
Masters Theses

Student Theses and Dissertations

Spring 1987

Design of a meta-assembler for microprogramming and a survey of microprogram optimization techniques

Rahul Saxena

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses



Part of the [Electrical and Computer Engineering Commons](#)

Department:

Recommended Citation

Saxena, Rahul, "Design of a meta-assembler for microprogramming and a survey of microprogram optimization techniques" (1987). *Masters Theses*. 488.

https://scholarsmine.mst.edu/masters_theses/488

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

DESIGN OF A META-ASSEMBLER FOR MICROPROGRAMMING
AND A SURVEY OF MICROPROGRAM OPTIMIZATION TECHNIQUES

BY



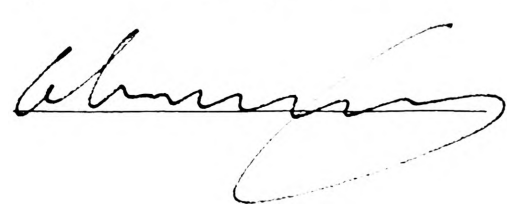
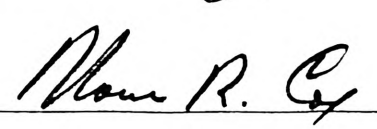
RAHUL SAXENA, 1959-

A THESIS

Presented to the Faculty of the Graduate School of the
UNIVERSITY OF MISSOURI - ROLLA
in Partial Fulfillment of the Requirements for the Degree
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

1987

Approved by

 (Advisor) 
 

ABSTRACT

The purpose of this thesis is to develop and implement a meta-assembler for microprogramming. Different methods for optimizing microprogram execution and storage are also investigated.

The meta-assembler is of an adaptive type. It allows complete flexibility in the definition of the target machine op-codes and microinstruction field formats. The assembly process consists of two main phases. In the first phase the assembler builds a description of the target machine in terms of its microinstruction field format definitions. In the second phase the source program is assembled into object microcode. The assembler is written in the language Pascal. The assembler is fast, efficient and the syntax allows easy development of source code.

Different techniques for optimization of execution time and storage of microprograms are investigated, these include the description of the high level language SIMPL which allows high level microprogramming and generates optimized horizontal microcode.

ACKNOWLEDGEMENT

I wish to express my sincere gratitude to Dr. Darrow F. Dawson for his continued guidance and advice throughout the preparation of this thesis. I also thank Mr Richard M. Strandberg, Dr Norman Cox and Dr Alan Cummings for their editorial assistance.

TABLE OF CONTENTS

	Page
ABSTRACT	ii
ACKNOWLEDGEMENT	iii
LIST OF ILLUSTRATIONS	vi
I. INTRODUCTION	1
II. A MICROPROGRAM META-ASSEMBLER	4
A. MICROPROGRAM ASSEMBLER DESIGN CONSIDERATIONS	6
B. DESIGN OF A MICROPROGRAM META-ASSEMBLER	8
1. The Definition Process	10
a. Data Structure For Definition Table	10
b. Construction Of The Definition Table	12
2. ASSEMBLY	15
a. Procedure FIRSTPASS	15
b. Procedure SECONDPASS	19
c. Function FINDBITS	24
d. Procedure ERROR	24
e. Procedure INSERTMICROFIELD	24
III. OPTIMIZATION FOR HORIZONTAL MICROPROGRAMMING	26
A. GLOBAL OPTIMIZATION	27
B. LOCAL OPTIMIZATION	28
1. Detection Of Program Parallelism: Single Assignment Approach	28
2. Single Identity Microprogramming Language (SIMPL)	30

3. Constrained Statements	33
C. SIMPL COMPILATION	33
1. Syntatic Analysis	36
2. Semantic Analysis	36
3. Concurrency and Timing Analysis	36
4. Microoperation Timing Optimization	39
IV. MINIMIZATION OF ROM WIDTH	49
A. GENERAL CONSIDERATIONS FOR ROM WIDTH REDUCTION	49
1. Coding Efficiency	49
2. Function Extraction And Bit Packing	52
3. Conflict Groups	54
B. ALGORITHMS	59
1. A Conflict Partition Algorithm	59
2. A Compatibility Class Algorithm	64
V. CONCLUSION	70
BIBLIOGRAPHY	72
VITA	73

LIST OF ILLUSTRATIONS

Figure		Page
1	Types of Assemblers	5
2	Control Flow for the Meta-assembler	9
3	Data Structure for the Definition Table	11
4	Record for Storing Field Attributes	13
5	Flow Chart for Procedure BUILDDEFTABLE	16
6	Flow Chart for Procedure FIRSTPASS	18
7	Symbol Table	21
8	Flow Chart for Procedure SECONDPASS	22
9	Flow Chart for Function FINDBITS	25
10	Variable Dependency of Statements	31
11	SIMPL Compilation Procedure	34
12	SIMPL Microprogram for 64-bit Floating Point Multiplication	35
13	Sequential Semantic Code Generated by Semantic Analysis	37
14	Earliest Execution Timing of Concurrent Microoperations	40
15	Latest Execution Timing of Concurrent Microoperations	41
16	Flow Chart for Resolving Resource Conflict in Critical Microoperations	43
17	Flow Chart for Finding the Minimum Sequence	45
18	Concurrency Achieved in the Object Microprogram	48
19	Coding efficiency Chart	51
20	ROM for Coding Efficiency Example	53
21	ROM with Internal Boolean Groups	55
22	ROM with External Boolean Groups	56

23 ROM with no Encoding 57

24 Conflict Diagram 61

25 Flow Chart for Conflict Partition Algorithm 62

26 Encoded ROM 65

27 ROM for example on Compatibility Class Algorithm . . 67

I. INTRODUCTION

Microprogramming was introduced by Wilkes in 1951 as a systematic alternative to the usual somewhat ad hoc procedure used for designing the control section of a digital computer. Though initially microprogramming was defined as "the technique of designing the control circuit of a digital computer to formally interpret and execute a given set of machine operations as an equivalent set of sequences of microoperations." Today, microprogramming is used in a wide variety of applications for example emulation, support for operating systems, graphics, communication controllers etc.

An objective of this thesis was to develop a meta-assembler for microprogramming. Microprogrammable machines can be configured to emulate different target machines. A general purpose software tool is required, which can be used for microprogramming different host machines and also can accommodate changes in the definition of the target machine. A meta-assembler can be used in different host microprogrammable machines and can be configured to generate microcode for different target machine architectures.

A meta-assembler for microprogramming was designed and implemented. The operation of the assembler is divided into two main steps. In the first step the assembler collects information about the target machine. In the second step the object code for this target machine is produced.

One question that needs to be addressed, is that of the use of high-level languages for microprogramming. There are some disadvantages to the use of high-level languages for microprogramming. High-level language programming is concerned with implementing an algorithm with little or no concern about the internal details of the machine in which it is to be run, whereas microprogramming consists principally of establishing a specific machine architecture in order to implement some class of algorithms efficiently. The relative inefficiency of object code produced by compilers defeats the purpose of defining an inner machine fast enough to be a base for further programming levels. The optimization techniques used in optimizing compilers can be applied for producing microcode, however, these techniques still do not match a good assembler programmer and furthermore they do not apply very well for horizontal machines where a new level of optimization (parallelism) can be put to work. Another disadvantage of high level microprogramming languages is that they have not yet achieved machine independence. One major attempt to develop a high-level language (SIMPL) to produce highly parallel and efficient object microprograms is described in this thesis.

Microprograms are optimized for execution time optimization and control store optimization. Compilers which produce programs in an algorithmic fashion (eg, Syntax directed compilers) must produce correctly functioning routines regardless of program sequences involved. Hence, inefficiencies are introduced to protect against special cases. Another cause of inefficient code might be that a program is written to ease its maintenance and understanding, while a different

organization of program subparts may be more desirable to reduce execution time. Another reason for inefficient code can be the use of library functions, if the program includes code drawn from the system library, the user is not familiar with the internal structure of the library routine. Therefore he may utilize only portions of the routine or perform checks that are also performed in the library code. Techniques for optimizing microcode execution time and control store requirements are investigated.

II. A MICROPROGRAM META-ASSEMBLER

Every computer system is accompanied by at least one program assembly language. An assembler is required to translate the assembly programs into machine language. As a consequence, at least one program assembler must be developed for every computer system. This need led researchers to look for ways to automate the construction of program assemblers. This led to the creation of a new software tool called meta-assemblers.

The objective of meta-assemblers is to facilitate the construction of program assemblers. Likewise, the development of microprogrammed computer systems led to microprogram assembly languages, microprogram assemblers and microprogram meta-assemblers. With the advent of microprogrammed computers, microprogram assembly tools analogous to program assembly tools started appearing. At the beginning these microprogram assembly tools were physically and functionally distinct from the program assembly tools. Later some meta-assemblers which could act both as program meta-assemblers and as microprogram meta-assembler were constructed.

Figure 1 shows how dedicated and meta-assemblers work. A meta-assembler is furnished with a "meta-language". The implementation of an assembler with the help of a meta-assembler is a matter of describing the desired assembler to the meta-assembler, using the meta-language. this description is called the assembler definition.

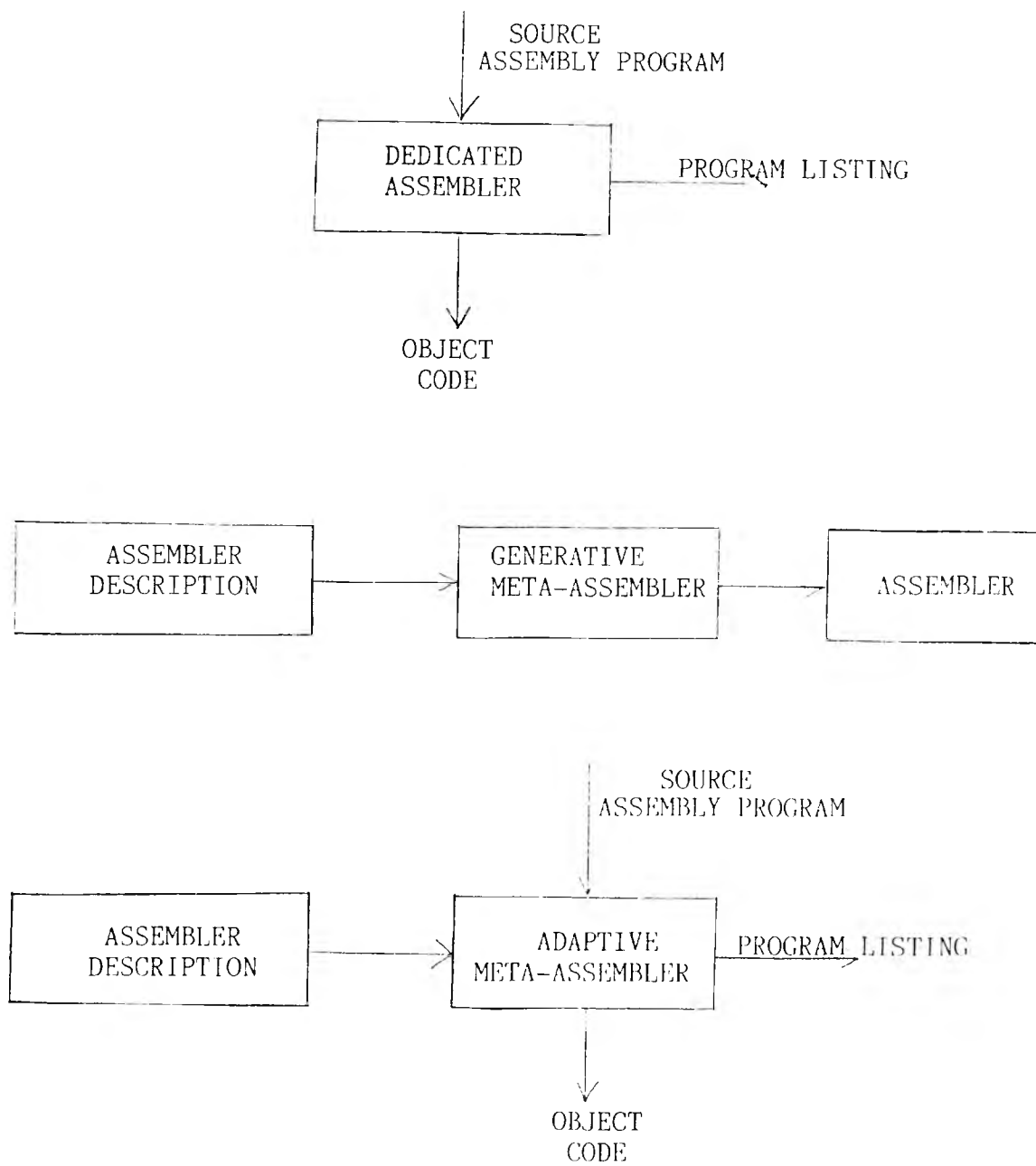


Figure 1. Types of Assemblers

With the help of the assembler definition, the meta-assembler can either generate the desired assembler or adapt itself to operate as the desired assembler. In the first case the meta-assembler is called a generative meta-assembler. In the second case the meta-assembler is called the adaptive meta-assembler.

A. MICROPROGRAM ASSEMBLER DESIGN CONSIDERATIONS

The following discussion is directed specifically towards designing microprogram assemblers for bit-slice computers, though it also applies for other microprogrammable computers.

Bit-slice computers consist of high-speed circuit chips, several of which are required to form a complete processor. These bit-slice devices must be connected together with careful attention to logical and timing detail. The sequence of internal register level operations necessary to complete a function step is usually controlled through a sequence of bit patterns retrieved from the microprogram memory by a sequence controller. The definitions of sequences and functions stored in this memory comprise a microprogram. Completion of a good design with bit-slice chips requires a detailed consideration of the hardware aspects like the connection and timing details of the chips and the firmware aspects like the functions and bit patterns of the microprogram.

A special requirement of microprogram assemblers which distinguish them from other assemblers is the redefinable multiple-field format of the object code.

A microprogram segment must specify the following things.

1. Sequence of microprogram control flow.
2. Control codes for ALU chips.
3. Register addresses.
4. Timing and enabling conditions for latches and switches.
5. Constants for comparison, preloading or masking.

The control code groups may be bit patterns for direct control of gates, or they may be encoded functions. The typical microinstruction then is a bit pattern of several fields, each of the fields may be of different length in bits. Specifying the content of each microword in an assembly language requires multiple assignments.

The typical line of a bit-slice microprogram therefore, differs from a conventional one-address computer instruction line in that it has multiple opcodes. Also different opcode patterns may call for different field groupings in successive microinstructions. A jump instruction, for example, might call for only two fields: the field specifying the jump function and a long field giving the jump address within the microprogram memory. On the other hand, an ALU operation might require several short fields containing codes giving the first and second operand source location within the register set, the code for the ALU operation to be performed, the destination of the result, and bits to control the handling of carries and condition codes. The bit-slice assembly format should allow several alternative groupings of multiple assignments.

B. DESIGN OF A MICROPROGRAM META-ASSEMBLER

To achieve the changeability of the assembler's target machine the processing of the program is split into two phases:

1. Definition
2. Assembly

Figure 2 shows the general scheme of the microprogram meta-assembler. The definition phase reads a specification of the microinstruction formats including number, sizes, positions and default values of the several fields of each format of microinstruction. The mnemonic tags to be associated with each opcode and, the association with the appropriate formats and field values are also defined. The first main step of the assembler is to acquire these definitions and to construct an internal description of the target machine.

In the second or assembly phase the processing by the assembler is more like assembly language processing of a fixed architecture computer. The basic task is to scan lines of input symbols translating mnemonic codes, statement labels and field values into sequences of binary microinstructions. Fields specified in the definition phase are pulled together, aligned and stuffed into microwords.

The post processing phase reformats the binary object code for use. The use may be simulation directly from the numerical format of the assembly output or it may be implemented in ROM or PROM according to the format specified for the memory design or may be implemented by PLA's specified by the designer. The post processing phase is not implemented in this design.

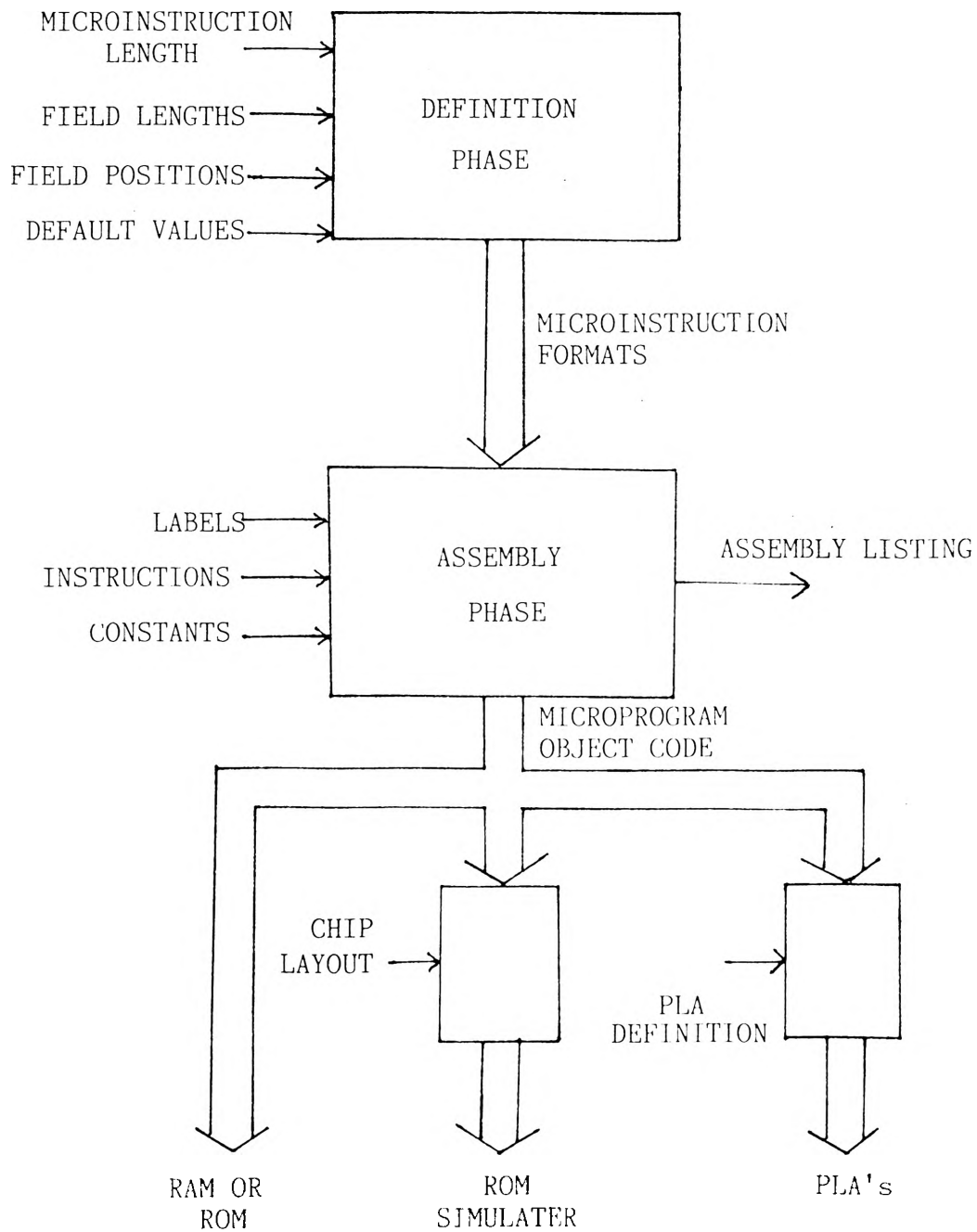


Figure 2. Control Flow for the Meta-Assembler

1. The Definition Process. Procedure definition reads the definitions of the field formats and constructs a definition table which represents the description of the target machine.

a. Data Structure For Definition Table. The data structure used for storing the microinstruction definition is shown in figure 3. The array ARRY1 is a array of pointers. Each element of ARRY1 has two pointers, FIRST and LAST, which point to the beginning and end respectively of a doubly linked list of records associated with that element. Each of these records is the first node of a separate linked list. These linked list have records of type NODEREC, which are used for storing information about a microinstruction field, each of these linked lists are used for storing information about a microinstruction format definition which may consist of the definition of several fields in that microinstruction. The different fields of the record NODEREC are shown in figure 4.

As it is expected that the entries in the definition table would be referenced frequently during the assembly process, a hash function was used to index into ARRY1. The hash function computes the sum of the ordinal values of the first, second, fourth, fifth, seventh, ninth, seventeenth and nineteenth characters of the microinstruction field format definition names and then computes the mod with the size of the array ARRY1, this value is used as the index for array ARRY1.

It is possible that more than one definition name will hash into the same index, hence the technique of chaining is used to resolve collisions. All the records in a linked list have the same hash address.

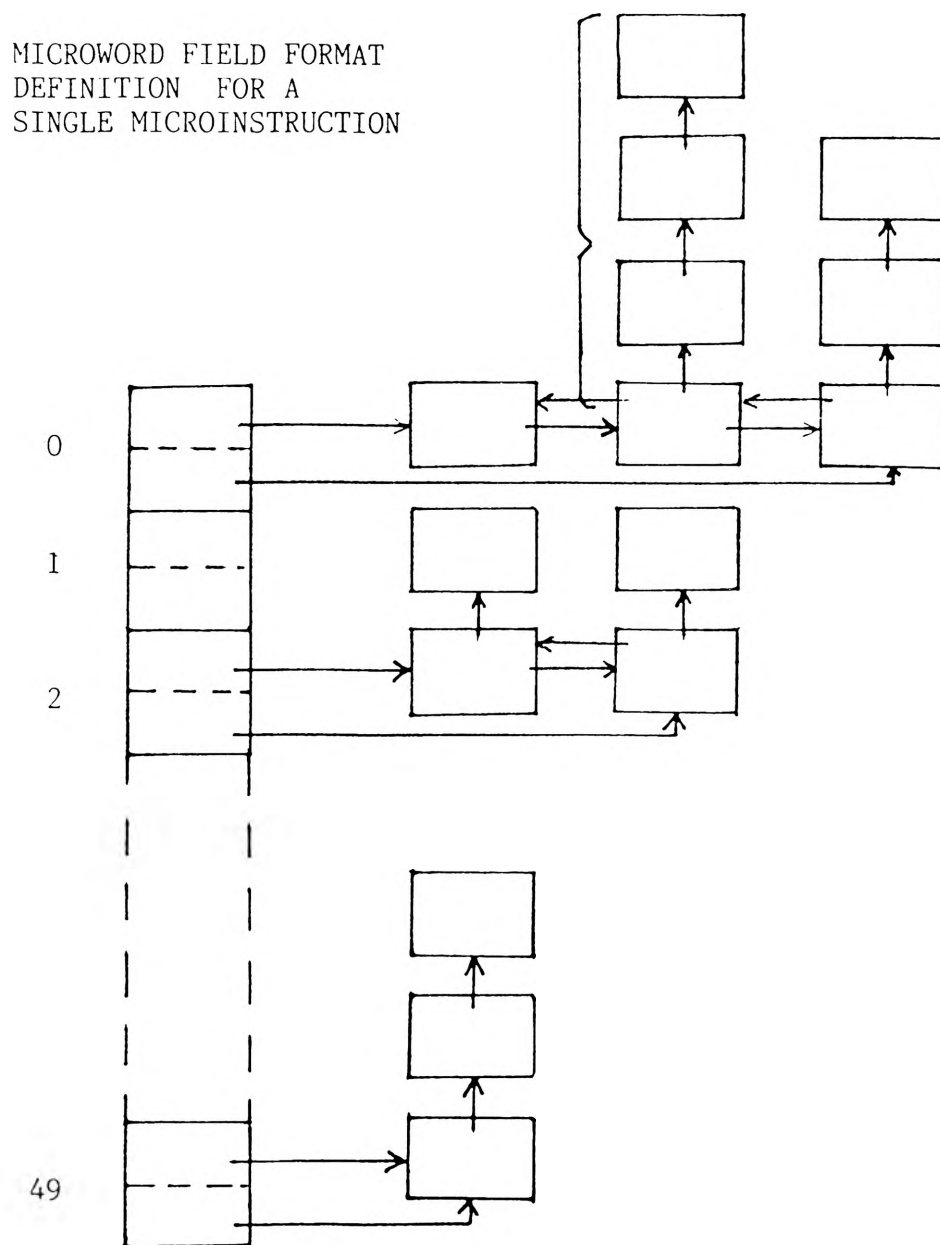


Figure 3. Data Structure for the Definition Table

When a new definition is to be added, the hashing function is computed to calculate the array index, then the record is inserted with alphabetical ordering of the definition name. This reduces the time required to search for a definition in a linked-list.

In a microinstruction field format definition, any number of fields in each format may be defined. A record is associated with each field definition. As the definition of each field in a microinstruction format definition is constructed, it is inserted at the end of the linked list associated with each microinstruction format definition.

The use of dynamic variables to construct records results in efficient utilization of memory space. The design of this data structure is an important consideration for the performance of the meta-assembler.

b. Construction Of The Definition Table. The procedure BUILDDEFTABLE constructs the definition table. The flow chart for procedure BUILDDEFTABLE is shown in figure 5. The format of microinstruction format definition is as follows:

definition name DEF (fieldposition v/c number of bits <modifier> value).

The assembler also allows a numerical value to be assigned to a name. The format is shown below:

name EQU <modifier> value.

Items enclosed in < > are optional. Items enclosed in () can be

SYMB	FORMAT DEFINITION NAME
DEFTYPE	TYPE OF DEFINITION
VARIABLE	CONSTANT OR VARIABLE
NUMBITS	NUMBER OF BITS FOR THE FIELD
LEFTPOS	POSITION OF FIELD
VALUE	DEFAULT VALUE OF FIELD
MODIFIER	MODIFIER FOR THE FIELD
LLINK	POINTER TO PREVIOUS FORMAT DEFINITION
RLINK	POINTER TO NEXT FORMAT DEFINITION
NFIELD	POINTER TO NEXT FIELD DEFINITION IN THE SAME MICROINSTRUCTION FORMAT DEFINITION

Figure 4. Record for Storing Field Attributes

repeated more than once. The microinstruction format definition begins with the definition name followed by the keywords `def`, this is followed by the specifications of a field in the microword. The type of field (variable or constant) is specified by `v` or `c` respectively, this is followed by the number of bits in the field. Specification of the field modifier is optional. Finally a numerical value is specified which corresponds to the default value in the case of a variable field and is the value of the constant in the case of the constant field.

The definition file is read one word at a time, and the attributes of the instruction field is stored in the definition table. The constants defined by the EQU definition are stored in the symbol table.

When the word `DEF` is read from the definition file, the procedure `DEFTABLE` starts searching for information for the current definition of the microinstruction field formats. A record of type `NODEREC` is created to store this information. The fields of this record are shown in figure 4. This record is pointed to by the pointer `WORK`. The position of the field is stored in `WORK|.LEFTPOS`. If the field is defined as a variable field then the boolean variable `WORK|.VARIABLE` is set to `TRUE` or else it is set to `FALSE`. The number of bits in the fields is assigned to `WORK|.NUMBITS`. The optional modifier is stored in `WORK|.MODIFIER`. The default value in the case of a variable field is stored in `WORK|.VALUE`.

When all the information for a field definition is available in the record pointed to by `WORK` this record is inserted in the appropriate linked list as explained in the section 'Data Structure for the

Definition Table'. If a linked list does not exist a new one is created. Any number of fields may be defined for a microinstruction field format.

When the procedure DEFTABLE reads the word WORDLENGTH in the definition file, it expects a number to follow, This number is stored in a variable MICROWORDLENGTH which indicates the width of the target microprogram memory.

Symbolic names which are defined as constant values are stored in a table called the SYMBOL TABLE. This reduces the time to search for the values during the assembly process.

If a error occurs in the definition specification then procedure ERROR is called which outputs the appropriate error message, and the position of the error in the definition file. The procedure DEFINITION returns after a error occurs.

2. ASSEMBLY. The assembly phase consists of the procedure FIRSTPASS and the procedure SECONDPASS.

a. Procedure FIRSTPASS. The flow chart for procedure FIRSTPASS is shown in figure 6. During the first pass the assembler uses the location counter to construct the symbol table, this allows the second pass to use offsets of the identifiers to generate operand addresses. This also allows forward references in the microprogram.

The symbol table with a few example entries is shown in figure 7 The symbol table is a array of records of type SYMBREC. The field SYMBOL stores the symbol name. The field VALUE stores the value of

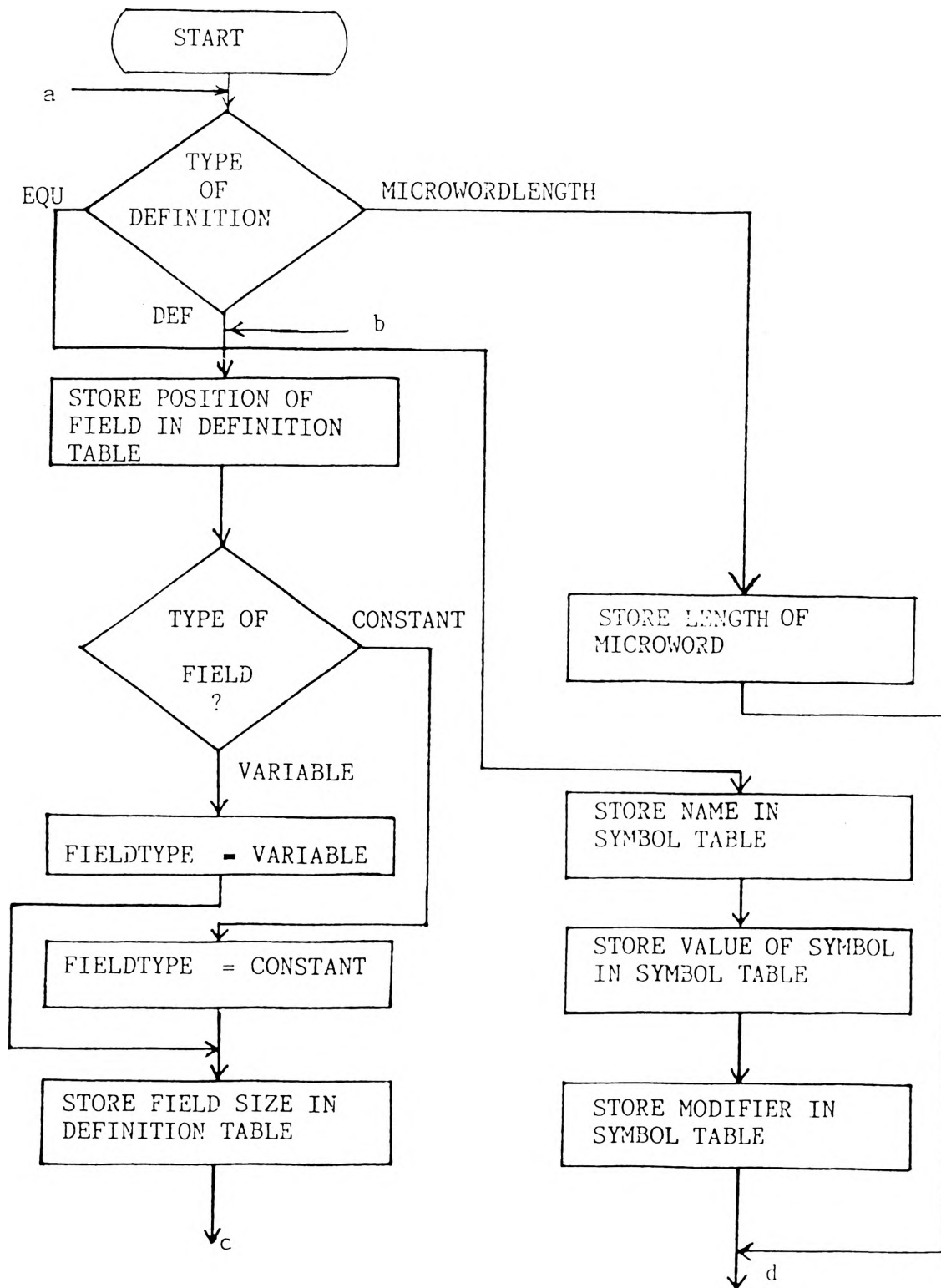


Figure 5. Flow Chart for Procedure BUILDDEFTABLE

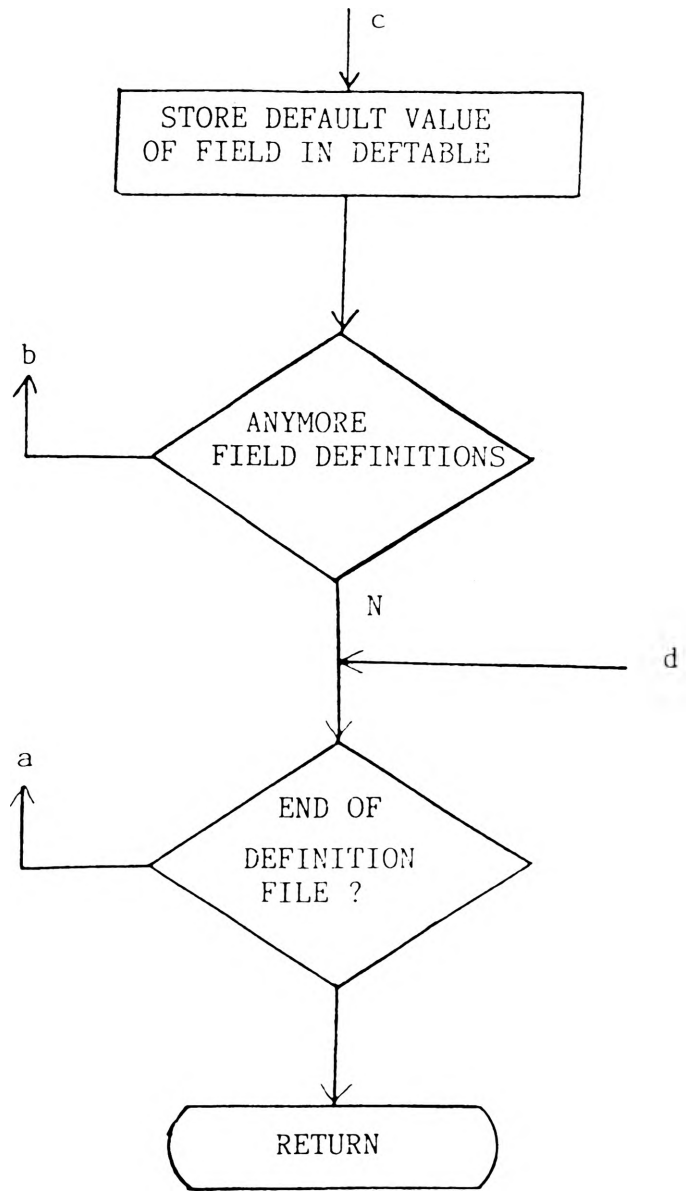


Figure 5. Continued

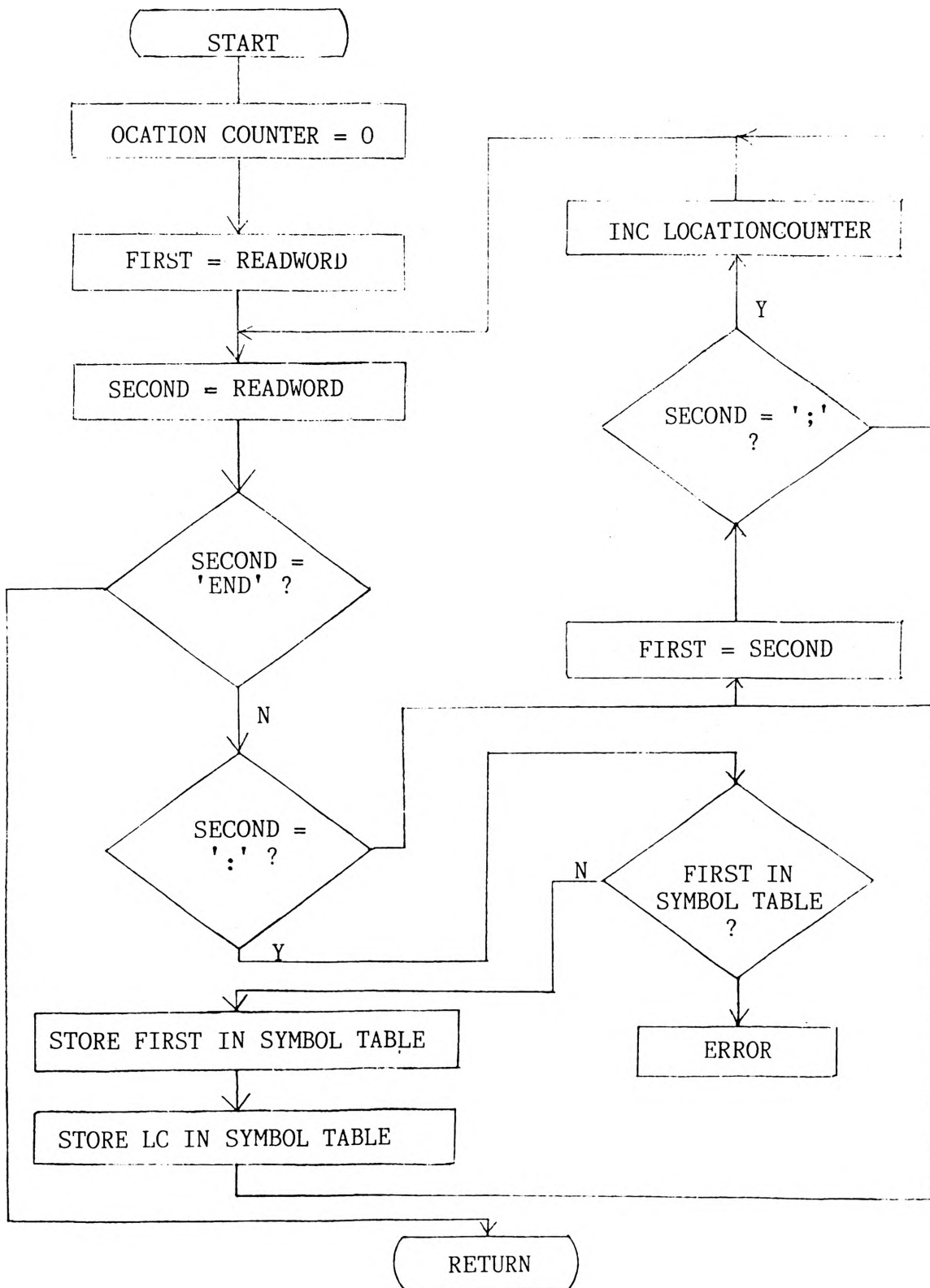


Figure 6. Flow Chart for Procedure FIRSTPASS

the symbol, the field RELOCATIONBIT is a boolean variable which is set to TRUE in case the value of the symbol changes if the microprogram is relocated in memory, otherwise it is set to FALSE. The MODIFIER field is used for storing any modifier if specified for the VALUE field.

The procedure FIRSTPASS scans the source program and when a statement label is read, the label is inserted in the symbol table, and the relative position of the microinstruction in the microprogram memory is inserted in the VALUE field of the symbol table. The RELOCATIONBIT field for this entry in the symbol table is set to TRUE, indicating that the VALUE field will change if the microprogram is relocated, the modifier field is not used for statement labels.

At the end of each statement the relocation counter is incremented. If the same statement-label is defined more than once, procedure ERROR is called and the procedure FIRSTPASS returns. Values of constants defined during the execution of procedure DEFINITION are also entered, but the RELOCATION bit for the corresponding entry in the symbol table is set to FALSE, as these values do not change if the microprogram is relocated.

b. Procedure SECONDPASS. The flowchart for procedure SECONDPASS is shown in figure 8. The format of the assembly line is described as follows.

First a optional statement label can be specified. The statement label must be followed by ':'. The microprogram assembler statement

begins with a microinstruction format definition name followed by numerical values or symbols. The symbol may be a symbolic constant defined during the definition phase or statement labels (for a branch address). The numerical values are assigned to the default value. Values can be preceded by an optional modifier. More than one microinstruction format definition may be specified, these microinstruction format definitions are separated by the character '&'. If a bit in the variable field of a microinstruction format definition overlaps a bit specified in a variable of a previous microinstruction format definition then that bit can be changed only if variable OVERLAP was set to TRUE during the initialization.

In the second pass, fields specified in the definition phase are pulled together, aligned, and stuffed into a microprogram word (microinstruction). The value of a constant field is looked up from the definition table. If a numerical value is specified for a variable field it is inserted into the field position specified by LEFTPOS for this field in the definition file, if a symbol is specified for a variable then the value of the symbol is looked up from the symbol table. If in the source program a value is not specified in the variable field then the default value is looked up from the definition table. Before a field is inserted in the microword, it is modified by the associated modifier. If a microinstruction format definition name that is specified in the source file does not exist in the definition table, then the procedure error is called. If procedure ERROR is called then a error message is displayed, and the position of the error in the source file is indicated. After a error occurs the assembly process is stopped and the procedure

SYMBOL	VALUE	MODIFIER	RELOCATIONBIT
INV	1011	*	FALSE
START	0000	%	TRUE

Figure 7. Symbol Table

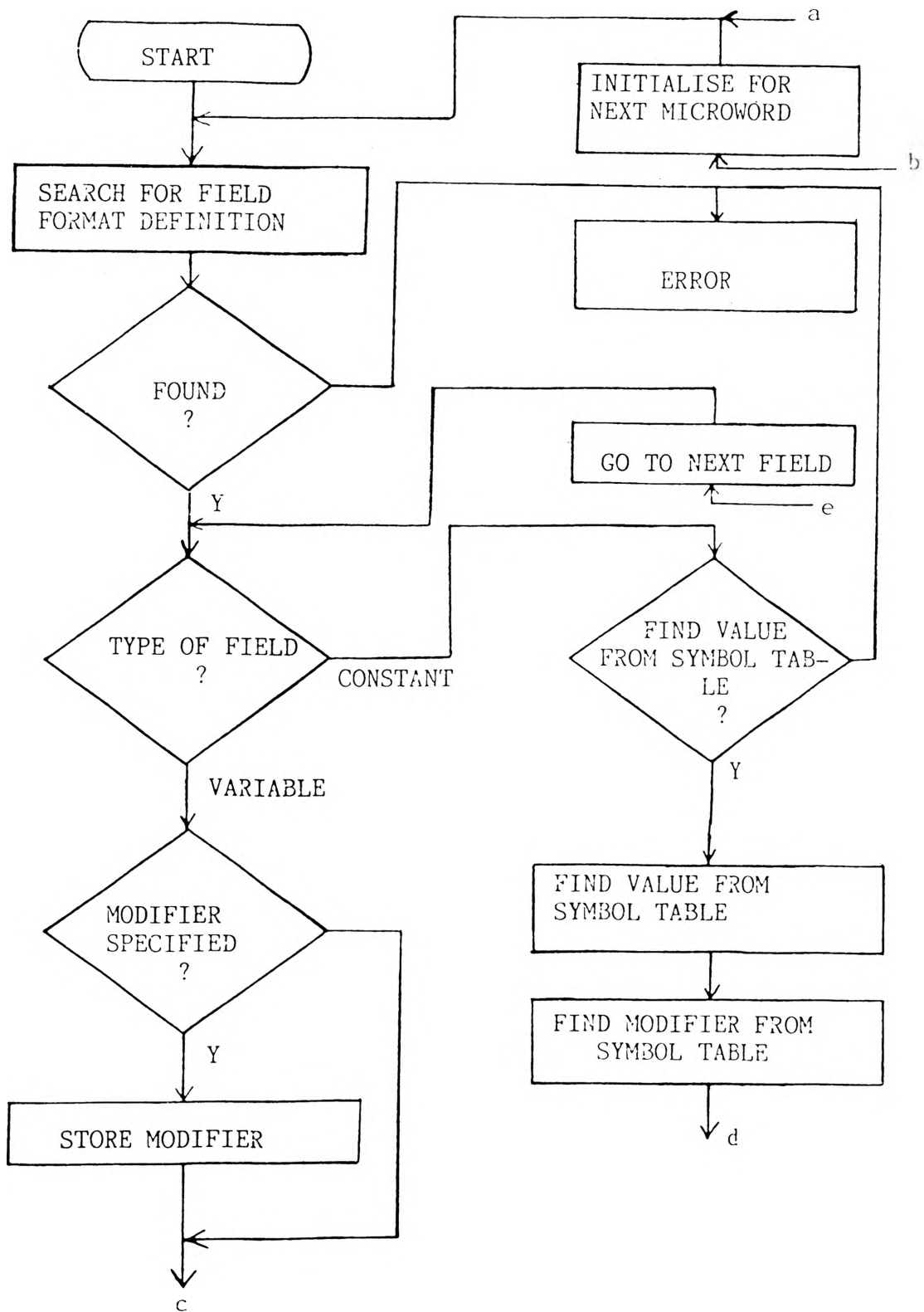


Figure 8. Flow Chart for Procedure SECONDPASS

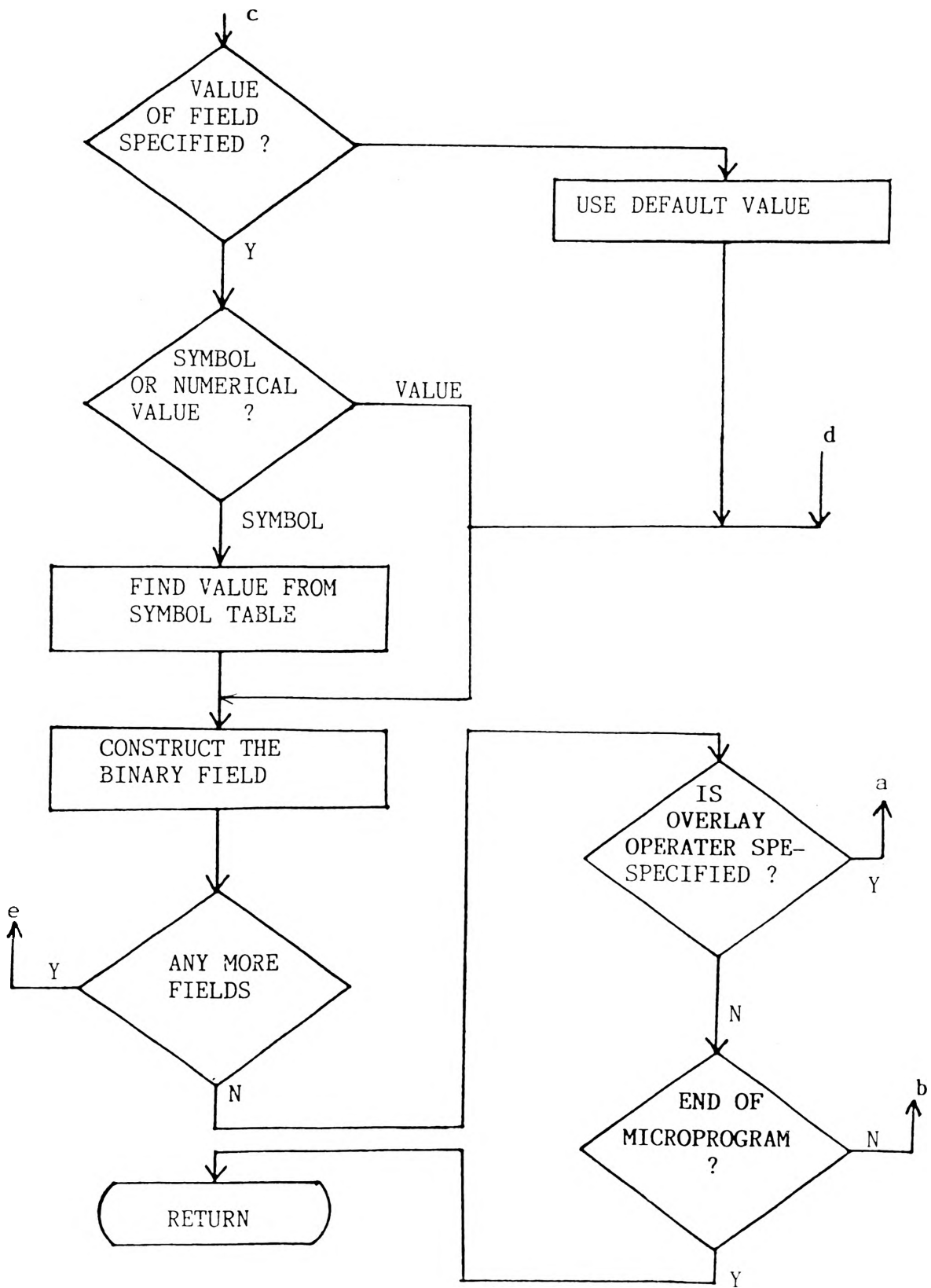


Figure 8. Continued

SECONDPASS returns.

c. Function FINDBITS. The flow chart for function FINDBITS is shown in figure 9. The assembler allows all numerical values to be specified in Hex, Octal or Decimal (by using a suffix of H, Q, or D respectively). If a suffix is not used the number is assumed to be decimal by default. The function FINDBITS converts the number to a binary string.

d. Procedure ERROR. When the procedure ERROR is called a error message is displayed which indicates the type of error, also the position of the error (the line number and the position of the error in that line) are displayed. The boolean variable flag is set to FALSE, this stops the assembly process.

e. Procedure INSERTMICROFIELD. This procedure inserts the binary number passed to it as a parameter into the microword at the position specified by LEFTPOS. The binary string is modified by the modifier specified by MODF before insertion in the microword.

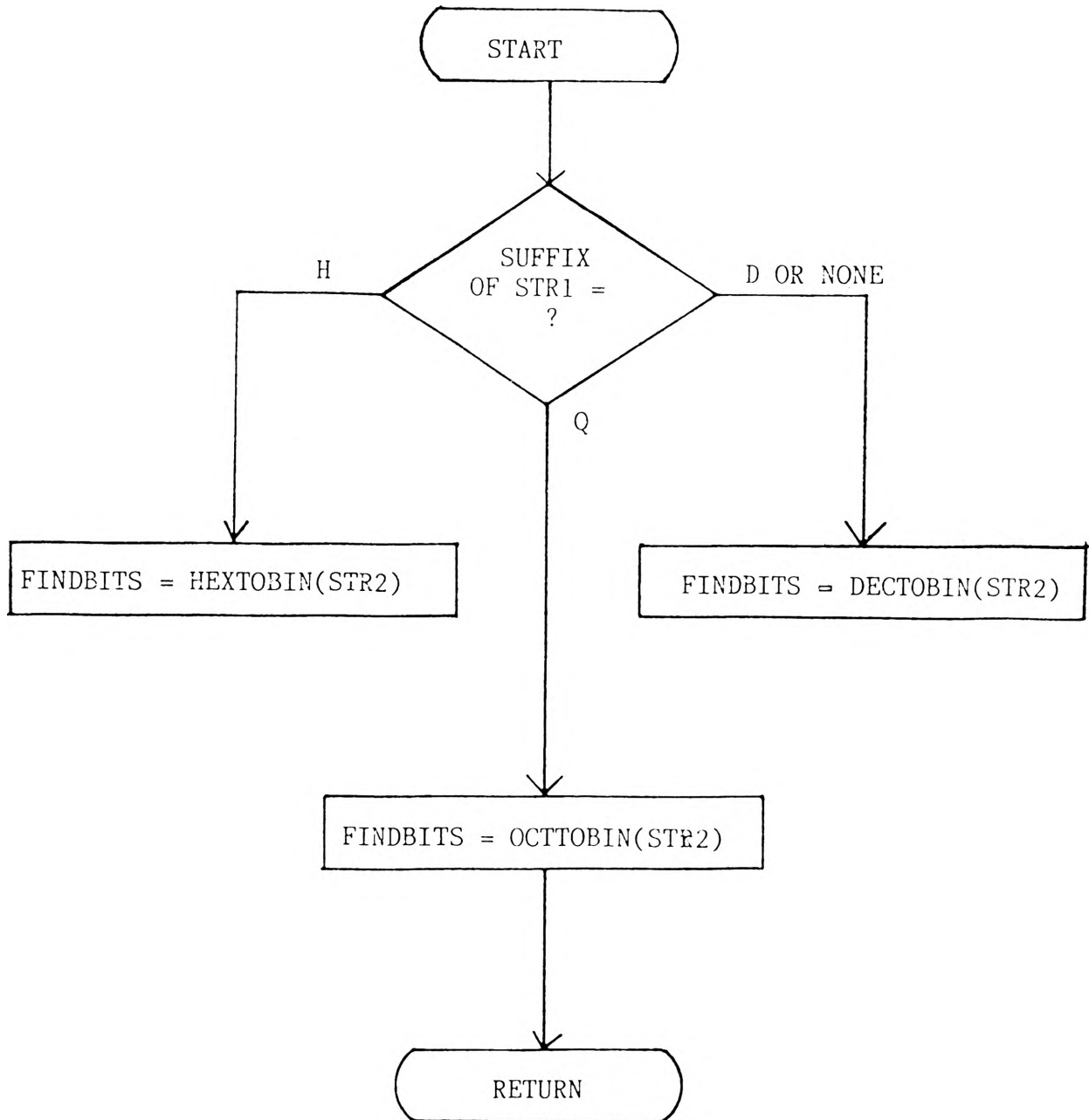


Figure 9. Flow Chart for Function FINDBITS

III. OPTIMIZATION FOR HORIZONTAL MICROPROGRAMMING

Machine or assembler languages are difficult to use for coding large and sophisticated microprograms. It was, therefore unavoidable that high-level languages be developed to provide microprogrammers with the facilities already offered to programmers. The nature of microprogramming, however, requires other features from a high-level language (and its compilers) that are not always to be found at the programming level. These are:

1. The compiler should provide optimized code. (This is particularly relevant for compilers implemented to provide object microcode for a horizontal machine in order to take full advantage of the data path parallelism)
2. The language must be flexible, so that it can be compiled into a variety of microcodes far exceeding the variety of machine languages. The format of microinstruction vary considerably from one machine to another, ranging from a format (vertical) resembling machine-language instructions to a much more elaborate format (horizontal) containing several concurrent microoperation specifications.

The advantages expected from a high-level microprogramming language are generally the same as those offered by high-level programming languages. These are:

1. Writing programs is easier because algorithms rather than implementation details are described. This also makes reading and understanding the programs easier.
2. The ability to write machine-independent programs provides portability for those programs on any machine where a compiler for the

same language is implemented (This is a classical advantage of high-level languages that is not always implemented).

A high-level microprogramming language should enable the user to write microprograms in a conventional sequential, and procedural fashion and permit these microprograms to be compiled into efficiently executable microcode. The desirable properties of a high-level microprogramming language must be a compromise between machine dependency, ease of detecting and representing explicit and implicit parallelism, and the innate "naturalness" required for all programming languages to establish effective man-machine communications.

A high-level language tends to produce inefficient object code unless some effective optimization techniques are applied. A microprogram generated from a high-level language may be optimized with respect to execution time and memory space at two program levels, global and local.

A. GLOBAL OPTIMIZATION

The global level refers to the instruction streams that can be partitioned into independently executable program segments. Global level optimization generally minimizes microprogram execution time by multiprocessing. This involves detection of parallel microprogram streams and code movement. For detection of parallel microprogram streams a microprogram is partitioned into independently executable segments that may be assigned to multiple processors for concurrent execution.

B. LOCAL OPTIMIZATION

Local microprogram optimization is performed on each independently executable microprogram segment. Reduction of execution time is achieved by the exploitation of concurrent micro-operations, this also results in the minimization of memory requirements.

There are two major considerations for local microprogram optimization through the use of concurrent microoperations. The first is the detection of parallel executable operations in a microprogram depending upon data dependency constraints. The second is the limited availability of microprogram resources. Since parallel operations at this level are performed by a single processor, it requires both detection of parallel executable operations in a microprogram and allocation of concurrently usable microprogram resources. Two parallel executable operations detected in a microprogram, for instance, may not be executed in parallel if they use the same resources such as an adder, shifter, register etc (A resource may be a storage resource or a datapath resource). The high level language SIMPL detects parallel microoperations and resolves resource conflicts automatically allowing local optimization of horizontal microprograms.

1. Detection Of Program Parallelism: Single Assignment Approach.

The single assignment approach is used for detecting parallelism in microprograms. This approach confines a variable to represent no more than one value throughout program execution. A variable therefore may be assigned a value and never be reassigned another value during the course of program execution. This restrictive property permits a

statement to be executed without regard to its sequential order in the program as soon as the variables necessary for its execution have been assigned values. Consider the following program segment.

b = 4 1

c = 12.5 2

a = b / c 3

d = a*b 4

e = c*b - a 5

f = c/b6 6

g = 3 * c 7

After the first scan of the program, the variable dependency of a statement is represented by a set of variables that must have values assigned before the execution of the statement is initiated. Figure 10 shows the variable dependency of the statement in the example program segment. Statements 1 and 2 may immediately be executed concurrently since they do not depend on any variable for execution. After variables b and c have been defined, all the statements whose variable dependency is satisfied by one or both of these variables can be executed. Thus, statements 3, 6, and 7, may be executed concurrently.

Statements 4 and 5 will be executed upon satisfaction of their variable dependency that occurs after the execution of statement 3. Thus the single assignment property enables detection as well as execution of parallel executable statements in a sequential program without regard to the order of the statements appearing in a program. The memory requirement of this approach is proportional to n , the number of source program statements.

2. Single Identity Microprogramming Language (SIMPL). One property of SIMPL that distinguishes SIMPL from other microprogramming languages is that it allows the compiler to easily detect parallelism in the sequentially coded high-level source microprogram and to compile it into an efficient object microprogram in a horizontal format. SIMPL allows the user to microprogram sequentially using conventional high-level programming techniques with little notion of parallelism and object code optimization, thereby drastically reducing his burden of microprogram coding. It permits automatic optimization and compilation of sequential microprograms into horizontal microprograms.

For the single assignment concept to be adaptable to the design of microprogramming languages certain differences in the design approaches with programming languages must be clarified. In particular, the environment in which these languages will be used is quite different. A single assignment language adapts to a multiprocessing environment, whereas a microprogramming language is used to specify multiple elemental operations in horizontal microinstructions. The problems

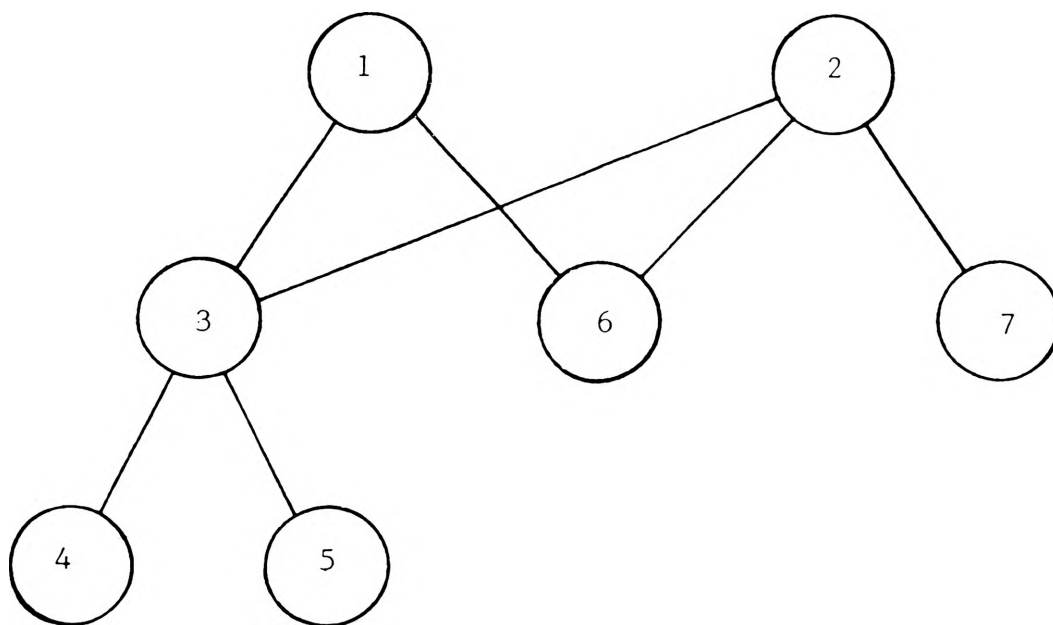


Figure 10. Variable Dependency of Statements

associated with the application of the single assignment concept to the microprogramming language design arise from the following differences with programming languages.

1. Variables in a microprogram represent storage resources.
2. It is not the resources themselves, but rather their contents to which the single assignment concept must be applied.
3. Concurrent processes are executed by multiple processors asynchronously whereas concurrent micro-operations are executed synchronously in one clock cycle.
4. Concurrent processes are realized in a microprogram by parallel data paths and registers instead of multiple processors.
5. A programming language can be translated to a machine independent intermediate code whereas a microprogramming language is translated into a directly executable microprogram that generally consists of complex horizontal microinstructions.

To apply this single identity principle, a source microprogram is analyzed and partitioned into subblocks. A subblock is a portion of a microprogram having at most one control transfer (single or multiple-branch) microoperation to other subblocks. It can be considered as an independent segment of a microprogram and, hence can be analyzed independently for detecting possible parallel executable statements.

As soon as all subblocks have been defined, an analysis for parallelism is initiated on each subblock applying the single identity principle. By the single identity principle, a variable that has been

assigned a value is considered to be defined for all the statements that appear before a statement that reassigns a new value to the variable, the same variable is not considered to be defined for the statements referring to it elsewhere.

3. Constrained Statements. In many computers an out-gate timing pulse of a register is generated earlier than an in-gate pulse in the same clock cycle. This means that there will be no resource conflict if two statements that use this register as source and destination respectively are executed simultaneously. The statements are called constrained statements. Once a statement is in the execution ready list, all of its following constrained statements can be checked for possible simultaneous execution.

C. SIMPL COMPILATION

The SIMPL compiler compiles a sequentially coded source microprogram into an object microprogram in a horizontal microinstruction format. Besides basic syntactic and semantic analysis it identifies and localizes subblocks, performs concurrency and timing analysis, and optimizes the object microcode.

The SIMPL compilation procedure consists of the following four phases syntactic analysis, semantic generation, microoperation concurrency and timing analysis, and microoperation timing optimization. A schematic diagram of the compilation procedure is shown in figure 11. A description of the compilation procedure that follows will be illustrated by the example microprogram in figure 12.

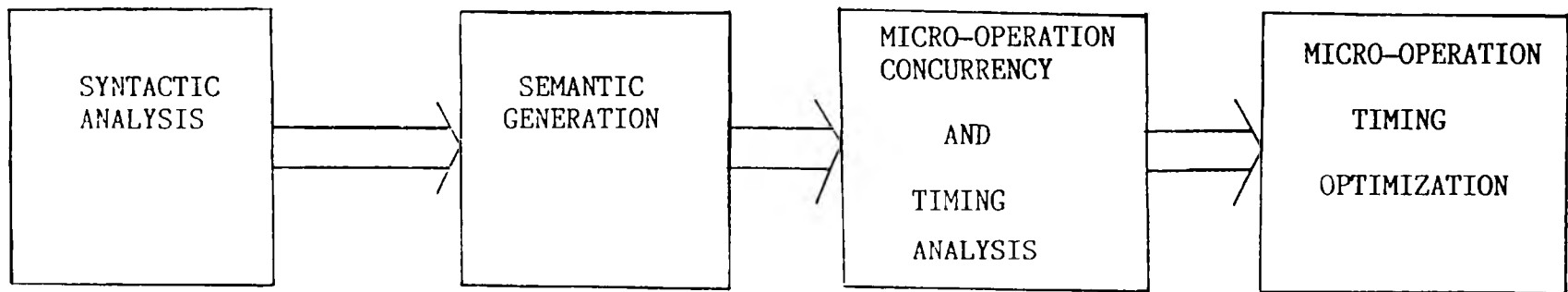


Figure 11. SIMPL Compilation Procedure.

```

begin
comment: determine sign of product in R3 ;
ACC = R1 and M1;
ACC = R2 OR ACC;
R3 = ACC and M1;
comment: force both operands to positive
if R1 < 0 then R1 = R0 - R1;
if R2 < 0 then R2 = R0 - R2;
comment: extract and determine exponent for product.
ACC = R1 and M3;
R4 = R2 and M3;
ACC = R4 + ACC;
R3 = R3 or ACC;
comment: extract mantissa and clear ACC;
R1 = R1 and M4;
R2 = R2 and M4;
ACC = R0;
comment: multiplication proper by shift and add.
while R2 <> 0 do begin
    ACC = ACC+ - 1;
    R2 = R2+ - 1 ;
    if UF = 1 then ACC = R1 + ACC;
end;
comment: if product mantissa overflows, adjust to normalize;
if ACC and M5 <> 0 then begin
    ACC = ACC+ - 1;
    R3 = R3 + M1;
end;
comment: pack exponent and mantissa into floating-point format;
R3 = R3 or ACC;
comment complement mantissa if product sign is negative;
if R3 < 0 then ACC = R0 - ACC;
end;

```

Figure 12. SIMPL Microprogram for 64-bit
Floating Point Multiplication

1. Syntactic Analysis. Syntactic analysis verifies the grammatical correctness of each statement and constructs variable name tables that will be referenced in the subsequent compilation phases.

2. Semantic Analysis. In this phase the semantics of a statement is identified and corresponding symbolic code generated. Several statements are linked to form a block. Each block constitutes an independent set of microoperations, upon which subsequent concurrency analysis is performed. After the completion of this phase, an intermediate microprogram consisting of a number of blocks that contain nonoptimized sequential semantic code is generated. Fig 13 shows the intermediate microprogram. The broken lines in the figure indicate the partitioning of the sequential code into subblocks.

3. Concurrency and Timing Analysis. The intermediate microprogram in the form of partitioned semantic code is subjected to further analysis for detection of local parallelism. This analysis determines the minimum number of cycles to execute each subblock without regard to microprogram resource allocation. It is performed in four steps. The first step scans sequential symbolic code of every subblock, determines the variable dependency of each microoperation and links all microoperations in a subblock according to their variable dependency. The linked microoperations in symbolic code generated in this step constitute the maximal possible parallelism (in each subblock) that assumes no microprogram resource contention.

Applying the single identity principle to the linked

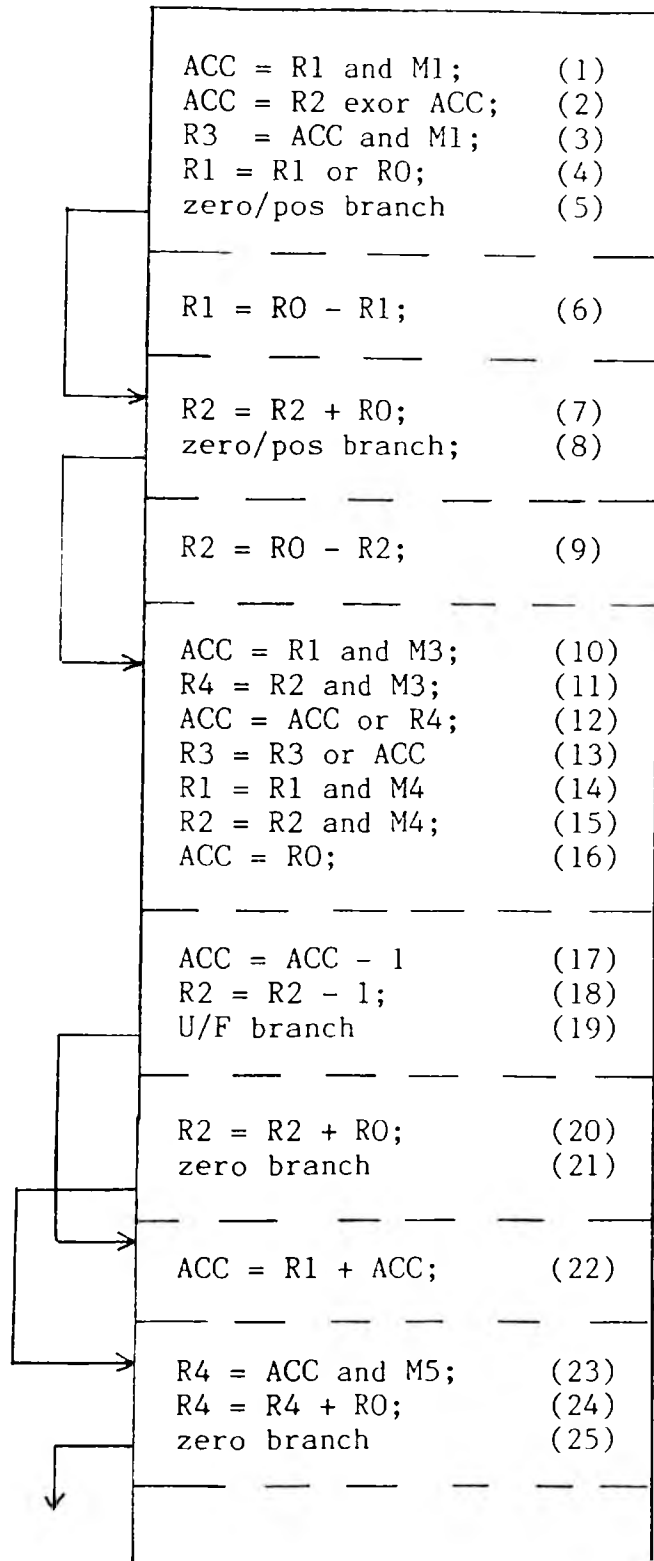


Figure 13. Sequential Semantic Code Generated by Semantic Analysis

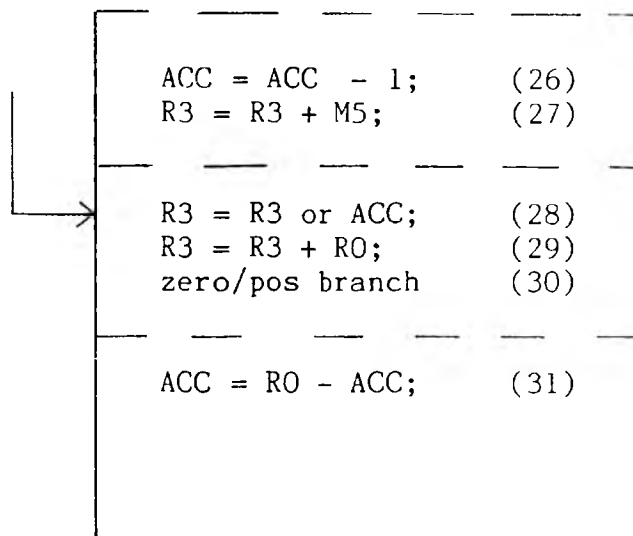


Figure 13. Continued

microoperations, the second step determines the earliest execution time of each microoperation. Figure 14 shows the earliest execution time of the example microprogram. The numbers in the figure refer to the corresponding statements in figure 12. Note that statements 13 and 16 are constrained statements that may be executed concurrently if the in-gate and out-gate timings of the ACC (accumulator) permit.

The third step scans the linked microoperation whose earliest and latest execution times are the same. These are called critical microoperations. In figure 15 underlined numbers indicate noncritical microoperations. The critical microoperations constitute the minimum sequence for a complete microprogram provided unlimited microprogram resources may be always allocated.

4. Microoperation Timing Optimization. This phase of compilation introduces complete machine dependence to generate the object code. The hardware organization and operating characteristics are defined by the microinstruction definition that is represented internally in the compiler. In each subblock, a timeframe refers to execution timing of concurrent, critical microoperations relative to those assigned to other time frames.

Thus each line of figure 14 and 15 constitutes a time frame. Critical microoperations assigned to the same time frame are examined for their resource usage and conflict, and if they create resource conflict, they are ordered in sequence in the object code to resolve their resource contention. Sequential ordering of critical microoperation is confined within the same time frame relative to those assigned

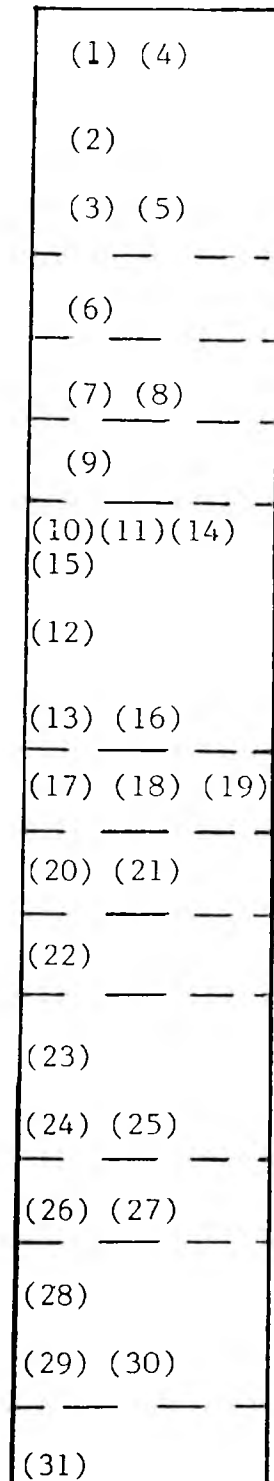


Figure 14. Earliest Execution Timing of Concurrent Microoperations

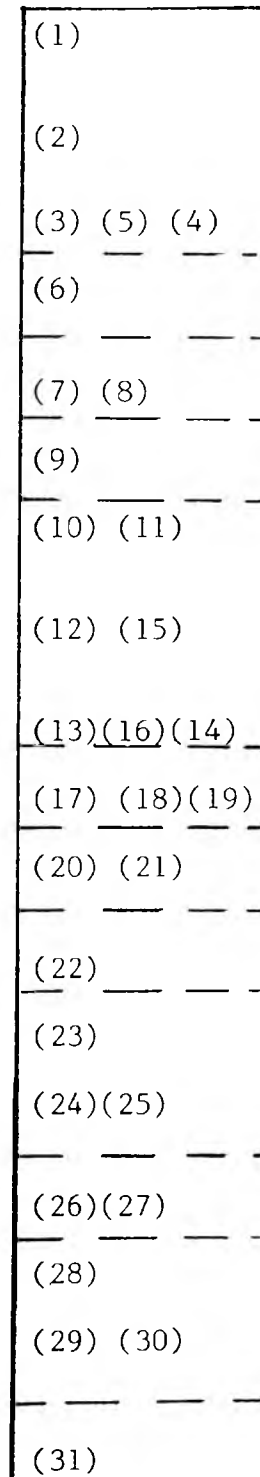


Figure 15. Latest Execution Timing of Concurrent Microoperations

to other time frames. For example, two critical microoperations assigned to the same time frame that use the same adder cannot be executed concurrently, hence they are reassigned to separate steps with the same time frame in the object code. If they use no conflicting resource and can be coded in the same microinstruction they are coded for concurrent execution. This process generates a minimum set of microinstructions that includes all the critical microoperations without resource conflict. The remaining noncritical microoperations are placed in this set of microinstructions at their earliest execution times to achieve higher parallelism. This procedure is illustrated by the flowcharts of figure 16 and 17. Figure 16 shows the procedure for rearranging critical microoperations by testing resource conflict. Resource conflict is determined by two tests labelled "any operation conflict" and "register conflict" in the figure. The i th element of array K contains the number of critical microoperations that are assigned to the i th time frame. This number is used to exhaustively test resource conflict of all critical microoperations at each time frame. This procedure generates a minimum set of microinstructions that constitute a sequence of critical microoperations. Figure 17 shows the procedure for detecting concurrency of noncritical micro-operations and placing them in the minimum set of microinstruction. Variables CONCURNT and NEXT in the figure are lists of microoperations that may be specified in the current and next microinstructions respectively. Two tests labelled "operation conflict" and "register conflict" detect resource conflict. Each noncritical microoperation is coded into appropriate fields of a

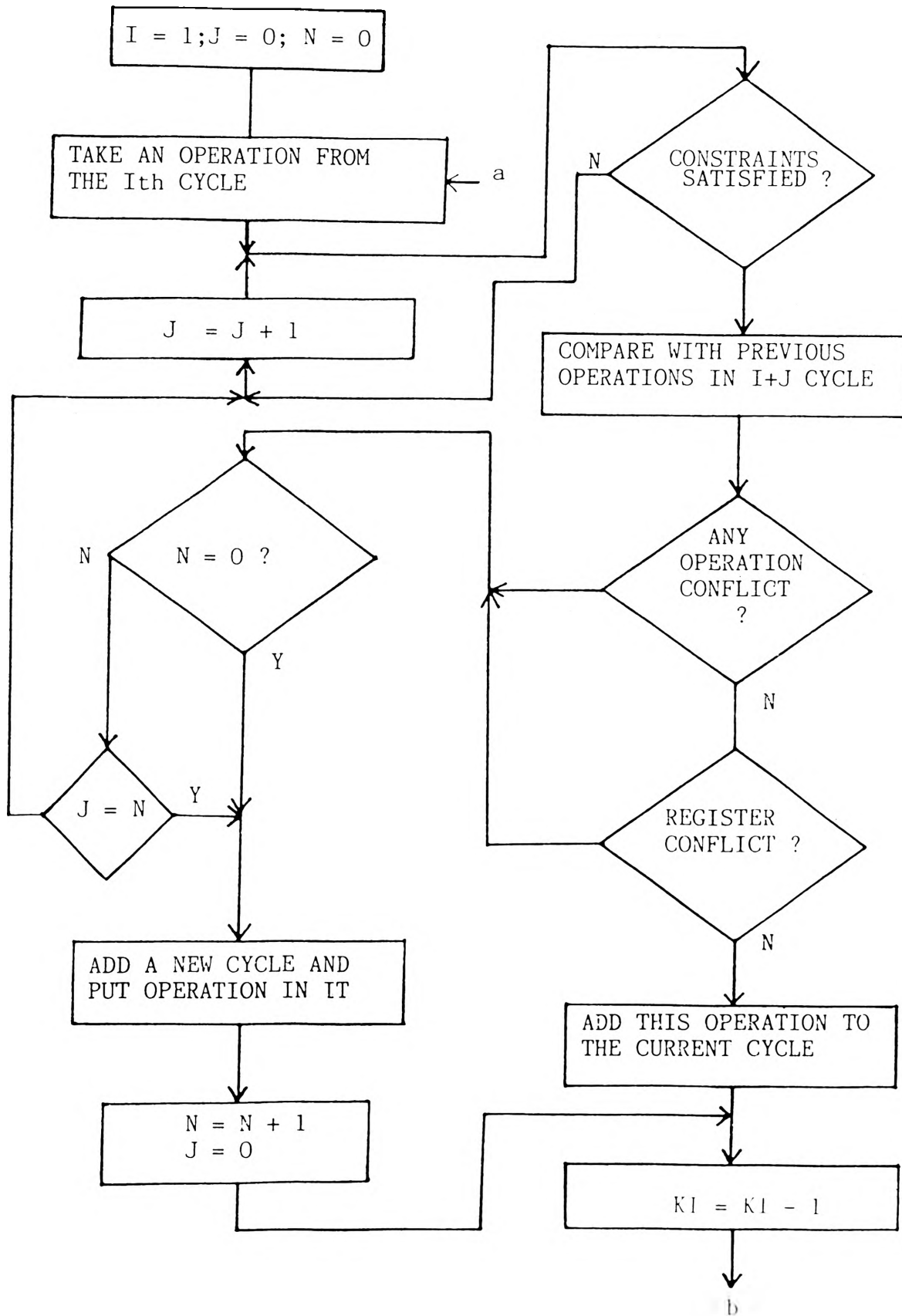


Figure 16. Flow Chart for Resolving Resource Conflict in Critical Microoperations

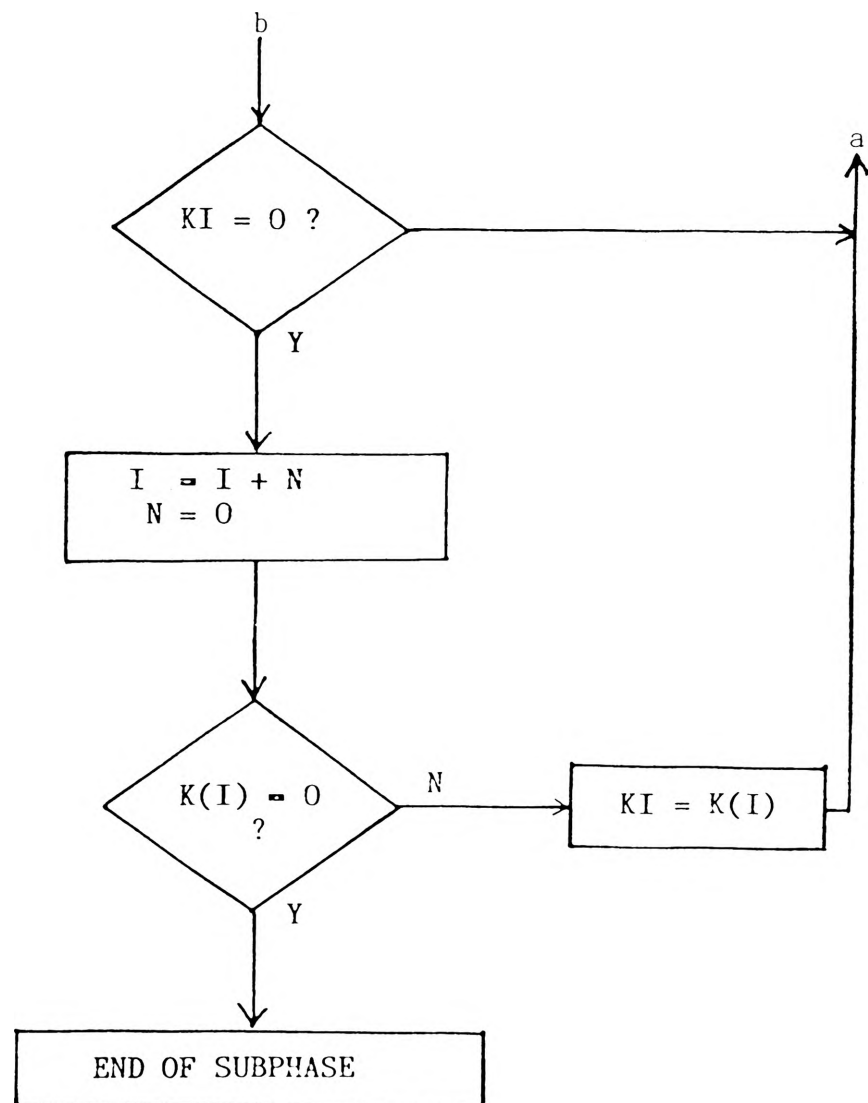


Figure 16. Continued

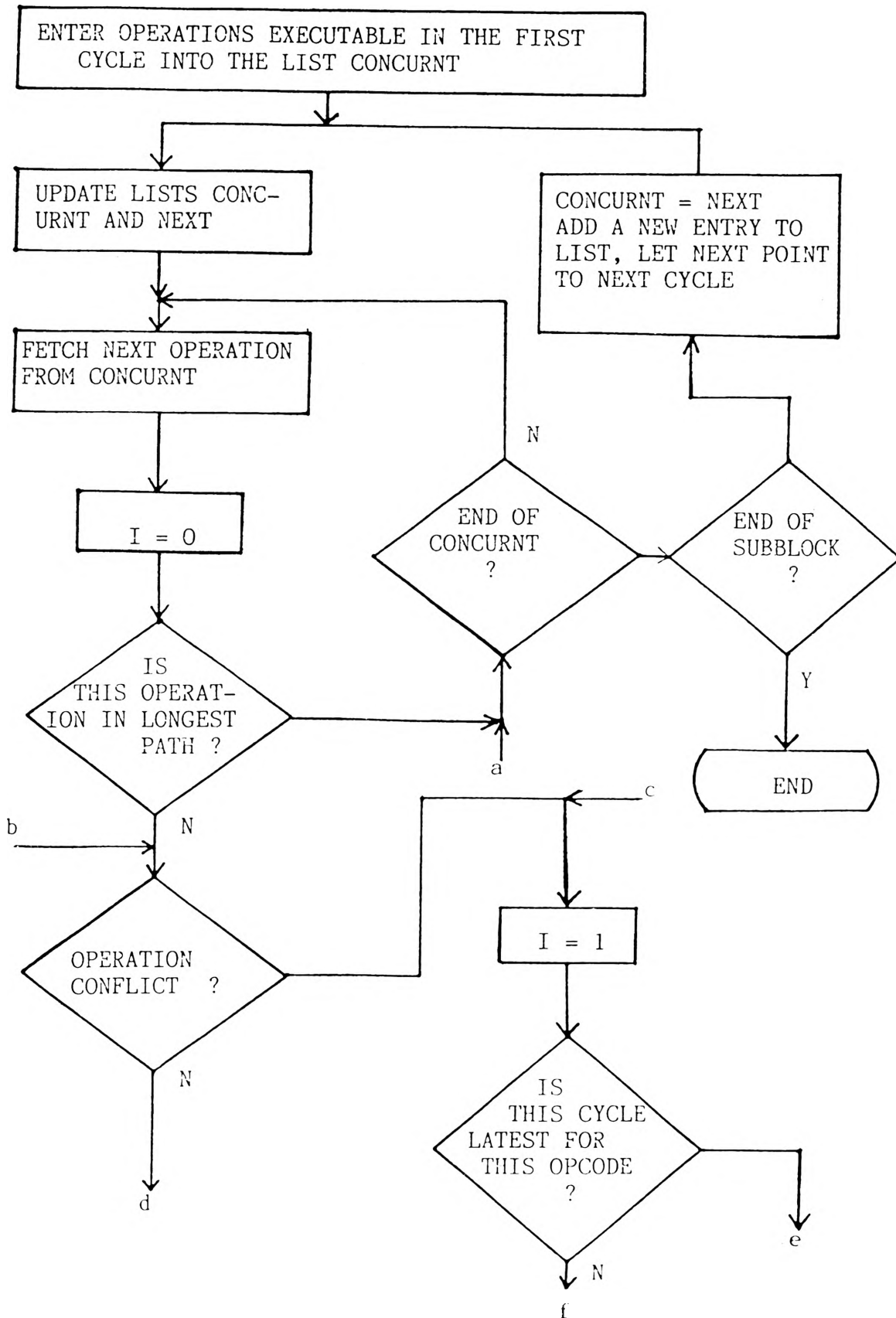


Figure 17. Flow Chart for Finding the Minimum Sequence

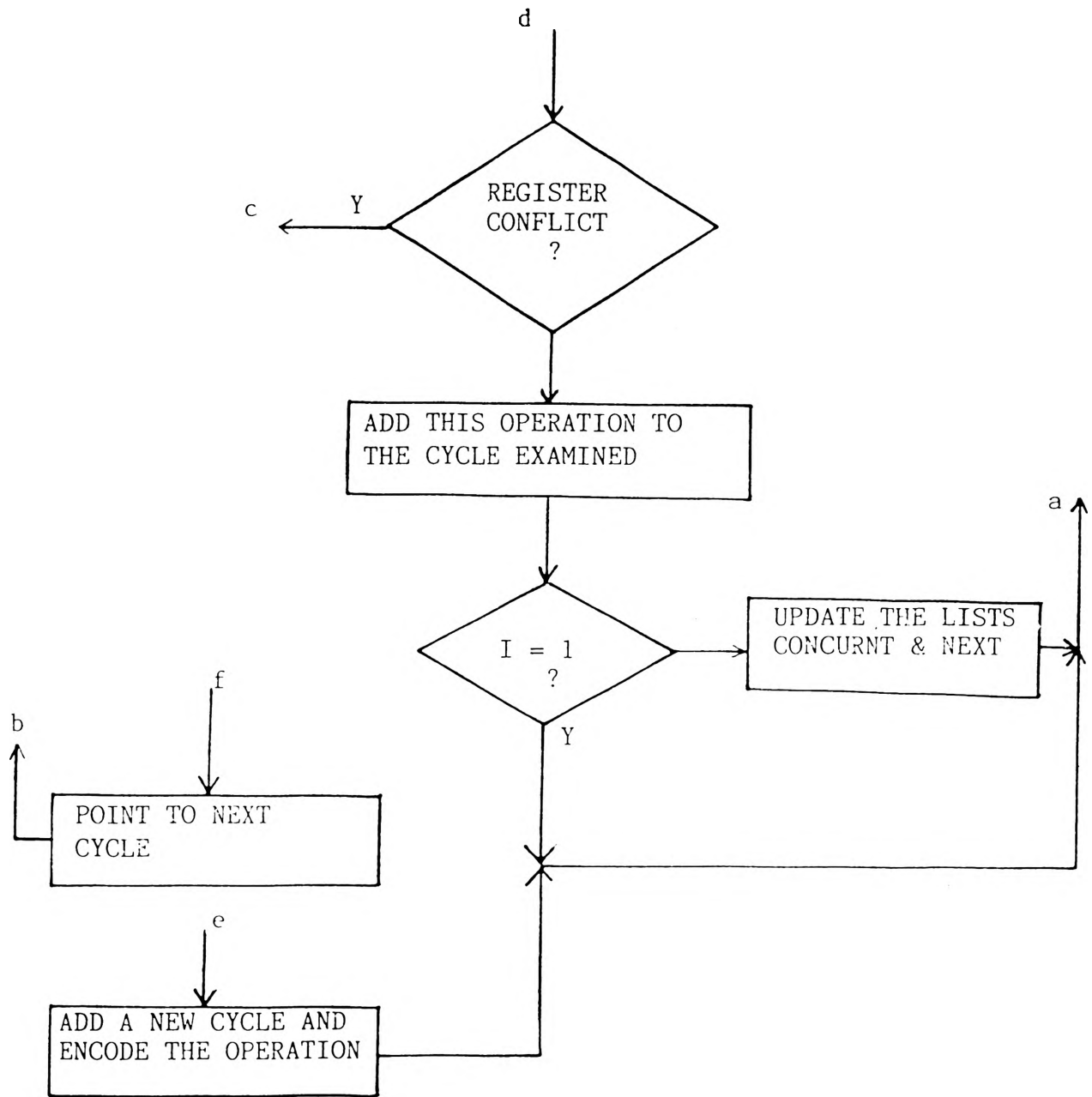


Figure 17. Continued

microinstruction that executes at its earliest execution time without resource conflict. If it cannot be coded into a microinstruction before its latest execution time, an additional microinstruction is introduced. Figure 18 shows concurrency of microoperations coded in the object microprogram.

SIMPL may be compiled effectively into vertical as well as horizontal microprograms. The SIMPL compiler is designed to be highly transportable. Its compilation procedure involves largely manipulation of the source and intermediate microprograms for optimization. These procedures are independent of the machine for which microprograms are compiled. Machine dependency is introduced only at the last phase of compilation. This suggests that small modifications should suffice to adapt the compiler for different microprogrammed machines.

(1) (4)
(2)
(3) (5)
(6)
(7) (8)
(9) (10)
(10)
(11)
(12) (14)
(13) (16)
(15)
(17)
(18) (19)
(20) (21)
(22)
(23)
(24) (25)
(26)
(27)
(28)
(29) (30)
(31)

Figure 18. Concurrency Achieved in the Object Microprogram

IV. MINIMIZATION OF ROM WIDTH

Some of the factors that are to be considered for optimizing microprograms using bit reduction techniques are coding efficiency, bit packing, and function extraction. Coding efficiency is an ad hoc procedure for estimating microcode utilization. Unfortunately this is insufficient to completely describe microcode utilization because a degree of randomness in the bit patterns of the ROM needs to be considered. Bit packing and function extraction are rudimentary forms of ROM width reduction. The microcode is analyzed on a column-wise basis to search for patterns, symmetry, or functions which can be removed from the control storage and replaced by external circuitry. As an ad hoc approach they are employed, at best, in short microprograms. For longer microprograms specialized methods like conflict partition algorithms need to be used. A drawback of reducing ROM width is that it results in increasing path delays in circuitry external to the ROM and hence results in lower speed. Since any bit reduction will require some encoding of the micro-orders there will be a loss of flexibility. Hence the inherent flexibility of a wide microinstruction is lost as microinstructions are encoded more tightly.

A. GENERAL CONSIDERATIONS FOR ROM WIDTH REDUCTION

1. Coding Efficiency. Coding efficiency is a measure of the degree of randomness of the bits of ROM. As such it serves to identify an "utilization level" of the ROM as a control store for the microprograms via the pattern of the 1's and 0's in the rows and

columns for the actual microprogram under investigation. The bit patterns are analyzed based upon purely statistical considerations. If the degree of randomness is low, pattern extraction by subsequent external hardware implementation may be possible. However if the bits are random, it may be difficult to extract patterns which can be implemented by external logic.

The measure of randomness will be defined as the coding efficiency, CE, of a ROM. CE is a upper limit on the amount of information stored in the ROM expressed as a percentage of the maximum amount storable in the same number of bits.

Let B be the width of the ROM and W be the number of words in the ROM. The number of bits of information, N, contained in the ROM is:

$$N = P_0 \log_2 1/P_0 + P_1 \log_2 1/P_1 \quad B.W$$

and the coding efficiency is defined as:

$$C = N/B.W \cdot 100 \%$$

A graph of coding efficiency vs fractional bit content is shown in figure 19. To employ the coding efficiency as a measure, count the lesser number of 1's or 0's in each column and divide by the number of available ROM bit positions to get the fractional bit content. This is the fractional bit content of the ROM. The potential maximum coding efficiency of the ROM can be determined from figure 19. Complementing a column leaves information content unchanged. The number obtained from figure 19 represents the percentage of theoretically maximum coding efficiency for a ROM of some given size. For example, if a

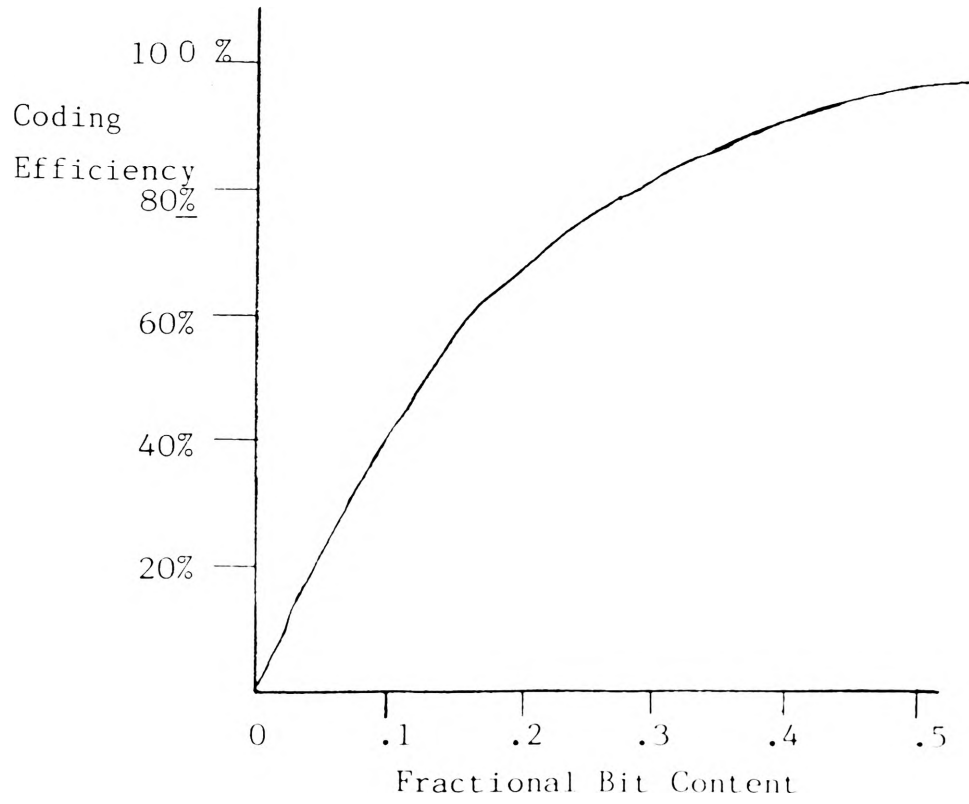


Figure 19. Coding efficiency Chart

fractional bit content of 0.22 is obtained for some ROM, the ROM processes a coding efficiency within 75% of the theoretical maximum.

The coding efficiency measures the degree of randomness in the distribution of bits in the ROM. If a microprogram has maximum efficiency, little can be done to reduce the number of bits. Hence an exercise to reduce the ROM may not succeed. Efficiencies less than maximum suggest the existence of a pattern in the ROM, although extraction may not be obvious. Coding efficiency simply measures the capacity of a ROM to be minimized but does not give any indication of what can be done successfully to reduce the width.

As an example of ROM coding efficiency, consider the ROM shown in figure 20. This microprogram has six lines of code with six output control signals. The fractional bit content is $11/36 = 0.3$. From figure 19, a fractional bit content of 0.3 is equivalent to an 88% maximum coding efficiency. Thus this ROM is coded to within 88 % of its theoretically maximum coding efficiency for a 36 bit ROM. In figure 20 no apparent visual bit pattern can be observed. This ROM is sufficiently utilized.

2. Function Extraction And Bit Packing. These are elementary procedures for ROM minimization. In the case of function extraction columns in the ROM are examined for some intrinsic boolean relationship. For example, if the fourth column in a ROM is identical to the sixth column in the same ROM, this equivalence relationship allows the elimination of one column from the ROM, Hence the bit dimension of the microprogram is reduced by one bit. In the case of bit packing the

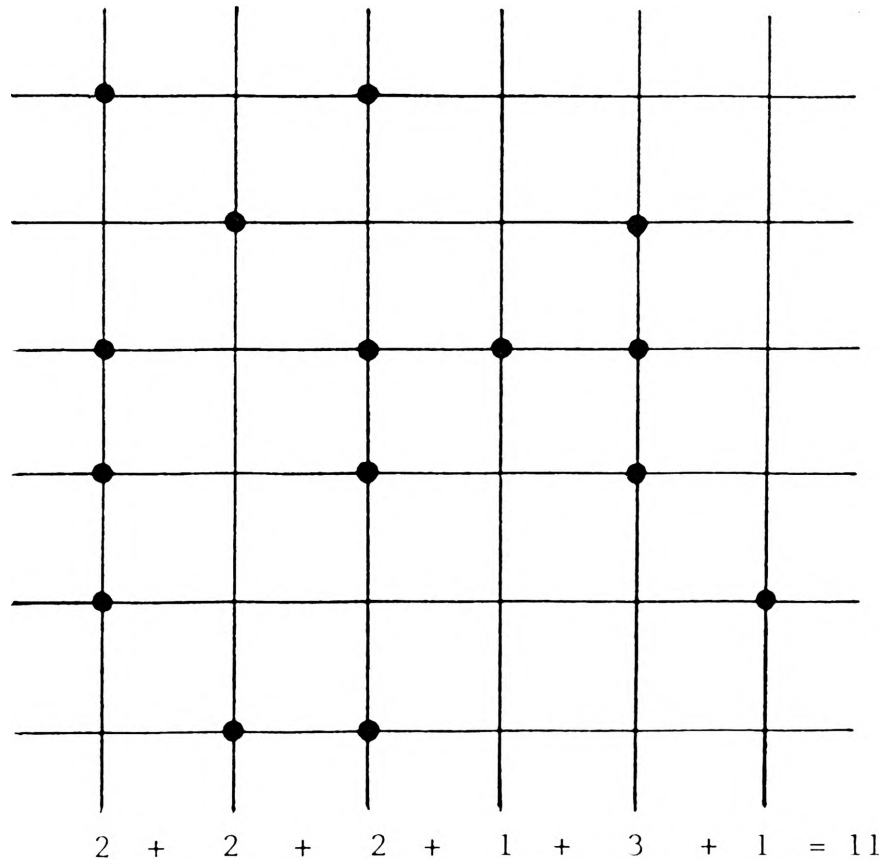


Figure 20. ROM for Coding Efficiency Example

columns are again examined to find individual micro-orders or groups of micro-orders that are individually activated or group-wise activated respectively. These micro-orders are non-conflicting. Hence such micro-orders can be encoded without destroying the microprogram.

As an example consider figure 21. Here an eight word microprogram energizes eight micro-orders $m_1..m_8$. Note that micro-orders m_1 and m_2 , whenever energized are equivalent for every ROM word. Therefore one column can be eliminated since they are identical. The ROM is now reduced by one column. Column 4 is the boolean complement of column 3. The complement of column 3 can be used for energizing micro-order m_4 . In figure 22, combine columns 1 and 2 into one column and combine previous columns 3 and 4 into a new column 2. The ROM is now reduced by two columns 1 & 2, namely the boolean AND of columns 1 and 2 by using the external gate inverter in figure 22. Lastly column 7 and 8 in the previous ROM cannot be reduced. The eight column ROM is reduced to five columns.

3. Conflict Groups. A group of micro-orders are nonconflicting if no two micro-orders in the group are activated simultaneously. For example consider the ROM of figure 23. Here the last microinstruction when retrieved energizes micro-order m_5 and no other micro-order. Also when the fifth microinstruction is called m_5 is energized alone. If other micro-orders are found they are singly activated for any microinstruction. These mutually exclusive activated micro-orders can be grouped into one field, and a decoder can be used at the output of the ROM to select outputs.

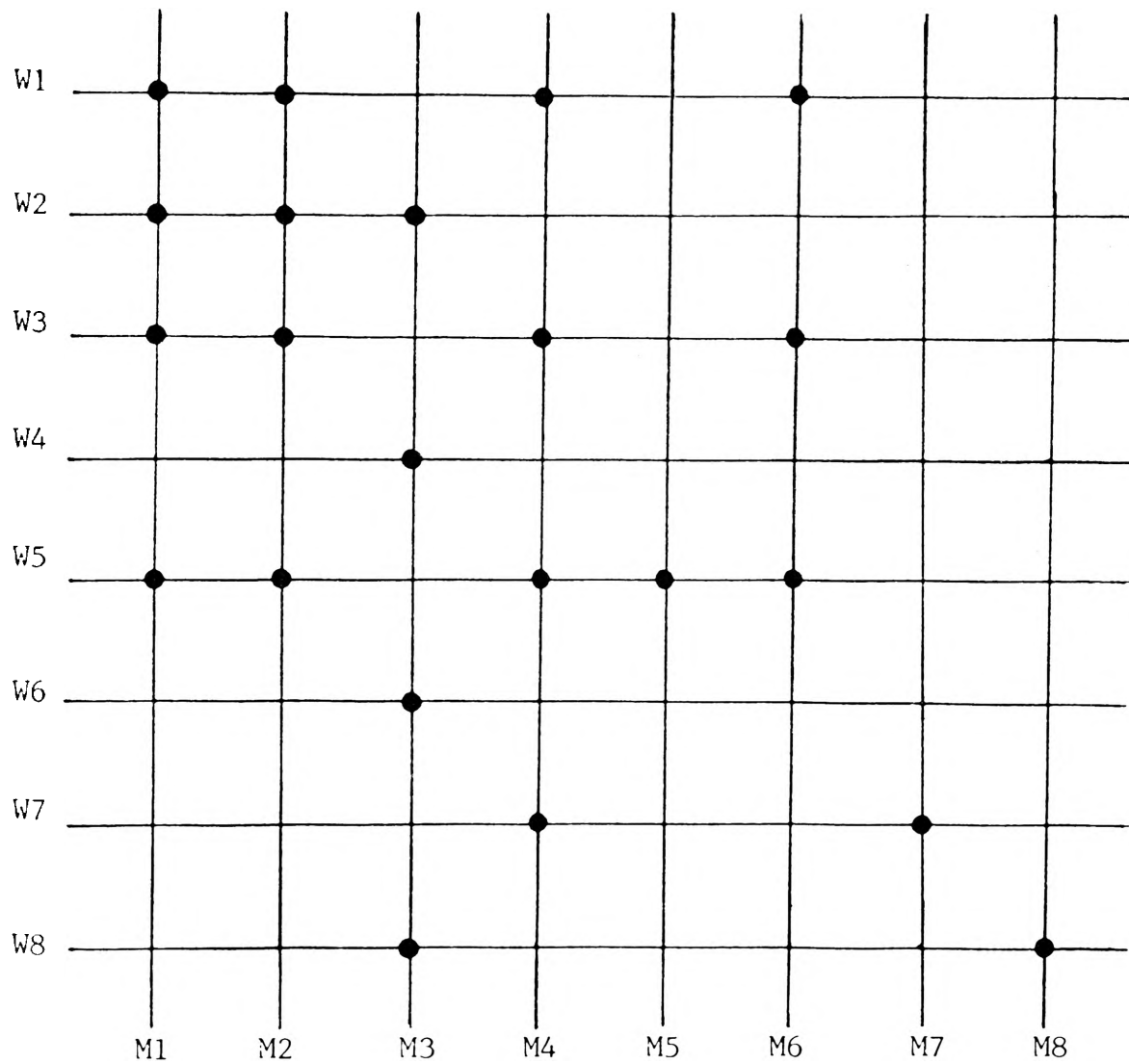


Figure 21. ROM with Internal Boolean Groups

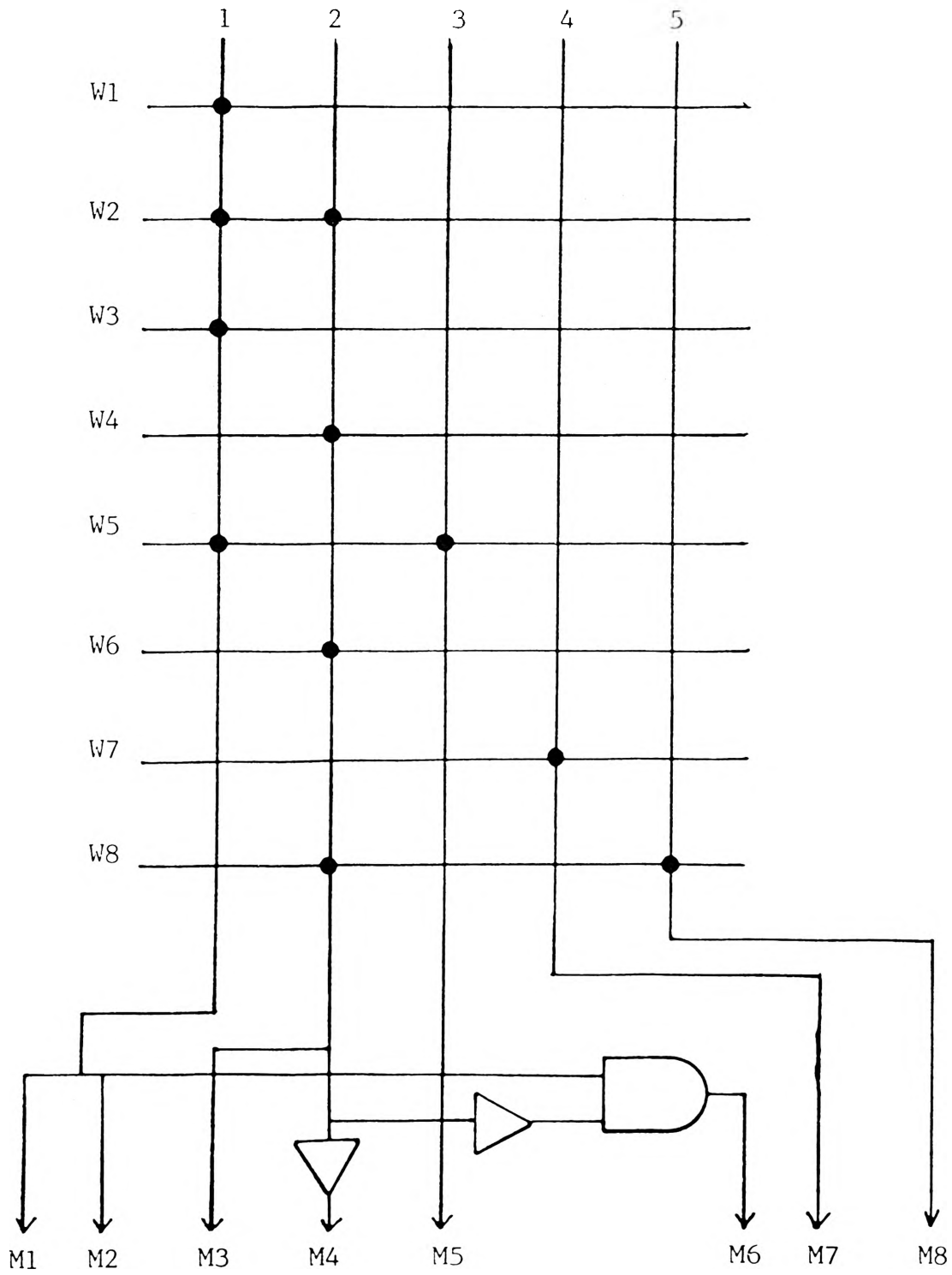


Figure 22. ROM with External Boolean Groups

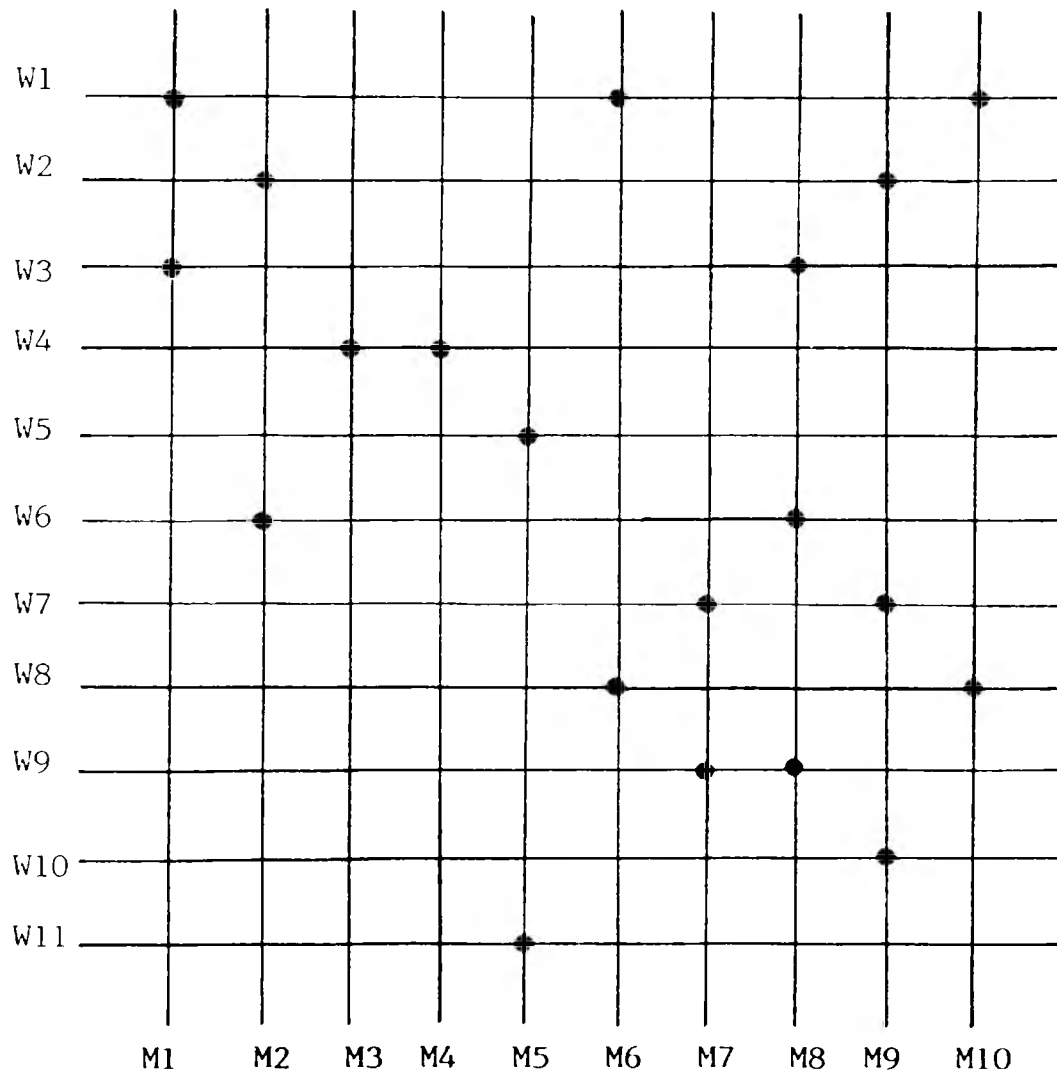


Figure 23. ROM with no Encoding

A graphical procedure is described to implement the method of examining a ROM layout for conflict groups. In figure 23 there are ten micro-orders to be energized, $m_1 \dots m_{10}$. By examining simultaneous column outputs or micro-orders, a conflict diagram can be generated. This conflict diagram will be used in grouping or encoding the original ROM in order to reduce the ROM width.

When micro-order m_1 is activated in microinstruction 1 and 3, micro-order m_6 and m_{10} are simultaneously activated. In the conflict diagram a line is drawn from m_1 to m_{10} . This indicates that when m_1 is energized, it is also possible that either m_6 or m_{10} may be simultaneously energized. In the third microinstruction, micro-order m_8 is energized along with m_1 . Hence in figure 24 a connecting line is drawn between m_1 and m_8 . At this point when micro-order m_1 is energized, it is possible that m_6 , m_8 , or m_{10} may be also energized and thus m_1 conflicts with m_6 , m_8 , or m_{10} . The procedure is repeated for all remaining micro-orders and their conflicts, making the appropriate connecting link (to designate a conflict) between the conflicting micro-orders.

As it is needed to find columns of a ROM which never appear simultaneously as outputs, combine into a set (field) all nonconflicting micro-orders. This grouping procedure is aided by the conflict diagram.

B. ALGORITHMS

1. A Conflict Partition Algorithm. A algorithm for searching through conflict sets to obtain compatible sets for encoding bits is described. The flowchart for the algorithm is shown in figure 25. The first step is to search through the conflict diagram for the set or sets which have the greatest number of conflicts and to isolate all such sets. The objective is to separate these sets into groups which cannot be used as inputs to the same decoder. These partitioned sets are called disjoint sets. The next step is to group remaining sets into compatibility sets. Eventually such sets will serve as inputs to decoders. Finally check to see that all micro-orders have been assigned, returning to the first step, or completing the ROM width reduction by encoding these sets into fields.

To illustrate this algorithm consider the ROM of figure 23. Begin by grouping disjoint n -maximal conflict sets, where n is the largest conflict micro-order to be found. In figure 23 start with m_1 , which is three maximal conflict. Micro-order m_2 is two-maximal conflict which is compatible with m_1 , m_3 is a one-maximal conflict micro-order, and m_5 is compatible with all micro-orders.

The diagram is examined to find all lesser conflicting micro-orders which are compatible with the previous entries. The process is continued until all micro-orders are searched.

A NOP (no operation) is included in every set in the disjoint grouping (m_1, m_2, m_3, m_5, m_7). The NOP is included as a micro-order

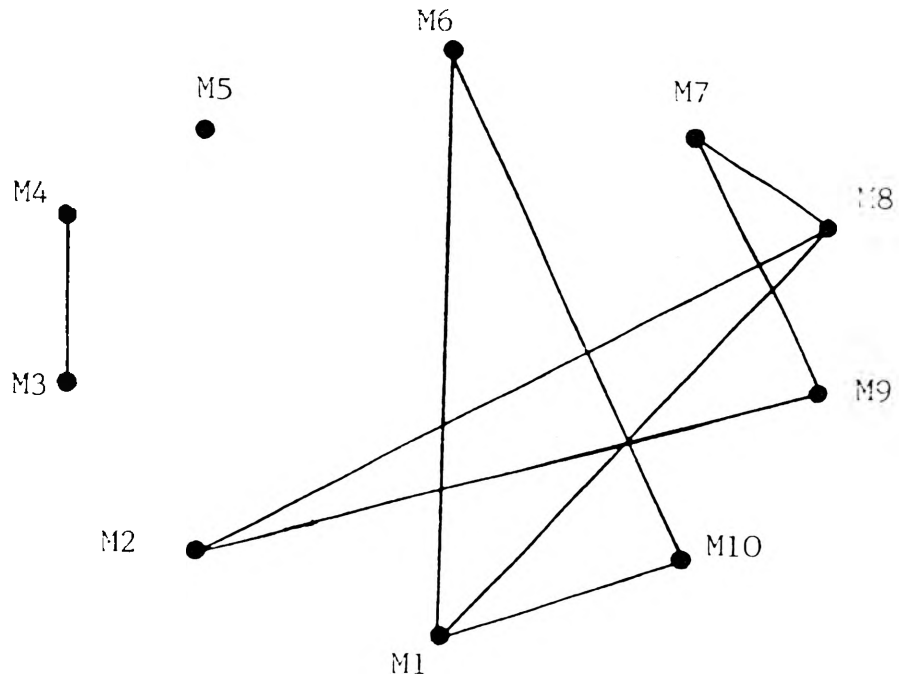


Figure 24. Conflict Diagram

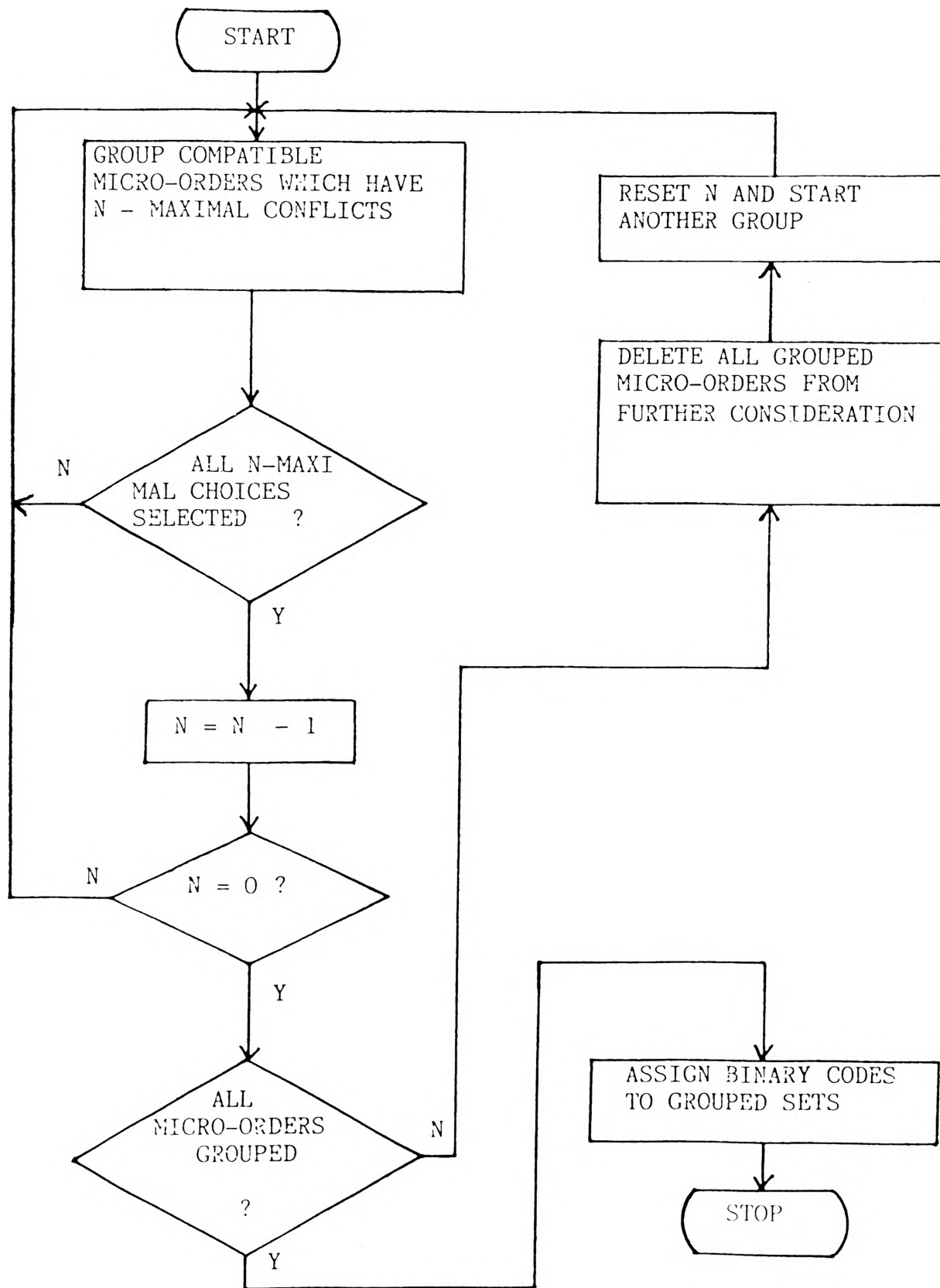


Figure 25. Flow Chart for Conflict Partition Algorithm

for a maximum number of six elements in this set. A one out of eight decoder is chosen. The eight possible choices need only three inputs. m_1, m_2, m_3, m_5, m_7 can be reduced to three columns. Next, the value of n is checked, if n is not equal to 1 then the algorithm proceeds through the conflict diagram again checking for those micro-orders which have not been incorporated in a previous set. Beginning with the three-maximal conflict micro-orders, Micro-order m_8 conflicts with m_7, m_2 and m_1 . Arbitrarily choose m_8 as a member of a new set. Since no other three-maximal conflict micro-order exists (which has not been assigned to some previous disjoint group), include any two maximal micro-order compatible with m_8 . Since m_9 is compatible with m_8 it can be included. Similarly m_4 and m_6 can also be included in this set. Again determine if all compatible micro-orders have been found at the current n -maximal conflict level. In this example m_{10} remains. However all other micro-orders have been tested for compatibility with m_{10} . Hence, m_{10} is assigned to a separate group and the search is now complete. Finally each compatible group is encoded.

Application of the conflict partition algorithm to the ROM in figure 23 generates the following set:

m_1, m_2, m_3, m_5, m_7

m_4, m_6, m_8, m_9

m_{10} .

Each of these fields also includes a NOP. For field 1A assign the coding 000 to the NOP, 001 to micro-order m_1 , 010 to micro-order m_2 etc. A three to eight decoder is used. Likewise for field 1B again a three to eight decoder is used. The final encoded ROM with the decoded output is shown in figure 26. The ROM has been reduced from 10-bit width to 7-bit width.

The conflict group approach represents another ad hoc method for ROM reduction, which is useful for short microprograms. It is possible to combine both function extraction and conflict grouping in the conflict-partition algorithm. Micro-orders m_3 and m_4 in the original ROM are identically activated for all microinstructions. m_3 and m_4 could have been reduced to one line before invoking the conflict grouping procedure.

2. A Compatibility Class Algorithm. In this algorithm, instead of pursuing the bit-reduction problem by isolating the conflict groups, an exhaustive search is done through the microprogram words for micro-orders that are compatible. Also the use of smallest number of decoders is obtained. This algorithm essentially partitions the micro-orders into a minimal number of groups. The procedure begins by finding the word, w_j , which has the largest number of micro-orders, m_j . The minimal number of groups cannot be less than m_j . In the worst case the maximal number of groups cannot exceed the the number of bits in the word. In w_j , assign a group designation, G_i to each micro-order. Proceed through the remaining words one at a time to

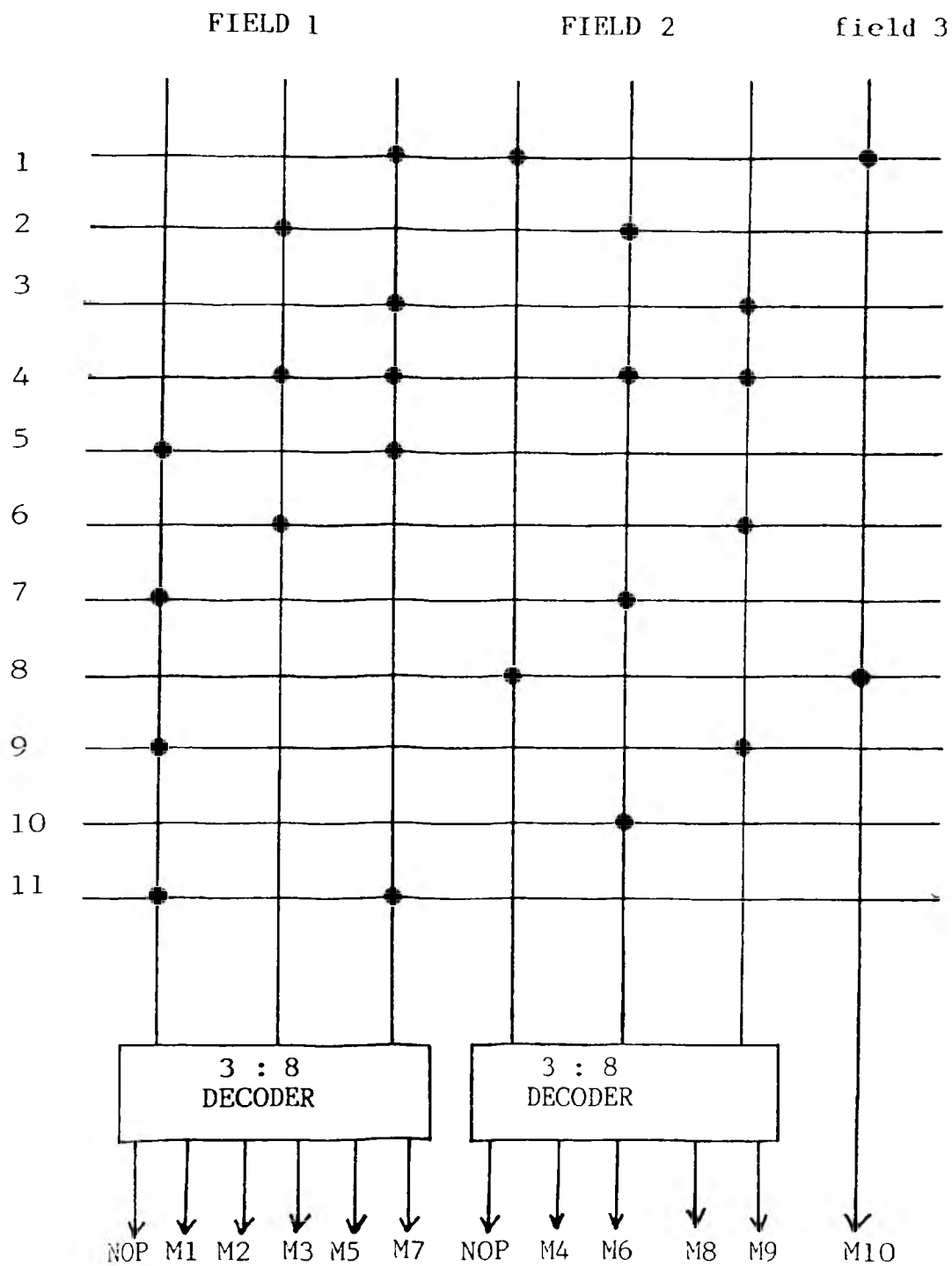


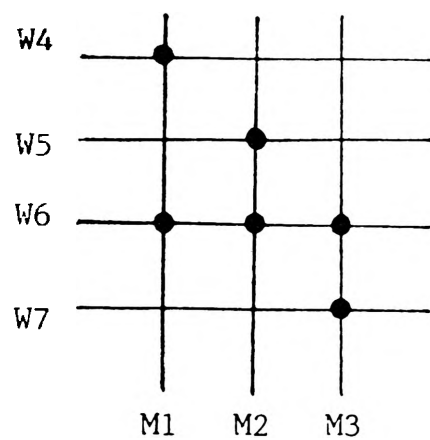
Figure 26. Encoded ROM

assign micro-orders to the previous groups if they are pairwise compatible or else introduce a new group for each incompatible micro-order. Micro-order assignment to the groups is made according to the following set of rules:

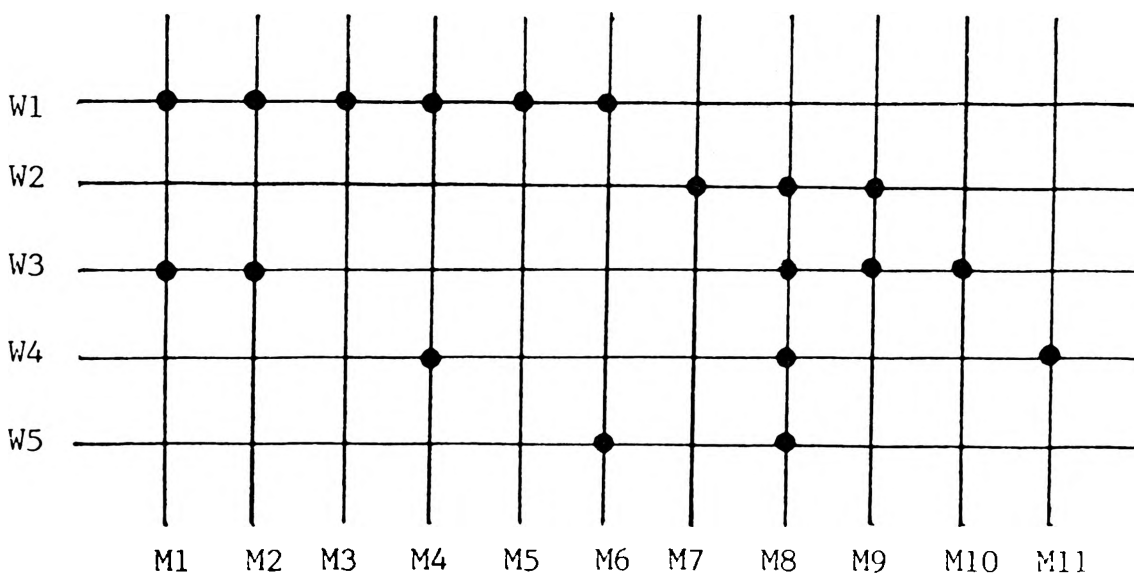
Rule 1. All micro-orders in the same word must be placed in different groups.

Rule 2. Previously assigned micro-orders found in the current word must not have been combined into the same group. Any new micro-order in the current word must be assigned to groups not in prior use in the word.

This algorithm must exhaustively evaluate all micro-order candidates for possible pairwise compatibility. If after examining the first six words, the compatibility property (rule1) is violated between two micro-orders in the seventh word (because of a previous group assignment of these two micro-orders), the algorithm then reassigns the micro-orders in the previously examined words so as to insure compatibility of the two micro-orders in the seventh word. This is shown in figure 27a. Suppose that m_1 and m_2 have been assigned to G_1 , but upon examination of w_6 , it is seen that m_1 and m_2 are incompatible (rules1 and rules2) hence m_1 and m_2 are reassigned to different groups (rule1). Assign m_2 to G_2 . Also m_3 in w_6 cannot be assigned to G_1 or G_2 (rules 1 and 2). It is these type of situations that require multiple iterations through the entire ROM. This, of course is a



(a)



(b)

Figure 27. ROM for example on Compatibility Class Algorithm

drawback for manual reduction but less so for automated means. It has been shown that it is possible to obtain the minimal number of groups. However this minimal number of groups does not guarantee that the minimal width of the ROM has been achieved.

The algorithm is applied to the ROM of figure 27b. Begin with the word the largest number of micro-orders. Since w_1 , has the maximum number of micro-orders, assign one micro-order to each of six groups as shown below:

	G_1	G_2	G_3			
step1	(m_1)	(m_2)	(m_3)	(m_4)	(m_5)	(m_6)
step2	(m_1)	(m_2, m_7)	(m_3, m_9)	(m_4, m_8)	(m_5)	(m_6)
step3	(m_1)	(m_2, m_7)	(m_3, m_9)	(m_4, m_8)	(m_5, m_{10})	(m_6)
step4		(m_3, m_9, m_{11})	(m_4)	(m_5, m_8)	(m_6, m_{10})	
step5	...	no change	...			

In w_2 , m_7 , m_8 , and m_9 , are activated. Hence rule1 requires that each of these micro-orders be placed in different groups. The micro-order m_3 is already assigned to G_3 and m_7 , m_8 , and m_9 are activated. Hence, rule1 requires that each of these micro-orders be placed in different groups. The micro-order m_3 is already assigned to G_3 and m_7 , m_8 and m_9 must be separated. Assign m_7 to G_2 since it is compatible with m_2 . Likewise, assign m_8 to G_4 and m_9 to G_3 . For w_3 , m_1 and m_2 have already been assigned but rule 2 prevents the assigning of m_8 , m_9 or m_{10} to any groups which contain m_1 and m_2 . m_8 and m_9 will be ignored since they are already assigned and m_{10} will be assigned to G_5 (m_{10} cannot be assigned to G_4 since rule2 prohibits this assignment). In w_4 , m_{11} cannot be assigned to G_4 (by rules 1 and 2).

Also in w_4 it is seen for the first time, that m_8 and m_4 are not compatible (by rule 1). Hence return to the step which made the m_8 assignment and reassign m_8 (as well as any other micro-orders if necessary). Reassign m_8 to G_5 . However, in w_3 m_{10} needs to be reassigned (Since m_8 , m_9 , and m_{10} must be separated). Assign m_{10} to G_6 . Now, upon return to w_4 , only m_{11} remains unassigned. Choose G_3 . For w_5 , all micro-orders have now been assigned. The groupings are now complete and are shown below.

G_1 G_2 G_3 G_4 G_5 G_6

(m_1) (m_2, m_7) (m_3, m_9, m_{11}) (m_4) (m_6, m_{10}) .

V. CONCLUSION

Efficient microprogram assemblers are of primary importance for developing firmware for microprogrammable machines. A small but firm step has been taken towards the design of system software for microprogramming. The development of a meta-assembler eliminates the need for developing separate assemblers for different host and target machine configurations, Hence a microprogram meta-assembler is an excellent tool in an environment for firmware development. The assembler is fast, efficient and has a reasonable syntax.

Although microprogram assemblers are widely used for microprogramming, there are some advantages of using high-level languages for microprogramming. Large and sophisticated microprograms are easier to write in a high-level language, also reading and understanding the programs is easier. A high-level language for microprogramming should produce efficient object microcode. The code should be optimized for taking advantage of the parallelism in the data-path of the machine. The complexity of high-level microprogram language compilers and the limited use of microprogramming have limited there development. The current microprogramming trends may change this situation. Detection of concurrently executable microoperations is an important consideration for effective horizontal microprogramming. The detection of concurrency is highly machine dependent and requires knowledge of highly intricate features of a machine. Techniques used for detection of concurrency and for generating highly parallel and efficient object microprograms in horizontal format are investigated.

Minimization of the dimensions of the control memory belongs to the general problem of optimizing microprograms. A single microprogram memory word may be as much as 100 bits long. At some point in the of a microprogrammable control unit, it may be desirable to minimize the storage dimensions of the ROM. Techniques used for optimization of control memory space are analyzed.

BIBLIOGRAPHY

1. Powers, Michael V. and Hernandez, Jose H. "Microprogram Assemblers for Bit-Slice Microprocessors" IEEE, Transactions on Computers, pp 108-120, July 1978.
2. Skordalakis E. "Meta-assemblers" IEEE MICRO pp 6-16 1983.
3. Ramamoorthy C.V. and Tsuchiya, Masahiro "A high-level language for Horizontal Microprogramming" IEEE, Transactions on Computers, Aug. 1974.
4. Andrews, Michael "Principles of Firmware Engineering in Microprogram control", Computer Science Press, Inc. Potomac, Maryland.
5. Tanenbaum Andrew S. "Structured Computer Organization", Prentice-Hall, Inc. Englewood Cliffs, New Jersey 07632.
6. Klier R.L. and Ramamoorthy C.V. "Optimization Strategies for Microprograms" IEEE , Transactions on Computers, July 1971.

VITA

Rahul Saxena was born on June 6, 1959 in Jodhpur, India. He received his primary and secondary education in New Delhi, India. He has received his college education from Visveswarya College of Engineering, Bangalore and the University of Missouri-Rolla, in Rolla, Missouri. He received a Bachelor of Science in Electrical Engineering from Visveswarya College of Engineering, Bangalore, India in July 1981.

He has been enrolled in the Graduate School of the University of Missouri-Rolla since August 1985. He held the position of Graduate Teaching Assistant in the Electrical Engineering department since August 1985 and was a Research Assistant in the Mechanical Engineering department in the fall semester 1986.