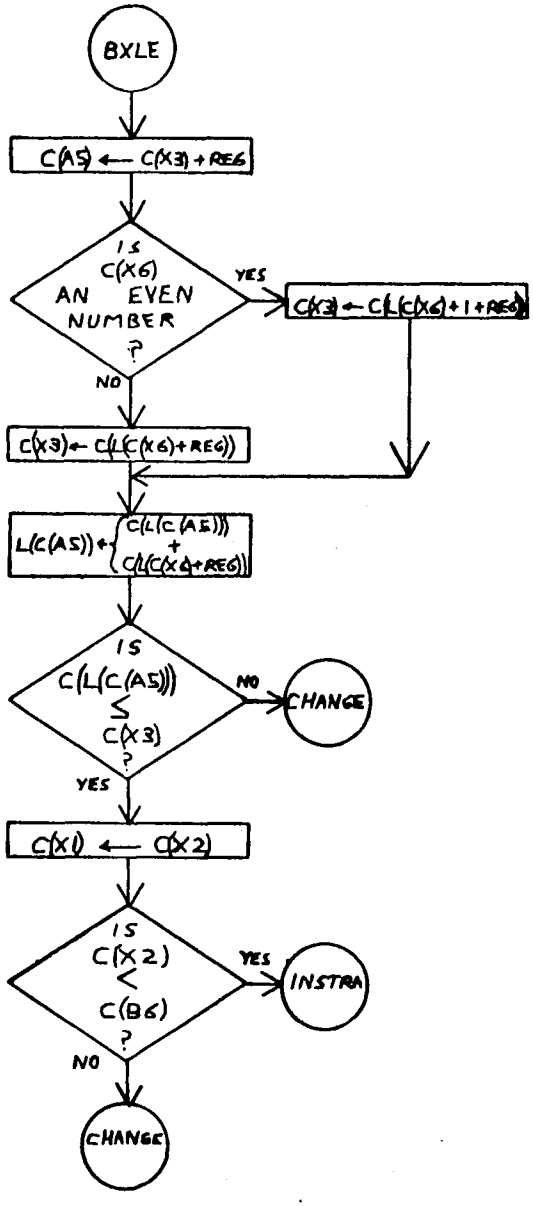

NI
C(X2) ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
C(X4) ← RIGHTMOST TWO BITS OF C(X2) (BYTE ADDRESS)
C(A2) ← ACTUAL ADDRESS OF BYTE ADDRESS
C(X2) ← C(C(A2))
C(X7) ← BYTE FROM C(X2) WHICH IS SPECIFIED BY C(X4)
C(X7) ← C(X7) .AND. C(X3)
SIBOOL
SIBOOL
C(X6) ← 1
IS C(X7) ZERO ?
YES
C(X6) ← 0
NO
L(CODE) ← C(X6)
L(C(A2)) ← REPLACE BYTE IN C(X2) SPECIFIED BY C(X4) WITH C(X7)
CHANGE

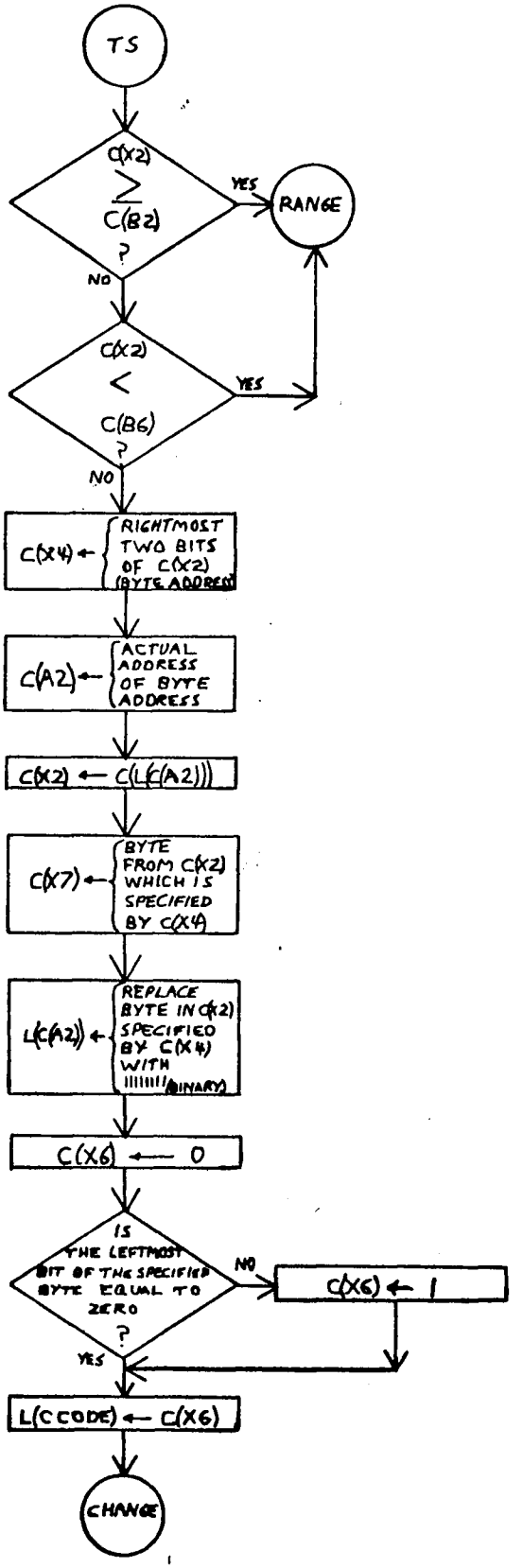

01
X1
C(X2) ≥ C(B2) ?
YES
NO
RANGE
C(X2) < C(B6) ?
YES
NO
C(X4) ← RIGHT MOST TWO BITS OF C(X2) (BYTE ADDRESS)
C(A2) ← ACTUAL ADDRESS OF BYTE ADDRESS
C(X2) ← C(L(C(A2)))
C(X7) ← BYTE FROM C(X2) WHICH IS SPECIFIED BY C(X4)
C(X7) ← C(X7) .OR. C(X3)
C(X7) ← C(X7) .XOR. C(X3)
SIBOOL

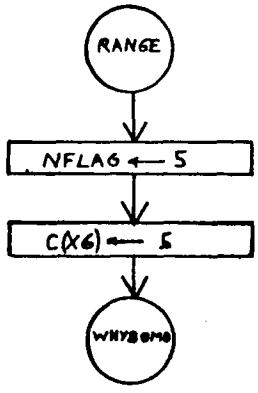

TM
C(X2) ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
C(X4) ← RIGHTMOST TWO BITS OF C(X2) (BYTE ADDRESS)
C(A2) ← ACTUAL ADDRESS OF BYTE ADDRESS
C(X2) ← C(L(C(A2)))
C(X7) ← BYTE FROM C(X2) WHICH IS SPECIFIED BY C(X4)
C(X6) ← 1
IS C(X3).AND. C(X7) EQUAL TO 0 ?
YES
C(X6) ← 0
NO
IS .NOT. C(X3).OR. C(X7) ALL ONES ?
YES
C(X6) ← 3
NO
L(CODE) ← C(X6)
CHANGE

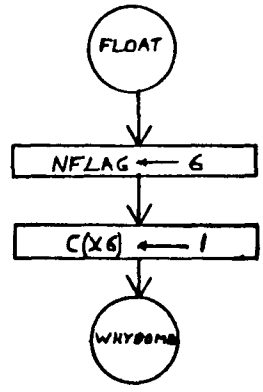

SLL
C(X2) ← RIGHTMOST SIX BITS OF C(X2)
L(C(X3)+REG) IS LEFT SHIFTED C(X2) PLACES
CHANGE
SRL
C(X2) ← RIGHTMOST SIX BITS OF C(X2)
L(C(X3)+REG) IS RIGHT SHIFTED C(X2) PLACES
CHANGE
SLDL
C(X3) AN EVEN NUMBER ?
NO
REGERR
YES
C(X2) ← RIGHTMOST SIX BITS OF C(X2)
EVEN ODD REGISTER PAIR SPECIFIED BY C(X3)
THE 64 BITS OF THE EVEN ODD REGISTER PAIR SPECIFIED BY C(X3) ARE LEFT SHIFTED C(X2) PLACES
CHANGE

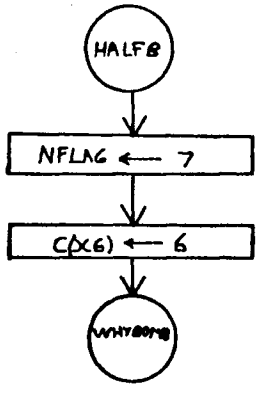

SRDL
C(X3) AN EVEN NUMBER ?
NO
REGERR
YES
C(X2) ← RIGHTMOST SIX BITS OF C(X2)
EVEN ODD REGISTER PAIR SPECIFIED BY C(X3)
THE C# BITS OF THE EVEN ODD REGISTER PAIR SPECIFIED BY C(X3) ARE RIGHT SHIFTED C(X2) PLACES
CHANGE
BXH
C(A5) ← C(X3)+REG
IS C(X6) AN EVEN NUMBER ?
YES
C(X3) ← C(L(C(X6)+1+REG))
NO
C(X3) ← C(L(C(X6)+REG))
L(C(A5)) ← C(L(C(A5))) + C(L(C(X6)+REG))
IS C(L(C(A5))) ≥ C(X3) ?
NO
CHANGE
YES
C(X1) ← C(X2)
NO
IS C(X2) < C(B6) ?
YES
INSTRA

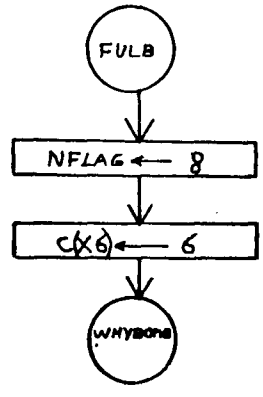

BXLE
C(A5) ← C(X3) + REG
IS C(X6) AN EVEN NUMBER ?
YES
C(X3) ← C(L(C(X6)+1+REG))
NO
C(X3) ← C(L(C(X6)+REG))
L(C(A5)) ← C(L(C(A5))) + C(L(C(X4)+REG))
IS C(L(C(A5))) ≤ C(X3) ?
NO
CHANGE
YES
C(X1) ← C(X2)
IS C(X2) < C(B6) ?
YES
INSTRA
NO
CHANGE

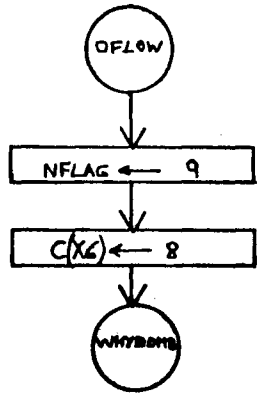

TS
C(X2) ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
C(X4) ← RIGHTMOST TWO BITS OF C(X2) (BYTE ADDRESS)
C(A2) ← ACTUAL ADDRESS OF BYTE ADDRESS
C(X2) ← C(L(C(A2)))
C(X7) ← BYTE FROM C(X2) WHICH IS SPECIFIED BY C(X4)
L(C(A2)) ← REPLACE BYTE IN C(X2) SPECIFIED BY C(X4) WITH !!!!!!!! (BINARY)
C(X6) ← 0
IS THE LEFTMOST BIT OF THE SPECIFIED BYTE EQUAL TO ZERO ?
NO
C(X6) ← 1
YES
L(CCODE) ← C(X6)
CHANGE

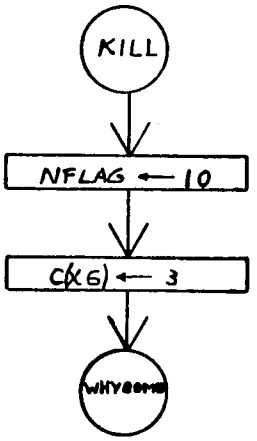

RANGE
NFLAG ← 5
C(X6) ← 5
WHYBOMB

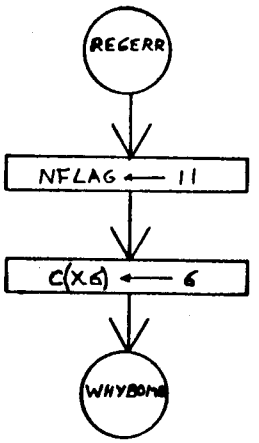

FLOAT
NFLAG ← 6
C(X6) ← 1
WHYBOMB

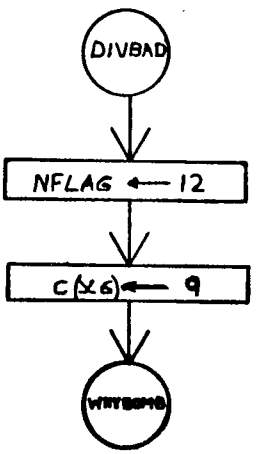

HALFB
NFLAG ← 7
C(X6) ← 6
WHYBOMB

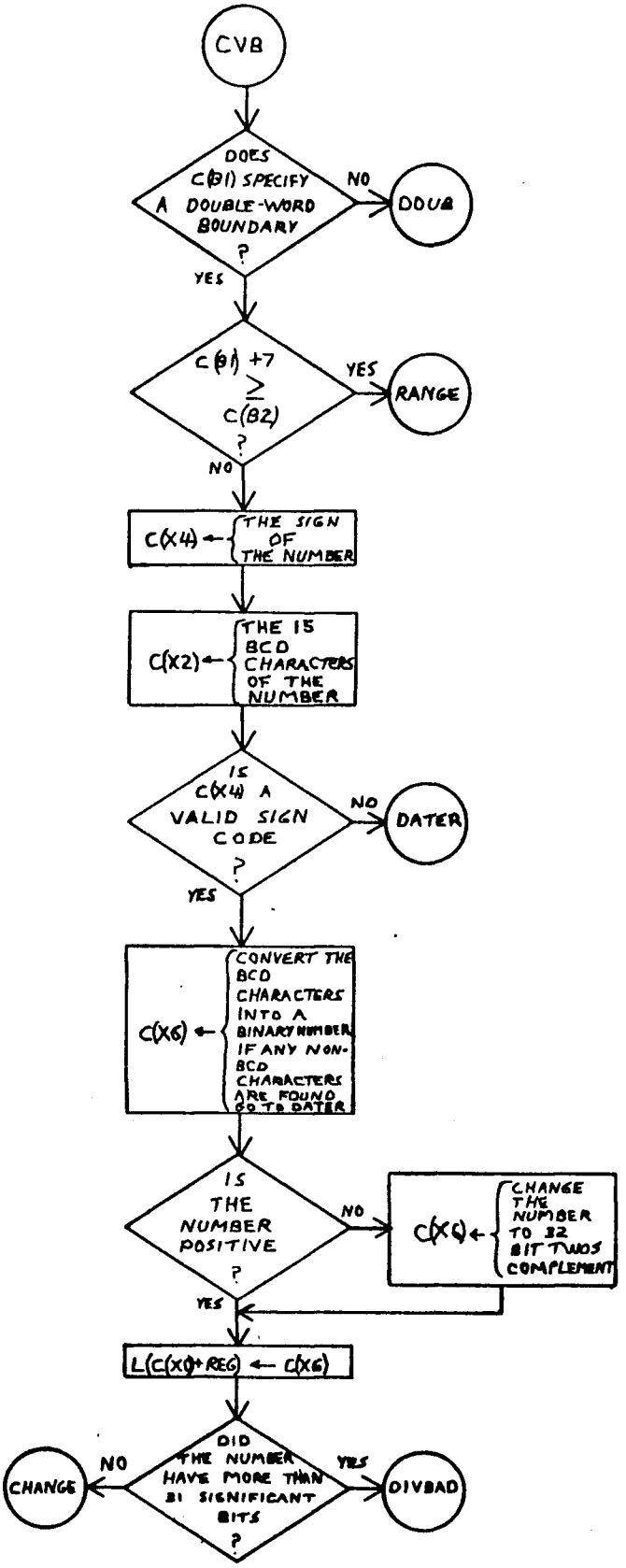

FULB
NFLAG ← 8
C(X6) ← 6
WHYBOMB

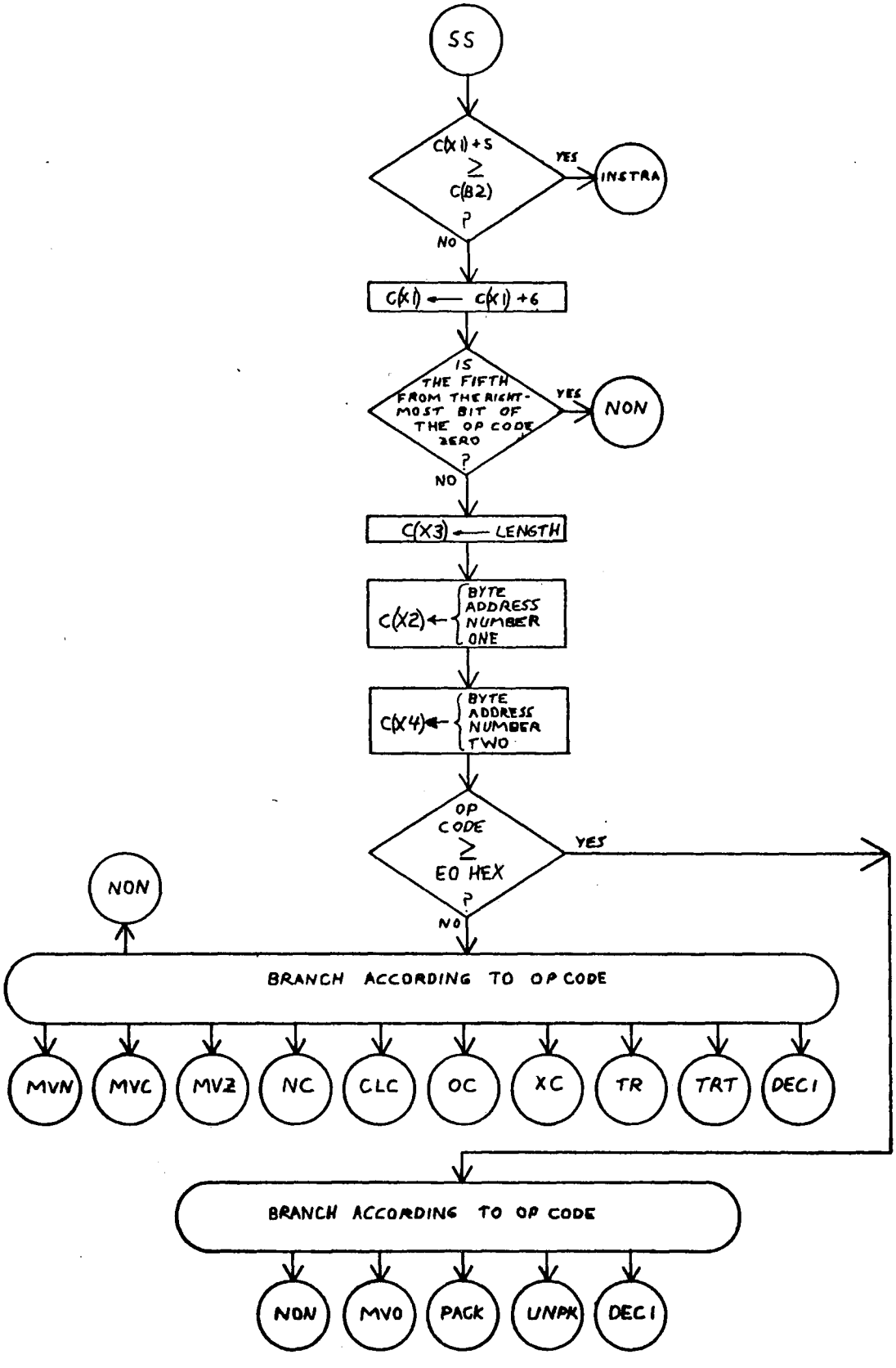

OFLOW
NFLAG ← 9
C(X6) ← 8
WHYBOMB

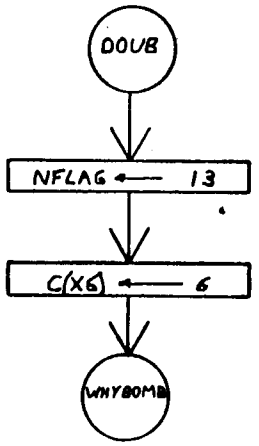

KILL
NFLAG ← 10
C(X6) ← 3

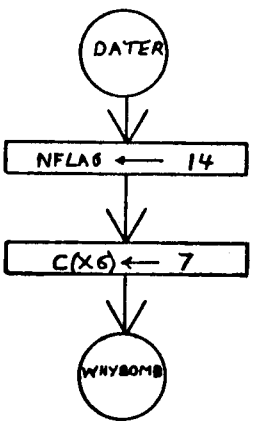

REGERR
NFLAG ← 11
C(X6) ← 6

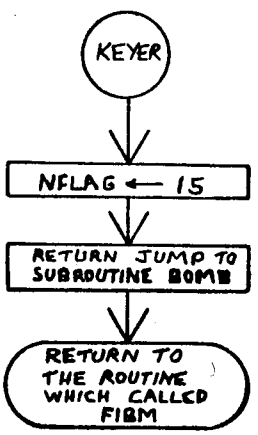

DIVBAD
NFLAG ← 12
C(X6) ← 9

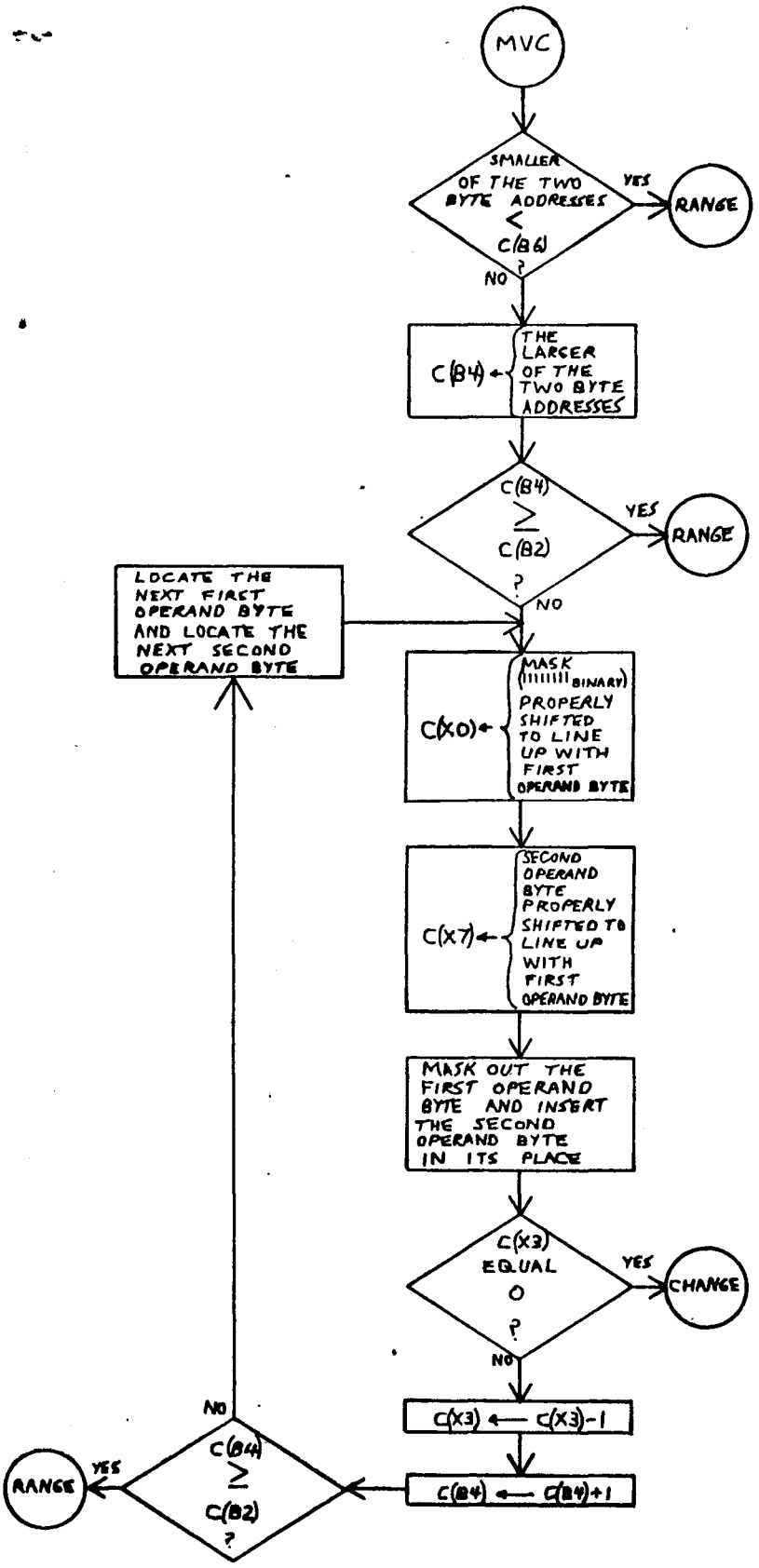

CVB
DOES C(B1) SPECIFY A DOUBLE-WORD BOUNDARY ?
NO
DOUB
YES
C(B1) +7 ≥ C(B2) ?
YES
RANGE
NO
C(X4) ← THE SIGN OF THE NUMBER
C(X2) ← THE 15 BCD CHARACTERS OF THE NUMBER
IS C(X4) A VALID SIGN CODE ?
NO
DATER
YES
C(X6) ← CONVERT THE BCD CHARACTERS INTO A BINARY NUMBER. IF ANY NON-BCD CHARACTERS ARE FOUND GO TO DATER
IS THE NUMBER POSITIVE ?
NO
C(X6) ← CHANGE THE NUMBER TO 32 BIT TWOS COMPLEMENT
YES
L(C(X0)+REG) ← C(X6)
DID THE NUMBER HAVE MORE THAN 31 SIGNIFICANT BITS ?
NO
CHANGE
YES
DIVBAD

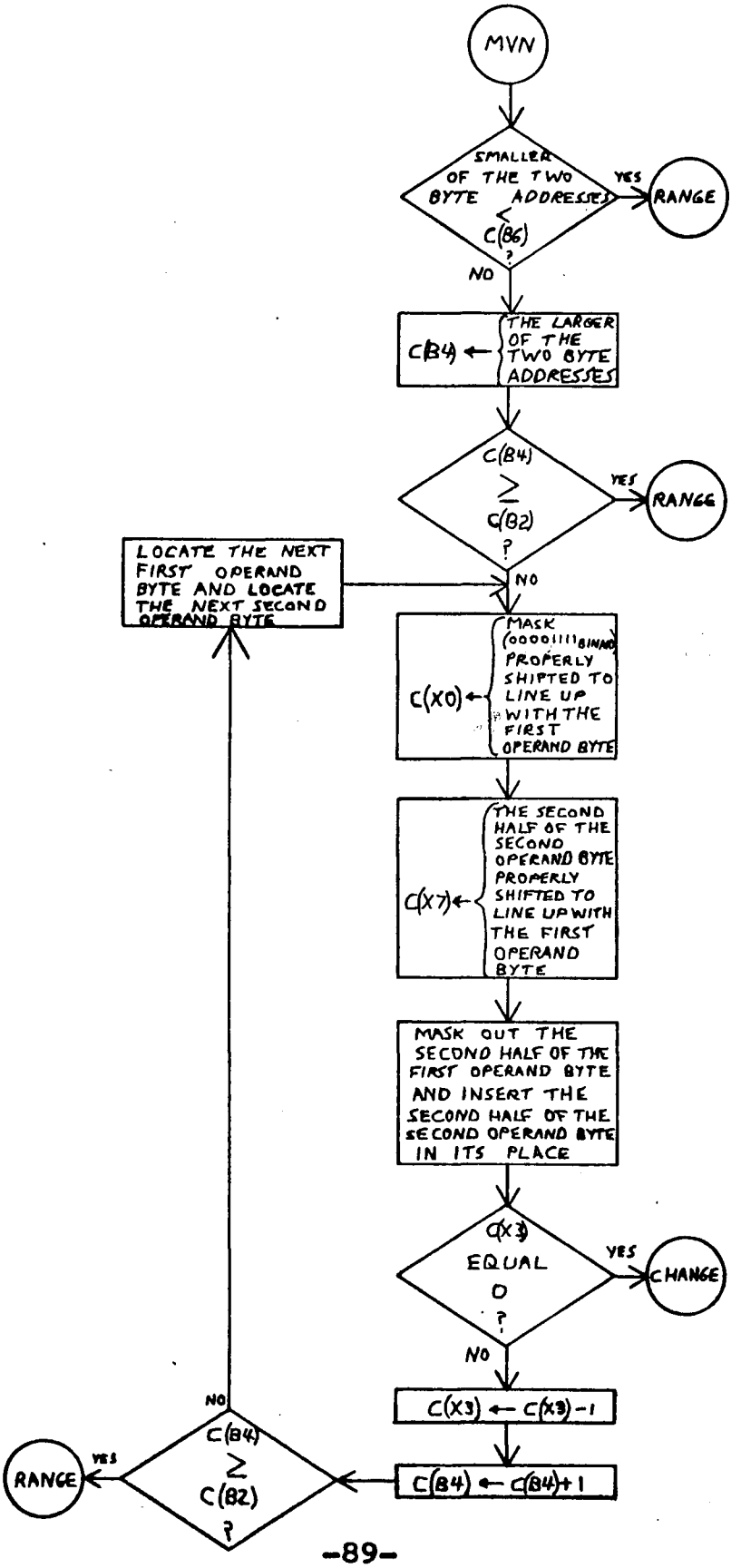

SS
C(X1)+S ≥ C(B2) ?
YES
INSTRA
NO
C(X1) ← C(X1)+6
IS THE FIFTH FROM THE RIGHT-MOST BIT OF THE OP CODE ZERO ?
YES
NON
NO
C(X3) ← LENGTH
C(X2) ← BYTE ADDRESS NUMBER ONE
C(X4) ← BYTE ADDRESS NUMBER TWO
OP CODE ≥ E0 HEX ?
YES
NO
NON
BRANCH ACCORDING TO OP CODE
MVN
MVC
MVZ
NC
CLC
OC
XC
TR
TRT
DEC1
BRANCH ACCORDING TO OP CODE
NON
MVO
PACK
UNPK
DEC1

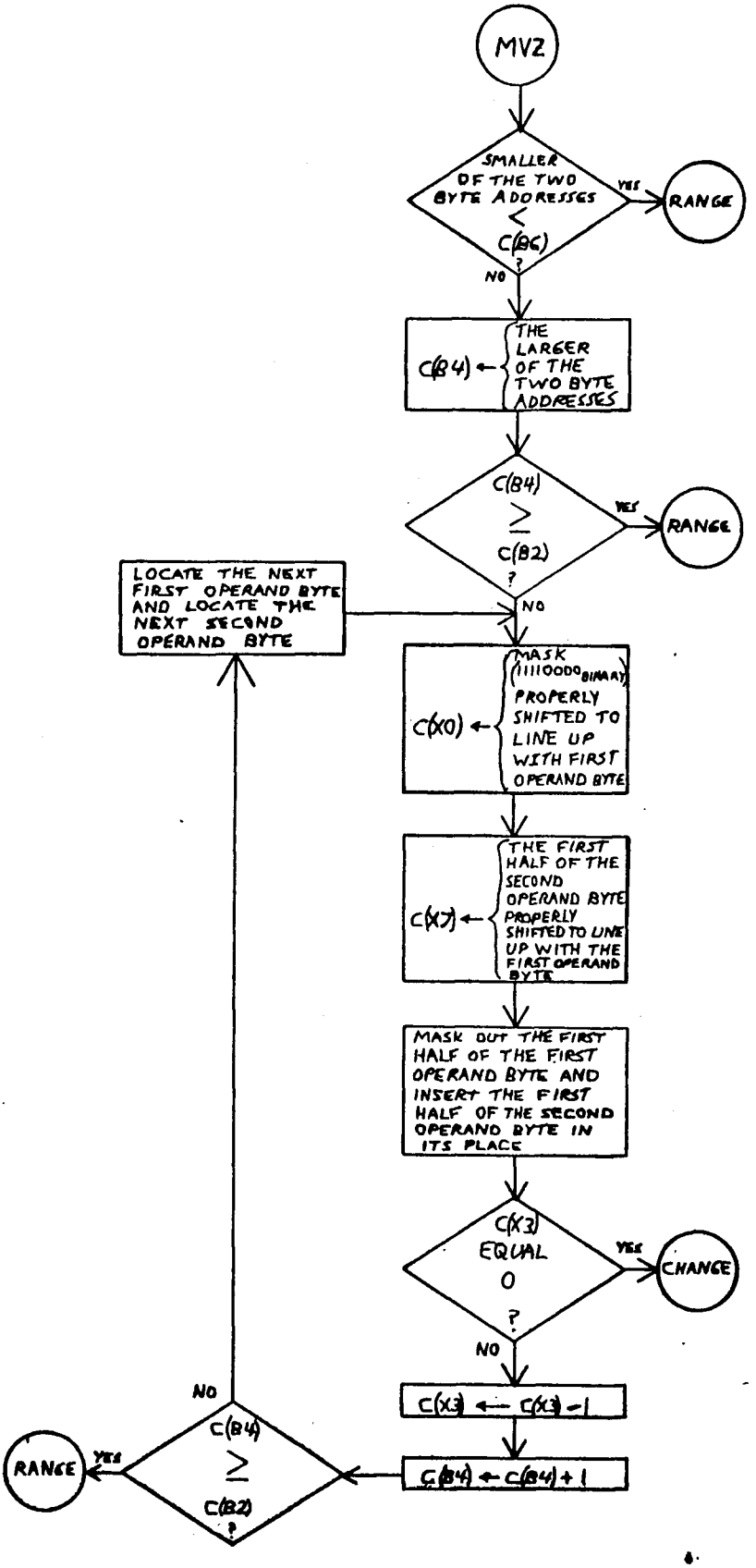

DOUB
NFLAG ← 13
C(X6) ← 6
WHYBOMB

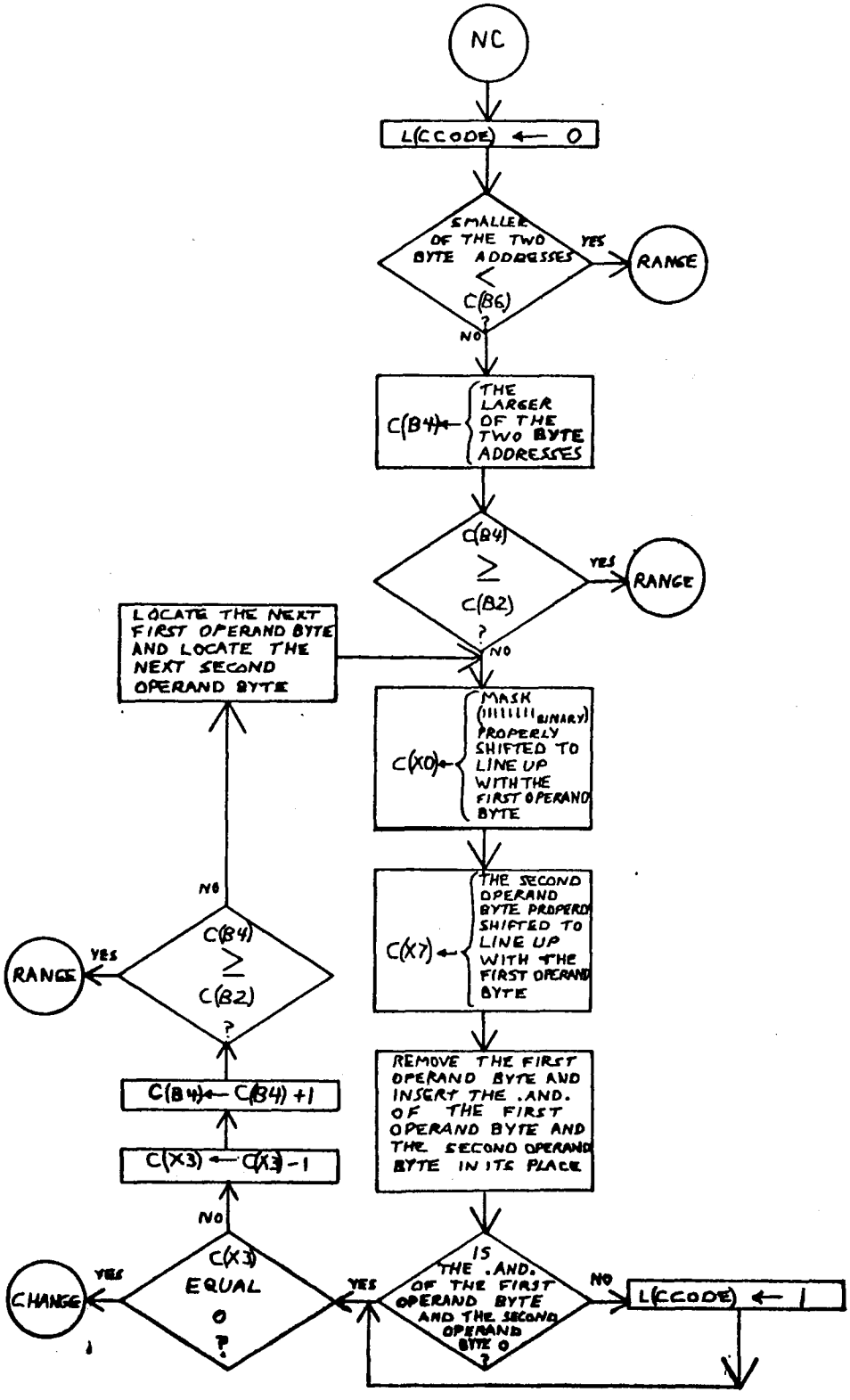

DATER
NFLAG ← 14
C(X6) ← 7
WHYBOMB

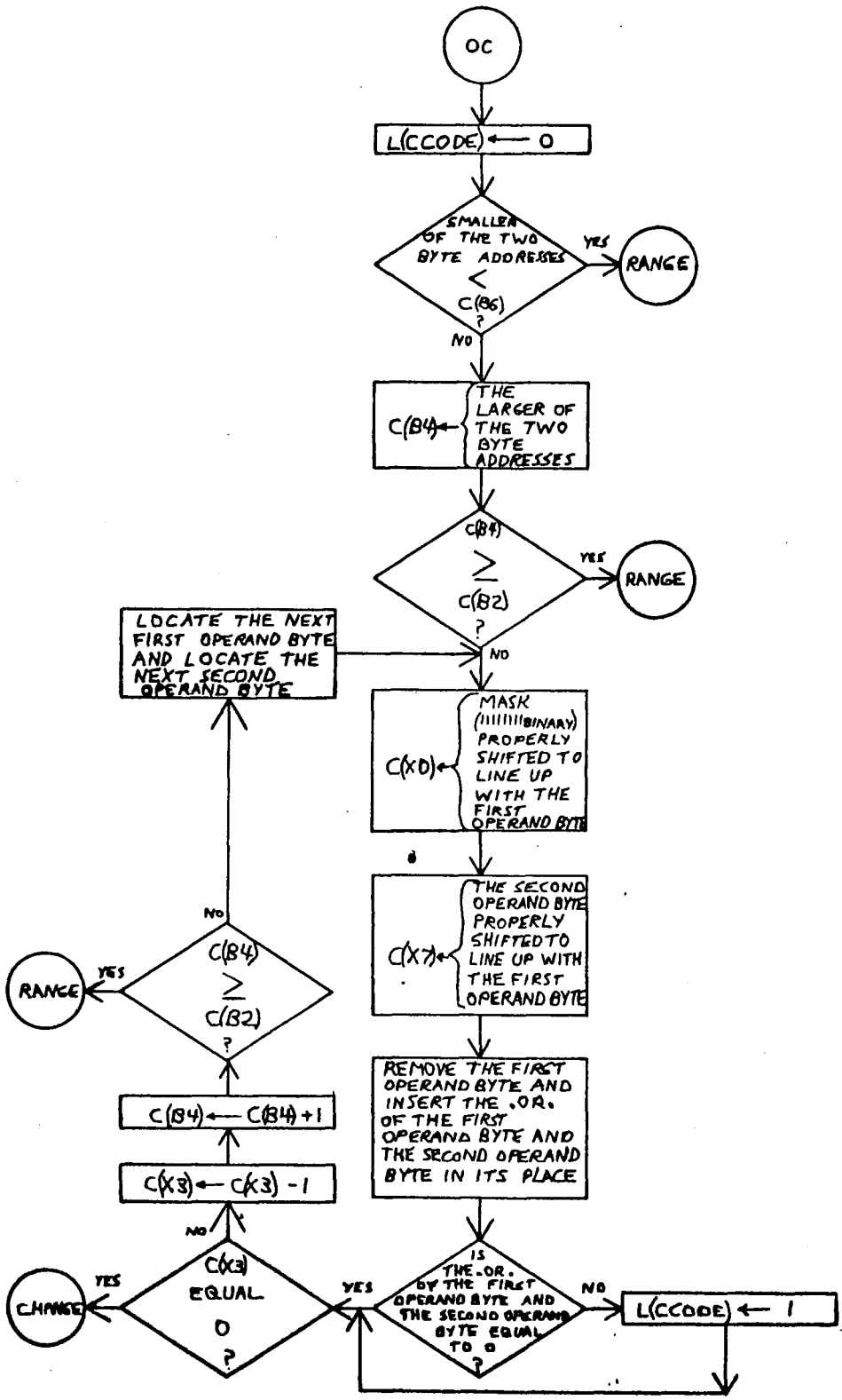

KEYER
NFLAG ← 15
RETURN JUMP TO SUBROUTINE BOMB
RETURN TO THE ROUTINE WHICH CALLED FIRM

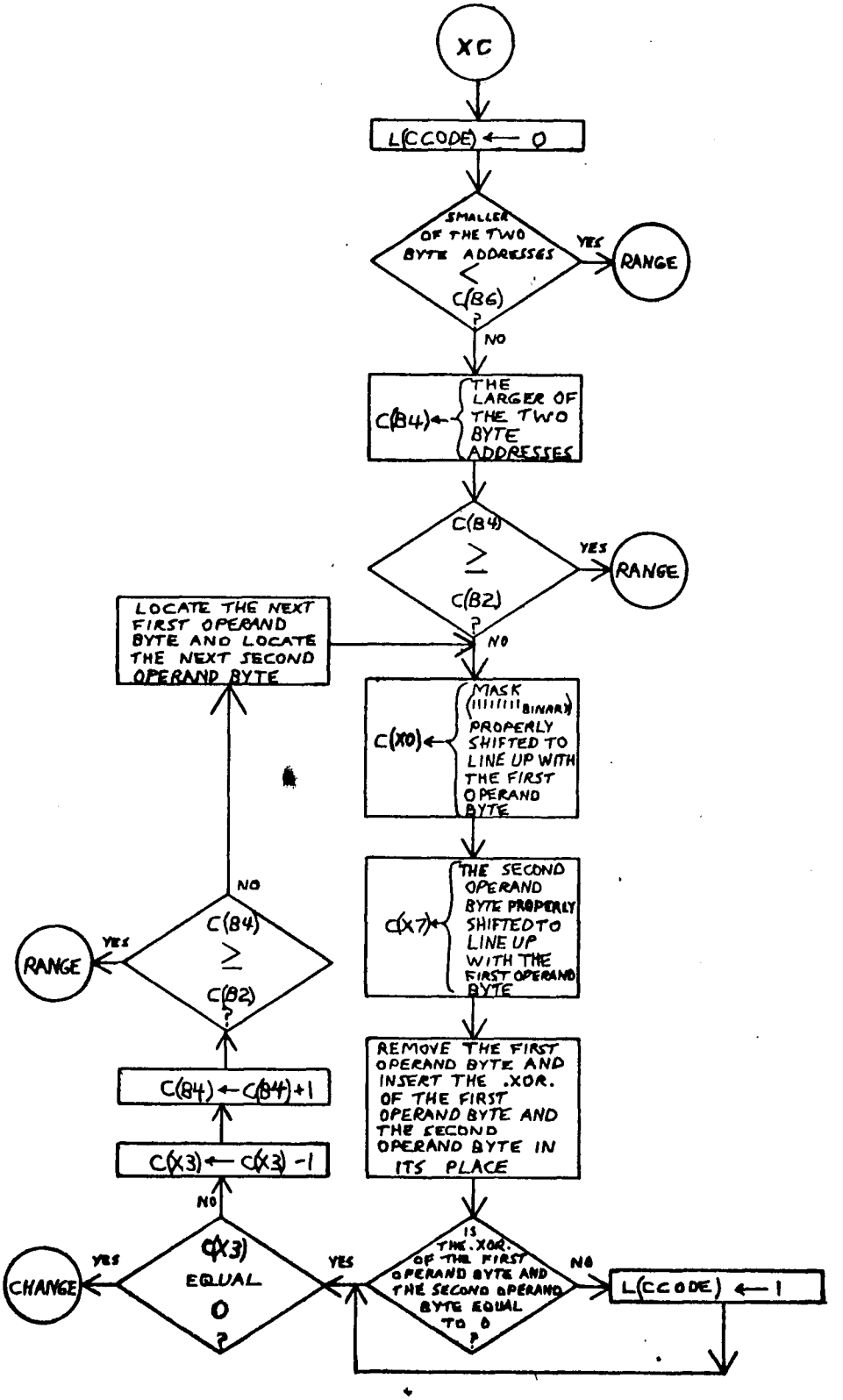

MVC
SMALLER OF THE TWO BYTE ADDRESSES < C(B6) ?
YES
RANGE
NO
C(B4) ← THE LARGER OF THE TWO BYTE ADDRESSES
C(B4) ≥ C(B2) ?
YES
RANGE
NO
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE
C(X0) ← MASK (11111111 BINARY) PROPERLY SHIFTED TO LINE UP WITH FIRST OPERAND BYTE
C(X7) ← SECOND OPERAND BYTE PROPERLY SHIFTED TO LINE UP WITH FIRST OPERAND BYTE
MASK OUT THE FIRST OPERAND BYTE AND INSERT THE SECOND OPERAND BYTE IN ITS PLACE
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3)-1
C(B4) ← C(B4)+1
C(B4) ≥ C(B2) ?
NO
YES
RANGE

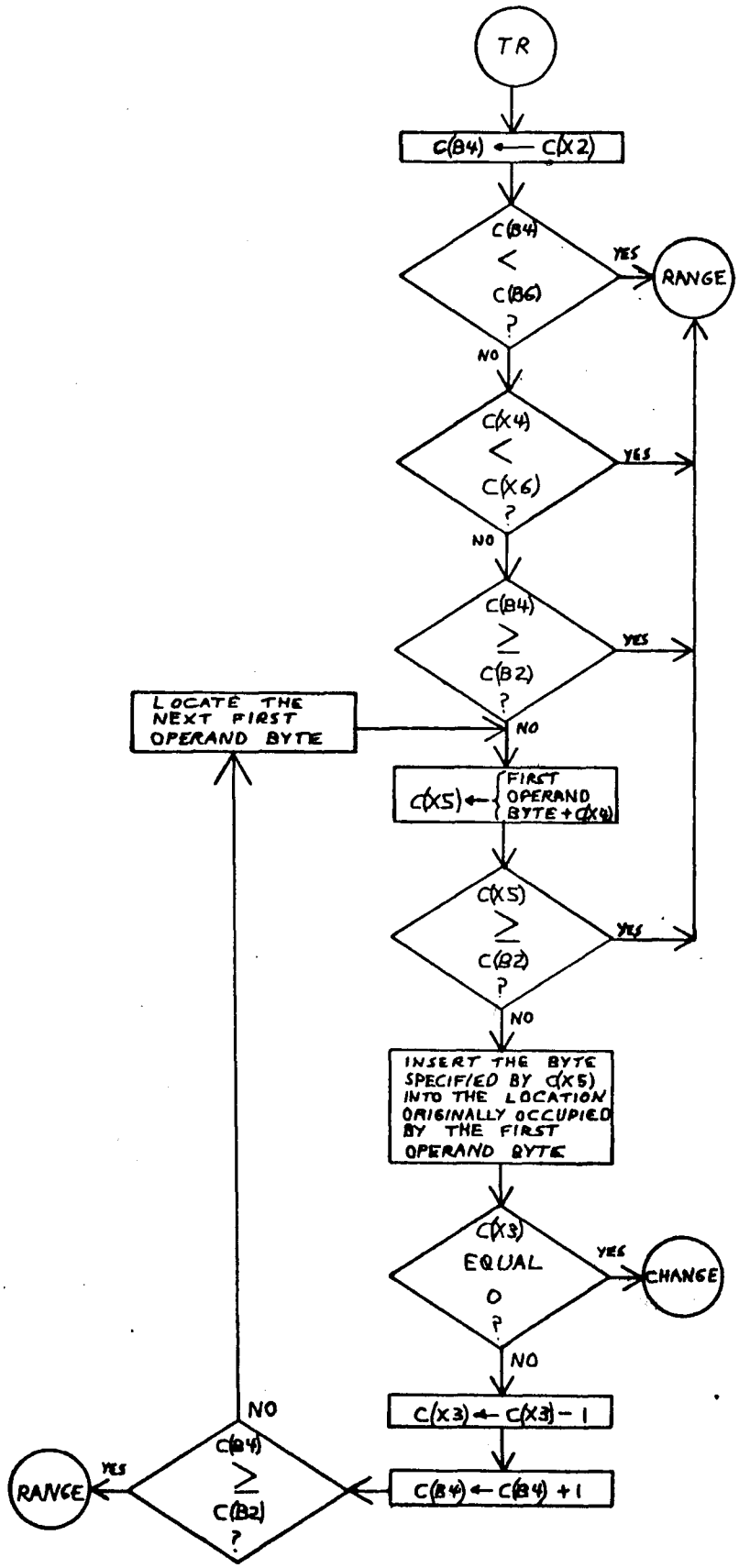

MVN
SMALLER OF THE TWO BYTE ADDRESSES
C(B6)
?
YES
RANGE
NO
C(B4) ←
THE LARGER OF THE TWO BYTE ADDRESSES
C(B4)
≥
C(B2)
?
YES
RANGE
NO
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE
C(X0) ←
MASK (00001111)BINARY PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
C(X7) ←
THE SECOND HALF OF THE SECOND OPERAND BYTE PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
MASK OUT THE SECOND HALF OF THE FIRST OPERAND BYTE AND INSERT THE SECOND HALF OF THE SECOND OPERAND BYTE IN ITS PLACE
C(X3)
EQUAL
0
?
YES
CHANGE
NO
C(X3) ← C(X3)-1
C(B4) ← C(B4)+1
NO
C(B4)
≥
C(B2)
?
YES
RANGE

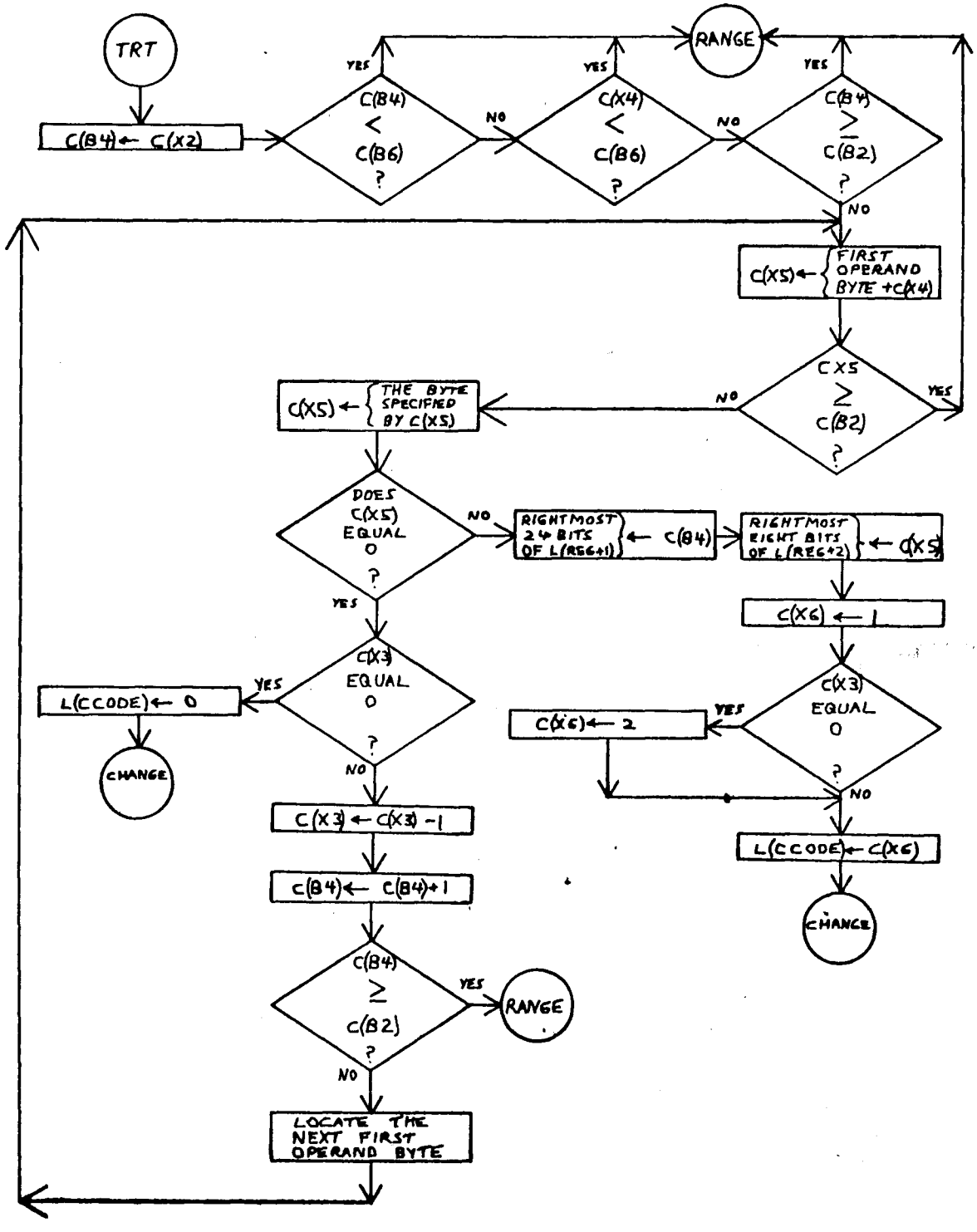

MVZ
SMALLER OF THE TWO BYTE ADDRESSES
<
C(BC)
?
YES
RANGE
NO
C(B4) ←
THE LARGER OF THE TWO BYTE ADDRESSES
C(B4)
≥
C(B2)
?
YES
RANGE
NO
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE
C(X0) ←
MASK (11110000 BINARY) PROPERLY SHIFTED TO LINE UP WITH FIRST OPERAND BYTE
C(X7) ←
THE FIRST HALF OF THE SECOND OPERAND BYTE PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
MASK OUT THE FIRST HALF OF THE FIRST OPERAND BYTE AND INSERT THE FIRST HALF OF THE SECOND OPERAND BYTE IN ITS PLACE
C(X3)
EQUAL
0
?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
C(B4) ← C(B4) + 1
NO
C(B4)
≥
C(B2)
?
YES
RANGE

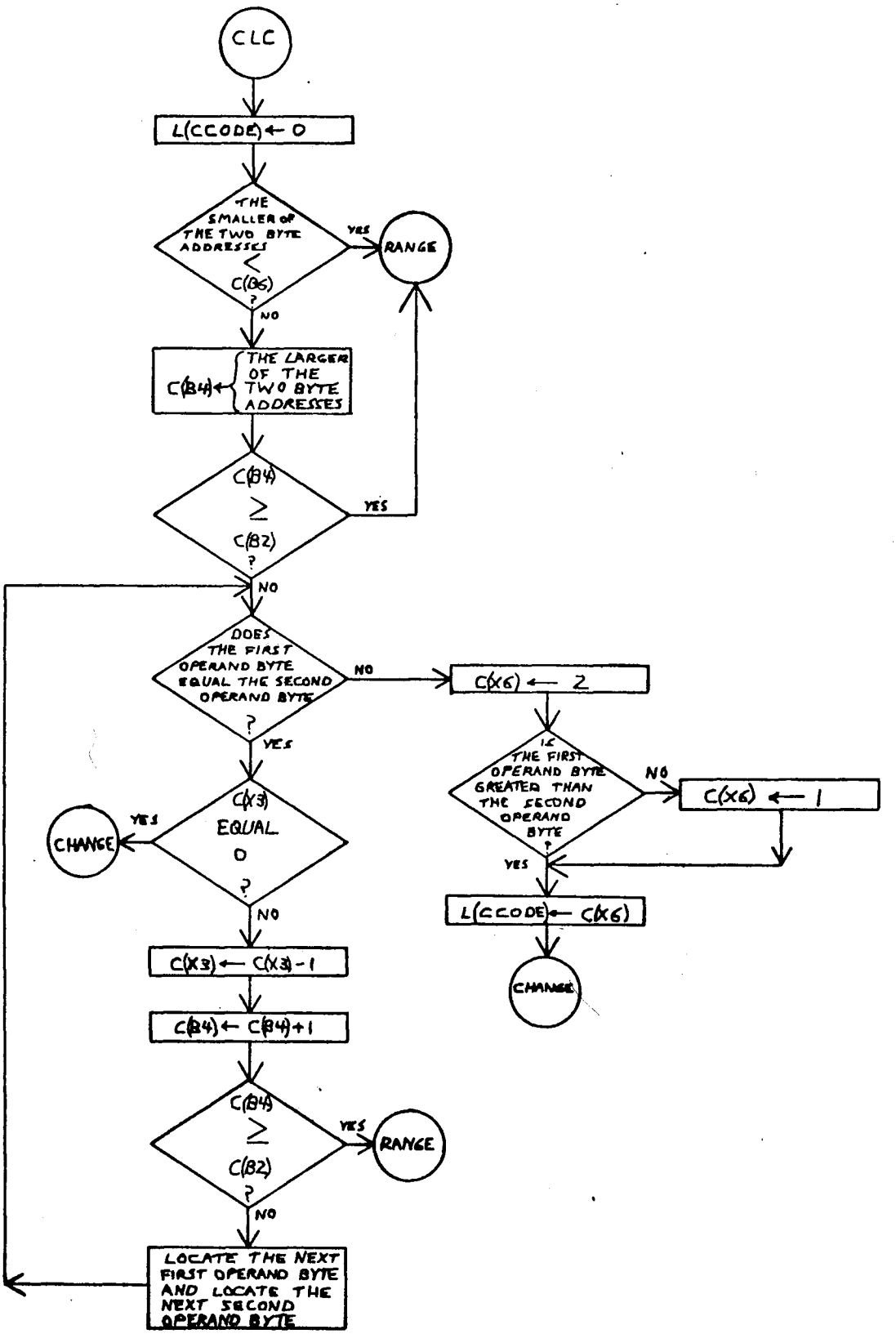

NC
L(CCODE) ← 0
SMALLER OF THE TWO BYTE ADDRESSES < C(B6) ?
YES
RANGE
NO
C(B4) ← THE LARGER OF THE TWO BYTE ADDRESSES
C(B4) ≥ C(B2) ?
YES
RANGE
NO
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE
C(X0) ← MASK (11111111 BINARY) PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
C(X7) ← THE SECOND OPERAND BYTE PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
REMOVE THE FIRST OPERAND BYTE AND INSERT THE .AND. OF THE FIRST OPERAND BYTE AND THE SECOND OPERAND BYTE IN ITS PLACE
IS THE .AND. OF THE FIRST OPERAND BYTE AND THE SECOND OPERAND BYTE 0 ?
YES
NO
L(CCODE) ← 1
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
C(B4) ← C(B4) + 1
C(B4) ≥ C(B2) ?
YES
RANGE
NO

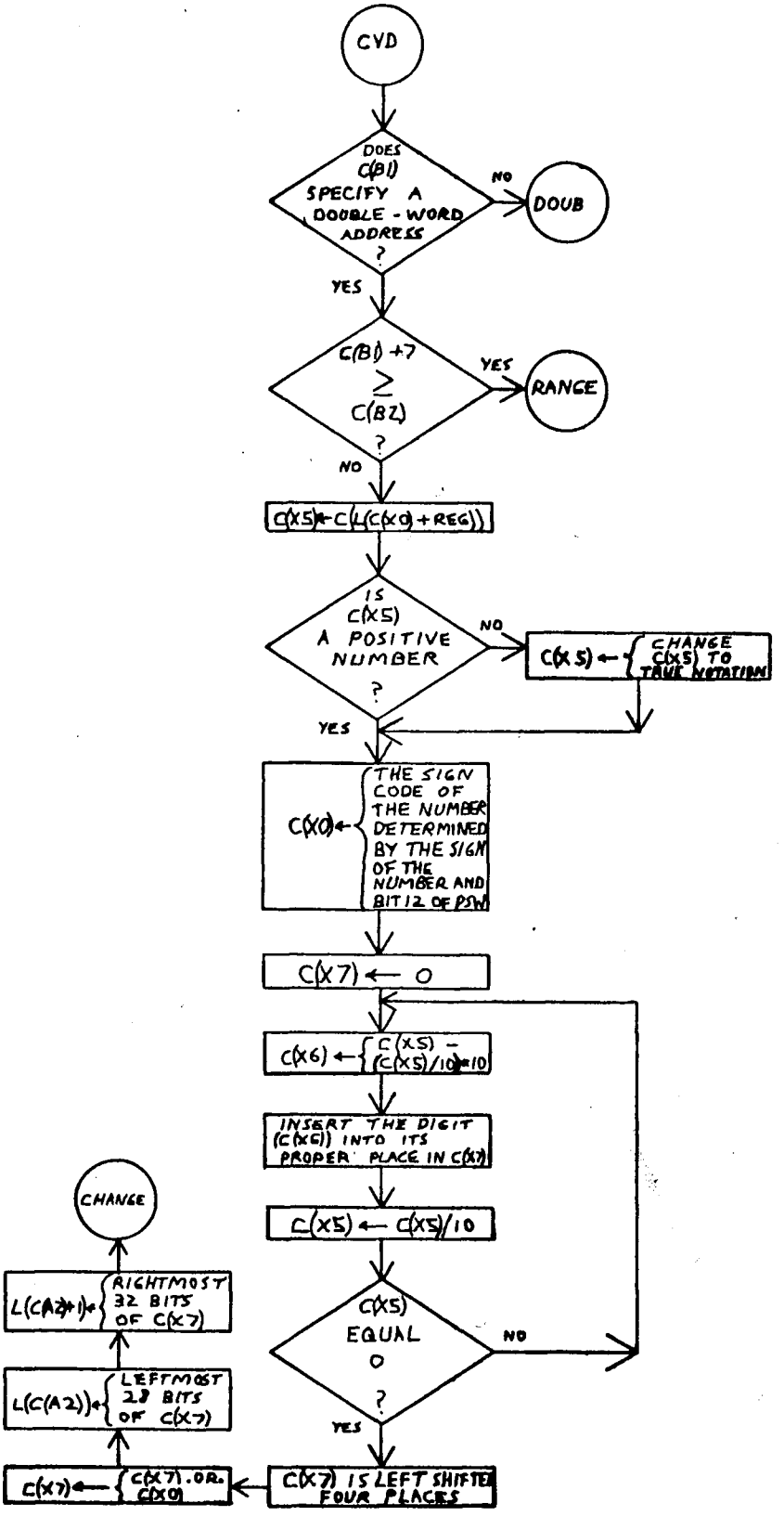

OC
L(CCODE) ← 0
SMALLER OF THE TWO BYTE ADDRESSES < C(B6) ?
YES
RANGE
NO
C(B4) ← THE LARGER OF THE TWO BYTE ADDRESSES
C(B4) ≥ C(B2) ?
YES
RANGE
NO
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE
C(X0) ← MASK (11111111 BINARY) PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
C(X7) ← THE SECOND OPERAND BYTE PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
REMOVE THE FIRST OPERAND BYTE AND INSERT THE .OR. OF THE FIRST OPERAND BYTE AND THE SECOND OPERAND BYTE IN ITS PLACE
IS THE .OR. BY THE FIRST OPERAND BYTE AND THE SECOND OPERAND BYTE EQUAL TO 0 ?
YES
NO
L(CCODE) ← 1
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
C(B4) ← C(B4) + 1
C(B4) ≥ C(B2) ?
YES
RANGE
NO

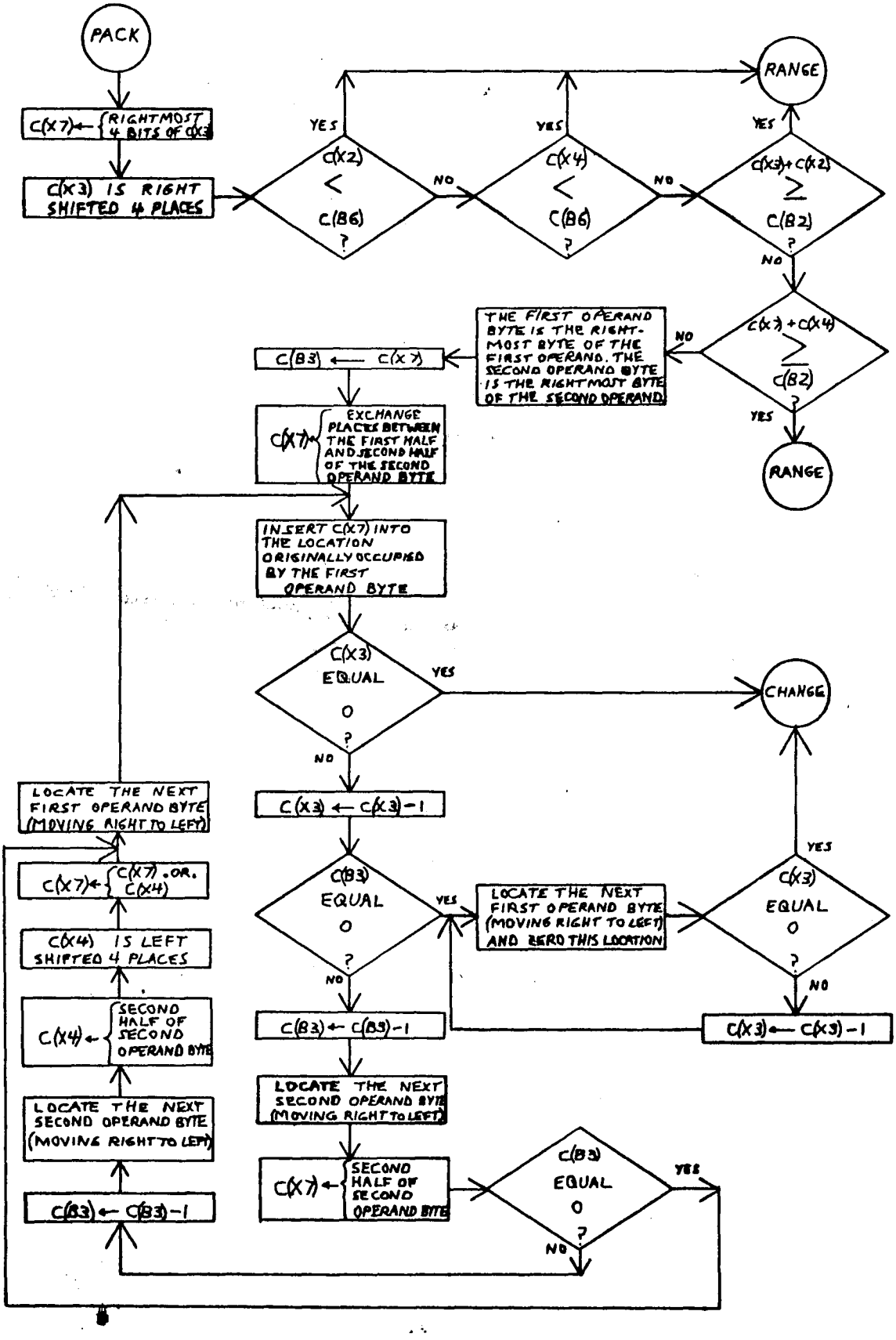

XC
L(CCODE) ← 0
SMALLER OF THE TWO BYTE ADDRESSES < C(B6) ?
YES
RANGE
NO
C(B4) ← THE LARGER OF THE TWO BYTE ADDRESSES
C(B4) ≥ C(B2) ?
YES
RANGE
NO
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE
C(X0) ← MASK (11111111 BINARY) PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
C(X7) ← THE SECOND OPERAND BYTE PROPERLY SHIFTED TO LINE UP WITH THE FIRST OPERAND BYTE
REMOVE THE FIRST OPERAND BYTE AND INSERT THE .XOR. OF THE FIRST OPERAND BYTE AND THE SECOND OPERAND BYTE IN ITS PLACE
IS THE .XOR. OF THE FIRST OPERAND BYTE AND THE SECOND OPERAND BYTE EQUAL TO 0 ?
YES
NO
L(CCODE) ← 1
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) -1
C(B4) ← C(B4)+1
C(B4) ≥ C(B2) ?
YES
RANGE
NO

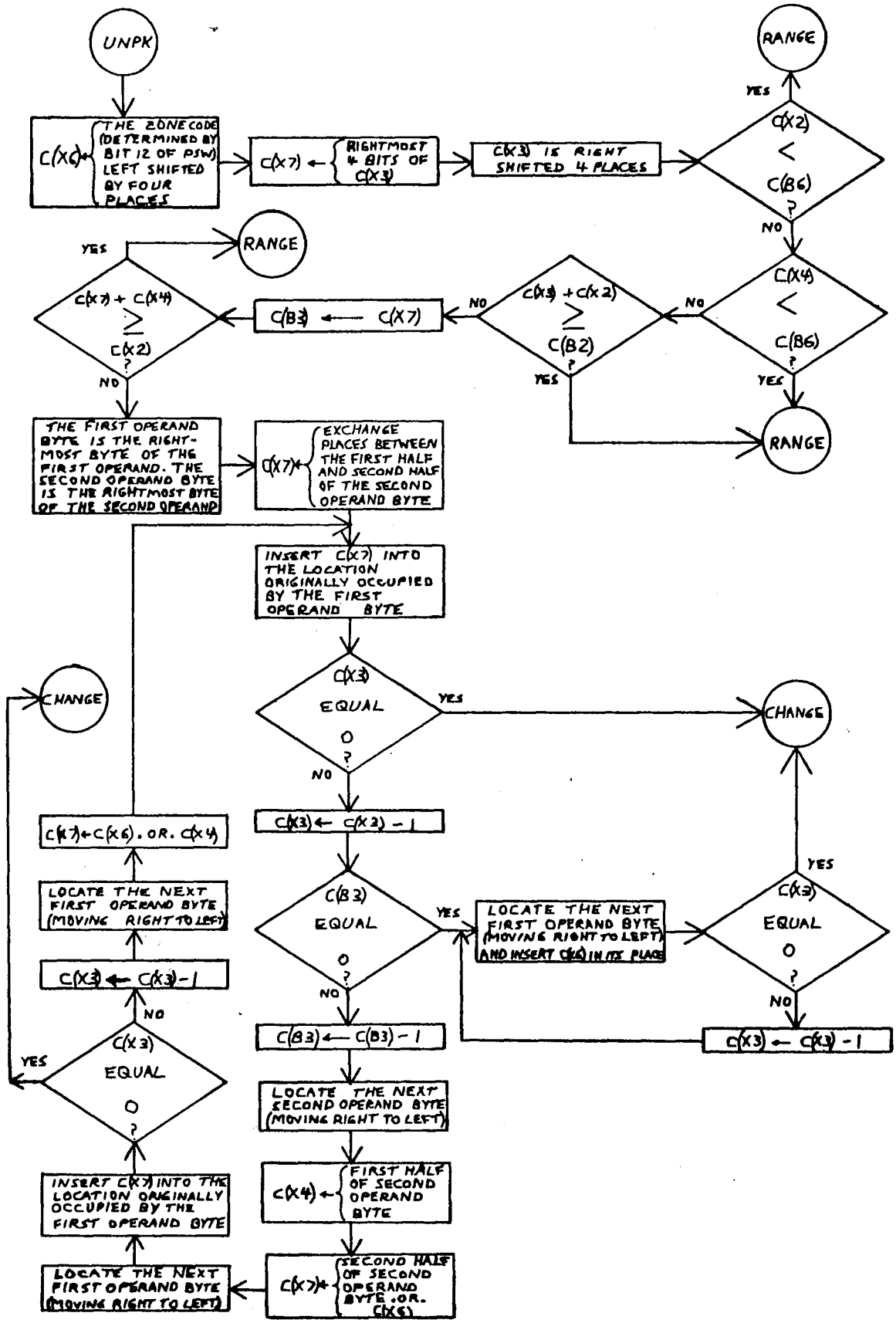

TR
C(B4) ← C(X2)
C(B4) < C(B6) ?
YES
RANGE
NO
C(X4) < C(X6) ?
YES
NO
C(B4) ≥ C(B2) ?
YES
NO
LOCATE THE NEXT FIRST OPERAND BYTE
C(X5) ← { FIRST OPERAND BYTE + C(X4)
C(X5) ≥ C(B2) ?
YES
NO
INSERT THE BYTE SPECIFIED BY C(X5) INTO THE LOCATION ORIGINALLY OCCUPIED BY THE FIRST OPERAND BYTE
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
C(B4) ← C(B4) + 1
C(B4) ≥ C(B2) ?
NO
YES
RANGE

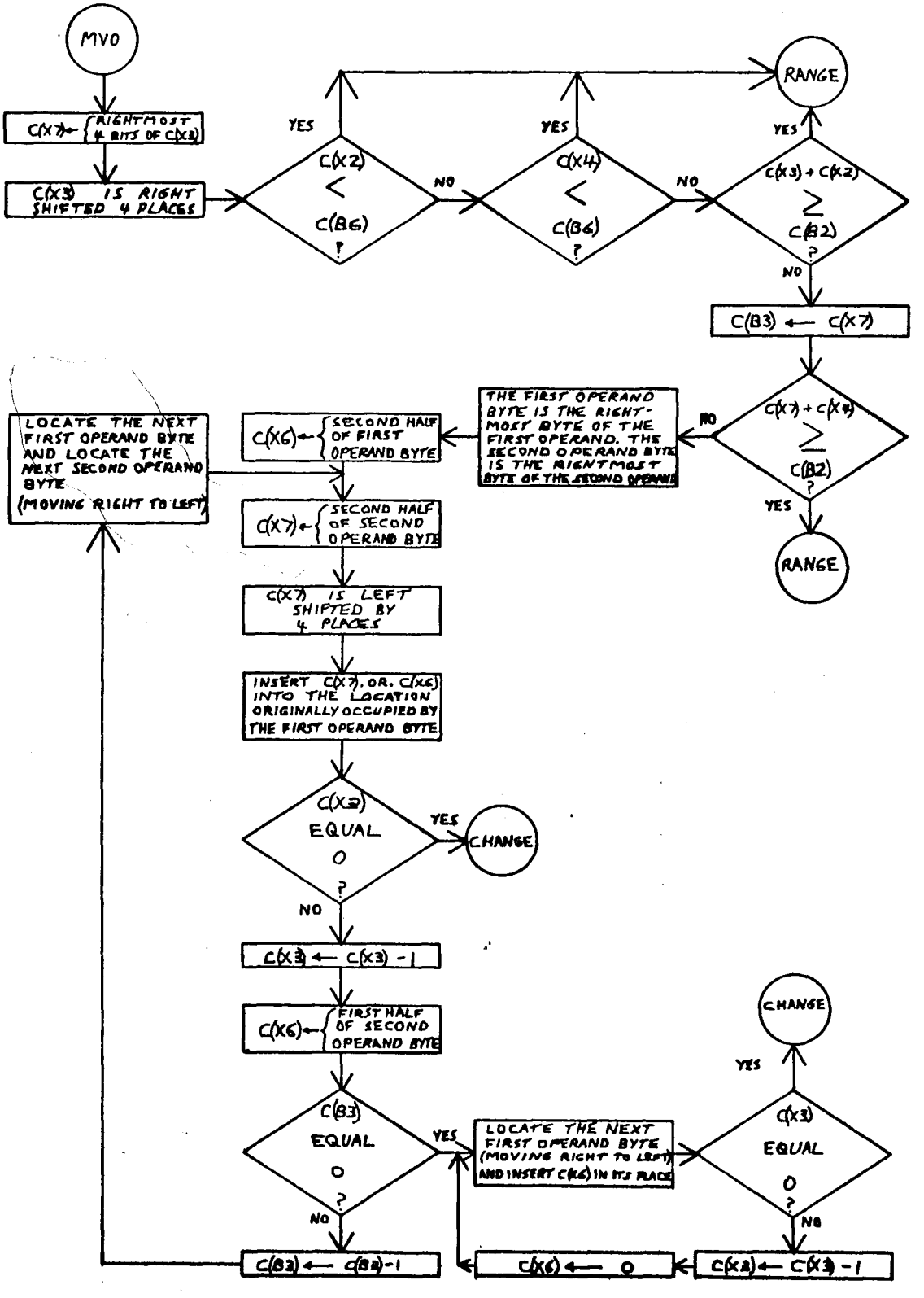

TRT
C(B4) ← C(X2)
C(B4) < C(B6) ?
C(X4) < C(B6) ?
C(B4) ≥ C(B2) ?
YES
NO
RANGE
C(X5) ← FIRST OPERAND BYTE + C(X4)
C X5 ≥ C(B2) ?
C(X5) ← THE BYTE SPECIFIED BY C(X5)
DOES C(X5) EQUAL 0 ?
RIGHTMOST 24 BITS OF L(REG+1) ← C(B4)
RIGHTMOST EIGHT BITS OF L(REG+2) ← C(X5)
C(X6) ← 1
C(X3) EQUAL 0 ?
L(CCODE) ← 0
CHANGE
C(X6) ← 2
C(X3) ← C(X3) - 1
C(B4) ← C(B4) + 1
C(B4) ≥ C(B2) ?
LOCATE THE NEXT FIRST OPERAND BYTE
L(CCODE) ← C(X6)

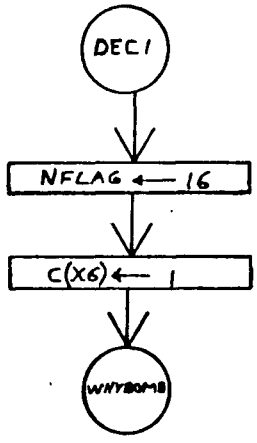

CLC
L(CCODE) ← 0
THE SMALLER OF THE TWO BYTE ADDRESSES < C(B5) ?
YES
RANGE
NO
C(B4) ← THE LARGER OF THE TWO BYTE ADDRESSES
C(B4) ≥ C(B2) ?
YES
NO
DOES THE FIRST OPERAND BYTE EQUAL THE SECOND OPERAND BYTE ?
NO
C(X6) ← 2
IS THE FIRST OPERAND BYTE GREATER THAN THE SECOND OPERAND BYTE ?
NO
C(X6) ← 1
YES
L(CCODE) ← C(X6)
CHANGE
YES
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
C(B4) ← C(B4) + 1
C(B4) ≥ C(B2) ?
YES
RANGE
NO
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE

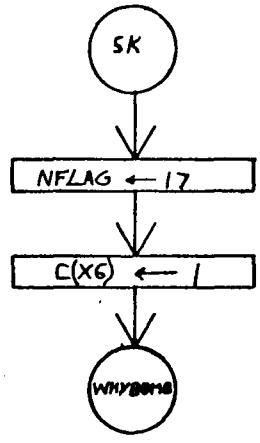

CVD
DOES C(B1) SPECIFY A DOUBLE-WORD ADDRESS ?
NO
DOUB
YES
C(B1)+7 ≥ C(B2) ?
YES
RANGE
NO
C(X5) ← C(L(C(X0)+REG))
IS C(X5) A POSITIVE NUMBER ?
NO
C(X5) ← CHANGE C(X5) TO TRUE NOTATION
YES
C(X0) ← THE SIGN CODE OF THE NUMBER DETERMINED BY THE SIGN OF THE NUMBER AND BIT 12 OF PSW
C(X7) ← 0
C(X6) ← C(X5) − (C(X5)/10)*10
INSERT THE DIGIT (C(X6)) INTO ITS PROPER PLACE IN C(X7)
C(X5) ← C(X5)/10
C(X5) EQUAL 0 ?
NO
YES
C(X7) IS LEFT SHIFTED FOUR PLACES
C(X7) ← C(X7) .OR. C(X0)
L(C(A2)) ← LEFTMOST 28 BITS OF C(X7)
L(C(A2)+1) ← RIGHTMOST 32 BITS OF C(X7)
CHANGE

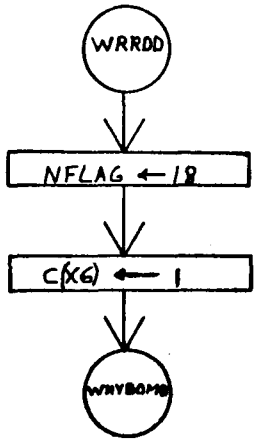

PACK
C(X7) ← {RIGHTMOST 4 BITS OF C(X3)
C(X3) IS RIGHT SHIFTED 4 PLACES
C(X2) < C(B6) ?
YES
NO
C(X4) < C(B6) ?
YES
NO
C(X3)+C(X2) ≥ C(B2) ?
YES
RANGE
NO
C(X7)+C(X4) ≥ C(B2) ?
YES
RANGE
NO
THE FIRST OPERAND BYTE IS THE RIGHT-MOST BYTE OF THE FIRST OPERAND. THE SECOND OPERAND BYTE IS THE RIGHTMOST BYTE OF THE SECOND OPERAND.
C(B3) ← C(X7)
C(X7) ← {EXCHANGE PLACES BETWEEN THE FIRST HALF AND SECOND HALF OF THE SECOND OPERAND BYTE
INSERT C(X7) INTO THE LOCATION ORIGINALLY OCCUPIED BY THE FIRST OPERAND BYTE
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3)-1
C(B3) EQUAL 0 ?
YES
LOCATE THE NEXT FIRST OPERAND BYTE (MOVING RIGHT TO LEFT) AND ZERO THIS LOCATION
C(X3) EQUAL 0 ?
YES
NO
C(X3) ← C(X3)-1
NO
C(B3) ← C(B3)-1
LOCATE THE NEXT SECOND OPERAND BYTE (MOVING RIGHT TO LEFT)
C(X7) ← {SECOND HALF OF SECOND OPERAND BYTE
C(B3) EQUAL 0 ?
YES
NO
C(B3) ← C(B3)-1
LOCATE THE NEXT SECOND OPERAND BYTE (MOVING RIGHT TO LEFT)
C(X4) ← {SECOND HALF OF SECOND OPERAND BYTE
C(X4) IS LEFT SHIFTED 4 PLACES
C(X7) ← {C(X7) .OR. C(X4)
LOCATE THE NEXT FIRST OPERAND BYTE (MOVING RIGHT TO LEFT)

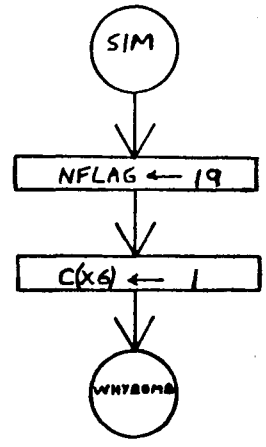

UNPK
C(X6) ← THE ZONE CODE (DETERMINED BY BIT 12 OF PSW) LEFT SHIFTED BY FOUR PLACES
C(X7) ← RIGHTMOST 4 BITS OF C(X3)
C(X3) IS RIGHT SHIFTED 4 PLACES
C(X2) < C(B6) ?
YES
RANGE
NO
C(X4) < C(B6) ?
YES
RANGE
NO
C(X3) + C(X2) ≥ C(B2) ?
YES
NO
C(B3) ← C(X7)
C(X7) + C(X4) ≥ C(X2) ?
YES
RANGE
NO
THE FIRST OPERAND BYTE IS THE RIGHTMOST BYTE OF THE FIRST OPERAND. THE SECOND OPERAND BYTE IS THE RIGHTMOST BYTE OF THE SECOND OPERAND
C(X7) ← EXCHANGE PLACES BETWEEN THE FIRST HALF AND SECOND HALF OF THE SECOND OPERAND BYTE
INSERT C(X7) INTO THE LOCATION ORIGINALLY OCCUPIED BY THE FIRST OPERAND BYTE
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
C(B3) EQUAL 0 ?
YES
LOCATE THE NEXT FIRST OPERAND BYTE (MOVING RIGHT TO LEFT) AND INSERT C(X6) IN ITS PLACE
C(X3) EQUAL 0 ?
YES
NO
C(X3) ← C(X3) - 1
NO
C(B3) ← C(B3) - 1
LOCATE THE NEXT SECOND OPERAND BYTE (MOVING RIGHT TO LEFT)
C(X4) ← FIRST HALF OF SECOND OPERAND BYTE
C(X7) ← SECOND HALF OF SECOND OPERAND BYTE .OR. C(X6)
LOCATE THE NEXT FIRST OPERAND BYTE (MOVING RIGHT TO LEFT)
INSERT C(X7) INTO THE LOCATION ORIGINALLY OCCUPIED BY THE FIRST OPERAND BYTE
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
LOCATE THE NEXT FIRST OPERAND BYTE (MOVING RIGHT TO LEFT)
C(X7) ← C(X6) .OR. C(X4)

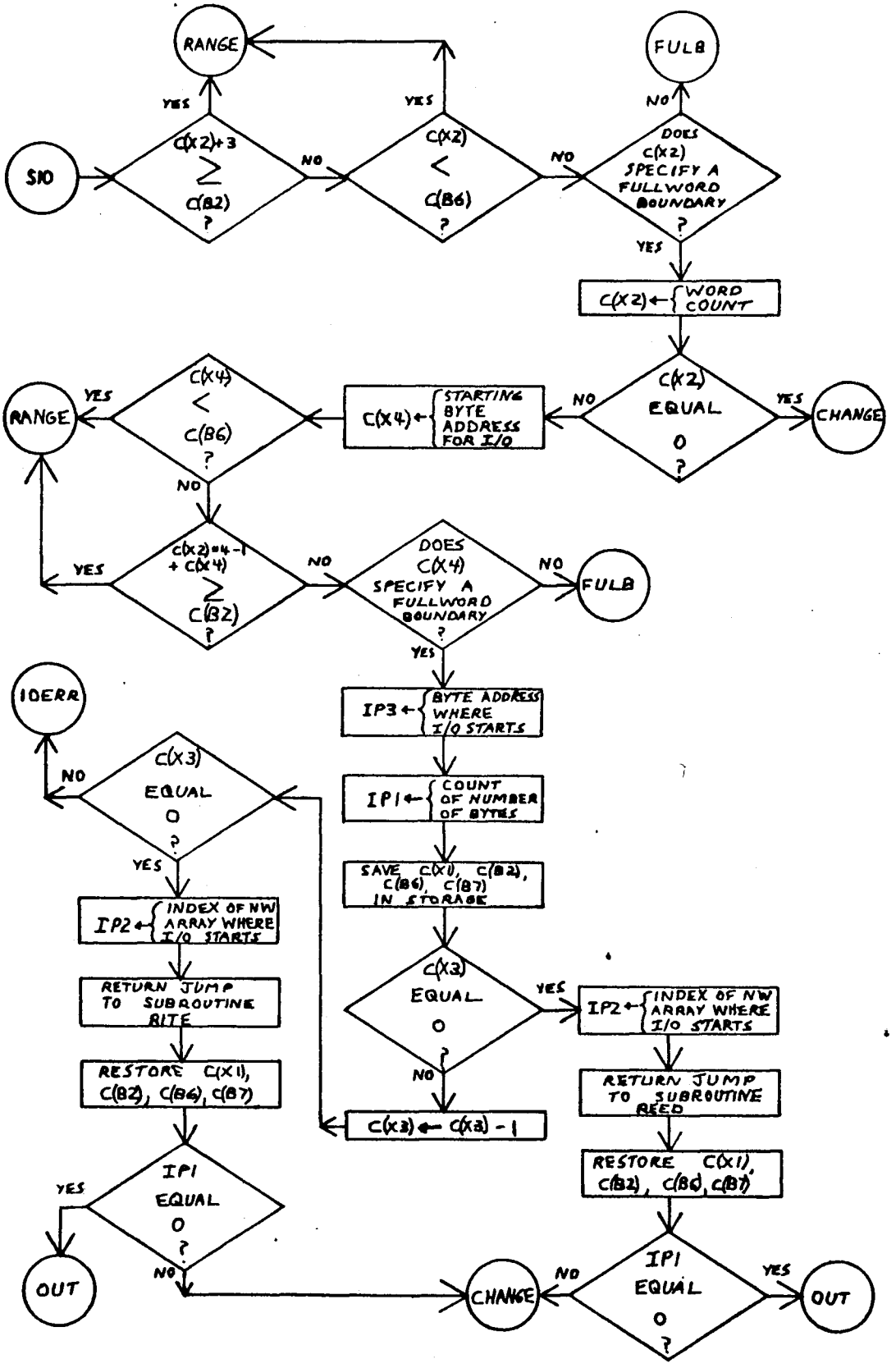

MVO
C(X7) ← {RIGHTMOST 4 BITS OF C(X3)}
C(X3) IS RIGHT SHIFTED 4 PLACES
C(X2) < C(B6) ?
YES
NO
C(X4) < C(B6) ?
YES
NO
C(X3) + C(X2) ≥ C(B2) ?
YES
NO
RANGE
C(B3) ← C(X7)
C(X7) + C(X4) ≥ C(B2) ?
NO
YES
RANGE
THE FIRST OPERAND BYTE IS THE RIGHTMOST BYTE OF THE FIRST OPERAND. THE SECOND OPERAND BYTE IS THE RIGHTMOST BYTE OF THE SECOND OPERAND
C(X6) ← {SECOND HALF OF FIRST OPERAND BYTE}
LOCATE THE NEXT FIRST OPERAND BYTE AND LOCATE THE NEXT SECOND OPERAND BYTE (MOVING RIGHT TO LEFT)
C(X7) ← {SECOND HALF OF SECOND OPERAND BYTE}
C(X7) IS LEFT SHIFTED BY 4 PLACES
INSERT C(X7) OR C(X6) INTO THE LOCATION ORIGINALLY OCCUPIED BY THE FIRST OPERAND BYTE
C(X3) EQUAL 0 ?
YES
CHANGE
NO
C(X3) ← C(X3) - 1
C(X6) ← {FIRST HALF OF SECOND OPERAND BYTE}
C(B3) EQUAL 0 ?
YES
LOCATE THE NEXT FIRST OPERAND BYTE (MOVING RIGHT TO LEFT) AND INSERT C(X6) IN ITS PLACE
NO
C(B2) ← C(B2) - 1
CHANGE
C(X3) EQUAL 0 ?
YES
NO
C(X6) ← 0
C(X2) ← C(X3) - 1

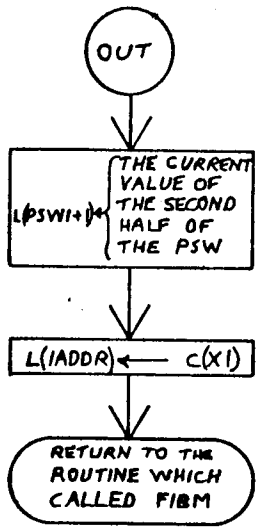

DEC1
NFLAG ← 16
C(X6) ← 1
WHYBOMB


SK
NFLAG ← 17
C(X6) ← 1
WHYBOMB


WRRDD
NFLAG ← 18
C(X6) ← 1
WHYBOMB


SIM
NFLAG ← 19
C(X6) ← 1
WHYBOMB


SIO
C(X2)+3 ≥ C(B2) ?
YES
RANGE
NO
C(X2) < C(B6) ?
YES
NO
DOES C(X2) SPECIFY A FULLWORD BOUNDARY ?
NO
FULB
YES
C(X2) ← WORD COUNT
C(X2) EQUAL 0 ?
NO
C(X4) ← STARTING BYTE ADDRESS FOR I/O
YES
CHANGE
C(X4) < C(B6) ?
YES
RANGE
NO
C(X2)*4-1 + C(X4) ≥ C(B2) ?
YES
NO
DOES C(X4) SPECIFY A FULLWORD BOUNDARY ?
NO
FULB
YES
IOERR
IP3 ← BYTE ADDRESS WHERE I/O STARTS
C(X3) EQUAL 0 ?
NO
YES
IP1 ← COUNT OF NUMBER OF BYTES
SAVE C(X1), C(B2), C(B6), C(B7) IN STORAGE
IP2 ← INDEX OF NW ARRAY WHERE I/O STARTS
RETURN JUMP TO SUBROUTINE BITE
RESTORE C(X1), C(B2), C(B6), C(B7)
C(X3) EQUAL 0 ?
YES
NO
C(X3) ← C(X3) - 1
IP2 ← INDEX OF NW ARRAY WHERE I/O STARTS
RETURN JUMP TO SUBROUTINE REED
RESTORE C(X1), C(B2), C(B6), C(B7)
IP1 EQUAL 0 ?
YES
OUT
NO
CHANGE
IP1 EQUAL 0 ?
NO
YES
OUT


OUT
L(PSWI+1)
THE CURRENT VALUE OF THE SECOND HALF OF THE PSW
L(IADDR) ← C(X1)
RETURN TO THE ROUTINE WHICH CALLED FIBM

VI Improving LUIS

6.1 Introduction

While the Lehigh University IBM 360 Simulator is quite useful in its present form, there are a number of changes which could be made to improve it. Section 2 describes how more memory could be added to the simulated IBM 360. Section 3 describes how the decimal and floating-point feature instructions could be added to the simulator. Section 4 suggests a modification to LUIS which would enable the user to select whether the simulator sent normal or abbreviated messages to the user. Section 5 describes how additional commands could be added to the simulator.

6.2 Adding More Memory

The maximum storage capacity of an IBM 360 is $16,777,216_{decimal}$ byte addresses. Unfortunately the CDC 6400 can only provide the user with a maximum of $40,960_{decimal}$ sixty bit storage locations. Therefore it is necessary to limit the size of the memory of the simulated IBM 360 to a small fraction of the maximum capacity. Since byte addresses are often placed into eighteen bit B registers by subroutine FIBM, byte addresses must be restricted to a maximum length of seventeen significant bits. This fact restricts the largest possible byte address to $131,071_{decimal}$.

LUIS allows the user to specify the portion of storage which will be used. The user may specify any starting address between 000000 and 01D8A8 hexadecimal ($121,000_{decimal}$). The user's portion of storage may contain a maximum of ten thousand bytes. While the size of the simulated memory is currently limited to ten thousand bytes, this limit could be increased. By increasing the dimension of the NW array and by modifying the DEC subroutine so that an integer larger than 9,999 could be read, the maximum size of the simulated memory could be increased. Naturally, even if this was done, it would still be necessary to limit the largest possible byte address to a number less than $131,072_{decimal}$.

6.3 Adding More Instructions

The decimal feature instructions are not available in LUIS. However, one could easily add these instructions to LUIS. Currently a request for any of these instructions causes subroutine FIBM to branch to a section of the subroutine called DECI. DECI causes an error message to be printed and terminates execution of the user's program. One could add additional sections to FIBM which would perform the operations specified by the decimal feature instructions. Subroutine FIBM could then be changed so that a request for a decimal feature instruction would cause FIBM to branch to the appropriate section where the operation would be performed.

Currently a request for any of the floating-point feature instructions causes FIBM to branch to a section of the subroutine called FLOAT. FLOAT causes an error message to be printed and terminates execution of the user's program. One could add additional sections to FIBM which would perform the operations specified by the floating-point feature instructions. Sections RR and RX of subroutine FIBM would have to be modified to include branches to the sections of FIBM which would perform the floating-point operations. Naturally it would be necessary to add floating-point registers to LUIS. These registers could be storage locations in the "STATUS" common block.

6.4 Abbreviated Messages From LUIS

Whenever the user must supply some information to the simulator, the simulator specifies the exact nature and format of the information required. Usually the message is in the form of a one or two line sentence. These messages are especially useful to the person who has not used the simulator previously. Unfortunately these messages can limit the speed with which an experienced user can use the simulator. This fact is especially true when using a teletype.

For the experienced user, a two or three word phrase would provide sufficient information so that the user would know what type of information the simulator required. The simulator could be modified to provide two additional requests. One request would cause the simulator to go into the abbreviated mode where all messages sent to the user would be in an abbreviated form. Another request would place the simulator in the normal mode. In the normal mode all messages would be printed by the simulator in the manner currently being used.

6.5 Additional Simulator Commands

At the present time there are eleven different commands which the user may issue when the simulator is in the request mode. The main routine uses a series of IF statements to branch to the section of the routine which handles the particular request. Additional commands could be added to the simulator. Additional IF statements would be added to the present series which would check for the new commands. These IF statements would then branch to sections of the main routine which would handle the new commands.

References

1. A22-6821-6, IBM System/360 Principles of Operation, Copyright 1967, International Business Machines Corporation.

2. COMPASS Version 3 Reference Manual, Copyright 1974, Control Data Corporation.

3. Grishman, Ralph: "Assembly Language Programming for the Control Data 6000 Series," Algorithmic Press, New York, Copyright 1971.

4. SCOPE Reference Manual Version 3.4.1, Copyright 1974, Control Data Corporation.

Appendix A

The following material is reproduced from the manual, IBM System/360 Principles of Operation. This material should be of value to those who are unfamiliar with the IBM 360 structure.

Instruction Format

The length of an instruction format can be one, two, or three halfwords. It is related to the number of storage addresses necessary for the operation. An instruction consisting of only one halfword causes no reference to main storage. A two-halfword instruction provides one storage-address specification; a three-halfword instruction provides two storage-address specifications. All instructions must be located in storage on integral boundaries for halfwords. Figure 13 shows five basic instruction formats.

The five basic instruction formats are denoted by the format codes RR, RX, RS, SI, and SS. The format codes express, in general terms, the operation to be performed. RR denotes a register-to-register operation; RX, a register-and-indexed-storage operation; RS, a register-and-storage operation; SI, a storage and immediate-operand operation; and SS, a storage-to-storage operation. An immediate operand is one contained within the instruction.

For purposes of describing the execution of instructions, operands are designated as first and second operands and, in the case of branch-on-index instructions, third operands. These names refer to the manner in which the operands participate. The operand to which a field in an instruction format applies is generally denoted by the number following the code name of the field, for example, R_1, B_1, L_2, D_2.

In each format, the first instruction halfword consists of two parts. The first byte contains the operation code (op code). The length and format of an instruction are specified by the first two bits of the operation code.

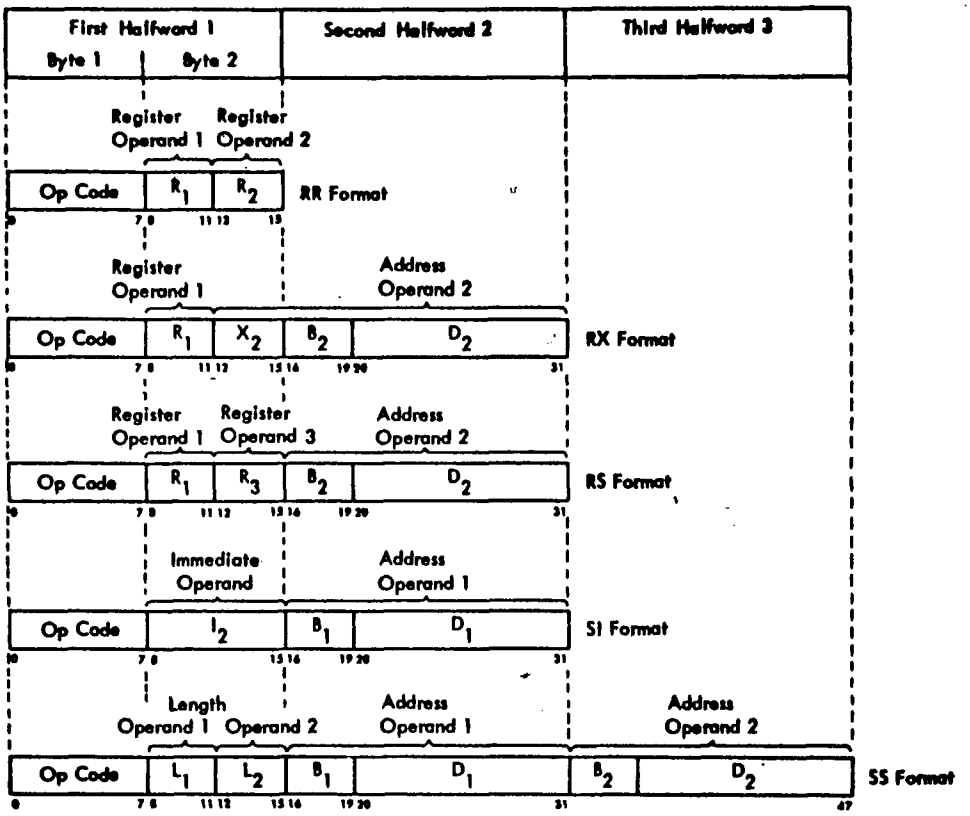


Figure 13. Five Basic Instruction Formats

INSTRUCTION LENGTH RECORDING

BIT POSITIONS (0-1)	INSTRUCTION LENGTH	INSTRUCTION FORMAT
00	One halfword	RR
01	Two halfwords	RX
10	Two halfwords	RS or SI
11	Three halfwords	SS

The second byte is used either as two 4-bit fields or as a single eight-bit field. This byte can contain the following information:

Four-bit operand register specification (R_1, R_2, or R_3)
Four-bit index register specification (X_2)
Four-bit mask (M_1)
Four-bit operand length specification (L_1 or L_2)
Eight-bit operand length specification (L)
Eight-bit byte of immediate data (I_2)

In some instructions a four-bit field or the whole second byte of the first halfword is ignored.

The second and third halfwords always have the same format:

Four-bit base register designator (B_1 or B_2), followed by a 12-bit displacement (D_1 or D_2).

Address Generation

For addressing purposes, operands can be grouped in three classes: explicitly addressed operands in main storage, immediate operands placed as part of the instruction stream in main storage, and operands located in the general or floating-point registers.

To permit the ready relocation of program segments and to provide for the flexible specifications of input, output, and working areas, all instructions referring to main storage have been given the capacity of employing a full address.

The address used to refer to main storage is generated from the following three binary numbers:

Base Address (B) is a 24-bit number contained in a general register specified by the program in the B field of the instruction. The B field is included in every address specification. The base address can be used as a means of static relocation of programs and data. In array-type calculations, it can specify the location of an array and, in record-type processing, it can identify the record. The base address provides for addressing the entire main storage. The base address may also be used for indexing purposes.

Index (X) is a 24-bit number contained in a general register specified by the program in the X field of the instruction. It is included only in the address specified by the RX instruction format. The RX format instructions permit double indexing; i.e., the index can be used to provide the address of an element within an array.

Displacement (D) is a 12-bit number contained in the instruction format. It is included in every address computation. The displacement provides for relative addressing up to 4095 bytes beyond the element or base address. In array-type calculations the displacement can be used to specify one of many items associated with an element. In the processing of records, the displacement can be used to identify items within a record.

In forming the address, the base address and index are treated as unsigned 24-bit positive binary integers. The displacement is similarly treated as a 12-bit positive binary integer. The three are added as 24-bit binary numbers, ignoring overflow. Since every address includes a base, the sum is always 24 bits long. The address bits are numbered 8-31 corresponding to the numbering of the base address and index bits in the general register.

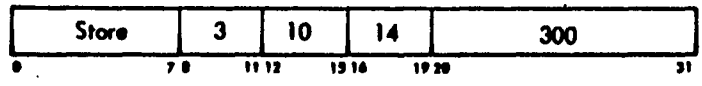
The program may have zeros in the base address, index, or displacement fields. A zero is used to indicate the absence of the corresponding address component. A base or index of zero implies that a zero quantity is to be used in forming the address, regardless of the contents of general register 0. A displacement of zero has no special significance. Initialization, modification, and testing of base addresses and indexes can be carried out by fixed-point instructions, or by BRANCH AND LINK, BRANCH ON COUNT, or BRANCH-ON-INDEX instructions.

As an aid in describing the logic of the instruction format, examples of two instructions and their related instruction formats follow.

RR Format

Execution of the ADD instruction adds the contents of general register 9 to the contents of general register 7 and the sum of the addition is placed in general register 7.

RX Format



Execution of the STORE instruction stores the contents of general register 3 at a main-storage location addressed by the sum of 300 and the low-order 24 bits of general registers 14 and 10.

Sequential Instruction Execution

Normally, the operation of the CPU is controlled by instructions taken in sequence. An instruction is fetched from a location specified by the instruction address in the current PSW. The instruction address is then increased by the number of bytes in the instruction fetched to address the next instruction in sequence. The instruction is then executed and the same steps are repeated using the new value of the instruction address.

Conceptually, all halfwords of an instruction are fetched from storage after the preceding operation is completed and before execution of the current operation, even though physical storage word size and overlap of instruction execution with storage access may cause actual instruction fetching to be different. Thus, it is possible to modify an instruction in storage by the immediately preceding instruction.

A change from sequential operation may be caused by branching, status switching, interruptions, or manual intervention.

Branching

The normal sequential execution of instructions is changed when reference is made to a subroutine, when a two-way choice is encountered, or when a segment of coding, such as a loop, is to be repeated. All these tasks can be accomplished with branching instructions. Provision is made for subroutine linkage, permitting not only the introduction of a new instruction address but also the preservation of the return address and associated information.

Decision-making is generally and symmetrically provided by the BRANCH ON CONDITION instruction. This instruction inspects a two-bit *condition code* that reflects the result of a majority of the arithmetic, logical, and I/O operations. Each of these operations can set the code in any one of four states, and the conditional branch can specify any selection of these four states as the criterion for branching. For example, the condition code reflects such conditions as nonzero, first operand high, equal, overflow, channel busy, zero, etc. Once set, the condition code remains unchanged until modified by an instruction that reflects a different condition code.

The two bits of the condition code provide for four possible condition code settings: 0, 1, 2, and 3. The specific meaning of any setting is significant only to the operation setting the condition code.

List of Instructions by Set and Feature

Standard Instruction Set

NAME	MNEMONIC	TYPE		CODE
Add	AR	RR	C	1A
Add	A	RX	C	5A
Add Halfword	AH	RX	C	4A
Add Logical	ALR	RR	C	1E
Add Logical	AL	RX	C	5E
AND	NR	RR	C	14
AND	N	RX	C	54
AND	NI	SI	C	94
AND	NC	SS	C	D4
Branch and Link	BALR	RR		05
Branch and Link	BAL	RX		45
Branch on Condition	BCR	RR		07
Branch on Condition	BC	RX		47
Branch on Count	BCTR	RR		06
Branch on Count	BCT	RX		46
Branch on Index High	BXH	RS		86
Branch on Index Low or Equal	BXLE	RS		87
Compare	CR	RR	C	19
Compare	C	RX	C	59
Compare Halfword	CH	RX	C	49
Compare Logical	CLR	RR	C	15
Compare Logical	CL	RX	C	55
Compare Logical	CLC	SS	C	D5
Compare Logical	CLI	SI	C	95
Convert to Binary	CVB	RX		4F
Convert to Decimal	CVD	RX		4E
Diagnose		SI		83
Divide	DR	RR		1D
Divide	D	RX		5D
Exclusive OR	XR	RR	C	17
Exclusive OR	X	RX	C	57
Exclusive OR	XI	SI	C	97
Exclusive OR	XC	SS	C	D7
Execute	EX	RX		44
Halt I/O	HIO	SI	C	9E
Insert Character	IC	RX		43
Load	LR	RR		18
Load	L	RX		58
Load Address	LA	RX		41
Load and Test	LTR	RR	C	12
Load Complement	LCR	RR	C	13
Load Halfword	LH	RX		48
Load Multiple	LM	RS		98
Load Negative	LNR	RR	C	11
Load Positive	LPR	RR	C	10
Load PSW	LPSW	SI	L	82

NAME	MNEMONIC	TYPE		CODE
Move	MVI	SI		92
Move	MVC	SS		D2
Move Numerics	MVN	SS		D1
Move with Offset	MVO	SS		F1
Move Zones	MVZ	SS		D3
Multiply	MR	RR		1C
Multiply	M	RX		5C
Multiply Halfword	MH	RX		4C
OR	OR	RR	C	16
OR	O	RX	C	56
OR	OI	SI	C	96
OR	OC	SS	C	D6
Pack	PACK	SS		F2
Set Program Mask	SPM	RR	L	04
Set System Mask	SSM	SI		80
Shift Left Double	SLDA	RS	C	8F
Shift Left Single	SLA	RS	C	8B
Shift Left Double Logical	SLDL	RS		8D
Shift Left Single Logical	SLL	RS		89
Shift Right Double	SRDA	RS	C	8E
Shift Right Single	SRA	RS	C	8A
Shift Right Double Logical	SRDL	RS		8C
Shift Right Single Logical	SRL	RS		88
Start I/O	SIO	SI	C	9C
Store	ST	RX		50
Store Character	STC	RX		42
Store Halfword	STH	RX		40
Store Multiple	STM	RS		90
Subtract	SR	RR	C	1B
Subtract	S	RX	C	5B
Subtract Halfword	SH	RX	C	4B
Subtract Logical	SLR	RR	C	1F
Subtract Logical	SL	RX	C	5F
Supervisor Call	SVC	RR		0A
Test and Set	TS	SI	C	93
Test Channel	TCH	SI	C	9F
Test I/O	TIO	SI	C	9D
Test Under Mask	TM	SI	C	91
Translate	TR	SS		DC
Translate and Test	TRT	SS	C	DD
Unpack	UNPK	SS		F3

Note: A"C" in the TYPE column means that the condition code is set. An"L" in the TYPE column means that a new condition code is loaded.

Appendix B

LEHIGH UNIVERSITY IBM 360 SIMULATOR

```
ENTER THE SMALLEST ADDRESS IN YOUR PROGRAM
AS AN EIGHT DIGIT HEXADECIMAL NUMBER.
00000000
ENTER THE MAXIMUM SIZE OF YOUR PROGRAM (IN BYTES)
AS A FOUR DIGIT DECIMAL INTEGER.
0080
ENTER THE PSW AS A 16 DIGIT HEXADECIMAL NUMBER.
0000000000000000
REQUEST=INSERT
ENTER THE STARTING ADDRESS (OF THE COLLECTION OF BYTES)
AS AN EIGHT DIGIT HEXADECIMAL NUMBER.
00000000
ENTER THE NUMBER OF BYTES WHICH WILL BE INSERTED AS
A FOUR DIGIT DECIMAL INTEGER.
0072
IF YOU HAVE THE BYTES ON THE FILE PGM, TYPE PGM.
IF YOU ARE GOING TO ENTER THE BYTES, TYPE INPUT.
PGM
THE BYTES HAVE BEEN READ
REQUEST=DUMP
TYPE ALL OR PARTIAL
ALL
00000  05709C00   00004  702E5810   00008  70365910   0000C  703A47A0
00010  70145810   00014  703A5910   00018  703E47A0   0001C  70205810
00020  703E5010   00024  70429C01   00028  703247F0   0002C  70000000
00030  00037036   00034  00017042   00038  00000023   0003C  00000034
00040  00000012   00044  00000034   00048  00000000   0004C  00000000
REQUEST=EXECUTE
YOUR PROGRAM WANTS TO READ SOME WORDS
IF YOU HAVE THE WORDS ON THE FILE PGM, TYPE PGM.
IF YOU ARE GOING TO ENTER THE WORDS, TYPE INPUT.
INPUT
ENTER THE WORDS. (ONE WORD PER LINE)

00038 -0000004C
0003C -00000039
00040 -0000000D

          WORDS READ INTO MEMORY BY THE USER'S PROGRAM

00038  0000004C
```

```
0003C  00000039
00040  0000000D
```

WORDS PRINTED FROM MEMORY BY THE USER"S PROGRAM

```
00044  0000004C
YOUR PROGRAM WANTS TO READ SOME WORDS
IF YOU HAVE THE WORDS ON THE FILE PGM, TYPE PGM.
IF YOU ARE GOING TO ENTER THE WORDS, TYPE INPUT.
INPUT
ENTER THE WORDS. (ONE WORD PER LINE)

00038 -00000004
0003C -0000000A
00040 -000000FF
```

WORDS READ INTO MEMORY BY THE USER"S PROGRAM

```
00038  00000004
0003C  0000000A
00040  000000FF
```

WORDS PRINTED FROM MEMORY BY THE USER"S PROGRAM

```
00044  000000FF
YOUR PROGRAM WANTS TO READ SOME WORDS
IF YOU HAVE THE WORDS ON THE FILE PGM, TYPE PGM.
IF YOU ARE GOING TO ENTER THE WORDS, TYPE INPUT.
INPUT
ENTER THE WORDS. (ONE WORD PER LINE)

00038 -00000000
0003C -00000000
00040 -000ZZZZW
```

WORDS READ INTO MEMORY BY THE USER"S PROGRAM

```
00038  00000000
0003C  00000000
00040  000ZZZZW
```

```
    **** AN INVALID CHARACTER APPEARED IN A HEXADECIMAL NUMBER.
THE BYTES WERE REMOVED FROM MEMORY
REQUEST=STATUS

        PROGRAM STATUS WORD     0000000090000006


        INSTRUCTION ADDRESS     00006
        CONDITION CODE     1                    ILC     2

                          GENERAL REGISTERS
    0  00000000     1  000000FF     2  00000000     3  00000000
    4  00000000     5  00000000     6  00000000     7  40000002
    8  00000000     9  00000000     A  00000000     B  00000000
    C  00000000     D  00000000     E  00000000     F  00000000


REQUEST=STORE
TYPE ALL OR PARTIAL
ALL
THE BYTES HAVE BEEN DUMPED ONTO FILE STORE
REQUEST=BYE
 EXIT
```

Vita

Leonard I. Horey, son of Helen and Henry Horey, was born on January 19,1951 in Newark, New Jersey. He did his undergraduate work at Lehigh University and in 1973 received the Degree of Bachelor of Science in Electrical Engineering with highest honors.

The author was a teaching assistant in the Department of Electrical Engineering throughout his graduate program.