A STRUCTURED MICROPROGRAM SET FOR THE SUMC COMPUTER TO EMULATE THE IBM SYSTEM/360 MODEL 50

by

CESAR R. GIMENEZ

A THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in

The Department of Electrical Engineering

of

The Graduate School

of

The University of Alabama in Huntsville

Huntsville, Alabama

1975

142

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE
P. O. Box 1247
Huntsville, Alabama 35807

THESIS APPROVAL FORM

Name of Candidate C. R. Gimenez

Major Subject Electrical Engineering

Title of Thesis A Structured Microprogram Set for the SUMC Computer to Emulate the IBM System/360 Model 50

Approved by:

Thesis Committee:

	Chairman Date
Date	Date
Date	Date

Department Chairman Date

Dean of School Date

Dean of Graduate Studies and Research Date

ABSTRACT

The thesis consists of an explanation of the similarities between regular and structured microprogramming; an explanation of machine branching architecture (particularly in the SUMC computer) required for ease of structured microprogram implementation; the implementation of a structured microprogram set in the SUMC to emulate the IBM System/360; and finally a comparison of the structured set with a nonstructured set previously written for the SUMC.

ACKNOWLEDGMENTS

The author wishes to thank Professor Charles A. Halijak; without his kind assistance this thesis would not have been published. The author wishes to acknowledge: Robert Asquith, of MSFC, who designed the SUMC, and Charles A. Williams, who collaborated with the author in the design and checkout of the original SUMC microcode set.

The author is grateful for the financial assistance for his undergraduate and graduate education provided by persons or institutions which follow: Cesar U. and Nora Gimenez (parents), Stetson University, U.S. Office of Education, Diocese of South Florida, Chrysler Corporation, Pratt and Whitney Aircraft, and Sperry Rand.

BIOGRAPHICAL SKETCH

Cesar Raul Gimenez, P.E. was born in [redacted]. In 1960 he moved to Miami, Florida, where he completed his secondary education. He holds a Bachelor in Electrical Engineering from Georgia Tech (1967). He has designed data acquisition, supervisory control, video, and computer systems. He is a registered Professional Engineer in Alabama and Florida.

PATENT NOTICE

DEDICATION

To Linda, Michael, and Teri.

CONTENTS

CONTENTS
(Concluded)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Concluded)

CHAPTER I

INTRODUCTION

1.1 Scope

This thesis assumes the reader is familiar with computer microprogramming. It is intended to be a practical document that bridges the gap between computer architecture and microprogramming. It is written so that it appeals to those interested in computer architecture, computer languages, and microprogramming.

A summary of the similarities between regular and structured microprogramming is given in Chapter II. The mathematical foundations for these are discussed in depth in the works of Dijkstra, Glushkov, Ito, and Mills given in the References. The basic architectural structures needed to implement structured microprograms are explained in Chapter III. The SUMC, a versatile aerospace microprogrammable computer developed by NASA, is introduced in Chapter IV. Then, the basic structured primitives for the SUMC are developed in Chapter V. These are used, throughout Chapter VI, to implement a subset of structured microprograms which emulate the IBM System/360. Finally, the structured microprograms are compared with the corresponding nonstructured microprograms.

1.2 Background of Thesis

The Data System Laboratory, Marshall Space Flight Center (MSFC), Huntsville, Alabama, is engaged in the development of a family of computers known as Space Ultrareliable Modular Computers (SUMC). The main guideline in the SUMC program is to develop a microprogrammable computer family capable of emulating existing ground commercial computers. As an employee of Sperry Rand, Huntsville, Alabama, supporting Data Systems Lab (under contract No. NAS8-21812), the author was assigned to write the microcode for the SUMC Breadboard (SUMC BB) used in the emulation of an IBM System/360. The original microcode set [20] occupied 1700 memory locations. Delivery delays of a similar microcode memory, to be used in the development of a SUMC input/output processor (IOP), forced MSFC to require the SUMC microcode be rewritten within a 1K (1024) memory module, to free the other SUMC BB memory module for IOP use. The SUMC microcode was rewritten to fit within a 1K memory module. Careful attention was exercised so that not only microcode size requirements were met, but also the resulting microcode instruction set was faster in execution time.

The author has had an interest in regular expressions since 1968. While reading articles on regular microprogramming, he noticed great similarities between regular and structured programming. Structured programming guidelines, have been issued by MSFC to be used in the SUMC software development program. Similar guidelines, however, have not been issued in the area of microprogramming. This thesis explores the use of structured microprogramming in the SUMC Breadboard. This objective was not brought about from a mere desire to illustrate structured microprogramming implementation, but rather to illustrate structured microprogramming implementation under realistic design constraints. Most of the current literature on structured programming concepts deals with high level language implementation. Since high level languages are, by design, machine independent, the application of structured programming techniques to them is rather straightforward. On the other hand, microprogramming is machine and real time dependent. Thus, the application of structured programming techniques in this area is more difficult. But most important, all design is subject to unpredictable restrictions (such as those caused by the delivery delay discussed above). Thus, the author's approach was to rewrite the microcode used in the SUMC BB emulation of the IBM System/360, with similar constraints. He purposely, preserved the sequence of events within the microprograms whenever possible. In this manner (with the algorithms as a fixed parameter), the two microcode sets could be more easily compared. At the same time, he was insured that the structured microprogram set was a valid one. That is, once this set is assembled into machine language and loaded into memory, it will successfully emulate an IBM System/360 in the problem state.

1.3 Objectives of Thesis

The objectives of this thesis are as follow:

Objective 1. To point out the similarities between regular and structured microprogramming. This objective was fulfilled.

Objective 2. To study the branching architecture required for ease of structured microprogramming implementation. This objective was also fulfilled.

Objective 3. To write a complete structured microcode set for the SUMC BB for the emulation of an IBM System/360.

This objective was partially but adequately fulfilled. Rather than rewritting the complete microcode set, the author selected microprograms subsets which exemplify the implementation techniques and point out the main differences between the sets. By doing this, he eliminated many repetitious sections from the thesis.

Objective 4. To compare the size, speed, and ease of implementation of the structured and nonstructured microprogram sets. This objective was fulfilled.

CHAPTER II

MICROPROGRAMMING

2.1 The Microprogrammable Computer

Consider a simplified computer model consisting of two blocks: an operational and a control automaton shown in Fig. 2.1. The output of the operational automaton is the string of values of the logical conditions (predicates) p_1, p_2, ..., p_n. If the current state s_j of the automaton belongs to S, then the logical condition is taken to be satisfied ($p_j = 1$). Otherwise, the condition is not satisfied ($p_j = 0$). The output signals (microoperations) of the control automaton (m_1, m_2, ..., m_n) are identified with certain transformations of the set of states of the automaton. That is, m_j: $S \rightarrow S$. Note that the output of the control automaton corresponds to an input string of the operational automaton.

A microprogrammable computer is a structure:

$$C = \left\{ S,I,O,M,P \right\}$$

of partial functions for which there exist: The set of states, S; the set of input sequences, I; the set of output sequences, O; the set of microprograms, M; and the set of predicates, P. The input function I is related to S by a mapping of I into S. Thus, the input function, I: $I \rightarrow S$. Similarly, for the output function, O: $S \rightarrow O$; for the microprogram set, M: $S \rightarrow S$; and for the predicate set, P: $S \rightarrow (0,1)$.

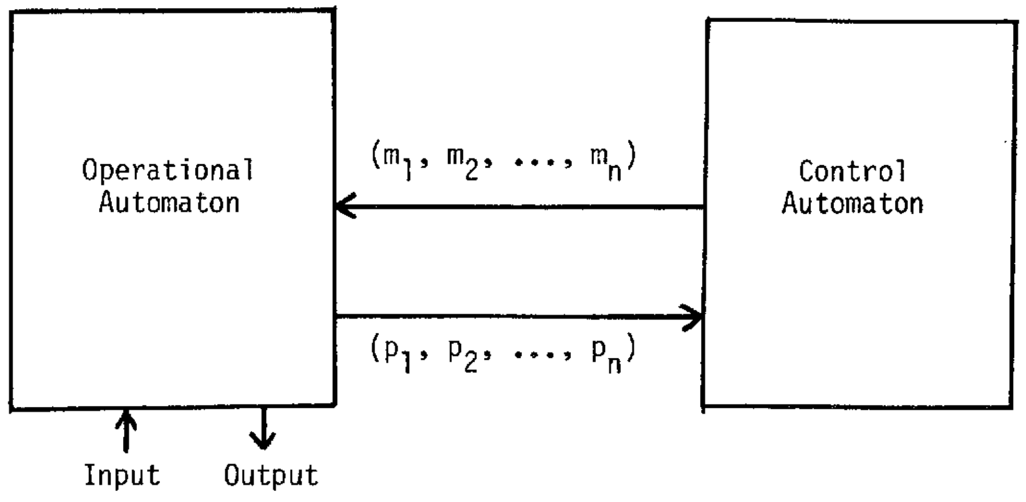


Figure 2.1. Microprogrammed Computer Model

2.2 Formal Definition of Regular Microprograms

A class of regular microprograms [8] can be defined recursively as follows:

1. Individual microoperations $(e, m_1, m_2, \ldots, m_n)$ are regular microprograms. The NO-OPERATION microoperation is e.

2. If x and y are regular microprograms, then regular microprograms are formed by the following rules:

   a. Concatenation

      $x \cdot y$; meaning x followed by y

   b. Decision expressions

      $< x \vee y >_p$; meaning $p \cdot x \vee \tilde{p} \cdot y$

   c. Iteration

      $[\, x \,]_p$; meaning while p do x

3. Where p is a predicate term as defined below:
   a. Individual predicates (T, F, p_1, p_2, ..., p_n) are predicate terms.
   b. If x is a regular microprogram and p is a predicate, then $x \cdot p$ is a predicate term.
   c. If p and q are predicate terms, then $p \vee q$, $p \wedge q$, $\tilde{p}$ are predicate terms.

2.3 Regular Microprogram Meaning

A regular microprogram has meaning only when an interpretation is given. An interpretation for regular microprograms is specified by a computer $C = \{S, I, O, M, P\}$ on which a regular microprogram is represented. For any state s of S, regular microprograms will be given to the following meaning:

1. $e(s) = s$
2. $x \cdot y(s) = y(x(s))$
3. $< x \vee y >_p(s)$ = if $p(s)$ then $x(s)$, else $y(s)$
4. $[\ x\ ]_p(s)$ = while $p(s)$ do $x(s)$
5. $T(s)$ = true; $F(s)$ = false
6. $x \cdot p(s) = p(x(s))$
7. $(p \vee q)(s)$ = if $p(s)$ then true, else $q(s)$
8. $(p \wedge q)(s)$ = if $p(s)$ then $q(s)$, else true
9. $(p \rightarrow q)(s)$ = if $p(s)$ then $q(s)$, else false
10. $(\tilde{p})(s)$ = if $p(s)$ then false, else true
11. $p(s) \cdot q(s) = q(s) \cdot p(s) = p(q(s)) = q(p(s))$

2.4 Practical Aspects of Regular Microprogramming

But why should one be concerned with regular microprograms? As it is clearly shown in the works of Glushkov [3, 4]: "Any microprogram can be represented in regular form. There exists an algorithm for transformation of arbitrary microprograms written in ordinary form into regular form". Ito [8] has shown that any microprogram can be written in an ordinary flowchart or an automaton diagram, and that this flowchart or automaton diagram can be represented by a linear system of equations of regular microprograms. Furthermore, any linear system of equations of regular microprograms can be solved and its solutions are in the class of regular microprograms.

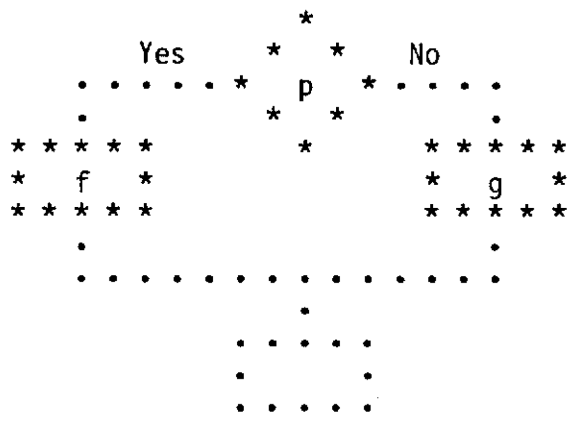
2.5 Structured Microprogramming

Structured programming is currently receiving a great deal of attention. The theorems providing the mathematical foundations of structured programming given by Mills [14] are: The structure Theorem ("Any flowchartable program logic can be represented by the expansions of as few as three types of structures"), the TOP-DOWN Corollary ("structured programs can be written or read top-down"), the Correctness Theorem (under certain conditions, "a program can be proved correct by a tour of its program tree"), and the Expansion Theorem which gives the rules for top-down decomposition. The three types of control structures which are usually used as a basis for the control structures of flowchartable programs are:

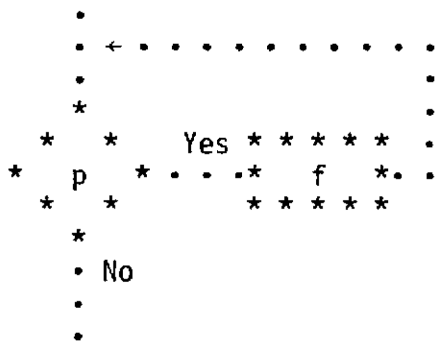
1. f then g

```
* * * * *   * * * * *
*   f   * → *   g   *
* * * * *   * * * * *
```

2. If p then f, else g



3. While p do f



Structured microprogramming is defined as the implementation of microprograms utilizing only concatenation, decision expressions, and iteration primitives.

It is obvious that the three structured microprogramming primitives correspond to the expressions given under Rule 2 in the formal definition of regular microprograms. Note that f corresponds to x, and g corresponds to y.

2a. Concatenation

$x \cdot y \qquad \rightleftarrows \qquad$ f then g

2b. Decision expressions

$< x \vee y >_p$ $\rightleftarrows$ If p then f, else g

2c. Iteration Do while

$[\, x \,]_p$ $\rightleftarrows$ While p, do x

When implementating structured microprogramming (or programming in general) it is common practice to implement the iteration primitive $[\, x \,]_{\tilde{p}}$ (while not p do x), instead of $[\, x \,]_p$. This is done because it is easy to load some iteration counter with a positive value $(1 < v < N)$ prior to entering the iteration. The iteration count is decremented by a fixed number (usually 1) with each pass through the iterative loop, until the iteration count is made zero (or sometimes negative) prior to exiting the primitive. Refer to Section 5.3. Note that either $[\, x \,]_p$ or $[\, x \,]_{\tilde{p}}$ can be used as a primitive without losing the validity of the conclusions previously made. The iteration primitive when implemented is always finite. That is, a finite number of passes are made prior to exiting the iteration.

CHAPTER III

STRUCTURED MICROPROGRAMMING BRANCHING ARCHITECTURE

3.1 Machine Dependence

Microprogramming is machine dependent. Thus, the ease with which one can implement structured microprograms in a given microprogrammable computer is closely governed by the machines' branching architecture. Implementation of the aforementioned control primitives is significantly more difficult at the macro level because specific return addresses must be saved within the macros.

3.2 Ease of Implementation

The relationship between branching architecture and ease of structured microprogramming implementation is best shown by example. In the following pages three machines (X, Y, and Z) will be discussed. A table summarizes the machine commands used in microinstruction sequencing. Each control primitive is implemented for each machine both at the micro and macro level. For each primitive, a microprogram and a macroprogram are given. The programs are further illustrated by the corresponding micro and macro flowcharts. The following abbreviations are used for the sequencer and alternate sequencer registers: SEQ and ASEQ respectively. Two other utility registers LITERAL (LIT) and TEMPORARY (T) are used.

TABLE 3.1

Successor Commands for Machine X

COMMAND	NEST INSTRUCTION ADDRESS	RETURN ADDRESS
STEP	(SEQ) + 1	--------------
SKIP	(SEQ) + 2	--------------
SAVE	(SEQ) + 1	(SEQ) → (ASEQ)
CALL	(ASEQ) + 1	(SEQ) → (ASEQ)
JUMP	(ASEQ) + 1	--------------
RETN	(ASEQ) + 2	--------------
WAIT	(SEQ)	--------------

The successor commands for Machine X [16], are given in Table 3.1. The default successor command is STEP. The machine has two pointers into control memory, SEQ and ASEQ. It has subroutine capabilities provided by the CALL and RETN commands. Both true and false successors can be specified in each instruction. Figures 3.1 through 3.6 further illustrate the implementation of structured primitives given in the text for Machine X.

At the micro level, the structured microprogram primitives can be easily implemented. However, implementation of these primitives at the macro level is more difficult because specific return addresses must be inserted within the macros.

Primitive: f then g

Microprogram

```
f, STEP
g
```

```
• • • • •
• ENTRY •
• • • • •
    •
    •
* * * * *
*   f   *
* * * * *
    • STEP
    •
* * * * *
*   g   *
* * * * *
```

Figure 3.1. Concatenation Microflowchart for Machine X

Primitive: f then g

Macroprogram

f_1, STEP

f_2, STEP

•

•

•

f_n, STEP

G_1 - 1 =: ASEQ

JUMP

G_1: g

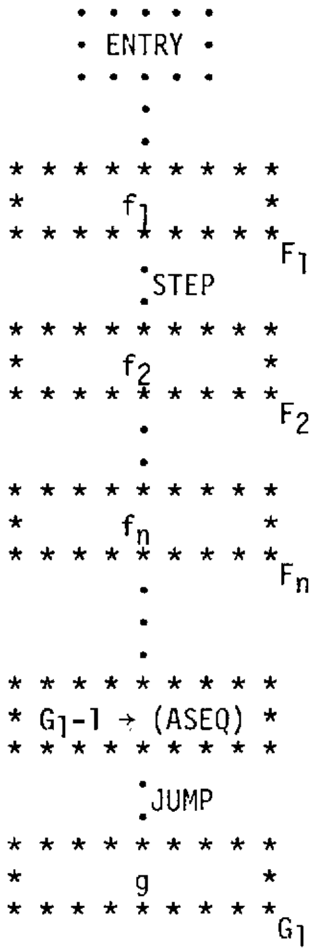


Figure 3.2 Concatenation Macroflowchart for Machine X

Primitive:	If p then f, else g
Microprogram	
	If p then SKIP, else STEP
G:	g, SKIP
F:	f, STEP

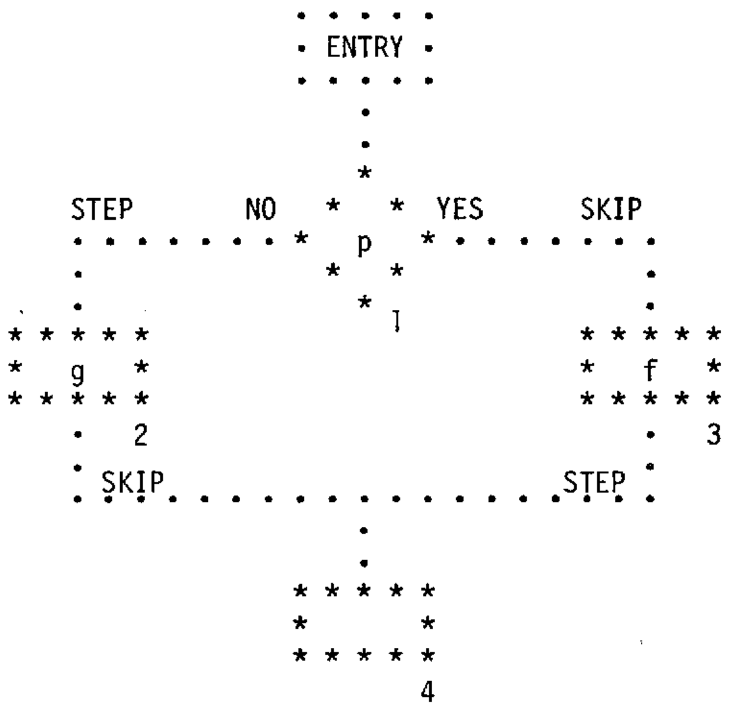


Figure 3.3. Decision Microflowchart for Machine X

```
Primitive:      If p then f, else g

Macroprogram
                F_1 - 1 =:  ASEQ

                If p then CALL, else STEP

                G_1 - 1 =:  ASEQ

                If (not p) then CALL, else STEP

     F:         f:  f_1
                    f_2
                     ⋮
                    f_n-1
                    f_n, RETN
                    ------

     G:         g:  g_1
                    g_2
                     ⋮
                    g_m, JUMP
```

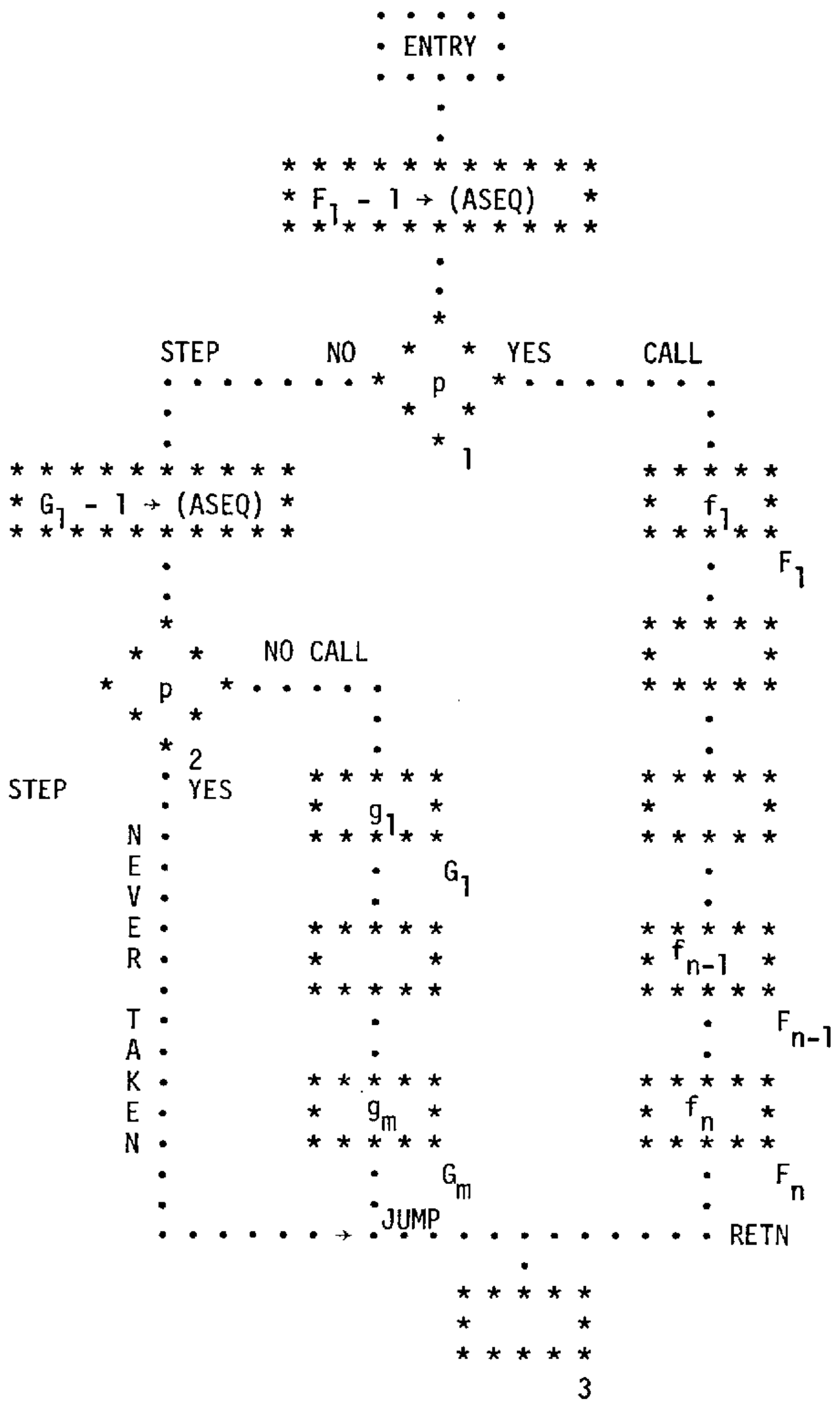


Figure 3.4. Decision Macroflowchart for Machine X

```
Primitive:    While p̃, do f

Microprogram
              ROM - 1 =:  ASEQ

     ROM:     If (not p) then STEP, else SKIP

              f, JUMP

              g
```

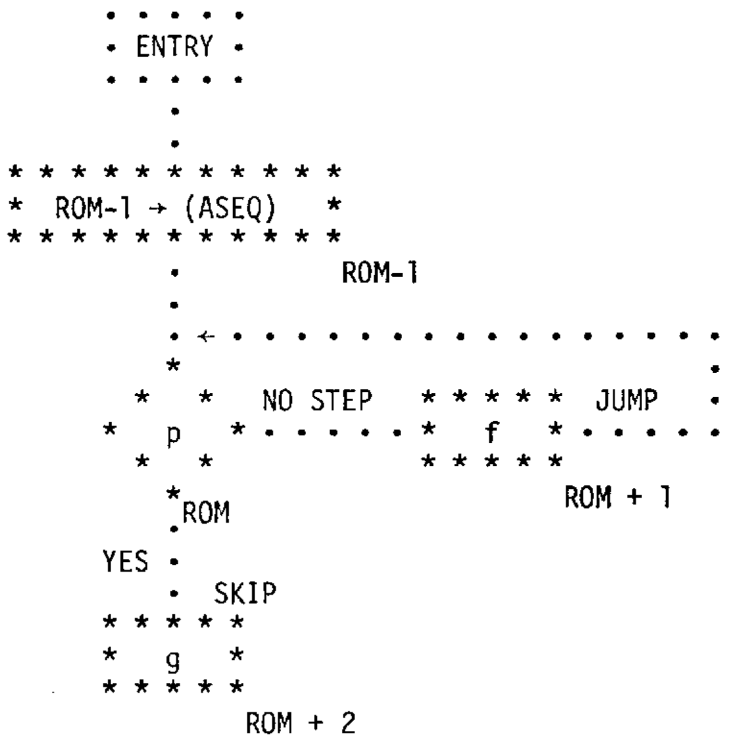


Figure 3.5. Iteration Microflowchart for Machine X

Primitive: While $\tilde{p}$, do f

Macroprogram

```
     ROM:      F_1 - 1 =:  ASEQ
               If (not p) then JUMP, else STEP
     G:        g
     F:        f:  f_1
                   f_2
                   :
                   f_n-1
               ROM - 2 =:  ASEQ
               f_n, JUMP
```

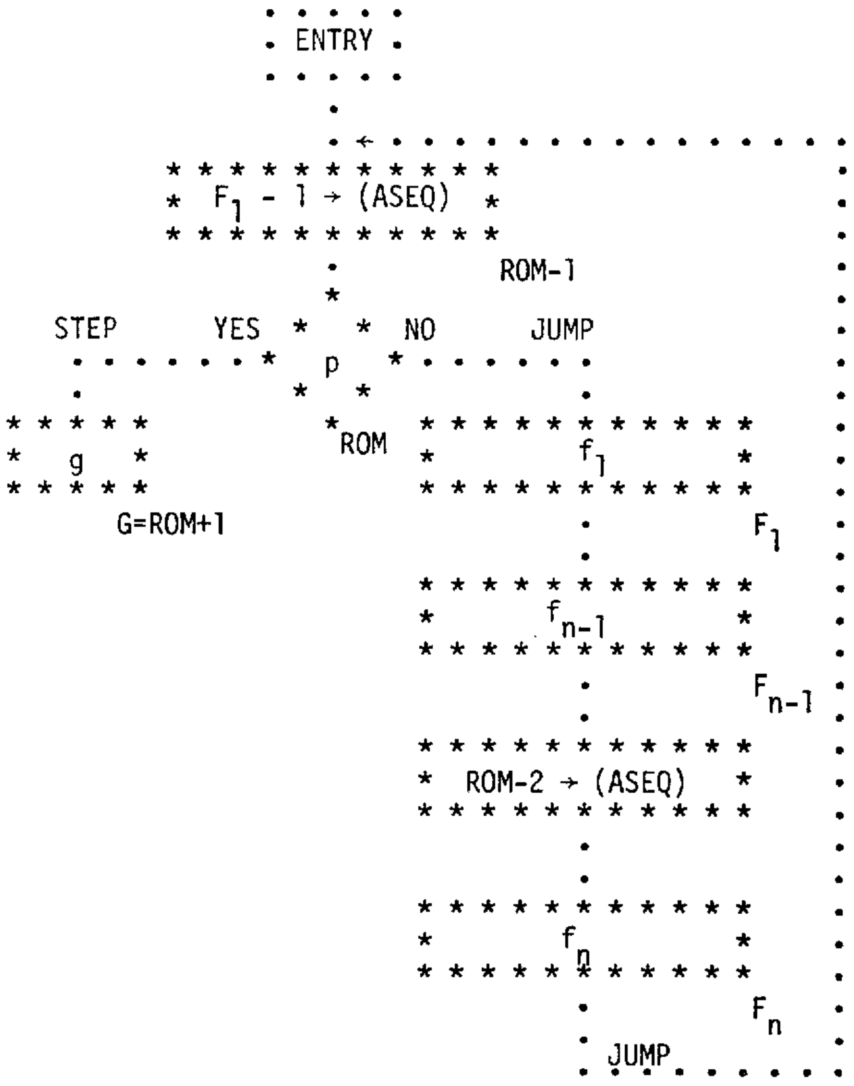


Figure 3.6. Iteration Macroflowchart for Machine X

TABLE 3.2

Successor Commands for Machine Y

COMMAND	NEXT INSTRUCTION ADDRESS	RETURN ADDRESS
STEP	(SEQ)+1	----------------
SKIP	(SEQ)+2	----------------
SAVE	(SEQ)+1	(SEQ)+1 → (ASEQ)
CALL	(LIT)	(SEQ)+1 → (ASEQ)
JA	(ASEQ)	----------------
JL	(LIT)	----------------
JT	(T)	----------------

The successor commands for Machine Y [17] are given in Table 3.2. The machine has four pointers into control memory: SEQ, ASEQ, LIT, and T. The STEP, SKIP, SAVE, and CALL commands are similar to those of Machine X. Only one successor can be specified in each microinstruction. The default successor is always STEP. Machine Y has three JUMP commands. At the macro level, the existence of multiple JUMP commands simplifies returns from the function f and facilitates modular decomposition of the microcode. However, specific return addresses must still be inserted within the macros.

Primitive: f then g

Microprogram

```
f, STEP
g
```

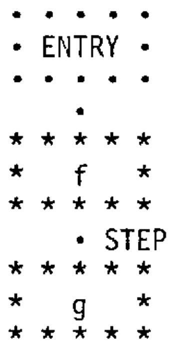


Figure 3.7. Concatenation Microflowchart for Machine Y

```
Primitive:     f then g

Macroprogram
               f₁, STEP
               :
               fₙ, STEP
               G₁ =:  ASEQ, JA
               - - - - - -
    G₁:        g
```

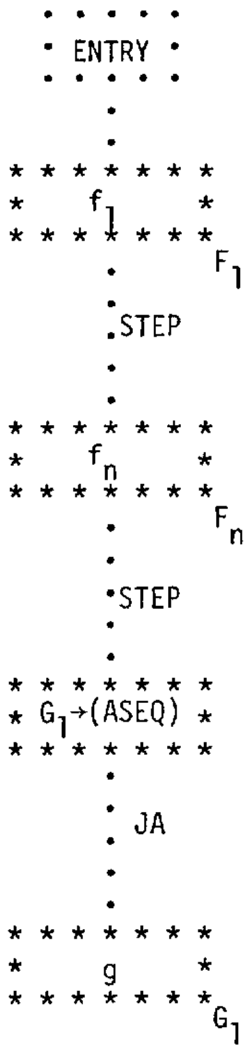


Figure 3.8. Concatenation Macroflowchart for Machine Y

Primitive: If p then f, else g

Microprogram

```
If p then SKIP, else STEP
g, SKIP
f, STEP
```

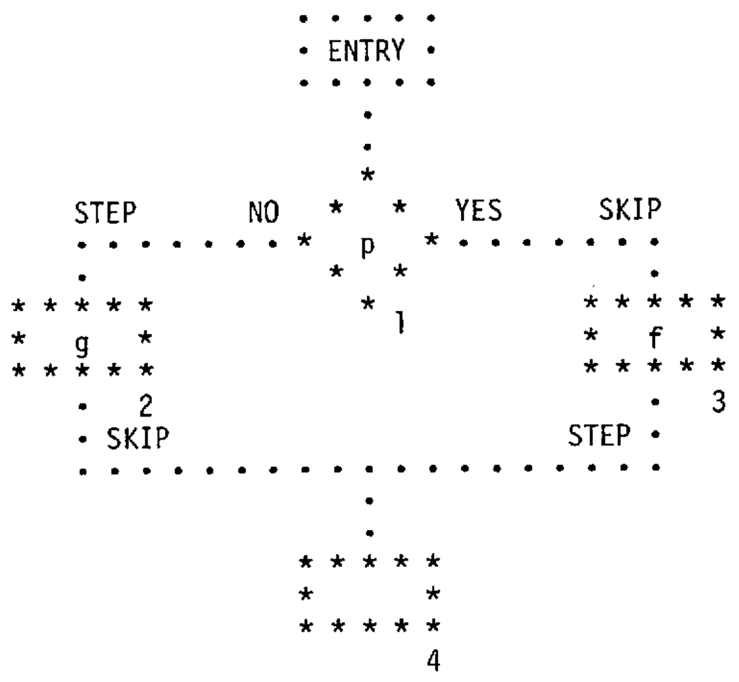


Figure 3.9. Decision Microflowchart for Machine Y

Primitive: If p then f, else g

Macroprogram

F_1 =: LITERAL
If p then CALL, else STEP
G_1 =: LITERAL
If (not p) then CALL, else STEP

F: f: f_1
⋮
f_{n-1}
F_n, JA
- - - - - -

G: g: g_1
⋮
g_m, JA

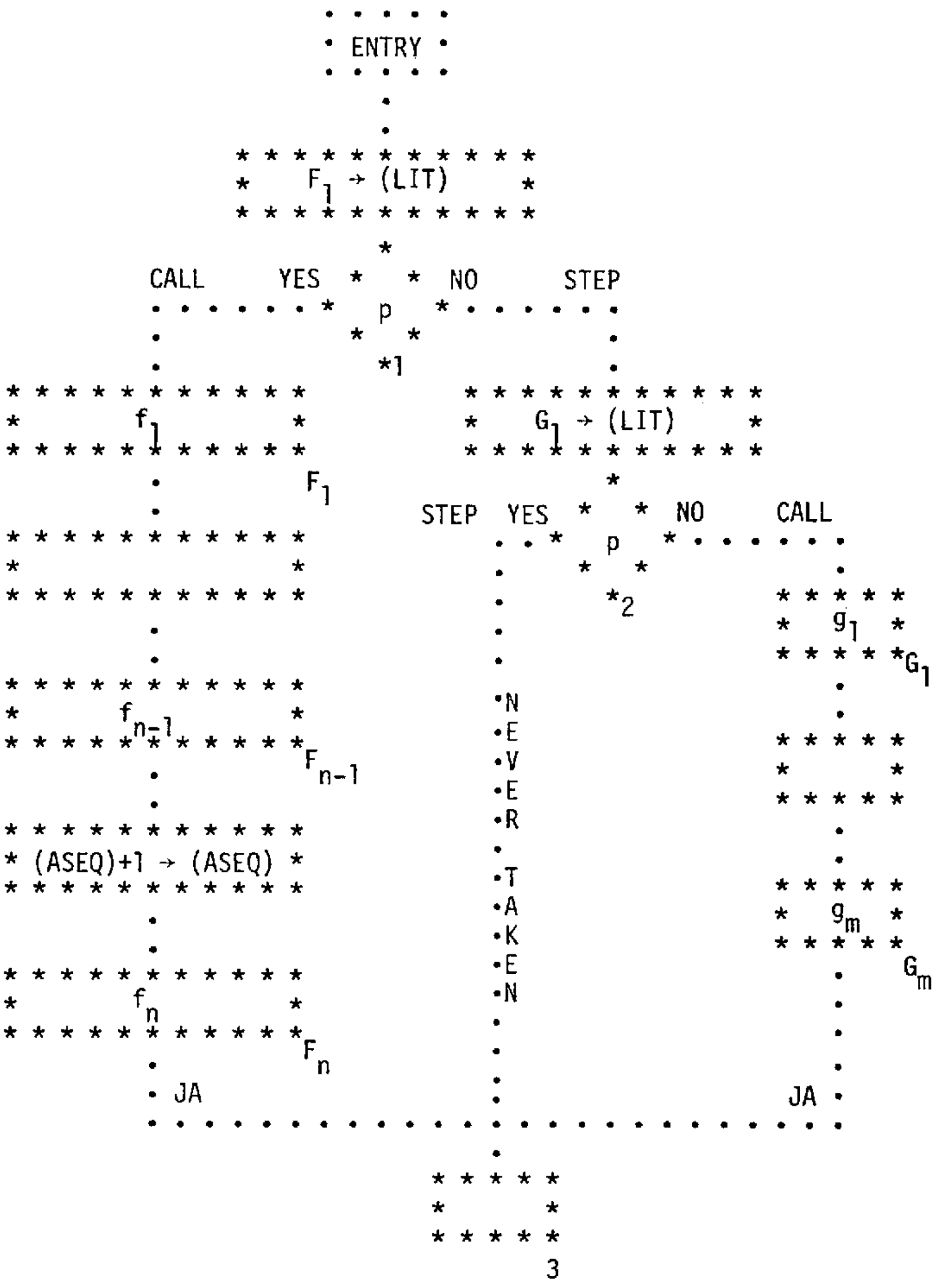


Figure 3.10. Decision Macroflowchart for Machine Y

```
Primitive:     While p̃, do f

Microprogram
               ROM =:  ASEQ
     ROM:      If (not p) STEP, else SKIP
               f, JA
               g
```

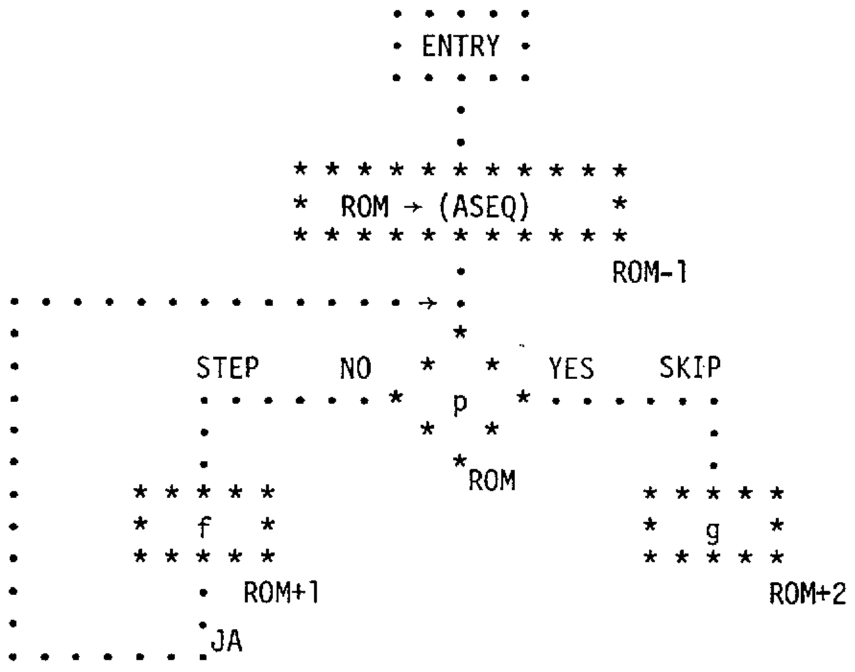


Figure 3.11. Iteration Microflowchart for Machine Y

```
Primitive:     While p̃, do f

Macroprogram
               F1 =:  ASEQ, STEP
               ROM =:  LIT
     ROM:      If (not p) then JA, else STEP
               - - - - - -
     F1:       f:  f1
                   :
                   fn, JL
```

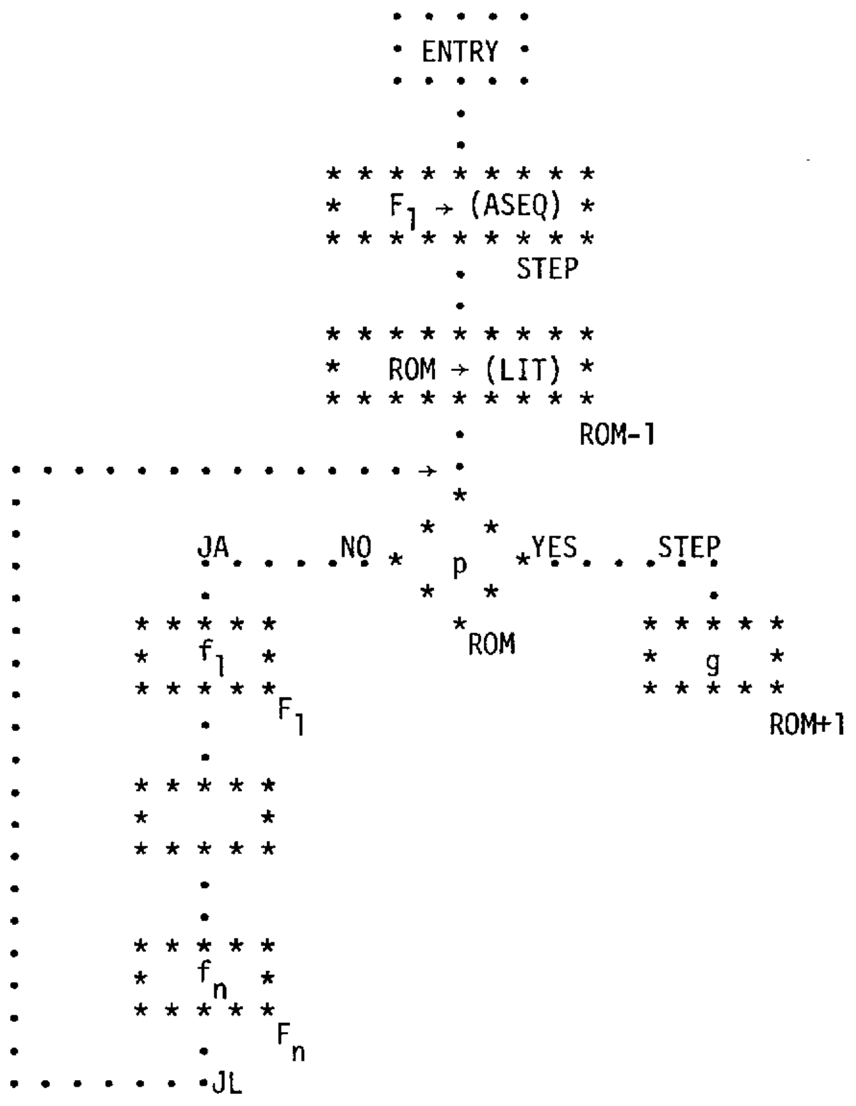


Figure 3.12. Iteration Macroflowchart for Machine Y

The successor commands for Machine Z are given to Table 3.3. At the micro level, the implementation of structured primitives is identical to that of Machine Y. However, at the macro level, the implementation of the decision primitive is considerably easier since this successor set allows for multiple return addresses to be specified outside of the macro instruction blocks. Thus, substantial improvements in microprogramming efficiency and a reduction in control memory size are possible with Machine Z.

Microprogrammable machines are not easy to program structuredly unless the branching architecture capabilities provide for easy implementation of the structured primitives. In particular, a successor set consisting of STEP, SKIP, SAVE, two CALL, and two JUMP commands, should be sufficient to implement structured microprograms with ease.

TABLE 3.3

Successor Commands for Machine Z

COMMAND	NEXT INSTRUCTION ADDRESS	RETURN ADDRESS
STEP	(SEQ)+1	----------------
SKIP	(SEQ)+2	----------------
SAVE	(SEQ)+1	(SEQ)+1 → (ASEQ)
CALLA	(ASEQ)	(SEQ)+1 → (ASEQ)
CALL	(LIT)	(SEQ)+1 → (ASEQ)
JA	(ASEQ)	----------------
JL	(LIT)	----------------

Primitive: f then g

Microprogram

```
f, STEP
g
```

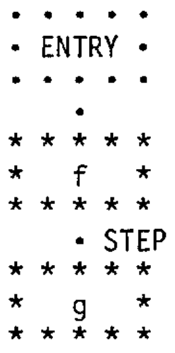


Figure 3.13. Concatenation Microflowchart for Machine Z

Primitive: f then g

Macroprogram

F_1: f_1, STEP

⋮

f_n, STEP

G_1 =: ASEQ, JA

- - - - - -

G_1: g

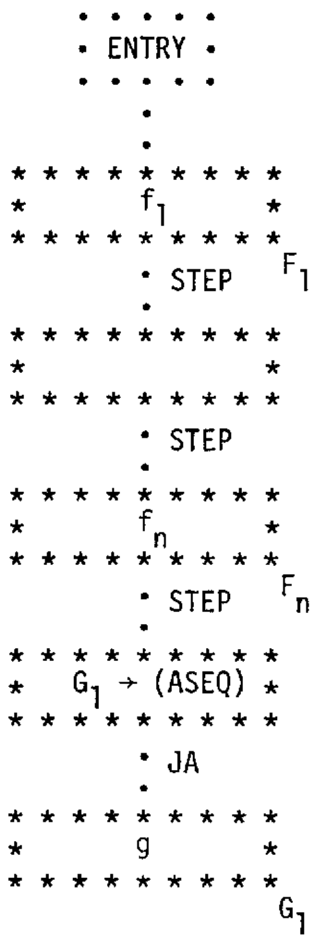


Figure 3.14. Concatenation Macroflowchart for Machine Z

Primitive: If p then f, else g

Microprogram

```
        If p then SKIP, else STEP
G:      g, SKIP
F:      f, STEP
```

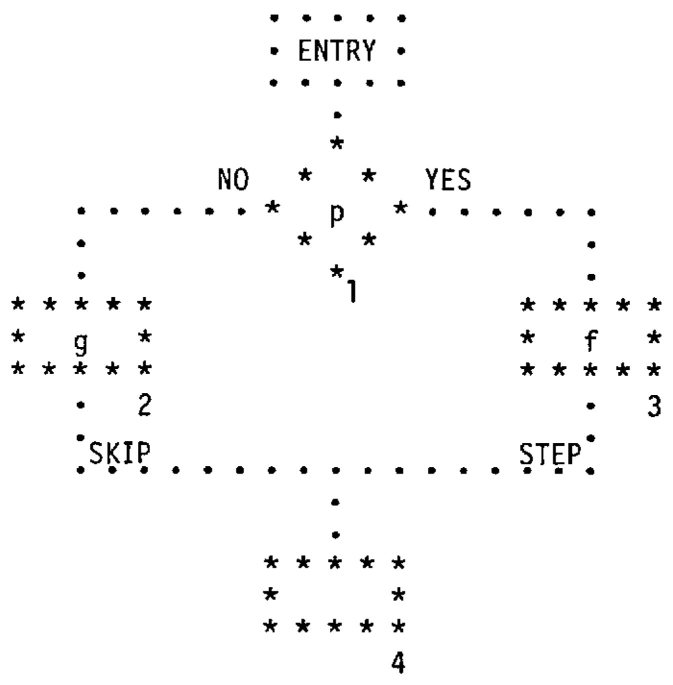


Figure 3.15. Decision Microflowchart for Machine Z

Primitive: If p then f, else g

Macroprogram

F_1 =: ASEQ

G_1 =: LIT

If p then CALLA, else CALL

F_1: f: f_1

$\vdots$

f_n, JA

- - - - - -

G_1: g: g_1

$\vdots$

g_m, JA

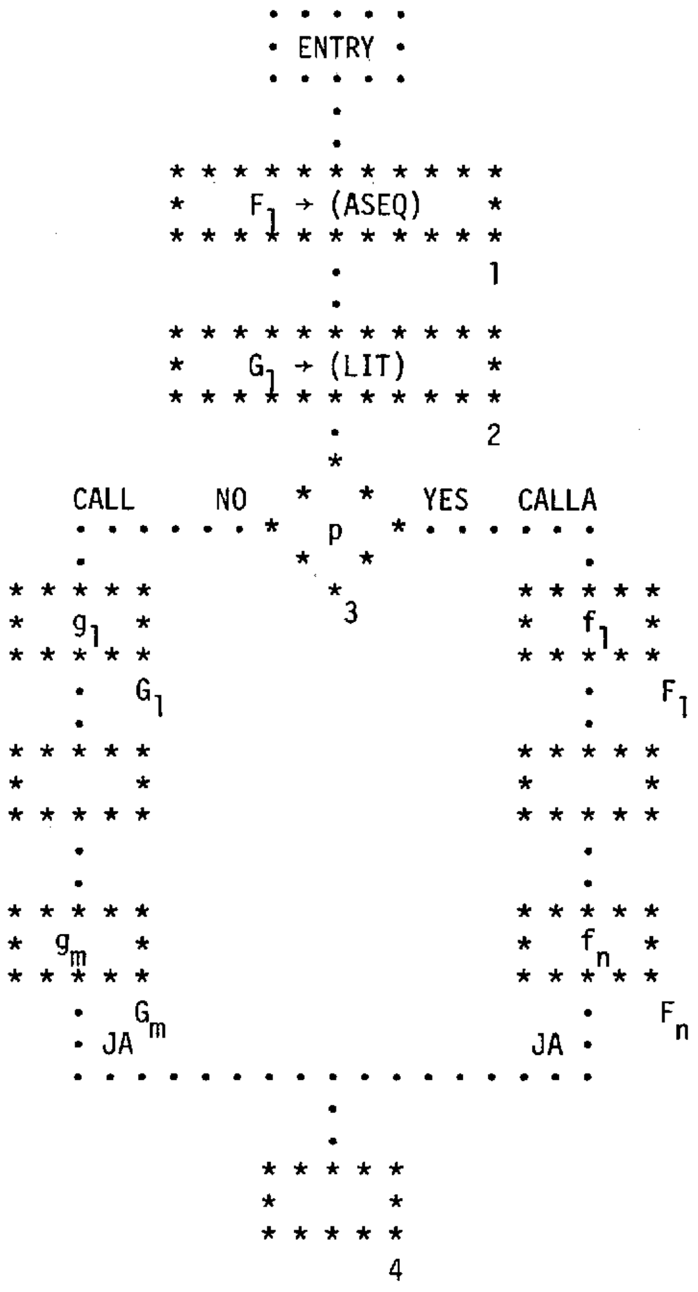


Figure 3.16. Decision Macroflowchart for Machine Z

```
Primitive:      While p̃, do f

Macroprogram
                ROM =:  ASEQ

     ROM:       If (not p) then STEP, else SKIP
     F:         f, JA
     G:         g
```

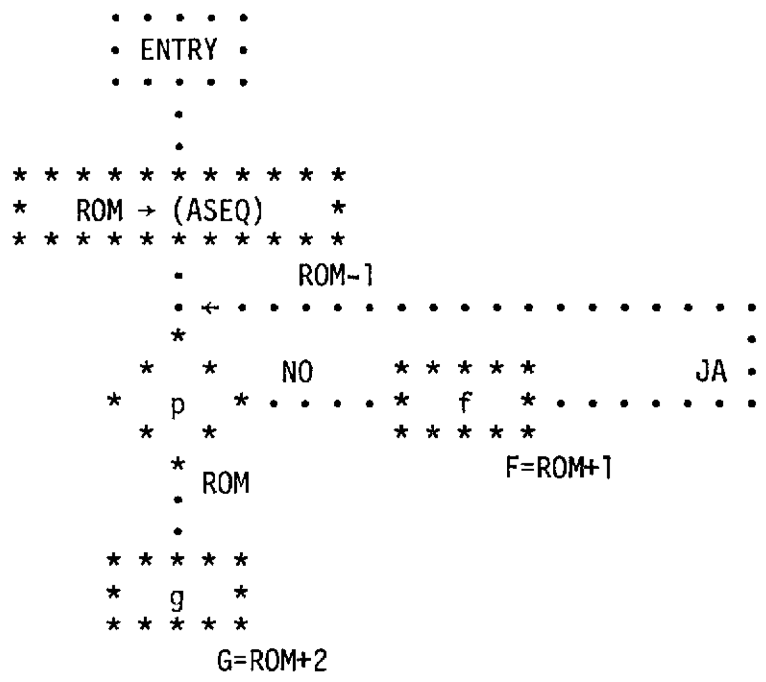


Figure 3.17. Iteration Microflowchart for Machine Z

```
Primitive:      While p̃, do f

Macroprogram
                F1 =:  ASEQ

                ROM =:  LIT

     ROM:       If (not p) then JA, else STEP
     G:         g, STEP
                - - - - - -
     F1:        f:  f1
                    :
                    :
                    f(n-1)
                    fn, JL
```

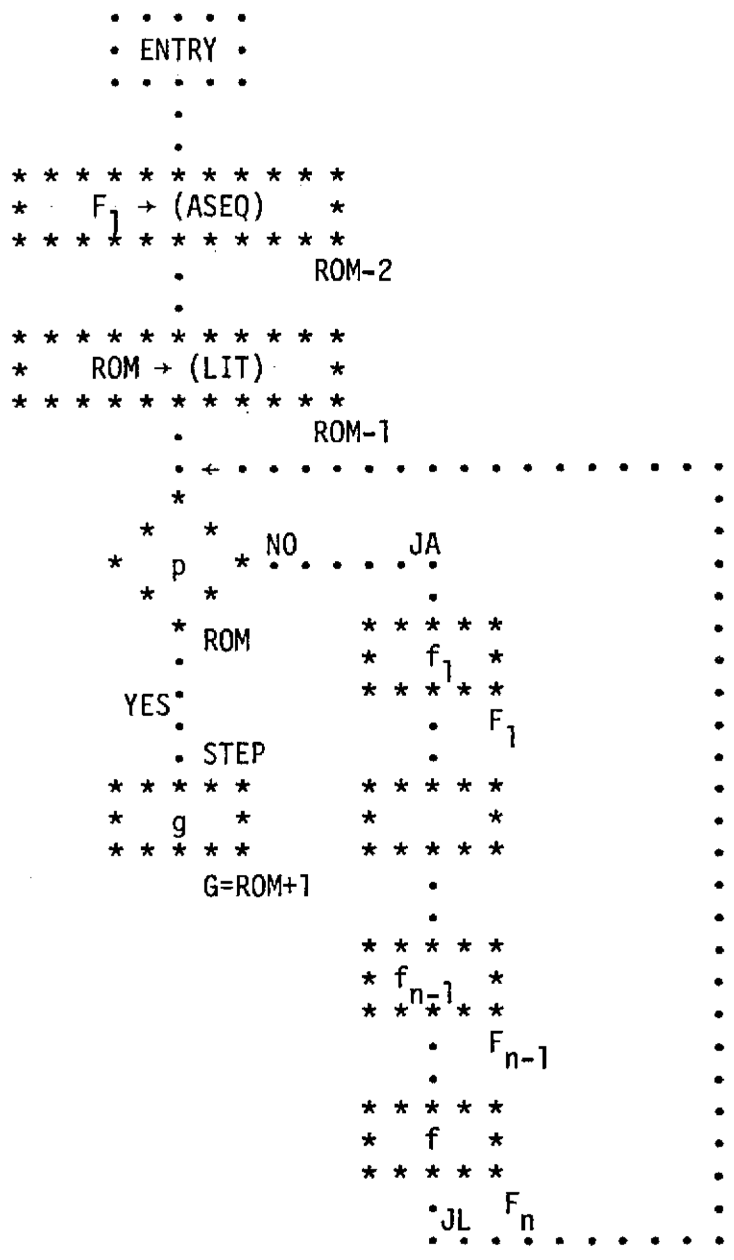


Figure 3.18. Iteration Macroflowchart for Machine Z

CHAPTER IV
SPACE ULTRARELIABLE MODULAR COMPUTER (SUMC)

4.1 General Description

The simplified SUMC block diagram shown in Fig. 4.1 depicts SUMC's logical construction. Six major logic blocks are shown and are briefly described herein to provide the reader a better understanding of SUMC's microprogrammed control. These logic blocks are the Main Memory Unit, the Scratch Pad Memory, the Arithmetic Logic Unit, the Multiplexer Register Unit, the Floating Point Unit, and the Control Unit.

The Main Memory Unit (MMU) is a 32 bit plated wire memory which is used to store program instructions and data. The memory addressing scheme allows access of up to 4 billion words.

Scratch Pad Memory (SPM) consists of sixty-four 32 bit registers. Factors which determine SPM use and allocation are:

1. Instruction format and repertoire.
2. Memory addressing scheme.
3. I/O and interrupt processing scheme.

Scratch Pad Memory register assignments for the 32 bit floating point SUMC Breadboard are shown in Fig. 4.2.

The Arithmetic Logic Unit (ALU) consists of three multiplexers and two parallel arithmetic units. Multiplexers are used to select the data source(s) for the two arithmetic units which perform the required logical or arithmetic operation.

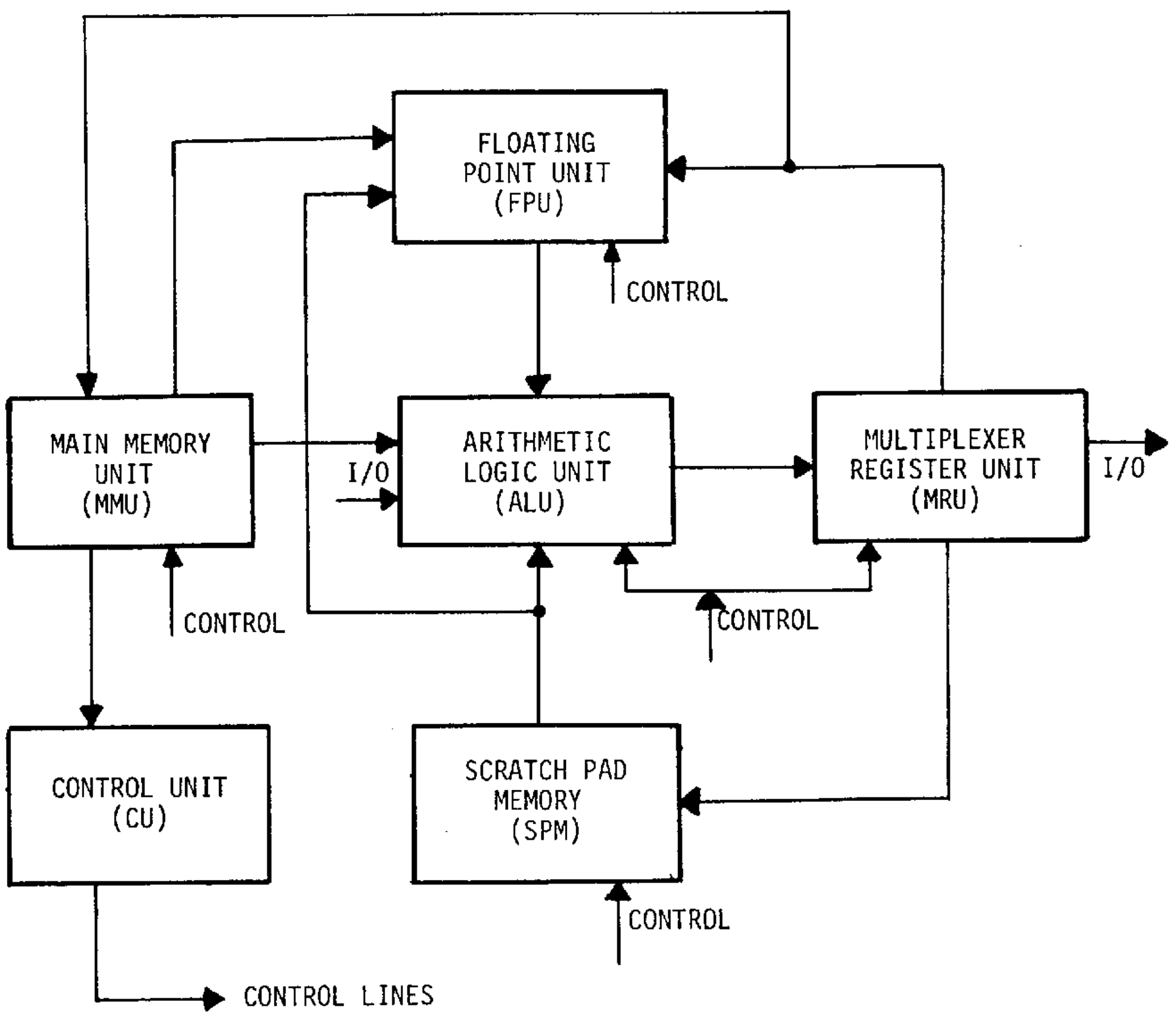


Figure 4.1. Simplified SUMC Block Diagram

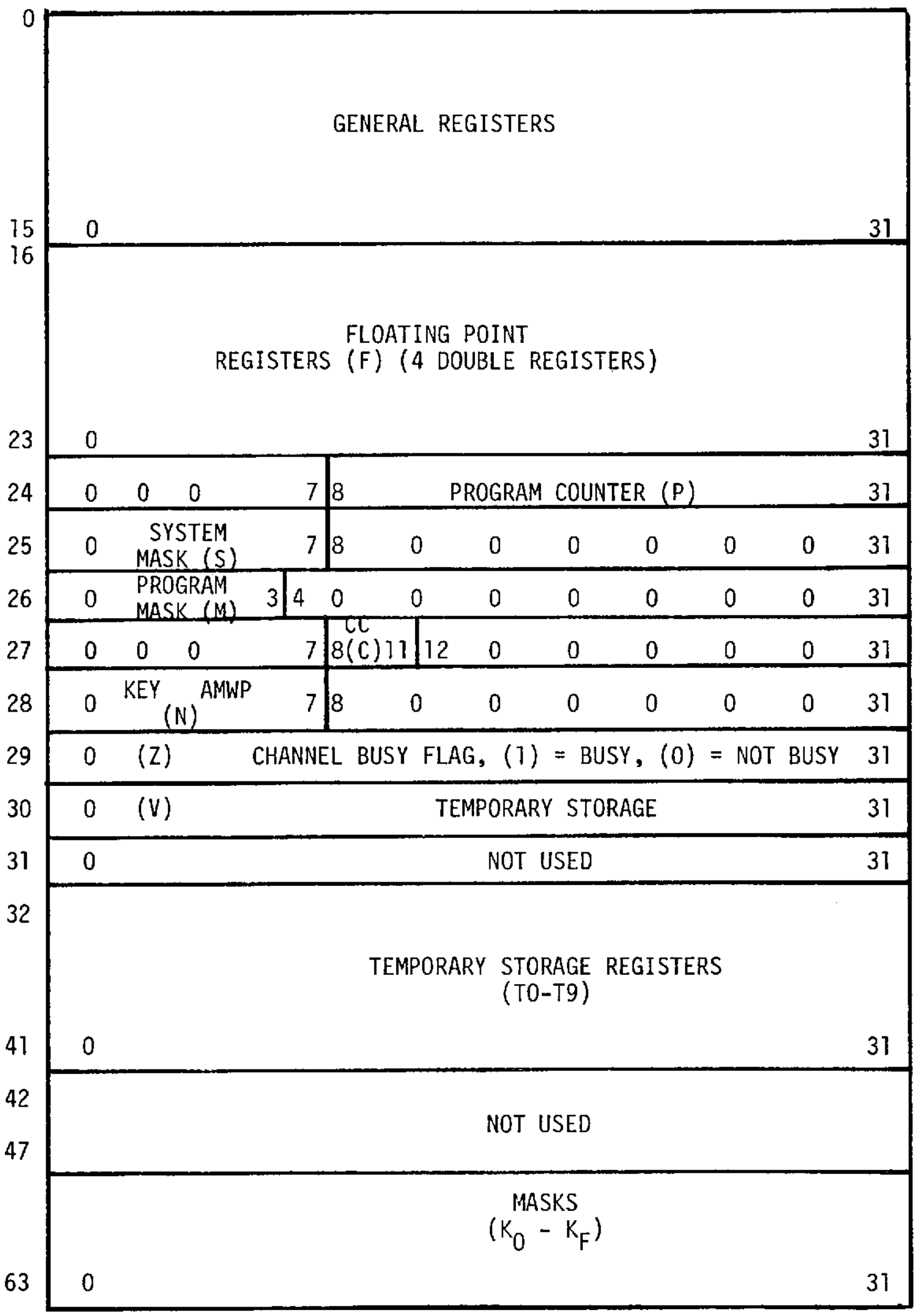

0
GENERAL REGISTERS
15 0 31
16
FLOATING POINT
REGISTERS (F) (4 DOUBLE REGISTERS)
23 0 31
24 0 0 0 7 8 PROGRAM COUNTER (P) 31
25 0 SYSTEM MASK (S) 7 8 0 0 0 0 0 0 31
26 0 PROGRAM MASK (M) 3 4 0 0 0 0 0 0 0 31
27 0 0 0 7 8 CC (C) 11 12 0 0 0 0 0 31
28 0 KEY AMWP (N) 7 8 0 0 0 0 0 0 31
29 0 (Z) CHANNEL BUSY FLAG, (1) = BUSY, (0) = NOT BUSY 31
30 0 (V) TEMPORARY STORAGE 31
31 0 NOT USED 31
32
TEMPORARY STORAGE REGISTERS
(T0-T9)
41 0 31
42
NOT USED
47
MASKS
(K_0 - K_F)
63 0 31

Figure 4.2. Sample Scratch Pad Memory Map

The Multiplexer Register Unit (MRU) consists of three multiplexers and three registers. The MRU is used to transfer data from the ALU to the Main and Scratch Pad Memory units, and to retain the results of intermediate microinstructions during microprogram execution.

The Floating Point Unit (FPU) consists of a 32 bit multiplexer, an 8 bit Exponent Arithmetic Logic Unit (EALU), and an 8 bit Exponent Register (ER). The FPU is used for computing the characteristic, and for normalization, of floating point numbers.

The Control Unit (CU) decodes the program instruction and provides the ALU, SPM, MRU, FPU and MMU control required to execute the instruction. The major units within the CU and their function are:

1. Instruction Register (IR)
   The IR is a 32 bit register which contains the computer instruction currently being executed. The contents of the 8 bit operation code field is used to address the IAROM.
2. Instruction Address Read Only Memory (IAROM)
   The IAROM is a 256 word 24 bit read only memory which contains the starting addresses of the computer instruction microprogram stored in the MROM, and additional instruction format control bits. The 8 bit operation code of each computer instruction specifies the IAROM location which contains the MROM starting address for the microprogram that must be executed to perform the computer instruction. The 10 least significant bits of the addressed IAROM location are gated to the Sequence Control Unit (SCU) for MROM addressing.

3. Sequence Control Unit (SCU)

   The SCU functions as an address register for the Microprogram Read Only Memory (MROM). The value contained in the SCU specifies an MROM location (microinstruction) to be broadcast for SUMC control. SCU values are modified during microinstruction execution to provide microprogram sequencing.

4. Microprogram Read Only Memory (MROM)

   The MROM is a 1024 word 72 bit memory which contains the prestored sequences of microinstructions (microprograms) required to fetch and execute program instructions, initiate and control input/output operations, and respond to external interrupts. A microprogram is executed by broadcasting the contents of one or more MROM locations to the ALU, MRU, FPU, SPM, MMU, and CU.

5. Iteration Counter (IC)

   The IC is used to control microinstruction sequencing. The contents of the IC may be interrogated and/or modified under microinstruction control. Microprogram transfer or microinstruction reiteration may be affected depending on the value contained by the IC at the time of interrogation.

4.2 System Organization and Data Flow

A more detailed SUMC block diagram is depicted in Fig. 4.3. The system is structured and interconnected to provide a versatile and orderly flow of data between the arithmetic section and memory. The

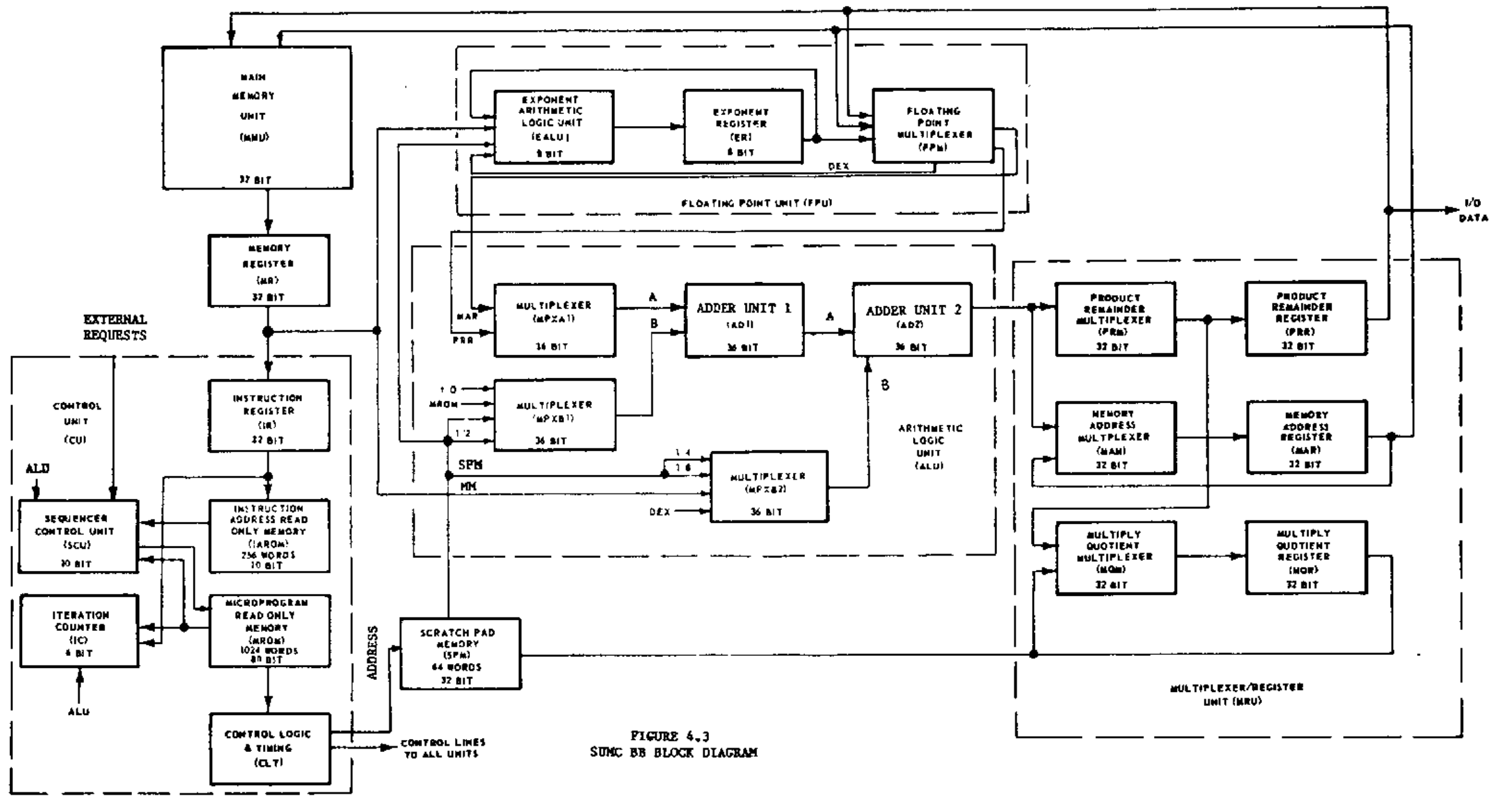

MAIN MEMORY UNIT (MMU) 32 BIT
MEMORY REGISTER (MR) 32 BIT
EXTERNAL REQUESTS
CONTROL UNIT (CU)
ALU
SEQUENCER CONTROL UNIT (SCU) 10 BIT
INSTRUCTION REGISTER (IR) 32 BIT
INSTRUCTION ADDRESS READ ONLY MEMORY (IAROM) 256 WORDS 10 BIT
ITERATION COUNTER (IC) 6 BIT
MICROPROGRAM READ ONLY MEMORY (MROM) 1024 WORDS 80 BIT
ALU
CONTROL LOGIC & TIMING (CLT)
CONTROL LINES TO ALL UNITS
ADDRESS
SCRATCH PAD MEMORY (SPM) 64 WORDS 32 BIT
EXPONENT ARITHMETIC LOGIC UNIT (EALU) 8 BIT
EXPONENT REGISTER (ER) 8 BIT
FLOATING POINT MULTIPLEXER (FPM)
DEX
FLOATING POINT UNIT (FPU)
MAR
PRR
MULTIPLEXER (MPXA1) 36 BIT
A
B
ADDER UNIT 1 (AD1) 36 BIT
A
ADDER UNIT 2 (AD2) 36 BIT
B
± 0
MROM
1/2
MULTIPLEXER (MPXB1) 36 BIT
SPM
MM
1/4
1/8
DEX
MULTIPLEXER (MPXB2) 36 BIT
ARITHMETIC LOGIC UNIT (ALU)
PRODUCT REMAINDER MULTIPLEXER (PRm) 32 BIT
PRODUCT REMAINDER REGISTER (PRR) 32 BIT
MEMORY ADDRESS MULTPLEXER (MAM) 32 BIT
MEMORY ADDRESS REGISTER (MAR) 32 BIT
MULTIPLY QUOTIENT MULTIPLEXER (MQm) 32 BIT
MULTIPLY QUOTIENT REGISTER (MQR) 32 BIT
MULTIPLEXER/REGISTER UNIT (MRU)
I/O DATA

FIGURE 4.3
SUMC BB BLOCK DIAGRAM

data path for information transferred between main memory and peripheral devices (I/O) is through the Arithmetic and the Multiplexer Register Units.

The hardware is organized into six functional sections Main Memory Unit (MMU), Scratch Pad Memory (SPM), Arithmetic Logic Unit (ALU), Multiplexer Register Unit (MRU), Floating Point Unit (FPU), and the Control Unit (CU).

4.3 Functional Description

Main Memory Unit (MMU)

The main memory addressing scheme provides both base and index addressing capability. Details of how instructions are read from main memory and executed are described later.

Scratch Pad Memory (SPM)

The SPM consists of sixty-four 32 bit semiconductor registers. The memory is divided into four groups of 16 registers each. Each group provides nondestructive read storage for 8 general purpose registers, 4 floating point registers, a program counter for instruction address storage, and other utility functions such as program and machine status, interrupt return, and temporary storage.

The two high order bits of SPM address (register group select) is provided by the interrupt source. The four low order address bits (register select) are provided by the instruction being executed or by the microcode.

Memory addressing and read/write operations are under microprogram control. This control also provides for partial read/write operations which allows floating point fractions to be accessed and operated upon independent of their characteristics.

Arithmetic Logic Unit (ALU)

The arithmetic section of the CPU consists of two 36 bit adder units and three 36 bit multiplexers.

Normal operations consist of performing arithmetic or logical operations on one, two, or three 32 bit operands selected by the input multiplexers. Operand selection and the operations to be performed by the arithmetic section are specified by the microcode stored in MROM. Multiplexer outputs are zero when no operand selection is specified.

Four extender bits (least significant bit positions) provide the capability for operating on 36 bit operands. These positions are used during special arithmetic operations for multiply, divide and square root algorithms, and when performing a long shift of the 64 bit word contained in the multiplexer register section PRR and MAR registers.

MPXA1 is a two input 36 bit multiplexer which selects the data for input A for the first adder unit (AD1). Only one of the two inputs is used in this configuration. This input consists of the 32 bit Floating Point Multiplexer (FPM) output and the four most significant bits (sign and positions 1 through 3) of the multiplexer register section MAR register.

MPXB1 is a three input 36 bit multiplexer which selects the data for input B of adder unit AD1. The three inputs consist of two 32 bit words and two partial word inputs. They are:

1. I/O is a 32 bit source input from external input/output hardware.
2. SPM is a 32 bit source input from scratch pad memory. Multiplexer control and internal connections allow

selection of full word or half word scratch pad memory. Half SPM is obtained by gating the SPM input shifted right one bit position with a sign fill in the vacated high order bit position.

3. MROM is a 10 bit source input (transfer field) from the microprogrammed read only memory. The inputs are connected to bit positions 22 through 31 of MPXB1.
4. Status inputs are various machine and program status bits connected to the MPXB1 positions not used by the MROM input. Selection of status information is independent of MROM selection.

MPXB2 is a three input 36 bit multiplexer which selects the data for input B of the second adder unit (AD2). The output of AD1 provides the information for input A of AD2. The three inputs to MPXB2 are:

1. SPM is a 32 bit data source from scratch pad memory. Input connections are made such that 1/4 or 1/8 scratch pad memory words may be selected. This is accomplished by gating the SPM input shifted right two and three bit positions respectively with a sign fill in the vacated high order positions. The least significant bits are shifted into the extended bit positions of the MPXB2 output.
2. MM is a 32 bit source input from main memory. Multiplexer control is implemented to allow either full or partial word selection. Positions 8 through 31 can be selected

for floating point fraction operations. Positions 20 through 31, the instruction displacement field, can also be selected for computing main memory address.

3. EALU is an 8 bit input from the exponent arithmetic logic unit. EALU inputs are connected to the most significant 8 positions of MPXB2. This input provides a data path for logically combining floating point characteristics and fractions in the arithmetic section.

Multiplexer Register Unit (MRU)

The multiplexer register section consists of three 32 bit multiplexer register pairs.

These multiplexer register pairs control the flow of data between the arithmetic section output and the main and scratch pad memory. They also provide temporary storage for intermediate operands obtained during a series of arithmetic or logical operations.

The multiplexer register pairs are shown in Fig. 4.3 and are identified as follows:

1. Product Remainder Multiplexer/Register (PRM/PRR);
2. Memory Address Multiplexer/Register (MAM/MAR);
3. Multiply Quotient Multiplexer/Register (MQM/MQR).

Product Remainder Multiplexer (PRM) is a four input 32 bit gating network which also has shifting capability. The arithmetic section output is connected to the four PRM inputs in a manner that allows the input to be shifted either right or left, or gated directly to the PRM output.

Depending on the microcode used, the input may be:

1. Gated direct to PRM output.
2. Shifted right 1 or 4 positions at the PRM output. The vacated high order bits are either replaced with the arithmetic sign or with zero depending on the specific microcode used. This corresponds to an arithmetic or logical right shift respectively.
3. Shifted left 1, 2, or 4 positions at the PRM output. The vacated low order bits are either replaced with the output of the four extended bit positions or with zero depending on the specific microcode used. This corresponds to a long or short left shift respectively.

As shown in Fig. 4.3, the PRM output is connected to the PRR and MQM input and is also available externally. Information is entered into the PRR only when specified by the microcode.

Memory Address Multiplexer (MAM) is a four input 32 bit gating network with shifting capabilities similar to that of the PRM. The output is entered into the MAR only when specified by the microcode.

Two data sources, the ALU and the MAR outputs, are connected to the four MAM inputs. Depending on the microcode used, the MAM output is:

1. The arithmetic section output (positions sign through 31).
2. The four extended positions of the arithmetic section output and the contents of MAR positions 4 through 31.

3. Same as for 2 right shifted 1 or 4 positions. The vacated high order positions are either replaced with the least significant bits of the arithmetic section output (positions 28 through 31) or zero depending on the specific microcode used. This corresponds to a long or short right shift in the MAM.
4. Same as for 2 left shifted 1, 2 or 4 positions. The vacated low order bits are replaced with zero.

Multiply Quotient Multiplexer (MQM) in a two input 32 bit gating network that has shifting capabilities similar to that of the PRM and MAM.

Inputs are provided by the PRM and MQR outputs. Depending on the microcode used the MQM output is:

1. The PRM output.
2. The MQR output right shifted four positions. The sign bit replaces the vacated high order positions.
3. The MQR output left shifted 1 or 2 positions. The vacated low order positions are replaced with zero.

Floating Point Unit (FPU)

The floating point section contains logic for computing exponents and for normalizing floating point fractions. These operations are performed by three functional units interconnected as shown in Fig. 4.3. These units are: Exponent Arithmetic Logic Unit (EALU), Exponent Register (ER), and Floating Point Multiplexer (FPM).

Exponent Arithmetic Logic Unit (EALU)

Arithmetic operations are performed on data selected by three multiplexers whose outputs are connected to two adder units. Data selection and arithmetic operations are specified by control memory microcode.

Exponent fields from both the scratch pad and main memory outputs are connected to the EALU input. Also connected to the input are the exponent register output and the derived exponent (DEX) from the floating point multiplexer. The derived exponent specifies the number of hexadecimal digits the floating point multiplexer input must be shifted (right or left) to produce a normalized fraction. Exponent arithmetic logic unit outputs are connected to the ER and to MPXB2 of the arithmetic section as shown in Fig. 4.3.

Exponent Register (ER)

The Exponent Register provides temporary storage for results of arithmetic operations performed by the EALU. Exponent register outputs are connected to the EALU input, and to the floating point multiplexer for controlling fraction normalization.

Floating Point Multiplexer (FPM)

The Floating Point Multiplexer, provides logic for normalizing floating point fractions. Normalization of long (64 bit) or short (32 bit) operands can be performed depending on the specific microcode used.

Outputs from the multiplexer register section PRR and MAR registers provide a 64 bit input to the FPM. These two registers provide temporary storage for the most and least significant words, respectively, of the fraction to be normalized. Floating point multiplexer logic generates a 5 bit derived exponent (DEX) which specifies the number of hexadecimal digits the contents of PRR and MAR must be shifted to obtain a normalized fraction. The DEX value is transferred to the ER (via the EALU) when specified by control memory microcode.

The exponent register contents provides control for the FPM normalizing logic. This logic, when enabled by the microcode, shifts the 64 bit input the number of positions specified by the value contained in the ER. The FPM output is a 32 bit word connected to input multiplexer MPXA1 of the arithmetic section (refer to Fig. 4.3). This output consists of either the least or most significant half of the normalized input, depending on the specific microcode used.

When not enabled, the FPM normalizing logic gates the specified portion of the 64 bit input to the output without normalizing. This provides a data path between the PRR or MAR output and the arithmetic section during fixed point arithmetic or logical operations.

Control Unit (CU)

The control unit contains six functional units that control SUMC operation and data flow. These units are: Instruction Register (IR), Instruction Address Read Only Memory (IAROM), Sequence Counter (SC), Iteration Counter (IC), Microprogram Read Only Memory (MROM), and Control Logic and Timing (CLT).

Instruction Register (IR)

The IR is a 32 bit register. Each computer instruction to be executed is first gated to the IR for temporary storage. The instruction operation code (8 bits) identifies the IAROM location containing the instruction microprogram starting address.

Instruction Address Read Only Memory (IAROM)

The IAROM is a 256 word 16 bit memory constructed from read only memory storage elements. Each IAROM word is associated with a specific computer instruction operation code, thus allowing an instruction repertoire of up to 256 instructions.

The 10 least significant bits of each IAROM word specify the starting address for the microprogram that must be executed to perform instruction operations. This information is gated into the sequence counter for control memory addressing. The remaining six bits identify instruction characteristics that allow functions to be implemented in hardware which simplify firmware design. These characteristics are: data addressing boundaries, register specification limitations, and a memory operand flag which indicates an operand from main memory is required for instruction execution.

Sequence Control Unit (SCU)

The Sequencer register is a 10 bit register. It functions as an address register for the Microprogram Read Only Memory (MROM). Sequencer register contents are modified under microinstruction control to provide microprogram sequencing. Modification is conditional or unconditional depending on the specific microcode used.

The value contained in the sequencer register can be incremented by 1, or initialized from either of three sources. These sources are: Arithmetic Logic Unit (ALU); Instruction Address Read Only Memory (IAROM); and the Microprogram Read Only Memory (MROM). Both external and internal status lines are monitored by sequence counter control logic to provide microprogram sequence control. External status lines are: interrupt request, input request, and output request. Internal status lines are: overflow, arithmetic section output sign, EALU sign, and iteration counter status.

Iteration Counter (IC)

The Iteration Counter is a 6 bit counter. It is used to implement microprogram loops for instructions requiring repeated operations such as shift, divide, multiply, and square root. Depending on the specific microcode used, the IC value can be decremented in either 1 or 4 bit steps. It can also be initialized from either of three sources: the PRM, the PRR, or the MROM.

Microprogram Read Only Memory (MROM)

The MROM is a 1024 word 72 bit memory. This memory contains the prestored control words (microinstructions) required to fetch and execute program instructions, initiate and control I/O operations, and respond to external requests.

Each 72 bit control word is divided into fields where the control bits in each field specify the operations to be performed by the associated SUMC sections. Table 4.1 shows the control word format.

4.4 Computer Control

The computer can be operated in either of two modes (normal or manual) that can be selected from the computer operators panel.

In the normal mode of operation all I/O requests, input data, and computer control signals are generated by external sources. In the manual mode of operation, all external inputs are disabled and replaced by similar functions which can be generated manually from the computer

TABLE 4.1

SUMC Control Memory Word Format

BIT	FUNCTION
1-10	ROM Transfer Address
11-12	Condition Selection
13-16	Sequencer and Iteration Control
17-18,59	FPM Control
19-24	EALU Control
25-26	Main Memory Control
27-30	Load Register
31-32	MQM Control
34-36	MAM Control
38-41	PRM Control
43-44	ALU Control
45-47	Adder 2 Control
48-50	Adder 1 Control
51-55	MPXB2 Control
56-58,60	MPXB1 Control
61	MPAX1 Control
62-63	SPM Access
64	SPM Read/Write
65-66	SPM Address Modifier
67-72	SPM Address
37,42,58,60	Special Control

operators panel. The manual operating mode provides for manual loading or modification of programs and for microprogram verification.

The following is a brief description of the signals which control the computer and I/O operations:

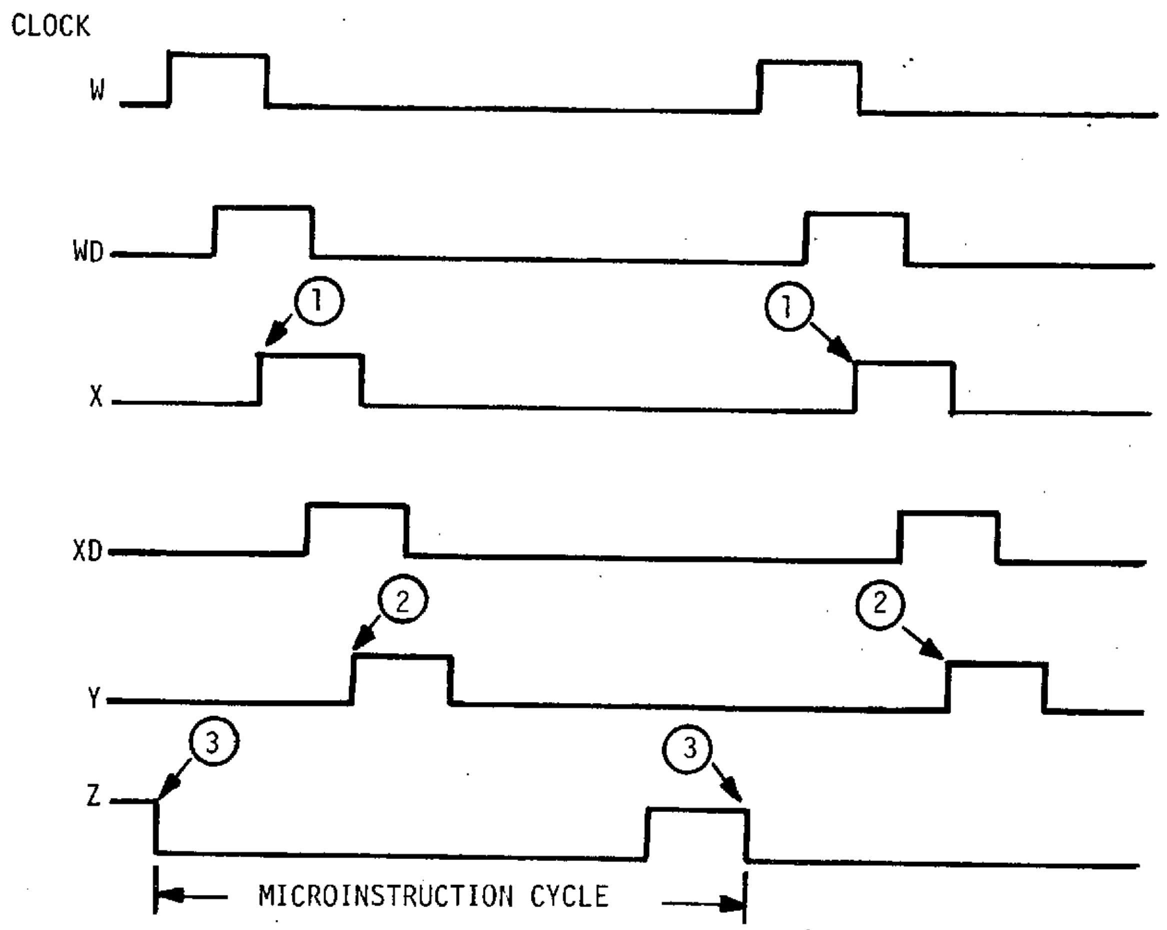
1. Computer stop - Computer stop disables the timing logic during fetch of the instruction following the computer stop request. No SUMC operations are performed until the computer stop request is removed.
2. Computer start - Computer start removes the computer stop request and enables the timing logic
3. Program halt - Program halt disables the IAROM output (IAROM output is forced to a fixed MROM address) causing transfer to the program halt microprogram. This microprogram decrements the program counter contents (instruction address) and returns to the fetch microprogram. All I/O requests are processed in the program halt condition. Normal instruction execution continues when the program halt condition is removed.
4. Program start - Program start removes the program halt condition and allows normal instruction execution to continue.
5. Data output request - Data output request is detected during the fetch microprogram and causes transfer to the I/O microprogram. Data is accessed from the memory location specified by the address on the I/O input bus.

6. Data input request - Data input request is detected during the fetch microprogram and causes transfer to the I/O microprogram. The data to be stored is the second of two words input from the I/O bus. The first word specifies the memory location where the data is to be stored.
7. Interrupt request - Interrupt request is detected during the fetch microprogram. The I/O microprogram performs the operations required for the particular interrupt identified by the data word on the I/O input bus.

4.5 Timing

The timing logic consists of an oscillator and logic for generating six signals depicted in Fig. 4.4. Three of the signals (X, Y, and Z) are used for control of basic operations while the remaining signals provide for special control functions and timing variations. The basic microinstruction cycle consists of five operations which are described as follows:

1. Selection of the control word to be broadcast for operational control. This occurs at clock time Z when the sequence counter contents are updated by the previous microinstruction. The updated MROM address selects the new control word that is broadcast for operational control.
2. Start memory read/write operation if specified by microcode. Scratch pad and main memory write operations are initiated at clock time X. Address and data to be stored must be loaded into the appropriate registers during a



1 *(Start main memory read/write.)
*(Start SPM write.)

2 Update SPM address.
*(Set instruction register.)

3 *(Set MQR, PRR, MAR, ER)
Update sequence and iteration counters.

*These operations are performed only when specified by the microcode.

Figure 4.4. Basic SUMC Timing Signals

previous microinstruction cycle. The SPM address register is not updated until the SPM write operation is completed. This allows an operand from SPM to be operated upon and returned during the same microinstruction cycle. The SPM address is updated at clock time Y.

3. Perform arithmetic or logical operations specified by control word. These operations occur during the complete microinstruction cycle. It should be noted however that scratch pad memory operands are not available until clock time Y where the address is updated as specified by the current control word.

4. Gate results of arithmetic or logical operations into temporary storage registers of multiplexer register section for later use or for transfer to main or scratch pad memories. This occurs at clock time Z.

5. Update the sequence and iteration counter contents, as specified by the microcode and status signals, to maintain the desired microprogram control. This occurs at clock time Z.

CHAPTER V

SUMC BRANCHING ARCHITECTURE

5.1 Sequence Control Unit (SCU)

The purpose of the SUMC Sequence Control Unit (SCU) is to provide the address of the next microinstruction to be executed. To achieve this purpose, selected data is brought into the sequencer register. Data is selected from one of three sources as outlined below:

IAROM - The 10 least significant bits of the IAROM are loaded into the sequencer. This value is the microprogram starting address.

MROM - The 10 least significant bits of an MROM control word are loaded into the sequencer. This value is the address of the microinstruction to be executed next if a condition checked for is met.

PRM - The 10 least significant bits of the PRM multiplexer are loaded into the sequencer. This value is the address of the next microinstruction to be executed.

5.2 The SUMC Conditional Checks

The SUMC has a number of conditional checks which are used to decide which microinstruction is to be executed next [2]. These checks can be grouped into two categories: The Decision checks and the Iterative checks. The Decision checks can be grouped into three categories: The I/O checks, JI; the FALSE checks, JF; and the TRUE checks, JT.

The I/O Checks (JI).

For each of these checks, if the I/O condition checked for is true; the sequencer register is loaded with the MROM transfer field; if false, the contents of the sequencer are incremented. The I/O checks are defined as follows:

JINT(N)-Jump on interrupt to N.

JIDOT(N)-Jump on interrupt or data out request to N.

JIO(N)-Jump on interrupt or I/O request to N.

The FALSE Checks (JF)

If the condition checked for is false, the sequencer is loaded with the MROM transfer field; if true the sequencer contents are incremented. The FALSE checks are defined as follows:

JNOF(N)-Jump on no ALU overflow to N.

JNXOF(N)-Jump on no EALU overflow to N.

JNZ(N)-Jump if FPM not zero.

JNRSX(N)-Jump on no register specification to N.

The TRUE Checks (JT)

For the TRUE checks, if the condition checked for is true, the sequencer is loaded with the MROM transfer field; if false, the sequencer contents are incremented. The TRUE checks are defined as follows:

JN(N)-Jump if ALU negative.

JNX(N)-Jump if EALU negative.

JIA14-Jump if IAROM control bit 14=1.

JIA13(N)-Jump if IAROM control bit 13=1.

The Iterative Checks (JC)

For the Iterative checks, if the count condition checked for is satisfied, the sequencer is loaded with the MROM transfer field; otherwise the count is decremented and the same microinstruction is executed (for JCH); or the next microinstruction is executed (for JCA). The Iterative checks are defined as follows:

JCZH(N)	-	Jump to N if iteration counter contents is zero. Otherwise, decrement the iteration counter contents by 1 and repeat microinstruction.
JCL4H(N)	-	Jump to N if iteration counter contents is less than 4. Otherwise decrement the iteration counter contents by 4 and repeat microinstruction.
JCZA(N)	-	Jump to N if iteration counter contents is zero. Otherwise, decrement the iteration counter contents by 1 and execute next microinstruction.
JCL4A(N)	-	Jump to N if iteration counter contents less than 4. Otherwise, decrement the iteration counter contents by 4 and execute next microinstruction.

The SUMC successor commands are summarized in Table 5.1. The unconditional branching commands are STEP and J (Jump); the decision commands are JI (Jump on I/O Condition), JT (Jump on TRUE), JF (Jump on FALSE); the iterative commands are represented by JCH (Jump on count condition true or hold), and JCA (Jump on count condition true of advance).

TABLE 5.1

Successor Commands for SUMC BB

COMMAND	NEXT INSTRUCTION ADDRESS	INSTRUCTION ADDRESS
STEP	(SEQ) + 1	-------------------
J	$(\text{MROM})_X \rightarrow (\text{SEQ})$	-------------------
JI	$_I< (\text{MROM})_X \rightarrow (\text{SEQ})\ V\ (\text{SEQ})+1 >$	-------------------
JT	$_T< (\text{MROM})_X \rightarrow (\text{SEQ})\ V\ (\text{SEQ})+1 >$	-------------------
JF	$_F< (\text{MROM})_X \rightarrow (\text{SEQ})\ V\ (\text{SEQ})+1 >$	-------------------
JCH	$_C< (\text{MROM})_X \rightarrow (\text{SEQ})\ V\ (\text{SEQ}) >$	-------------------
JCA	$_C< (\text{MROM})_X \rightarrow (\text{SEQ})\ V\ (\text{SEQ})+1 >$	-------------------

5.3 SUMC Structured Microprogram Primitives

In this section, the structured microprogram primitives are implemented for the SUMC BB. A microprogram, a macroprogram, and their corresponding flowcharts are given for each primitive. The decision primitive is implemented for both True and False SUMC checks.

Primitive: f then g

Microprogram

```
f, STEP
g
```

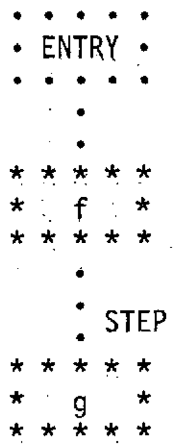


Figure 5.1 SUMC Concatenation Microflowchart

Primitive: f then g

Macroprogram

```
             f_1, STEP
             f_2, STEP
             :
             f_n, J(G1)
             ------
    G_1:     g
```

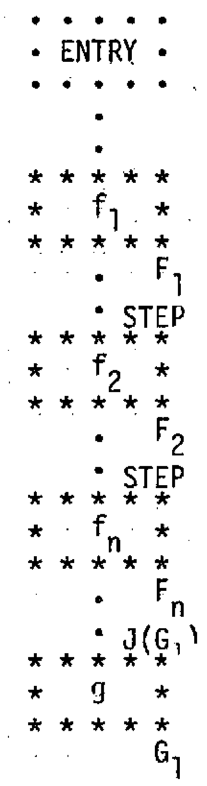
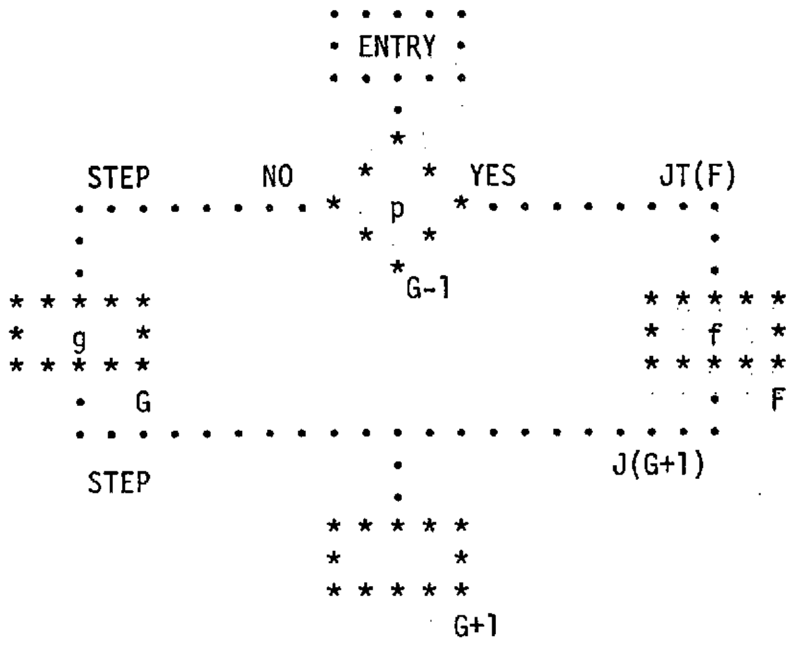


Figure 5.2. SUMC Concatenation Macroflowchart

```
Primitive:      If p then f, else g

Microprogram for TRUE Checks

                If p then JT(F), else STEP

                g, STEP

                - - - - - -

     F:         f, J(G+1)

Microprogram for FALSE Checks

                If p then STEP, else JF(G)

                f, STEP

                - - - - - -

     G:         g, J(F+1)
```

For TRUE Checks



For FALSE Checks

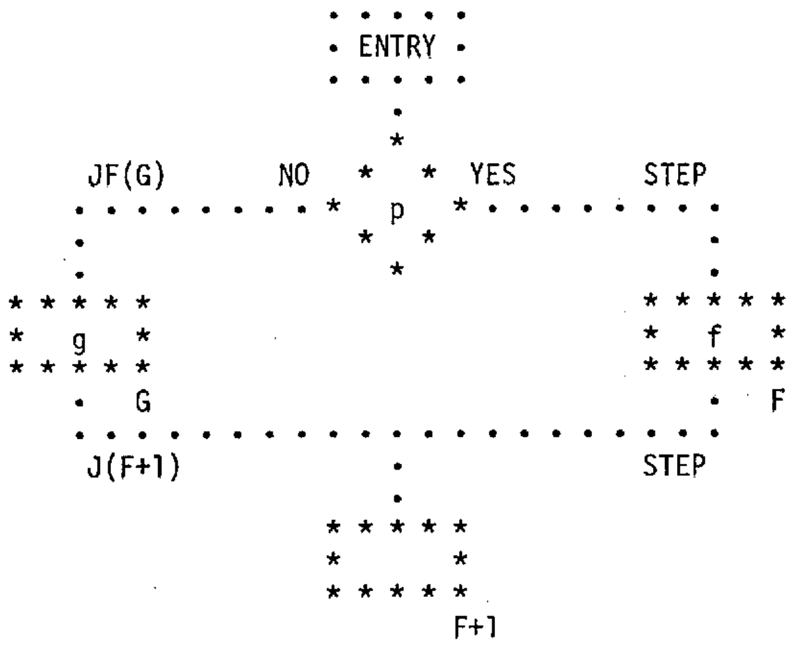


Figure 5.3. SUMC Decision Microflowcharts

Primitive: If p then f, else g

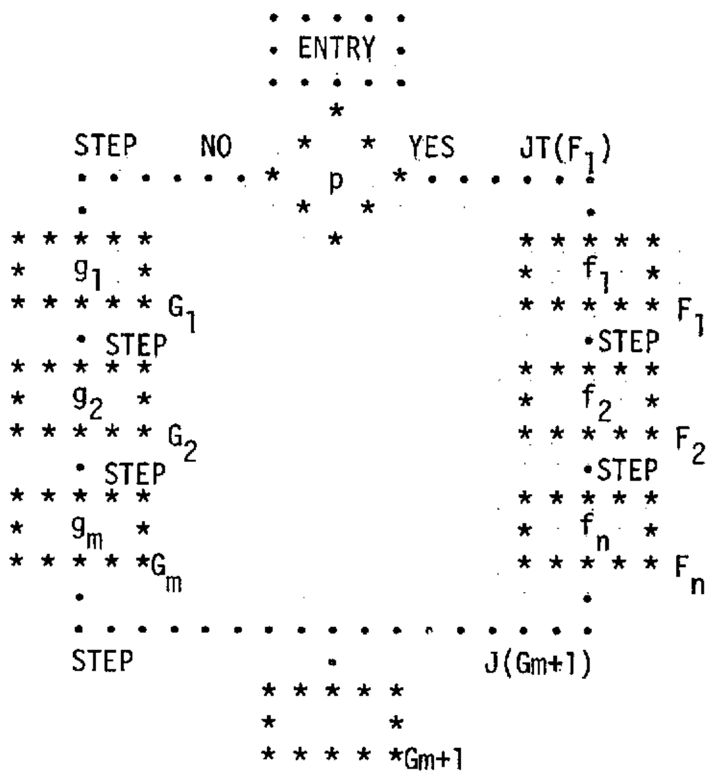
Macroprogram for TRUE Checks

```
              If p then JT(F_1), else STEP
      G_1:    g_1, STEP
              g_2, STEP
              .
              .
              .
              g_n, STEP
              - - - - - -
      F_1:    f_1, STEP
              f_2, STEP
              .
              .
              .
              f_n, J(Gn+1)
```

Macroprogram for FALSE Checks

```
              If p then STEP, else JF(G_1)
      F_1:    f_1, STEP
              f_2, STEP
              .
              .
              .
              f_n, STEP
              - - - - - -
      G_1:    g_1, STEP
              g_2, STEP
              .
              .
              .
              g_m, J(F_n+1)
```

For TRUE Checks



For FALSE Checks

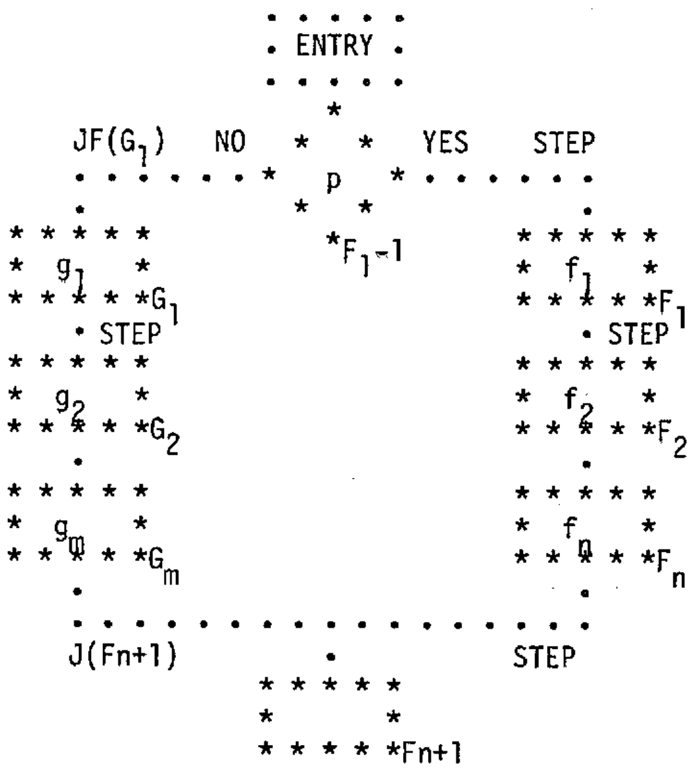


Figure 5.4. SUMC Decision Macroflowcharts

Primitive: While $\tilde{p}$, do f

While [(IC) $\neq$ 0], do f

Microprogram

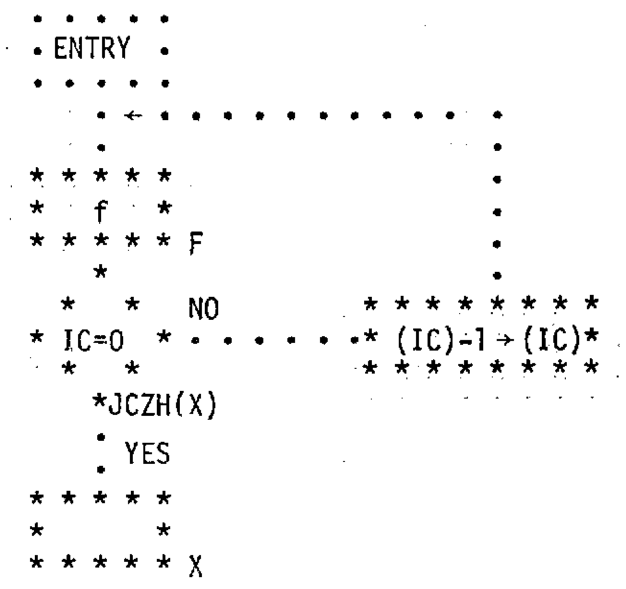
If (not p) then f, else JCZH (X)

While [IC) $\nless$ 4], do f

Microprogram

If (not p) f, else JCL4H (X)

While [(IC) ≠ 0], do f



While [(IC) ≮ 4], do f

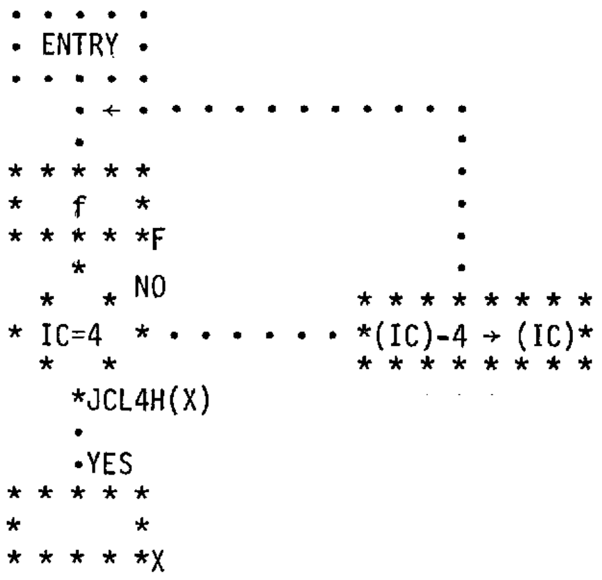


Figure 5.5. SUMC Iteration Microflowchart

Primitive: While $\tilde{p}$, do f

While [(IC) $\neq$ 0], do f

Macroprogram

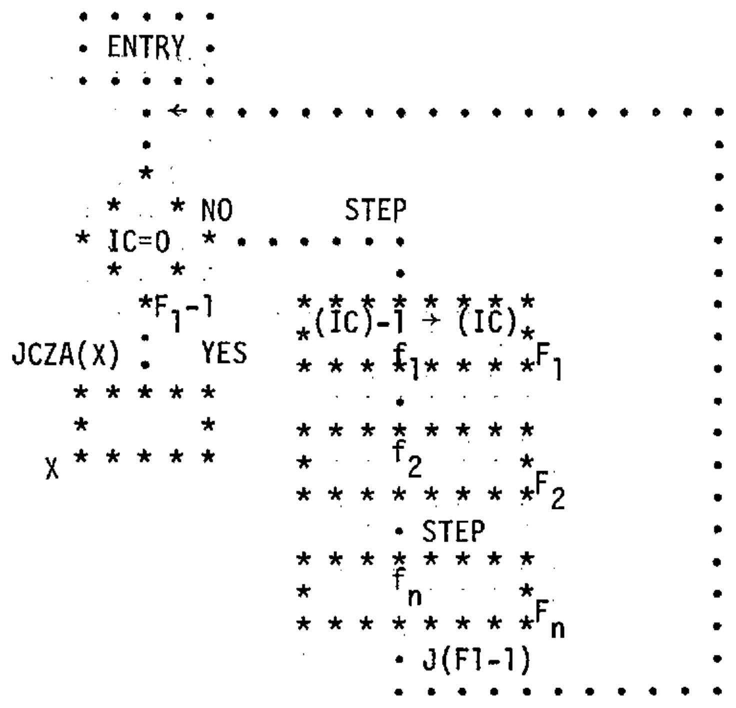
```
F_1-1:    If (not p) then STEP, else JCZA(X)
F_1:          f_1, STEP
              f_2, STEP
              :
              f_n, J(F_1-1)
              - - - - - -
X:        CONTINUE
```

White [(IC) $\nless$ 4], do f

Macroprogram

```
F_1-1:    If (not p) STEP, else JCL4A(X)
F_1:          f_1, STEP
              f_1, STEP
              :
              f_n, J(F_1-1)
              - - - - - -
X:        CONTINUE
```

While [(IC $\neq$ 0)], do f



While [(IC) $\nless$ 4], do f

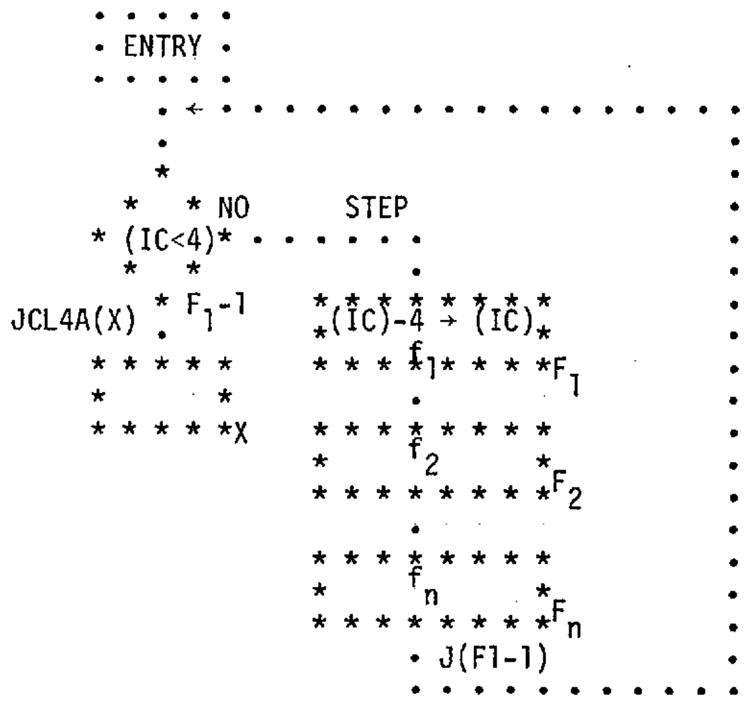


Figure 5.6. SUMC Iteration Macroflowchart

It is possible to implement other iterative loops by using the SUMC hardware in the following manner. A positive iteration (refer to Fig. 5.7) count, N, is initially loaded into a temporary register, T. Subsequent microinstructions execute the desired process, f. Then, the count is decremented by 1 and checked for zero. If the count is not zero, the execution of f is repeated; otherwise, control is transferred to some other microprogram through a JNZ(X) microorder. This technique is used to implement N iterations in the loop. The disadvantage of this technique is that the PRR must be loaded at least one microinstruction prior to making the check.

Another way of implementing an iterative loop is as follows: A negative count, -N, is initially loaded into T (Ref. Fig. 5.8) subsequent microinstructions execute the desired process f. Then, the count is incremented by 1 and checked for negative. If the count is negative, the execution of f is repeated. If the count is not negative, the sequencer is advanced. This technique allows /N/+1 iterations in the loop. This technique is better because it frees the PRR.

A third type of iterative loop is used in the implementation of the SI format instructions [6]. Formats are discussed in Section 6.2. This technique uses a JIA microorder in conjunction with a flag (refer to Fig. 5.9). During fetch, the flag is set to zero. Some time later in fetch, the sequencer addresses the microprogram starting location as a result of JIA microorder. Thus microprogram control is transferred to this location. The first step of the microprogram checks the flag. Since it is zero, the sequencer is advanced. Subsequent microinstructions in the microprogram perform the desired process, f. Then, the flag is changed to a one. To form the loop, a JIA microorder is called. Since the IAROM is still pointing to the microprogram starting location, control is once again transferred to this location. Since the tag is set to a one, the sequencer is loaded with the MROM transfer address. Thus, control is transferred to some other microprogram. This technique implements a "one pass" iterative loop.

5.4 Subroutines

The SUMC computer does not have a hardware CALL. However it is still possible to handle subroutines (and thus implement the CALL) by using one of the scratchpad temporary registers to store the return address.

In practice, the process is as follows: at some time during the microprogram (at a cost of one microinstruction) the desired return address is stored in a temporary register (usually T5), then at some subsequent microinstruction the subroutine is entered through an unconditional jump. The desired subroutine is executed. The last step of the subroutine consists of gating the temporary register contents to the sequencer, thus transferring to the previously selected microinstruction (return address). This technique, however, does not allow for subroutine nesting. The SS format flowcharts in Chapter VI illustrate the technique.

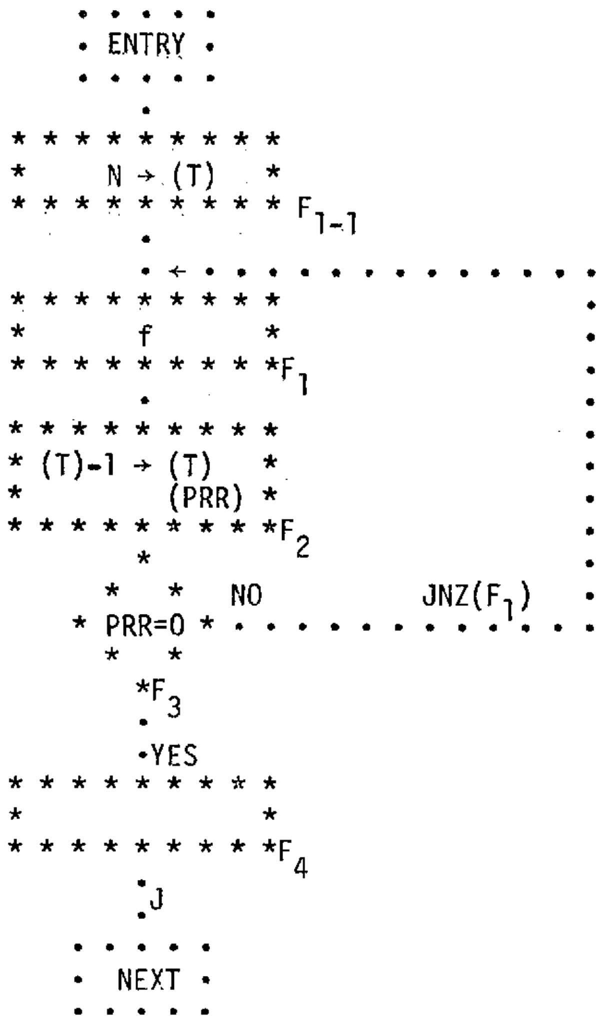


Figure 5.7. SUMC N-Iterative Loop

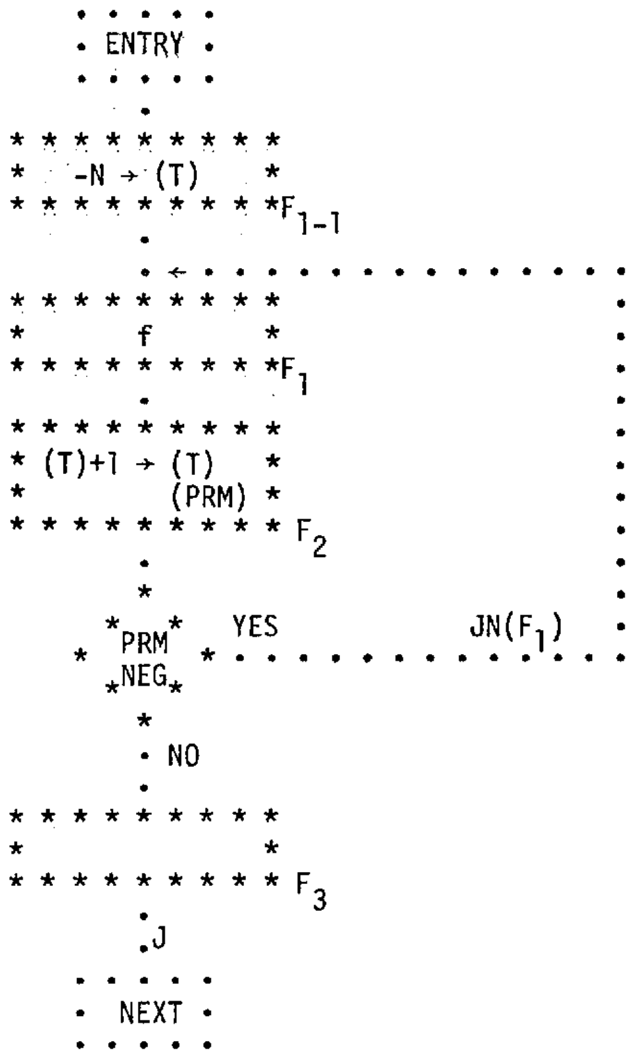


Figure 5.8. SUMC N+1-Iterative LOOP

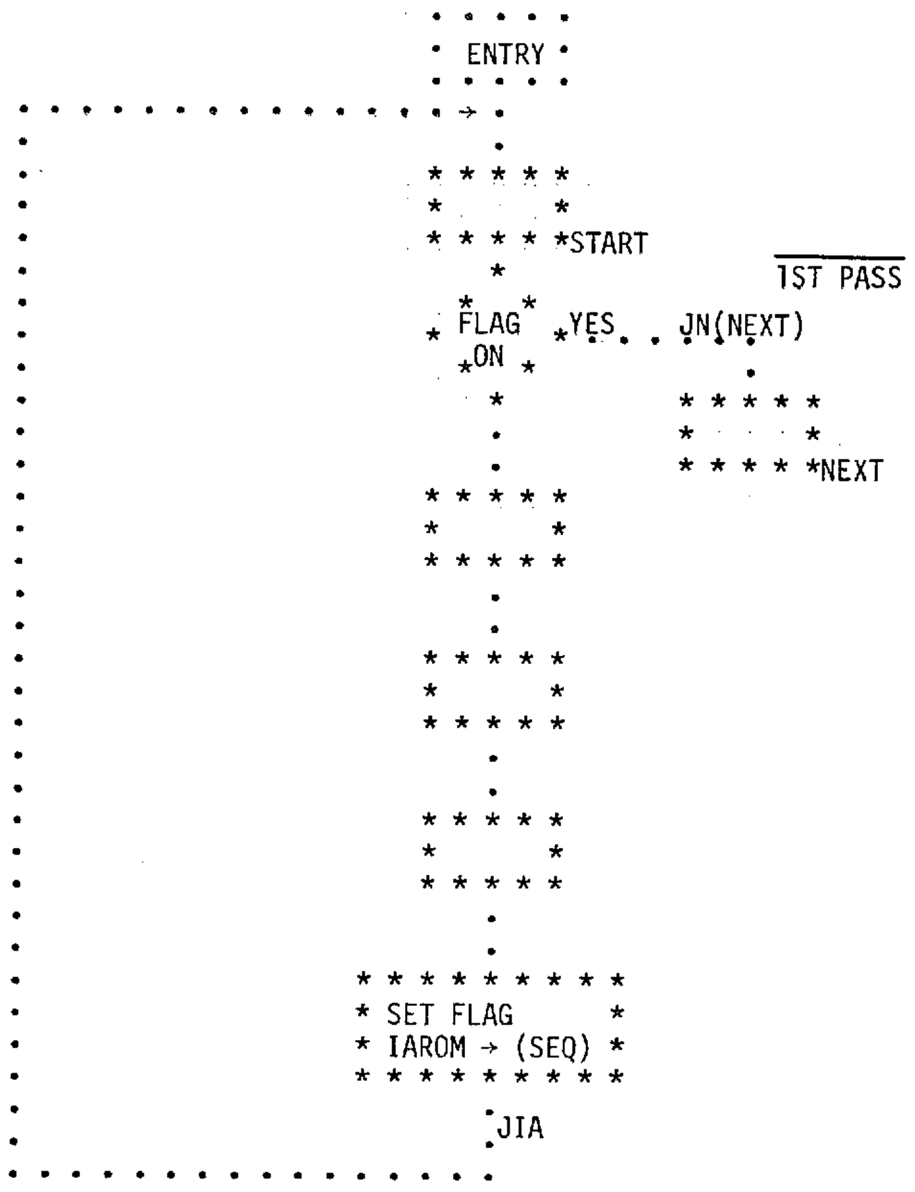


Figure 5.9. SUMC DO ONCE LOOP

CHAPTER VI

SUMC BB MICROCODE IMPLEMENTATION

6.1 Emulation

One of the main advantages of a microprogrammable computer is its emulation ability. Emulation is defined as the imitation of one system by another such that the imitating system accepts the same data and programs and achieves the same result as the initial system. There may be a difference in execution time to achieve the same results.

6.2 IBM System/360 Instruction Formats

The length of an instruction format can be one, two, or three halfwords. 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.

The five basic instruction formats [6] are denoted by the format codes RR, RX, RS, SI, and SS. These codes express 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; and SS, a storage-to-storage operation.

6.3 Structured Microprograms

This section contains a subset of SUMC structured microprograms in flowchart form which emulate the IBM System/360 instruction set [6].

Decimal and floating point instructions are not implemented. The author had intended to code the microprograms represented by the flowcharts, and run diagnostics on them. However, since the available hardware is undergoing expansion modifications it was not wise to do so. A corresponding nonstructured microprogram set (contain floating point instructions), previously written by the author [21] has been thoroughly verified by the use of diagnostics. The author exercised great care not to alter the emulating process; thus, the implemented structured microprograms should be valid or sufficiently close to valid to allow realistic conclusions.

The structured microprogram flowcharts shown in Figs. 6.1 through 6.68 are divided into the following groups: RR format, RX format, RS format, SI format, SS format, and housekeeping. Housekeeping microprograms fetch instructions and operands to be executed, and store program status information.

In the following flowcharts, instruction mnemonics correspond to those used for IBM System/360 instructions. A microprogram name which is formed by adding one or more letters to an instruction mnemonic refers to a continuation microprogram. A continuation microprogram, unless listed separately in the List of Figures, appears in the same figure as the instruction mnemonic, the microprogram is listed in the List of Figures. For example, SIF1 is listed as Fig. 6.27. Instruction opcodes are shown in hexadecimal form in the upper right hand corner of the instruction entry block.

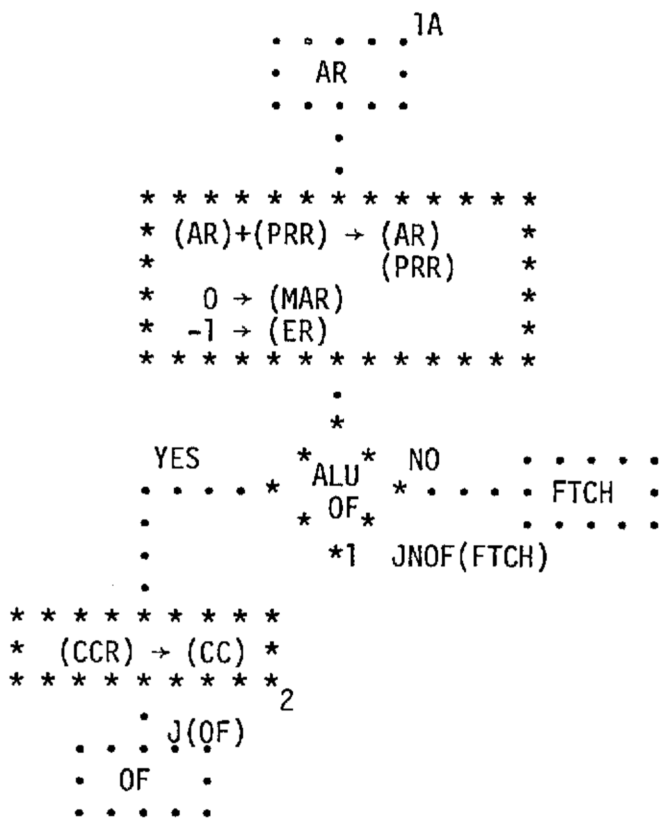


Figure 6.1. Add Register (AR) Microprogram

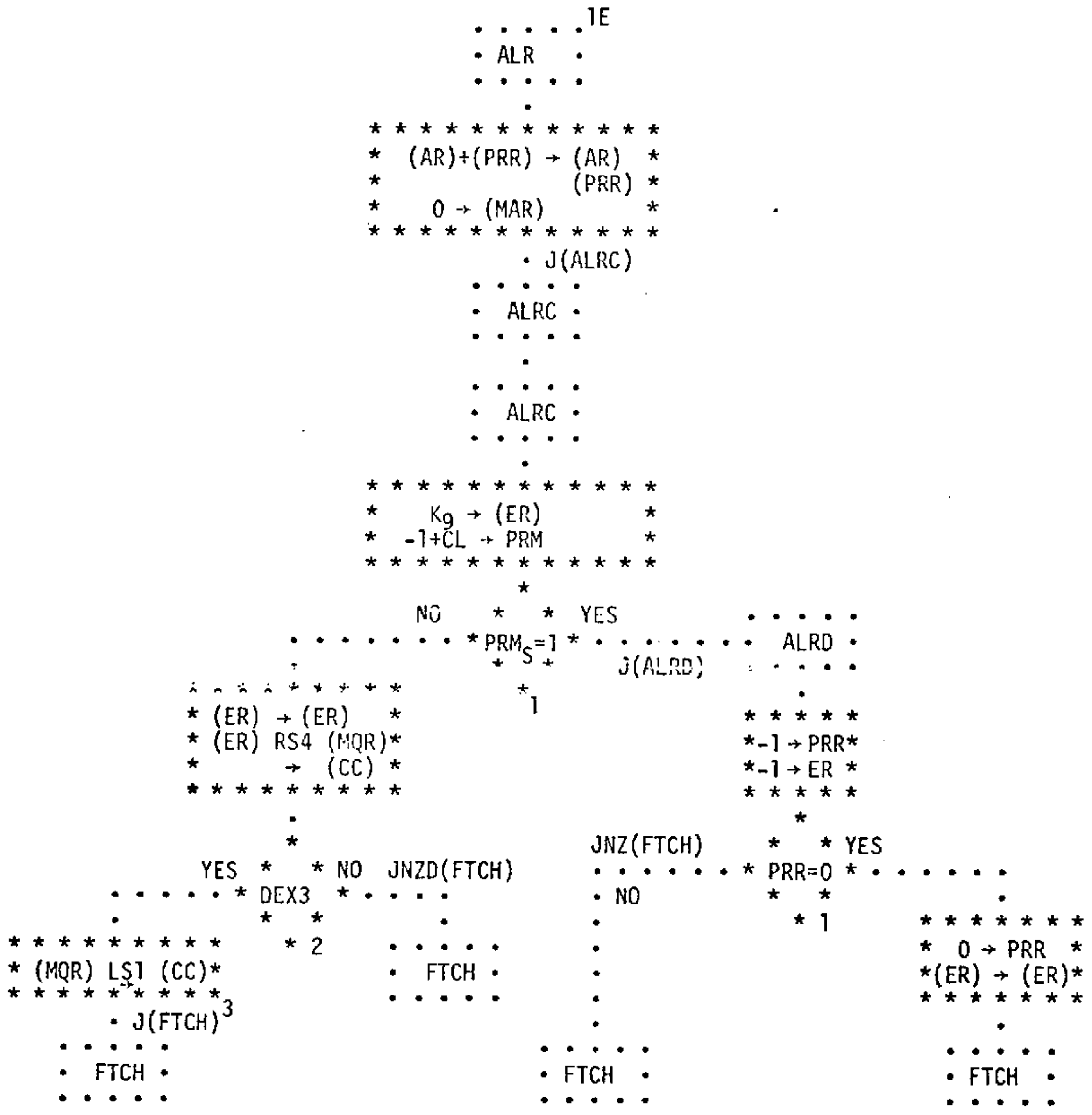


Figure 6.2. Add Logical Register (ALR) Microprogram

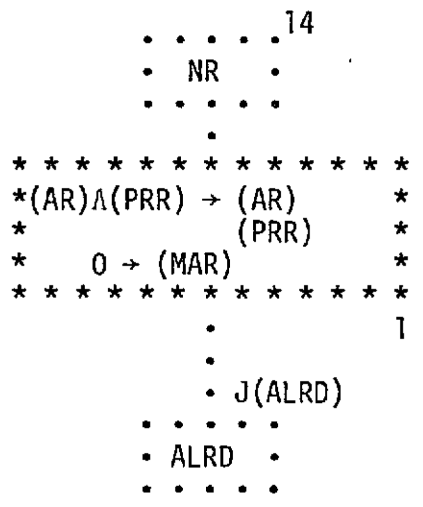


Figure 6.3. And Register (NR) Microprogram

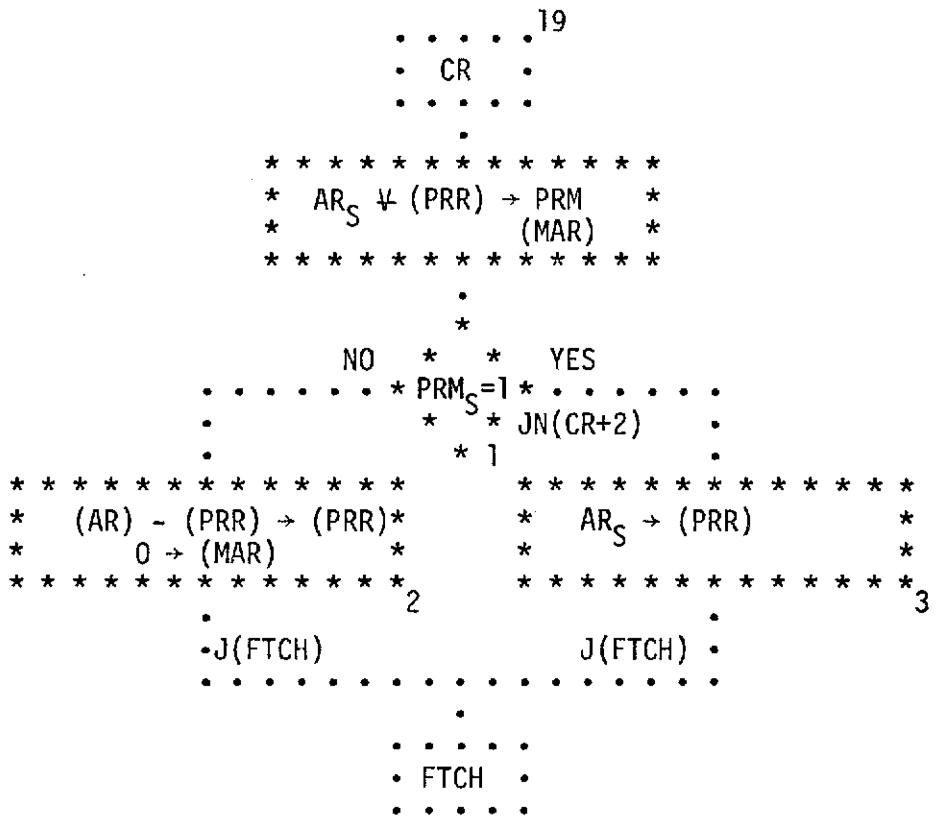


Figure 6.4. Compare Register (CR) Microprogram

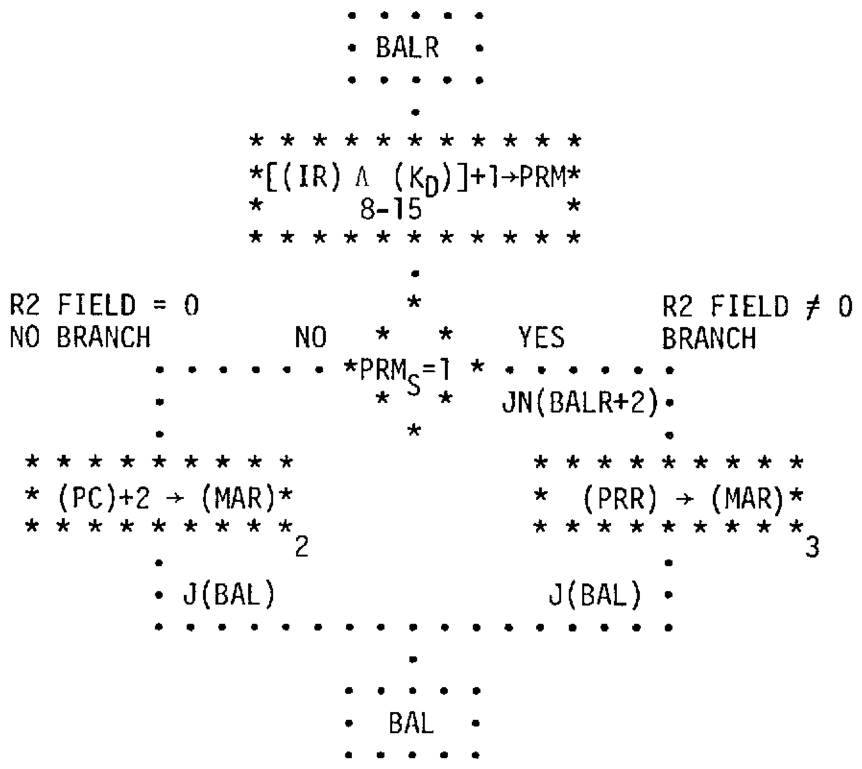


Figure 6.5. Branch and Link Register (BALR) Microprogram

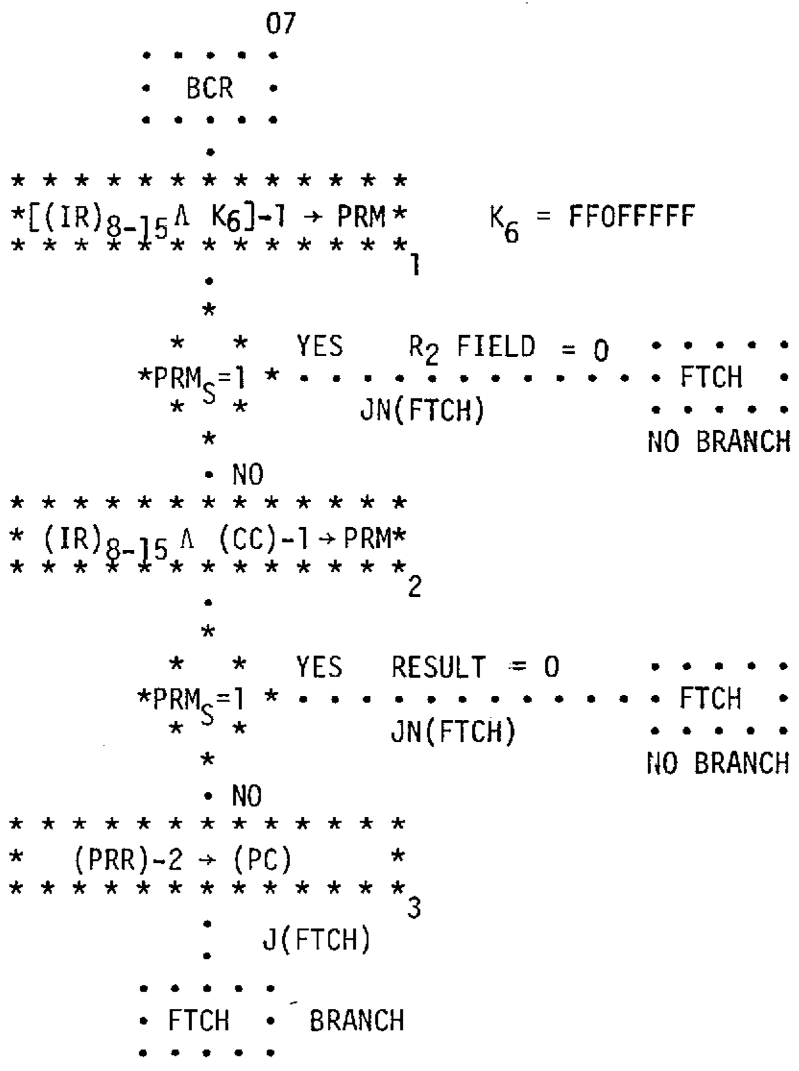


Figure 6.6. Branch on Condition Register (BCR) Microprogram

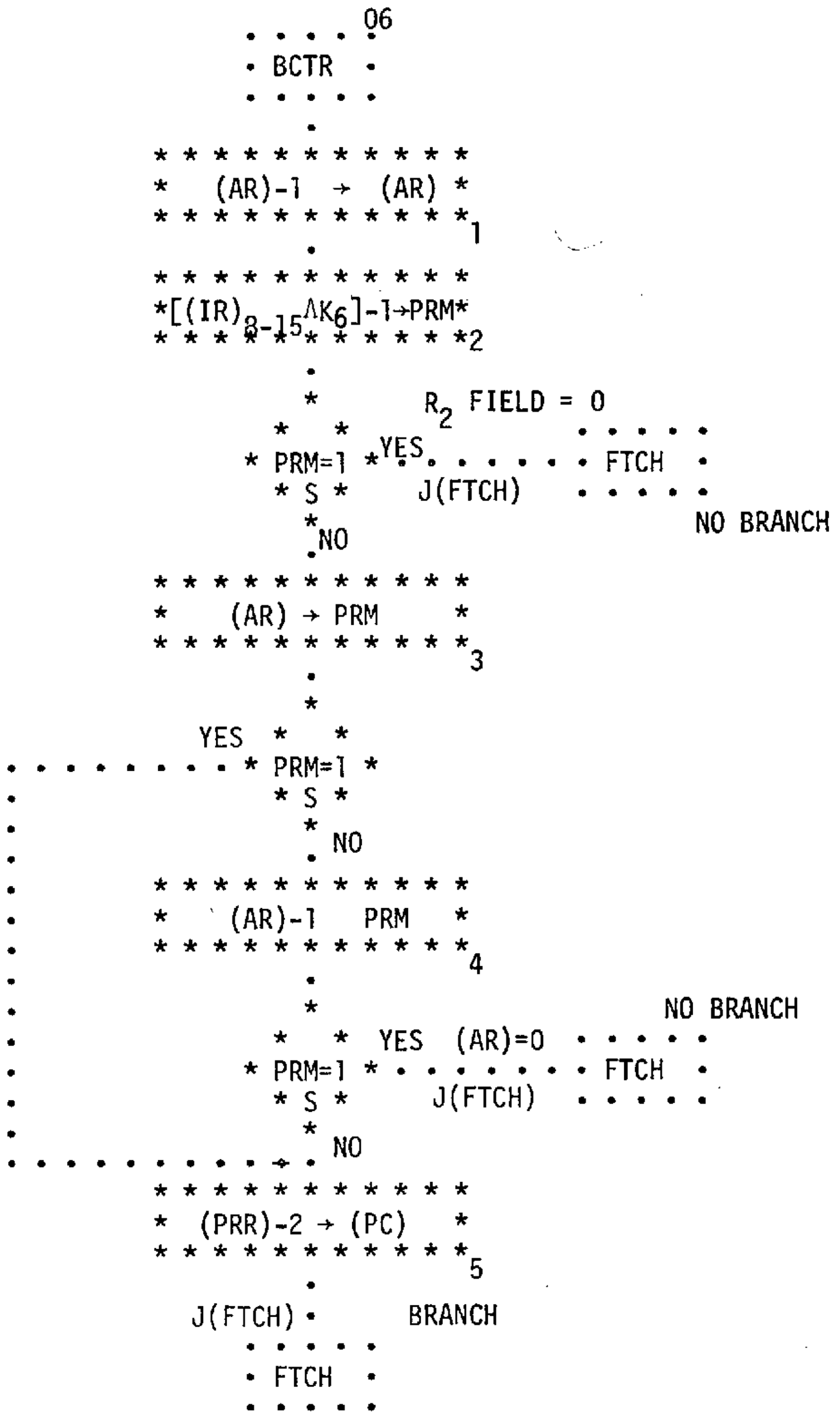


Figure 6.7. Branch on Count Register (BCTR) Microprogram

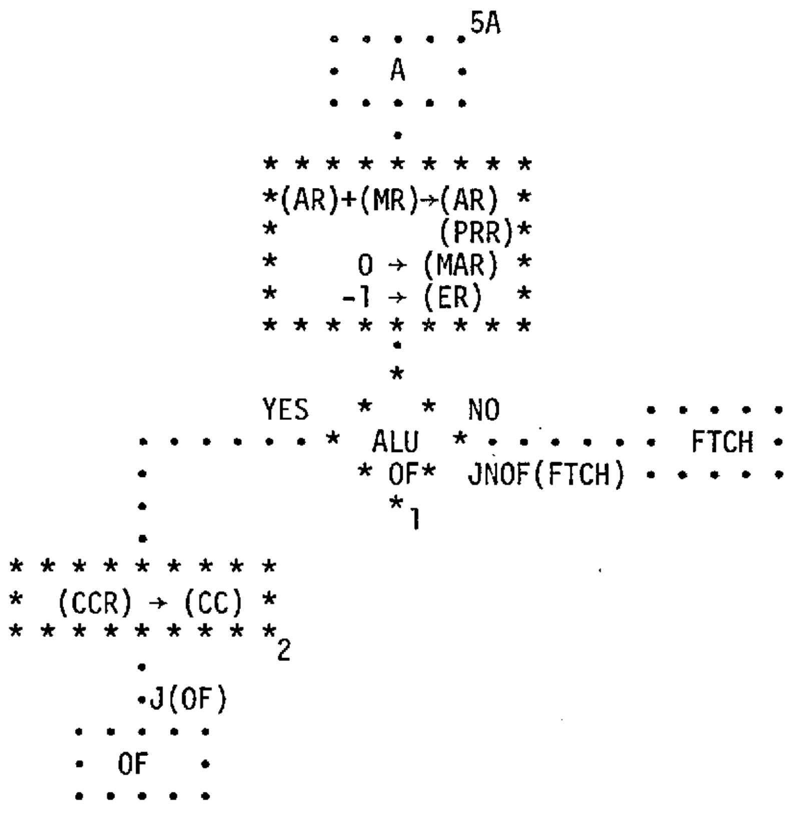


Figure 6.8. Add (A) Microprogram

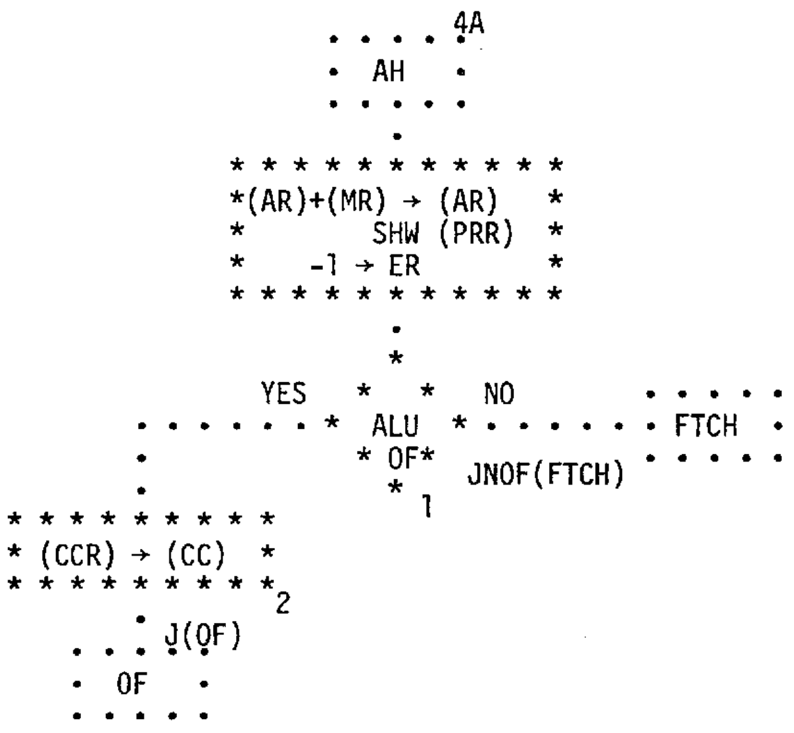


Figure 6.9. Add Halfword (AH) Microprograms

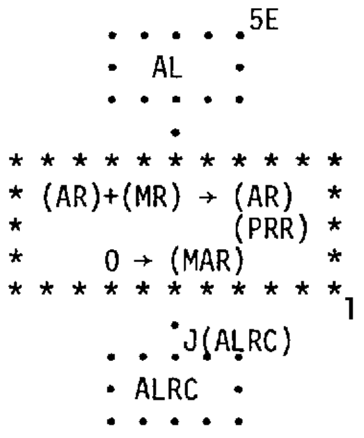


Figure 6.10. Add Logical (AL) Microprogram

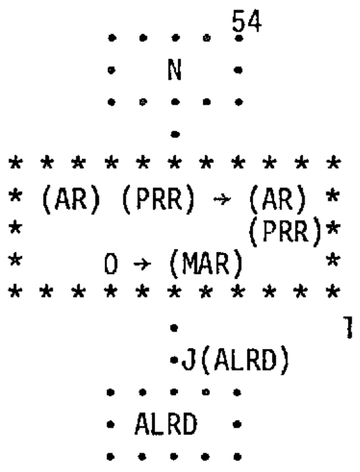


Figure 6.11. And (N) Microprogram

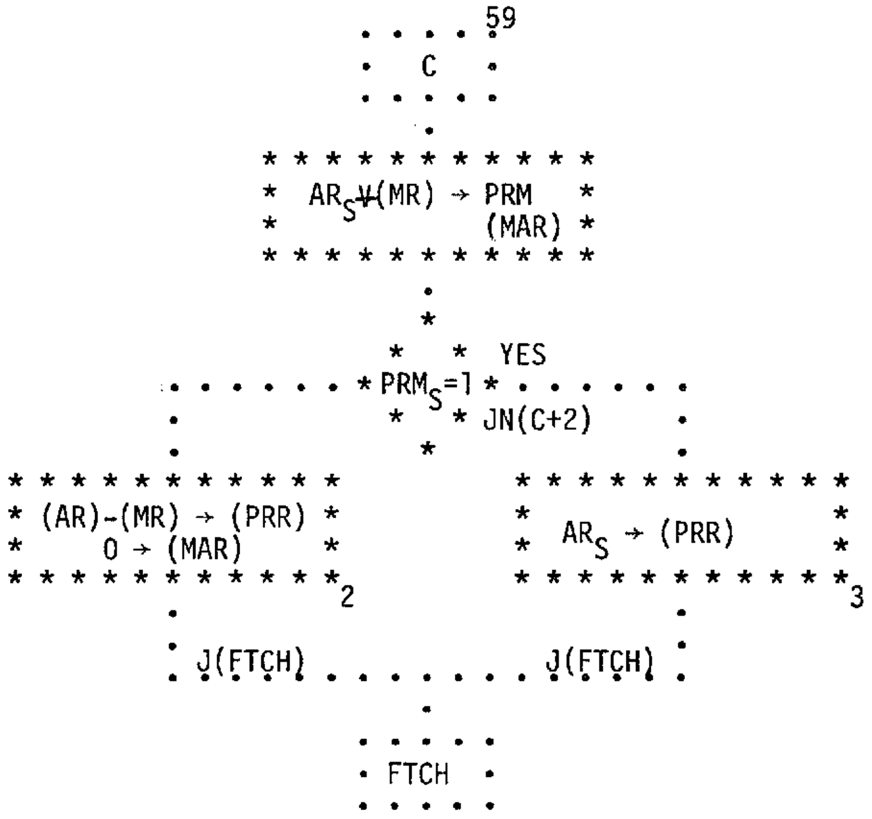


Figure 6.12. Compare (C) Microprogram

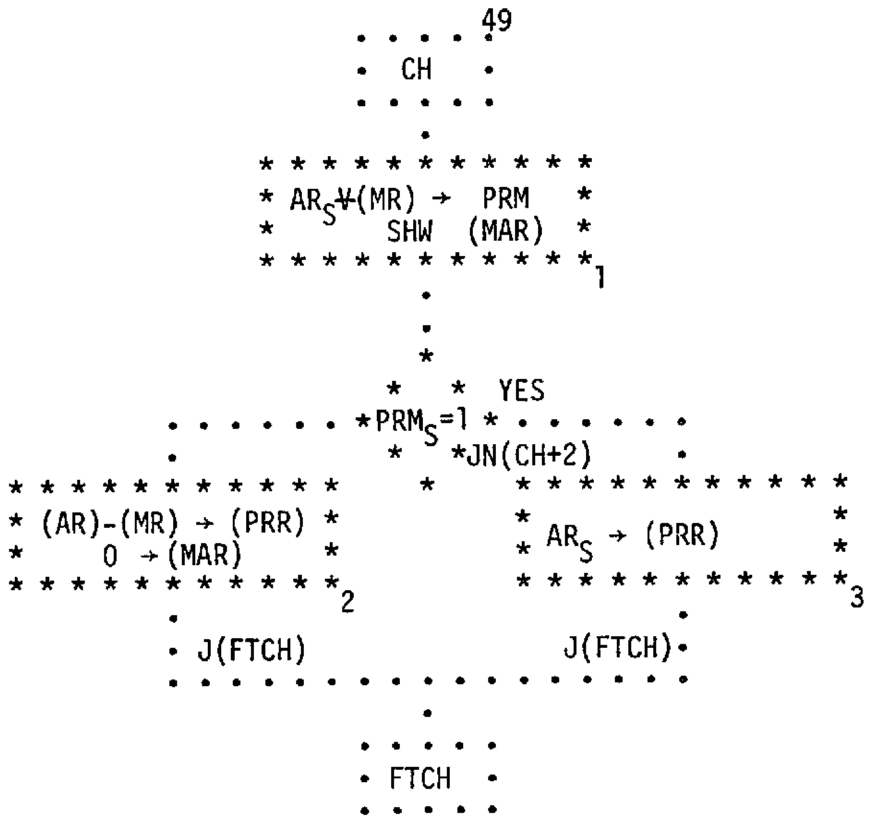


Figure 6.13. Compare Halfword (CH) Microprogram

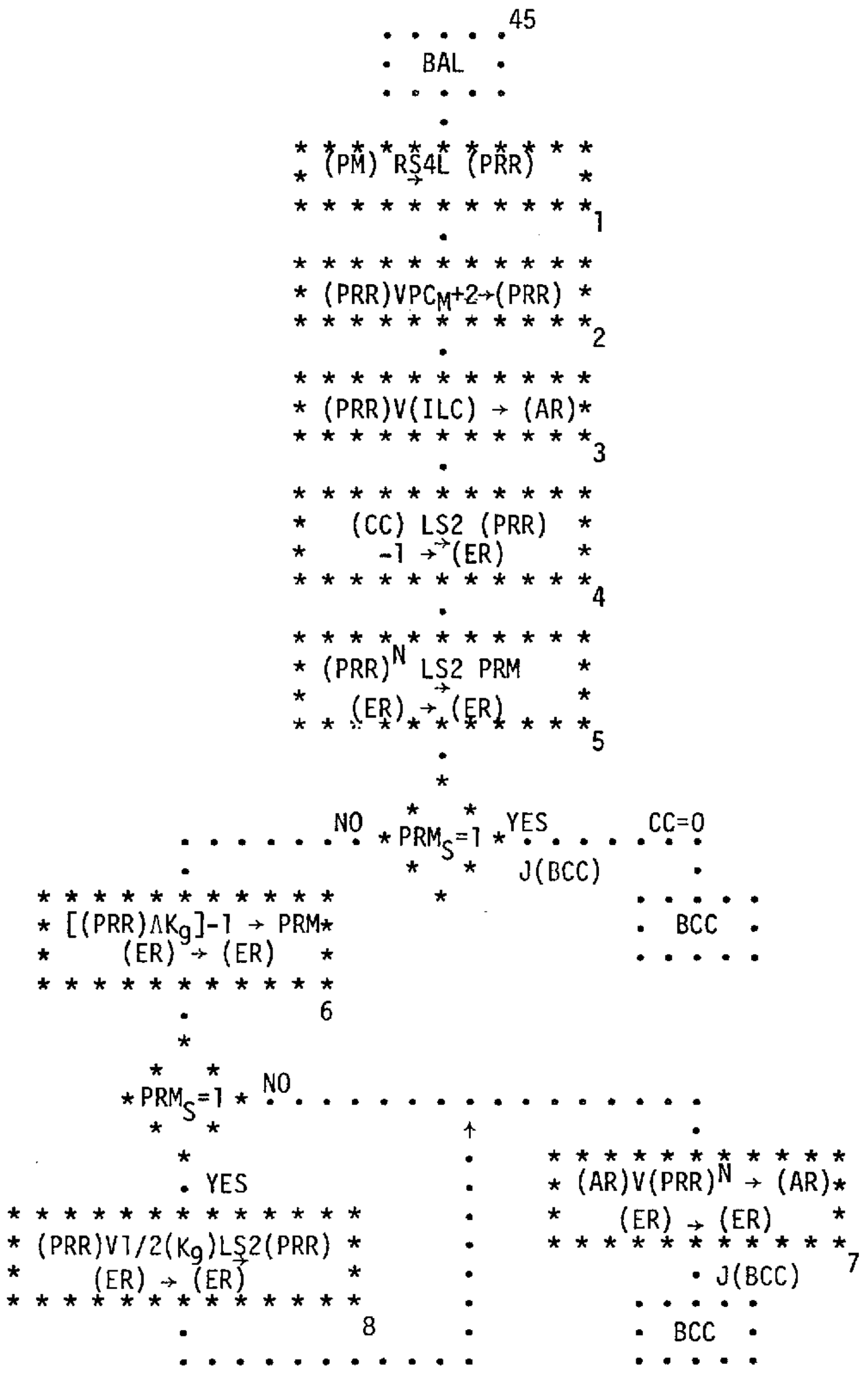


Figure 6.14. Branch and Link (BAL) Microprogram

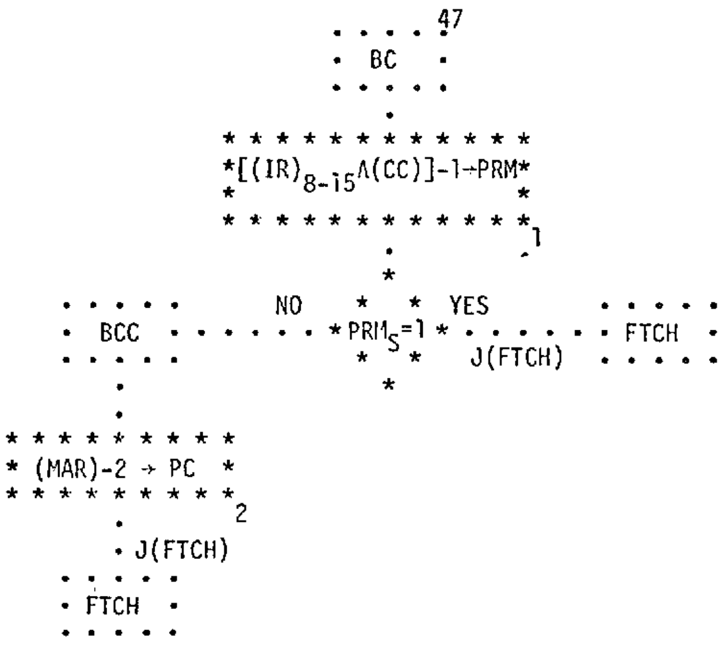


Figure 6.15. Branch on Condition (BC) Microprogram

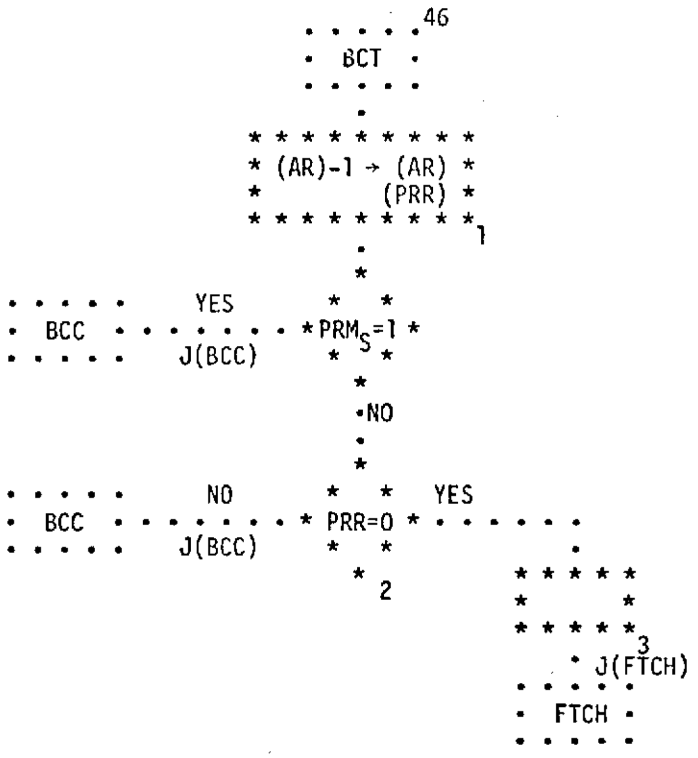


Figure 6.16. Branch on Count (BCT) Microprogram

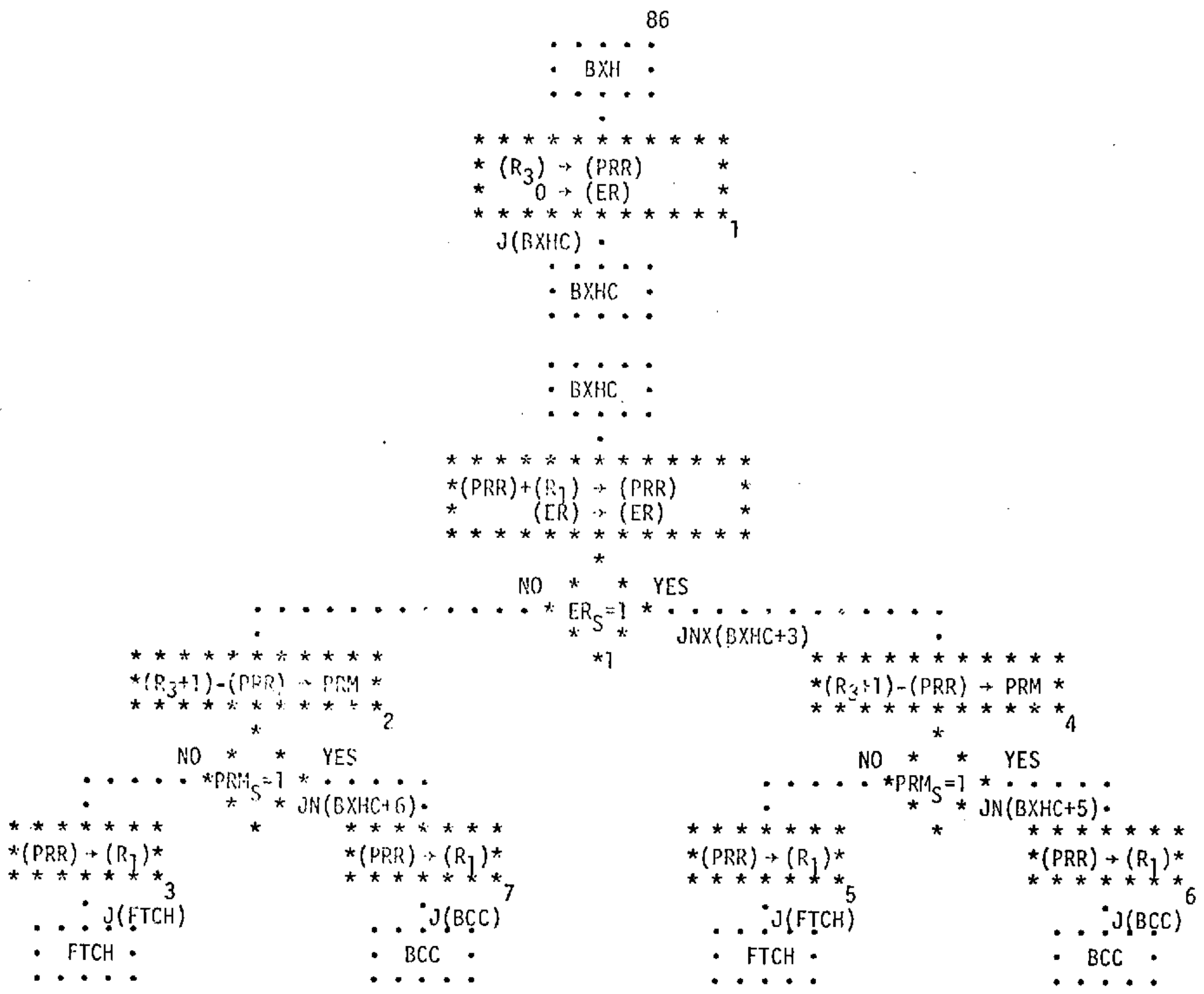

86
BXH
$(R_3) \rightarrow (PRR)$
$0 \rightarrow (ER)$
1
J(BXHC)
BXHC
BXHC
$(PRR)+(R_1) \rightarrow (PRR)$
$(ER) \rightarrow (ER)$
NO
YES
$ER_S=1$
1
JNX(BXHC+3)
$(R_3+1)-(PRR) \rightarrow PRM$
2
$(R_3+1)-(PRR) \rightarrow PRM$
4
NO
YES
$PRM_S=1$
JN(BXHC+6)
NO
YES
$PRM_S=1$
JN(BXHC+5)
$(PRR) \rightarrow (R_1)$
3
$(PRR) \rightarrow (R_1)$
7
$(PRR) \rightarrow (R_1)$
5
$(PRR) \rightarrow (R_1)$
6
J(FTCH)
J(BCC)
J(FTCH)
J(BCC)
FTCH
BCC
FTCH
BCC

Figure 6.17. Branch on Index High (BXH) Microprogram

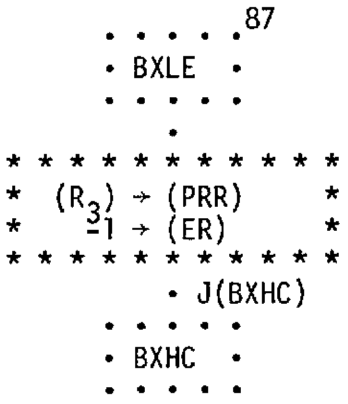


Figure 6.18. Branch on Index Low or Equal (BXLE) Microprogram

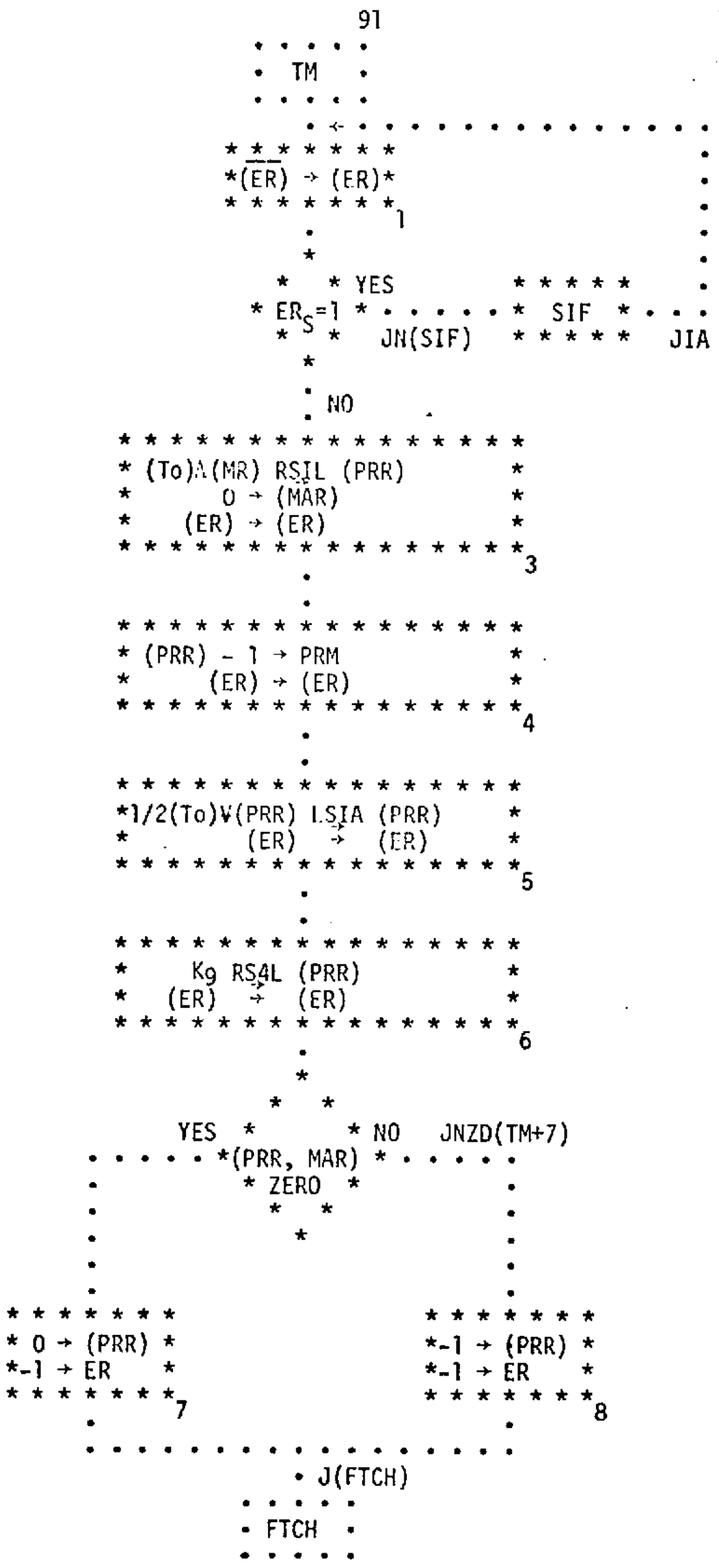


Figure 6.19. Test Under Mark (TM) Microprogram

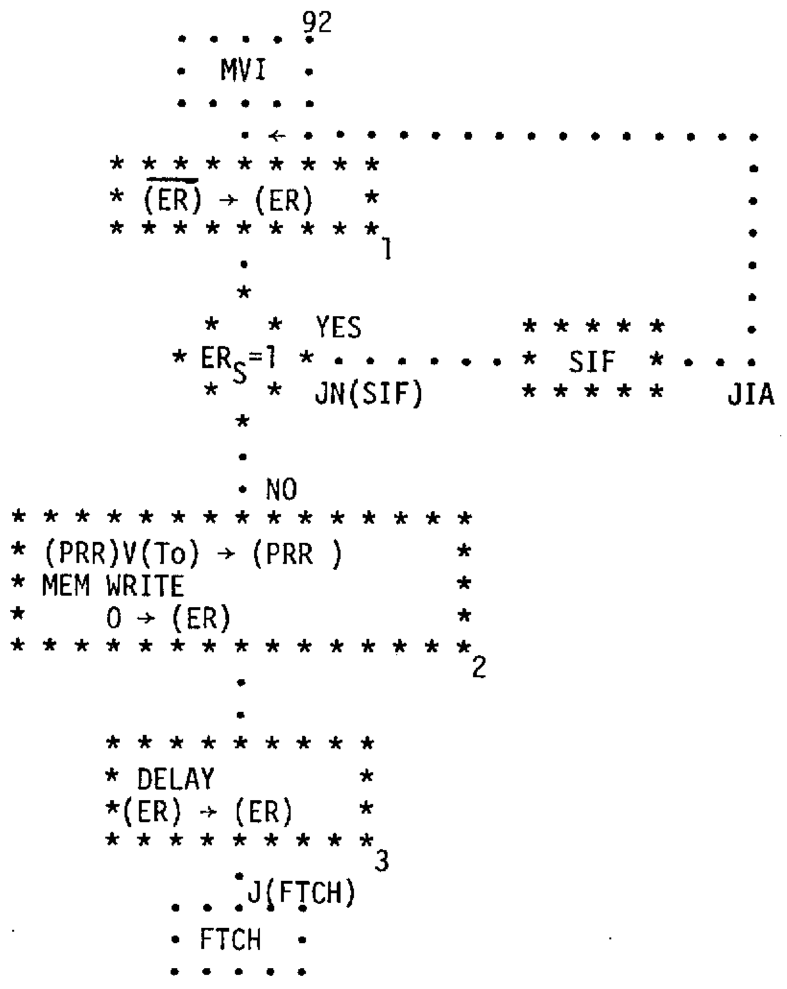


Figure 6.20. Move Immediate (MVI) Microprogram

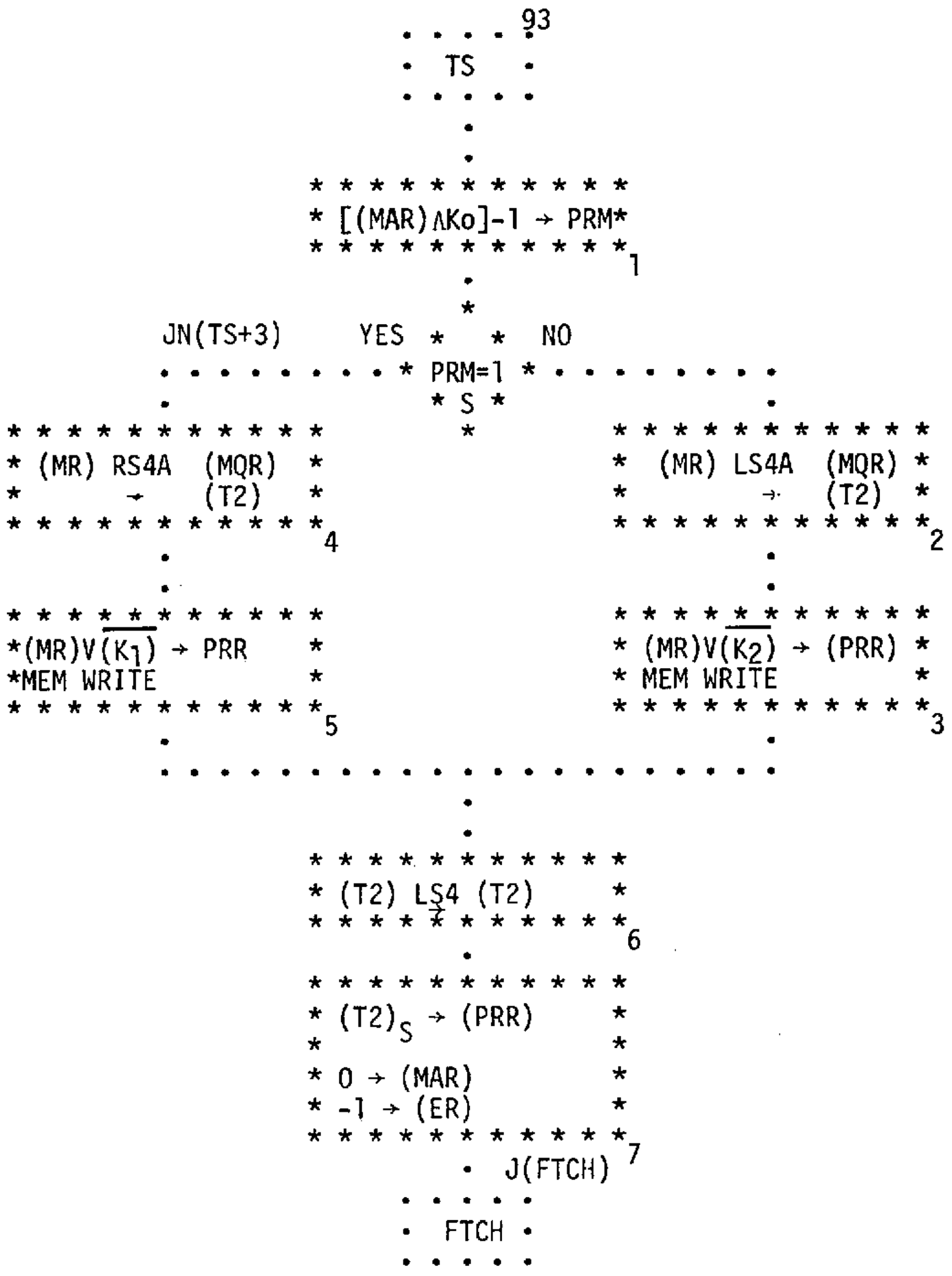


Figure 6.21. Test and Set (TS) Microprogram

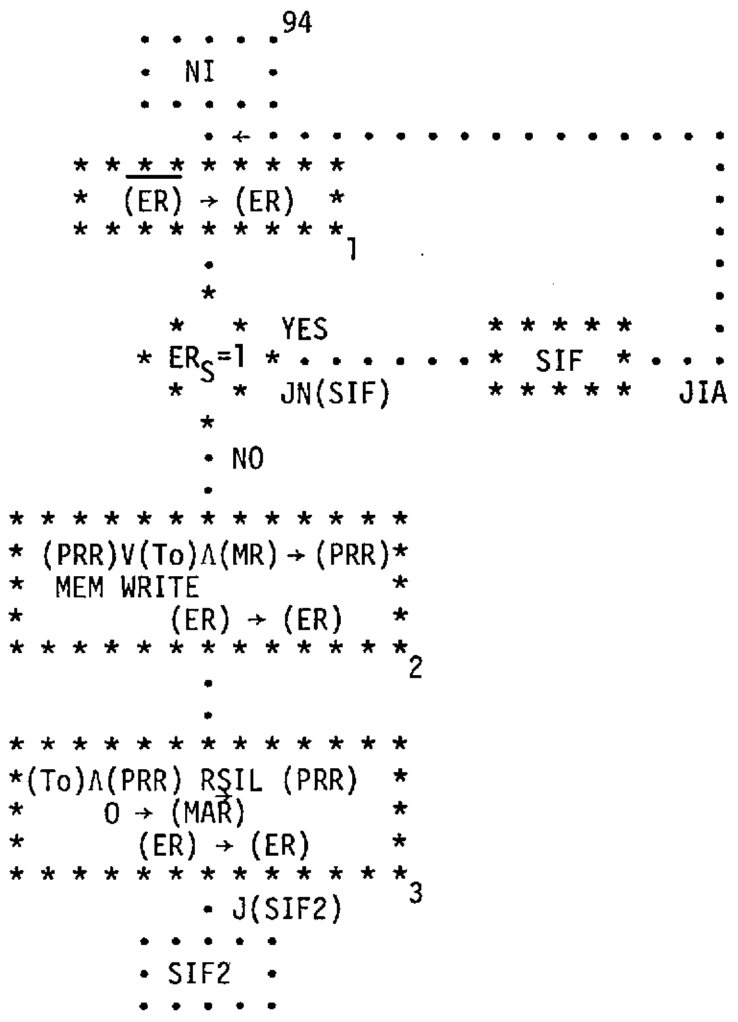


Figure 6.22. And Immediate (NI) Microprogram

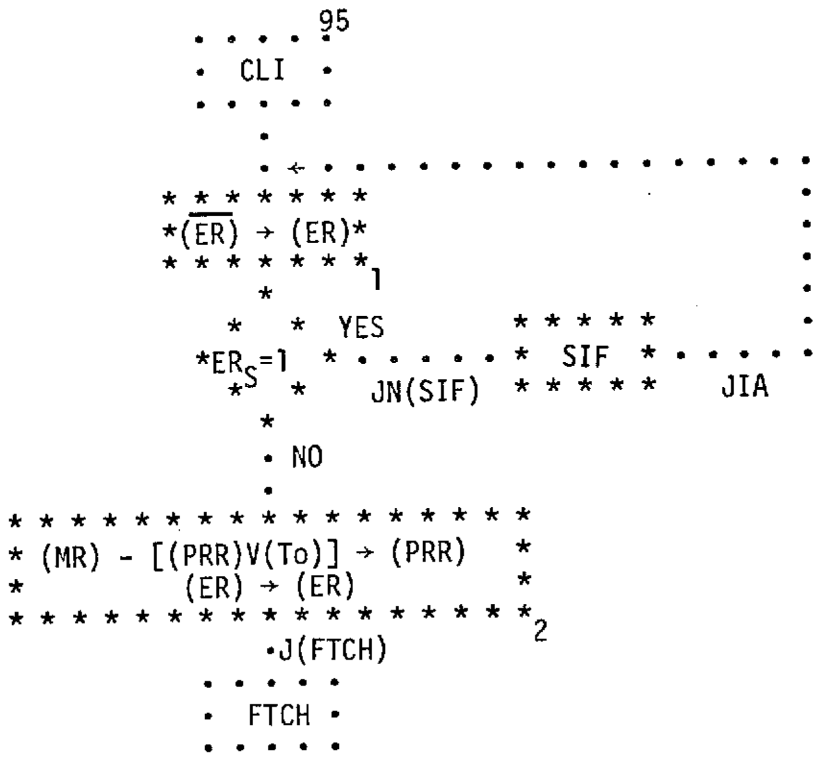

95
CLI
$(\overline{ER}) \rightarrow (ER)$
1
YES
$ER_S=1$
SIF
JN(SIF)
JIA
NO
$(MR) - [(PRR)V(To)] \rightarrow (PRR)$
$(ER) \rightarrow (ER)$
2
J(FTCH)
FTCH

Figure 6.23. Compare Logical Immediate (CLI) Microprogram

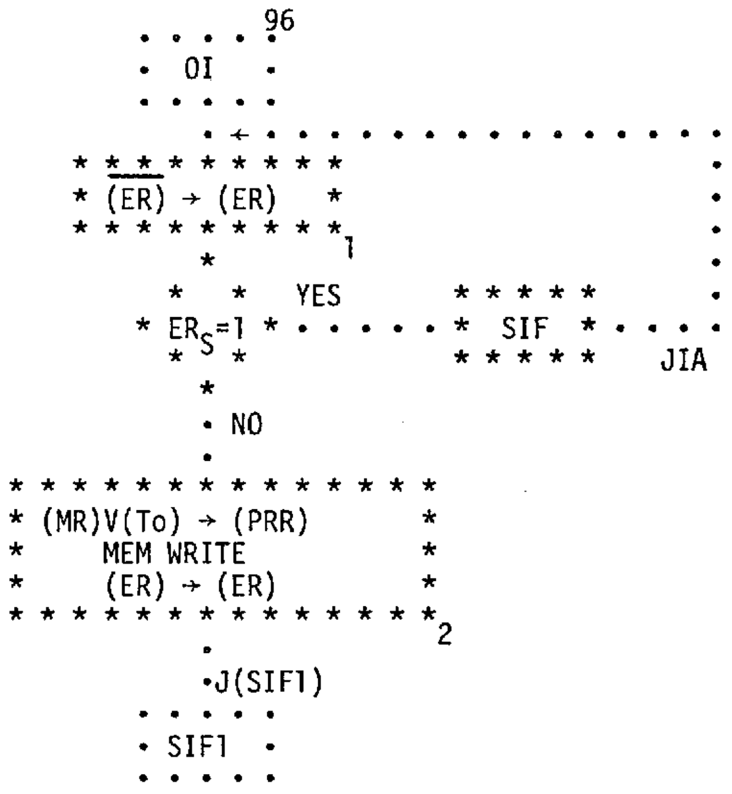


Figure 6.24. Or Immediate (OI) Microprogram

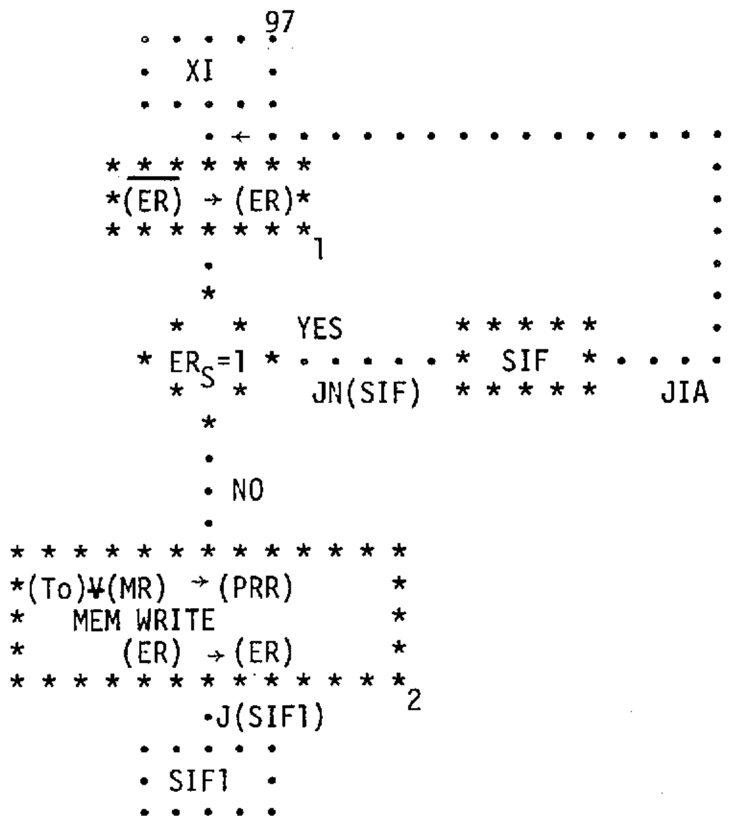


Figure 6.25. Exclusive Or Immediate (XI) Microprogram

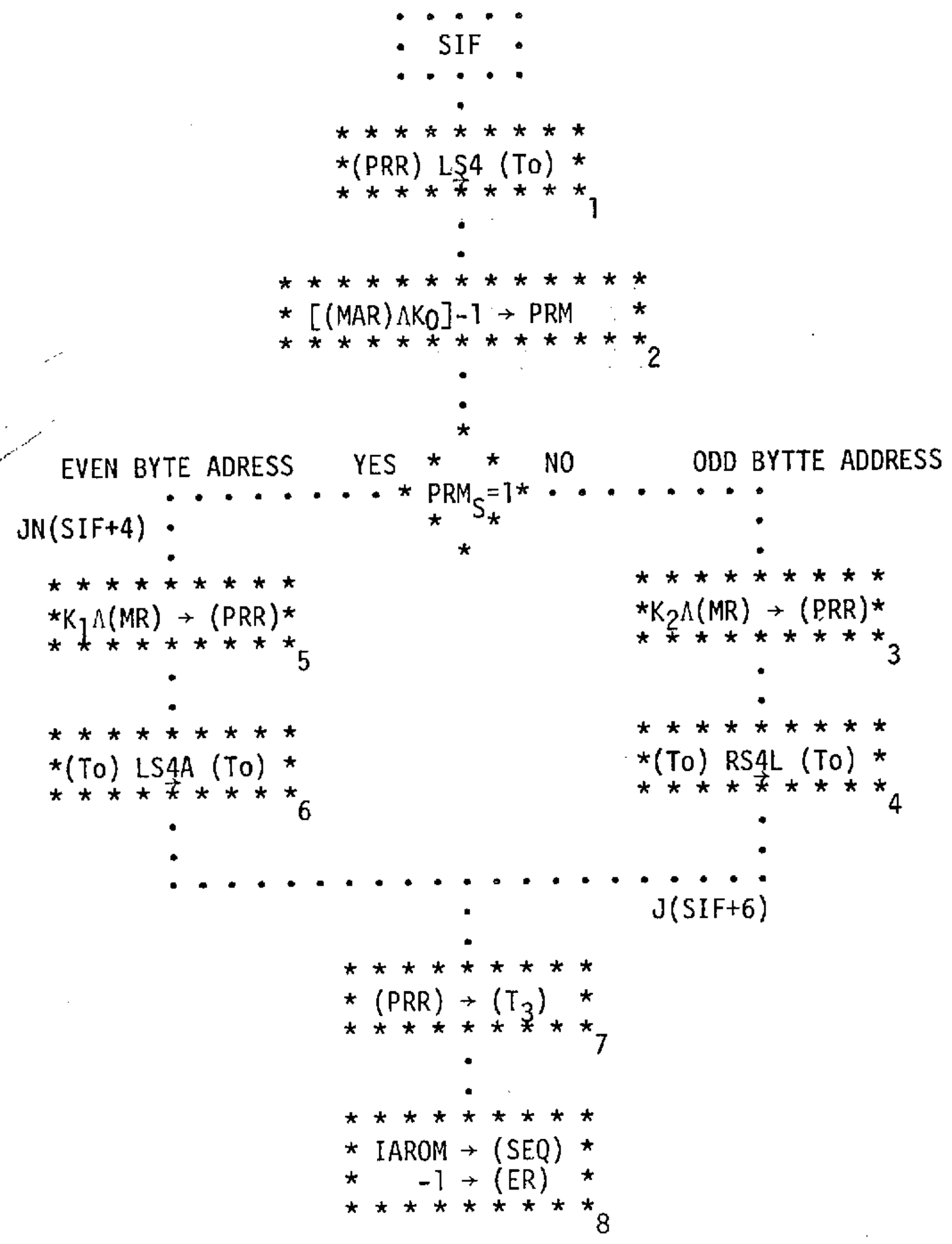


Figure 6.26. SI Format (SIR) Microprogram

PRECEDING PAGE BLANK NOT FILMED

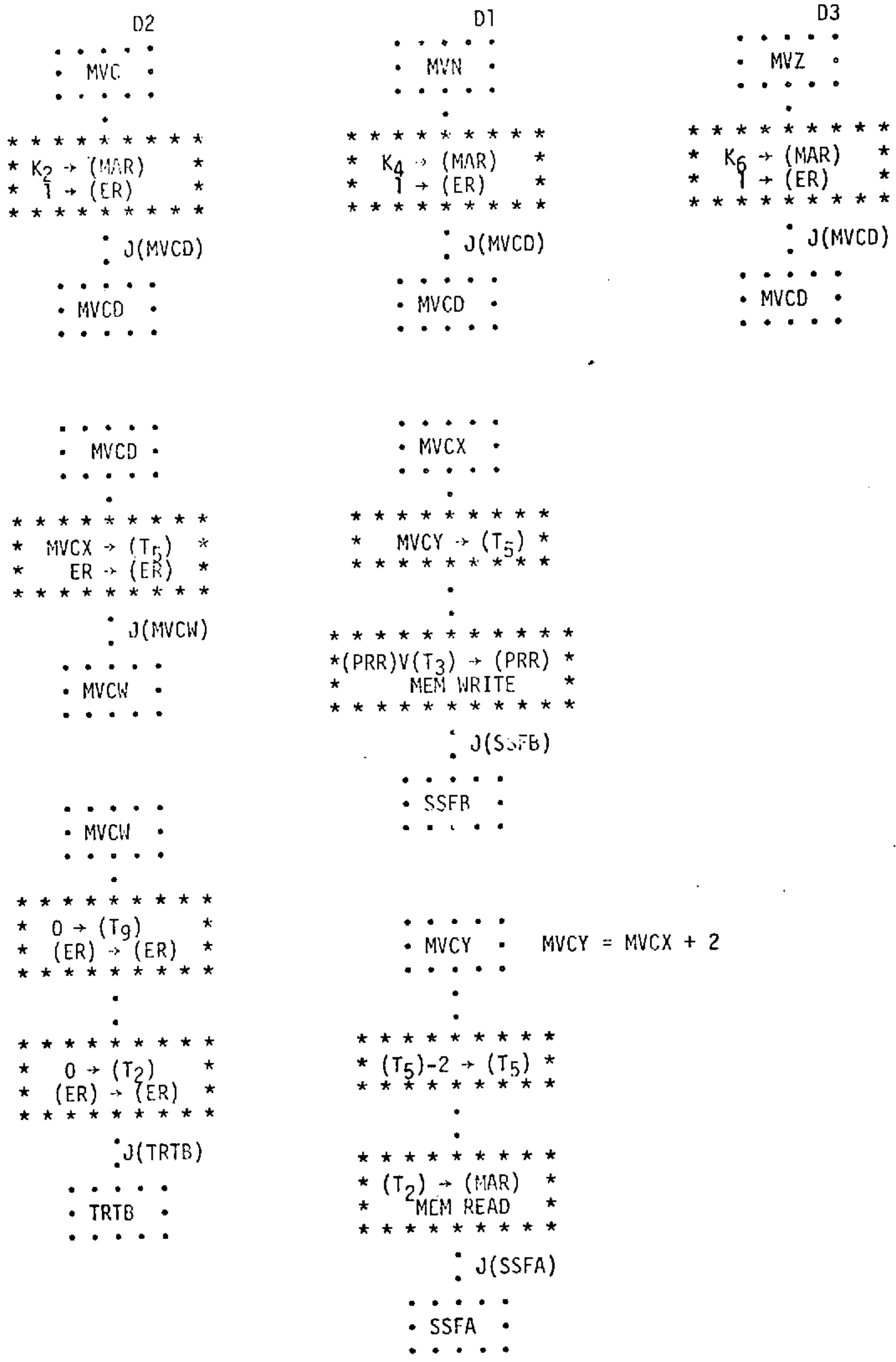


Figure 6.28. Move Microprograms (MVC, MVN, MVZ)

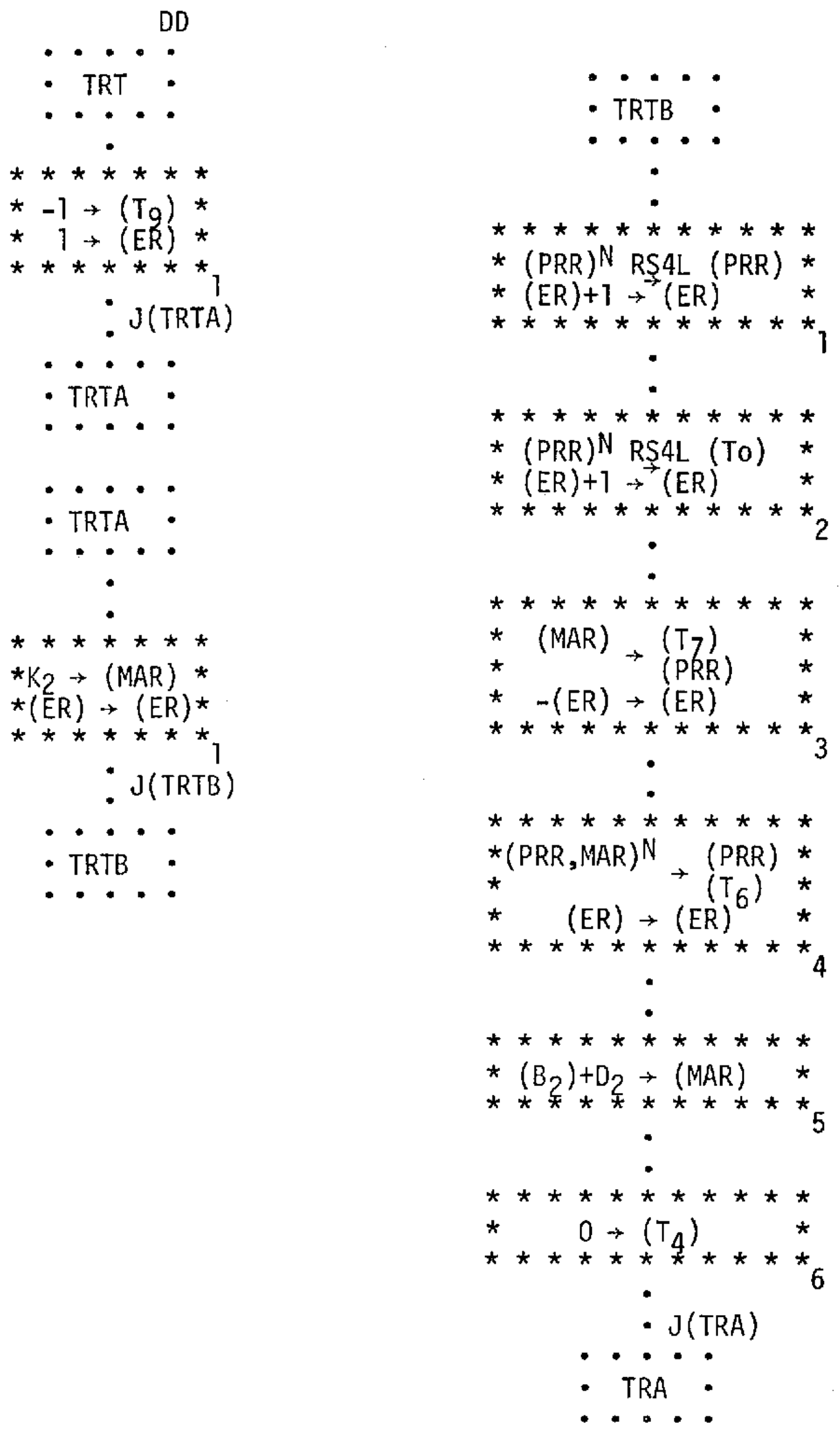


Figure 6.29. Translate and Test (TRT) Microprogram

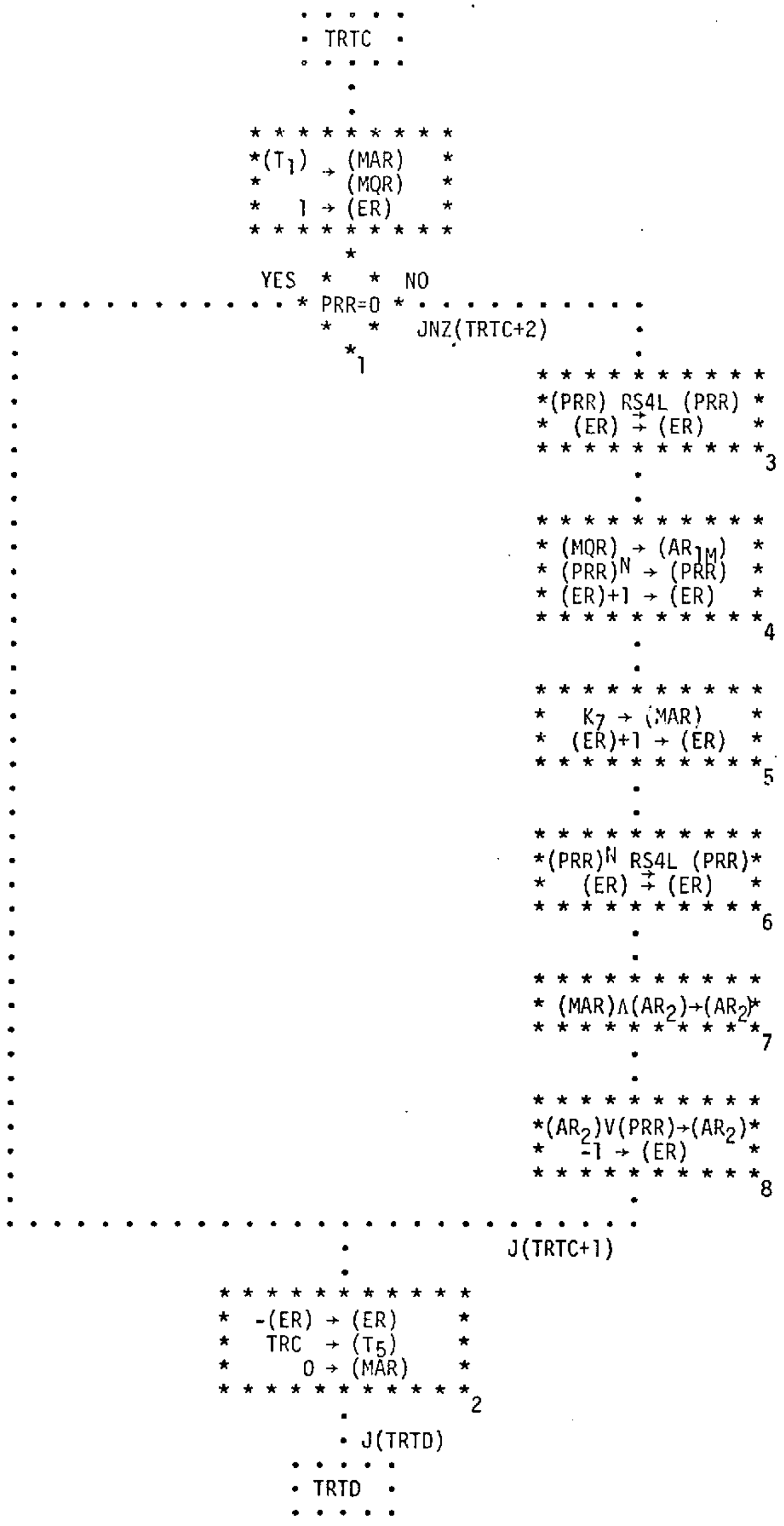


Figure 6.30. TRTC Microprogram

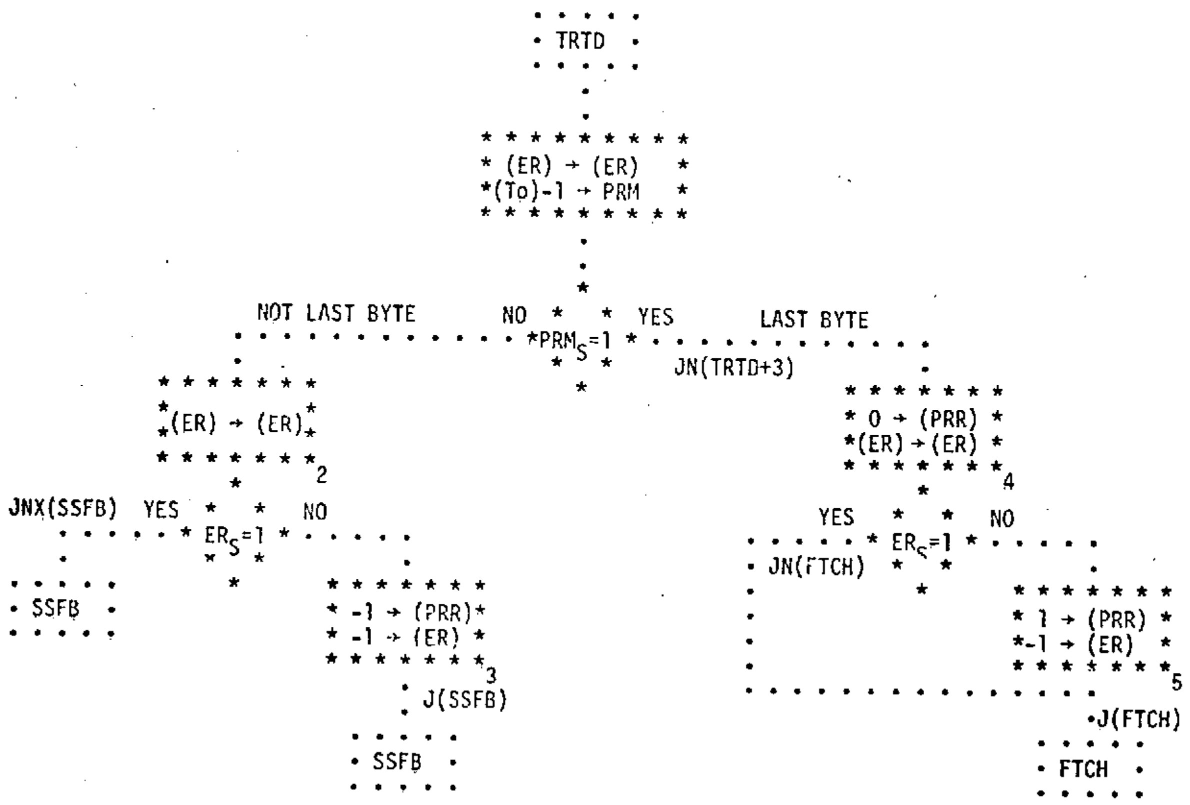


Figure 6.31. TRTD Microprogram

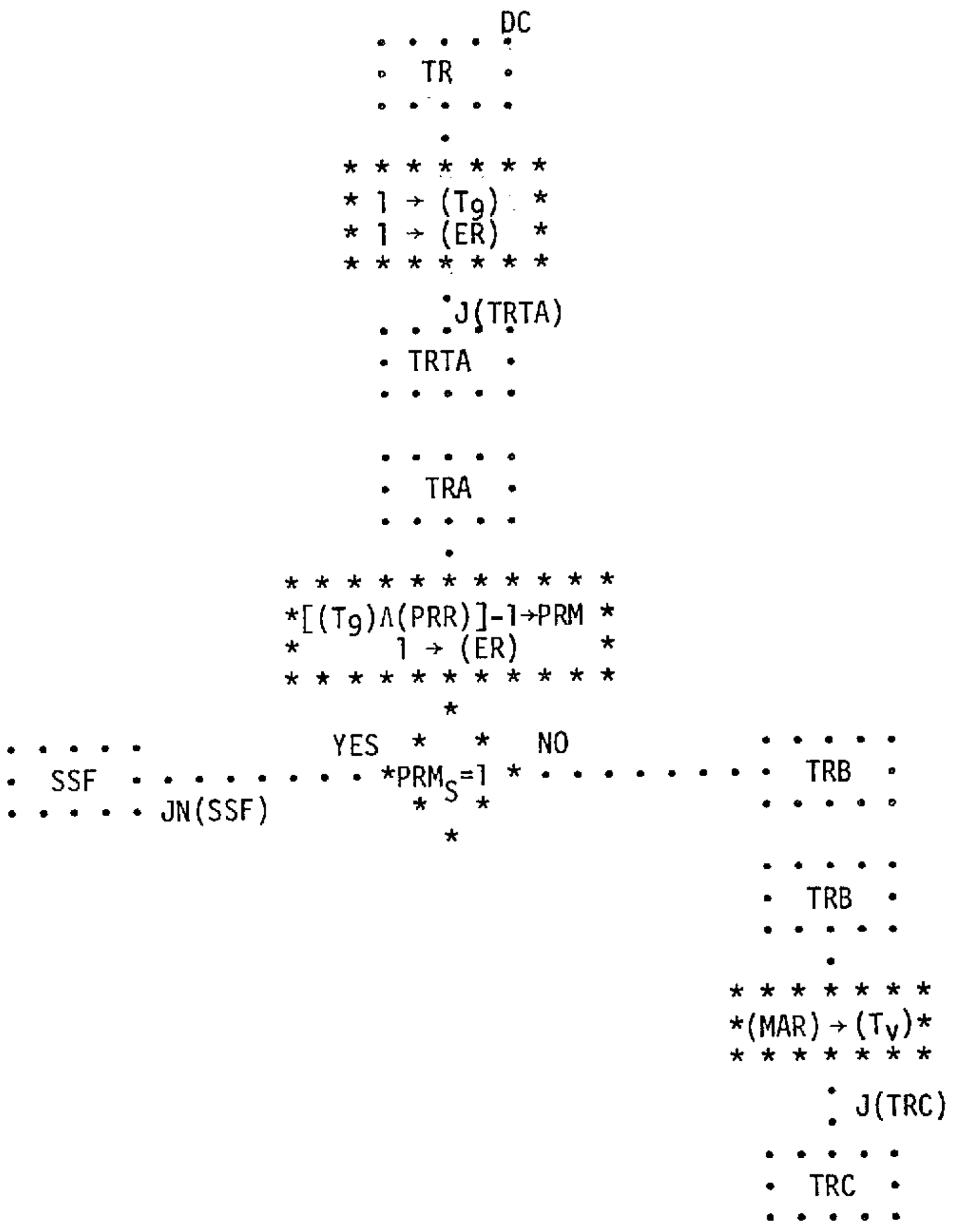


Figure 6.32. Translate (TR) Microprogram

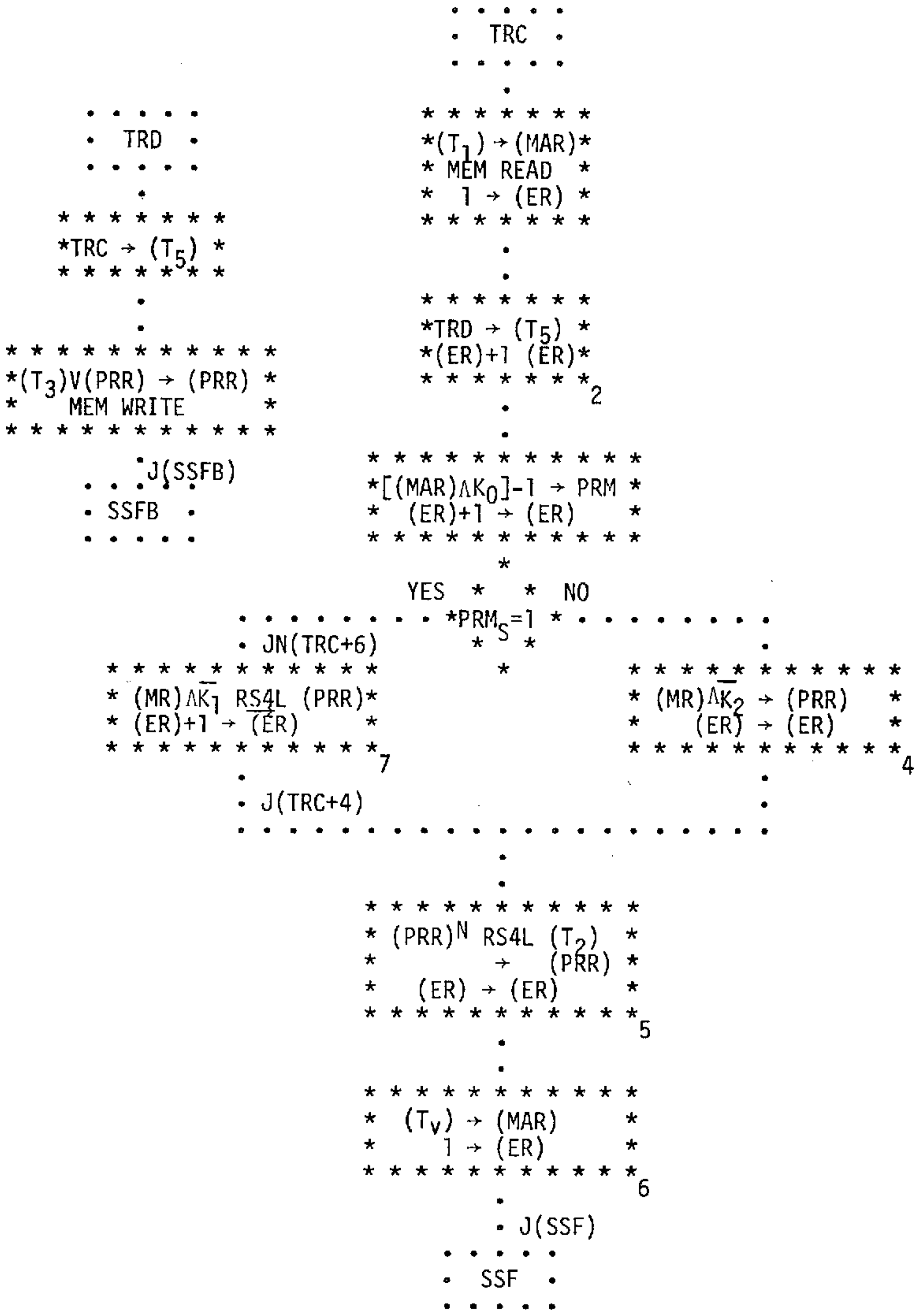

TRC
$(T_1) \rightarrow (MAR)$
MEM READ
$1 \rightarrow (ER)$
$TRD \rightarrow (T_5)$
$(ER)+1 \ (ER)$
2
$[(MAR) \wedge K_0]-1 \rightarrow PRM$
$(ER)+1 \rightarrow (ER)$
YES
NO
$PRM_S=1$
JN(TRC+6)
$(MR) \wedge \overline{K_1}$ RS4L (PRR)
$(ER)+1 \rightarrow (ER)$
7
J(TRC+4)
$(MR) \wedge \overline{K_2} \rightarrow (PRR)$
$(ER) \rightarrow (ER)$
4
$(PRR)^N$ RS4L $(T_2) \rightarrow (PRR)$
$(ER) \rightarrow (ER)$
5
$(T_V) \rightarrow (MAR)$
$1 \rightarrow (ER)$
6
J(SSF)
SSF
TRD
$TRC \rightarrow (T_5)$
$(T_3)V(PRR) \rightarrow (PRR)$
MEM WRITE
J(SSFB)
SSFB

Figure 6.33. TRC Microprogram

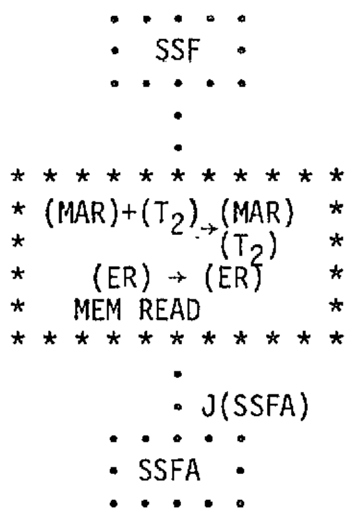


Figure 6.34. SS Format (SSF) Microprogram

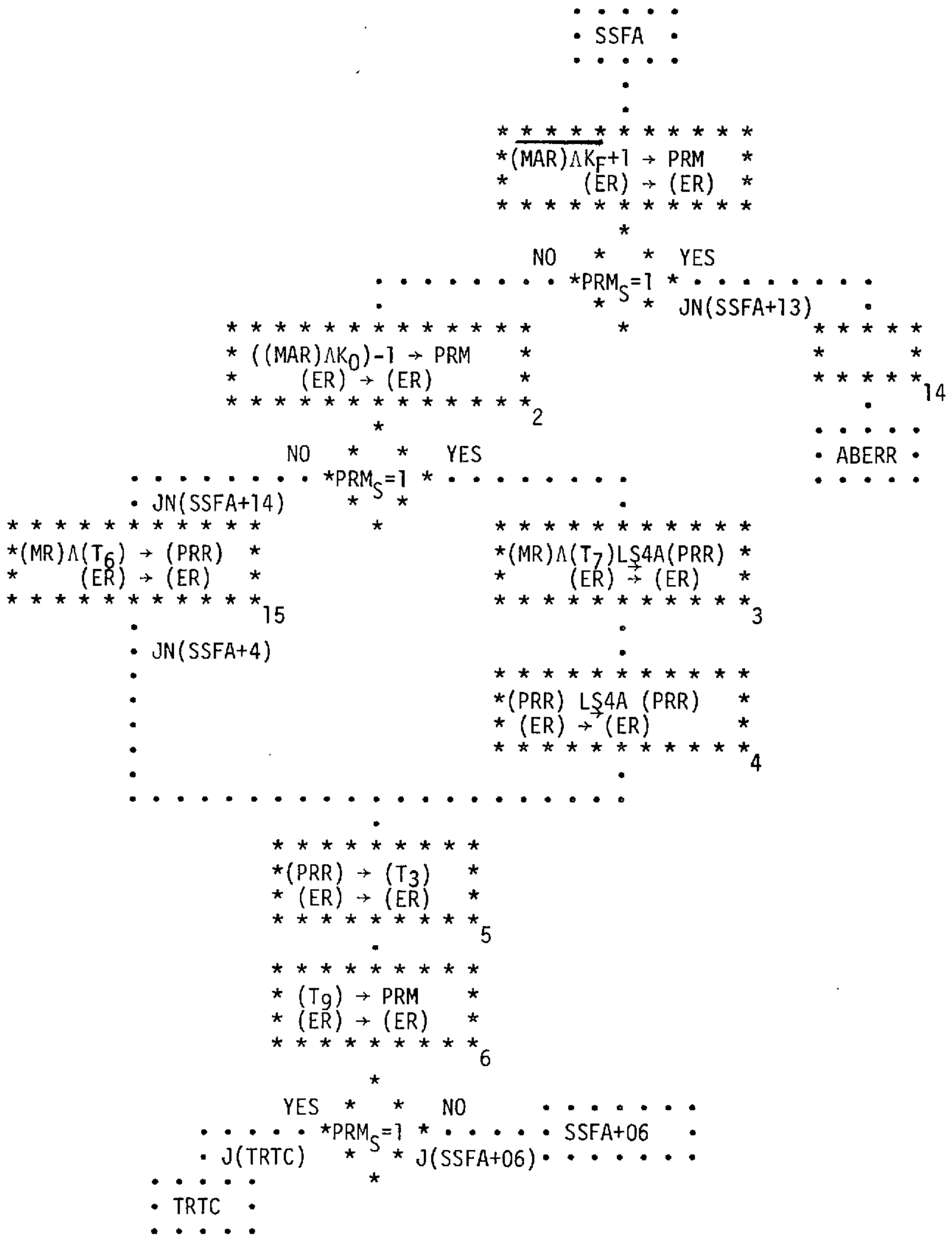


Figure 6.35. SS Format (SSFA) Microprogram

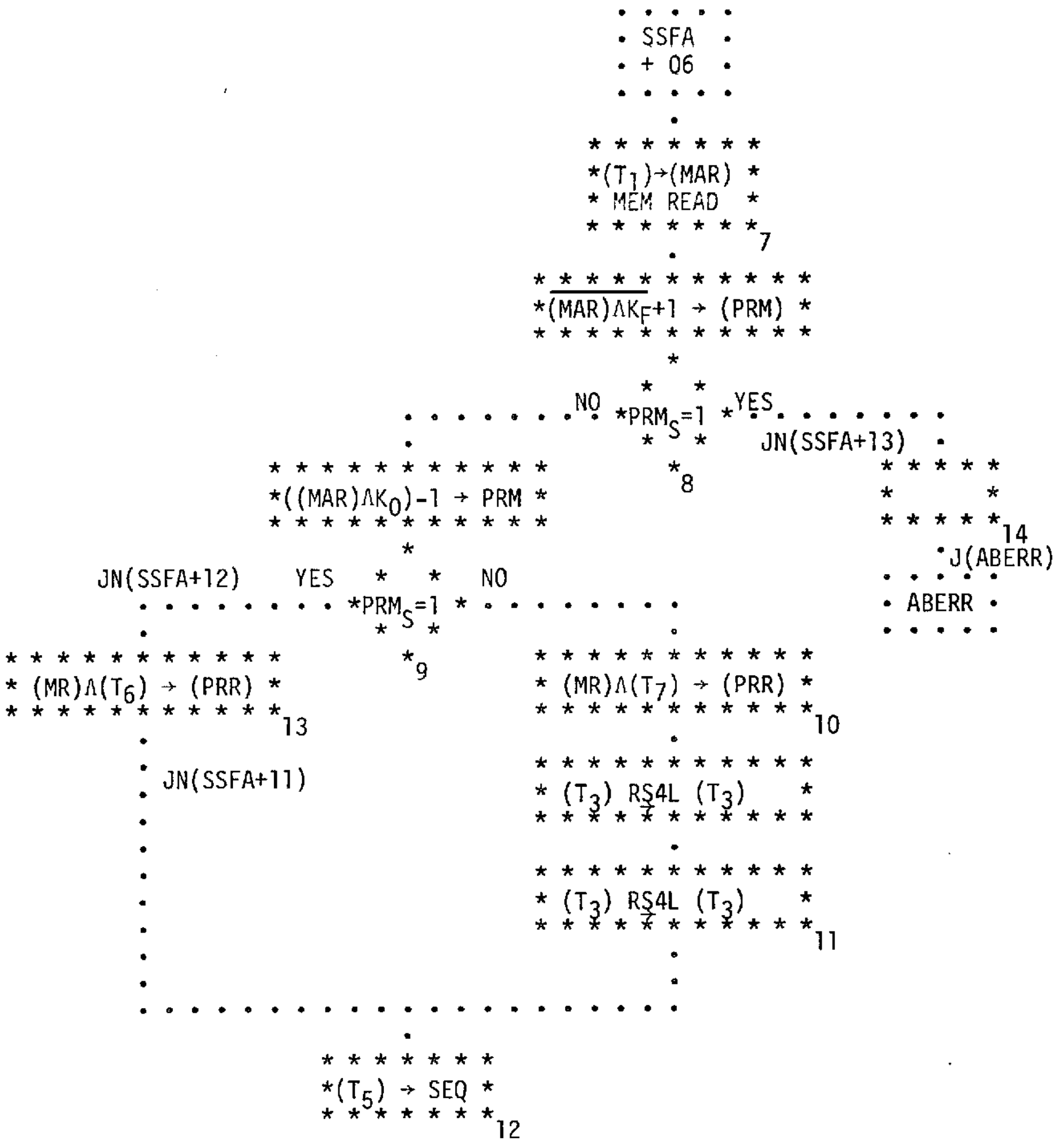

SSFA + 06
$(T_1) \rightarrow (MAR)$ MEM READ
7
$\overline{(MAR)\Lambda K_F}+1 \rightarrow (PRM)$
NO
$PRM_S=1$
YES
8
JN(SSFA+13)
14
J(ABERR)
ABERR
$((MAR)\Lambda K_0)-1 \rightarrow PRM$
JN(SSFA+12)
YES
$PRM_S=1$
NO
9
$(MR)\Lambda(T_6) \rightarrow (PRR)$
13
$(MR)\Lambda(T_7) \rightarrow (PRR)$
10
JN(SSFA+11)
(T_3) RS4L (T_3)
(T_3) RS4L (T_3)
11
$(T_5) \rightarrow SEQ$
12

Figure 6.36. SS Format (SSF+06) Microprogram

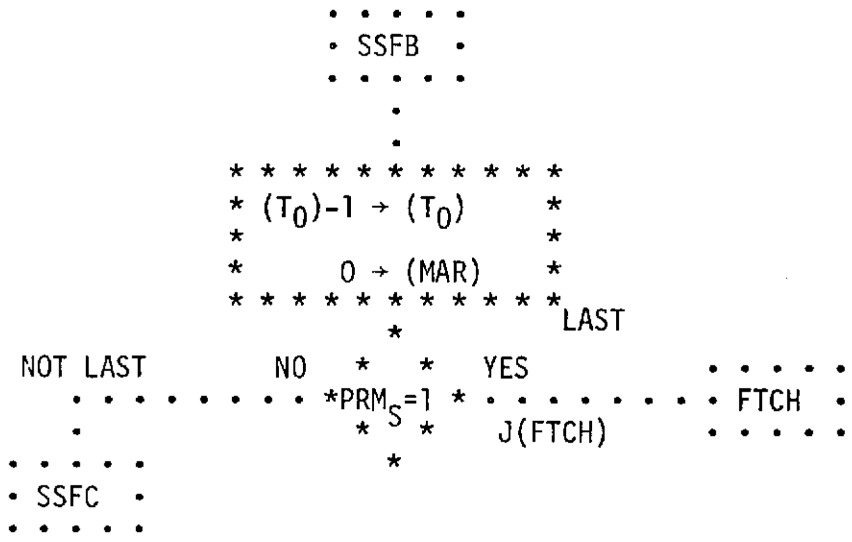


Figure 6.37. SS Format (SSFB) Microprogram

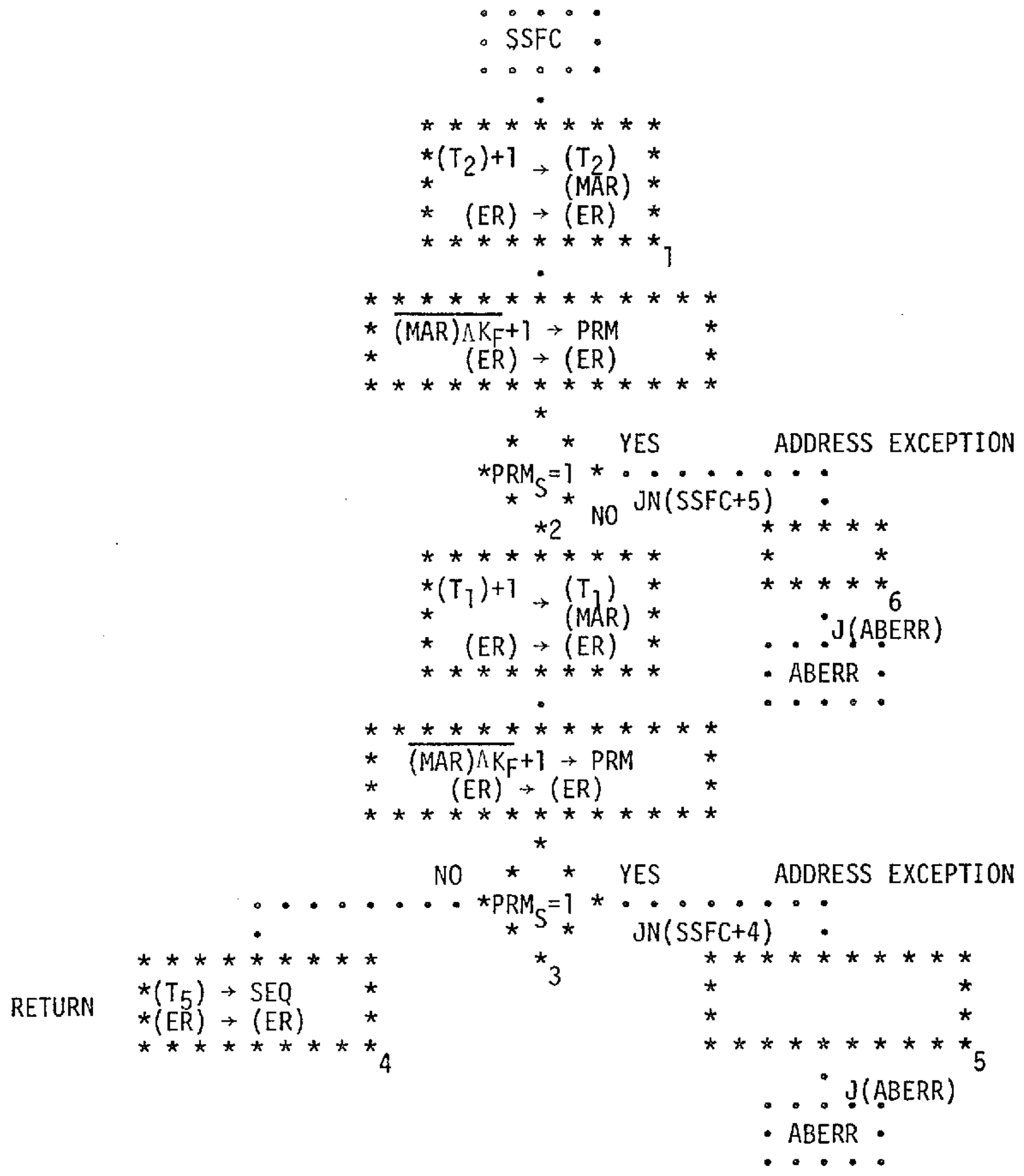


Figure 6.38. SS Format (SSFC) Microprogram

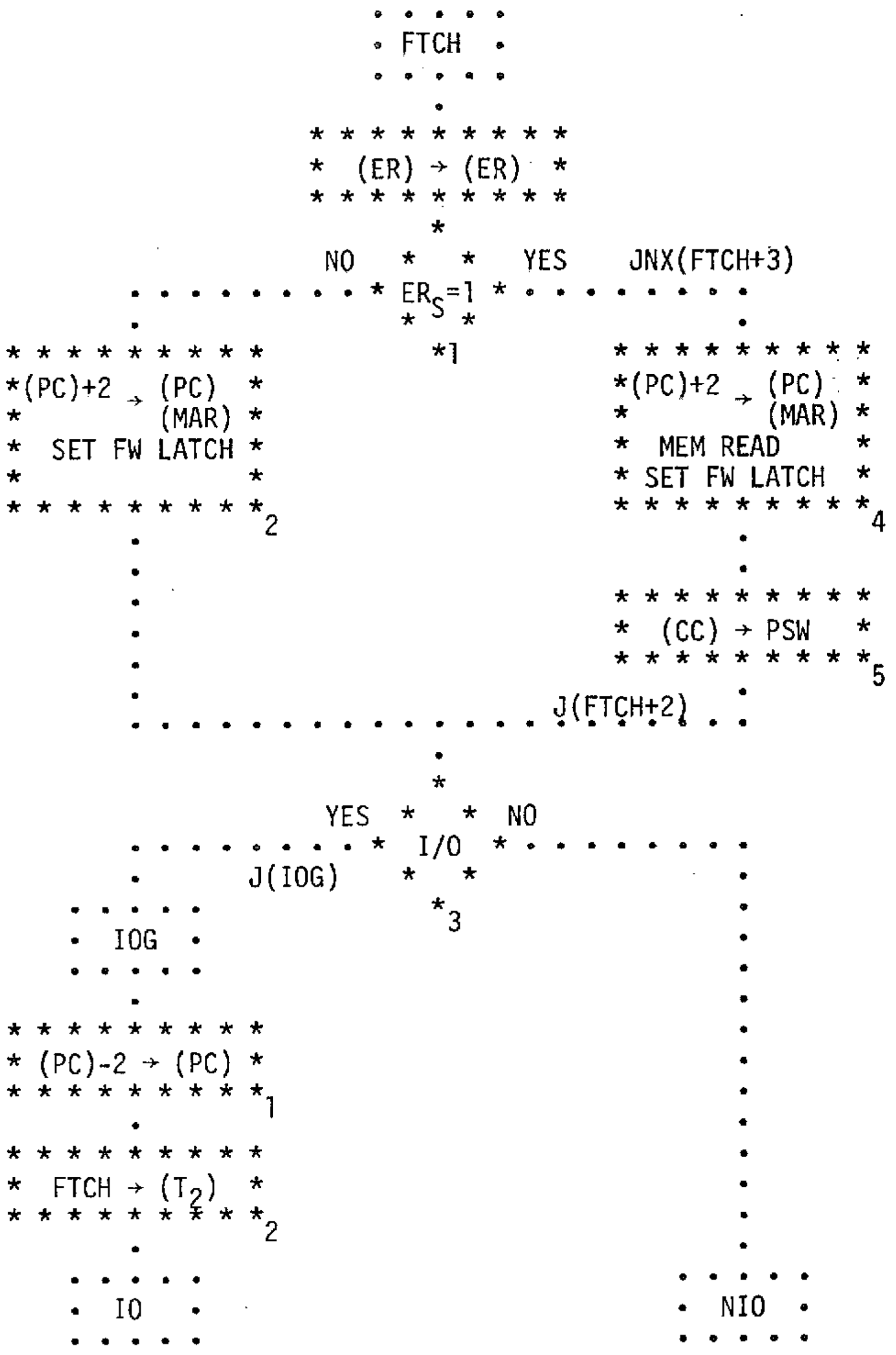


Figure 6.39. Fetch (FTCH) Microprogram

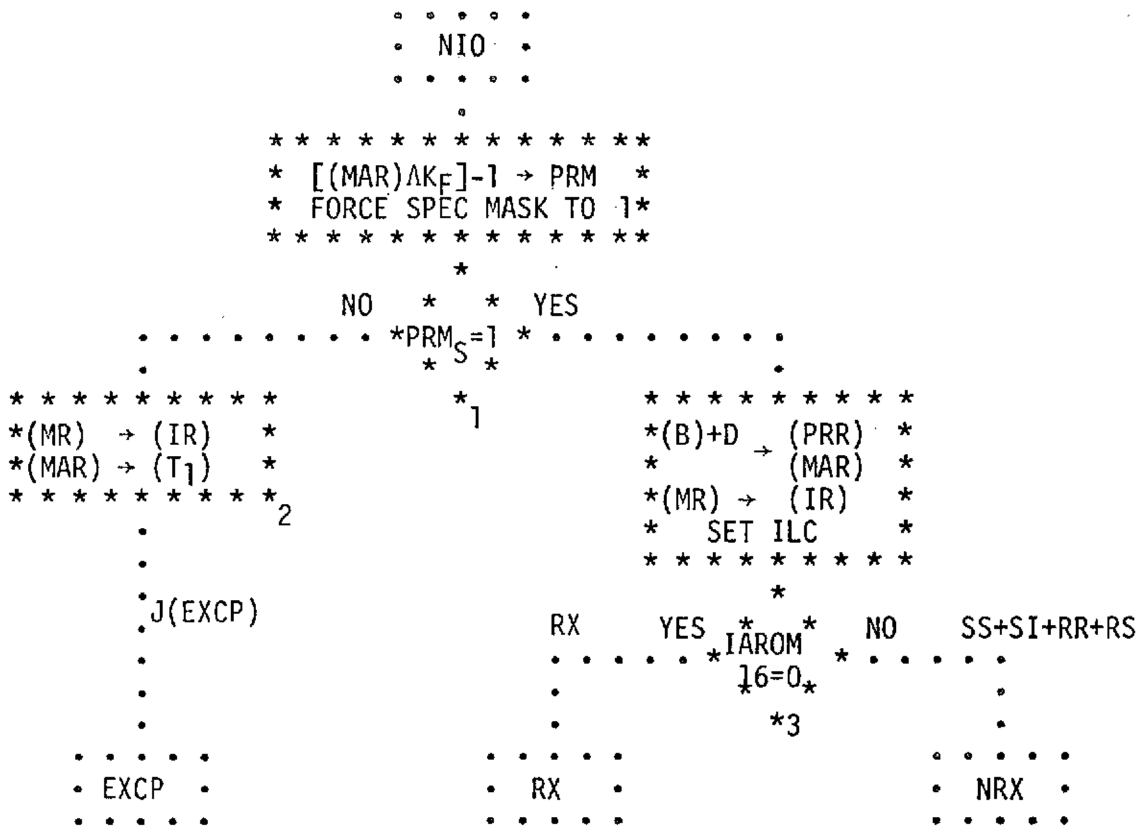


Figure 6.40. NIO Microprogram

```
          • • • • •
          • EXCP  •
          • • • • •
              •
              •
      * * * * * * * * *
      *(PC)-2 → (PC)  *
      *  SET ILC      *
      * * * * * * * * *1
              •
      * * * * * * * * *
      *   0 → (PRR)   *
      * ILC → (MAR)   *
      *  -1   ER      *
      * * * * * * * * *2
              •
    * * * * * * * * * * *
    *(PRR,MAR)^N RS1(PRR) *
    *    (ER) → ER        *
    * * * * * * * * * * *3
              •
    * * * * * * * * * * *
    * (PRR)+(PC) → (PC) *
    * * * * * * * * * * *4
              •
      * * * * * * * * *
      *(T1) → (MAR)   *
      * * * * * * * * *5
              •
              • J(ABERR)
          • • • • •
          • ABERR •
          • • • • •
```

Figure 6.41. EXCP Microprogram

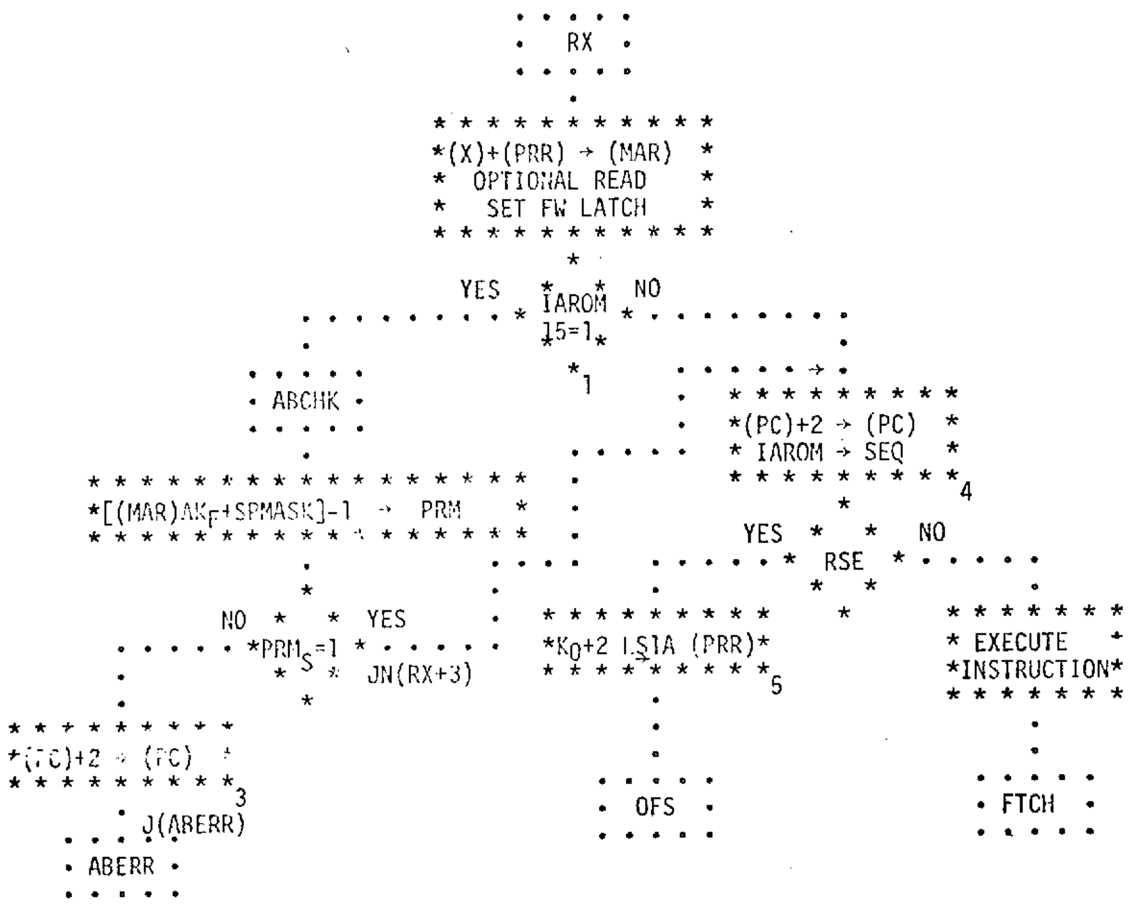


Figure 6.42. RX Microprogram

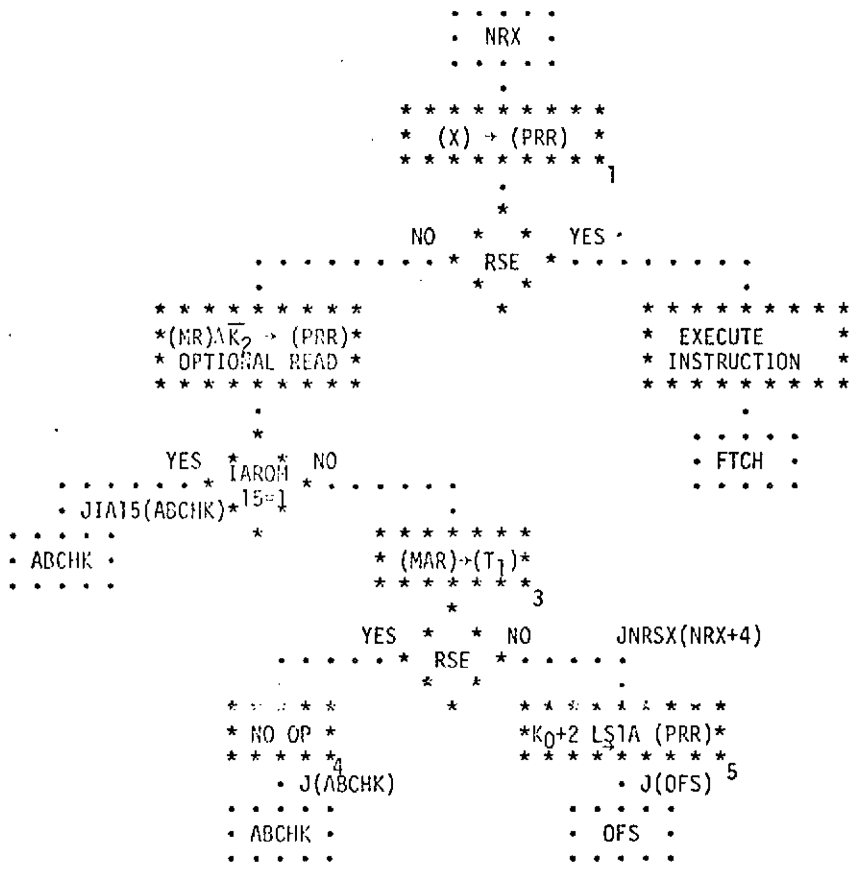


Figure 6.43. NRX Microprogram

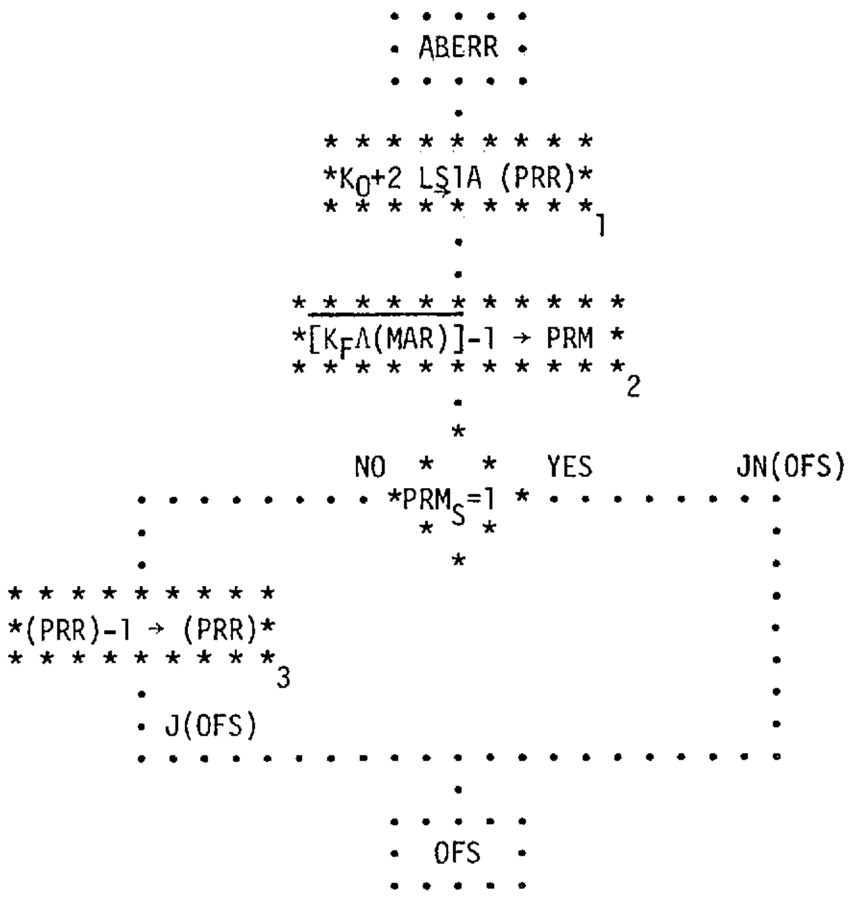

ABERR
K_0+2 L§1A (PRR)
1
$[K_F \wedge (MAR)]-1 \rightarrow$ PRM
2
NO
YES
JN(OFS)
$PRM_S=1$
$(PRR)-1 \rightarrow (PRR)$
3
J(OFS)
OFS

Figure 6.44. ABERR Microprogram

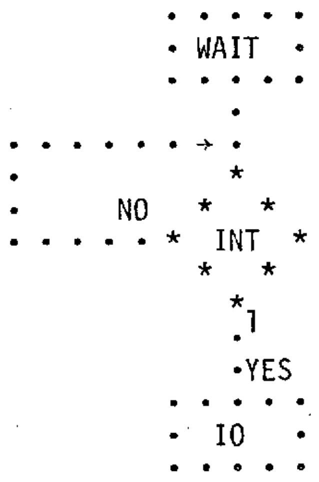


Figure 6.45. WAIT Microprogram

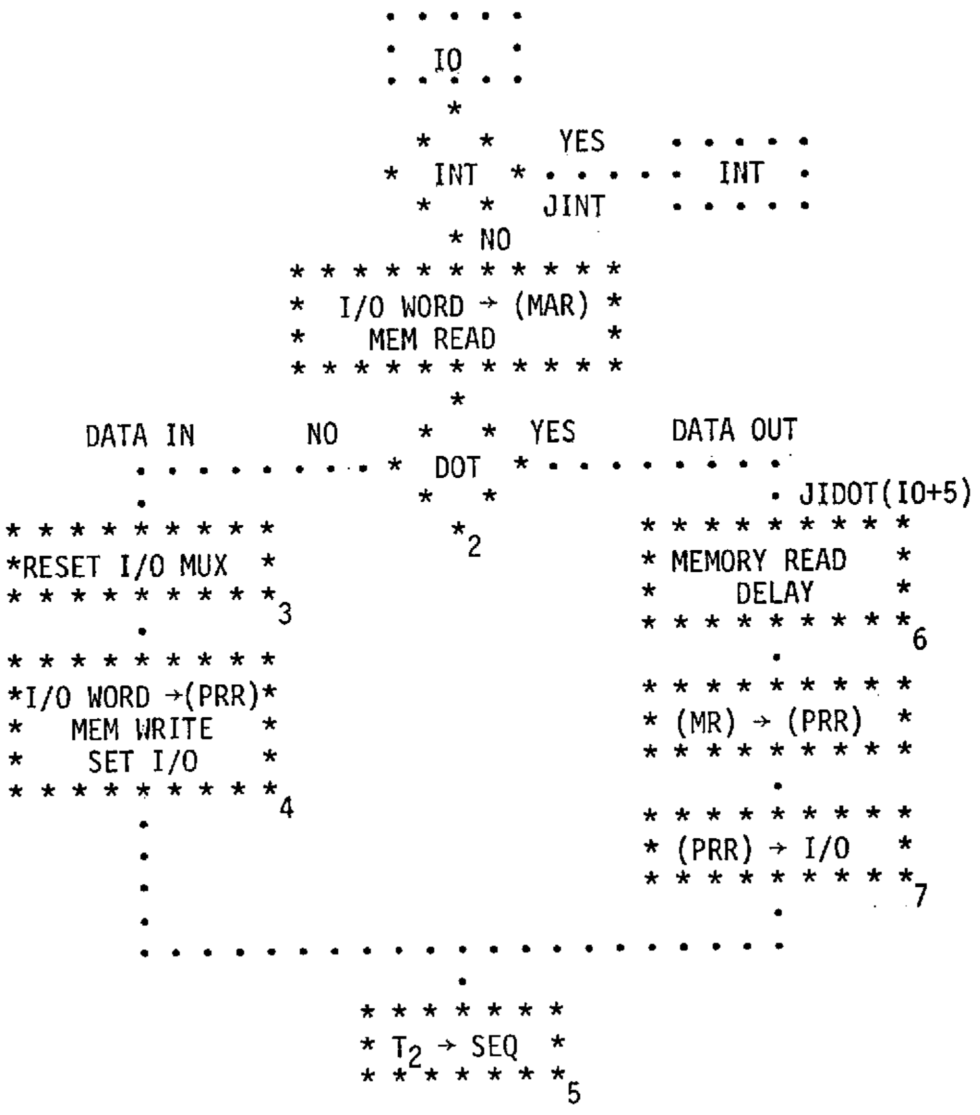


Figure 6.46. IO Microprogram

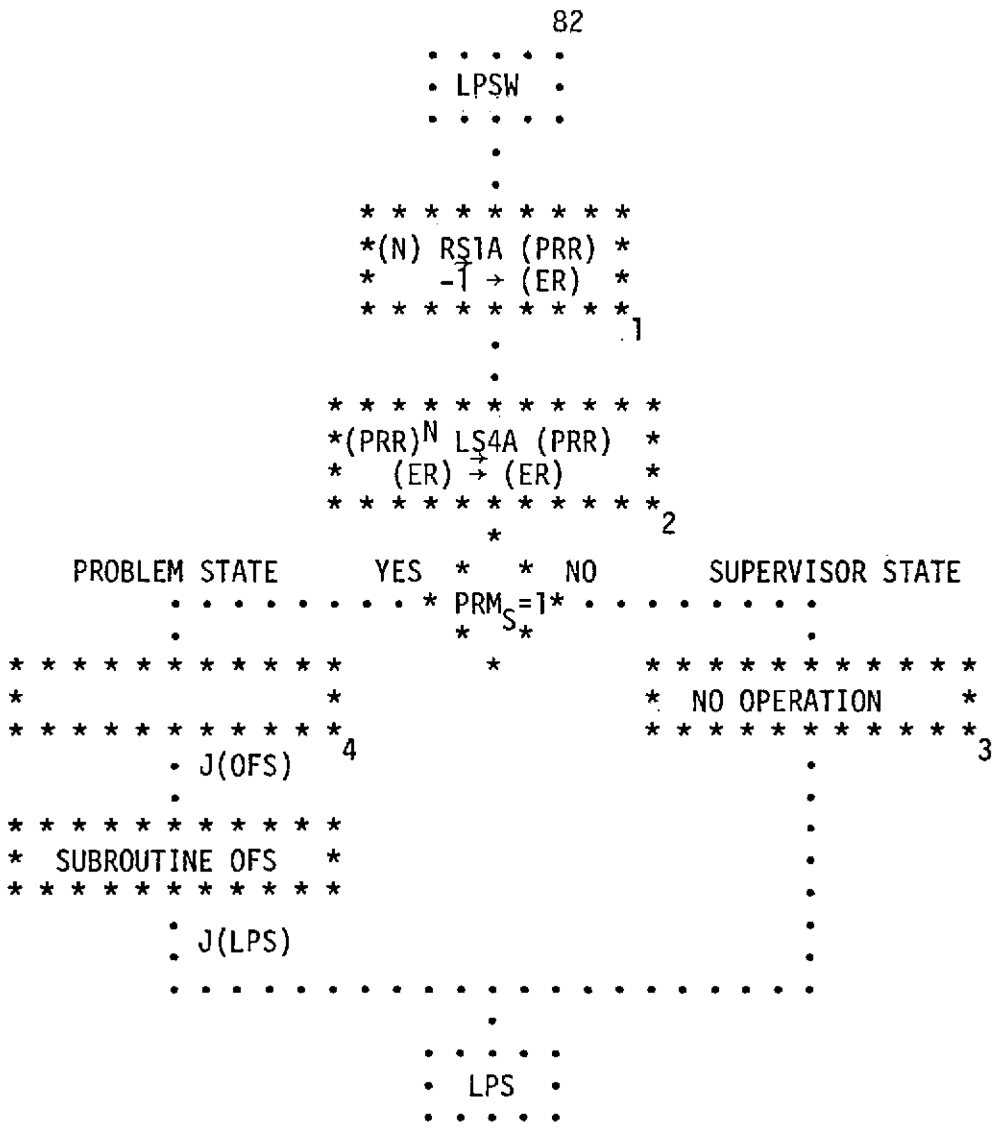


Figure 6.47. Load Program Status Word (LPSW) Microprogram

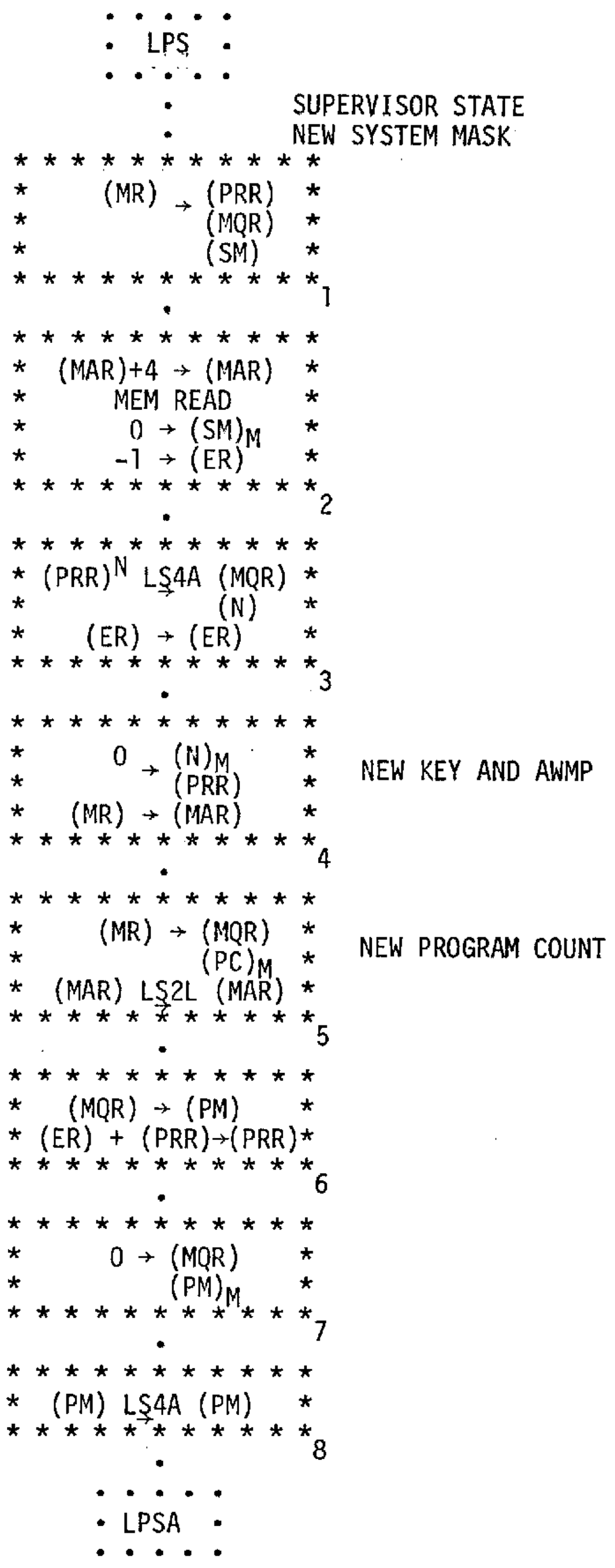


Figure 6.48. LPS Microprogram

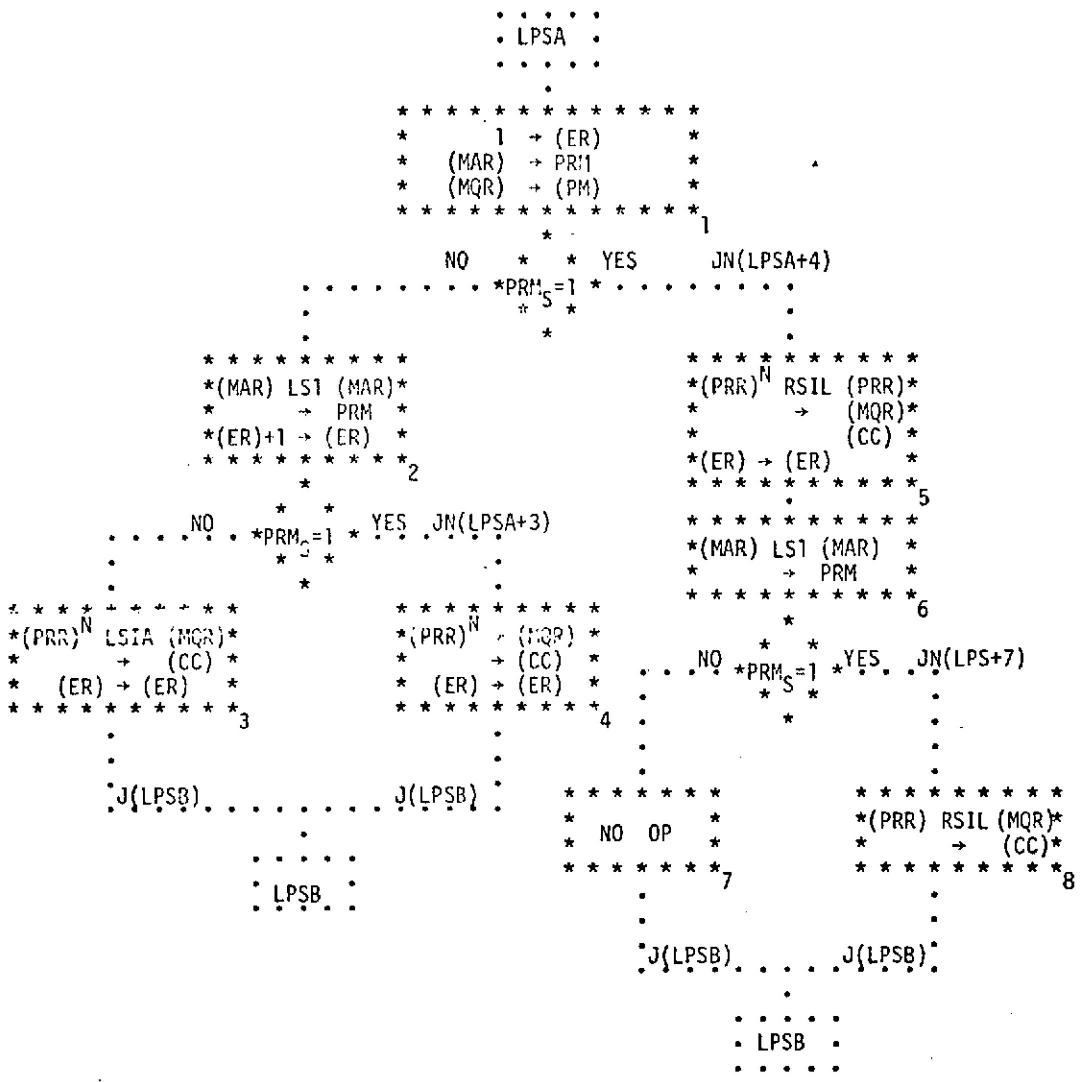


Figure 6.49. LPSA Microprogram

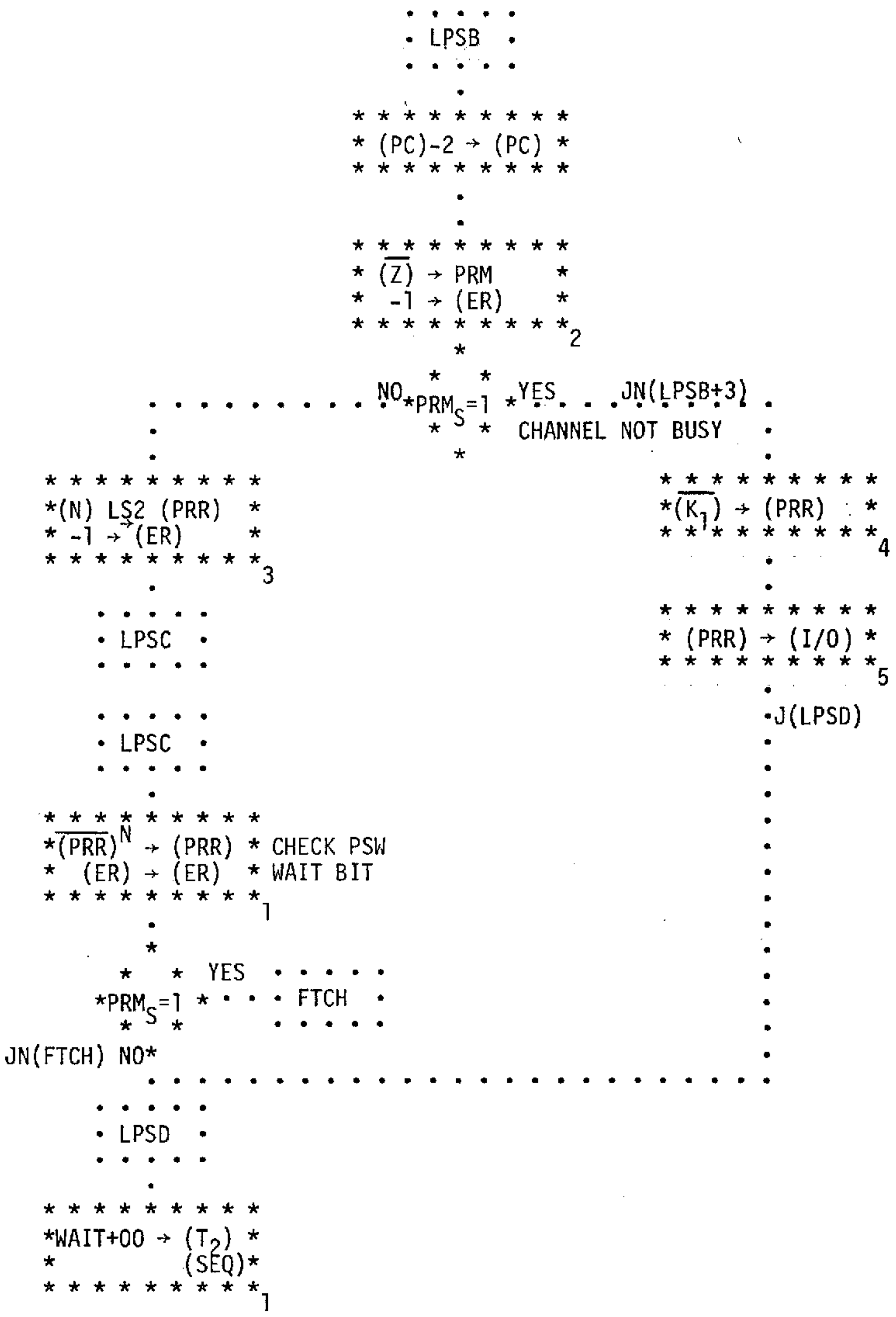
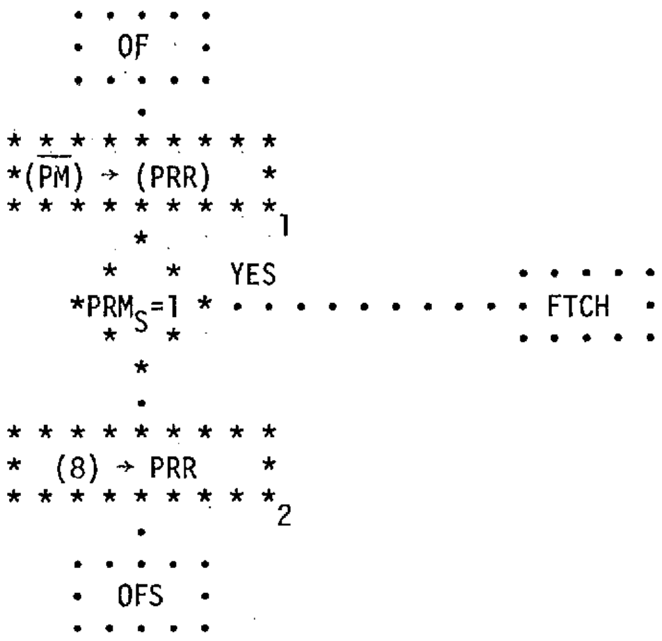


Figure 6.50. LPSB, LPSC, LPSD Microprograms



NOTE: SUBROUTINE OFS RETURNS TO FTCH

Figure 6.51. Overflow (OF) Microprogram

```
           . . . . .
           .  OFS  .
           . . . . .
               .
     * * * * * * * * * * *
     *    40 → (MAR)     *
     * * * * * * * * * * *1
               .
               .
     * * * * * * * * * * *
     * (N) RS4L (MQR)     *
     * * * * * * * * * * *2
               .
               .
     * * * * * * * * * * *
     *  (MQR) RS4A (SM)M *
     * * * * * * * * * * *3
               .
               .
     * * * * * * * * * * *
     *(PRR)+(SM) → (PRR) *
     *    MEM WRITE      *
     *     1 → (ER)      *
     * * * * * * * * * * *4
               .
               .
     * * * * * * * * * * *
     * (MAR)+4 → (T0)    *
     * (ER)+1 → (ER)     *
     * * * * * * * * * * *5
               .
               .
     * * * * * * * * * * *
     *(PM) RS4L (MQR)    *
     *  (ER) → (ER)      *
     * * * * * * * * * * *6
               .
               .
     * * * * * * * * * * *
     * (MQR) RS4A (MQR)  *
     *  (CC) → (MAR)     *
     *  (ER) → (ER)      *
     * * * * * * * * * * *7
               .
               .
           . . . . .
           . OFSA  .
           . . . . .
```

Figure 6.52. OFS Microprogram

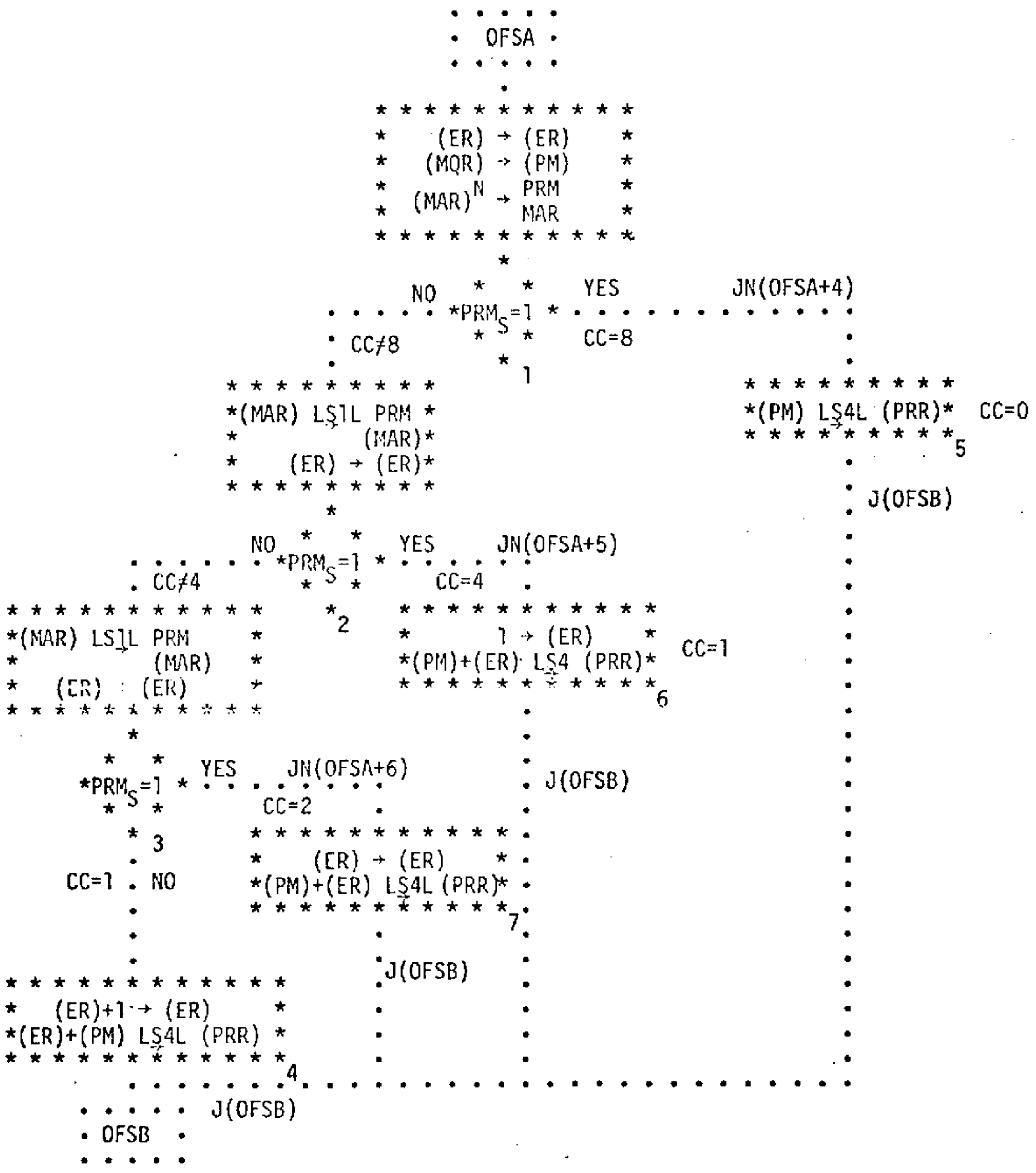


Figure 6.53. OFSA Microprogram

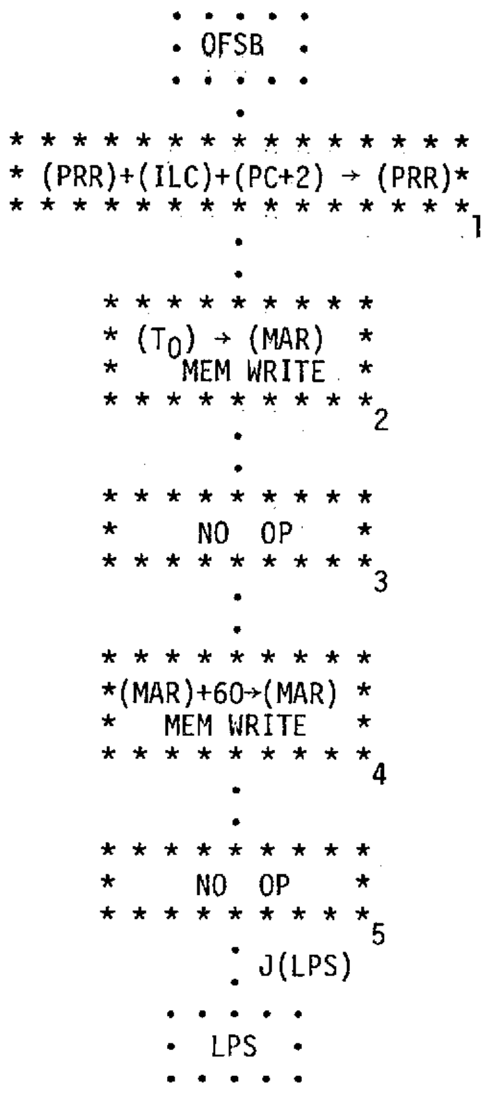


Figure 6.54. OFSB Microprogram

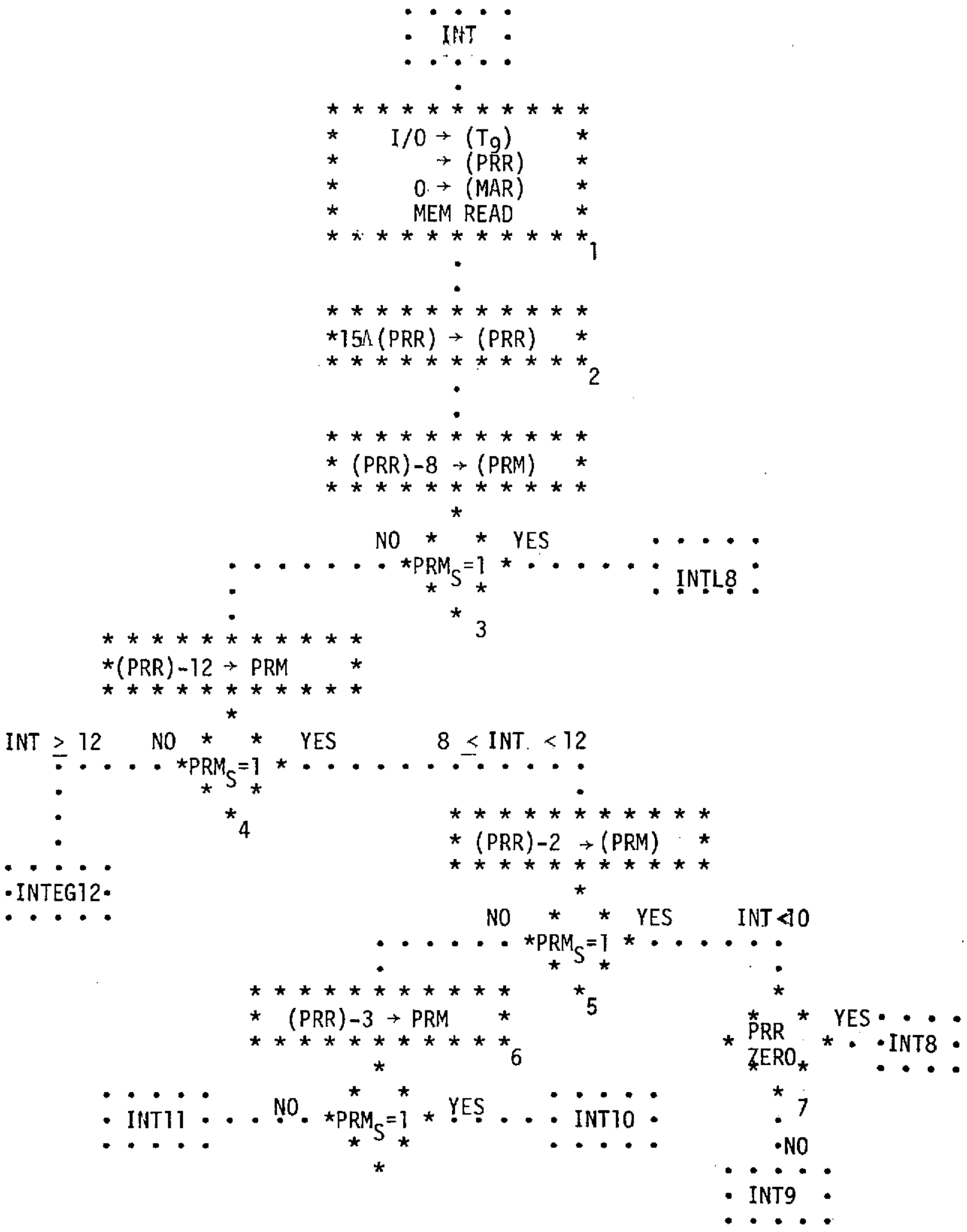


Figure 6.55. Interrupt (INT) Microprogram

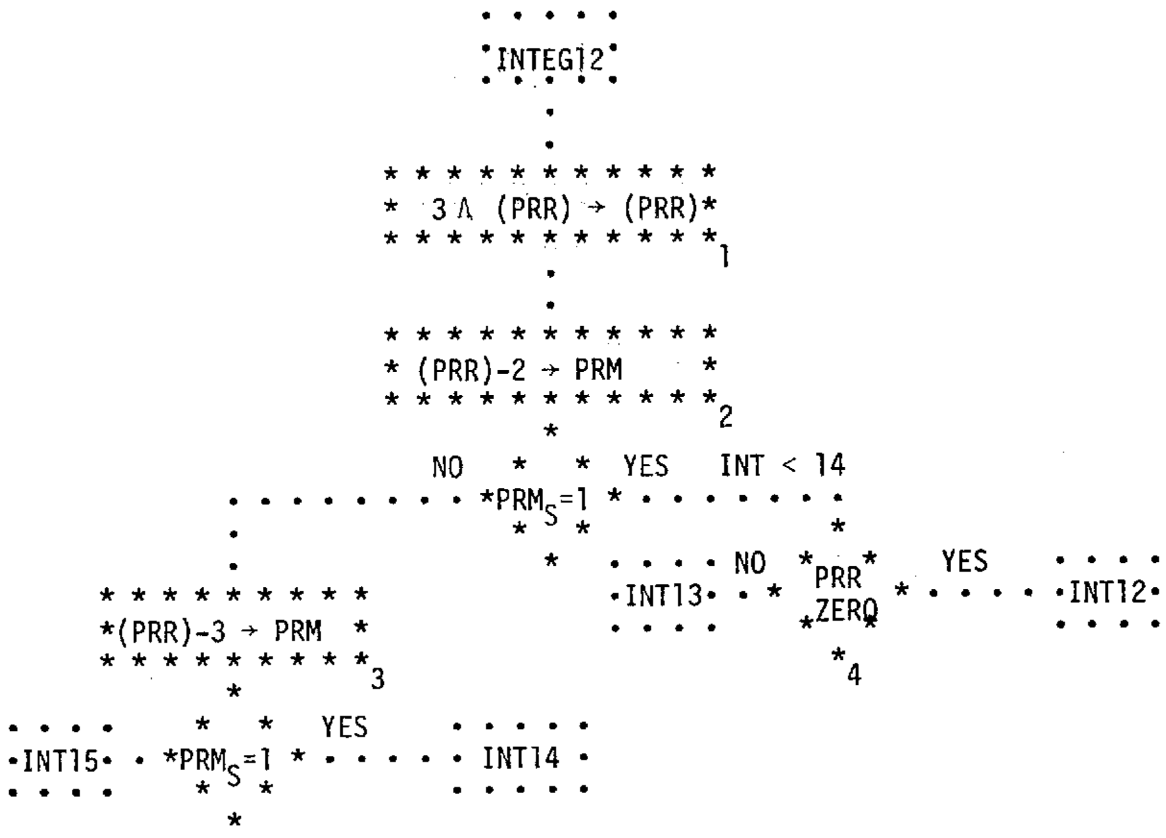


Figure 6.56. INTEG12 Microprogram

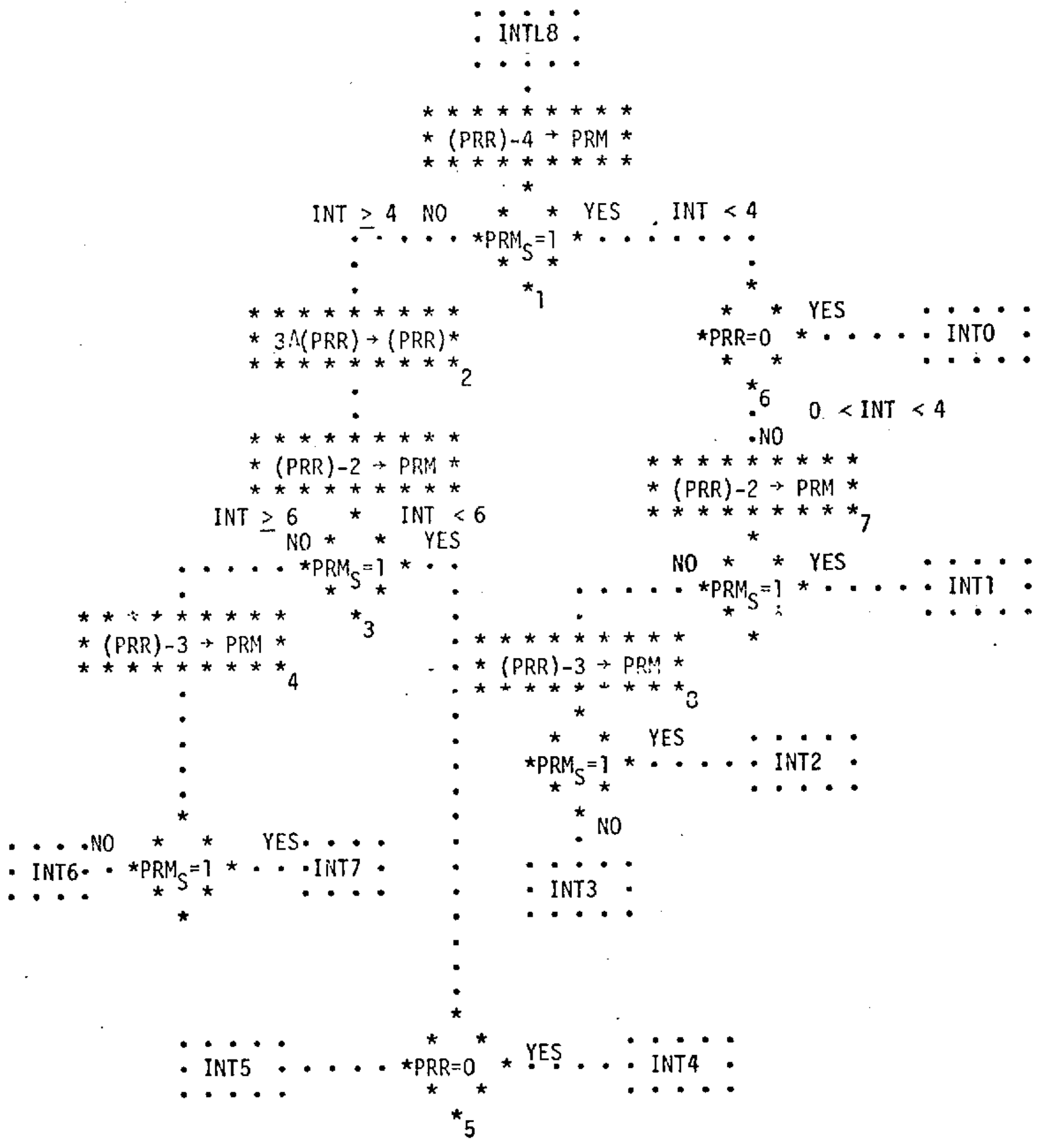


Figure 6.57. INTL8 Microprogram

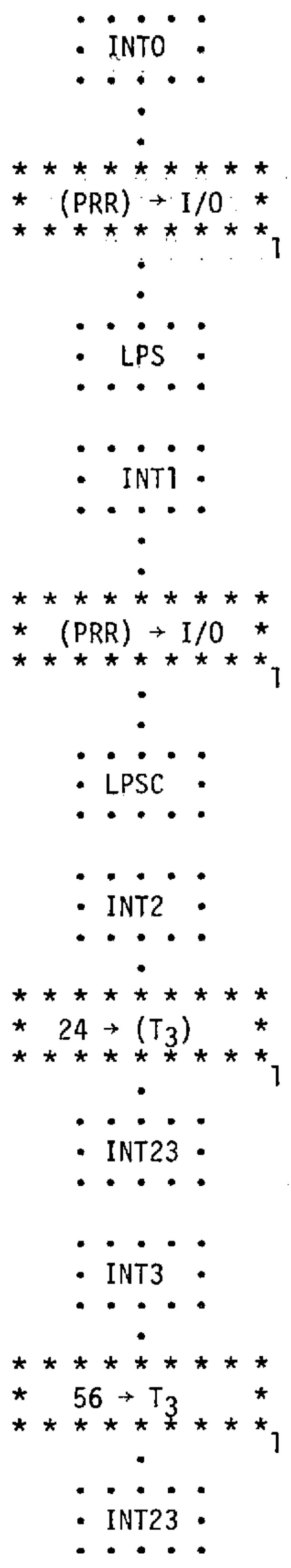


Figure 6.58. INT0, INT1, INT2, INT3 Microprograms

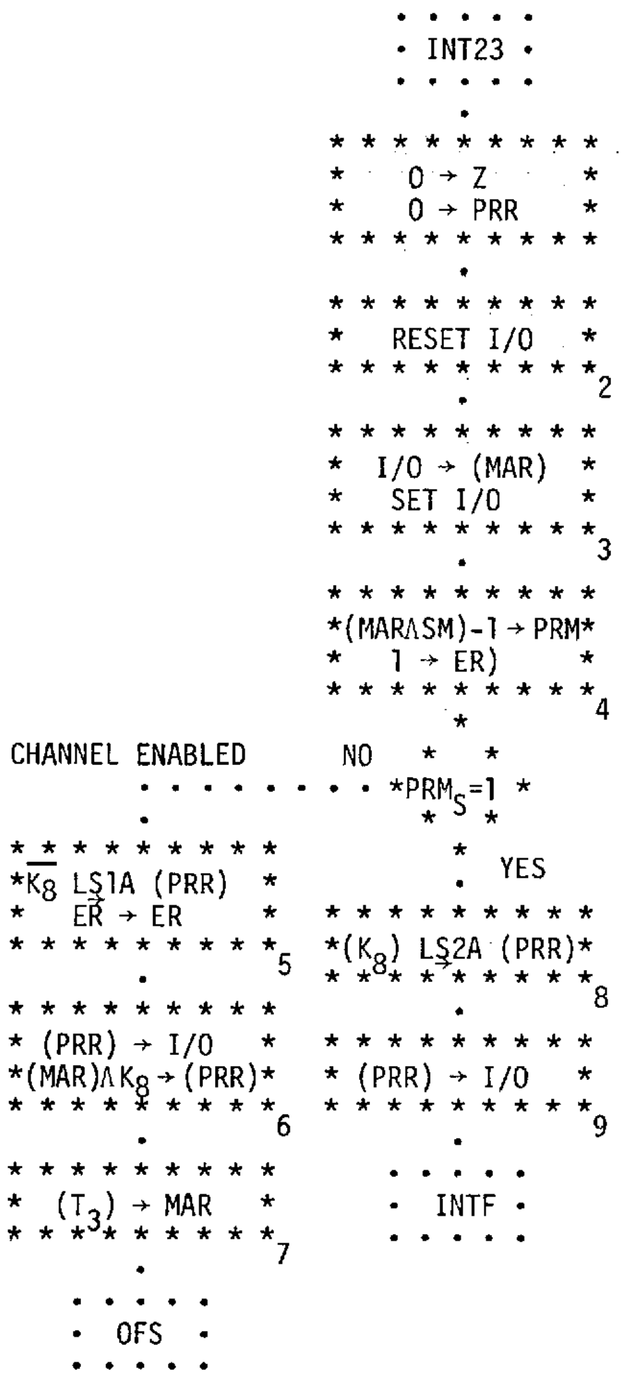


Figure 6.59. INT23 Microprogram

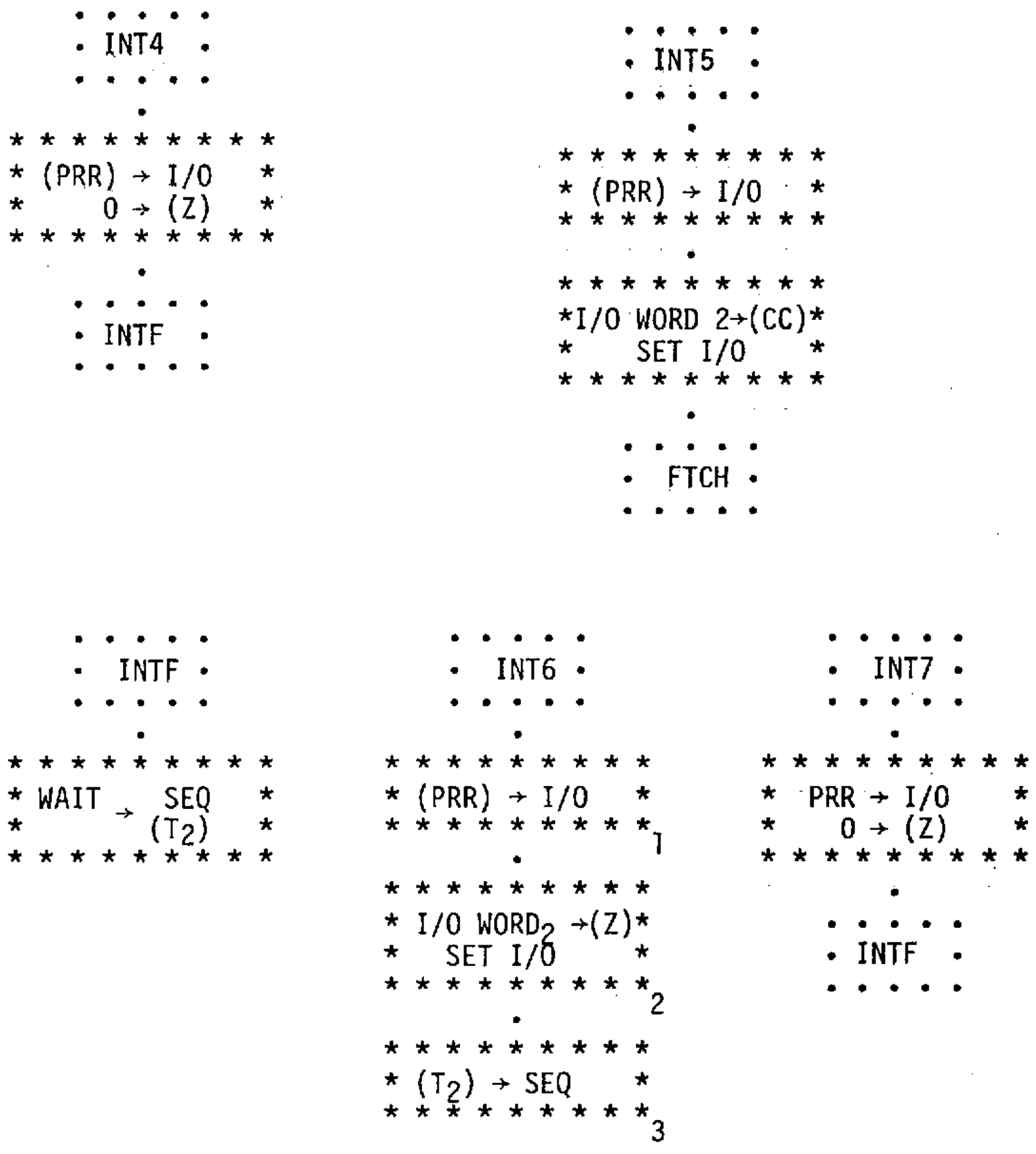


Figure 6.60. INT4, INT5, INT6, INT7, INTF Microprograms

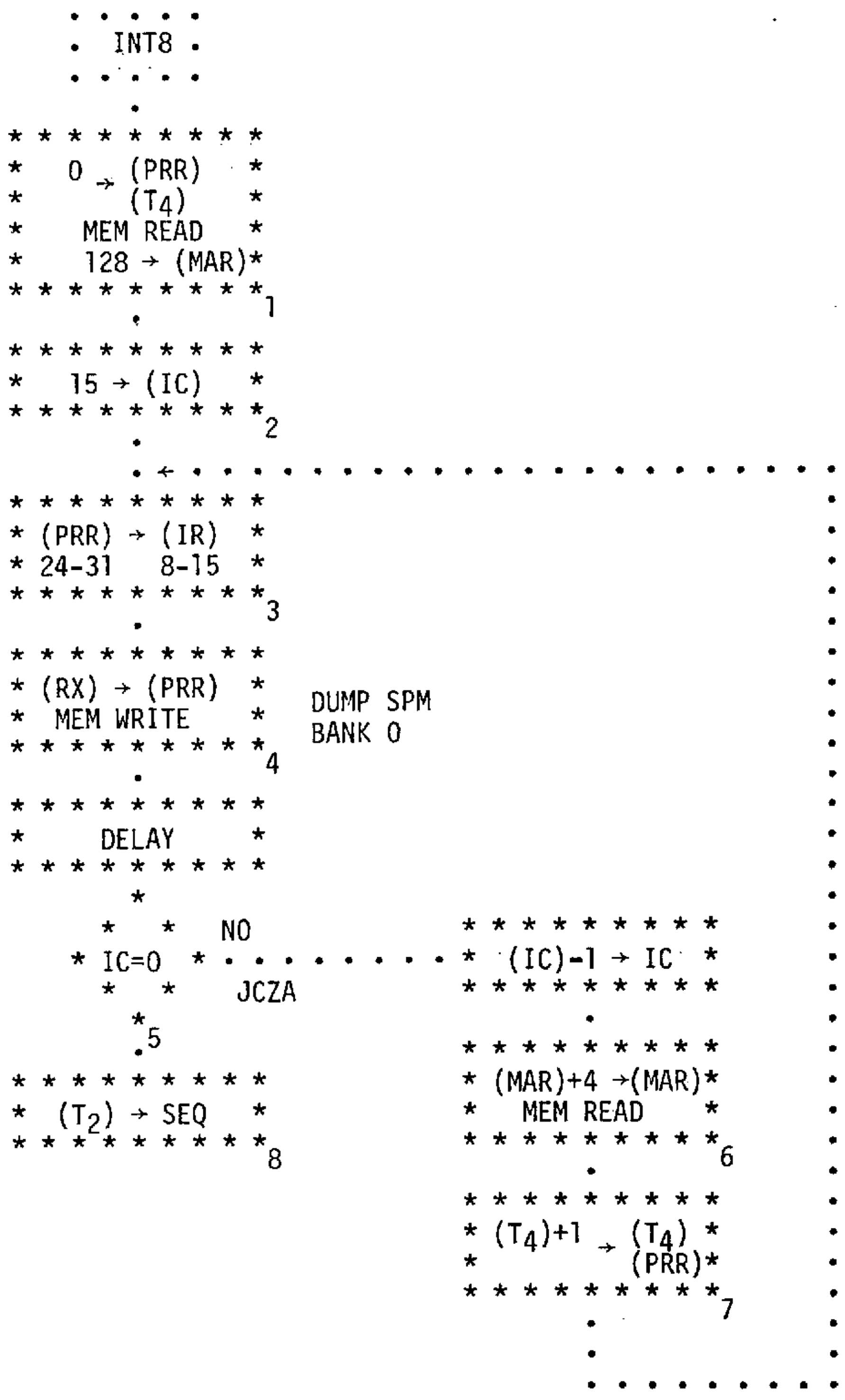


Figure 6.61. INT8 Microprogram

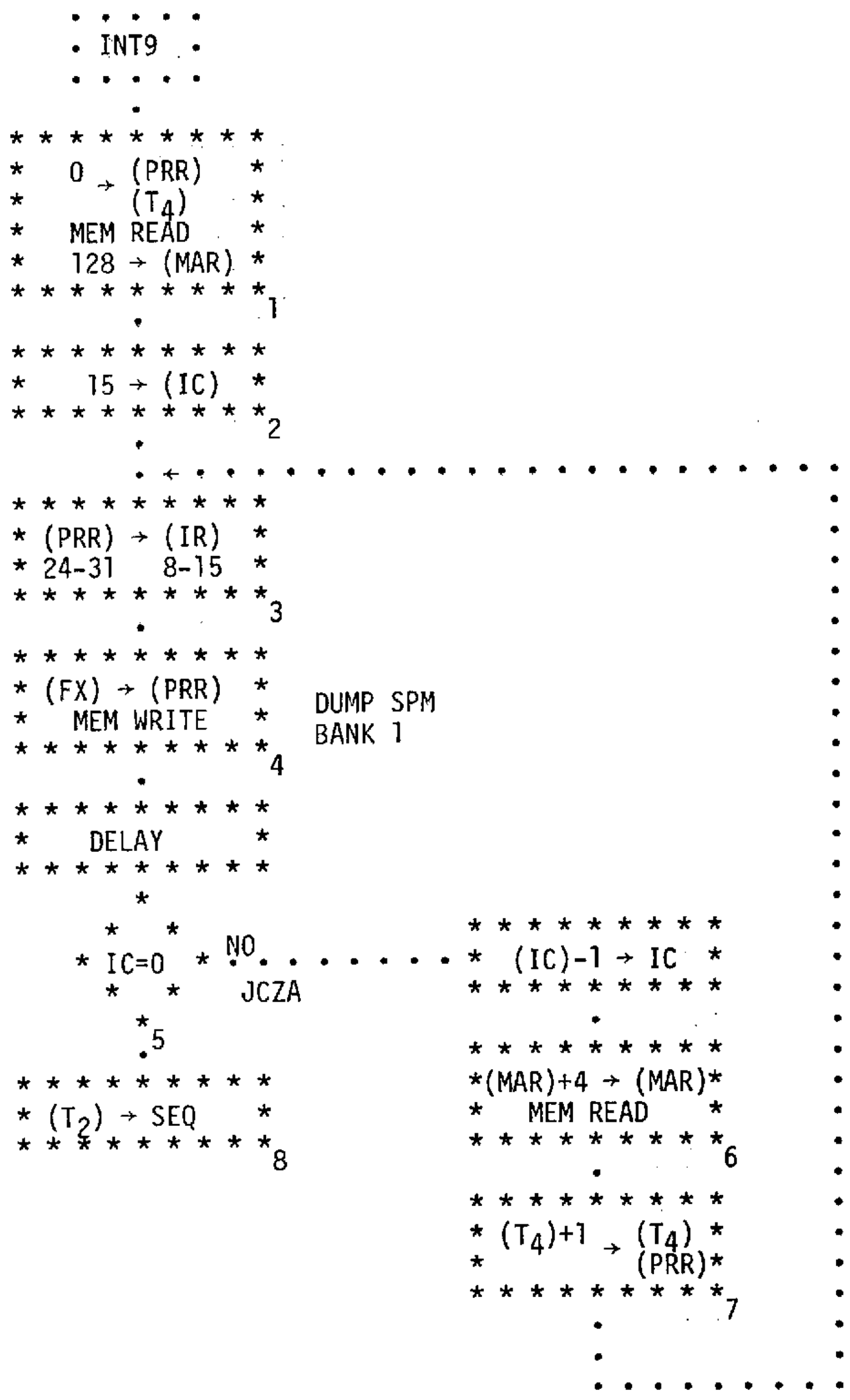


Figure 6.62. INT9 Microprogram

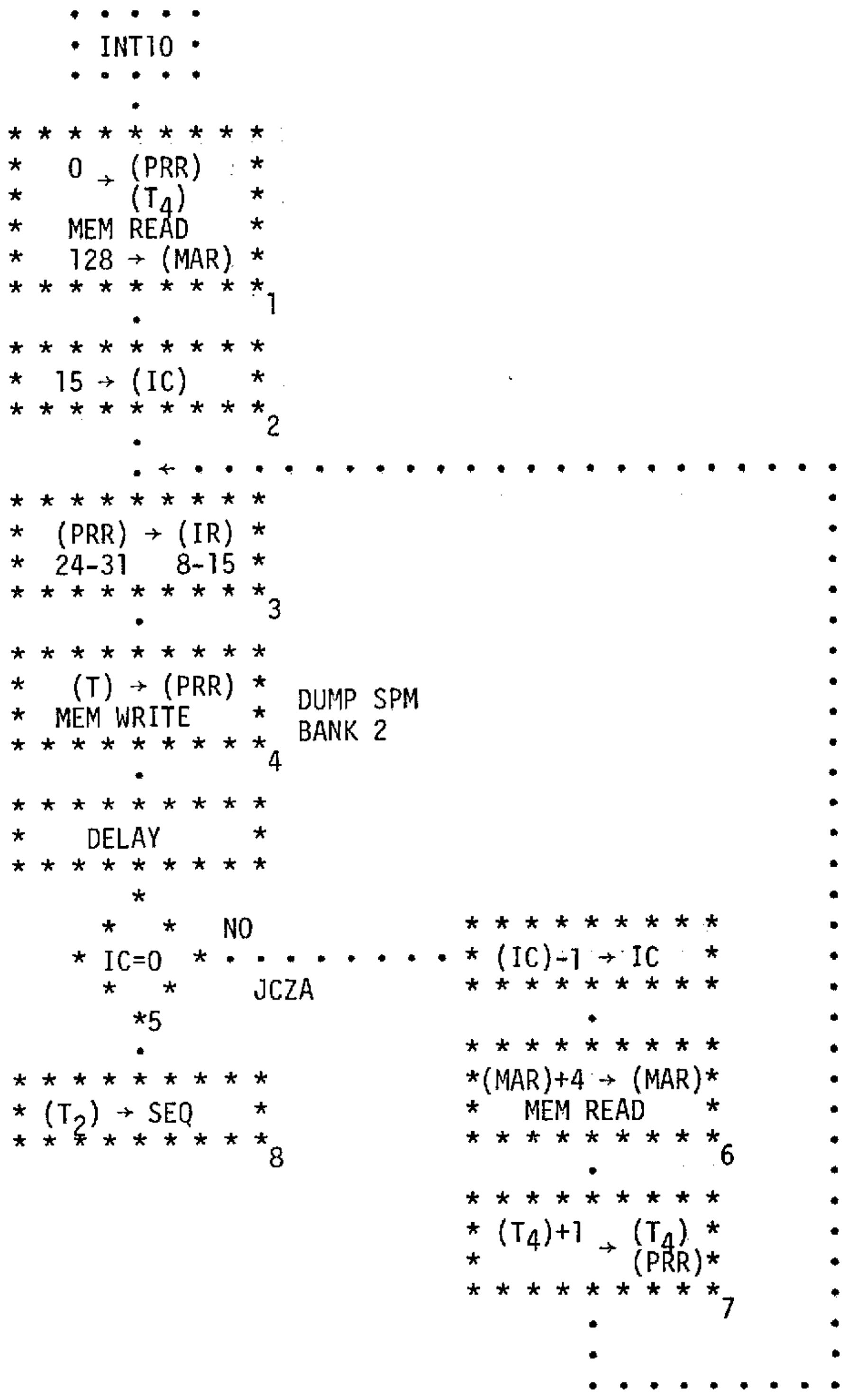


Figure 6.63. INT10 Microprogram

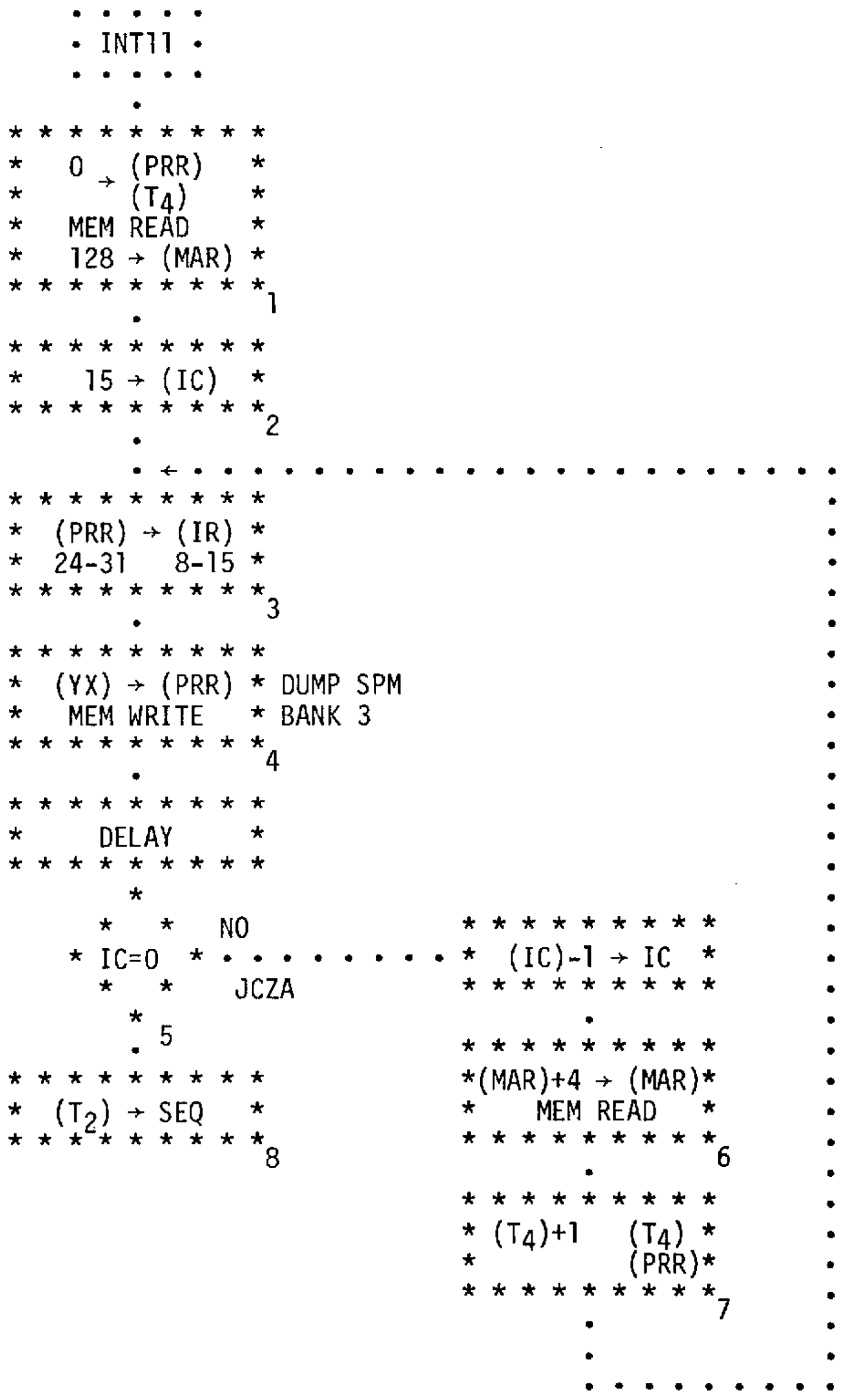


Figure 6.64. INT11 Microprogram

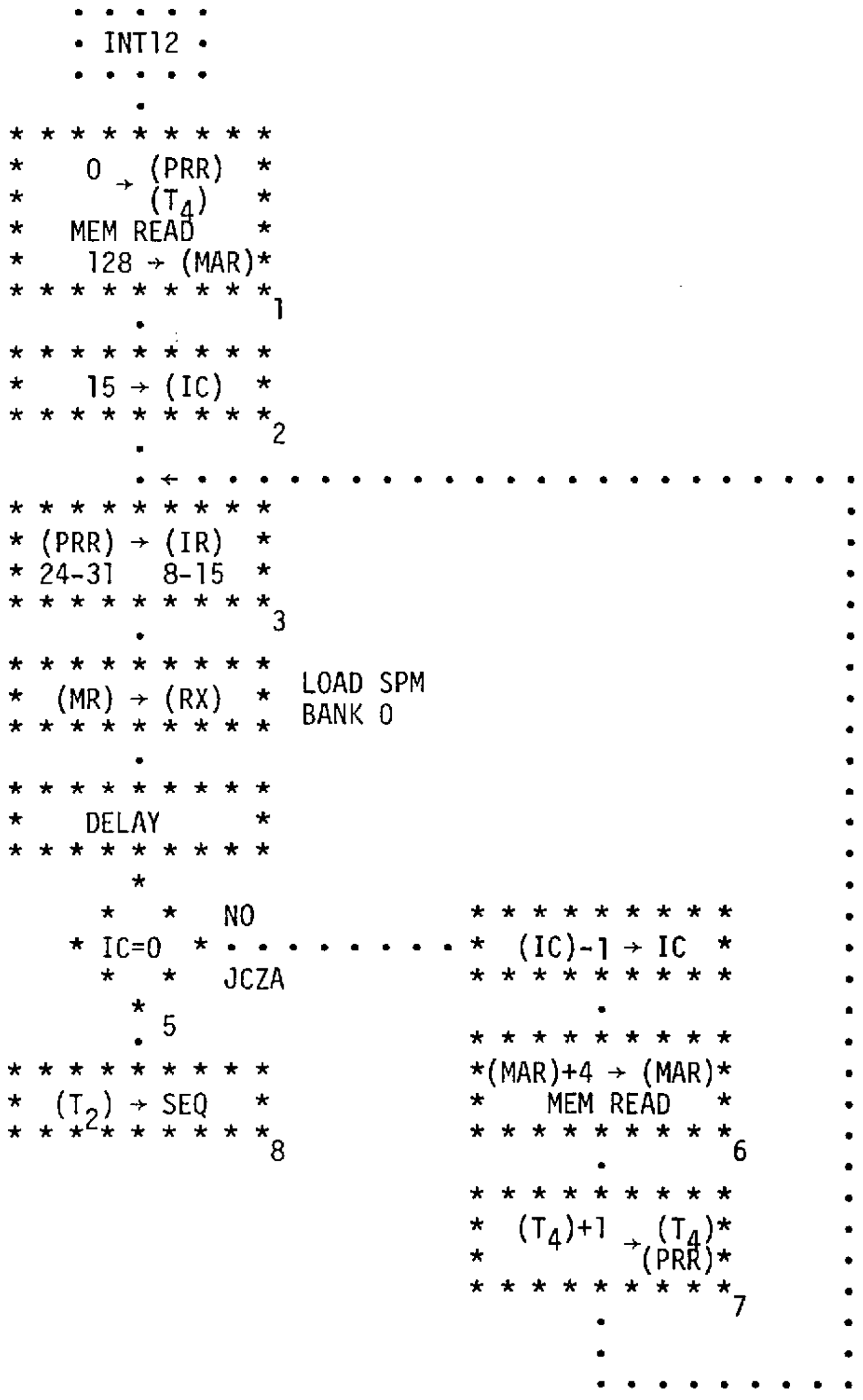


Figure 6.65. INT12 Microprogram

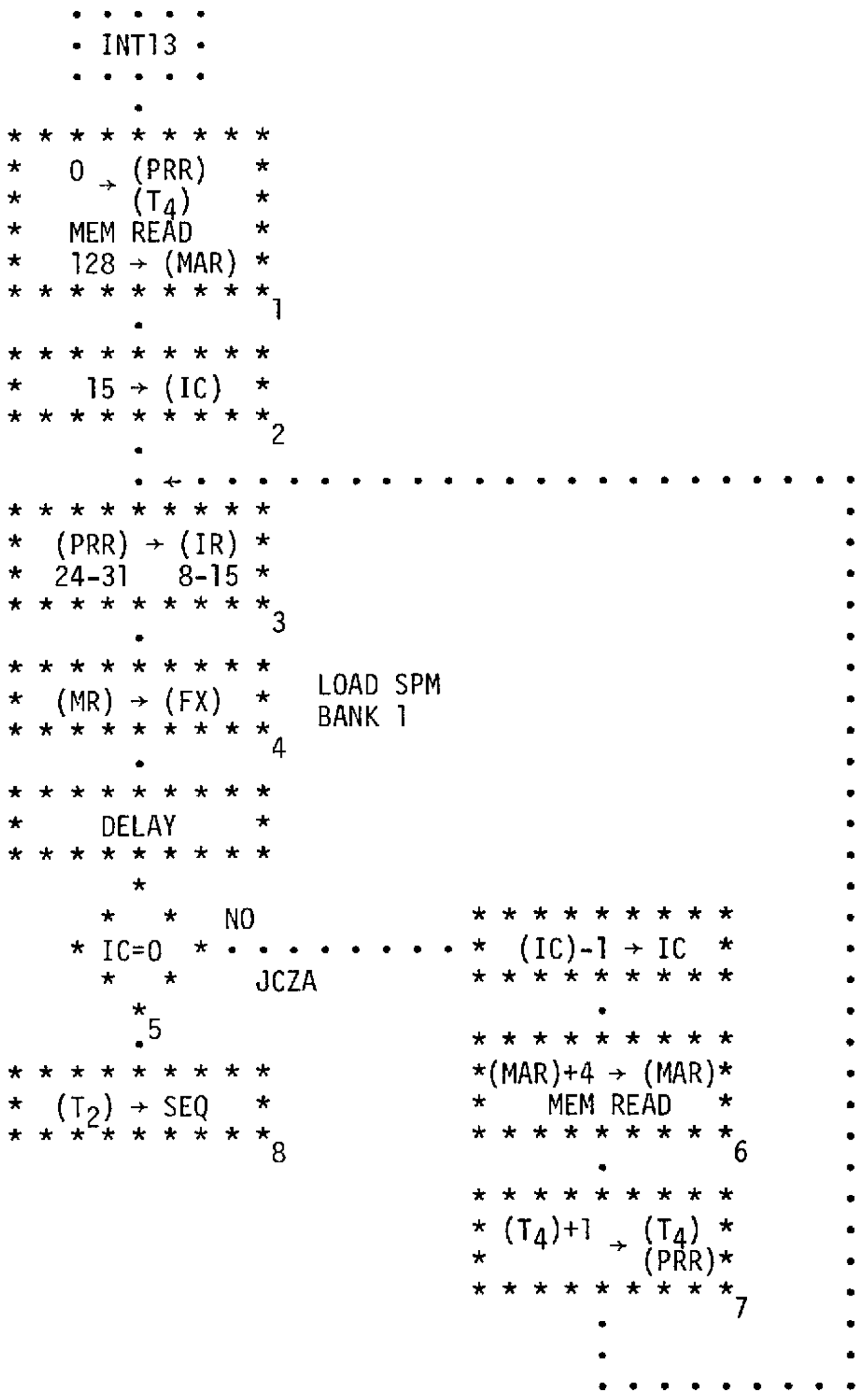


Figure 6.66. INT13 Microprogram

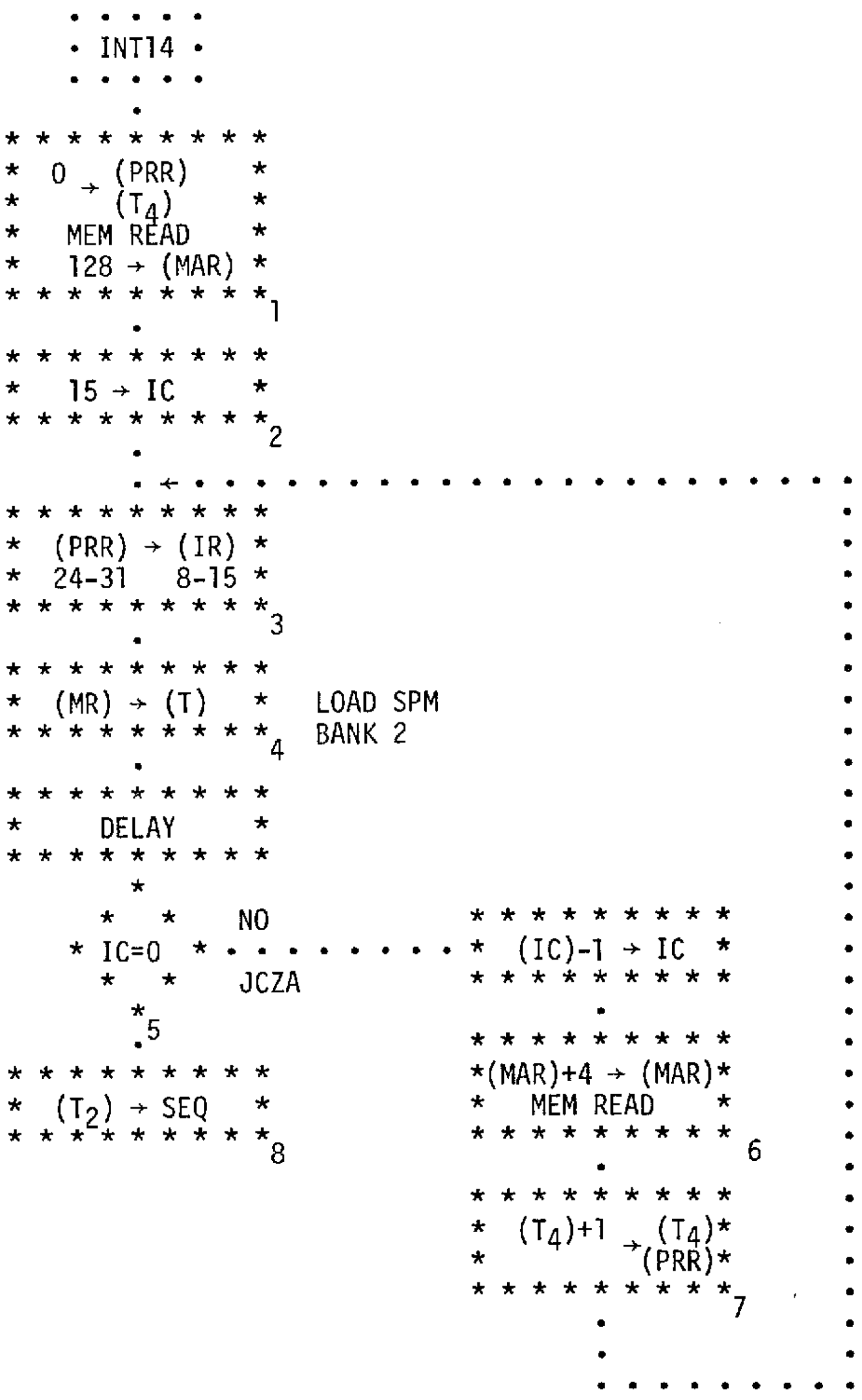


Figure 6.67. INT14 Microprogram

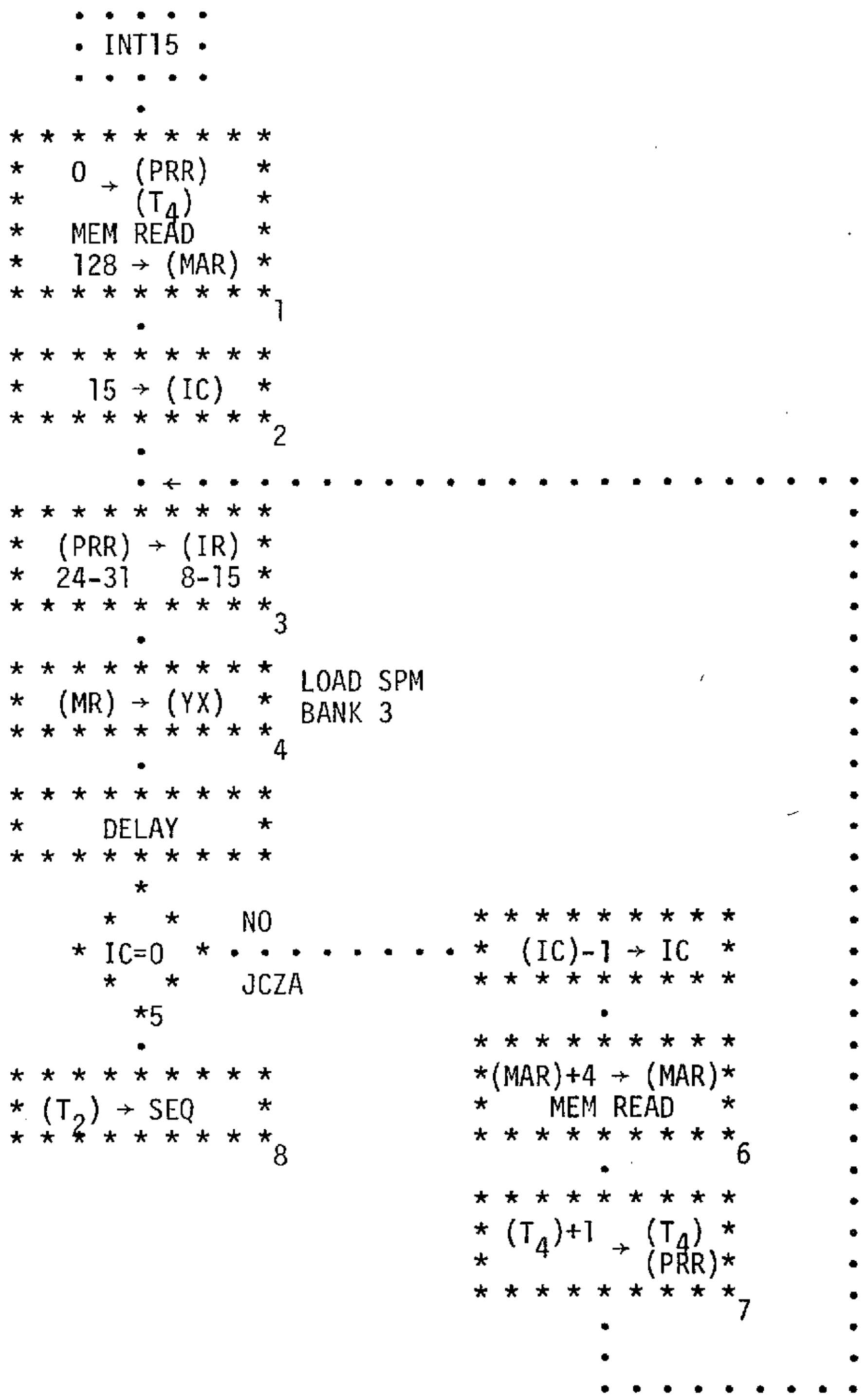


Figure 6.68. INT15 Microprogram

CHAPTER VII

COMPARISON

7.1 Conclusions

A comparison of the structured and nonstructured microprograms used in the emulation of the IBM System/360 by the SUMC BB brought to light the following conclusions:

1. Structured instruction microprograms were essentially as efficient as the corresponding nonstructured microprograms. That is, execution time and microprogram size remained essentially the same.

2. The implementation of structured housekeeping microprograms (Fetch, Overflow, Interrupt, etc.) was considerably more difficult. In general, these structured microprograms were slower and occupied more control memory locations than the nonstructured ones. For instance, in the Interrupt microprograms the lack of a CASE (COMPUTED GO TO) statement primitive forces the microprogrammer to use 15 decision blocks rather than one to decide which interrupt to process. This represents up to a 30 percent reduction in speed (execution speed varies with each interrupt processed), and a 16 percent increase in the size of the microcode memory portion which handles interrupts. Furthermore, note that INT8, INT9, INT10, INT11, INT12, INT13, INT14, and INT15 are identical except for Step 4. These microprograms could be combined by

using the forbidden CASE primitive. Because of this restriction, the size of the structured Interrupt microprogram subset is 200 percent greater than the size of the nonstructured Interrupt microprogram subset. There were no significant changes in speed or size for Fetch and Overflow microprograms. The author feels the housekeeping area is the crucial one when implementing structured microprograms because real time response limitations or control memory size restrictions may not permit the microprogrammer to use structured primitives wholly in housekeeping microprograms.

3. Structured RR, RX, RS and SI format microprograms were easily implemented in the SUMC;speed and microcode memory size were not sacrificed.

4. Structured SS format microprograms remained as fast as the nonstructured ones. However, it was not possible to share microcode memory locations, within these microprograms, as easily as it was with the nonstructured microprograms. Consequently, the microcode memory size was increased by 15 percent for the structured set.

5. Structured microprograms were better organized and thus should be more readily and quickly understood by users.

One must not forget that microprogramming is machine dependent. Thus, these conclusions apply only to the SUMC BB.

7.2 Further Research

The study of the branching architecture required for ease of implementation of structured microprograms is not complete. The branching architecture of other microprogrammable machines and of the newly developed microprocessors should be studied.

REFERENCES

1. Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System", Communications of the ACM, May 1962.

2. Gimenez, C. R., and Williams, C. A., SUMC BB/360 Microprogrammers Reference Manual, Sperry Rand, 1974.

3. Glushkov, V., "Automata Theory and Formal Microprogram Transformations", Kibernetika, Vol. 1, No. 5, 1965.

4. Glushkov, V., "Automata Theory and Structural Design Problems of Digital Machines", Kibernetika, Vol. 1, No. 1, 1965.

5. Glushkov, V., "Minimization of Microprograms and Algorithm Schemes", Kibernetika, Vol. 2, No. 5, 1966.

6. IBM System/360 Principles of Operation, IBM Systems Development Division, Publication No. GA22-6821-8, May 1970.

7. Intel 8080 Single Chip Eight Bit Parallel Central Processor Unit, Intel, April 1974.

8. Ito, T., "A Theory of Formal Microprograms", Sigmicro Newsletter, Vol. 4, No. 1, April 1973.

9. Ito, T., "On Proving the Correctness of Programs", Fourth Princeton Conference on Information Sciences and Systems, 1970.

10. Ito, T., "Some Formal Properties of a Class of Program Schemata", IEEE Symposium on Switching and Automata Theory, 1968.

11. Jones, L. H., "The Role of Instruction Sequencing in Structured Microprogramming", Sigmicro Newsletter, Vol. 4, No. 3, October 1973.

12. National Semiconductor IMP 8 Programming Manual, National Semiconductor, December 1973.

13. Navy Electronics Laboratory, "Microprocessor Developments for Project 2175", August 1973.

14. Mills, Harlan, "Mathematical Foundations for Structured Programming", IBM Research Report FSC 72-6012, February 1972.

15. G. Noguez, "A Standardized Microprogram Sequencing Control with a Push Down Storage", Fifth Annual Workshop on Microprogramming, University of Illinois, 1972, (Preprints).

REFERENCES (Concluded)

16. Reigel, E. W., Faber, U., and Fisher, D. A., "The Interpreter -- A Microprogrammable Building Block System", AFIPS Conference Proceedings, Vol. 40, SJCC 1972, pp. 705-723.

17. Roberts, J. D., Jr., Ihnat, J., and Smith, W. R., Jr., "Microprogrammed Control Unit (MCU) Programming Reference Manual", Sigmicro Newsletter, Vol. 3, No. 3, October 1972.

18. Space Ultrareliable Modular Computer (SUMC), S&E-ASTR-C-005, 1972.

19. SUMC Microinstruction Assembler, S&E-ASTR-C-002, Rev. A, October 1972.

20. SUMC 360 Microcode and Microprogram Flowcharts, Sperry Rand, December 1972.

21. SUMC 360 Microcode and Microprogram Flowcharts, Sperry Rand, December 1973.

22. Tenny, Ted, "Structured Programming in Fortran", Datamation, July 1974.

23. Yourdon, E., "A Brief Look at Structured Programming and TOP DOWN Program Design", Modern Data, June 1974.

APPENDIX A

NOTATION

Symbol	Meaning
$=:$	is loaded into
$:$	corresponds to or is related to
$\rightleftarrows$	corresponds to
$A \rightarrow B$	mapping of A into B
$\rightarrow$	loaded into
$\bullet$	concatenation
$< a \vee b >_p$	if p then a; if not p then b
$[b]_p$	while p do b
$(\quad)$	contents of
$\vee$	logical OR
$\veebar$	exclusive OR
$\wedge$	logical AND
$\sim$	logical NOT
$\overline{}$	logical NOT
$<$	less than
$\nless$	not less than
$=$	equal to
$\neq$	not equal to

APPENDIX B

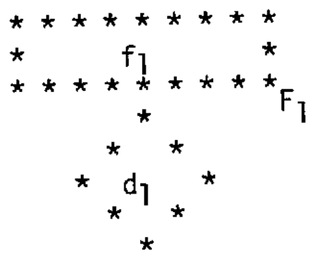
FLOWCHART CONVENTIONS

```
.   .   .   .   .
.               .          Entry or exit point.
.   .   .   .   .


*   *   *   *   *
*               *          Microinstruction cycle.
*   *   *   *   *


        *
    *       *
*               *          Decision block.
    *       *
        *


lower case letter      =>  a function of process
within a micro-
instruction cycle
or decision block


upper case letter      =>  MROM location
on lower right hand
corner of a micro-
instruction cycle
or decision block
```

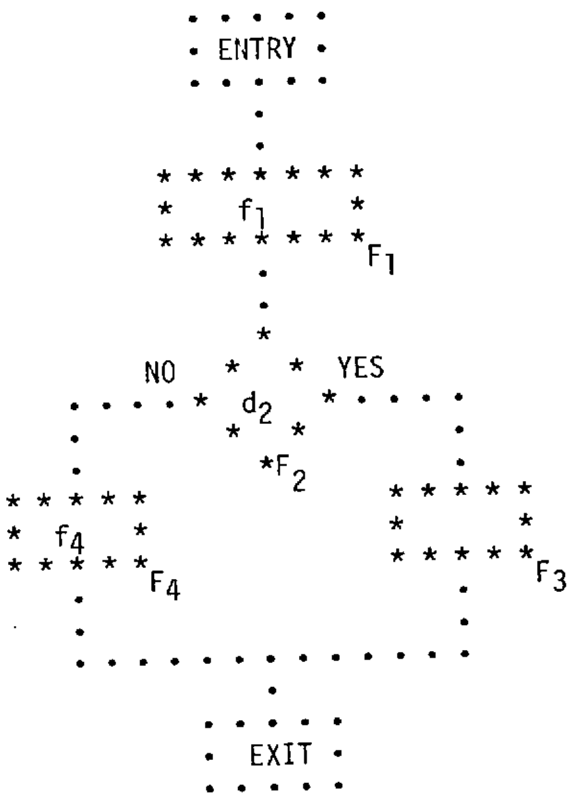
EXAMPLES

Example

Explanation



Microinstruction cycle with decision block. F_1 refers to MROM location; f_1 to some function performed during F_1; d_1 refers to some decision performed during F_1.



Microprogram begins at microinstruction cycle F_1 is followed by decision block F_2. d_2 refers to some decision performed during F_2; if the condition checked for during F_2 is true, a No-Operation cycle F_3 is performed prior to going to EXIT; if the condition checked for during F_2 is false; f_4 is performed during microinstruction cycle F_4 prior to EXIT.

APPENDIX B

FLOWCHART SYMBOLS

( )	Contents of
$\wedge$	Logical "AND" operation
$\vee$	Logical "OR" operation
$\veebar$	Logical "EXCLUSIVE OR" operation
+	Addition
-	Subtraction
$\overline{(\quad)}$	One's complement
-( )	Negation (Two's complement)
$\rightarrow$	Data transfer or flow
N	(Used either as a subscript or superscript). Normalized
LS4A	Left shift four arithmetic bits
LS2A	Left shift two arithmetic bits
LS1A	Left shift one arithmetic bits
LS4L	Left shift four logical bits
LS2L	Left shift two logical bits
LS1L	Left shift one logical bits
RS4A	Right shift four arithmetic bits
RS1A	Right shift one arithmetic bits
RS4L	Right shift four logical bits
RS1L	Right shift one logical bits

APPENDIX B

FLOWCHART ABBREVIATIONS

AR	Accumulator Register
ALU	Arithmetic Logic Unit
B	Base Register
CC	Condition Code
CCR	Condition Code Register
D	Displacement Field
DEX	Derived Exponent
EALU	Exponent Arithmetic Logic Unit
ER	Exponent Register
IAROM	Instruction Address Read-Only-Memory
IC	Iteration Counter
IR	Instruction Register
K	SPM Mask Registers (see Appendix C)
M	Program Mask
MAR	Memory Address Register
MAM	Memory Address Multiplexer
MQR	Multiplier/Quotient Register
MR	Memory Register
MROM	Microcode Read-Only-Memory
N	Interrupt Status Register
OF	Overflow
PC	Program Counter

FLOWCHART ABBREVIATIONS
(Concluded)

PRM	Product Remainder Multiplexer
PRR	Product Remainder Register
R	Register
ROM	Microcode Read-Only-Memory
S	System Mask
SPM	Scratch Pad Memory
T	Temporary Register
X	Register

APPENDIX B

FLOWCHART SUBSCRIPTS

M	Mantissa
N	Normalized • Also used as a superscript
OF	Overflow
S	Sign bit
SHW	Sign Extended Halfword
V	Used with Temporary register Tv
0-F	Hexadecimal 0-15
+1	Register specified by selected IR field +1

APPENDIX C

SPM MASK REGISTERS

K_0	=	00000001
K_1	=	00FFFFFF
K_2	=	FF00FFFF
K_3	=	F0FFFFFF
K_4	=	FFF0FFFF
K_5	=	0FFFFFFF
K_6	=	FF0FFFFF
K_7	=	FFFFFF00
K_8	=	0000FFFF
K_9	=	01000000
K_A	=	00FF0000
K_B	=	F0000000
K_C	=	00F00000
K_D	=	000FFFFF
K_E	=	0000F000
K_F	=	00FF0000

APPENDIX D

EXTERNAL INTERRUPTS

Name	Function
INT0	Power Up
INT1	Release CPU
INT2	External Interrupt
INT3	Channel 1
INT4	System Reset
INT5	Set Condition Code
INT6	Set Channel Busy
INT7	Initial Program Load
INT8	Store SPM Bank 0
INT9	Store SPM Bank 1
INT10	Store SPM Bank 2
INT11	Store SPM Bank 3
INT12	Load SPM Bank 0
INT13	Load SPM Bank 1
INT14	Load SPM Bank 2
INT15	Load SPM Bank 3